# IMPROVED HEURISTICS FOR W–SET AND K–TREE GENERATION ON FINITE STATE MACHINES

by

KAMİL TOLGA ATAM

Submitted to the Graduate School of Engineering and Natural Sciences in partial fulfilment of the requirements for the degree of Master of Science

Sabancı University

December 2020

# IMPROVED HEURISTICS FOR W-SET AND K-TREE GENERATION ON FINITE STATE MACHINES

Approved by:

Date of Approval: December 23, 2020

## ABSTRACT

IMPROVED HEURISTICS FOR W–SET AND K–TREE GENERATION ON FINITE STATE MACHINES

KAMİL TOLGA ATAM

COMPUTER SCIENCE AND ENGINEERING M.Sc. THESIS, DECEMBER 2020

Thesis Supervisor: Assoc. Prof. Hüsnü Yenigün

Keywords: Finite State Machines, State Identification Sequences, W-sets, K-sets, K-trees

Finite State Machine (FSM) based testing methods utilize State Identification Sequences which are used to identify the states of a black box implementation as corresponding to the states of an FSM given as the specification. There are different types of state identification sequences. Some of these state identification sequences are not guaranteed to exist for all specifications. There is one particular type of state identification sequences, W-set based state identification sequences, which are known to exist for any minimal, deterministic, completely specified FSM. Although W-set based state identification sequences are known for a very long time, most of the works in FSM based testing literature do not prefer to use them when testing for an implementation without a reliable reset feature, since the length of W-set based state identifications sequences in this case are exponential in the cardinality of the W-sets. There are some recent works that suggest reducing the length of the W-set based state identification sequences. In fact, instead of W-sets, which are sets of preset experiments, these new methods can make use of so-called K-sets, which are set of adaptive experiments that again always exist. Furthermore, these new methods suggest applying not all elements of W–sets/K-sets, but instead an adaptive structure, called a K-tree is used to orchestrate the application of the elements of the K-set. However, there are no extensive experimental studies for these new methods. In addition, no algorithms are given for the construction of K-trees. In this work, we first present some W-set construction algorithms to construct better

W-sets, in terms of both the cardinality and the total length of the sequences. We compare our W-set algorithms experimentally to the algorithms that exist in the literature. We also present algorithms to constructs K-sets and K-trees. Finally, we present an extensive experimental study for state identification sequences. The results show that, although W-set based state identification sequences have been considered practically infeasible due to the exponentially long sequences, the usage of K-trees make state identification sequences very short and practically usable. Utilizing K–sets in the generation of K–trees also yields better results than utilizing W–sets.

## ÖZET

SONLU DURUM MAKİNELERİNDE W–KÜMESİ VE K–AĞACI TÜRETİMİ İÇİN SEZGİSEL ALGORİTMALARIN İYİLEŞTİRİLMESİ

KAMİL TOLGA ATAM

BİLGİSAYAR BİLİMİ VE MÜHENDİSLİĞİ YÜKSEK LİSANS TEZİ, ARALIK 2020

Tez Danışmanı: Doç. Dr. Hüsnü Yenigün

Anahtar Kelimeler: Sonlu Durum Makineleri, Durum Saptama Dizileri, W–kümeleri, K–kümeleri, K–ağaçları

Sonlu Durum Makinesi (FSM) bazlı test yöntemleri, Durum Saptama Dizileri adı verilen, kara-kutu olarak verilen bir makinedeki durumların tasarımdaki durumlarla örtüşüp örtüşmediğini saptamaya yarayan dizileri kullanır. Bilinen farklı durum saptama dizisi çeşitleri vardır. Bu durum saptama dizilerinden bazılarının her tasarım için var olacağı kesin değildir. Ancak, bir durum saptama dizisi çeşidinin - W–kümesi bazlı durum saptama dizileri - indirgenmiş, gerekirci, ve tam belirtimlenmiş tüm sonlu durum makineleri için her zaman bulunduğu bilinmektedir. W–kümesi bazlı durum saptama dizileri uzun süredir bilinmesine karşın, literatürdeki birçok güvenli tekrar başlatma özelliği olmayan makineler için geliştirilen yöntemler tarafından tercih edilmemektedir. Bunun sebebi, W–kümesi bazlı yöntemlerin ürettiği dizilerin uzunluğunun W–kümesindeki eleman sayısına göre üstel olarak artmasıdır. Bazı güncel çalışmalar, W–kümesi bazlı durum saptama dizilerinin uzunluklarının düşürülebileceğini önermektedirler. Aslında bu yöntemler, önayarlı deneylerden oluşan W–kümeleri yerine, uyarlanabilir deneylerden oluşan K–kümelerini de kullanabilmektedirler. K–kümeleri de W–kümeleri gibi her indirgenmiş, gerekirci ve tam belirtimli tüm sonlu durum makineleri için bulunmaktadırlar. Bundan da öte, bu yeni yöntemler W–kümesinin/K–kümesinin tüm elemanları kullanılmadan da bir durum saptama dizisi üretilebileceğini öne sürmektedirler. Bu yöntemlerde, sonlu durum makinesinin bir durumu için hangi W–kümesi/K–kümesi elemanlarının kullanılacağını saptamak adına, uyarlanabilir bir yapı olan K–ağaçları kullanılır. Fakat, literatürde bu yeni yöntemlerle alakalı bir deneyli çalışma henüz yapılmamıştır. Buna ek olarak, K–ağaçlarının nasıl üretileceği ile alakalı bir algoritma da verilmemiştir. Bu çalışmada, öncelikle daha iyi W–kümeleri (hem eleman sayısı hem de toplam uzunluk bakımından) oluşturulması için W–kümesi oluşturma algoritmaları sunulmaktadır. Bu W–kümesi algoritmaları literatürdeki diğer algoritmalarla deneysel olarak karşılaştırılmaktadır. Aynı zamanda bu çalışmada, K–kümesi ve K–ağacı üretimi için algoritmalar da önerilmektedir. Son olarak, durum saptama dizileri ile ilgili kapsamlı bir deneysel çalışma sunulmaktadır. Bu deneysel çalışmalar, W–kümesi bazlı durum saptama dizileri tarihsel olarak pratikte kullanışsız gözükse de, K–ağacı yardımıyla oluşturulduğunda, bu durum saptama dizilerinin çok kısa ve kullanışlı sonuçlar ürettiklerini göstermektedir. Ayrıca, K–ağacı üretilirken K–kümelerinin kullanılması da W–kümelerine göre bir avantaj sağlamaktadır.

## DEDICATION

*To my family who pushed me to achieve the best and pointed the highest as target,*

*To the most supportive and kind-hearted teacher who gave me much freedom in choosing what and when to learn,*

*To my dearest friends who made this university worthwhile and memorable,*

*And to Buse who made this thesis possible by not letting me give up*

## TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

## LIST OF ABBREVIATONS

**ADS** Adaptive Distinguishing Sequence .................... 1, 2, 6, 7, 31, 63, 64

**FSM** Finite State Machine iv, vi, xii, 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 27, 28, 29, 30, 35, 40, 43, 44, 59, 60, 61, 62, 63, 64

**IADS** Incomplete Adaptive Distinguishing Sequence .......................... 7

**PADS** Partial Adaptive Distinguishing Sequence .. ix, xii, 7, 8, 14, 28, 29, 30, 33, 34, 52, 62

**PDS** Preset Distinguishing Sequence ............................... 1, 2, 63, 64

**SIS** State Identification Sequence . xiii, 1, 2, 3, 6, 12, 13, 14, 25, 26, 28, 31, 32, 33, 43, 44, 46, 48, 52, 55, 56, 57, 58, 59, 60, 63, 64

# 1. INTRODUCTION

Software testing is crucial, as it can greatly enhance the reliability of a system and reduce the development and maintaining cost of a software project. Finite state machine (FSM) models have been widely used as a mechanism to reflect the abstract behaviour of a software system. In recent years, even several web interactive frameworks (such as React.JS and Redux) have used the state machine approach (Banks & Porcello, 2017). In this technique, what user sees in a web page is defined to be a reflection of the state of the web page software. With such common usage of finite state machines, FSM-based testing approaches have been developed to test not only software but sequential circuits, communication channels and web systems (Binder, 2000; Chow, 1978; Friedman & Menon, 1971; Haydar, Petrenko & Sahraoui, 2004; Holzmann & Lieberman, 1991).

