Heuristics for Unique Input Output Sequence Computation

by Hakan Kaynar

Submitted to the Graduate School of Sabancı University
in partial fulfillment of the requirements for the degree of
Master of Science

Sabanci University

June, 2008

Heuristics for Unique Input Output Sequence Computation

Hakan Kaynar

EECS, Master's Thesis, 2008

Thesis Supervisor: Hüsnü Yenigün

Keywords: Formal Testing Methods, Checking Sequences, UIO Sequences

Abstract

In this thesis, several heuristic methods are proposed for the computation of Unique Input Output (UIO) Sequences for the states of a given finite state machine. UIO computation problem is known to be a hard problem. The methods suggested in this work are based on unfolding an exponential tree as the other methods existing in the literature. However, our methods perform a search guided by some heuristic information. We also introduce a parameter for inference based UIO sequence computation for a trade off between the memory used for the computation and the UIO sequence length. Based on a randomly generated set of finite state machines, an extensive experimental study is also provided to compare the performance of our methods between each other and to those already exist in the literature.

Benzersiz Girdi Çıktı Dizilerinin Bulunması için Bazı Sezgisel Yöntemler

Hakan Kaynar
EECS, Yüksek Lisans Tezi, 2008
Tez Danışmanı: Hüsnü Yenigün

Anahtar Kelimeler: Biçimsel Sınama Yöntemleri, Kontrol Dizisi, Benzersiz Girdi Çıktı Dizileri

Özet

Bu tez çalışmasında, sonlu durum makinalarında Benzersiz Girdi Çıktı (BGÇ) Dizilerinin bulunması için bazı sezgisel yöntemler önerilmektedir. BGÇ dizilerinin hesaplanmasının zor bir problem olduğu bilinmektedir. Bu çalışmada önerilen yöntemler de, literatürde bulunan diğer yöntemler gibi üstel büyüklükte bir ağaç yapısına dayanmaktadır. Fakat, bu çalışmada önerilen yöntemler bu ağacı oluşturulması sırasında yapılan aramayı bazı sezgisel yöntemlerle yönlendirilmektedir. Bu yönlendirilmiş aramanın dışında, çıkarım kullanarak BGÇ dizisi bulan yöntemlere de değinilmiş ve bu yöntemlerin bir dezavantajı olan uzun diziler çıkarma sorununa bir çare olarak, sınırlı çıkarım yapma önerilmiştir. Rasgele üretilen sonlu durum makinaları kullanılarak, bu çalışmada önerilen yöntemlerin birbirleri ve literatürde bulunan diğer yöntemler ile karşılaştırması yapılmıştır.

Acknowledgments

I would like to express my gratitude to my supervisor, Hüsnü Yenigün. His quality of perspective, brilliant problem solving approaches and constructive comments have provided an important support throughout in this work. Only with his guidance, constructing a satisfactory thesis is possible for me. I would like to give my deep thanks to him for giving me the opportunity to work with him.

I would like to thank to Ersoy Bayramoğlu for giving brilliant ideas in the first year of my thesis study.

I would like to thank my friends Berk Çallı and Adil Küsen for feedbacking and providing a considerable perspective throughout my thesis, even though our area of interests are different.

I owe my loving thanks to my friend Başak Sönmez, my sister Zeynep Kaynar and my parents Kamuran Kaynar and Sema Kaynar. Without their support and encouragement, I may not find the strength to finish this thesis. I dedicate this thesis to them.

The financial support of Sabanci University is gratefully acknowledged.

Contents

List of Figures

List of Tables

1 Introduction

Nowadays, the computer systems are relatively large and complex, hence they are more error–prone than ever. Their reliability is very important due to their ubiquitous usage in everyday life. When their applications in safety critical domains are considered, the importance of their reliability is appreciated even more. Ensuring the reliability, or at least establishing a certain level of reliability of such systems is not easy. Several approaches have been proposed for increasing the reliability of these systems addressing the entire spectrum of their development cycle, starting from the checking of the consistency of the requirements, to the testing of the actual product. These methods can be classified as formal or informal. Formal methods use a mathematically supported framework to analyze the systems (for example model checking, automated theorem proving, etc.) whereas informal methods would lack such a mathematical infrastructure but would rather be practice oriented techniques such as software development process models or some good programming techniques.

Among these methods, testing is the only one that is related to the actual product. The other methods are all related to and operates on a *model* of the actual product. Although these methods are quite valuable and can increase the reliability considerably, by eliminating the errors introduced at the early stages of the development cycle, testing is unavoidable. It is unavoidable at least to catch those errors that can be introduced during the transformation of the model into the actual product. Even when this transformation is not performed by a human (which is the main source errors in these systems) but it is an automated process and testing might still be necessary. For example, one would probably want to test every single chip produced considering the possible production errors introduced during the manufacturing process.

In this work, we consider the testing of *reactive systems*. Unlike computational systems which accept an input, carry out a computation and present the result at the end of their execution, reactive systems consist of components that interact with each other and with their environment by some form of communication, and that will probably run forever. For example, a program taking the factorial of a number or solving a linear programming problem is a computational system. However, a program controlling the process at a nuclear reactor or controlling an airplane in auto–pilot mode is a reactive system. From now on we will refer to the reactive systems as systems.

Testing of a system is performed by an external tester which applies a sequence of inputs and verifies corresponding outputs. The input sequence applied and the expected out sequence is called a *test case*. Exhaustive testing, that is testing every possible behavior of the system, will require huge (if not infinite) amount of time and space since even simple systems will have quite a large number of different possible test cases. This makes exhaustive testing practically infeasible. In addition, the limited controllability and observability of the implementation under test (IUT) complicates the testing.

There are several approaches other than exhaustive testing. These methods aim to find test cases that will increase the reliability of the IUT without testing every possible behavior. Every test case successfully passing through the IUT would obviously increase the reliability of the IUT; however, the basic idea is to select a minimal set of test cases while maximizing the reliability they provide.

The testing methods are classified into different groups based on several factors. However, a general top–level classification is *white–box testing* and *black–box testing*. The methods that are classified as white–box testing are based on deriving the test cases by using the implementation details, such as the source code of a program. On the other hand, black–box testing methods do not assume any knowledge about the actual internals of the IUT. They instead use a *model* or a *specification* which describes the intended behavior of the IUT. The test cases are derived from such a specification. Therefore, black–box testing methods are also called as *model based testing* or *specification based testing*.

Finite State Machines (FSM) are widely used as the specification formalism in various areas including sequential circuits, software and communication protocols [1, 9, 3, 16, 22, 29, 40, 38, 19]. A *state* of the system is a representation of a stable condition at which the system is, until an action (e.g. an application of an input by the environment) occurs. This action causes the system to produce a *response* (e.g. an output signal sent to the environment) that can be observed. It also causes the system to move from current state to a new state, which is called *transition*.

The formal methods for generating test cases for checking the conformance of the IUTs to their FSM based specifications have been an interesting and active research area [34, 25, 27, 28, 33, 36, 17]. Lee and Yannakakis provide an excellent survey of the techniques in [17]. Some of these formal methods are based on *transition testing* only. These techniques embody the test sequences by considering the transitions in the specification [36, 14, 15, 21].

The application of these test cases to the IUT would just take the tester on a tour along the transitions of the IUT representing the transitions of its FSM specification. It is known that an IUT successfully passing such a test is not necessarily error free. There are more powerful techniques for test case generation from an FSM specification. *A checking experiment* [14, 15, 8, 4, 11] (where the test case is called a *checking sequence*) is one such approach. In a checking sequence, not only the transitions are traversed, but also the states of the FSM specification are tested one by one. Although a checking sequence is more powerful than a sequence testing only the transitions, it is also known that an IUT passing a checking experiment successfully is not necessarily a correct implementation. However, there are incorrect implementations that would be caught by checking experiments but not by the techniques testing the transitions only.

Whether a checking experiment or just a transition testing approach, both techniques rely on the notion of *state verification*. In other words, the test cases produced by these techniques would have parts in them to verify that the IUT is at particular states at particular steps during the application of the test case. Briefly explained, in order to understand the correct implementation of a transition in the IUT, the test case forces the IUT to execute the transition (to check if it will respond as expected) and then also the state that is reached after the execution of the transition is verified (to check if the transition leads to the expected state).

Three main techniques are proposed for state verification: *distinguishing sequence(DS)* [5, 6, 14], *characterizing set (CS)* [8, 14] and *unique input/output (UIO) sequence* [7, 11, 26]. The test sequence generation approaches which use the above mentioned techniques are D-method [8, 32, 10, 8, 14], W-method [2, 8, 14] and U-method [26, 1, 35, 40, 38], respectively. Even though these tree techniques do not show any significant difference that concerns fault coverage [29], the usage of UIO sequences has several advantages.

- For an FSM that has no distinguishing sequence, there may exist a UIO sequence for each state [1].
- A UIO sequence length is shorter than distinguishing sequence length.
- In practice, the test sequences that are generated using UIO sequences are shorter than those produced with characterizing set.

[1], [37] are two methods that use UIO sequences for state verification on the basis of transition testing and checking experiment problem respectively.

Since UIO sequences will be used inside the test cases many times (everytime a state needs to be verified), using short UIO sequences is desirable. Sabnani and Dahbura [26] proposed an algorithm to compute UIO sequences, which is based on the breadth–first expansion of a tree. Since the apporach is an exhaustive search of the tree in a breadth first manner, it finds the shortest possible UIO sequences. However, it takes exponential time since the tree explored grows exponentially. The bad news is that UIO sequences may not exist for an FSM (or for some states of the FSM) and even checking the existence of UIO sequences is known to be PSPACE–complete [16].

Naik [20] proposed a method that uses inference rules. That is, some UIO sequences are found using the approach given in [26] and some UIO sequences can be inferred from already known UIO sequences by using a set of rules. This decreases the execution time in practice; on the other hand, the length of the UIO sequences found increases considerably. In [24], UIO sequences are constructed using meta–heuristic optimization techniques such as simulated annealing and genetic algorithms. Due to the formulation of the search in this work, some UIO sequences may not be found even if they exist. Also, Ahmad *et al.* proposed a method based on a heuristic breadth–first search of the tree [12]. Their formulation is based on the binary encoding of the states, the inputs and the outputs of the FSM. It also proposes an inferencing approach which increases the length of the UIO sequences found.

The contributions of this work can be listed as follows:

- Several heuristic methods for the UIO sequence search problem are proposed. These methods are based on the exploration of the search tree like the previous methods. However, some heuristics are used to guide the search during the tree expansion.
- These heuristics are also combined with some of the techniques already suggested in the literature, especially the techniques given in [20] and [12], to improve those techniques further.
- A relatively extensive experimental study is provided based on randomly generated instances of FSMs.

The remainder of the thesis is structured as follows. In Chapter 2, an introductory background on FSMs is provided. The notation and the formalism that will be used in this thesis are introduced. A very simple but expensive way of finding UIO sequence concludes the chapter. Chapter 4 firstly introduces two techniques for UIO search to form a basis of comparison. It then explains the methods that are proposed by this work. For each method, some small scale experimental results are provided in order to materialize the performance. We explain how one of the disadvantages of the inferencing methods can be controlled in Chapter 5. We combine our heuristic search methods with that of [20] and also suggest a control mechanism to avoid finding long UIO sequences. Finally, in Chapter 6, we provide the results of our experimental study in detail. The concluding remarks are provided in Chapter 7.

2 Preliminaries

A finite state machine M is defined by $M = (S, I, O, \delta, \lambda)$, where S refers to the set of states $S = \{s_1, ..., s_n\}$, I denotes the finite set of input symbols $I = \{i_1, ..., i_p\}$, O denotes the finite set of output symbols $O = \{o_1, ..., o_q\}$. $\delta : S \times I \to S$ is the transition function and $\lambda : S \times I \to O$ is the output function. For simplicity, a finite state machine can be represented as a directed and labeled graph $G = (V, E)$. Each state $s \in S$ of FSM M is represented by a unique vertex $v \in V$ in G. Similarly, an edge $(v, i/o, v') \in E$ represents a transition of FSM M where $s, s' \in S$ and $\delta(s, i) = s'$ and $\lambda(s, i) = o$. The source and destination vertices v and v' of the edges are the source and the destination states s and s' of the corresponding transition, respectively. The label of the edge, i/o, represents the input and the output of the transition. For an edge $e = (v, i/o, v')$, we will use $head(e) = v$, $tail(e) = v'$, and $lbl(e) = i/o$ to denote the source vertex, the destination vertex and the label of the transition, respectively. In Figure 1, an example FSM can be observed.

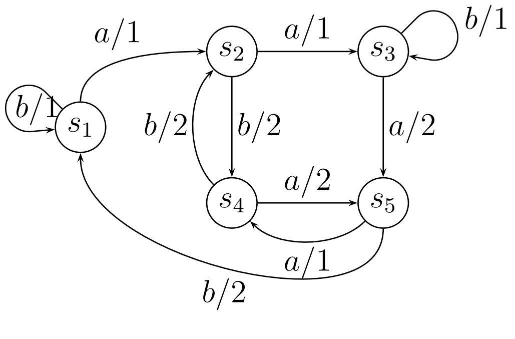


Figure 1: The FSM M_0

Since δ and λ are functions (rather than a relation), this definition of an FSM necessarily describes a *deterministic* machine. In other words, from a state s there is at most one transition with at most one output symbol defined. In this work, we only consider such deterministic machines.

Let $|.|$ denote both the length of the sequences and size of the sets. Then, $|S|$,$|I|$ and $|O|$ denote the number of states, the number of input symbols and the number of output symbols, respectively, whereas for a sequence of input symbols $X \in X^\star$, $|X|$ denotes the length of the sequence.

An I/O sequence is a pair of sequences X/Y such that $X \in I^*$, $Y \in O^*$ and $|X| = |Y|$. We extend the transition and output functions from a single input symbol to an I/O sequence as follows. $\delta(s, x_1x_2...x_k) = \delta(\delta(s, x_1), x_2...x_k)$ and $\lambda(s, x_1x_2...x_k) = \lambda(s, x_1)\lambda(\delta(s, x_1), x_2...x_k)$.

For a state $s \in S$, a *unique input output (UIO)* sequence is an I/O sequence X/Y such that $\forall s' \in S$, $s' \neq s$ implies $\lambda(s, X) \neq \lambda(s', X)$. In other words, there is no other state in FSM M which gives the output sequence Y to the input sequence X, except s. For instance, $aa/11$ is a UIO sequence for s_1 of the machine M_0 given in Figure 1. The response of all the states to the input sequence aa are given in Table 1. As can be seen from this table, the response of the state s_1 is unique among the responses of all the states.

A state may have more than one UIO sequence. It is easy to see that $aab/111$ and $aaa/112$ are also UIO sequences for the state s_1 of M_0 based on the fact that, for an I/O sequence X/Y, if a prefix of X/Y is a UIO sequence for a state s, then X/Y must be a UIO sequence for the state s as well. Let us define the length of a I/O sequence X/Y as $|X/Y| = |X| = |Y|$. The UIO sequences of a state might have different lengths. $aa/11$ and $aab/111$

Table 1: The responses of the states of $M0$ to the input sequence "*aa*"

State	*Input*	*Output*
s_1	aa	11
s_2	aa	12
s_3	aa	21
s_4	aa	21
s_5	aa	12

are UIO sequences for the state s_1 of length 2 and 3. A UIO sequence with minimum length is called a *shortest UIO sequence* for a state s. A closer look to Figure 1 will reveal that $aa/11$ is a shortest UIO sequence for the state s_1 since no I/O sequence of length 1 can be a UIO sequence for s_1. There are only two possible I/O sequences of length 1 from the state s_1, namely the sequence $a/1$ and the sequence $b/1$, and both of these sequences are also I/O sequences for some other states. Hence, they are not UIO sequences.

A state can have more than one shortest UIO sequence. For example, $ba/11$ is also a UIO sequence for the state s_1 of M_0.

It is also possible for a state not to have a UIO sequence at all, although there is no such state in M_0 in Figure 1 ($ab/11$, $ba/12$, $bab/211$ and $baa/211$ are UIO sequences for the states s_2, s_3, s_4 and s_5 respectively). In general, finding a shortest UIO sequence for a state would be desirable for those states with a UIO sequence, but unfortunately even checking the existence of a UIO sequence for a given state is PSPACE–complete[16].

2.1 UIO Computation

The discovery of UIO sequences for the states of a finite state machine M is performed by generating what we call the *UIO tree* of M. A UIO tree node is labeled by a set of *initial state–current state* pairs called *ICS* pairs. We will denote an ICS pair as $[s, s']$, where s and s' are states in the FSM M. For an ICS pair $[s, s']$, s is said to be the initial state and s' is said to be the current state.

An ICS pair $[s, s']$ is said to be *valid* for an I/O sequence X/Y iff $\delta(s, X) = s'$ and $\lambda(s, X) = Y$. Informally, if the FSM M starts from the initial state s and the input sequence X is applied, M will produce the output sequence Y, and the current state at the end will be s'.

For an ICS pair $[s, s']$, we use $s = init([s, s'])$ and $s' = curr([s, s'])$ to access the initial and the current states in the ICS pair. We extend these notations to the set of ICS pairs as follows: For a set of ICS pairs L, $init(L) = \{init(\rho) \mid \forall \rho \in L\}$ and $curr(L) = \{curr(\rho) \mid \forall \rho \in L\}$.

After these definitions, we are now ready to define the UIO tree.

Definition 1 *A* UIO tree *is a rooted tree and characterized by the following rules:*

1. *Each node is labeled by a set of ICS pairs. For a node TN, we will use $lbl(TN)$ to denote this label of the tree node.*

2. *Each edge is labeled by an I/O pair x/y where $x \in I$ and $y \in O$.*

3. *For each non–leaf node TN, for all $x \in I$ and $y \in O$, there is an outgoing edge from TN with the label x/y. Therefore a non–leaf node will have exactly $|I| \times |O|$ children.*

4. *A node TN is a leaf when $lbl(TN) = \emptyset$.*

5. *The root node has the label $\{[s,s] | \forall s \in S\}$.*

6. *For a node TN let us define $X/Y = path(TN)$ as the I/O sequence obtained by concatenating the I/O symbol pairs on the edges of the path from the root to TN. For all ICS pairs $[s,s']$, $[s,s'] \in lbl(TN)$ iff $[s,s']$ is valid for X/Y.*

We now explain several properties of UIO trees.

Remark 2

A node TN in the UIO tree such that $|lbl(TN)| = 1$ is an indication of a UIO sequence. Let $[s,s']$ be the only ICS pair at this node, and let $X/Y = path(TN)$. Due to (6) in Definition 1, any ICS pair for which X/Y is valid should be in $lbl(TN)$. Since there is only one such ICS pair, this means that among all the states only s can produce the output sequence Y to the input sequence X, and hence X/Y is a UIO sequence for the state s.

Remark 3

Let TN be a node in a UIO tree and x/y be an I/O pair (where $x \in I$ and $y \in O$). We denote the child of TN for the I/O pair x/y as $TN^{x/y}$. The label of $TN^{x/y}$ can be computed from the label of the TN as follows:

$$lbl(TN^{x/y}) = \{[s, \delta(s',x)] \mid \forall [s,s'] \in lbl(TN), \lambda(s',x) = y\}$$

For example (by using M_0 from Figure 1) if $lbl(TN) = \{[s_2, s_4], [s_4, s_2], [s_5, s_1]\}$ is the label of a node, then the label of the node $TN^{a/1}$ will be $lbl(TN^{a/1}) = \{[s_4, s_3], [s_5, s_2]\}$.

Based on (5) of Definition 1 and Remark 3, it is possible to develop an algorithm to generate the UIO tree of an FSM in a breadth–first manner. The algorithm will start from a tree with a single node, which is the root, and will generate all the children of all the nodes by visiting the generated nodes in a breadth first manner.

Remark 4

Let TN and TN' be UIO tree nodes such that TN' is a descendant of TN.

We have $init(lbl(TN')) \subseteq init(lbl(TN))$.

This is easy to see based on Remark 3. The initial states do not change from a parent to a child. They can only disappear in the child.

Remark 5

For a tree node TN, all of the initial states are unique. That is $|lbl(TN)| =$

$|init(lbl(TN))|$. However, the current states may not be unique. In other words, $|lbl(TN)| \geq |curr(lbl(TN))|$.

Note that by Remark 3, an initial state in an ICS pair in the label of a node is transferred into the label of a child node as is. Therefore, it is not possible for two labels in the child node to have the same initial state. However, this is not true for the current states. The current states change from a parent node to a child node, and it is possible for two different current states in the parent to be mapped on the same state in the child node. For example, (by using M_0 from Figure 1) if $lbl(TN) = \{[s_1, s_2], [s_2, s_3], [s_5, s_4]\}$ is the label of a node, then the label of the node $TN^{a/2}$ will be $lbl(TN^{a/2}) = \{[s_2, s_5], [s_5, s_5]\}$.

Definition 6 *Let TN be a node and $[s, s'] \in lbl(TN)$ be an ICS pair at TN. TN is said to be* homogeneous over the state s *iff there exists an ICS pair $[s'', s'] \in lbl(TN)$ such that $s \neq s''$. We will use $h(lbl(TN))$ to denote the set of initial states over which TN is homogeneous. TN is said to be* homogeneous *iff $h(lbl(TN)) = init(lbl(TN))$.*

To introduce the notation that will be used to depict UIO trees, a very small fragment of the UIO tree for the FSM M_0 of Figure 1 is given in Figure 2. Here, we have only the root of the UIT tree and two children of this root node, one for the I/O pair $a/1$ and one for the I/O pair $a/2$. We directly use the indices of the states to refer to the states (i.e. 1,2,3,4,5 are used rather than the names of the states s_1, s_2, s_3, s_4, s_5). The ICS pairs in the labels of the nodes are given vertically. That is, the ICS pairs of the node TN_1 are $\{[1, 2], [2, 3], [5, 4]\}$. One can see that the node TN_2 is a homogeneous node. A UIO tree node TN with $lbl(TN) = \emptyset$ will never be shown (actually such a node will never be generated).

As stated before, a very simple breadth–first generation of the UIO tree is possible. As the nodes are generated, it is possible to detect the UIO sequences found according to Remark 2. However, such an approach would generate an infinite tree since we did not specify any pruning conditions that can be used during the generation of the UIO tree. For pruning a UIO tree,

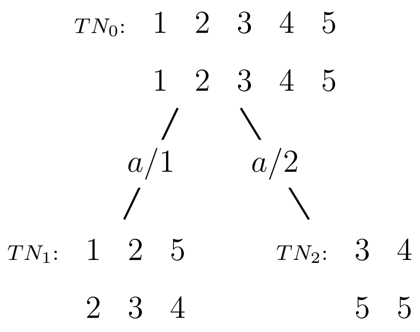


Figure 2: A Small Fragment of a UIO tree

there are some *termination conditions* common to all the methods that will be explained. These common conditions are given below. Method specific termination conditions will be introduced later within the sections of the corresponding methods.

- *Singleton Nodes:* Let TN be a node such that $|lbl(TN)| = 1$. According to Remark 2, TN will tell us a UIO. In fact if $lbl(TN) = \{[s, s']\}$, it will tell us a UIO of the state s. Consider any descendant TN' of TN. It is easy to see that either $lbl(TN') = \emptyset$ or $|lbl(TN)| = 1$ again. In the former case, it is already a leaf node as given in (4) of Definition 1. In the latter case, it is easy to show that $lbl(TN') = \{[s, s'']\}$ for some state s''. TN' will also tell us a UIO sequence but it will again be a UIO sequence for the same state s. The UIO sequence found by TN will be a prefix of the UIO sequence found by TN'. Therefore it is not necessary to continue the generation of the nodes from TN.

- *Homogeneous Nodes:* Let TN be a homogeneous node. In this case, for any ICS pair $[s, s'] \in lbl(TN)$, there will be another ICS pair $[s'', s'] \in lbl(TN)$ such that $s \neq s''$. Let TN' be a descendant of TN and let X/Y

be the I/O sequence labeling the path from TN to TN' and assume that $\lambda(s', X) = Y$. This means the ICS pair $[s, \delta(s', X)]$ will be a label of TN'. It also means that the ICS pair $[s'', \delta(s', X)]$ will be a label of TN'. This is based on the fact that whatever the current state of the ICS pair $[s, s']$ can do, the current state of the ICS pair $[s'', s']$ can also do, since they are the same state. Therefore, it will not be possible to separate these ICS pairs. So, a descendant TN' of TN with $lbl(TN') = 1$ will never exist. This means that it is not possible to find a UIO sequence by generating the descendants of TN, for this reason, the tree generation can be pruned at such a node.

- *Repetitive Nodes:* Let TN and TN' be two nodes such that $lbl(TN) = lbl(TN')$. The subtree rooted at TN will then be exactly the same as the subtree rooted at TN'. Therefore, it is sufficient to expand the UIO tree at one of these nodes only, and prune the generation at the other one.

Figure 3 displays a more complete form of the UIO tree for the FSM M_0 of Figure 1. It is still not complete, however, there are examples for the termination conditions explained above. The generation of the tree is pruned at $a/2$ successor of the root and $aa/12$ successor of the root because these nodes are homogeneous nodes. The tree is also pruned at the $bb/11$ successor of the node because this is a repetitive node, it is the same as the $b/1$ successor of the root. Note that in general the repetitive nodes may be at entirely different parts of the UIO tree. They just happened to be parent–child in this example by chance. The nodes marked by a an asterisk are the nodes where UIO sequences are found. So, the generation is also pruned at these nodes.