Once a system $N$ is implemented and the corresponding FSM $M$ is known, testing involves determining whether $N$ is a correct implementation of $M$. In this case, $N$ is given as a block box, where the test suite can only apply inputs and observe the outputs. This kind of tests are often applied as checking sequence experiments, where a series of specific input sequences are applied to the system $N$. Then, the outputs of the real system $N$ is compared to the outputs of corresponding FSM $M$. In this test suite, the sequences are specifically derived for testing two different aspects of a state machine: first, to check whether each state in $N$ can be identified as corresponding to a state $M$, and second, to check whether $N$ and $M$ have the same state transitions. Both of these checks are achieved by using sequences called *state identification sequences (SISs)* . State identification sequences are known to be produced by using *distinguishing sequences* (Gonenc, 1970; Hennie, 1964; Kohavi, 1978; Ural, Wu & Zhang, 1997), *unique input-output sequences* (Aho, Dahbura, Lee & Uyar, 1991; Sabnani & Dahbura, 1988; Vuong, 1989), *W–sets (also known as characterizing sets)* (Hennie, 1964; Kohavi, 1978; Kohavi, Rivierre & Kohavi, 1974; Lee & Yannakakis, 1996) and *K–sets* (Jourdan, Ural & Yenigün, 2016).

Distinguishing sequences are either preset or adaptive; *preset distinguishing sequence (PDS)* is a flat sequence; whereas *adaptive distinguishing sequence (ADS)* is a decision tree based on outputs of states. Both PDS and ADS of an FSM produce a unique output for each state of the FSM. A unique input-output sequence is an input-output sequence pair $(x,y)$ for a state $s$ such that the output sequence $y$ is produced by the application of the input sequence $x$ only if the FSM is in the state $s$. A W–set is a set of sequences such that for every state pair $\{s_i, s_j\}$ in the FSM there exists a sequence $x$ in the W–set such that $s_i$ and $s_j$ produce different output sequence for $x$. K–set is similar to W–set with a single difference: elements are partial decision trees, similar to ADSs.

Distinguishing sequences and unique input-output sequences are not guaranteed to exist for all FSMs. Showing whether an FSM has a PDS or whether the state of an FSM has unique input-output sequence is PSPACE-complete (Lee & Yannakakis, 1994), though showing whether an FSM has an ADS can be achieved in polynomial time complexity (Lee & Yannakakis, 1994). On the other hand, W–sets and K–sets can be found for every FSM, given that it is completely specified and minimal. Therefore, W–set/K–set based methods can be applied on a broader class of FSMs. To the best of our knowledge, there is no prior work that shows a particular polynomial upper-bound for the time complexity of W–set generation algorithms. It is obvious that W–set algorithms have polynomial upper bound, since the cardinality of W–set is polynomial and the cost of each sequence is also polynomial. However, in this document we give a particular polynomial upper bound for the W–set construction algorithms we suggest. Soucha & Bogdanov (2020) shows that K–sets can be built in polynomial time, as well.

Once an SIS is found for the FSM $M$, it should be applied on the implementation $N$ and the FSM $M$. Due to this fact, the length of the SIS is important. Throughout the literature, algorithms have been compared by the length of the SIS they produced. Each of these algorithms tried to bring a different approach for producing a shorter SIS from the very same W–set given. However, another way to improve (reduce) the SIS length is to optimize the W–set. If the SIS generation algorithms are fed a W–set with less elements and/or smaller elements, this might greatly improve the performance of the algorithms. Though, for more sophisticated SIS algorithms that compile an intermediate data structure - like K–trees in Jourdan et al. (2016) and similar structures implied in Kohavi (1978); Kohavi et al. (1974) -, these two qualities of W–sets may not be important at all.

In this work,

- We propose improved algorithms for W–set generation in Chapter 4. Some of these algorithms strictly aim to produce a W–set with a lower cardinality and a lower average item length.

- We then propose another W–set generation algorithm that is optimized for K–tree generation in Chapter 4, as well. K–tree is the main intermediate data structure that Kohavi (1978); Kohavi et al. (1974) and Jourdan et al. (2016) works with. We need to address that, the term *K–tree* was coined and K-trees are formally defined by Jourdan et al. (2016). It is only our optimistic interpretation that Kohavi (1978); Kohavi et al. (1974) intended to create a K–tree like data structure.

- We share the results of the experiments that we performed in order to compare the performances and the quality results of these W–set algorithms in Chapter 6.

- We propose an algorithm to generate K–trees from any W–set or K–set given for an FSM in Chapter 5. Although Jourdan et al. (2016) shows how K–trees can be used for SIS generation, they do not provide any methods for generating K–trees. The algorithm that we propose is an open-ended algorithm that can be tuned by the usage of any scoring procedures. We put forth four such procedures in the same chapter, as well.

- Lastly, in Chapter 7, we perform an experimental study by implementing all of the available W–set/K–set algorithms, the K–tree generation algorithm and SIS generation techniques to make a quality comparison of the SIS generation algorithms existing in the literature.

# 2. DEFINITIONS AND NOTATION

In this chapter, we briefly mention the concepts that we build our work upon and define the notation that we will use to explain them throughout the document.

A *deterministic finite state machine (FSM)* is defined by a quintuple $M = (S, I, O, \delta, \lambda)$ where $S$ is a finite set of $n$ *states*, $I$ is a finite alphabet consisting of $p$ *input letters* (or simply *inputs*), $O$ is a finite alphabet consisting of $q$ *output letters* (or simply *outputs*), $\delta : S \times I \to S$ is a partial *transition function* and $\lambda : S \times I \to O$ is a partial *output function*.

In an FSM $M$, for a state $s \in S$, an input $x \in I$ and an output $y \in O$; $\delta(s, x) = s'$ indicates that state $s$ is taken to state $s'$ and $\lambda(s, x) = y$ indicates that state $x$ produces output $y$, both when input letter $x$ is applied.

An FSM $M$ is considered to be *complete* or *completely specified* if the partial functions $\delta$ and $\lambda$ are functional. In simpler terms, the functions $\delta$ and $\lambda$ should be defined for each pair $\langle s, x \rangle \in S \times I$ exactly once. In this document, we consider only complete FSMs.

A sequence $w \in I^\star$ is called an *input word* (or simply *word*). The definitions of $\delta$ and $\lambda$ functions can be extended to accommodate words in the following manner: For a state $s \in S$, an input $x \in I$ and a word $w \in I^\star$, we define $\delta(s, x.w) = \delta(\delta(s, x), w)$ and $\lambda(s, x.w) = \lambda(s, x).\lambda(\delta(s, x), w)$ where $\epsilon$ is empty sequence and $\delta(s, \epsilon) = s$, $\lambda(s, \epsilon) = \epsilon$.

An FSM $M$ is called *strongly connected* if for any two states $s_i, s_j \in S$ there exists a word $w$ such that $\delta(s_i, w) = s_j$.

A word $w \in I^+$ is said to *separate the states $s_i$ and $s_j$* if $\lambda(s_i, w) \neq \lambda(s_j, w)$. In this case, $w$ is named a *separating sequence (or separating word) for $s_i$ and $s_j$*. Take the states $s_3$ and $s_6$ from the example FSM $M_1$ given in Figure 2.1. The outputs of these two states to the word *bab* are 011 and 010 respectively. In this case, the word *bab* separates this state pair $\{s_3, s_6\}$ and is called a separating sequence for $\{s_3, s_6\}$. In contrast, *ba* does not separate $\{s_3, s_6\}$; because they give the common output 01

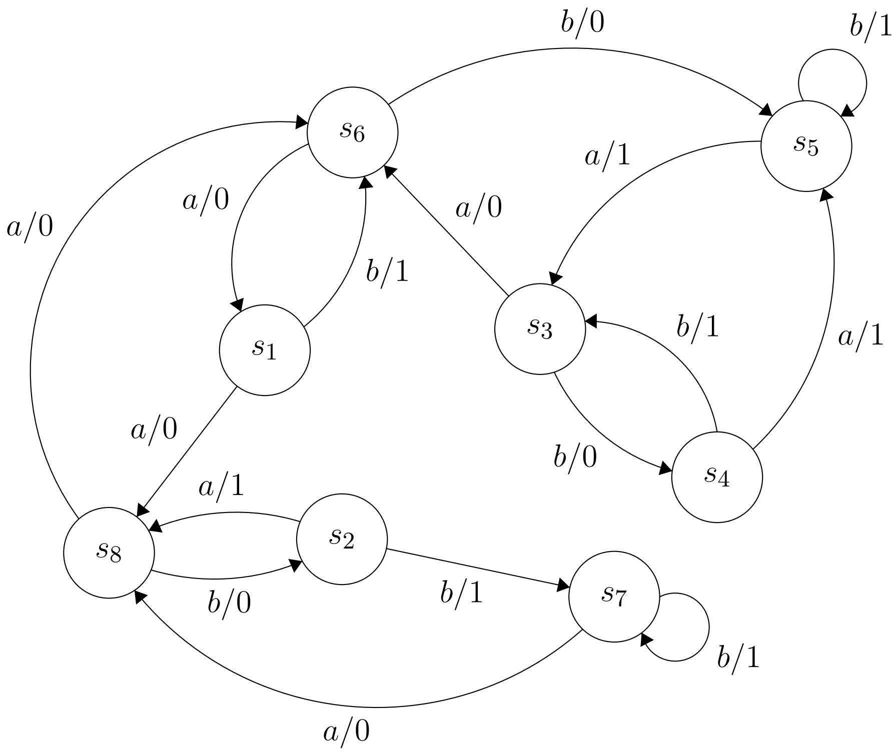

Figure 2.1 An example complete, minimal and strongly connected FSM $M_1$

to the word $ba$. Two states $s_i$ and $s_j$ are called *equivalent* in the absence of any such separating sequences. Building from this definition, two FSMs $M_i$ and $M_j$ are *equivalent* if each state $s_i$ from $M_i$ can be mapped equivalent to a state $s_j$ from $M_j$ and each state $s_j$ from $M_j$ can be mapped equivalent to a state $s_i$ from $M_i$.

A word $w \in I^+$ is said to *merge the states $s_i$ and $s_j$* if $\lambda(s_i,w) = \lambda(s_j,w)$ and $\delta(s_i,w) = \delta(s_j,w)$. For example, the states $s_2$ and $s_5$ from the example FSM $M_1$ (given in Figure 2.1) are merged by the word $aa$, because $\lambda(s_2,aa) = \lambda(s_5,aa) = 10$ and $\delta(s_i,w) = \delta(s_j,w) = s_6$.

An FSM $M$ is defined to be *minimal* if no equivalent FSMs for $M$ with less states than $|S_M|$ can be found. By combining this definition and the concept of separation, a natural and an obvious consequence follows:

**Lemma 1.** *In a minimal FSM $M$, each state pair $\{s_i,s_j\}$ must have a separating sequence.*

*Proof.* The proof of this lemma comes from contradiction. We may start by assuming that $M$ is minimal but there exists a state pair $\{s_i,s_j\}$ that does not have a separating sequence. By definition, these states are *equivalent*. Then, as we have a pair of equivalent states $s_i$ and $s_j$, we can merge them into a single state and the resulting FSM $M'$ is equivalent to $M$. As $M'$ is produced by reducing the state size of $M$ by 1, now we have an equivalent FSM to $M$ that has less state than it. This contradicts with the minimality of $M$. $\square$

For an FSM $M$, a set $W = \{w_1, w_2, \ldots, w_r\}$ of input sequences is called a *W-set* (or a *characterizing set*) if for each state pair $\{s_i, s_j\}$ of $M$, at least one element of $W$ is a separating sequence. For example, $W_1 = \{aa, ab, bb, baa\}$ is a W–set for the example FSM $M_1$ given in Figure 2.1. One can easily check that any pair of states of $M_1$ can be separated by using at least one of the sequences in $W_1$.

In literature, the term *state identification sequence (SIS)* is used as a broad term, as stated in Chapter 1. Basically, a state identification sequence is used to identify a state of an implementation as being similar to a state of the specification. When a single sequence, e.g. a distinguishing sequence, is available, then the application of this sequence at a particular implementation state is sufficient to identify the state of the implementation. However, when no such "single sequence identification" is possible, then multiple sequences need to be applied at the same implementation state. For example, when one has to use a W-set for state identification purposes, the elements of the W-set have to be applied to the same implementation state. This property that several sequences need to be applied at the same implementation state, requires repeated applications of the elements of the W-set using a particular strategy. One such strategy is given by the following recursive definition for a *state identification sequence* based on a W-set $W = \{w_1, w_2, \ldots, w_m\}$ (Gargantini, 2004; Hennie, 1964):

$$\begin{aligned} \beta_1 &= w_1 \\ \beta_r &= (\beta_{r-1} t^i_{r-1})^n w_r \end{aligned} \tag{2.1}$$

This equation is used to construct an SIS for a state $s_i$. In this equation, $t^i_j$ is called a *transfer sequence*, which takes the FSM back to state $s_i$ after the application of $w_j$ at $s_i$. In other words, $t^i_j$ is sequence such that $\delta(s_i, w_j t^i_j) = s_i$.

Here, the state identification sequence is $\beta_m$. The implementation $N$ corresponding to the FSM $M$ produces the same output as $M$ produces when starting from the state $s_i$ when $\beta_m$ is applied, then $N$ has a state similar to $s_i$ of $M$ and this state is the state of $N$ right before the application of the last $w_m$.

Although SIS is a general term (e.g. a distinguishing sequence is also an SIS), in this work we always use the term SIS to denote an SIS based on a W–set or a K–set.

An *adaptive distinguishing sequence (ADS)* for an FSM is a rooted tree with $n$ leaves. Every node in the ADS is assigned a state set $S' \subseteq S$ and an input letter $x \in I$. The root is labeled by $S$, the set of all states. If a node has a singleton state set, then this node is a leaf node for the tree. Every edge in the ADS is labeled with an output letter $y \in O$ where the edges leaving the same node have different labels. Consider a (non–root) node $v$ in an ADS and let $v'$ be the parent of $v$. Let $w \in I^+$ be the input sequence obtained by concatenating the input symbols from the root to $v'$ (including $v'$), and let $\beta \in O^+$ be the output sequence obtained by concatenating the output symbols on the edges from the root to $v$ (including the edge from $v'$ to $v$). A state $s \in S$ is in the set of states of $v \iff \lambda(s,w) = \beta$. In addition, if the input label of $v$ is $x$, then the node $v$ has an outgoing edge labeled with $\lambda(\delta(s,w),x)$ for each $s$ in the states of $v$.

A *partial adaptive distinguishing sequence (PADS)* - also known as *incomplete adaptive distinguishing sequence (IADS)* - is a generalization of ADS such that the number of leaves need not be $n$ and the leaf nodes need not have singleton state sets. With this property, a PADS does not necessarily separate all state pairs from each other, rather it creates a partition of states that need not have $n$ blocks. A PADS $Y$ is said to separate a state pair $\{s_i, s_j\}$ if there exist a node $v$ in $Y$ such that $s_i$ is in the state set labeling $v$ and $s_j$ is not.

For an FSM $M$, a set $\mathcal{Y} = \{Y_1, Y_2, \ldots, Y_r\}$ of PADSs is called a *K-set* if for each state pair $\{s_i, s_j\}$ of $M$, at least one element of $\mathcal{Y}$ separates $s_i$ and $s_j$. An example K–set $\mathcal{Y}'$ is given in Figure 2.2 for the sample FSM $M_1$.

A K–tree $\mathcal{T}$ for an FSM $M$ is a rooted tree with $n$ leaves, where each leaf is labeled by a distinct state of $M$. Non-leaf nodes of a K–tree are labeled by PADSs and implicitly contain a set of states of $M$. The root node contains $S$. For any non-leaf node $v \in \mathcal{T}$ and two states $s_i$ and $s_j$ that $v$ contains, $s_i$ and $s_j$ are contained by different children of $v \iff Y_v$ separates $\{s_i, s_j\}$.

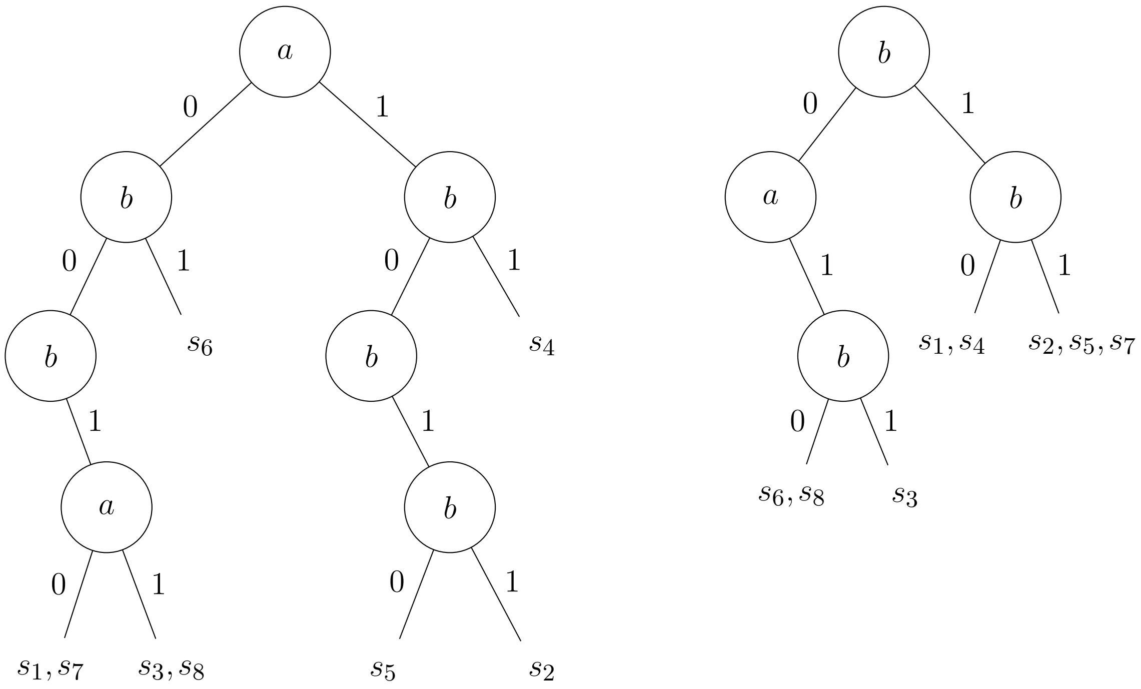

Figure 2.2 A K–set $\mathcal{Y}' = \{Y_1', Y_2'\}$ for the FSM $M_1$, where $Y_1'$ is the PADS on the left and $Y_2'$ is the PADS on the right