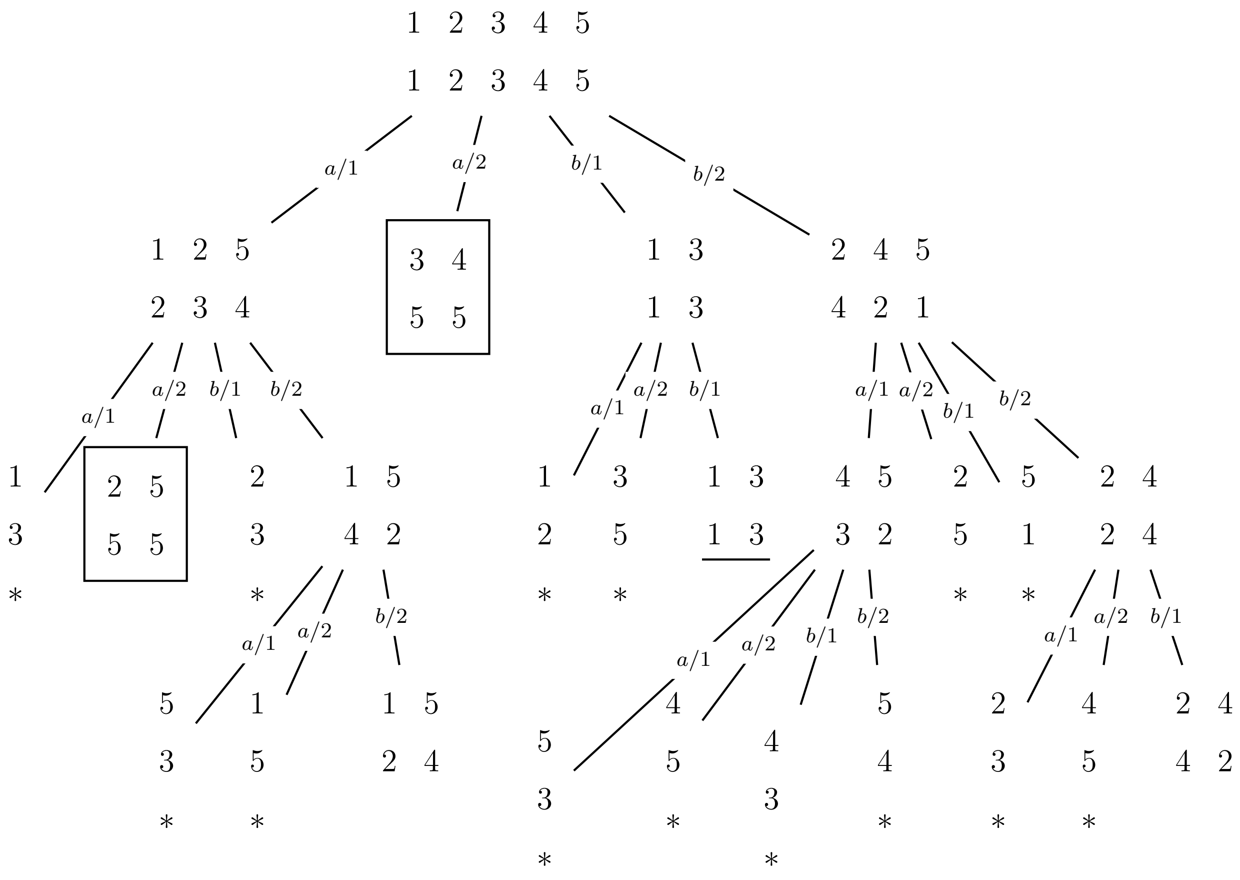


Figure 3: A More Complete UIO Tree for M_0 of Figure 1

2.2 Exhaustive UIO Computation

One method for generating UIO sequences relies on generating the UIO tree in a breadth–first manner by observing the termination conditions given on Page 27. One should also keep track of the set of states for which a UIO sequence is found, so that the algorithm can be terminated after finding at least one UIO for each state. Such an algorithm is depicted as Algorithm 1. When the variable named E in this algorithm is implemented as a queue, it generates the UIO tree in a breadth–first manner.

Algorithm 1: A UIO Sequence Computation Algorithm

```
1  $E = \emptyset$ ;                    // UIO tree nodes yet to be explored
2  $R = \emptyset$ ;    // states for which UIO sequences have been found
3  create the root of the UIO tree and insert it into $E$;
4  while (($E \neq \emptyset$) $\wedge$ ($R \neq S$)) do
       // S here is the set of all states
5      $TN$ = get and remove the next node in $E$;
6      forall the $x \in I, y \in O$ do
7          if (|$lbl(TN^{x/y})$| == 1) then
               // recall the notation $TN^{x/y}$ from Remark 3
8              Let $[s, s']$ be the ICS pair in $TN^{x/y}$;
9              $R = R \cup \{s\}$ ;
10         else if (($lbl(TN^{x/y}) > 1$) $\wedge$ ($TN^{x/y}$ is not repetitive) $\wedge$ ($TN^{x/y}$
           is not homogeneous)) then
11             $E = E \cup \{TN^{x/y}\}$ ;
```

Note that Algorithm 1 does not keep track of the actual UIO sequences found for the states. However, adding such a feature is trivial by inserting a line in the "then" part of the "if" statement to write down that $path(TN^{x/y})$ is a UIO sequence for the state s. Therefore, we omit this feature in Algorithm 1 and in all the other algorithms in this thesis.

3 Literature Review

Naik's Method

Naik [20] proposed a technique for finding UIO sequences efficiently. In his work, the introduced method computes UIO sequences with dramatical decrease in memory requirements. However, the found UIO sequences are very long when compared to the exhaustive UIO computation. The decrease in memory requirements and the increase in UIO sequence lengths were the results of *inference* mechanism. Informally, inference is an approach which infers new UIO sequences from existing UIO sequences. In order to infer a UIO for a state, that state should be the head state of a *unique transition* to a tail state for which a UIO is found by populating the UIO tree. Formally, a state s_i is *unique predecessor* of state s_j, if the label of the edge (s_i, s_j) is unique among all the incoming edges to s_j. So, the transition represented by an edge in the graph is a unique transition. For example, the unique transitions of M_0 can be observed in Figure 4.

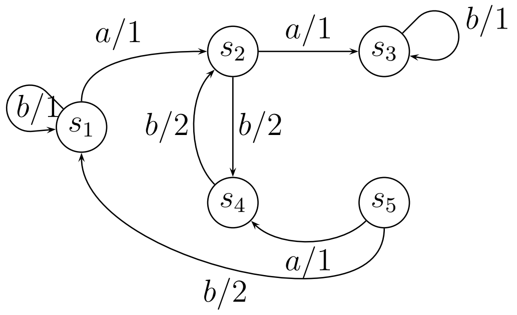


Figure 4: The Unique Transitions of M_0

If we know the UIO of any state, we may produce new UIO sequences for other states by prefixing the labels of unique transitions to the existing UIO sequence. An *inference rule* is obtained as follows.

$$UIO_j = lbl(e) + UIO_i \text{ where } e \text{ is a unique transition such that}$$
$$tail(e) = v_i, head(e) = v_j.$$

So, it is possible to infer new UIO sequences from the UIO sequences which is found by populating the UIO tree. In [20], the UIO generation is held in a *hybrid* manner. That is, a tree node is expanded by applying all input symbols and the next tree node that will be explored is selected randomly among the children of that tree node. So, the generation is held in depth–first manner. However, if a subtree that is rooted by a child node does not result in a UIO sequence, the generation algorithm will pass to next random children. So, every children of a node is examined in a breadth–first manner, but the subtrees are created in a depth–first manner.

The hybrid generation of UIO tree is not the only way to find UIO sequences that will be used in inference. Naik [20] proposed *projections* and *linear path* techniques for UIO sequence extraction for a state without constructing a UIO tree. Formally, a *projection* $G_{x/y} = (V', E')$ of a graph $G = (V, E)$ and an I/O pair x/y is a subgraph G such that:

$$V' = \{head(e), tail(e) | lbl(e) = x/y\}$$

$$E' = \{e | lbl(e) = x/y\}$$

The $a/1$ projection of M_0 can be observed in Figure 5. A path, denoted as P_v, in the projection is a sequence of edges which may end in a sink state or in a state which is already seen in the path. If a path ends in a sink state, it is a *linear* path and we may extract UIO sequences for some of the states in a linear path. Based on the structure of the paths, [20] suggests some ways to find UIO sequences without even constructing a UIO tree.

So, Naik [20] first finds UIO sequences using the paths in the projections and infers UIO sequences using these UIO sequences. If there exists a state for which a UIO sequence has not been found yet, then it generates UIO tree with the hybrid method described above and finds UIO sequences for further inferences. As a result, the inference and linear path mechanisms could find UIO sequences for all the states. However, due to the sequential prefixing of unique transition labels to existing UIO sequences, resulting UIO sequences have longer lengths when compared to the UIO sequences that would be found by the exhaustive method.

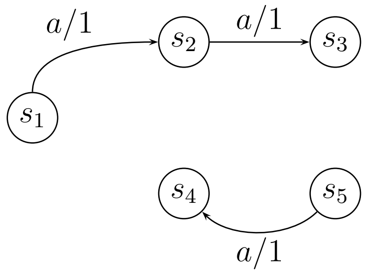


Figure 5: The $a/1$ Projection of FSM M_0

Genetic Algorithm

In [13] and [23], UIO computation problem is attacked by using a Genetic Algorithm (GA) approach. In these two works, the individuals are I/O

Table 2: Frequencies and ranks of I/O pairs in M_0

I/O pair	Frequency	Rank
$a/1$	3	1
$a/2$	2	0
$b/1$	2	0
$b/2$	3	1

sequences. The parents are selected with respect to the fitness functions for obtaining the next generation. The children are created with cross–over and single point mutations are held in order to preserve randomness in the population. When the termination conditions specified are satisfied, the generation algorithm terminates and generated I/O sequences are checked if those sequences are UIO sequences.

In [13], the proposed fitness function is in terms of the frequency of the transition labels in an FSM. The transition table of the FSM is examined before the pool generation and every transition is ranked with respect to the occurrence count of transition label in the FSM. The least frequent I/O label gets the lowest IO rank and the highest frequency I/O label gets the highest IO rank. In Table 2, the transition ranks of M_0 can be seen.

The quality of an I/O sequence is sum of the ranks of I/O labels which forms the sequence. For example, the sequence $ab/12$ has the fitness point of 2. That is, the rank of $a/1$ is 1 and the rank of $b/2$ is 1. The idea in this work is low frequency I/O sequences are more likely to be UIO sequences. For this reason, the fitness point gives high points to the sequences that has low transition rank sum.

In [23], the fitness function is build upon the analysis of the *splitting tree* of the FSM. A state splitting tree is a construct used to extract adaptive distinguishing sequences and UIOs from an FSM. Each node in the tree has a parent and children. The root node is composed of all set of states and has a null parent. With an input application, the children are grouped with respect to the outputs they produced. If all the leaf nodes are discrete, the splitting tree is complete and ready for adaptive DS and UIO extraction. A path from discrete partition node to tree root is a UIO sequence discovery.

In this work, the fitness of an I/O sequence is bound to the number of the discrete partitions and separated groups that it results in the state splitting tree. That is, for an I/O sequence, the state splitting tree is built with the guidance of the I/O pairs of that sequence. The quality score of an I/O pair that constructs the sequence is:

$$f(i) = \alpha \frac{x_i e^{x_i + \delta x_i}}{l_i^{\gamma}} + \beta \frac{y_i + \delta y_i}{l_i}$$

where i is the i^{th} I/O pair of the corresponding sequence, x_i denotes the number of existing discrete partitions, δx_i is the number of new discrete partitions, y_i is the number of existing seperated groups and δy_i is the number of new seperated groups. α, β and γ are constants. Thus, the fitness of an I/O sequence is:

$$F = \frac{1}{N} \sum_{i=1}^{N} f(i)$$

where N is the sequence's length.

So, the genetic algorithms that are proposed in [13, 23], initially generates a pool of I/O sequences. The algorithms pick successfull parents with the quality measures that is described here. Then, the children are created with cross–over and mutation and the algorithm passes to the parent selection for the next generation.

LANG Algorithm

In [12], a heuristic method has been proposed for UIO sequence computation. In this work, the FSMs which have binary input and output symbols have been considered and a UIO tree is constructed in order to search UIO sequences. For guiding the search, every UIO tree node is labeled as *active, inactive* and *dead* node. A tree node is said to be *active* if there exists an initial state of the node which is not homogeneous over the node and its UIO has not been found yet. A tree node is *inactive* if the algorithm finds UIO sequences for active initial states of that tree node via using other subtrees. A *dead* node is defined as a repetitive node or a tree node which has current states equal to the corresponding initial states.

In the algorithm, only the active nodes are used for children generation and the number of generated nodes in the tree is limited. If the node limit is reached and there exists states for which a UIO has not been found, they propose the *chain node* technique for finding UIO sequences from existing tree nodes. Formally, a node $TN_i = \{ICS_{i1}, ICS_{i2}, ...\}$ is a chain of another node $TN_j = \{ICS_{j1}, ICS_{j2}, ...\}$ if $|curr(ICS_i) \cap init(ICS_j)| = 1$. That is, only one current state in TN_i can be observed in TN_j as an initial state. Figure 6 demonstrates a chain node example. It can be seen that TN_0 is a chain node of TN_1 because only the current state s_4 can be seen in TN_1 as an initial state. In order to find a UIO sequence from the subtree rooted by TN_0, the algorithm has to separate s_4, s_8 and s_6 from each other. It can be observed that TN_1 has already done this separation with the sequence accumulated from tree root to TN_1. For this reason, if we apply same sequence to the TN_0, it is guaranteed to separate above mentioned states from each other and find UIO sequence of s_1.

TN_0:	1	2	3	TN_1:	4	7	3
	4	8	6		2	7	4

Figure 6: A Chain Node Example

So, Ahmad et.al. [12] proposed a breadth–first heuristic approach for FSM with binary input, output symbols. They have a dead, inactive and active node approach in order to guide UIO search. This work limits the population of the tree to some value and after this limit is reached, they find UIO sequences using *chain node* approach.

Sun et. al. Method

Another heuristic method for exploring UIO sequences is proposed by Sun et. al [30]. In their work, they have considered FSMs with binary I/O symbols and demonstrating a breadth–first heuristic method for finding UIO sequences. As a difference from other methods that is described in this section, this method does not simultaneously search UIO sequences for all state at a time. For every state $s_i \in S$, the algorithm constructs UIO tree and searches the UIO sequence of s_i.

In this work, every transition is accompanied by a *Distinguishing State Group (DSG).* That is, the set of states which are distinguished from the source state of the transition with the output response of the transition and the search for UIO seqence of a state s_i is held with the guidance of DSGs. The tree is rooted by s_i and there exists a set of states which should be distinguised from s_i and called as TBD. The first level of the tree are all the transitions headed by s_i and resulting TBD values for each transition are updated. In the next level, the input symbols that will be applied to the tree nodes are selected using greedy heuristic. That is, the transition that is invoked by the input symbol will maximize $TBD_{TN} \cap DSG_{TN}$.

4 UIO Computation Methods

In this chapter, we will first introduce two known methods to form a basis for the performance comparison for the methods that we will suggest. This will be followed by the explanations of our methods. Each section introduces a new approach and the order of the sections reflect a chronological and a logical order of the methods developed throughout the course of this work. The (memory and time) performance of the methods will be improved by each section in general. However, the last section introduces an unsuccessful attempt to improve the methods.

Throughout this chapter, we examine the performance of the methods on a fixed set of FSMs that we generated randomly. This set is composed of seven FSM groups where each group has the same number of states. The state sizes of the groups are 100, 500, 1000,1500, 2000, 2500 and 3000 and every FSM has four inputs and four outputs. There are 50 FSMs in each group. So, in total there are 350 FSMSs with a total of 530000 states.

We provide a more general experimental study later in Chapter 6.

4.1 Exhaustive UIO Computation

As one may have noticed, Algorithm 1 is probably not the best algorithm for finding UIO sequences. In fact, we will suggest several improvements on this algorithm. However, there are some immediate and obvious improvement opportunities.

For example, assume that the algorithm has been working for a while and it has accumulated UIO sequences for a set of states R. Also assume that there is a UIO tree node TN yet to be explored in E. If $init(lbl(TN)) \subseteq R$, we do not actually need to generate the subtree rooted at TN. The reason is the following: Let TN' be a descendant of TN such that $|lbl(TN')| = 1$. So, TN' tells us a UIO. It will tell a UIO for a state $s \in init(lbl(TN')) \subseteq init(lbl(TN)) \subseteq R$ (where the first $\subseteq$ follows from Remark 4). In other words, it will tell us a UIO for a state s for which a UIO found before. Therefore, it is not necessary to generate TN' and hence, it is not necessary to generate the subtree rooted at TN.

Another obvious improvement is the following. In [18], the *converging transitions* are defined in order to find the states for which a UIO sequence does not exist. Formally, a transition $\delta(s, x), s \in S, x \in I$ is converging if $\exists s' \in S$ such that $s \neq s', \delta(s, x) = \delta(s', x)$ and $\lambda(s, x) = \lambda(s', x)$. So, both s and s' produce the same output to x and they both end up in the same state. This means that a UIO for s or for s' cannot start with x. If all transitions of a state are converging, then it means there does not exist a UIO sequence for that state since it wouldn't be possible to start the UIO for that state with any input symbol. Note that this is only a sufficient condition for not having a UIO for a state.

Let us define $S' \subseteq S$ as the set of states for which there exists at least one non–converging transition. Suppose that Algorithm 1 has been working for a while and let R denote the set of states for which a UIO sequence has been found, TN be a node yet to be explore, and $h(lbl(TN))$ denote the set of initial states over which the tree node is homogeneous (see Definition 6). The subtree rooted at TN is only good for finding UIO sequences for the states in $init(lbl(TN)) \setminus (R \cup h(lbl(TN)) \cup (S \setminus S'))$. Firstly, it can only find UIO sequences of the states in $init(lbl(TN))$. Among these states, the algorithm has already found at least one UIO sequence for those states in $init(lbl(TN)) \cap R$. Second, it is not possible for TN to have a descendant for finding a UIO for a state in $h(lbl(TN))$. Furthermore, it is not possible to find any UIO sequence for the states in $S \setminus S'$.

Let us define the potential states of a node TN as

$$\Phi(TN) = init(lbl(TN)) \setminus (R \cup h(lbl(TN)) \cup (S \setminus S'))$$

since TN has a potential only for these states. If $|\Phi(TN)| \geq 1$, only then it makes sense to generate the subtree rooted at TN. We will update Algorithm 1 to reflect this consideration into the algorithm.

Another point we want to highlight is the following. Note that Algorithm 1 checks if a newly generated node TN is repetitive or not. This check is performed by a search on the entire tree. There are some tricks that one can play to speed up the search but in general it takes a huge amount of time to perform this check. The check actually is used to decrease the memory requirements of the algorithm (by not generating multiple copies of the same subtree) and thus, to decrease the time requirements. However, our experiments showed that removing the check speeds up the execution of the algorithm and but does not increase the memory requirement noticeably. Table 3 shows the difference of two exhaustive method versions where the first one employs the repetitive check by keeping UIO tree and the second version does not keep track of the repetitive nodes. The average time to analyze an FSM is extremely high with repetitiveness check. However, the average UIO sequence length and the average tree size do not even change when we remove the repetitiveness check. For this reason, in the rest of this thesis the repetitive check is not considered in the implementation of the methods that will be introduced.

Table 3: The comparison of the exhaustive method with and without repetitive check

Number of States	Check	Avg. UIO Length	Tree Size	Time
100	with	2.95	1138	179
	without	2.95	1138	29
500	with	3.94	10341	30980
	without	3.94	10341	221
1000	with	4.01	41697	640808
	without	4.01	41697	3142
1500	with	4.32	76627	2798436
	without	4.32	76627	8309

Note that, when the repetitiveness check is removed, there is actually no need to keep the tree in the memory anymore. Keeping the list of current leaves is sufficient for the purposes of the algorithms.

The updated algorithm can be seen in Algorithm 2 and experimental results can be seen in Figure 7.

We call the method described by Algorithm 2 as the *exhaustive method.*

Note also that in Algorithm 2, the potential of a node is checked when it is first created on line 2. When a node is picked as the node to be explored, its potential is checked again (line 2). The reason for the second check is that a node's potential may change (actually it can only get smaller) from

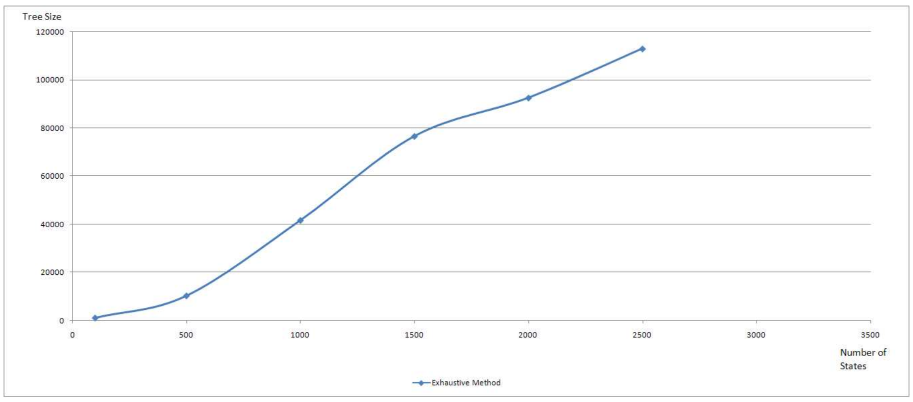


Figure 7: Memory Performances of the Exhaustive and the Random Methods

Algorithm 2: Exhaustive Method

```
1  $E = \emptyset$ ;                    // UIO tree nodes yet to be explored
2  $R = \emptyset$ ;    // states for which UIO sequences have been found
3  create the root of the UIO tree and insert it into $E$;
4  while $((E \neq \emptyset) \wedge (R \neq S'))$ do
5      $TN$ = get and remove the next node in $E$;
6      if $(|\Phi(TN)| \geq 1)$ then
7          forall the $x \in I, y \in O$ do
8              if $(|lbl(TN^{xy})| == 1)$ then
9                  Let $[s, s']$ be the ICS pair in $TN^{x/y}$;
10                 $R = R \cup \{s\}$;
11             else if $(|\Phi(TN^{x/y})| \geq 1)$ then
12                 $E = E \cup \{TN^{x/y}\}$ ;
```

the time it is created to the time it is picked, if in between these two time instances, the algorithm discovers some UIO sequences for those states that were in the potential of the node initially.

4.2 Random UIO Computation

The random UIO computation method is introduced in order to compare the exhaustive method and the heuristics that will be described in the next sections. Rather than exploring the nodes in a certain order, the random UIO computation generates the UIO tree by selecting the next node to be explored randomly among all the leaves of the partial UIO tree at that moment. In Figure 8, a UIO tree that is generated by the random method is illustrated. After the root node is expanded with all of the input/output pairs, we get the leaf set $\{TN_1, TN_2, TN_3, TN_4\}$. Among these nodes, TN_3 is selected randomly and is expanded. Expanding TN_3 finds the UIO sequences for s_1 and s_3 and a repeated node TN_7. In the next iteration, the set of nodes yet to be explored becomes $\{TN_1, TN_2, TN_4\}$, and TN_4 is selected randomly. When TN_4 is expanded, UIO sequences for s_2 and s_5 are found. Also, the nodes TN_8 and TN_{11} are added to the set of nodes yet to be explored, making this set $\{TN_1, TN_2, TN_8, TN_{11}\}$. Finally, the node TN_8 is picked randomly to be explored. When this node is expanded, UIO sequences for s_4 and s_5 (two UIO sequences for each one of them actually) are found. This should complete the execution of the algorithm since we now have at least one UIO sequence for every state.

The algorithm for the random method is described in Algorithm 3.

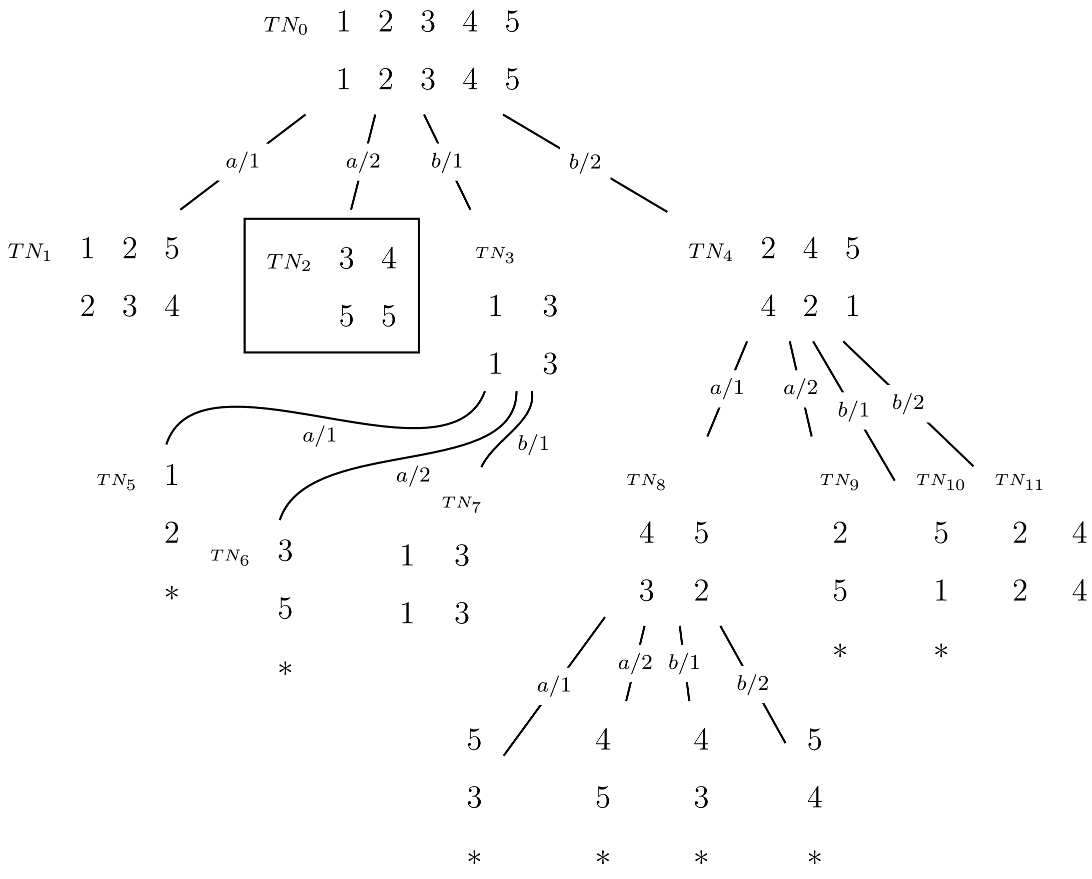


Figure 8: An Example UIO Tree as Generated by the Random Method

Algorithm 3: Random Method

```
1  $E = \emptyset$ ;                    // UIO tree nodes yet to be explored
2  $R = \emptyset$ ;    // states for which UIO sequences have been found
3  create the root of the UIO tree and insert it into $E$;
4  while $((E \neq \emptyset) \wedge (R \neq S'))$ do
5      $TN$ = pick a node from $E$ randomly and remove it from $E$ ;
6      if $(|\Phi(TN)| \geq 1)$ then
7          forall the $x \in I, y \in O$ do
8              if $(|lbl(TN^{x/y})| == 1)$ then
                   // found a UIO sequence
9                  Let $[s, s']$ be the ICS pair in $TN^{x/y}$;
10                 $R = R \cup \{s\}$;
11             else if $(|\Phi(TN^{x/y})| \geq 1)$ then
12                 $E = E \cup \{TN^{x/y}\}$ ;
```

We also give some experimental comparisons between the random and the exhaustive methods in figures 9, 10, and 11. First of all, interestingly, the random method uses less memory than the exhaustive method. One could expect that, during random search, the tree can be expanded in those parts that are of no use for finding UIO sequences. Hence, it may need much more memory then the exhaustive method. However, the fact is that we never consider a UIO tree node without a potential for expansion even in the random method. This behaviour turns the random method a bit to a guided search.