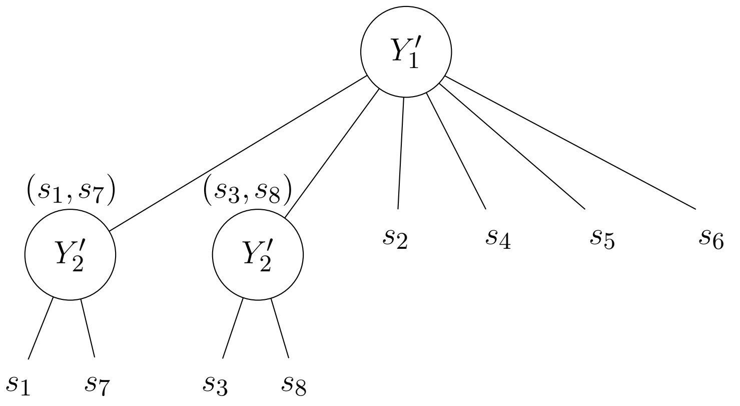

Figure 2.3 A K–tree $\mathcal{T}_1{}'$ for the FSM $M_1$, generated from the K–set $\mathcal{Y}'$

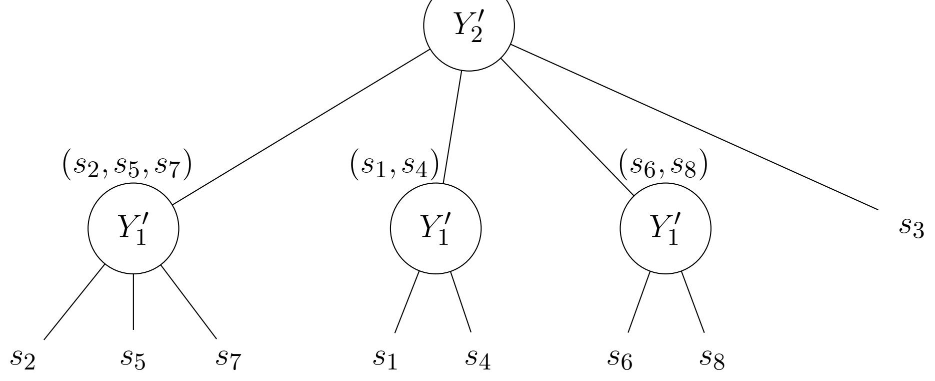

Figure 2.4 Another K–tree $\mathcal{T}_2{}'$ for the FSM $M_1$, generated from the K–set $\mathcal{Y}'$

# 3. RELATED WORK

This chapter is organized into two sections. In Section 3.1, we talk about the previous approaches for W–set generation. In Section 3.2, we go over the previous approaches for W–set based state identification sequence generation.

## 3.1 Related W–set Work

The literature for W–set (also known as *characterizing set*) generation methods is not thick. The first algorithm described for W–set generation was proposed in Gill (1962). Gill's algorithm works in a partitioning fashion. The partition starts with a single block of all states and an empty W–set. In each iteration, a non-singleton block is taken, the separating sequence of a pair in this block is applied to the states in that block and the block is broken into multiple blocks. In this process, the separating sequence that is used is added to the W–set. The process basically stops when all the blocks are singletons. This naive and straight-forward algorithm has been the base algorithm in the literature for quite a long time, despite having two obvious weaknesses: First, the algorithm misses the opportunity to see that, a sequence that is used for dividing a partition can indeed divide other partitions as well. Secondly, the algorithm does not include a useful post-processing opportunity: *prefix elimination.* In a W–set $W$ where $w_i, w_j \in W$, if $w_j$ is a prefix of $w_i$, then $W \setminus \{w_j\}$ is still a W–set. The reasoning is straight-forward: two states $s_i, s_j$ that are separable by $w_j$ must give different outputs to $w_j$ by the definition of separation. As $w_i$ has the prefix $w_j$, the first $|w_j|$ output letters of $s_i$ and $s_j$ as response to $w_i$ must be different as well. This imply that $w_i$ must separate all state pairs that are separable by its prefix $w_j$.

These two deficits of Gill's algorithm show that the version of this algorithm bears an inherent redundancy. However, in Chapters 6-7, we favor Gill's algorithm by adding prefix-elimination; because we want to make a fair comparison of the algorithms that we propose in this document. The original depiction of Gill's algorithm is given below as Algorithm 1.

**Algorithm 1:** Gill's W–set building algorithm

```
input  : An FSM $M = (S, I, O, \delta, \lambda)$, a W–set $W$
output: A W–set $W$ for $M$
1  $W = \emptyset$; // $W$:  current W-set, initially empty
2  $\Pi = \{S\}$; // $\Pi$:  state partition, initially all states are in a single block
3  while $\Pi$ has a non–singleton block do
4      take a non–singleton block $B \in \Pi$ and remove $B$ from $\Pi$
5      take a pair $\{s_i, s_j\}$ such that $s_i, s_j \in B$
6      let $u$ be a separating sequence for $\{s_i, s_j\}$
7      partition $B$ into blocks $B_1, B_2, \ldots, B_k$ such that
         $\forall s, s' \in B\,; s, s' \in B_i$ for some $B_i \iff \lambda(s,u) = \lambda(s',u)$
8      $\Pi = \Pi \cup \{B_1, B_2, \ldots, B_k\}$
9      insert $u$ to $W$
10 end
```

Another W–set generation technique proposal in the literature was made in Vasilevskii (1973). The mechanics of this technique is pretty simple. The technique keeps track of state pairs that are not separated yet, which initially should be all state pairs. It then finds the "Next" sequence that can separate an unseparated state pair in each iteration. *"Next"* happens in lexicographical order. The process stops when the current unseparated pairs set is empty. We abstain from calling this proposal an algorithm; because how the next sequence should be determined is unclear. If we consider a simple loop enumerating all input sequences lexicographically, it will start with the first letter (let's say $a$) and the first letter will not change for a huge number of iterations. Then, we know that any pair that is "merged" by $a$ cannot be solved by a sequence starting with $a$. Even though we limit the maximum input sequence length to $n-1$ (as this number is the upper-bound for the length of a separating sequence (Gill, 1962)), the algorithm has a great chance of iterating over all the sequences of length at most $n-1$ starting with $a$ and still has not completed the W–set. As Vasilevskii (1973)'s proposal does not include any other detail on the iteration mechanism, we may suggest that this technique is either too expensive or incomplete.

In Gargantini (2004), another algorithm for W–set generation is described. The algorithm is nearly identical to Gill's algorithm, with a slight difference. Gargantini's algorithm fixes the first weakness of Gill's algorithm that we mentioned above. It processes and breaks all the blocks (not only the block of consideration) when adding a new input sequence to W–set. To the best of our knowledge, this algorithm had not appeared in the literature in this exact way. Gargantini (2004) neither shows a reference for this idea, nor claims that it is novel. The algorithm is given below as Algorithm 2.

**Algorithm 2:** Gargantini's W–set building algorithm

```
input  : An FSM $M = (S, I, O, \delta, \lambda)$, a W–set $W$
output: A W–set $W$ for $M$
1  $W = \emptyset$; // $W$:  current W-set, initially empty
2  $\Pi = \{S\}$; // $\Pi$:  state partition, initially all states are in a single block
3  while $\Pi$ has a non–singleton block do
4      take a non–singleton block $B' \in \Pi$
5      take a pair $\{s_i, s_j\}$ such that $s_i, s_j \in B'$
6      let $u$ be a separating sequence for $\{s_i, s_j\}$
7      forall non-singleton block $B^i \in \Pi$ do
8          partition $B^i$ into blocks $B^i_1, B^i_2, \ldots, B^i_k$ such that
            $\forall s, s' \in B^i\,; s, s' \in B^i_j$ for some $B^i_j \iff \lambda(s,u) = \lambda(s',u)$
9          $\Pi = \Pi \cup \{B^i_1, B^i_2, \ldots, B^i_k\} \setminus \{B^i\}$
10     end
11     insert $u$ to $W$
12 end
```

During literature scan, we ran into the paper Miao, Liu & Mei (2010), which proposed a new W–set generation algorithm and comments on the previous approaches. We examined the new algorithm they proposed and could not get a clear idea on how they suggested to separate state pairs that cannot be separated by a single input letter. The time complexity analysis they made on their own algorithm yields to a time complexity of $O(n^2)$. However, it brings suspicion that they did not incorporate the number of input letters $p$ into the time complexity analysis. Their time complexity analysis also does not examine the more complex part where the states are carried with multiple input letters to find a separation point. The fact that this process might take $n-1$ times in the worst case (Gill, 1962) adds even a bigger doubt on the time complexity analysis made by the authors. One upside is, they apply prefix-elimination as the last step of their algorithm and also comment that Gill's algorithm lacks prefix elimination for bad. This is a point that we agree, as we explained above as well. The W–set algorithms that we propose in Chapter 4 (except one, for a valid reason) also make use of this technique. We do not present the pseudo-code of the algorithm Miao et al. (2010) proposes, as their paper has one already.

Another attempt to reduce the W–set was made in Bulut, Jourdan & Türker (2019).

Their algorithm took a different road and formed a breadth-first search tree of input letters to make all separation attempts with as short sequences as possible. This breadth-first search tree might expand until the depth $n-1$, as this number is the upper-bound for the length of a separating sequence (Gill, 1962). Different from Gill's approach, this algorithm keeps track of state pairs to be separated. In the beginning, all non-identical state pairs are included in this collection. At each depth, the permutations of input letters on the newly generated leaf nodes are applied to all non-separated state pairs. The sequences are added to the W–set as long as they can separate a non-separated state pair. This algorithm proves to produce better W–sets than Gill's algorithm (in terms of both cardinality and total length), in exchange of a performance penalty. Due to the exhaustive breadth first search approach, the algorithm shows exponential time complexity. In this algorithm, the input sequences of length 1 are almost always guaranteed to be included in the W–set. With this being said, it is obvious that this algorithm is prone to excessive growth if prefix elimination is not applied. Still, we do not see any mention of this technique in the original paper. However, similarly to Gill's algorithm, we will also consider this algorithm with prefix elimination during our empirical work in this document. We do not give the pseudo-code of this algorithm neither, as it is given in the original paper in a clear way.

## 3.2 Related W–set Based State Identification Work

In this section, we go over the approaches in the literature about W–set based state identification sequence generation. We explained in Chapter 1 that there are other SIS generation categories, but we exclusively worked on the W–set based methods for this document; hence we only recite the previous methods on this specific category of SIS methods.

The first W–set based SIS generation technique was proposed in Hennie (1964). The formula that Hennie generated is given in Equation 2.1. The rationale behind the formula is the following simple hypothesis: "If an FSM gets applied the same input sequence sufficiently many times, the last state it will reside in must be a state it previously applied the input sequence at". More specifically, this number is $n+1$ for an FSM with $n$ states. Additionally, if the input sequence causes the same output sequence generated by the FSM repeatedly, Hennie shows that this output repetition gets guaranteed after $n+1$ times. At this point, another sequence from the same W–set can be chosen and applied. By this, the method makes sure that two different sequences are applied to the same state of the implementation. This process is repeated recursively to ensure that all W–set elements (sequences) are applied to this same state of the implementation. The outputs produced reveal which state it is.

Hennie's idea is laid out confidently; but the resulting SISs seem to be quite long (Rezaki & Ural, 1995). The main reason of this length are (i) there are $n+1$ repetitions used and (ii) the length is exponential in the number of elements in the W-set. Hennie, in the same paper, suggests a method that can make the length of the SISs shorter. He puts forth that, if there are $p$ states that give the same output to the previously applied W–set elements, then instead of $n+1$, $p+1$ applications of the input sequence could be sufficient. Although this claim looks promising, Hennie does not give a clear definition of this idea and unfortunately does not prove his point. We did not find any authors that referred to or used this idea for SIS generation (except Jourdan et al. (2016)), they rather sticked to the original proposal with $n+1$ iterations. Jourdan et al. (2016) consider the possibility of applying only $p+1$ repetitions. Our work is also based on Jourdan et al. (2016). In the rest of this document, we will refer to Hennie's incomplete idea (that $p+1$ repetitions can be used) as Hennie Improved.

In Lee & Yannakakis (1996), the authors of the survey mention Hennie's original formula, but they "correct" the formula to use $n$ iterations rather than $n+1$. No other changes were done on Hennie's formula other than this.

In two papers by Kohavi *et al.*, specifically Kohavi (1978); Kohavi et al. (1974), another approach to SIS generation was mentioned. They propose an adaptivity based approach, where the states are partitioned according to their outputs to an applied input sequence. Then each partition is applied a different series of W–set elements and the partitioning continues until singleton partitions are reached. We understand that, this idea resembles the K–tree concept we defined in Chapter 2. Kohavi *et al.* also use a different iteration count than Hennie. The first element of W-set is applied $n+1$ times, similar to Hennie. All other partitions/nodes use $n'+1$, where $n'$ is the state count of their grandest parent node except the root. In Figure 3.1, a partial K–tree $\mathcal{T}_3$ is given to illustrate the working of this method. In $\mathcal{T}_3$, with Kohavi's method, the iteration count of the node labeled by $S$ is $|S|+1$, which is identical to Hennie's iteration count for all steps, $n+1$. On the other hand, the iteration counts of the nodes labeled by $S'$, $S''$ and $S'''$ (and any other children of these nodes) are all $|S'|+1$. With this approach, Kohavi *et al.*'s method is able to use lower iteration counts and avoid applying all W–set elements for the state

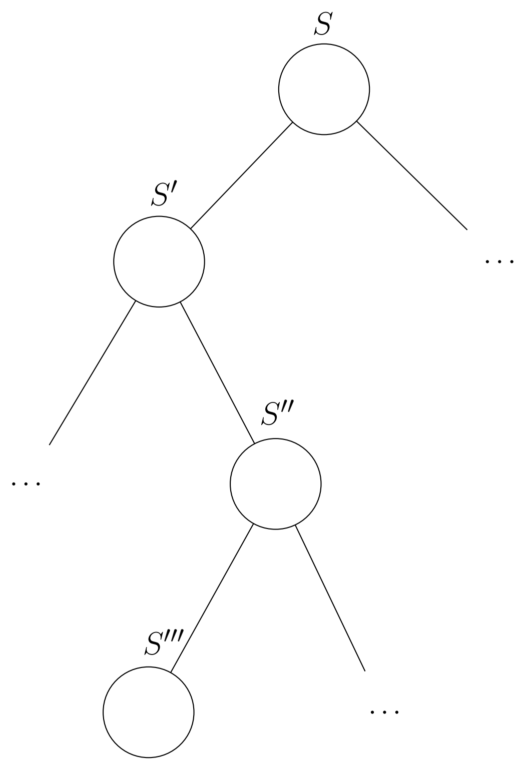

Figure 3.1 An example partial K–tree $\mathcal{T}_3$

identifications. However, we would like to remind that, Kohavi *et al.* does not give any formal definition of their method and the inference that they intended to use a K–tree-like structure is our optimistic take to their assertion.

Lastly, Jourdan et al. (2016) proposes a novel method for SIS generation, which we adhere to in this document as JUY. This paper defines the construct K–tree and proposes a SIS generation formula based on K–trees. The formula that was proposed by this paper still bears the same DNA as Hennie's, with only iteration count changes. With their definition of the formula, each partition/node on the K–tree may use $l+1$, where $l$ is the state count of the partition/node itself. If we take the example partial K–tree $\mathcal{T}_3$ again, the iteration count of the node labeled by $S$ is still $|S|+1$ and the iteration count of the node labeled by $S'$ is $|S'|+1$. These two are identical to Kohavi's iteration counts. However, the iteration counts of the nodes labeled by $S''$ and $S'''$ are calculated as $|S''|+1$ and $|S'''|+1$ respectively by JUY. This means that as we get deeper on the K–tree, the iteration count keeps decreasing, unlike Kohavi *et al*'s proposal. Another addition is, K–tree definition made in Jourdan et al. (2016) is compatible with PADS. So, a K–tree can be built by using a K–set as well. This situation gives another upper hand to the proposal JUY.

# 4. CONTRIBUTIONS ON W–SET GENERATION

W–sets are utilized in many FSM-based testing methods (Hennie, 1964; Kohavi, 1978; Kohavi et al., 1974; Lee & Yannakakis, 1996). Decreasing the cardinality and average element length of w–sets surely contributes to all of these methods. Hence, we propose three new algorithms in sections 4.1, 4.2 and 4.3 which aim to improve these two aspects. Apart from that, we have K–tree algorithms which are introduced in Section 5. These K–tree algorithms also use W–sets and can benefit from the improvements achieved by these new W–set algorithms. In Section 4.4, we propose another W–set algorithm that does not compete to improve W–set cardinality or average element length, but specifically enhance the K–tree quality.

In most of our algorithms, we use a separating pair graph generated from FSMs. The definition is given below.