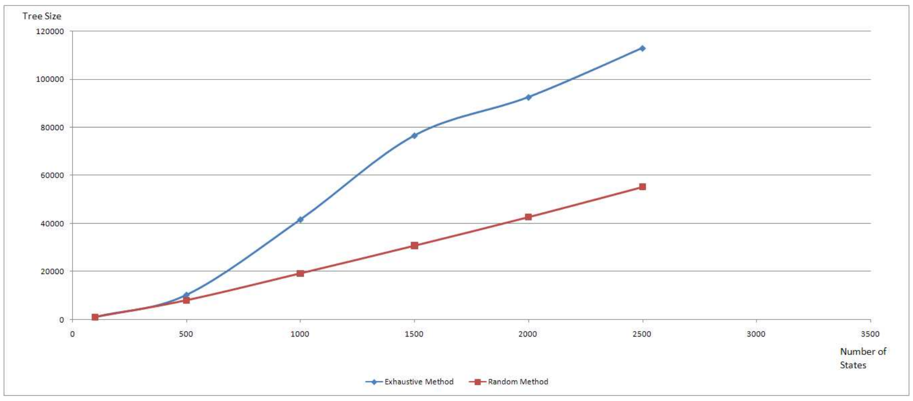


Figure 9: Tree Size Performances of the Exhaustive and the Random Methods

When we compare the performances of these two methods in terms of the length of the UIO sequences they find, we see that the exhaustive method is better than the random method. In fact this is quite expected, since the exhaustive method explores the UIO tree in a breadth–first manner and it is guaranteed to find the shortest UIO sequences.

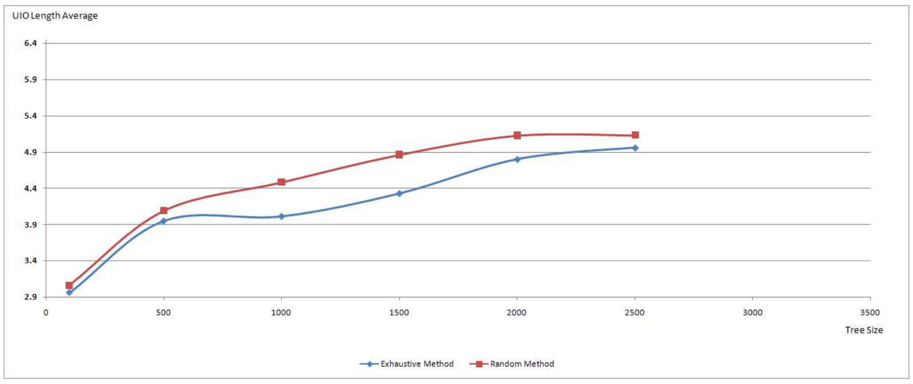


Figure 10: UIO Sequence Length Performances of the Exhaustive and the Random Methods

The time performance of the random method is also better than that of the exhaustive method, as expected based on the comparison of the performances in the tree size.

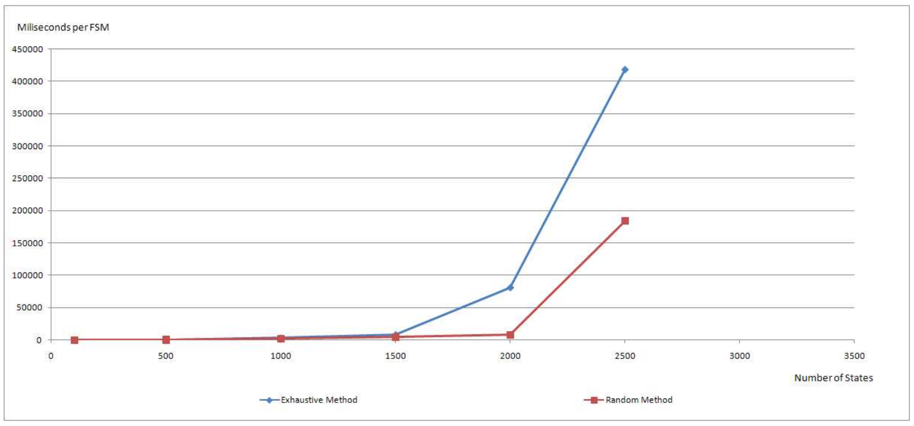


Figure 11: Time Performances of the Exhaustive and the Random Methods

4.3 Heuristic Method

At any given time during the execution of Algorithm 2 and Algorithm 3, any node in E can be picked to be explored in the current iteration. The first heuristic method that we will introduce will try to predict the *quality* of the subtree at a node without generating the subtree. One measure for quality can be considered as the number of states for which UIO sequences will be found, as this is the ultimate aim of the algorithms. The more the UIO sequences are found by generating a subtree, the more justified is the generation of that subtree.

We already have a measure for the states for which a UIO tree node TN has a hope for finding a UIO sequence, the potential $\Phi(TN)$. Therefore, the nodes with larger potentials will probably generate UIO sequences for more states.

However, it is also important how small or large the subtree will be, since the UIO sequences will only be found at the leaves of the subtree. For predicting the size of the subtree, the number of current states in the label of the node at the root of that subtree seems to be one measure. Let us give an example on this observation: Suppose we have two UIO tree nodes TN and TN' with the labels $\{[s_1, s_{11}], [s_2, s_{12}], [s_3, s_{12}], [s_4, s_{12}], [s_5, s_{12}]\}$ and $\{[s_1, s_{11}], [s_2, s_{12}], [s_3, s_{12}], [s_4, s_{14}], [s_5, s_{14}]\}$, respectively. Assume that $s_1 \in \Phi(TN)$ and $s_1 \in \Phi(TN')$. Note that s_2, s_3, s_4 and s_5 can be a potential state neither in TN nor in TN' since both TN and TN' are homogeneous over these states.

In order to find a UIO for state s_1, the subtree rooted at TN must have a path that separates s_{11} (the current state corresponding to the initial state s_1) from the other current states of the ICS pairs at TN. However, there is only one such other state, which is s_{12}. Within the subtree rooted at TN', in order to find a UIO sequence for s_1, the state s_{11} will have to be separated from the states s_{12} and s_{14}, which will probably be harder. Hence, the path will probably be longer and the subtree will probably be larger.

Therefore, as the first approximation for predicting the quality of a subtree rooted as a node TN, one can suggest the measure:

$$\frac{|\Phi(TN)|}{|curr(lbl(TN)))|}$$

By using this measure, two nodes having the same number of potential states and the same number of distinct current states proportionally will have the same heuristic value. However, if a node has a smaller current set, that means its subtree will be smaller. So, one may want to generate the subtree of such a node first. The idea is explained by using the following example:

$$TN_1\!: \begin{array}{cccc} 1 & 2 & 4 & 5 \\ 4 & 3 & 5 & 6 \end{array} \qquad TN_2\!: \begin{array}{cc} 4 & 5 \\ 5 & 6 \end{array} \qquad TN_3\!: \begin{array}{ccc} 4 & 5 & 3 \\ 1 & 2 & 2 \end{array}$$

Figure 12: Tree Node Examples for Heuristic Method

Suppose we have three UIO tree nodes as candidates to be explored TN_1, TN_2 and TN_3 as given in Figure 12, $S' = S$ and currently $R = \emptyset$. These nodes will have the following heuristic scores:

$$\frac{|\Phi(TN_1)|}{|curr(lbl(TN_1))|} = \frac{4}{4} = 1$$

$$\frac{|\Phi(TN_2)|}{|curr(lbl(TN_2))|} = \frac{2}{2} = 1$$

$$\frac{|\Phi(TN_3)|}{|curr(lbl(TN_3))|} = \frac{1}{2} = 0.5$$

Based on these heuristic scores, either TN_1 or TN_2 could be picked as the next node to be explored. However, it can be seen from the labels of these nodes that TN_2 promises a less complex subtree and possibly shorter UIO sequences for s_4 and s_5. This is because it only needs to separate the states s_5 and s_6. Therefore, rather than having a direct proportion, it might be a better idea to emphasize the size of the current set of a node in the heuristic point of that node. So, we define the heuristic point of a node as follows:

$$HP(TN) = \frac{|\Phi(TN)|}{|curr(lbl(TN))|^2}$$

With this definition, the heuristic points of the tree nodes given in Figure 12 will be as follows:

$$HP(TN_1) = \frac{|\Phi(TN_1)|}{|curr(lbl(TN_1))|^2} = \frac{4}{16} = 0.25$$

$$HP(TN_2) = \frac{|\Phi(TN_2)|}{|curr(lbl(TN_2))|^2} = \frac{2}{4} = 0.5$$

$$HP(TN_3) = \frac{|\Phi(TN_3)|}{|curr(lbl(TN_3))|^2} = \frac{1}{4} = 0.25$$

By using this idea, one can modify the random or the exhaustive method to consider a node with a maximum heuristic point in each iteration. Such an algorithm is given in Algorithm 4. This method will be called as the heuristic method.

When the tree sizes that are explored by the three methods introduced so far compared (Figure 13), the heuristic method is much better than both of the previous methods, as expected. Also note that, for the exhaustive and the random methods, only the results upto the FSM set with 2500 states are available. The tests for the FSM set with 3000 states could not even be completed with these methods due to memory limitations. However, the heuristic method could complete the tests for the FSM set with 3000 states.

It can actually go beyond 3000 states as explained in Chapter 6.

Algorithm 4: Heuristic Method

```
1  $E = \emptyset$ ;                    // UIO tree nodes yet to be explored
2  $R = \emptyset$ ;   // states for which UIO sequences have been found
3  create the root of the UIO tree and insert it into $E$;
4  while (($E \neq \emptyset$) $\wedge$ ($R \neq S'$)) do
5      $TN$ = pick a node from $E$ with maximum heuristic point;
6      remove $TN$ from $E$ ;
7      if ($|\Phi(TN)| \geq 1$) then
8          forall the $x \in I, y \in O$ do
9              if ($|lbl(TN^{x/y})| == 1$) then
                   // found a UIO sequence
10                 Let $[s, s']$ be the ICS pair in $TN^{x/y}$;
11                 $R = R \cup \{s\}$;
12             else if ($|\Phi(TN^{x/y})| \geq 1$) then
13                 $E = E \cup \{TN^{x/y}\}$ ;
```

The time requirement of the heuristic method is also much better than both the exhaustive and the random method, as seen in Figure 14. Based on these two performances (tree size and time), we can say that the heuristic point measure really works and it guides the search in the UIO tree toward those nodes that will report a UIO sequence.

When the performance of these methods are considered in terms of the length of the UIO sequences found, (see Figure 15), we see that it cannot

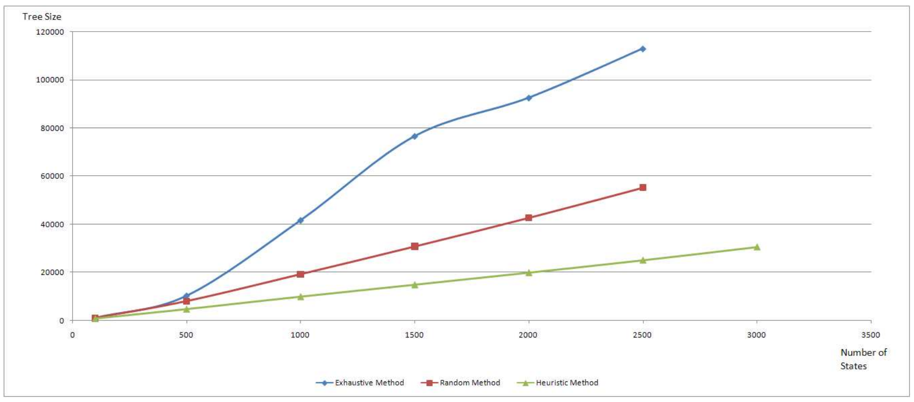


Figure 13: Tree Size Comparison for the Heuristic Method

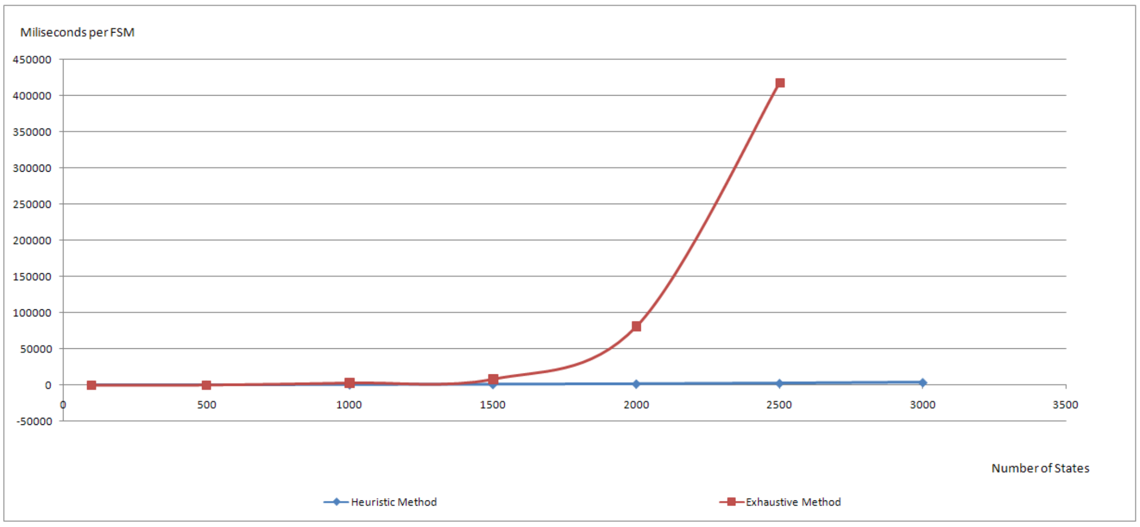


Figure 14: Time Comparison for the Heuristic Method

find as short UIO sequences as the exhaustive method, which might be expected. However, it seems that even the random method does better than the heuristic method, which is suprising. This may be due to the fact that the heuristic method focuses on a node and pushes the search almost in a depth–first manner toward that node. Thus, the UIO sequences found by the heuristic method tend to be longer on the average. However, the gap between the average UIO sequence lengths found by the exhaustive and the heuristic method is not very large.

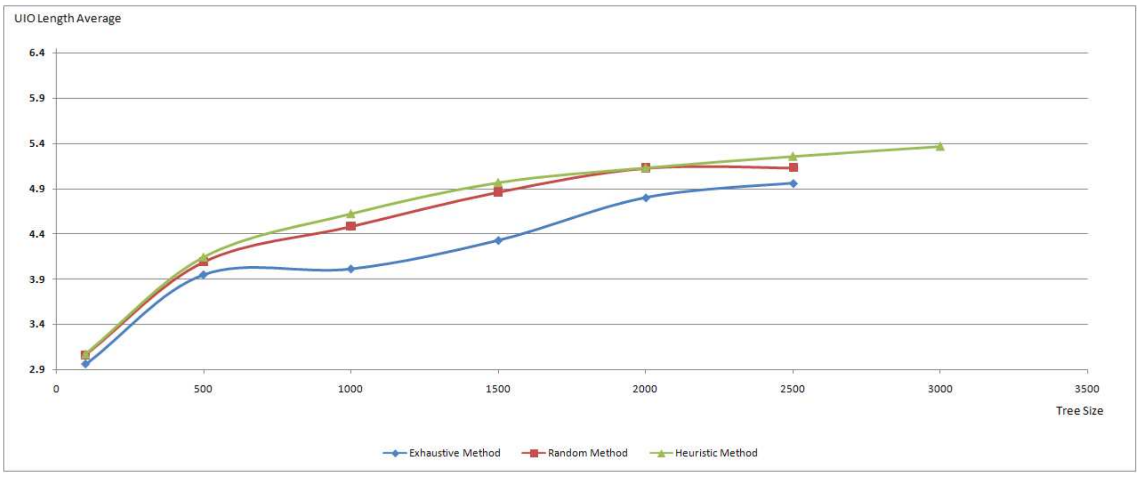


Figure 15: Heuristic Method in Comparison with Random and Exhaustive Method UIO Sequence Lengths.

In order to verify that the use of the heuristic point is really guiding the search toward fruitful parts of the UIO tree and the results are not better just because a disciplined way of exploration of tree is being used, the performance of the reverse of heuristic point has also been experimented. In other words, rather than picking the best node, the worst node is picked at each iteration.

As expected, this approach resulted in huge search trees, even bigger than those generated by the exhaustive method.

4.4 Heuristic Method with Global I/O Ranking

In [13], another heuristic is proposed for UIO sequence generation. However, the authors do not expose the problem in the form of a UIO tree expansion. Instead, they use a genetic algorithm to find I/O sequences likely to be UIO sequences. They formulate the problem in such a way that fitter the I/O sequence, the more likely for them to be UIO sequence.

The basic idea behind the heuristic in [13] is based on the notion of transition ranking. Let $f(x/y)$ be the frequency and $r(x/y)$ be the rank of the frequency of the I/O pair x/y. Formally,

$$f(x/y) = |\{s | \forall s \in S, \lambda(s, x) = y\}|$$

and

$$r(x/y) = |\{f(x'/y') | \forall x' \in I, y' \in O, f(x'/y') < f(x/y)\}|$$

Hence $f(x/y)$ will give us how many states produce the output y to the input x, or in other words, how many times the I/O pair x/y is seen in the FSM. On the other hand, $r(x/y)$ will be the rank of the I/O pair among all other I/O pairs. If $r(x/y) = 0$, this means the I/O pair x/y is the least frequent I/O pair in the FSM, if $r(x/y) = 1$, this means the I/O pair x/y is next least frequent I/O pair in the FSM, etc. Table 4 gives an example for the frequencies and ranks of I/O pairs by using the FSM M_0 of Figure 1.

The basic idea behind the heuristic of [13] is that the less the ranks of the transitions in an I/O sequence, the more the chance for it to be UIO sequence.

Table 4: Frequencies and ranks of I/O pairs in M_0

I/O pair	Frequency	Rank
$a/1$	3	1
$a/2$	2	0
$b/1$	2	0
$b/2$	3	1

We incorporate this heuristic into our methods by favoring the exploration of those paths in which rare transitions are used as much as possible. This is handled as follows. All the methods introduced so far pick a node TN to be explored. They then generate all the children of TN. Instead of this, we will now generate the children of TN one by one, starting with the child for the least frequent I/O pair. After generating a child of TN, we give the algorithm a chance to pick the node to be explored again. If TN is picked again, we will generate another child of TN, but this time with the next least frequent I/O pair.

Let $Q = \langle i_1/o_1, i_2/o_2, ...\rangle$ be a sequence where all I/O pairs (hence Q has $|I| \times |O|$ elements) are sorted in increasing order with respect to their ranks, breaking the ties randomly. Based on the information given in Table 4, it is easy to see that Q for M_0 of Figure 1 can be $Q = \langle b/1, a/2, a/1, b/2\rangle$.

Algorithm 5 displays the new method considering the ranking of the transitions. We call this method as "Heuristic Method with *Global* I/O Ranking" because we will later have an I/O ranking considering individual states. Currently, we consider the I/O ranking globally over the entire FSM.

The modifications required for the algorithm are as follows: There is now a preprocessing phase to compute the frequencies and the ranks of the transition. There is also a computation for Q, the global I/O ranking. Furthermore, when a node TN is picked to be explored in an iteration, it is not removed from E, since it is not necessarily fully expanded. It is removed from E only when TN is visited $|I| \times |O|$ times, which means every child of TN is created.

As can be seen in Figure 16, the tree size performance of the heuristic method with global I/O ranking is the best among all the methods introduced so far. So, we can assume that it is forcing the search to those parts of the UIO tree that will identify a UIO sequence.

When the time performance is considered, we see that its performance is very close to that of the heuristic method. Note that, in Figure 17, the time curves of the exhaustive and the random method are omitted, so that the comparison of the heuristic method with and without global I/O rankings can be seen in more detail.

Figure 18 compares the average UIO sequence lengths of the methods introduced so far. The heuristic method with global I/O ranking is the worst one. This can be expected since, with the introduction of I/O ranking, the search has become operating in a slightly more depth–first manner.

Finally, we would like to emphasize the following: The method presented in this section is based on the basic idea of favoring rare I/O pairs in the search. However, the FSM examples used for comparing the methods in this section have been created randomly. Therefore, the I/O pairs in the FSMs more or less have the same frequency. For such a set of FSMs, one might expect that the method based on I/O ranking will not actually work since there are no transitions which are less frequent than the other ones. However, as explained above, experimental results show that there is an improvement in the tree size performance.

Algorithm 5: Heuristic Method with Global I/O Ranking

```
1  compute $Q$ ;        // list of I/O pairs sorted wrt their ranks
2  $E = \emptyset$ ;                  // UIO tree nodes yet to be explored
3  $R = \emptyset$ ;    // states for which UIO sequences have been found
4  create the root of the UIO tree and insert it into $E$;
5  while $(E \neq \emptyset) \wedge (R \neq S')$ do
6      $TN$ = pick a node from $E$ that has maximum heuristic point;
7      if $(|\Phi(TN)| \geq 1)$ then
8          let $r$ be the current number of children of $TN$ ;
           // $TN$ has been visited before $r$ times
9          let $x/y$ be the $r + 1^{\text{st}}$ I/O pair in $Q$;
10         if $(|lbl(TN^{x/y})| == 1)$ then
               // found a UIO sequence
11             Let $[s, s']$ be the ICS pair in $TN^{x/y}$;
12             $R = R \cup \{s\}$;
13         else if $(|\Phi(TN^{x/y})| \geq 1)$ then
14             $E = E \cup \{TN^{x/y}\}$ ;
15         if $(r + 1 == |I| \times |O|)$ then
16             $E = E \setminus \{TN\}$ ;        // $TN$ is now fully expanded
```

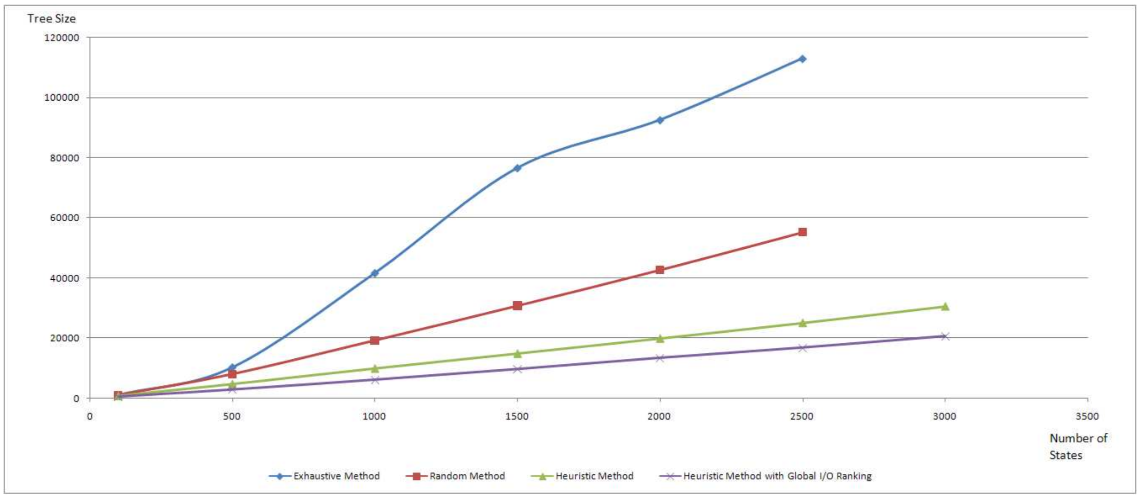


Figure 16: Heuristic Method with Global I/O Pairs in Comparison with Random and Exhaustive Method Results.

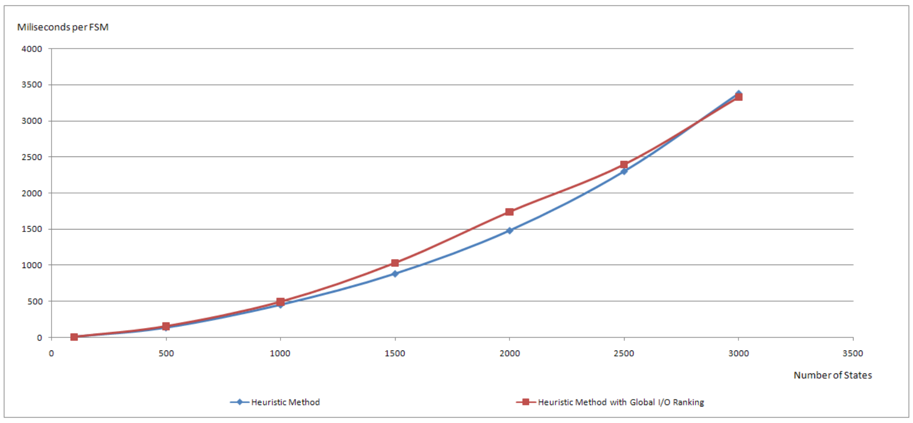


Figure 17: The Execution Time Comparison of Heuristic Method with Global I/O Pairs and Heuristic Method.

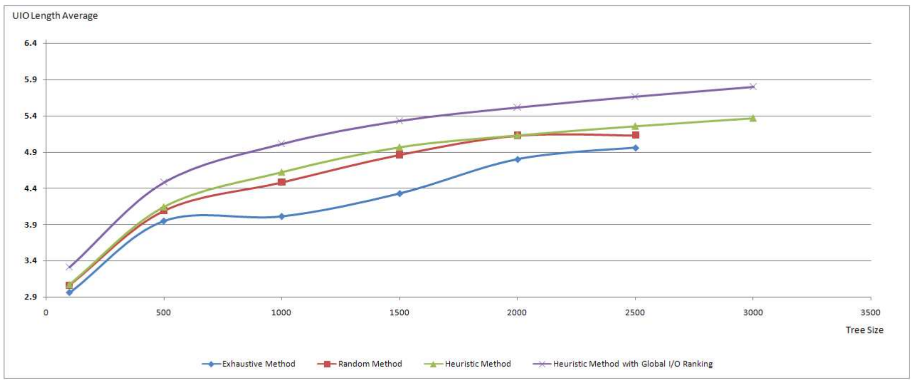


Figure 18: Heuristic Method with Global I/O Pairs in Comparison with Random and Exhaustive Method UIO Sequence Lengths.

We also tried to prove that the reduction in the tree size is really due to the fact that the less frequent I/O pairs are favored. To verify this, we generated the children of the nodes by using the reverse sorted Q. Hence, we first generate the child of a node by using the most frequent I/O pair, and then by using the next most frequent I/O pair, etc. In this trial, the tree size performance got worse as expected. Therefore, we conclude that, even when the frequencies of the I/O pairs have a constant distribution, the method still works.

In order to see the performance of this method (and the other methods that will be introduced and will be using I/O ranking information) when some I/O pairs are really rare, another set of FSMs have been generated by enforcing a certain distribution of the frequencies of I/O pairs. The results of the experiments by using this set of FSMs are given in Chapter 6.

4.5 State–Based Heuristic Method

In order to try out a standard search heuristic to cut down the tree size, the beam search technique has also been applied. When doing the beam search, the best k nodes are considered for exploration and the remaining nodes are terminated forever.

However, there is a pitfall with this approach. Assume that currently the set of states for which we are still searching for a UIO sequence is $S' \setminus R$. Suppose we pick the best k nodes $TN_1, TN_2, \ldots, TN_k$ having the best k heuristic points (the best k heuristic points need not be distinct). Let

$$\phi = \bigcup_{i \in \{1,2,\ldots,k\}} \Phi(TN_i)$$

If the search is restricted to the subtrees of these k nodes, then UIO sequences can be found only for those states in ϕ. If $\phi \subset (S' \setminus R)$, this means that the algorithm is losing the chance of finding a UIO sequence for a state in $(S' \setminus R) \setminus \phi$ completely by restricting the search to these k nodes.For this reason, instead of picking the best k nodes, we pick the best k nodes for each state in $S' \setminus R$ separately as follows: For a state s, let E_s be the set of nodes in E which has a potential for finding a UIO sequence for s. Formally

$$E_s = \{TN | s \in \Phi(TN)\}$$

Define E_s^* as the best k nodes in E_s having the best k heuristic points in E_s. If $|E_s| < k$ we take $E_s^* = E_s$. Then, the set of UIO tree nodes to be expanded is given as follows:

$$E^* = \bigcup_{s \in (S' \setminus R)} E_s^*$$

Table 5: E_s node sets for states

States	E_s	E_s^*
s_1	$\{TN_1, TN_3\}$	$\{TN_3\}$
s_2	$\{TN_1, TN_4\}$	$\{TN_4\}$
s_3	$\{TN_3\}$	$\{TN_3\}$
s_4	$\{TN_4\}$	$\{TN_4\}$
s_5	$\{TN_1, TN_4\}$	$\{TN_4\}$

The other UIO tree nodes in $E \setminus E^*$ are never expanded.

For example, in Figure 19, assume that $S' = S$ and currently $R = \emptyset$. Suppose the beam width k is 1. Table 5 gives E_s for each state. For s_1, $E_{s_1} = \{TN_1, TN_3\}$. The heuristic points for these nodes are $HP(TN_1) = 0.33$ and $HP(TN_3) = 0.5$. Since, we consider $k = 1$ best nodes the algorithm will pick $E_{s_1}^* = \{TN_3\}$. $HP(TN_4) = 0.33$, which is a tie with the heuristic point of TN_1. Assume that TN_4 is selected randomly to break the tie. The sets E_s^* are also given in Table 5 for each state. Based on this, the set E^* is found as $\{TN_3, TN_4\}$.

Since TN_1 is not in E^*, it never gets explored. Note that the TN_2 is a homogeneous node, so it would not be explored any how. After expanding the nodes in $E^* = \{TN_3, TN_4\}$, the UIO tree takes the form given in Figure 20.

As can be seen from Figure 20, the nodes TN_5, TN_6, TN_9 and TN_{10} will report UIO sequences for the states s_1, s_3 ,s_2, and s_5, respectively. In the remaining part of the search, the algorithm will be looking for a UIO sequence of s_4 only. In the next iteration, the candidate nodes to be explored will be

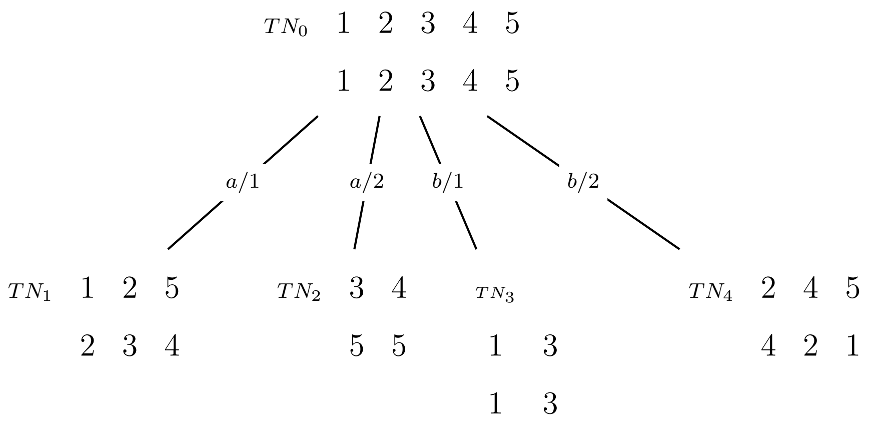


Figure 19: Example for State Based Heuristic Method: Iteration 1

TN_8 and TN_{11}, and the algorithm will pick $k = 1$ best nodes among these as explained above.

The algorithm implementing the approach described above is given as Algorithm 6. This method will be called as "State Based Heuristic Method".

Note that, the kind of search performed by this method executes in a more breadth–first manner than the original heuristic method and it operates similar to the exhaustive method. In the exhaustive method, all the nodes (with positive potentials) at a given level of the UIO tree are expanded. However, in the state based heuristic method, only a selected subset (E^*) of the nodes at a level is expanded. Since its nature is breadth–first, it is expected to have short UIO sequence lengths and high tree sizes. Figures 21 and 22 show that this expectation is correct. The tree size performance is actually much better than the exhaustive and the random methods and close the other methods suggested.

Since the algorithm finds k best nodes for each state in $S' \setminus R$ separately and explores more nodes than the heuristic method, it can be expected that

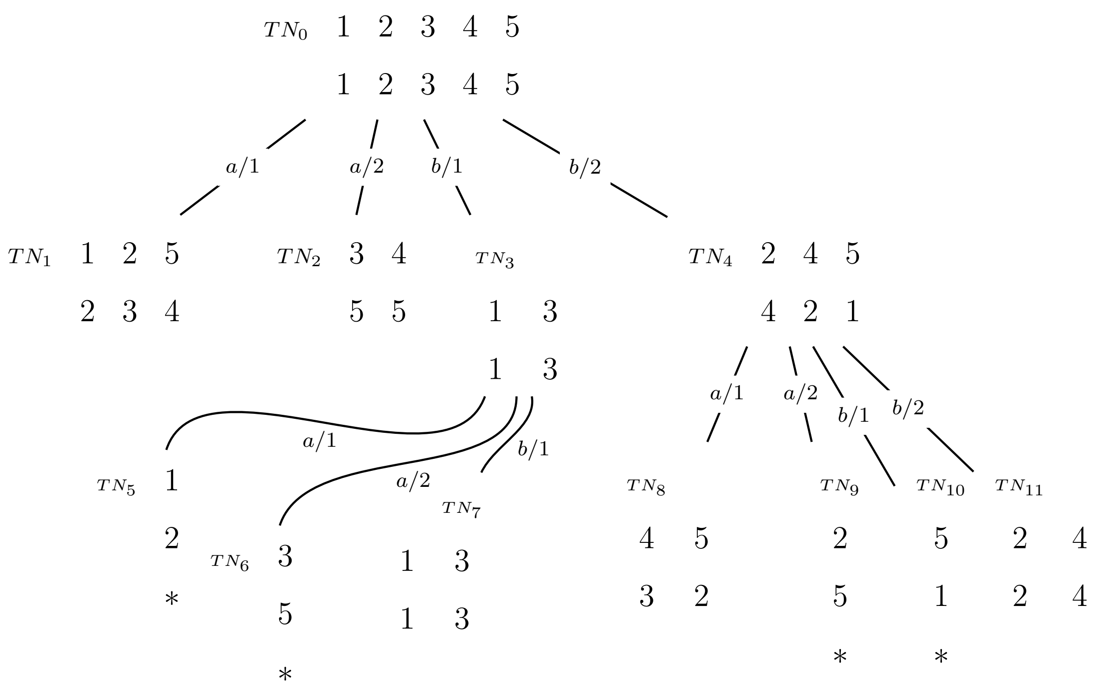


Figure 20: Example for State Based Heuristic Method: Iteration 2

Algorithm 6: State Based Heuristic Method

```
1  $E = \emptyset$ ;                              // UIO tree nodes yet to be explored
2  $R = \emptyset$ ;    // states for which UIO sequences have been found
3  create the root of the UIO tree and insert it into $E$;
4  while $(E \neq \emptyset) \wedge (R \neq S')$ do
5      $E^* = \emptyset$ ;                             // the nodes to be expanded
6      foreach $s \in (S' \setminus R)$ do
7          $E^*_s$ = pick the best $k$ nodes for $s$ from $E_s$;
8          $E^* = E^* \cup E^*_s$;
9      $E = \emptyset$ ;
10     forall the $TN \in E^*$ do
11         forall the $x \in I, y \in O$ do
12             if $(|lbl(TN^{x/y})| == 1)$ then
                   // found a UIO sequence
13                 Let $[s, s']$ be the ICS pair in $TN^{x/y}$;
14                 $R = R \cup \{s\}$;
15             else if $(|\Phi(TN^{x/y})| \geq 1)$ then
16                 $E = E \cup \{TN^{x/y}\}$;
```

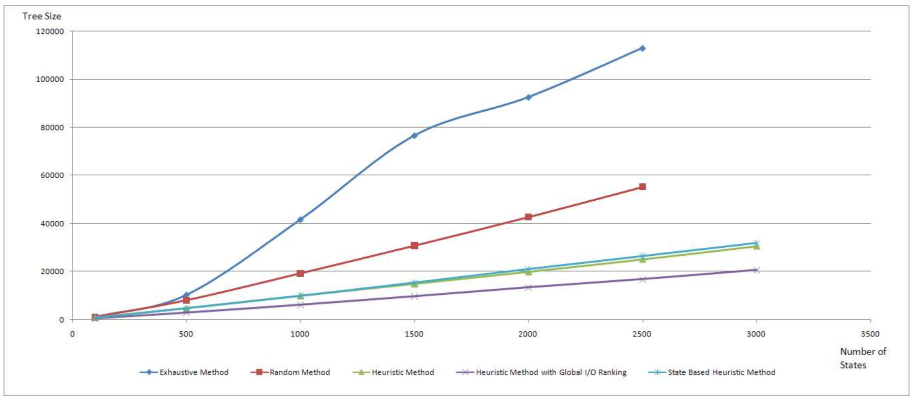


Figure 21: Tree Size Comparison for the State Based Heuristic Method

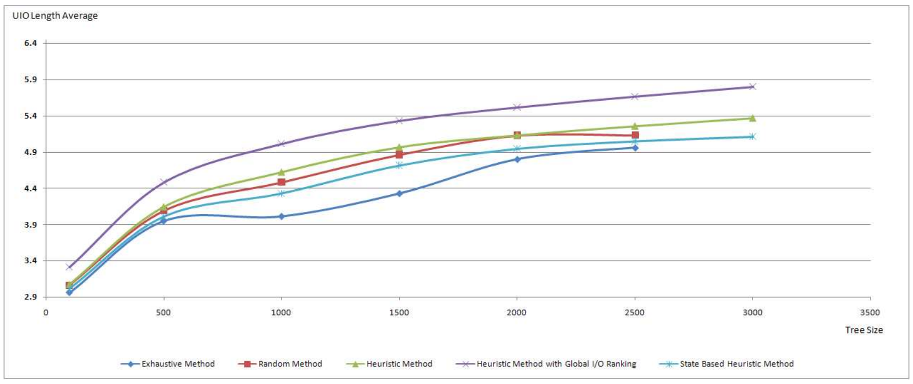


Figure 22: UIO Sequence Length Comparison for the State Based Heuristic Method

it will need more time than the heuristic method. This expectation is also correct as can be observed from Figure 23.

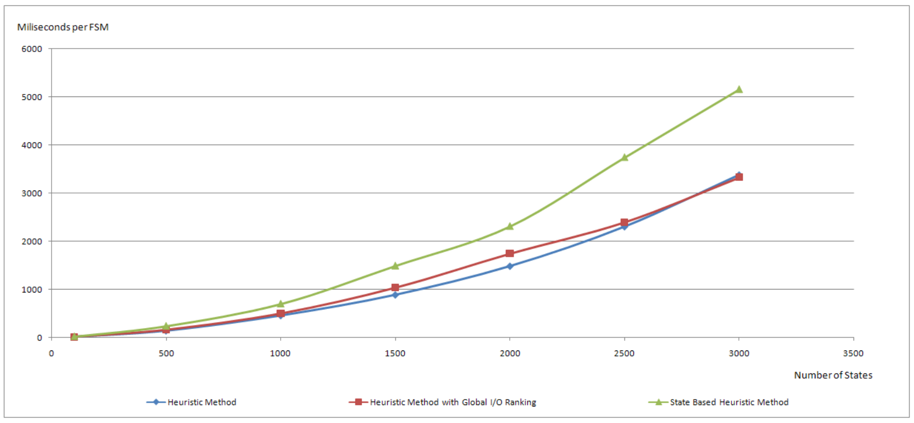


Figure 23: Time Comparison for the State Based Heuristic Method

Finally, it must be noticed that the state based heuristic method is not an exact method, in the sense that even if a state s has a UIO sequence, the state based heuristic method may fail to find a UIO sequence for s. This can happen for example when all the best k nodes picked for s at some iteration of the algorithm lead to homogeneous or repetitive nodes. This is the first inexact method among the methods introduced so far. However, during our experiments, we never encountered a case where the state based heuristic method fails to find a UIO sequence for a state which is known to have one.

Using higher k values might be suggested when the method fails to find a UIO sequence for a state with a UIO sequence. In fact, the value of k may considered as an important factor for the performance of the method. We experimented with different k values. Increasing the value of k obviously increases the tree size and the time required to complete the analysis. It also decreases the average UIO sequence lengths. This is expected as the search gets closer and closer to the exhaustive method as k is increased. However, even when $k = 1$, the method was able to find UIO sequences for all the states (with UIO sequences), and increasing the value of k does not decrease the average UIO sequence lengths (it is already not very bad for $k = 1$) in the same proportion it increases the tree size and the time. Therefore, $k = 1$ is used for all the experiments.

4.6 State–Based Heuristic Method with Global I/O Ranking

One natural improvement suggestion at this point can be to use the I/O ranking information inside the state based heuristric method. In other words, after picking the nodes to be expanded E^*, when handling a node $TN \in E^*$, do not generate all the children of TN, but generate only one child. This child of TN will be generated by considering the least frequent I/O pair not yet used to generate a child of the node TN.

The order of the I/O pairs to be considered is again calculated as described in Section 4.4. So, in this section, we assume that we have a list of I/O pairs Q ordered increasingly in terms of the ranks of the I/O pairs, as in Section 4.4.

Note that, in state based heuristic method, when a node TN is selected, i.e. $TN \in E^*$, TN is fully expanded (all children of TN are generated) and therefore TN is never considered again. However, when the children of TN are generated one by one, TN will survive and continue to be a candidate node to be expanded

- as long as it has at least one more child to be generated; and
- as long as the algorithm keeps selecting TN as one of the k best nodes for at least one state $s \in (S' \setminus R)$

Therefore, a node TN might be *partially expanded* during the execution of the algorithm. As soon as all the children of TN are created (i.e. when TN gets fully expanded) or as soon as the algorithm does not consider TN as one of the best k nodes for a state, TN (even if it is partially expanded) is taken out of the candidate nodes to be expanded and it is never picked again.

The algorithm implementing this new approach is given as Algorithm 7. The method will be called as "State Based Heuristic Method with Global I/O Ranking".

This method introduces some depth–first exploration into the picture since it considers a child of a node at a time. The effect of this feature is seen in the test results as an increase in the average UIO sequence lengths (Figure 25) and a decrease in the tree size (Figure 24). In fact, this method is the one with the smallest tree size among all the methods discussed so far.

The time requirement of this algorithm is also the best so far (Figure 26). It shares the last position in UIO sequence length performance together with the heuristic method with global I/O ranking. When the time and the tree size performances of these two methods are compared, it can be stated that the state–based heuristic method with global I/O ranking is superior to the heuristic method with global I/O ranking.

As mentioned in the previous section, the selection of only k nodes for a state s makes the method inexact, hence, it may not find a UIO sequence for s, even though it exists. In our experiments with $k = 1$, we have observed

Algorithm 7: State Based Heuristic Method with Global I/O Ranking

```
1  compute $Q$ ;       // list of I/O pairs sorted wrt their ranks
2  $E = \emptyset$ ;                      // UIO tree nodes yet to be explored
3  $R = \emptyset$ ;    // states for which UIO sequences have been found
4  create the root of the UIO tree and insert it into $E$;
5  while $(E \neq \emptyset) \wedge (R \neq S')$ do
6      $E^* = \emptyset$;
7      foreach $s \in (S' \setminus R)$ do
8          $E^*_s$ = pick the best $k$ nodes for $s$ from $E_s$;
9          $E^* = E^* \cup E^*_s$;
10     $E = \emptyset$;
       forall the $TN \in E^*$ do
11         let $r$ be the current number of children of $TN$;
           // $TN$ has been visited before $r$ times
12         let $x/y$ be the $r+1$st I/O pair in $Q$;
13         if $(|lbl(TN^{x/y})| == 1)$ then
               // found a UIO sequence
14             let $[s, s']$ be the ICS pair in $TN^{x/y}$;
15             $R = R \cup \{s\}$;
           else if $(|\Phi(TN^{x/y})| \geq 1)$ then
16             $E = E \cup \{TN^{x/y}\}$ ;
17         if $(r + 1 \mathrel{!}= |I| \times |O|)$ then
18             $E = E \cup \{TN\}$ ;    // $TN$ is not fully expanded yet
```

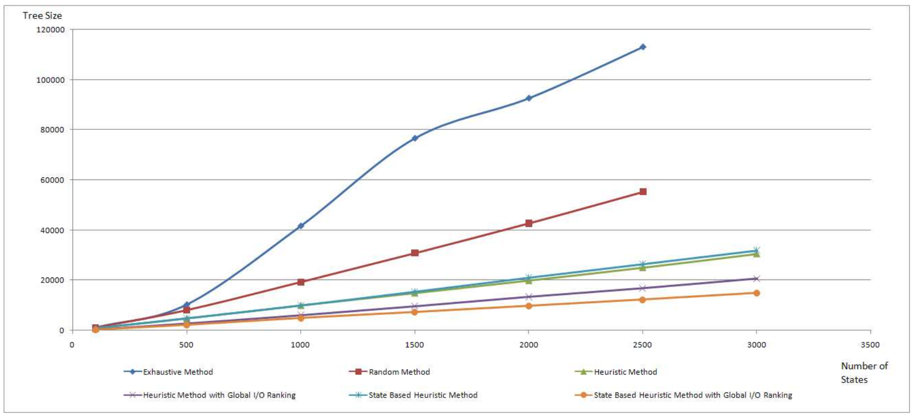


Figure 24: Tree Size Comparison for State Based Heuristic Method with Global I/O Ranking

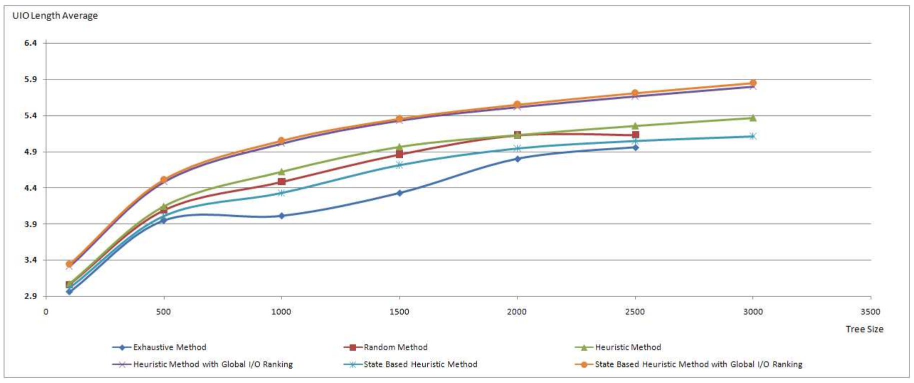


Figure 25: UIO Sequence Length Comparison for State Based Heuristic Method with Global I/O Ranking

that such a miss happens for 1/1000 states approximately. The miss rate can be reduced very easily by increasing the value of k, which in turn increases the tree size and the time slightly.

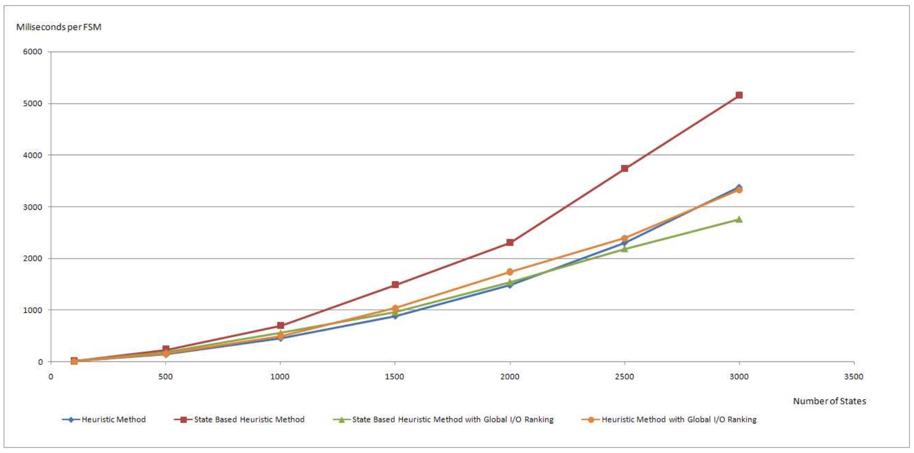


Figure 26: Time Comparison for State Based Heuristic Method with Global I/O Ranking

4.7 Depth–First Heuristic Method with Global I/O Ranking

Please note that all the methods introduced so far perform the search for UIO sequences for all the states simultaneously. In other words, during the search there is a list of states for which no UIO sequence has been found yet and the nodes are checked if they reveal a UIO sequence for one of these states. The heuristic score proposed also makes use of such an approach. The more the number of states for which it has a potential to find a UIO sequence, the higher the value of the heuristic point of that node.

Instead of such an approach, a separate search can also be used for each state individually. For example, the method described in [13] is such a method that repeats the search for each state separately.

Recall that we denoted the set of states for which we have a hope to find a UIO sequence as S'. Suppose that we explicitly set $S' = \{s_1\}$ and start a search (by using any one of the methods explained) provided that s_1 has at least one non–converging transition. This would be a search specific to the state s_1. After this search is completed, one can start another search by setting $S' = \{s_2\}$, so on and so forth.

When a search for a UIO sequence of a single state is considered, one of the natural suggestions can be to perform a depth–first search. We will show in this section that, when the heuristic method (with or without global I/O ranking) is considered in the context of a search for a single state, it actually reduces to a depth–first search of the UIO tree. The state based heuristic method (with or without the global I/O ranking) with $k = 1$ also reduces to depth–first search (this time without backtracking), when we consider it for finding a UIO sequence for only one state.

Suppose $S' = \{s\}$ for some state s. By using the definition of $\Phi(TN)$ of a node TN (see page 41), it is easy to see that during such a search, either $\Phi(TN) = \emptyset$ or $\Phi(TN) = \{s\}$. In the case that $\Phi(TN) = \emptyset$, TN has no potential for finding a UIO sequence for the state s and hence, it should not be considered for expansion during the search for a UIO sequence of s. Since the heuristic point of a node TN is computed as

$$HP(TN) = \frac{|\Phi(TN)|}{|curr(lbl(TN))|^2}$$

for a node TN with a potential to find a UIO sequence for the state s the heuristic point will be:

$$HP(TN) = \frac{|\Phi(TN)|}{|curr(lbl(TN))|^2} = \frac{|\{s\}|}{|curr(lbl(TN))|^2} = \frac{1}{|curr(lbl(TN))|^2}$$

Therefore for a node TN that has a potential for the state s, the only factor in the heuristic point of TN is the size of the current set of TN.

The following observation explains why the previous methods reduce down to a depth first search in the context of a search for a single state.

Remark 7

Let TN be a node and $TN^{x/y}$ be a child of TN. Either $\Phi(TN^{x/y}) = \emptyset$ or

$HP(TN^{x/y}) \leq HP(TN)$.

This remark is correct since the size of the current set of $TN^{x/y}$ cannot be greater than that of TN. Please see Remark 3 to recall how the current set of $TN^{x/y}$ is computed from the current set of TN. Each state in the current set of TN will be mapped into a state in the current set of $TN^{x/y}$ and some of the states in the current set of TN can be merged into the same state during this mapping. However, those current states that are already same in TN cannot be mapped into different states in $TN^{x/y}$.

According to Remark 7, when a node TN is selected (since it has the best heuristic score among all the alternatives) and expanded, a child of TN with a non–zero potential will have at least the same score as TN. Hence, it will also have the best heuristic score among all alternatives. Therefore, after generating a child $TN^{x/y}$ of TN, if $TN^{x/y}$ has a non–zero potential, the search can proceed by expanding $TN^{x/y}$, the child of the last node expanded. This is nothing but a depth–first search.

We implemented the depth–first search (with backtracking) of the UIO tree in the following manner. The method repeats the search for each state separately. The children of a node are generated one at a time. The order that is used to generate the children is based on the global I/O pair ranking information again. When a node is re–visited during backtracking, its next child with the next I/O pair in the global ranking is generated. Algorithm 8 implements the method.

Note that, due to the structure of the search, it is not necessary to find a node with the minimum heuristic point anymore by performing a search over all the nodes (by Remark 7). It is not even needed to compute the heuristic scores.