**Definition 1** (Given as *Distinguishing Automaton* in Definition 3 of Güniçen, İnan, Türker & Yenigün (2014))**.** *The separating pair graph $G = (V, E)$ of FSM $M = (S, I, O, \delta, \lambda)$ is a directed graph where the set of nodes and the set of edges are defined as follows:*

$V = \{\{s_i, s_j\} \mid s_i \neq s_j, s_i, s_j \in S\} \cup \{\mathit{Merged}, \mathit{Separated}\}$

$E \subseteq V \times I \times V$ *is a set of edges labeled with the input symbols of the FSM $M$.*

$\forall \{s_i, s_j\} \in S \times S$*, where* $s_i \neq s_j$*, and* $\forall x \in I$*,*

- $(\{s_i, s_j\}, x, \mathit{Separated}) \in E$ *iff* $\lambda(s_i, x) \neq \lambda(s_j, x)$
- $(\{s_i, s_j\}, x, \{\delta(s_i, x), \delta(s_j, x)\}) \in E$ *iff* $\lambda(s_i, x) = \lambda(s_j, x)$ *and* $\delta(s_i, x) \neq \delta(s_j, x)$
- $(\{s_i, s_j\}, x, \mathit{Merged}) \in E$ *iff* $\lambda(s_i, x) = \lambda(s_j, x)$ *and* $\delta(s_i, x) = \delta(s_j, x)$

On a separating pair graph, for any graph vertex $v$ corresponding to a state pair $\{s_i, s_j\}$ of FSM $M$, the label of a path from $v$ to *Separated* is a separating sequence for $\{s_i, s_j\}$. Hence, non–existence of a path from vertex $v$ to *Separated* means that the states $s_i$ and $s_j$ are equivalent. Using this observation, the minimality of an FSM can simply be performed by checking the backward reachability of all vertices corresponding to state pairs from the vertex *Separated*. Similarly, shortest separating sequences of state pairs can be discovered by a backward breadth first search from *Separated*. Similarly, a shortest separating sequence of states $s_i$ and $s_j$ can be found by finding a shortest path from the vertex $\{s_i, s_j\}$ to the vertex *Separated* in the separating pair graph. One can find shortest separating sequences of all state pairs by using a single *backward* Breadth First Search starting from the vertex *Separated*.

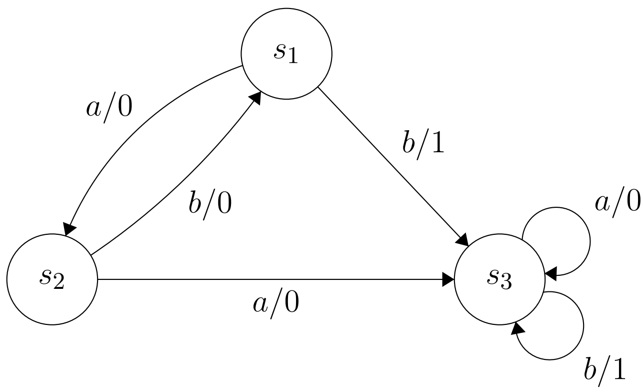

Figure 4.1 A small example FSM $M_2$

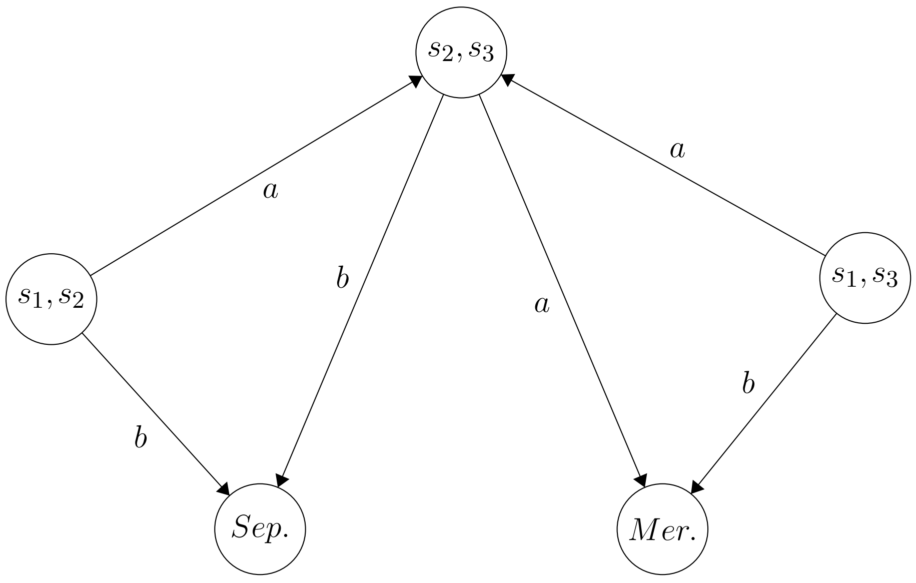

Figure 4.2 The separating pair graph of the FSM $M_2$ in Figure 4.1

Constructing the *separating pair graph*, checking the minimality of the FSM using a separating pair graph, and finding shortest separating sequences of all states pairs using a separating pair graph can all be performed in time $O(pn^2)$.

The algorithms in the upcoming sections all use state pairs for the sake of implementation simplicity. Designing the same algorithms by using state blocks is also possible. Due to implementing these algorithms via the state pairs approach, they have higher theoretical time complexities than what is normally possible. However, the time complexity analysis depends on the worst cases which we do not observe in reality and the run-times of our algorithms are pretty competitive, as Chapter 6 shows. The details of this situation is given in detail in Section 4.5.

## 4.1 The Algorithm: Chassis

As explained in the previous sections, a W–set consists of such sequences that each pair of states in the FSM can be separated by at least one of them. A very basic and inefficient observation is: A collection of the separating sequences of every pair makes a W–set. This is theoretically correct due to the fact that every pair is guaranteed to have a separating sequences in the set, which naturally qualifies for a W–set. In practice, this method builds W–sets with $n \times (n-1)/2$ elements. We can do better than this. As an enhancement, we may append a procedure that iteratively eliminates some of the sequences from the set and checks if it still is a W–set. This approach seems to yield a much smaller W–set for the FSM, but then the creation process is prolonged a lot. In the removal step of each sequence, all the remaining pairs should be checked against the remaining sequences in the set, in order to see whether they still can be separated with the remaining sequences or not. Although this approach may not be a fast method, it shows us that a subset of the set of all separating sequences can be used as a W–set. By this last inference, we can tweak the method such that we start with an empty set and populate it with separating sequences, rather than starting with separating sequences of all pairs and reducing from that point. By this last modification, we conclude the design of our algorithm called Chassis (**CHA**racterizing **S**et using **S**eparating **S**equences), which is given below as Algorithm 3.

**Algorithm 3:** CHASSIS

```
input  : An FSM $M = (S, I, O, \delta, \lambda)$
output: A W–set $W$ for $M$
1 compute the shortest separating sequence $w_{\{i,j\}}$ for every state pair $s_i, s_j \in S$
2 $W = \emptyset$; // $W$: current W-set, initially empty
3 $Z = \{\{s_i, s_j\} \mid s_i \neq s_j, s_i, s_j \in S\}$; // $Z$: set of pairs yet to be separated
4 while $|Z| > 0$ do // we still have non-separated pairs, need to add more sequences to $W$
5     take any pair $\{s_i, s_j\} \in Z$
6     $Z = Z \setminus \{\{s'_i, s'_j\} \in Z \mid s'_i$ and $s'_j$ are separated by $w_{\{i,j\}}\}$
7     insert $w_{\{i,j\}}$ to $W$
8 end
9 Remove the sequences from $W$ that are prefixes of some other sequences in $W$.
```

Algorithm 3 follows the approach developed in the previous paragraph. First, the algorithm computes the shortest separating sequences of every state pair at line 1. This is achieved by using *separating pair graph* as explained in Definition 1 and applying *backwards* breadth first search on it. Later, a current form of W–set is stored in $W$, which is initially empty. The algorithm also keeps track of non-separated state pairs in $Z$, which initially has all possible state pairs.

The main algorithm continues until all state pairs of FSM $M$ has a separating sequence in $W$. In every iteration, we take a random state pair $\{s_i, s_j\}$ from the non-separated state pair set $Z$. The shortest separating sequence $w_{\{i,j\}}$ of this state pair is applied to every state pair in $Z$. The state pairs which are separated by this sequence (including $\{s_i, s_j\}$ ) are removed from $Z$. This process is guaranteed to remove at least one state pair from $Z$ in each iteration, because a pair must be separated by its own shortest separating sequence naturally. Hence, the iteration is also guaranteed to stop. By the end of the algorithm, we may have included some unnecessary sequences in the W–set. If a sequence $w_i \in W$ is a prefix of another sequence $w_j \in W$, $w_j$ is capable of separating all the pairs that $w_i$ can. In this case, $w_i$ can be removed from $W$ safely. The last step of the algorithm traverses $W$ and handles these mentioned cases.