Note that Algorithm 8 performs a search for each state. In practice, it may perform a search for s, creating a tree. In the search performed for another state s', the algorithm may re–visit some parts of the tree explored during the search for s. Since memory allocation is rather an expensive operation in practice, the algorithm is implemented in the following way in order to improve the time performance in practice. The algorithm never deletes a tree node created. During the search, when the algorithm wants to visit a child $TN^{x/y}$ of a node TN, if $TN^{x/y}$ has never been visited before during the searches for the previous states, the node $TN^{x/y}$ is created. However, if the previous searches created the node $TN^{x/y}$, it is not created again. Therefore, the trees of the searches are shared.

The corresponding experimental results can be seen in Figure 27 in comparison with the aforementioned methods in previous sections. It can be observed from these results that this method has lower memory requirements

Algorithm 8: Depth First Heuristic Method with Global I/O Ranking

```
1  compute $Q$ ;        // list of I/O pairs sorted wrt their ranks
2  let $E$ be an initially empty stack for UIO tree nodes yet to be explored;
3  $R = \emptyset$ ;    // states for which UIO sequences have been found
4  let $T$ be the root of the UIO tree ;
5  forall the $s \in S$ do
6      push $T$ on stack $E$ ;
7      $found$ = false;
8      while (E is not empty $\wedge$ not(found))) do
9          $TN$ = top node on stack $E$ ;          // $TN$ is not popped
10         let $r$ be the current number of children of $TN$ ;
           // $TN$ has been visited before $r$ times
11         if $(r == |I| \times |O|)$ then
               // $TN$ has been expanded fully
12             pop $TN$ from $E$;
13         else
14             let $x/y$ be the $r + 1^{\text{st}}$ I/O pair in $Q$;
15             if $(init(lbl(TN^{x/y})) == \{s\})$ then
                   // found a UIO sequence for $s$
16                 $found$ = true; $R = R \cup \{s\}$;
               else if $(|\Phi(TN^{x/y})| == 1)$ then
18                 push $TN^{x/y}$ on stack $E$;
19     remove all nodes in $E$;
```

than State–Based Heuristic Method with Global I/O. However, it finds longer UIO sequences which is a consequence of depth–first approach and it can be seen in Figure 28. The time performance can be seen in Figure 29 and it is best so far when compared to the previous methods.

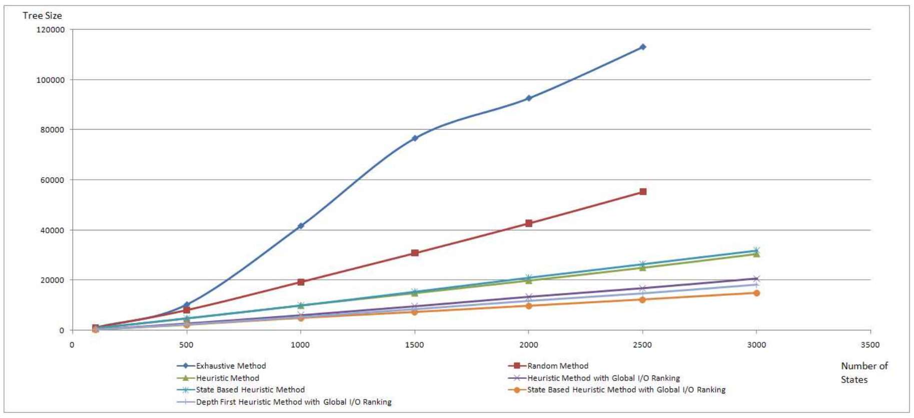


Figure 27: Tree Size Comparison for Depth First Heuristic Method with Global I/O Ranking

4.8 Depth–First Heuristic Method with State Based I/O Ranking

When the algorithm generates the children nodes using global I/O ranking for a specific state, say $s \in S$, it is possible to generate a child node that does not contain s as an initial state. Let $Q = \langle i_1/o_1, i_2/o_2, \ldots \rangle$ denote the list of I/O pairs of an FSM which is in increasing order with respect to the transition ranks. Let $[s, s']$ be an ICS pair within the label of a node TN, hence $s \in init(lbl(TN))$. According to Remark 3 on page 24, $s \in init(lbl(TN^{x/y}))$

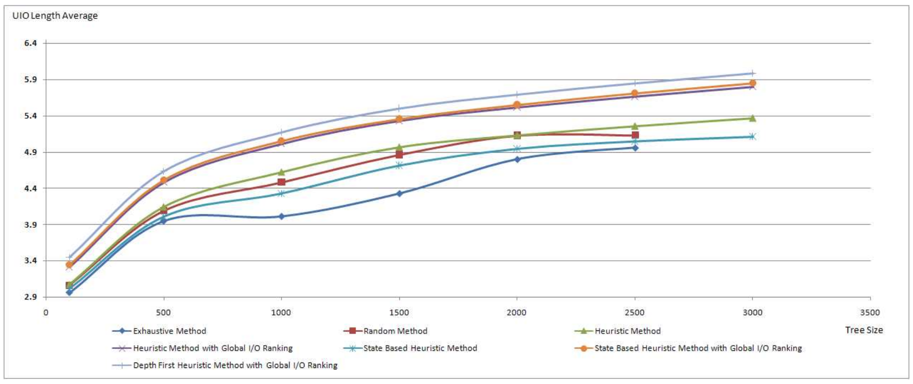


Figure 28: UIO Sequence Length Comparison for Depth First Heuristic Method with Global I/O Ranking

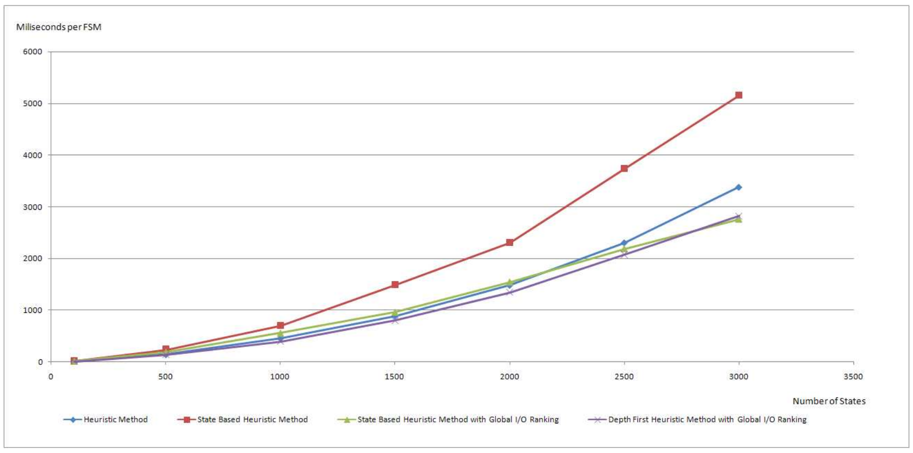


Figure 29: Time Comparison for Depth First Heuristic Method with Global I/O Ranking

if and only if $\lambda(s', x) = y$. Therefore, during a search of a UIO sequence of the state s, if we have such a node TN, it is only meaningful to consider $TN^{x/y}$ for those I/O pairs x/y where $\lambda(s', x) = y$. Hence, not all of the members of the list Q will be used.

For a state s, let Q_s be the projection of Q onto the I/O pairs of the state s. For example, for M_0 given in Figure 1, the global I/O ranking is $Q = \langle b/1, a/2, a/1, b/2\rangle$. The I/O pairs of the state s_1 is $\{a/1, b/1\}$. Therefore $Q_{s_1} = \langle b/1, a/1\rangle$. Note that Q has upto $|I| \times |O|$ elements but Q_s has maximum $|I|$ elements only. When the FSM is completely specified, Q_s will always have $|I|$ elements.

During the search for a UIO sequence of a state s, when considering the children of a node TN where $[s, s'] \in lbl(TN)$, it only makes sense to consider the children $TN^{x/y}$ of TN for those I/O pairs x/y that appear in $Q_{s'}$. For an I/O pair x/y that is not in $Q_{s'}$, it is guaranteed that $TN^{x/y}$ will have no potential for s.

The updated and final algorithm is given as Algorithm 9 and resulting experimental results are in Figure 30, Figure 28 and Figure 31.

Due to the new approach for picking the I/O pairs more wisely, the tree size performance is improved and this method becomes the best method in terms of the tree size constraint. It is also the best method in terms of the time performances.

When the UIO sequence length performance of this method is considered however, this method is guaranteed to find exactly the same UIO sequences that are found by the depth–first method with global I/O ranking. Therefore, the UIO sequence performance will be the same as the performance of the

Algorithm 9: Depth First Method with State Based I/O Ranking

```
1  $\forall s \in S$, compute $Q_s$ ;        // I/O pairs sorted wrt their ranks
2  let $E$ be an initially empty stack for UIO tree nodes yet to be explored;
3  $R = \emptyset$ ;    // states for which UIO sequences have been found
4  let $T$ be the root of the UIO tree ;
5  forall the $s \in S$ do
6      push $T$ on stack $E$ ;
7      $found$ = false;
8      while ($E$ is not empty $\wedge$ not(found))) do
9          $TN$ = top node on stack $E$ ;            // $TN$ is not popped
10         let $[s, s'] \in lbl(TN)$ ;   // such an ICS pair exists at $TN$
11         let $r$ be the current number of children of $TN$ ;
           // $TN$ has been visited before $r$ times
12         if ($r == |I|$) then
               // $TN$ has been expanded fully
13             pop $TN$ from $E$;
14         else
15             let $x/y$ be the $r + 1^{\text{st}}$ I/O pair in $Q_{s'}$;
16             if ($init(lbl(TN^{x/y})) == \{s\}$) then
                   // found a UIO sequence for $s$
17                 $found$ = true;
18                 $R = R \cup \{s\}$;
19             else if ($|\Phi(TN^{x/y})| == 1$) then
20                 push $TN^{x/y}$ on stack $E$;
21     remove all nodes in $E$;
```

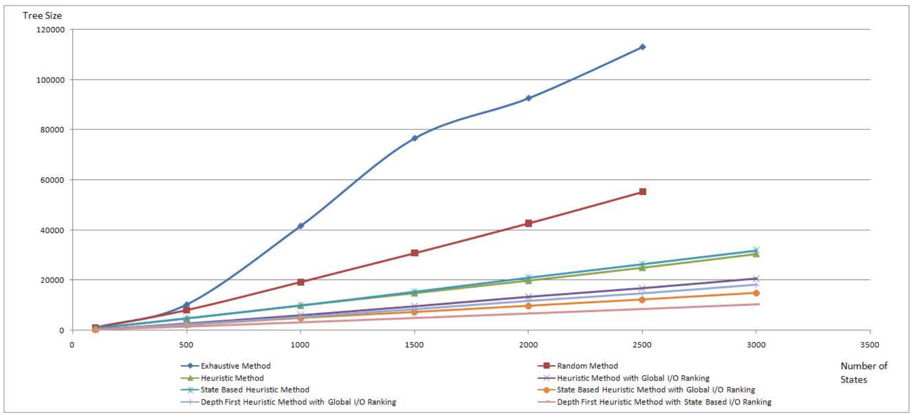


Figure 30: Tree Size Comparison for Depth First Heuristic Method with State Based I/O Ranking

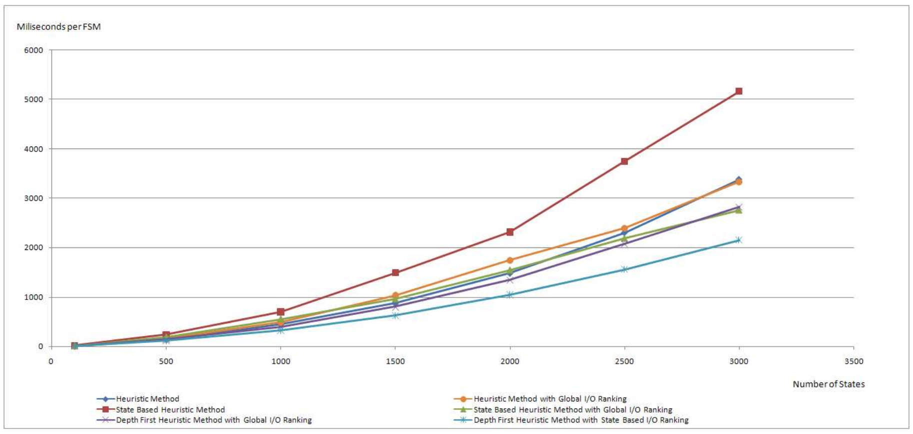


Figure 31: Time Comparison for Depth First Heuristic Method with State Based I/O Ranking

depth first method with global I/O ranking as seen in Figure 28.

4.9 Splitting Point

Let us consider a partial UIO tree and assume that we are looking at the current leaves of this partial UIO tree and trying to find a useful node to expand. It would be nice to know the size of the subtree that would be generated under these leaf nodes. One might prefer the expansion of the smaller subtrees first, because such subtrees will be revealing UIOs earlier than the other, bigger subtrees. Equivalently, it would be nice to expand a node whose current set of states are easily separable from each other.

For two states s and s', a separating sequence is an input sequence $X \in I^\star$ such that $\lambda(s, X) \neq \lambda(s', X)$. Such a separating sequence exists for each pair of states in a minimal machine. For a set of states S', we can similarly define a separating sequence.

When we consider a leaf node TN at a partial UIO tree, one has to separate the set of states $curr(lbl(TN))$ to find UIO sequences in the subtree that will be rooted at TN. Therefore, the length of the separating sequence for the set of states for $curr(lbl(TN))$ will be the depth of the subtree rooted at TN. However such a separating sequence may not exist for all subset of states. Furthermore, it is computationally expensive to find such a separating sequence as well.

The separating sequence of two states is the fundamental element in order to instrument this estimation. For the tree node's current states that correspond to tail states of the potential initial states, the algorithm takes the mean of pairwise separating sequence lengths.

The pairwise separating sequence of each state is calculated as follows: The algorithm creates a splitting graph that has $(|S| \times (|S| - 1))/2$ vertices and a dummy vertex representing the separation condition of these two states. Note that $(|S| \times (|S| - 1))/2$ represents all unordered state pairs excluding the reflexive state pairs. If two states have a transition that is invoked by the same input symbol and produce different outputs, the vertex that represents the state pair have an edge to the dummy state. If two states cannot be separated by this input symbol, there exists an edge from the vertex to the destination vertex which is the representative of the destination states headed by these two state pairs when they are invoked by the input symbol. Note that, two states must have a splitting sequence because of the fact that the FSM that is handled is minimal. So, the pairwise splitting sequence of a state pair is the shortest path from its splitting vertex to the dummy vertex. The splitting point of a pair of states (s, s'), $split(s, s')$, is considered to be the length of the shortest path from the vertex representing the state pair (s, s') to the dummy vertex of the graph. In order to predict the separability of the current states of a node TN based on the pairwise separability of the states, the following formula is used:

$$split(TN) = \frac{\sum\limits_{s,s' \in \Phi(TN)} split(s, s')}{|\Phi(TN)|}$$

However, the experimental results were not so powerful when we combined this splitting score to the aforementioned heuristic approaches. The problem is to not consider the current states as a whole. The pairwise separability information of the states is not a good measure for the separability of a set of states. The state complexity of current splitting graph is $|S|^2$ since we consider the separability of two states only. If we try to increase complexity of the analysis and consider the separability of k states, the resulting splitting graph will have an order of $|S|^k$. Thus, it grows exponentially and as k increases the algorithm will converge to the exhaustive search of UIO generation. In addition, when we analysed the splitting sequence lengths of the states, we encountered generally low values. Apparently, the mean of the splitting point values cannot give a good estimation.

5 Inference Handling

Another contribution of this work is on the *inference method* that is proposed by Naik in [20]. Inference is a mechanism that finds new UIO sequences by inferring from the existing ones without generating any UIO tree nodes. This decreases the size of the UIO tree dramatically.

The inference of new UIO sequences is held by extracting *unique transitions* of an FSM. A unique transition is an incoming transition of a state such that the corresponding I/O pair is unique among the incoming transitions to that state. Formally, the uniqueness condition is:

Let $e \in E$ and $E^e = \{\forall e' \in E : tail(e') = tail(e), lbl(e') = lbl(e)\}$.

If $|E^e| = 1$, then e is a unique transition.

If we know the UIO of a state s_i, we may produce a UIO sequence for a state s_j by the label of a unique transition from s_i to s_j and UIO of s_i. More formally, an *inference rule* is given as follows:

$$UIO_j = lbl(e) + UIO_i \text{ where } e \text{ is a unique transition such that } tail(e) = v_i, head(e) = v_j.$$

It is possible to extract such inference rules and represent all rules for an FSM by using an *inference graph* $IG = (V, E_{ig})$, which is described below:

$E_{ig} = \{\forall e \in E : |E^e| = 1\}$

The inference graph of M_0 given in Figure 1 can be seen in Figure 32. Observe that in Figure 1, (s_4, s_5) and (s_3, s_5) edges have the same label which violating the uniqueness. This graph represents the following rules:

- $UIO_5 = lbl(s_5, s_1) + UIO_1$
- $UIO_1 = lbl(s_1, s_2) + UIO_2$
- $UIO_4 = lbl(s_5, s_1) + UIO_2$
- $UIO_2 = lbl(s_2, s_3) + UIO_3$
- $UIO_2 = lbl(s_2, s_4) + UIO_4$
- $UIO_5 = lbl(s_5, s_4) + UIO_4$

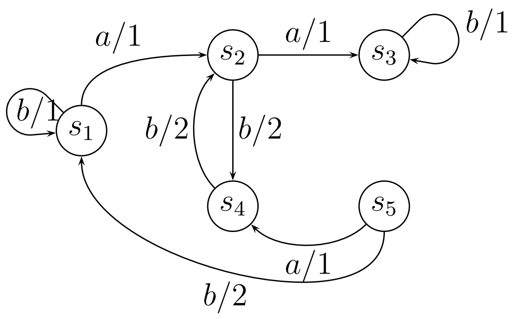


Figure 32: The Inference Graph of M_0

After finding the inference graph of a given FSM, Naik's method [20] has an attempt to produce new UIO sequences to the states for which no UIO sequence has been found yet. If the inference rules fail to find UIO sequences for those states, it then tries to find UIO sequences via generating the UIO tree randomly. This generation approach leads to a significant reduction of the tree size. However, a well known drawback of this inference method is that it generates longer UIO sequences.

5.1 Integration of the Inference Information

As we have described in Section 4.3, the heuristic method tries to find a subtree which has the maximum number of candidate UIO sequences while minimizing the size of the subtree. With the integration of the inference information, our heuristic will still try to find the subtree that will reveal UIO sequences but it will also take into account the UIO sequences that will be inferred by the UIO sequences that will be found in the UIO tree. We add *inference points* to a tree node in order to search intelligently the UIO tree and guide search.

We will define inference points of a tree node as follows: In an inference graph, as the number of incoming edges of a state increases, the number of inferred UIO sequences increases. In order to find the number of states for which a UIO will be constructed by inference, an all–pairs shortest path algorithm [31] is used. An example inference graph can be observed in Figure 33. If the UIO of s_1 is found, the UIO sequences for states s_2, s_4, s_5 will be found by inference rules. For this reason, in order to reduce the UIO tree size, it is better to find UIO of s_1 rather than finding the UIO sequence of, s_2, s_4, or s_5. Therefore, the inference point of a state s can be formulated as the number of states whose UIO sequences would be inferred once the UIO sequence of s is found.

However, the structure of the inference graph is also important for the length of the UIO sequences that will be inferred. When Figure 33 and Figure 34 are compared, it can be seen that the UIO length of s_2 is different in the case of s_1's UIO discovery even though in both cases, the discovery of a UIO sequence of the state s_1 will infer UIO sequences for the states s_2, s_4, and s_5. This difference should also be reflected to the inference point of a state. Hence, the inference score is defined as follows:

Let $IG = (S_{IG}, V_{IG})$ denote an inference graph, $s_j \Rightarrow s_i$ denote the existence of a path from s_j to s_i in an inference graph and $shPath(s_j, s_i)$ denote the length of the shortest path from s_j to s_i. So, the inference point of a state s_i for a tree node TN is:

$$\sum_{\substack{s_j, s_i \in S_{IG} \\ s_j \notin F \\ s_j \Rightarrow s_i}} \frac{1}{shPath(s_j, s_i)}$$

where F is the set of states for which a UIO sequence has been found.

The inference point of s_1 for a tree node TN such that $\Phi(TN) = \{s_1, s_2, s_3, s_4\}$ is $infPoint(s_1) = 1/1 + 1/2 + 1/2 = 2$ in Figure 33 whereas $infPoint(s_1) = 1/1 + 1/2 + 1/3 \cong 1.8$ in Figure 34.

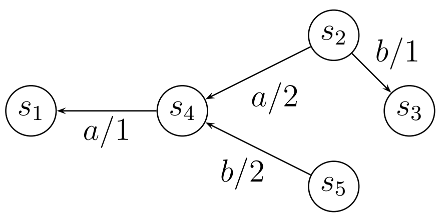


Figure 33: An Example Inference Graph

When a node's inference point is considered, one should analyze the initial states of a tree node. Simply, the ICS pairs in the potential set can be used for inference point calculation by summing up the inference points of the initial states of the ICS pairs. However, this can lead to an over–estimation

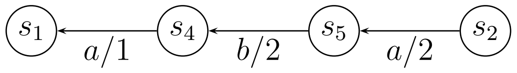


Figure 34: Another Example Inference Graph

of a tree node. For instance, let $lbl(TN) = \{[s_1, s'_1], [s_3, s'_3]\}$ be a node in UIO tree and considered FSM has the inference graph of Figure 33. The inference point of this tree node will be

$infPoint(TN) = infPoint(s_1) + infPoint(s_3)$

$infPoint(TN) = (\frac{1}{shPath(s_4,s_1)} + \frac{1}{\mathbf{shPath(s_2,s_1)}} + \frac{1}{shPath(s_5,s_1)}) + (\frac{1}{\mathbf{shPath(s_2,s_3)}})$

which will double count s_2 as an inference point and give an over-estimated result. That is, the discovery of s_1 and s_3 both invokes the inference of s_2's UIO sequence. We should not double count the contribution of s_2. For this reason, we redefined the inference point of a tree node as follows.

$$\sum_{\forall s_k \in (S \setminus F)} \max\{\frac{1}{shPath(s_k, s)} | \forall s \in \Phi(TN)\}$$

In the extend of inference graph in Figure 33, the inference point of a node TN with $lbl(TN) = \{[s_1, s'_1], [s_3, s'_3]\}$ will be:

$$infPoint(TN) = \max\{\frac{1}{shPath(s_4, s_1)}\} + \max\{\frac{1}{shPath(s_2, s_1)}, \frac{1}{shPath(s_2, s_3)}\} + \max\{\frac{1}{shPath(s_5, s_1)}\}$$

$infPoint(TN) = \max\{1/1\} + \max\{1/2, 1/1\} + \max\{1/2\} = 1 + 1 + 1/2 = 2.5$

The introduction of the inference information into the heuristic method will lead to changes in the heuristic point of a tree node. We redefine heuristic point in the case of inference usage as follows:

$$HP(TN) = \frac{|\Phi(TN)|}{|curr(lbl((TN)))|^2} + \frac{infPoint(TN)}{|curr(lbl(TN))|}$$

In Figure 35, 36 and 37, the tree size and UIO sequence length comparison can be observed. From Figure 35, it can be said that the resulting tree size values are dramatically low due to the inference mechanism. In Figure 36, it can be seen that the proposed heuristic method further improves Naik's tree size results. Actually, this improvement is expected; because we are forcing the algorithm to search the tree nodes in which we can make more inference. However, the more inference the algorithm does, the longer the resulting UIO sequences. The UIO sequence comparison can be observed in Figure 37.

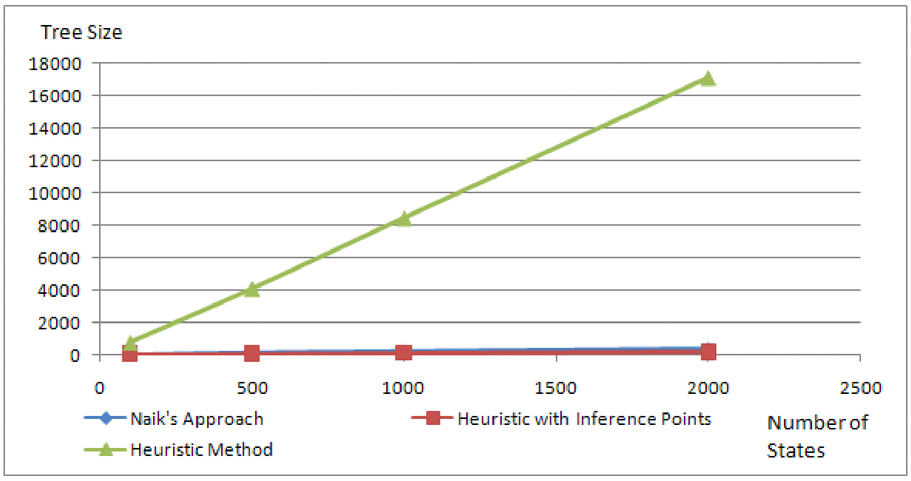


Figure 35: The Tree Size Values of Heuristic Method, Heuristic Method with Inference and Naik's Method.

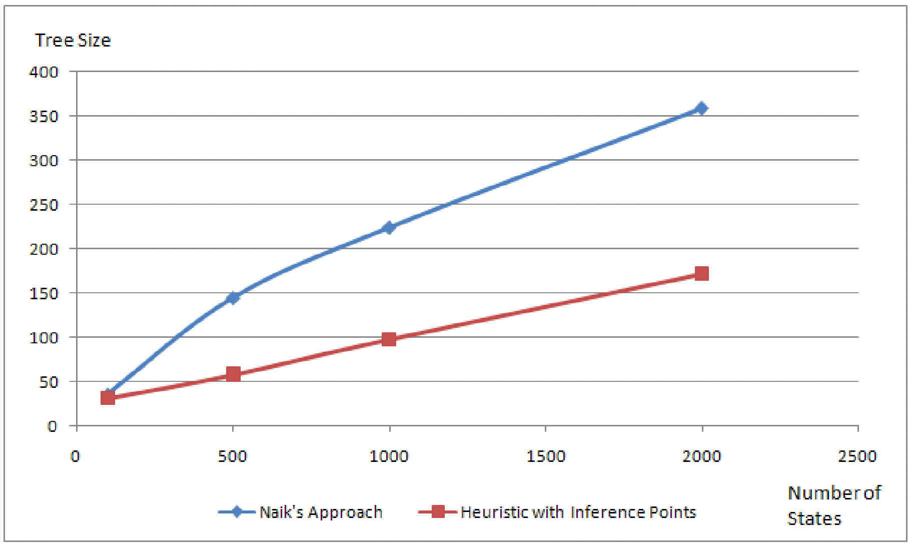


Figure 36: The Tree Size Values of Heuristic Method with Inference and Naik's Method.

5.2 Limited Inference

As expected and as shown by experimental results, the UIO sequences gets long when the inference method introduced in the previous section is used. In this section, we suggest a method to control the length of the UIO sequences produced when the inference method is used.

In this approach, when the UIO sequence of state s_i is found, only a limited length I/O sequence is allowed to be concatenated to the UIO of s_i. For this reason, the inference score of a state is different when the limited inference approach is employed. For example, in Figure 33, when the inference limit is set to 1, by using a UIO sequence of the state s_1, only the UIO sequence of the state s_4 is allowed to be inferred since it requires only

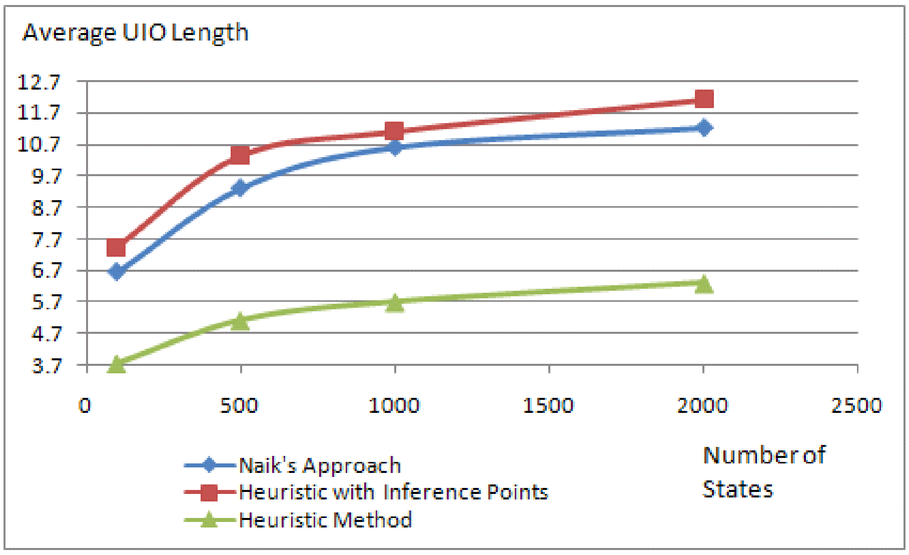


Figure 37: The UIO Sequence Lengths of Heuristic Method, Heuristic Method with Inference and Naik's Method.

one I/O label $(a/1)$ to be concatenated to the UIO sequence of the state s_1. The other states need 2 or more I/O labels to be concatenated to the UIO sequence of s_1. Therefore, when the inference limit is 1, the inference point of s_1 is 1.

In order to find limited length inference points, all–pairs shortest path algorithm is parameterized with the length value of the inference. As a result, the shortest paths using at most *inference depth* edges are found and the inference point of a tree node is calculated accordingly.

The experimental results can be seen in Figure 38. At the very low and right end we see the Naik's method with inference. It has the longest UIO sequence length and the smallest tree size. At the very left and top end we see the Naik's method "without" inference (i.e. a random UIO tree

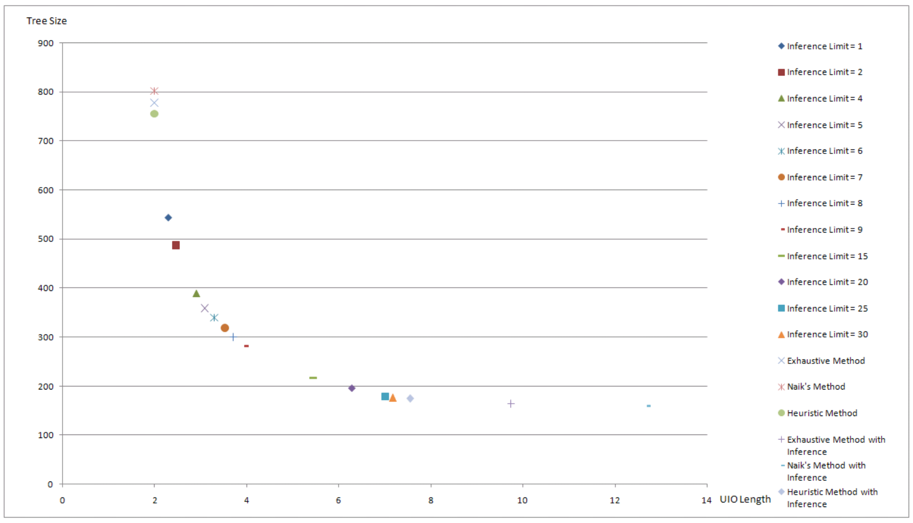


Figure 38: Different Inference Lengths in Comparison.

generation), the exhaustive method and our basic heuristic method (without any inference). As we allow inference to be introduced into our heuristic method and as we increase the amount of the limited inference, we see that a nice curve is formed connecting these two extrema. This proves that the limited inference can be used to control the average length of UIO sequences that will be generated by using the inference method. The cost that will be paid for getting shorter UIO sequences will be the memory and hence the time.

6 Experimental Results

The experimental results are conducted in Pentium 4/2.40 GHz computer that has 1024MB of RAM. The FSM and heuristic method implementations are coded in Java. The timing parameters in previous sections are measured using Java Execution Time Measurement Library. Also, the implementations are profiled and optimized for best performance with the aid of Eclipse TPTP plug-in.

In this section, the experimental results will be discussed. The first section compares the results of Depth–First Heuristic Method with LANG [12] algorithm. The second section will show the results for FSMs for which the transitions are arranged with different distributions and variances. In the third section, we will demonstrate the performance and the memory requirements of the proposed heuristic by experimenting on FSMs that have upto 50,000 states.

6.1 Experimental Results for Benchmark FSMs

The comparison between LANG [12] and Depth–First Heuristic Method with State Based I/O ranking on several MCNC benchmark FSMs [39] can be seen in Table 6. In these results, our technique is able to find the UIO sequences of all states that the exhaustive method has found. In comparison, the generated tree node counts are roughly same with LANG's method. Only in *beecount*, the proposed method created 2 times more tree nodes with respect to LANG. However in *dk16*, the method manages to find same set of UIO sequences by producing 50% of LANG's result. In addition, the UIO length values of the two FSMs, *bbtas and beecount*, has more values than the experimental results of LANG.

Table 6: The Comparison of Depth First Heuristic Approach with State Based IO Ranking and LANG

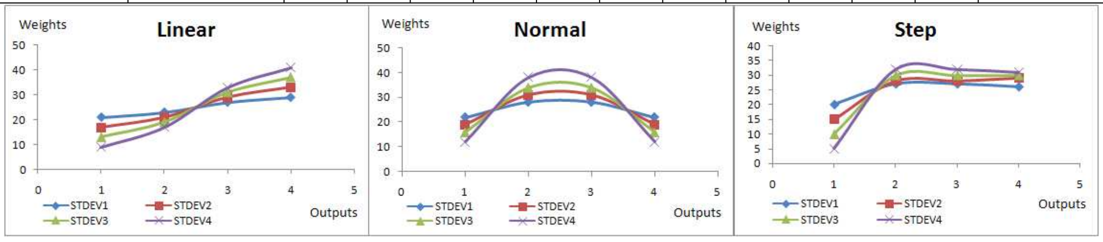
FSM	DFHM with SB IO Ranking		LANG	
	L_{max}	n_g	L_{max}	n_g
bbtas	9	26	8	26
beecount	3	15	1	7
dk14	1	22	1	25
dk15	2	15	2	20
dk16	3	44	3	89
dk17	2	14	2	20
dk27	3	10	3	9
dk512	4	30	4	30
mc	1	5	1	4
shiftreg	3	15	3	16
L_{max}: Maximum length of UIO sequence, n_g: Number of nodes generated.				

6.2 Experimental Results for FSMs

Some of our techniques use the frequency information of the input output labels of the transitions. Therefore, the distribution and the absolute frequency of these labels may have an affect on the performance of these methods. In this section will show the results for FSMs for which the transitions are arranged with different distributions and variances. For this section, we generated 3 classes of the FSMs which have linear distribution, normal distribution and step distribution. In every class, 4 different standard deviation values are considered with 50 FSMs that as seen in Table 6.2. Hence, we will run the aforementioned heuristics on 4,550 FSMs which have 12,350,000 states in total.

Table 7: FSM transition distributions and corresponding standard deviation values.

	Constant (%)	Linear (%)				Normal (%)				Step (%)			
i/o_1	25	21	17	13	9	22	19	16	12	20	15	10	5
i/o_2	25	23	21	19	17	28	31	34	38	27	28	30	32
i/o_3	25	27	29	31	33	28	31	34	38	27	28	30	32
i/o_4	25	29	33	37	41	22	19	16	12	26	29	30	31
STDEV	0	4	7	11	15	3	7	10	15	3	7	10	13



In Figures 39, 41, 43, each distribution is compared with respect to the standart deviation values. For the value $stdev4 \cong 15$, the linear and normal distributions have yielded minimum tree size. The step distribution, this value has resulted in with relatively higher value. Intuitively, we have expected that the tree size value of this standard deviation should be lower in the step distribution. However, we come to conclude that if only one I/O pair occurance is low, the number of states owning the transitions labeled with that I/O pair will decrease. Hence, the probability of encountering a state, in the current set of a tree node, which has a low frequency I/O pair will diminish and the usage of this I/O pair will lessen. For this reason, the tree will generally be constructed with I/O pairs that have relatively higher occurances yielding higher tree size. In Figures 40, 42, 44, the UIO sequence lengths are compared. Again, the standart deviation value $stdev4 \cong 15$ performed better when we look at these UIO lengths in linear and normal distribution. It can be claimed that in these distributions as the standart deviation values increase, the results will be better.

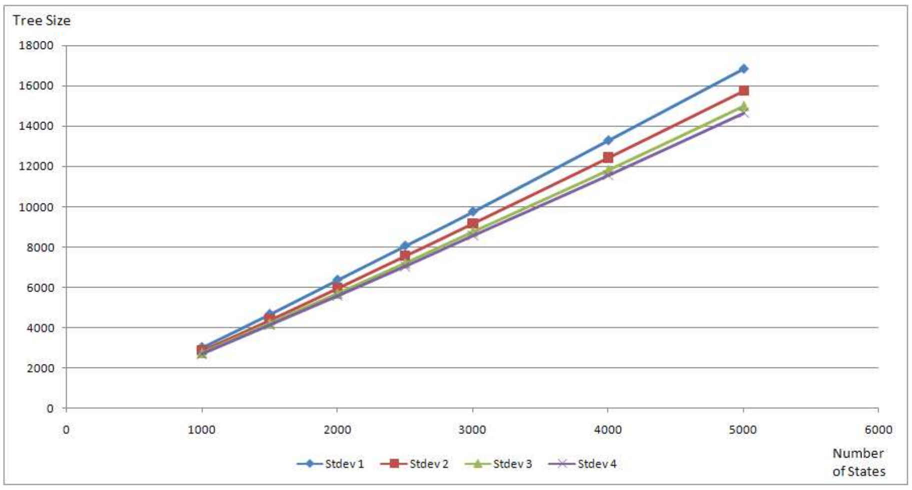


Figure 39: Linear Distribution in Comparison.

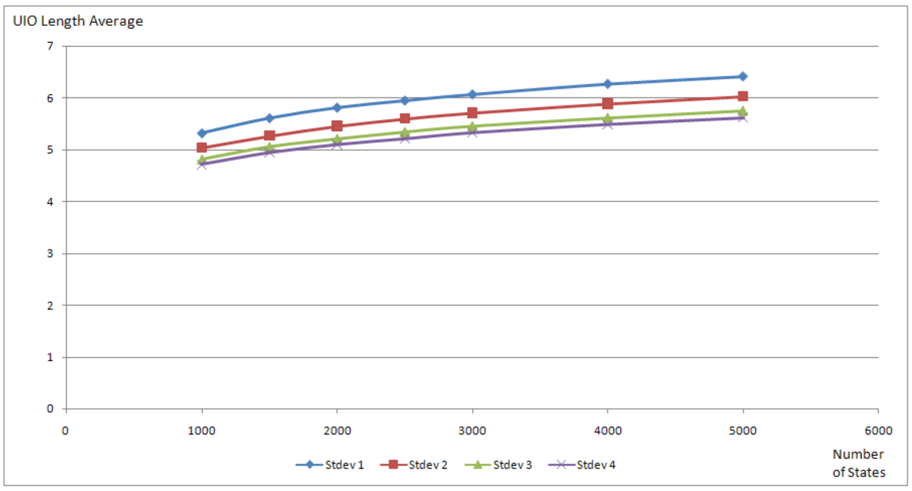


Figure 40: Linear Distribution in Comparison in terms of UIO Sequence Lengths.

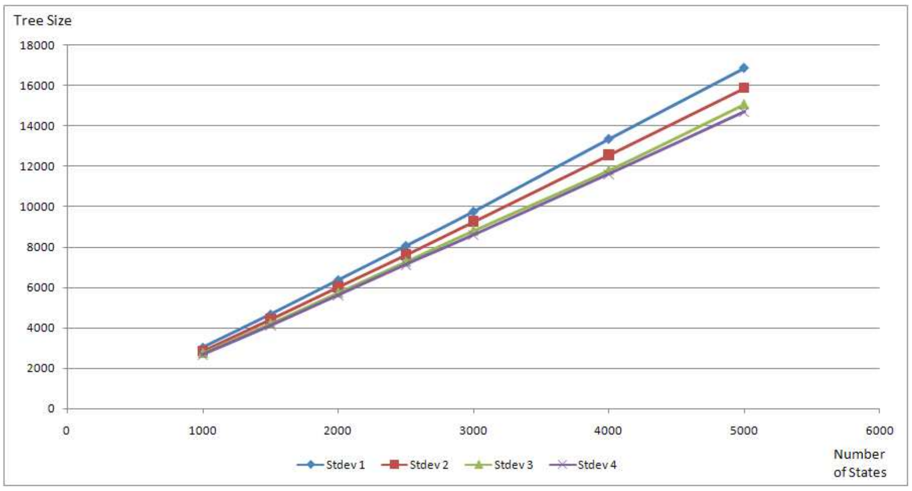


Figure 41: Normal Distribution in Comparison.

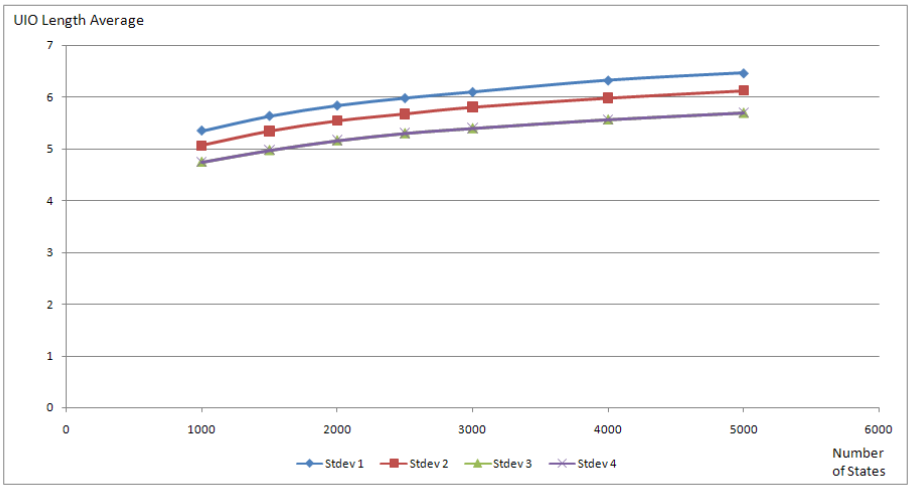


Figure 42: Normal Distribution in Comparison in terms of UIO Sequence Lengths.

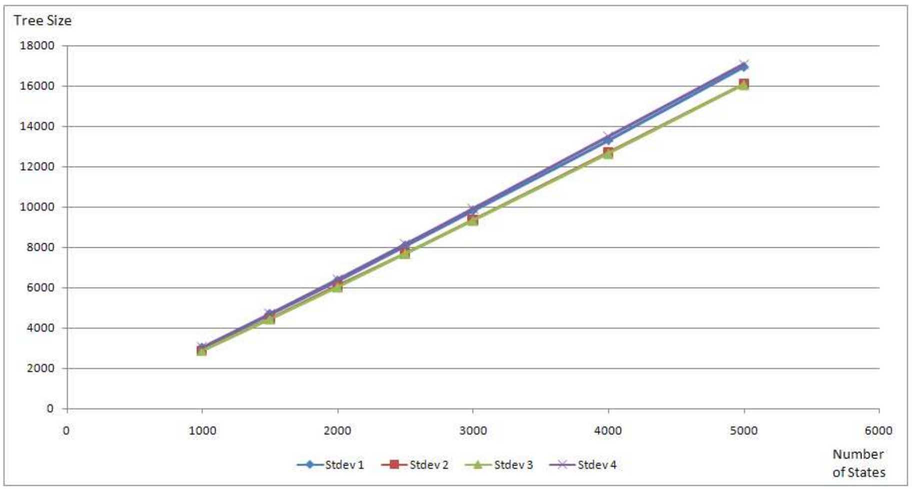


Figure 43: Step Distribution in Comparison.

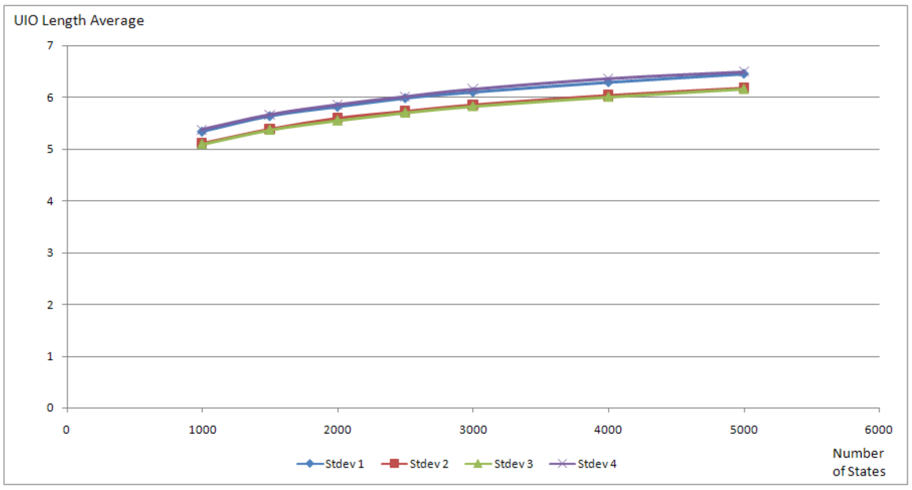


Figure 44: Step Distribution in Comparison in terms of UIO Sequence Lengths.

In Figure 45, 46, 47, 48, each standart deviation value is compared with respect to the different distributions. The values of $stdev1 \cong 3$ and $stdev2 \cong 7$ yielded roughly same tree sizes for all distributions. With standart deviation values of $stdev3 \cong 11$ and $stdev4 \cong 15$ the normal and linear distributions have provided minimum results in the case of the step distribution.

6.3 Experimental Results for Big FSMs

The methods described in this work are heuristics and since the underlying problem is a hard problem, the proposed methods might be expected to blow up at some FSM size. In order to test the performance of the proposed methods, we tried these methods on relatively big FSMs in this section. We have generated FSMs with number of states from 4,000 to 50,000. Each

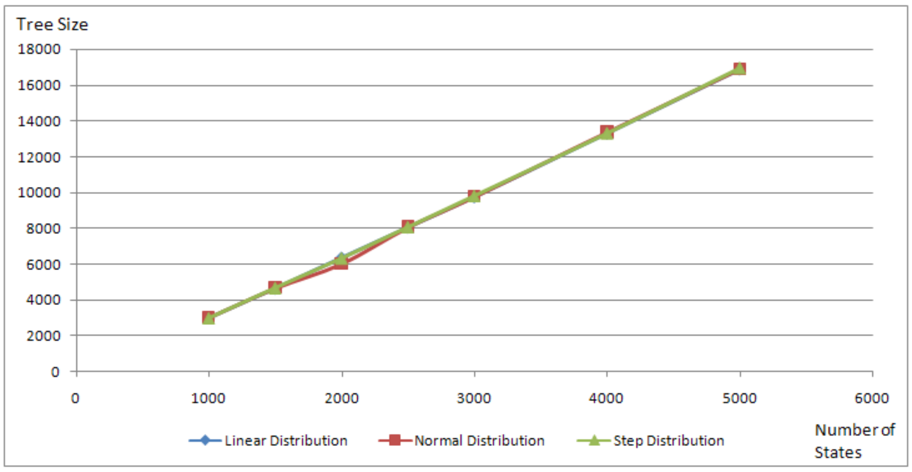


Figure 45: Distributions in Comparison with stdev ≅ 3.

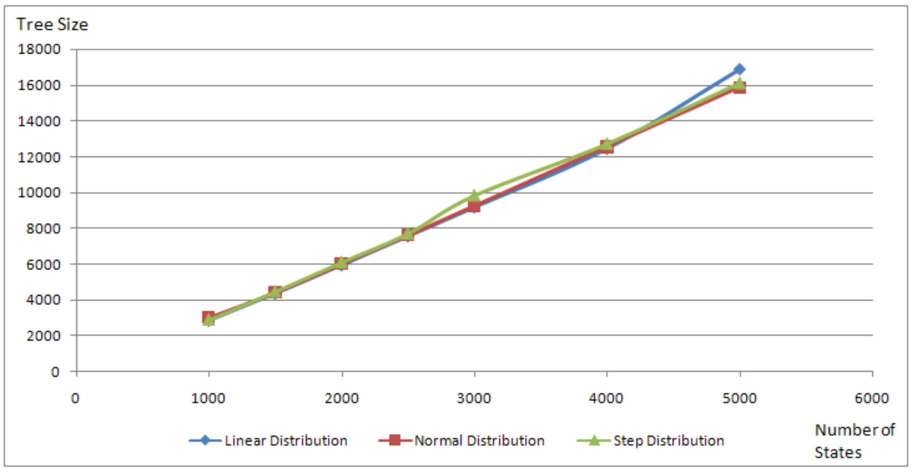


Figure 46: Distributions in Comparison with stdev ≅ 7.

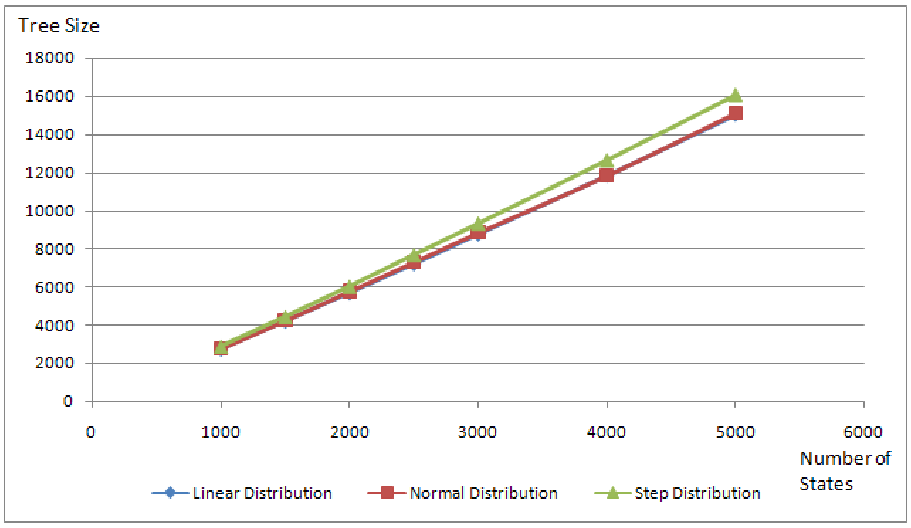


Figure 47: Distributions in Comparison with stdev ≅ 11.

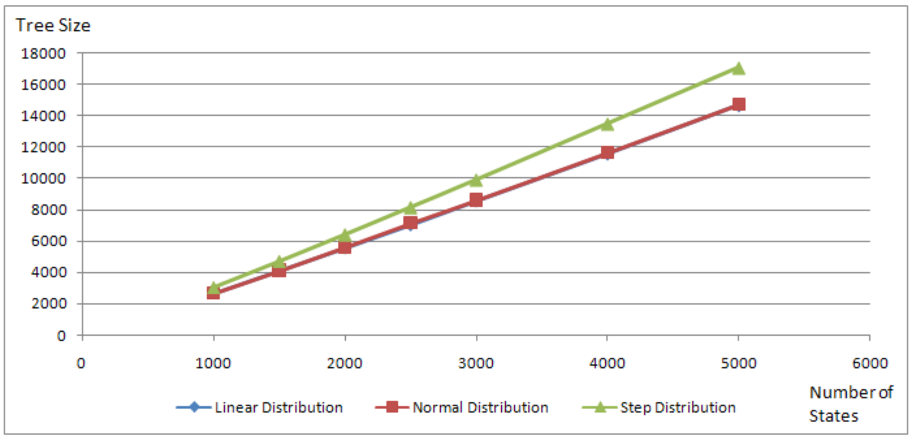


Figure 48: Distributions in Comparison with stdev ≅ 15.

group has 30 FSMs. The tests in this section were run on Intel Xeon 3.00 GHz with 8 GB of RAM.

In Figures 49, 50 and 51, the tree size, average UIO sequence length and timing values can be observed, respectively. The patterns of the methods are similar to their patterns in small FSMs. The State–Based Heuristic Method have reached its limit roughly at 30000 states. Also, the Depth–First Heuristic with Global I/O Method reaches its limit roughly at 45000 states. Here, reaching to its limit basically means that the method cannot handle all the FSMs in that group within the specified memory and time limits.