The precomputation part of Algorithm 3 (line 1) generates the separating pair graph and computes the shortest separating sequences of each state pair from it. This step has the time complexity of $O(pn^2)$ as mentioned in the beginning of this chapter. The symbol $n$ denotes the number of states in the FSM and $p$ denotes the number of input letters. The loop in line 4 may iterate at most $n-1$ times; because this loop adds a sequence to the W–set and a W–set can contain at most $n-1$ elements (Sandberg, 2004). For each iteration of the loop, all the remaining state pairs in

$Z$ are tested with a separating sequence. The remaining state pairs in $Z$ might be as many as $n \times (n-1)/2$. Sandberg (2004) also shows that the length of a shortest separating sequence of a state pair in an FSM is at most $n-1$. Consequently, the loop portion of Algorithm 3 has theoretical time complexity of $O(n^4)$. Merged with the precomputation part, the time complexity of the algorithm can be described as $O(pn^2 + n^4)$. Although this might seem high, we do not observe this high upper bound much in practice. So, this algorithm does perform well in real life, as our experiments in Chapter 6 show.

Figure 2.1 shows an example FSM $M_1$ for remarking the differences between the characteristics of the algorithms that we propose in this document. When Chassis (Algorithm 3) is applied to $M_1$, the resulting W–set is $\{aa, ab, bb, baa\}$. We note that this result has cardinality of 4 and total letter count of 9.

The algorithm Chassis behaves similarly to Gill's algorithm (given as Algorithm 1) internally. Although Gill's algorithm traces the progress of the separation in state partitions and our algorithm Chassis traces it as a set of unseparated state pairs, the decision mechanisms are the same. There are two improvements that Chassis accomplishes over Gill's algorithm: (i) all partitions are divided during the addition of a sequence to the W–set, (ii) prefix-elimination is applied for removing redundant elements from the W–set. As we explained in Section 3.1, the first improvement was used in Gargantini (2004) to depict a "traditional" method for W–set generation and the second improvement was briefly mentioned in Miao et al. (2010). At this point, Chassis is an algorithm that uses both of these approaches to propose a simple and easy-to-implement solution for W–set generation. Due to the fact that these improvement ideas were discovered and mentioned before us, we do not assert that this is a completely novel algorithm. However, we believe that Chassis combines the previously mentioned - but not stressed - approaches and declare it as a whole package, which also creates the **chassis** (base) for our novel algorithms proposed in the rest of this chapter.

## 4.2 The Algorithm: Chassis-C

Our first algorithm, as explained in the previous section, follows a simple approach for creating W–sets: It picks from the existing shortest merging sequences of state pairs and remove all the pairs that are separated by this sequence. This technique starts a new W–set element in every iteration and may result in a relatively high number of elements in the W–set. An observation during the implementation of Chassis was that some pairs indeed got closer to being separated by the sequences applied, but were not pursued to the end. For instance, we may consider a sequence $w$ being applied to all state pairs, including pair $\{s_i, s_j\}$. After the application of $w$, assume that $\{s_i, s_j\}$ is taken to another pair $\{s'_i, s'_j\}$ where $s'_i \neq s'_j$ but the output sequence generated by the application of $w$ to $s_i$ and $s_j$ are the same. At this point, the pair $\{s_i, s_j\}$ is neither separated nor merged. Any such pair may be eligible for separation by applying few more letters; however, Chassis (Algorithm 3) and its predecessors (Gill's and Gargantini's algorithms) ignores the progress achieved on these pairs and start over with a brand new sequence. This observation yielded us to create our second algorithm called Chassis-C. which is the "**C**hained" derivation of Chassis.

**Algorithm 4:** Chassis-C

```
input  : An FSM $M = (S, I, O, \delta, \lambda)$
output: A W–set $W$ for $M$
1  compute the shortest separating sequence $w_{\{i,j\}}$ for every state pair $s_i, s_j \in S$
2  $W = \emptyset$; // $W$: current W-set, initially empty
3  $Z = \{\{s_i, s_j\} \mid s_i \neq s_j, s_i, s_j \in S\}$; // $Z$: set of pairs yet to be separated
4  while $|Z| > 0$ do // we still have non-separated pairs, need to add more
     sequences to $W$
5      copy $Z$ into $\overline{Z}$
6      initialize $u = \varepsilon$
7      while $|\overline{Z}| > 0$ do
8          take a pair $\{s_i, s_j\} \in \overline{Z}$ such that $|w_{\{i,j\}}|$ is minimum amongst all pairs in $\overline{Z}$
9          $u = u.w_{\{i,j\}}$
10         $\overline{Z} = \{\{\delta(s'_i, w_{\{i,j\}}), \delta(s'_j, w_{\{i,j\}})\} \mid$
                   $\{s'_i, s'_j\} \in \overline{Z}$ , $s'_i$ and $s'_j$ are not separated nor merged by $w_{\{i,j\}}\}$
11     end
12     $Z = Z \setminus \{\{s'_i, s'_j\} \in Z \mid s'_i$ and $s'_j$ are separated by $u\}$
13     insert $u$ to $W$
14 end
```

In this algorithm, we again start with the standard step of shortest merging sequence calculation. Similarly to Algorithm 3 (Chassis), we initialize an empty W–set in the beginning denoted by $W$, and also set $Z$ which holds all the state pairs available: this set $Z$ indicates the pairs left to be separated. After the initial steps, the main loop starts just like Algorithm 3. Differently from it, Algorithm 4 (Chassis-C) does not create a new sequence every time it considers a shortest separating sequence of a state pair, but it creates W–set elements out of the concatenation of the several of these sequences.

In every iteration of the while loop in Line 4, we copy the currently active (non-separated) state pairs into a new set $\overline{Z}$, to be able to track them per iteration. We initialize an empty sequence $u$ which will hold the concatenation of several shortest merging sequences. The inner loop in Line 7 continues as long as we can find an unmerged and unseparated state pair left in $\overline{Z}$. The state pair $\{s_i, s_j\}$ in $\overline{Z}$ with the minimum $|w_{\{i,j\}}|$ is chosen in each iteration and $w_{\{i,j\}}$ is appended to $u$. The state pairs in $\overline{Z}$ that are merged or separated by $w_{\{i,j\}}$ are removed from $\overline{Z}$ and the other state pairs are considered to move with the input sequence $w_{\{i,j\}}$. The reason we remove merged pairs from $\overline{Z}$ is that they cannot be separated anymore by adding more letters naturally: such pairs has to be separated by another W–set sequence to be generated later on. Once we finish off all the state pairs in $\overline{Z}$, all the state pairs in $Z$ that are separated by $u$ are removed. The reason we remove separated pairs from $Z$ is that these pairs are not going to be needed to process in the rest of the algorithm anymore. The new sequence $u$ is ready and it is inserted into $W$, our current W–set.

**Lemma 2.** *The body of the loop on line 7 in Algorithm 4 is iterated at most $n-1$ times.*

*Proof.* During the process of Algorithm 4, each iteration of the loop on line 7 separates at least one state pair, hence two states from each other. Internally, states of the FSM are partitioned by the sequences formed so far, and the addition of a subsequence $w_{\{i,j\}}$ in an iteration has to increase the cardinality of this partition by at least one, because this subsequence separates a state pair. Having $n$ states, the cardinality of the partition grows up to $n$ during the lifetime of the algorithm. By this observation, the loop can iterate at most $n-1$ times. □

**Lemma 3.** *For a W–set $W = \{w_1, w_2, \ldots, w_k\}$ that Algorithm 4 generates,*

$$\sum_{i=1}^{k} |w_i| \leq (n-1)^2$$

*Proof.* Lemma 2 already shows that there can be at most $n-1$ iteration of the loop on line 7, each iteration of which adds a subsequence $w_{\{i,j\}}$ into the W–set. Each of these subsequences are taken from a shortest separating sequence of a state pair, whose length can be at most $n-1$ (Sandberg, 2004). Therefore, the total number of input letters in a W–set generated by Algorithm 4 can be at most $(n-1)^2$. □

Although the nested loop structure in Algorithm 4 seems to create a high running complexity, by using an amortized analysis, we show that it is not more expensive than Algorithm 3.

The precomputation part (line 1) has the time complexity of $O(pn^2)$ as mentioned earlier. The length of a separating sequence for a state pair in an FSM may be at most $n-1$ (Sandberg, 2004). The cardinality of $\overline{Z}$ may at most be $n \times (n-1)/2$ (which is the number of all state pairs). When we combine these two results, the cost of each execution of line 10 itself is $O(n^3)$. Lemma 2 implies that line 10 can be visited at most $n-1$ times. Therefore, the total cost of line 10 during the execution of Algorithm 4 is $O(n^4)$. Together with the precomputation, the total cost of Algorithm 4 is $O(pn^2+n^4)$. Similarly to Algorithm 3, Algorithm 4 performs much faster in practice than the theoretical complexity analysis shows, as well.

Once Chassis-C (Algorithm 4) is applied on the example FSM $M_1$ given in Figure 2.1, the W–set $\{aabba, bab\}$ is produced. We note that this result is better than Chassis in both departments, with cardinality of 2 and total letter count of 8.