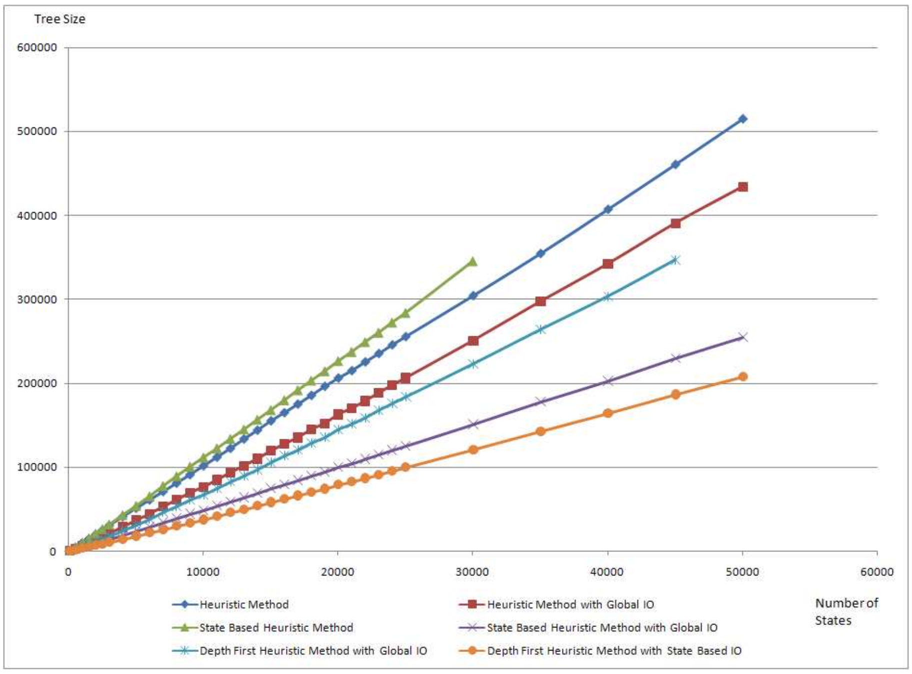


Figure 49: Tree Size Values for Big FSMs

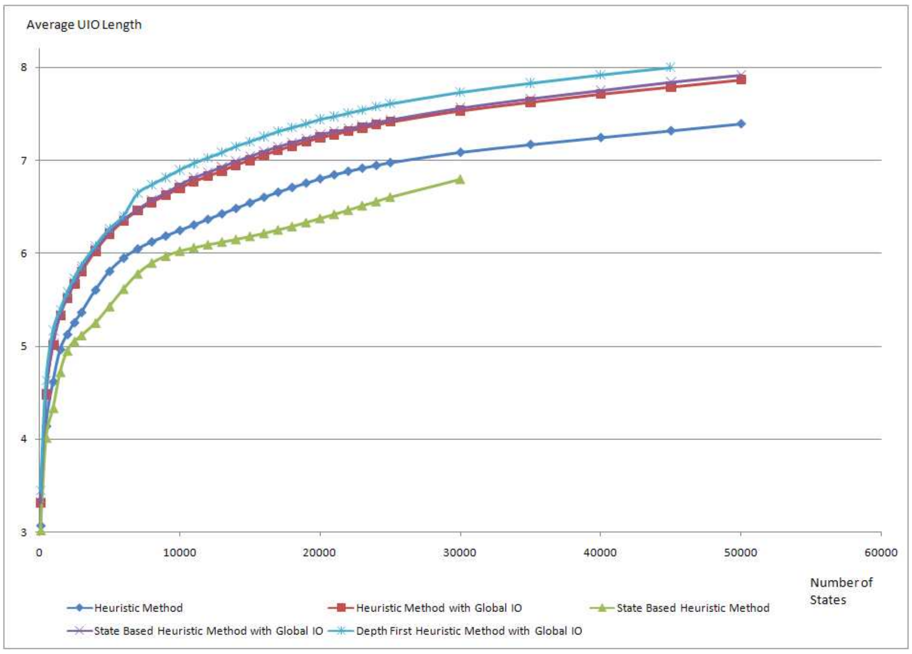


Figure 50: Average UIO Length Values for Big FSMs

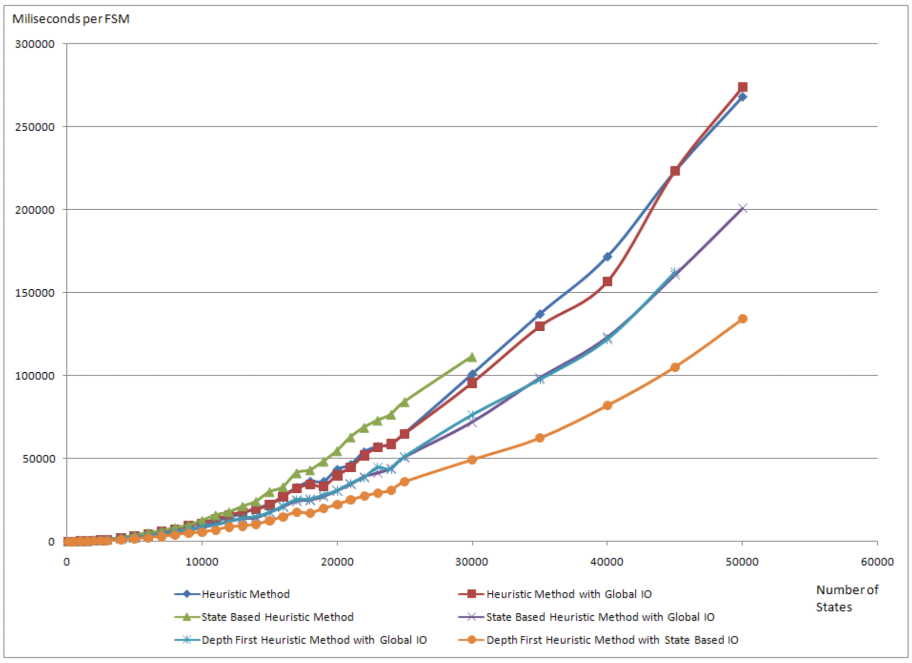


Figure 51: Average Timing Values for Big FSMs

7 Conclusion

In this work, a number of heuristic methods that deal with the UIO sequence generation problem are described. Since the systems from different areas can be modeled as a finite state machine, these techniques can be used in state verification which is a part of conformance testing and fault detection algorithms.

The methods are based on guiding the generation and the search in a UIO tree by using a heuristic information. The basic idea is to try to force the search to goto to the parts of the tree that will reveal UIO sequences of the states for which a UIO sequence has not found yet. An extensive experimental analysis for the suggested methods are also provided.

Each method introduced develops on a previous method. However, it is not easy/correct to say that a method is absolutely better than another method. There are several factors such the memory and the time that is used to generate the UIO sequences and the length of the UIO sequences. In general, the memory and the time requirements of the methods are directly correlated. However, the length of the UIO sequences found is inversely correlated to the time and the memory requirements. Therefore, a method finding shorter UIO sequences will be using more time and memory.

Rather than concluding a method to be superior to the others, the set of methods presented in this work should be considered as providing a spectrum of methods from which one can select based on her/his requirements (low UIO sequence length vs. low computational resources) for finding UIO sequences.

Another contribution of this work is the introduction of a trade off parameter between the UIO sequence length and the memory requirements for using the inference mechanism introduced by Naik in [20]. Naik's method uses very low memory but it has been criticized for generating very long UIO sequences. By using the limited inference approach introduced here, one can control the length of the UIO sequences that will be generated by Naik's approach at the expense of increased memory requirement.

References

[1] D. Lee A.V. Aho, A. T. Dahbura and M.U. Uyar. An optimization technique for protocol conformance test generation based on uio sequences and rural chinese postman tours. *IEEE Transactions on Communications*, pages 1604–1615, 1991.

[2] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng.*, pages 178–187, 1978.

[3] M.S. Chiang C.M. Huang and M.Y. Jiang. Uio: a protocol test sequence generation method using the transition executability analysis (tea). *Comput. Commun.*, 21:14621475, 1998.

[4] A. D. Friedman and P. R. Menon. *Fault Detection in Digifal Circuits.* NJ: Prentice-Hall, 1971.

[5] A. Gill. State-identification experiments in finite automata. *lnform. and Contr.*, 4:132–154, 1961.

[6] A. Gill. *Introduction to the Theory of Finite-State Machines.* New York: McGraw-Hill, 1962.

[7] S. M. Gohershtein. Check words for the states of a finite automaton. *Kibernetika*, page 4649, 1974.

[8] F. C. Hennie. Fault detecting experiments for sequential circuits. *Proc. 5th Ann. Symp. Switching Circuit Theory and Logical Design*, pages 95–110, 1964.

[9] R.M. Hierons and H. Ural. Uio sequence based checking sequences for distributed test architectures. *Inf. Softw. Technol.*, page 793803, 2003.

[10] Rob M. Hierons and Hasan Ural. Optimizing the length of checking sequences. *IEEE Trans. Comput.*, 55(5):618–629, 2006.

[11] E. P. Hsieh. Checking experiments for sequential machines. *IEEE Trans. Comput.*, pages 1152–1166, 1971.

[12] F.M. Ali I. Ahmad and A.S. Das. Lang - algorithm for constructing unique input/output sequences in finite-state machines. *IEE Proceedings - Computers and Digital Techniques*, pages 131– 140, 2004.

[13] M. Harman K. Derderian, R. M. Hierons and Q. Guo. Automated unique input output sequence generation for conformance testing of fsms. *The Computer Journal*, 2006.

[14] Z. Kohavi. *Switching and Finite Automata Theory.* New York: McGraw-Hill, 1978.

[15] P. K. Lala. *Fault Tolerant and Fault Testable Hardware Design.* NJ: Prentice-Hall, 1985.

[16] D. Lee and M. Yannakakis. Testing finite state machines: state identification and verification. *IEEE Trans. Comput.*, pages 306–320, 1994.

[17] D. Lee and M. Yannakakis. Principles and methods of testing fsms - a survey. *Proceedings of the IEEE*, 84:1090–1123, 1996.

[18] R. E. Miller and S. Paul. On the generation of minimal-length conformance tests for communication protocols. *IEEE/ACM Trans. Networking*, pages 284–289, 1993.

[19] R.E. Miller and S. Paul. On the generation of minimal-length conformance tests for communication protocols. *IEEE/ACM Trans. Netw.*, pages 116–129, 1993.

[20] K. Naik. Efficient computation of unique input/output sequences in finite-state machines. *IEEE/ACM Transactions on Networking*, 1997.

[21] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transition tours. *Proc. Ilth IEEE Fault Tolerant Comput. Symp.*, pages 238–243, 1981.

[22] I. Pomeranz and S.M. Reddy. Functional test generation for full scan circuits. *Proc. Conf. on Design, Automation and Test in Europe*, pages 396–403, 2000.

[23] M. Harman Q. Guo, R. M. Hierons and K. Derderian. Computing unique input/output sequences using genetic algorithms. In *FATES*, pages 164–177, 2003.

[24] M. Harman Q. Guo, R. M. Hierons and K. Derderian. Constructing multiple unique input/output sequences using metaheuristic optimisation techniques. *IEE Proceedings - Software*, pages 127– 140, 2005.

[25] K. K. Sabnani and A. T. Dahbura. A new technique for generating protocol tests. *Proc. 9th Data Commun. Symp.*, pages 36–43, 1985.

[26] K. K. Sabnani and A. T. Dahbura. A protocol test generation procedure. *Computer Networks and ISDN Syst.*, pages 285–297, 1988.

[27] B. Sarikaya and G. v. Bochmann. Synchronization and specification issues in protocol testing. *IEEE Trans. Commun.*, pages 389–395, 1984.

[28] D. Sidhu and T. Leung. Fault coverage of protocol test methods. *Proc. IEEE INFOCOMSS*, pages 80–85, 1988.

[29] D.P. Sidhu and T.K. Leung. Formal methods for protocol testing: a detailed study. *IEEE Trans. Softw. Eng.*, pages 413–426, 1989.

[30] Dechang Sun, Bapiraju Vinnakota, and Wanli Jiang. Fast state verification. In *DAC*, pages 619–624, 1998.

[31] R. L. Rivest T. H. Cormen, C. E. Leiserson and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 2001.

[32] H. Ural, X. Wu, and F. Zhang. On minimizing the lengths of checking sequences. *IEEE Transactions on Computers*, 46(1):93–99, 1997.

[33] M. U. Uyar and A. T. Dahbura. Optimal test sequence generation for protocols: The chinese postman algorithm applied to q.931. *Proc. IEEE Global Telecommun. Conf*, pages 68–72, 1986.

[34] G. v. Bochmann and C. A. Sunshine. A survey of formal methods. *Computer Networks and Protocols*, pages 561–578, 1983.

[35] S. T. Vuong W. Y. L. Chan and M. R. Ito. An improved protocol test generation procedure based on uios. *Proc. SIGCOM*, pages 283–294, 1989.

[36] B. Wang and D. Hutchinson. Protocol testing techniques. *Comput. Commun.*, pages 79–87, 1987.

[37] M. C. Yalcin. Distinguishing sequence based checking sequence generation implementation and improvements. Master's thesis, Sabanci University, 2006.

[38] B. Yang and H. Ural. Protocol conformance test generation using multiple uio sequences with overlapping. *ACM SIGCOMM: Communications, Architectures, and Protocols, Twente, The Netherlands, North-Holland, The Netherlands.*, pages 118–125, 1990.

[39] S. Yang. Logic synthesis and optimization benchmarks user guide, version 3.0. *Technical report, MCNC, North Carolina*, 1991.

[40] F. Lombardi Y.N. Shen and A.T. Dahbura. Protocol conformance testing using multiple uio sequences. *IEEE Trans. Commun.*, pages 1282–1287, 1992.

A Appendix

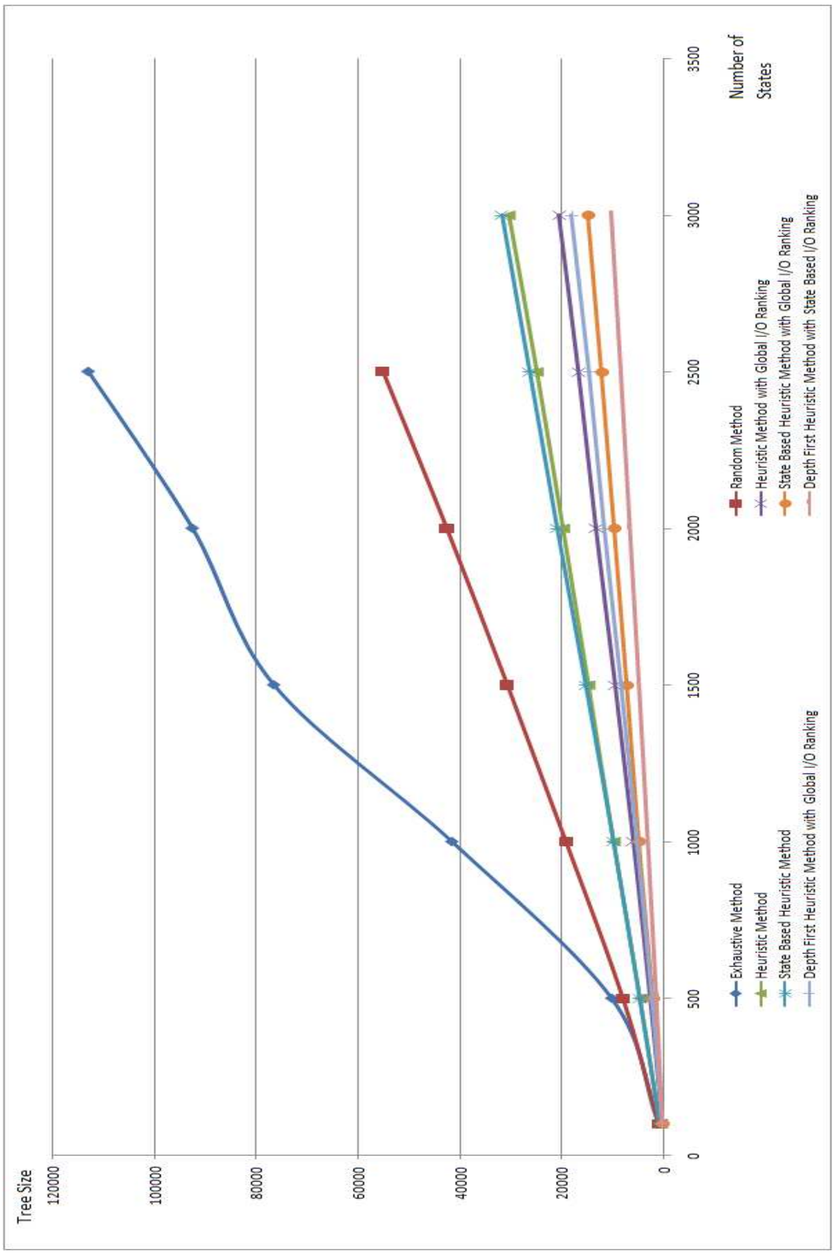


Figure 52: Depth–First Heuristic Method with State–Based I/O Ranking in Comparison.

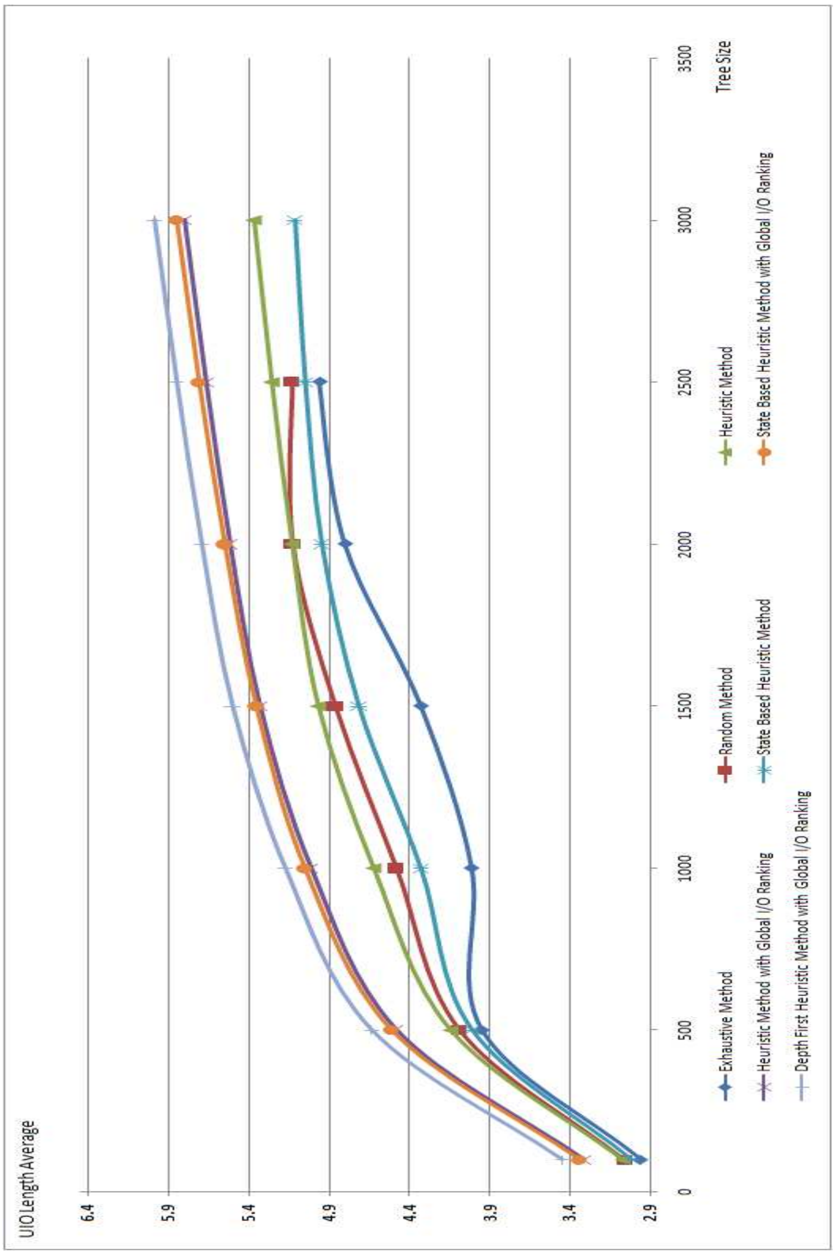


Figure 53: Depth–First Heuristic Method with Global I/O Ranking in Comparison in terms of UIO Sequence Lengths.

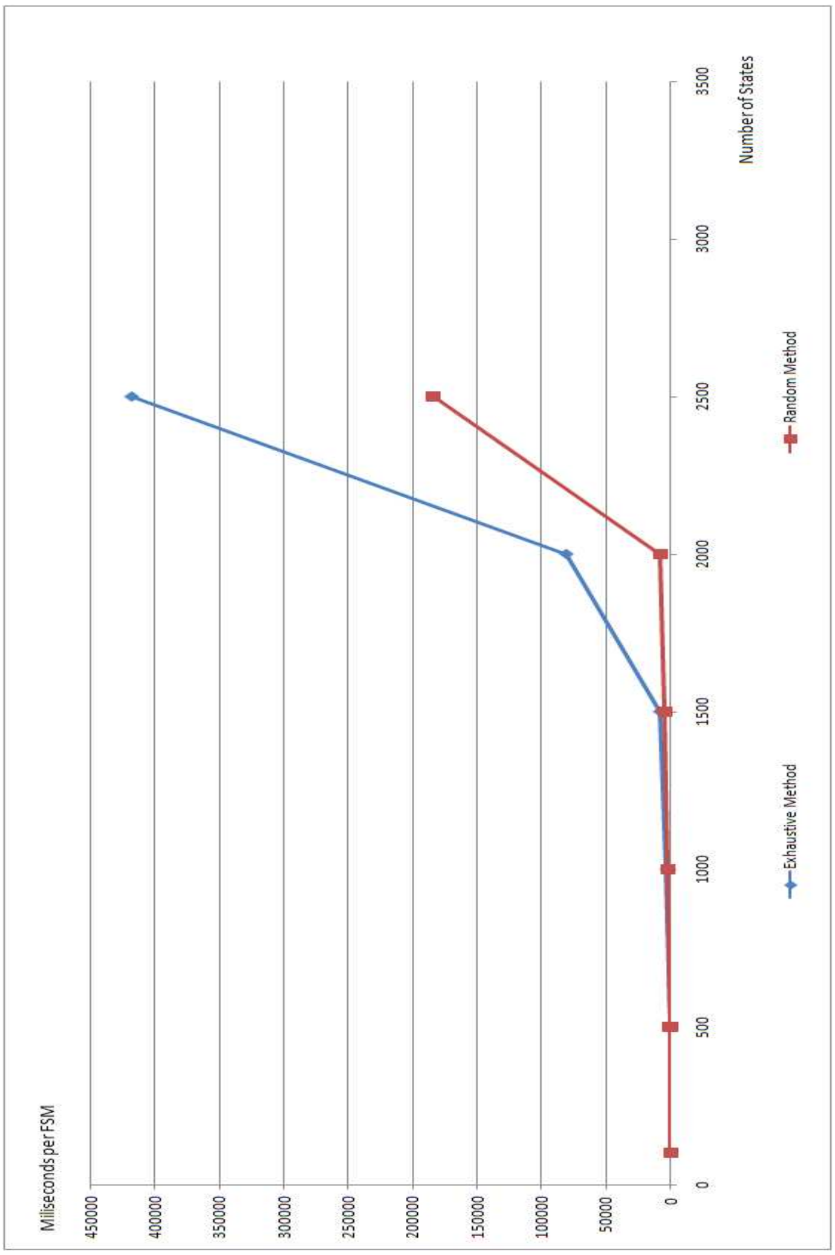


Figure 54: Time Performances of the Exhaustive and the Random Methods

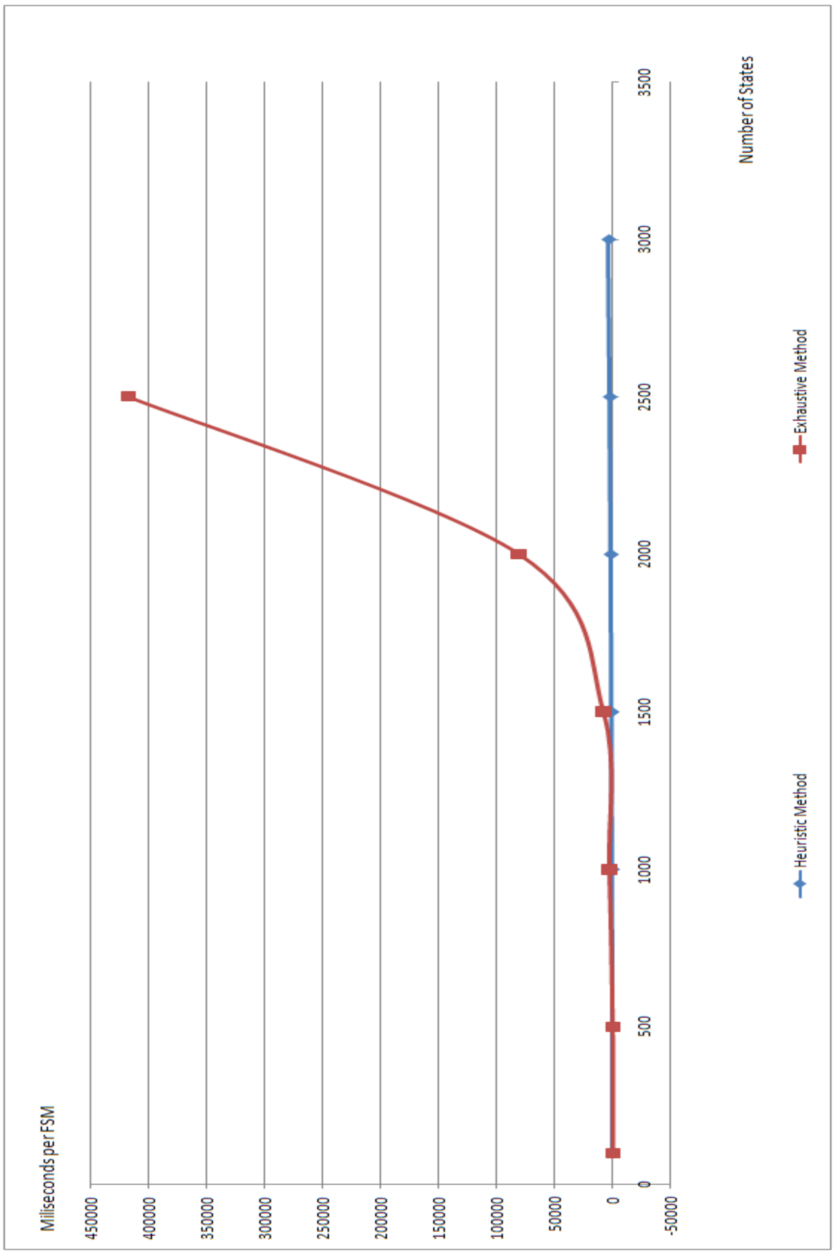


Figure 55: Heuristic Method Time Requirements in Comparison with Exhaustive Method.

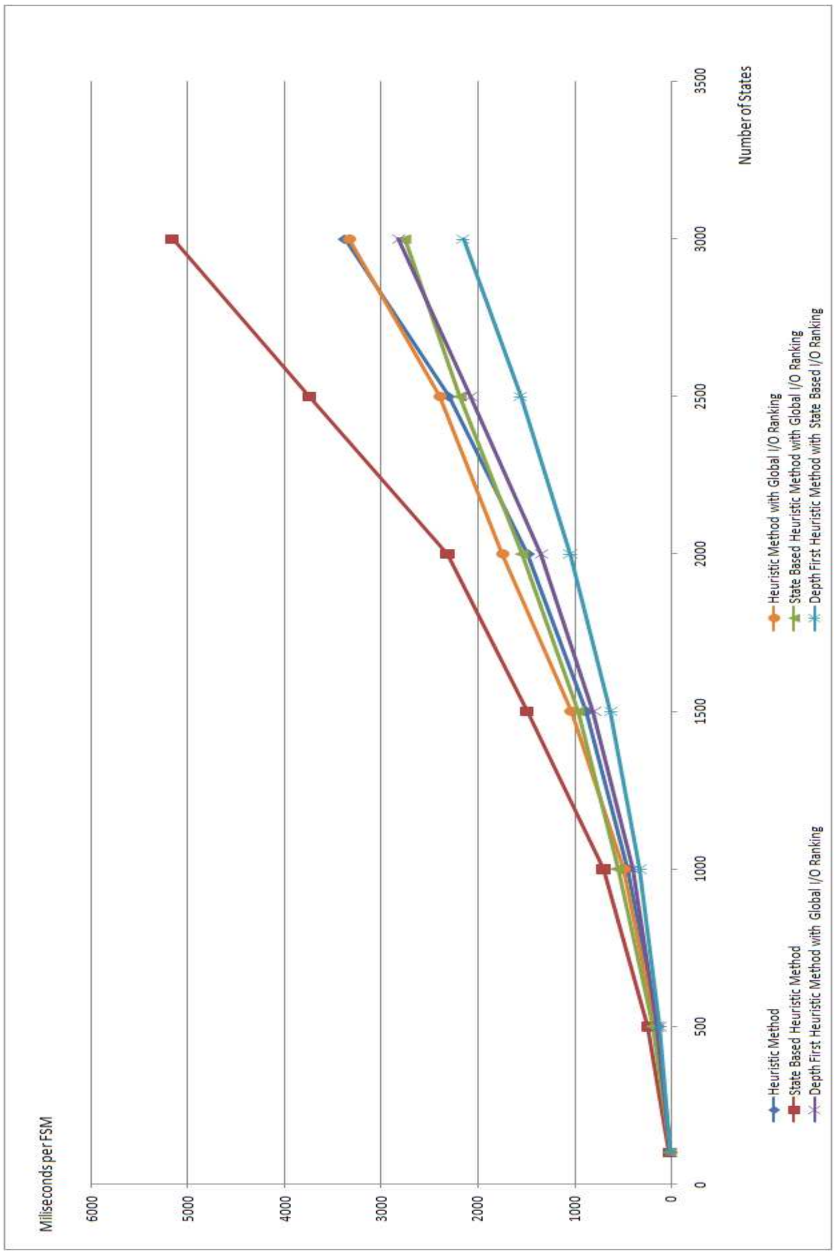


Figure 56: Time Comparison of Depth First Heuristic Method with Global I/O Ranking.

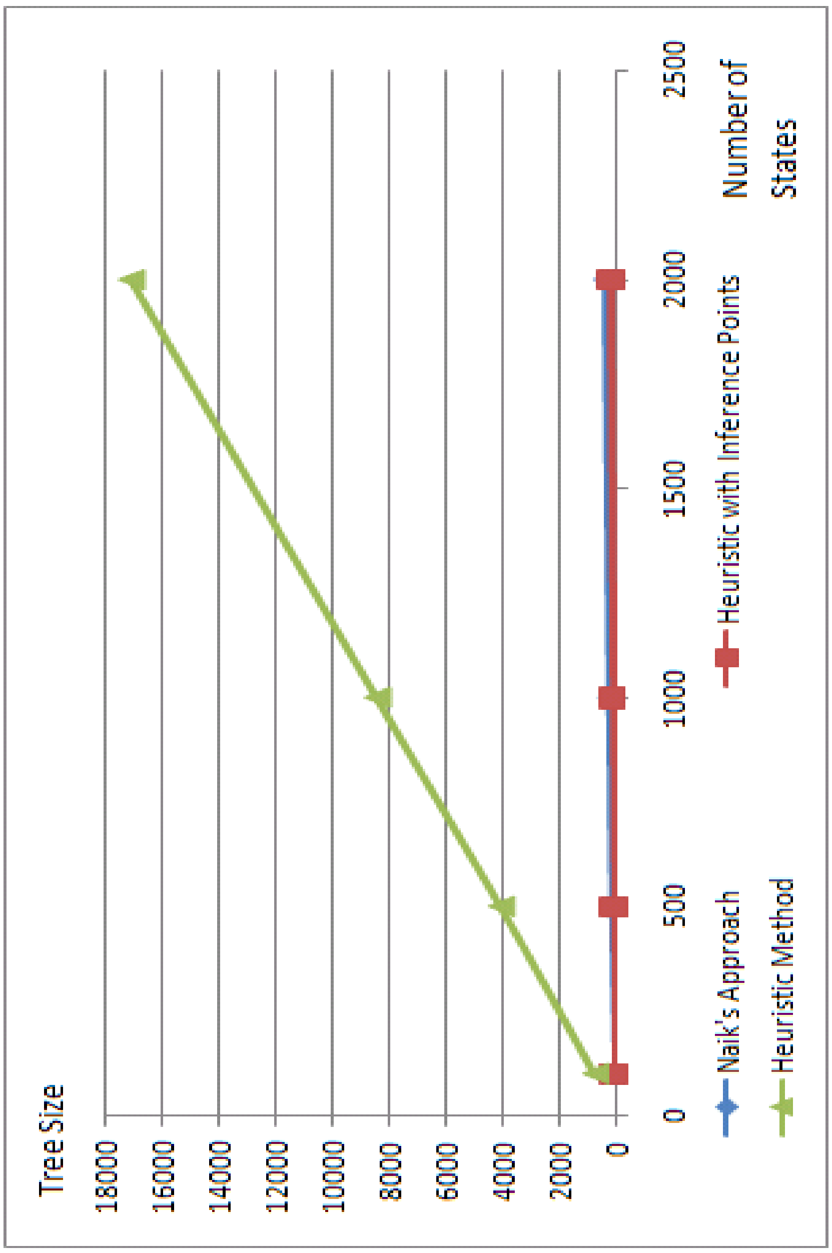


Figure 57: The Tree Size Values of Heuristic Method, Heuristic Method with Inference and Naik's Method.

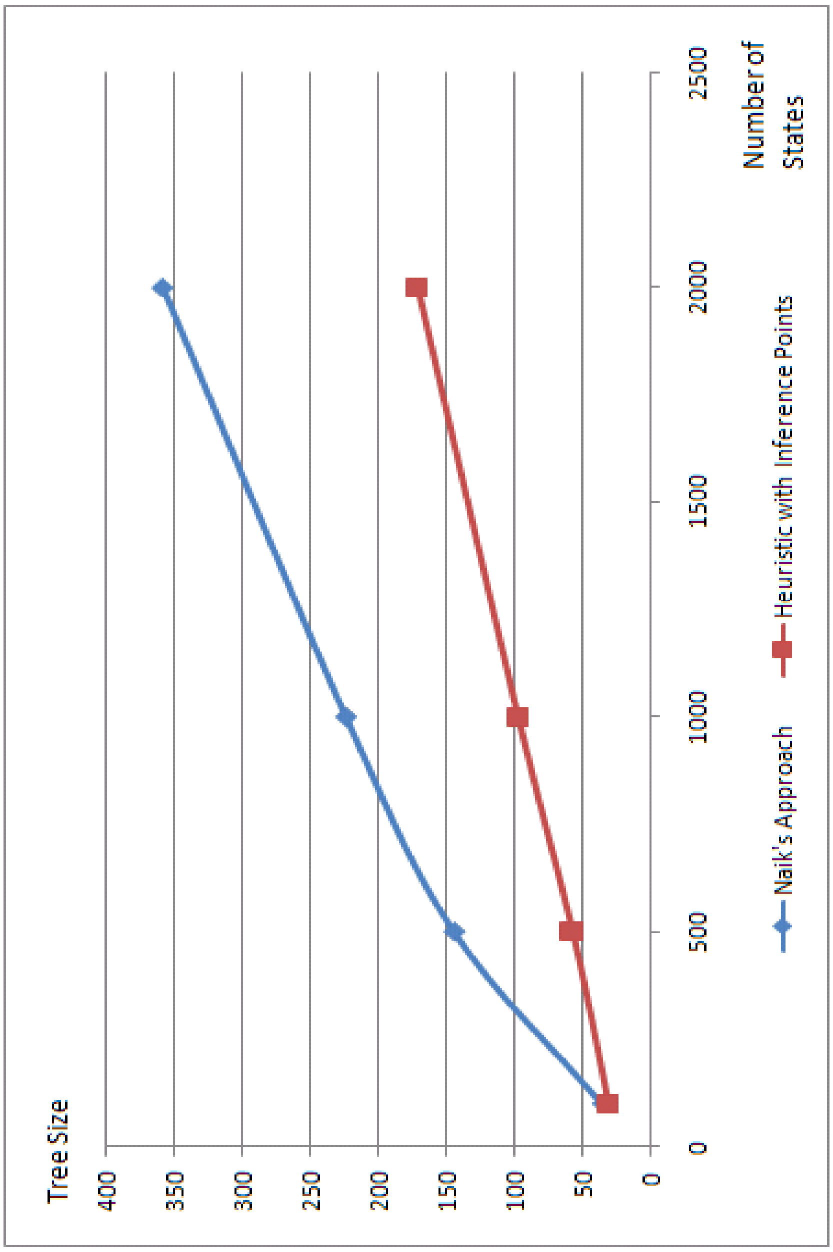


Figure 58: The Tree Size Values of Heuristic Method with Inference and Naik's Method.

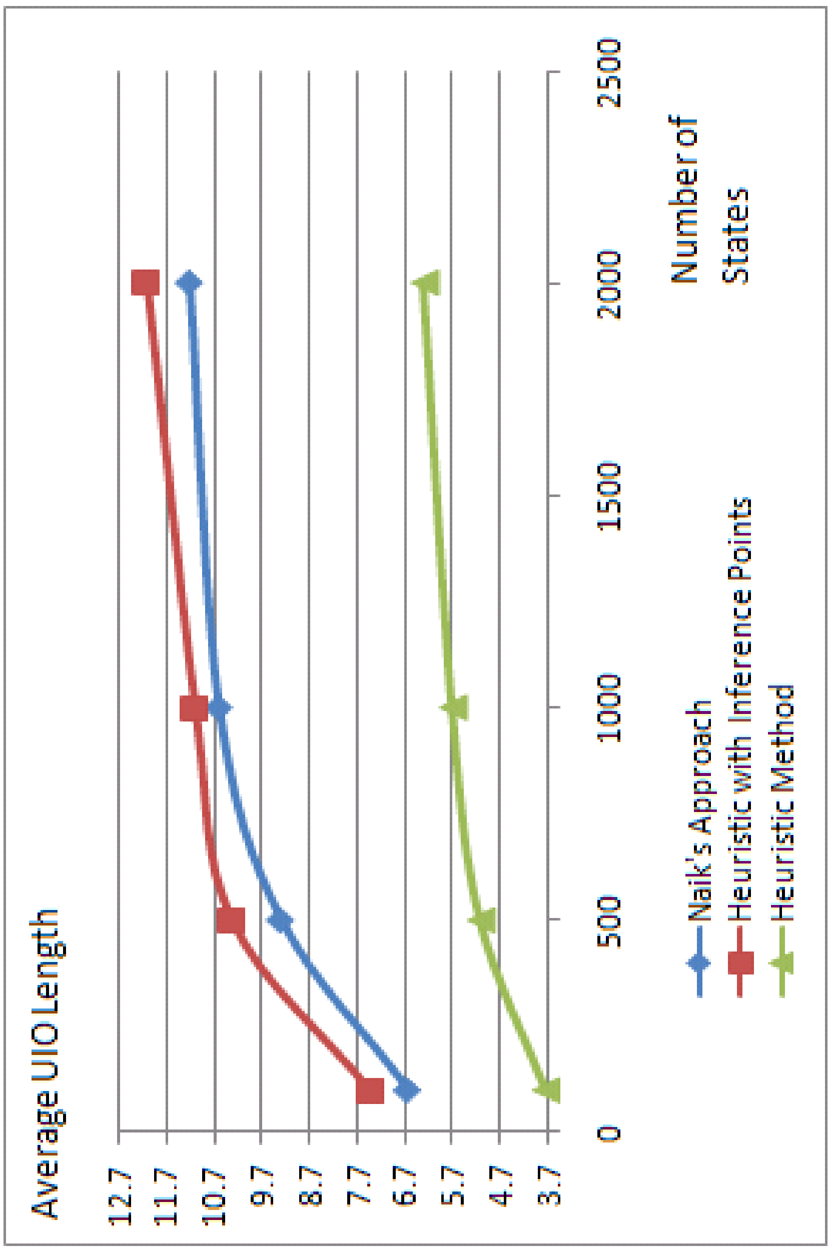


Figure 59: The UIO Sequence Lengths of Heuristic Method, Heuristic Method with Inference and Naik's Method.

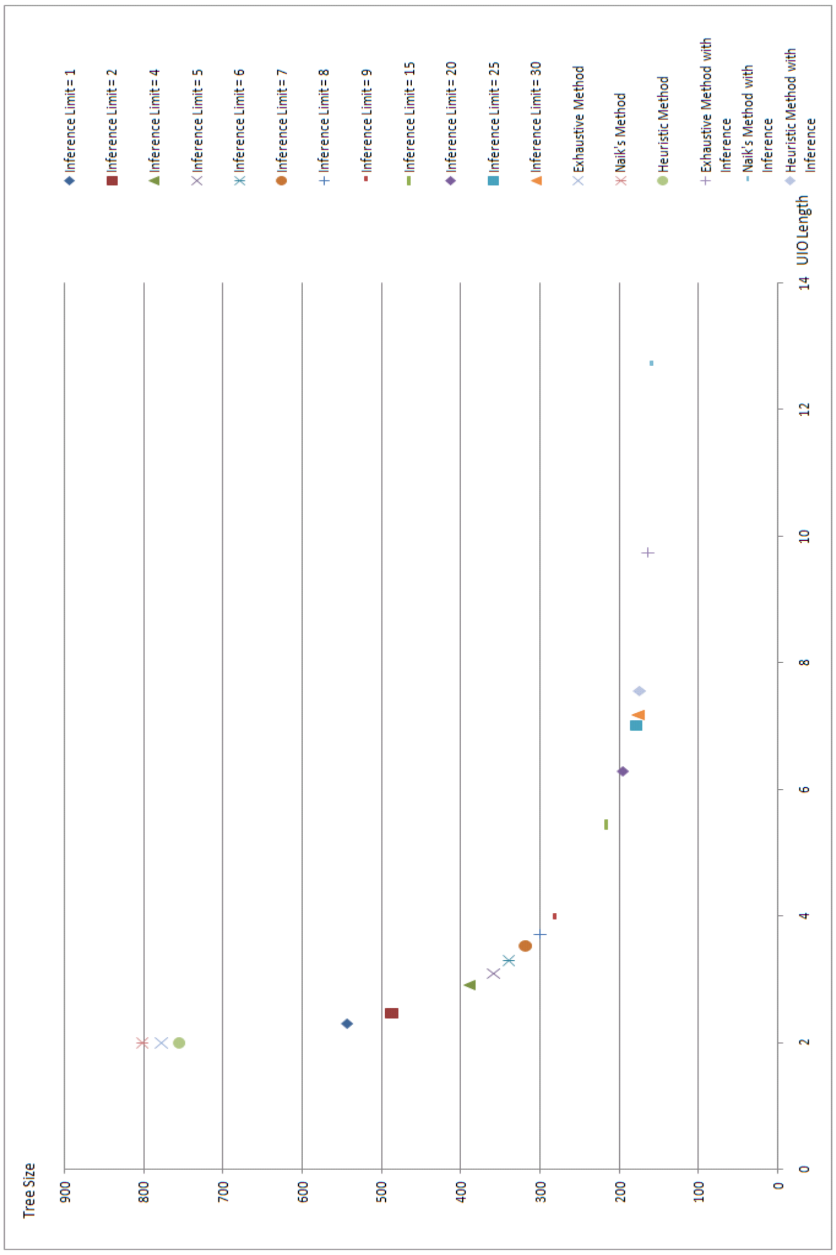


Figure 60: Different Inference Lengths in Comparison.

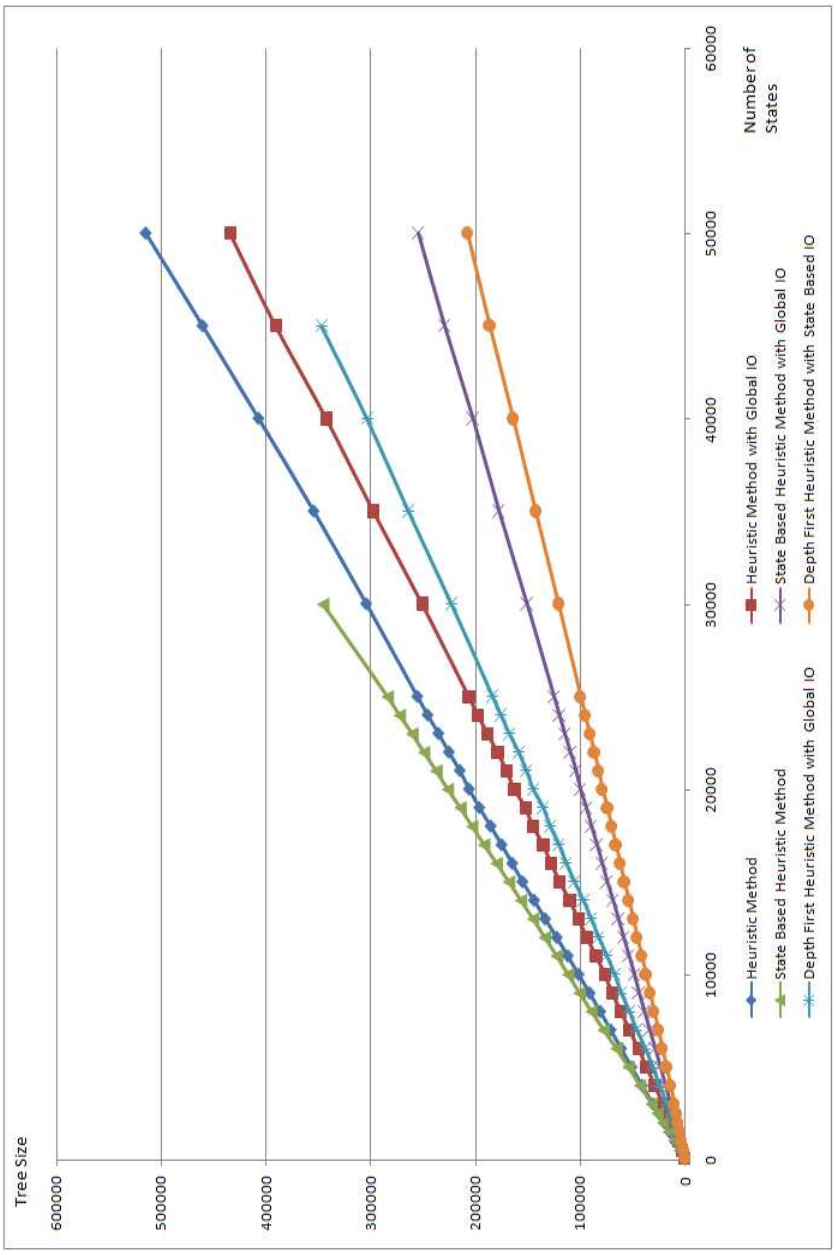


Figure 61: Tree Size Values for Big FSMs

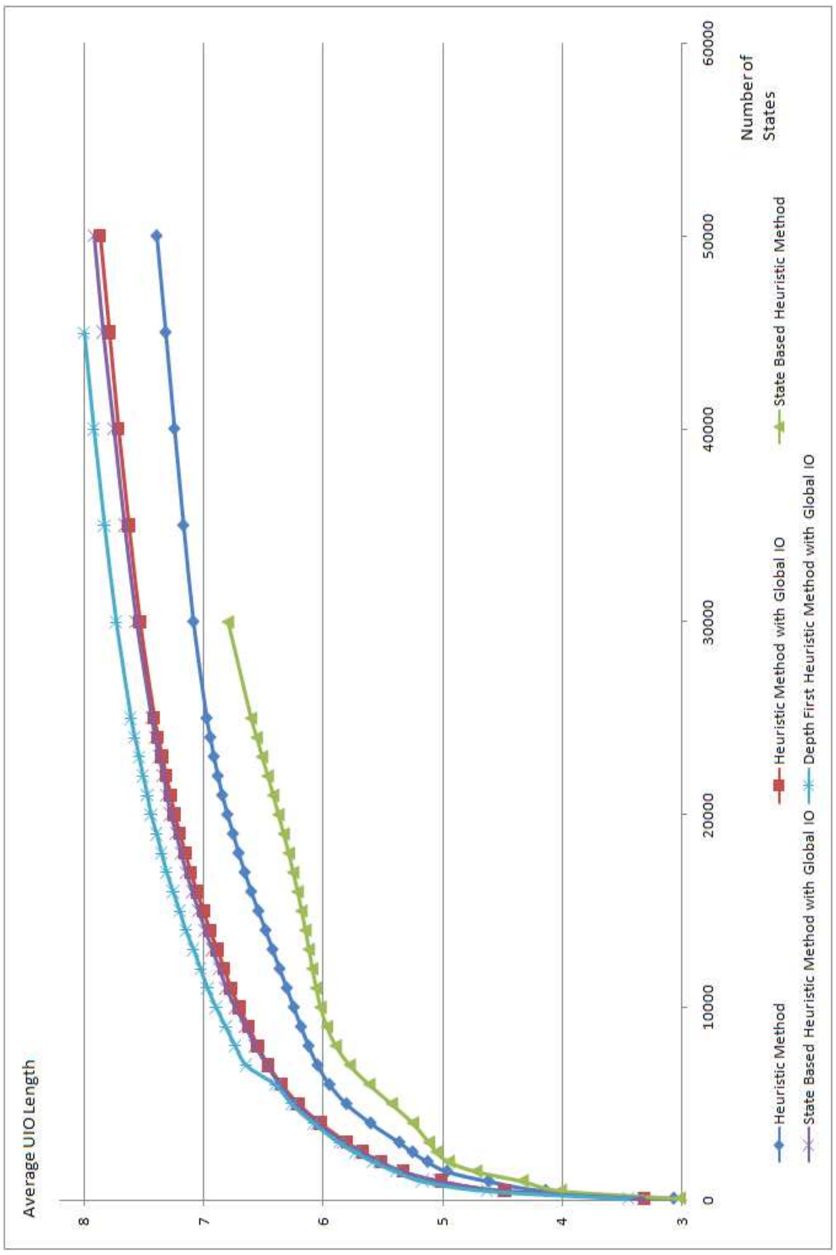


Figure 62: Average UIO Length Values for Big FSMs

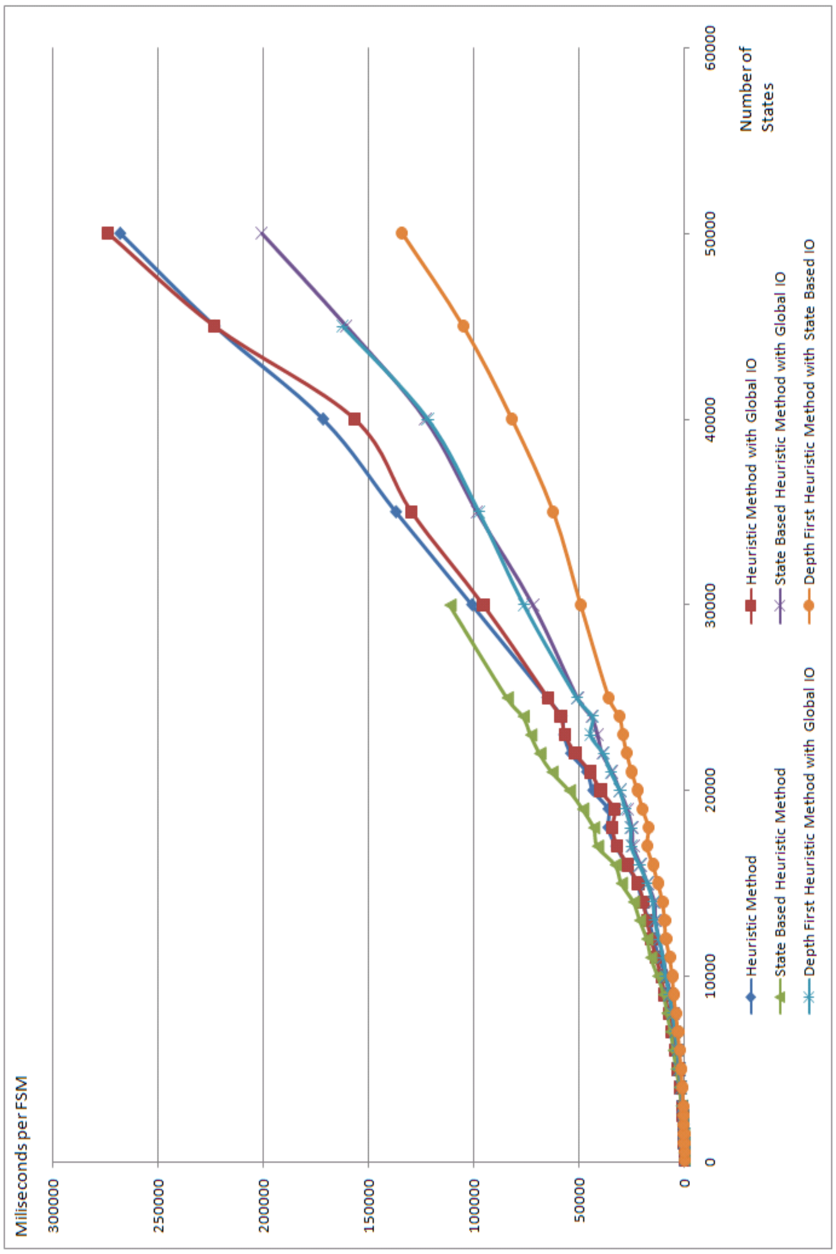


Figure 63: Average Timing Values for Big FSMs