## 4.3 The Algorithm: Chassis-CT

In Chassis-C (Algorithm 4), we create longer sequences by concatenating multiple shortest merging sequences of some state pairs. We prolong a sequence as long as we can find an available state pair (a pair that is not merged neither separated). During our manual testing of Chassis-C, we observed that we had very limited number of available state pairs in the last few steps of this sequence extension. Those few remaining state pairs might actually be separated by the sequences of the W–set that were generated later than the sequence of consideration. This observation gives the idea that last few extensions for the sequences might be redundant and hence unnecessary, when the sequences that are inserted later are considered. As a result, we see that it might be possible to apply some trimming to decrease the average length the sequences that Algorithm 4 presented in the previous section generates. This brings us to the following improved version of Chassis-C, named Chassis-CT (**T**rimmed derivation of Chassis-C):

**Algorithm 5:** Chassis-CT

```
input  : An FSM $M = (S, I, O, \delta, \lambda)$
output: A W–set $W$ for $M$
1 let $W = \{w_1, w_2, \ldots, w_k\}$[††] be a W–set computed for $M$ by Algorithm 4
2 for $i = k-1$ to 1 do // the last sequence cannot be trimmed
3     let $w'_i$ be the shortest prefix of $w_i$ such that $W \setminus \{w_i\} \cup \{w'_i\}$ is still a W–set
4     if $w'_i = \varepsilon$ then
5         $W = W \setminus \{w_i\}$
6     else
7         $W = W \setminus \{w_i\} \cup \{w'_i\}$
8     end
9 end
```

The implementation of the trimming, however, is more complicated. First, let us define $S^{\alpha}_{w_i}$, the set of states of pairs that an extension $\alpha$ of a sequence $w_i$ ”notes to separate”:

$S^{\alpha}_{w_i}$ is a set consisting of pairs of states $\{s, s'\} \in S^{\alpha}_{w_i}$ such that

- for all $j < i$, $\lambda(s, w_j) = \lambda(s', w_j)$ (i.e. no sequence that comes before $w_i$ can separate these pairs),
- $\lambda(s, w'_i\alpha) \neq \lambda(s', w'_i\alpha)$ (i.e. $w_i = w'_i\alpha$ separates these pairs),
- $\lambda(s, w'_i) = \lambda(s', w'_i)$ (i.e. the trimmed form $w'_i$ of $w_i$ cannot separate $s$ and $s'$).

In Algorithm 4, each extension of a sequence is noted to separate a set of state pairs as defined above. While trying to remove an extension $\alpha$ of a sequence $w_i \in W$, the set of state pairs $S^{\alpha}_{w_i}$ that were noted to be separated by $\alpha$ will no longer be separated by $w_i$. If some other set of sequences come out to separate all these pairs, then trimming $w_i$ by removing the extension $\alpha$ is safe. However, when $w_i$ is trimmed like this, the other sequences that guarantee the separability are not safe to change from thereafter. So, for each trimming, some sequences should be locked for future updates (trims). Because of this reason, the order of sequence selection becomes important.

In Algorithm 4, the generation method and order has some effect over this choice. Given two different sequences $w_i, w_j \in W$, if $w_i$ is generated before $w_j$ (i.e, $i < j$), we know that $w_i$ cannot separate any pair $w_j$ is noted to separate; because, this “noting to separate” notion is used for separating a state pair that is active (not separated by any earlier sequence). We can consequently say that, while trimming a sequence

[††]We consider the elements of $W$ as sorted with respect to the order of insertion of the sequences into $W$ by Algorithm 4. That is, we let $W = \{w_1, w_2, \ldots, w_k\}$ where $w_i$ is inserted into $W$ before $w_j$ if $i < j$.

$w_j$, any sequence $w_i$ such that $i < j$ does not need to be checked or locked. By this observation, we chose to traverse the sequences backwards, i.e. starting from the last generated one towards the first. By this choice, we make sure that we do not lock any sequence before it is processed for trimming. The last sequence $w_k$ cannot be trimmed as it does not have any successors to cover for the state pairs to be left uncovered.

**Lemma 4.** *The W–set $W$ that Algorithm 5 generates cannot be further trimmed.*

*Proof.* The proof is by construction. We may start by assuming that W–set $W$, which is the output of this algorithm, can be trimmed. This requires that some sequence $w_i \in W$ can be trimmed. Let us assume that $w_i = w_i'\alpha$ and $\alpha$ can be trimmed.

Note that, by definition of $S^{\alpha}_{w_i}$, no pairs of states $\{s, s'\} \in S^{\alpha}_{w_i}$ can be separated by $w_j$ for some $j < i$.

In addition, by the design of Algorithm 5, for at least one pair of states $\{s, s'\} \in S^{\alpha}_{w_i}$, there exists no $w_j$, $j > i$, that separates $\{s, s'\}$. Since, if there were such $w_j$, $j > i$, that separates $s$ and $s'$, Algorithm 5 would have trimmed $\alpha$ already and such $w_j$, $j > i$ did not get altered after the processing of $w_i$. □

The time complexity analysis of Algorithm 5 must definitely start by referring to the one of Algorithm 4. The time complexity of Algorithm 4 is found as $O(pn^2 + n^4)$ in Section 4.2. After the W–set is generated by Algorithm 4, we consider subsequences of W–set elements and check if the state pairs that these subsequences "note to separate" can be separated by the following W–set sequences. Each state pair in the FSM is noted to separate only once in a W–set. It means that we iterate over at most $n \times (n-1)/2$ state pairs, and each one only once. We may need to apply all the following sequences in the W–set to a state pair and each input letter applied adds up to the complexity. So, we need to analyze the total number of input letters that a W–set generated by Algorithm 4 may have. Though, Lemma 3 already proves that this number is upper-bounded by $(n-1)^2$. So, in Algorithm 5, each of our $n \times (n-1)/2$ many state pairs might be tested with a total of at most $(n-1)^2$ input letters. This shows that, the trimming part (line 2–9) of the algorithm has the time complexity of $O(n^4)$. By merging with the time complexity of Algorithm 4, which produces the W–set that gets trimmed, the total complexity of Algorithm 5 combined is $O(pn^2 + n^4)$.

Once Chassis-CT (Algorithm 5) is applied on the example FSM $M_1$ given in Figure 2.1, the W–set $\{aab, bab\}$ is produced. We note that Chassis-CT is indeed able to improve the W–set that is received from Chassis-C, by trimming the two letters *ba* from the end of the first sequence. With this, we note that Chassis-CT resulted in a W–set of cardinality 2 and total letter count 6.

## 4.4 The Algorithm: Chassis-P

This last W–set algorithm aims to generate a W–set that is suitable for using in K–tree generation. As explained in Chapter 2, a K–tree for an FSM $M$ can be generated from any W–set given for $M$. However, the properties of this W–set crucially affects the quality of the K-tree and therefore of the state identification sequence. For such reasons, generating a W–set with the shortest elements or the least number of elements are not a concern for the usage in K–tree based SIS methods. Indeed, even the inverse might be true. By this motivation, we designed the following W–set generation algorithm that mimics the process of a K–tree generation, called Chassis-P (**P**artitioned derivation of Chassis).

**Algorithm 6:** Chassis-P

```
input  : An FSM $M = (S, I, O, \delta, \lambda)$
output: A W–set $W$ for $M$
1  compute the shortest separating sequence $w_{\{i,j\}}$ for every state pair $s_i, s_j \in S$
2  $W = \emptyset$; // $W$: current W-set, initially empty
3  $\Pi = \{S\}$; // $\Pi$: state partition, initially all states are in a single block
4  while $\Pi$ has a non–singleton block do
5      take a non–singleton block $B \in \Pi$ and remove $B$ from $\Pi$
6      $\overline{Z} = \{\{s_i, s_j\} \mid s_i \neq s_j, s_i, s_j \in B\}$
7      initialize $u = \varepsilon$
8      while $|\overline{Z}| > 0$ do
9          take a pair $\{s_i, s_j\} \in \overline{Z}$ such that $|w_{\{i,j\}}|$ is minimum amongst all pairs in $\overline{Z}$
10         $u = u.w_{\{i,j\}}$
11         $\overline{Z} = \{\{\delta(s'_i, w_{\{i,j\}}), \delta(s'_j, w_{\{i,j\}})\} \mid$
                   $\{s'_i, s'_j\} \in \overline{Z}$, $s'_i$ and $s'_j$ are not separated nor merged by $w_{\{i,j\}}\}$
12     end
13     partition $B$ into blocks $B_1, B_2, \ldots, B_k$ such that
         $\forall s, s' \in B\, ; s, s' \in B_i$ for some $B_i \iff \lambda(s, u) = \lambda(s', u)$
14     $\Pi = \Pi \cup \{B_1, B_2, \ldots, B_k\}$
15     insert $u$ to $W$
16 end
```

In this algorithm, the first two lines are shared with all the previous algorithms proposed in this document so far. On line 3, we create a partition struct. A partition is basically a set of blocks, where blocks are a set of states. Initially, the partition $\Pi$ consists of a single block that contains all the states. Beginning from this point, we will *refine* $\Pi$ until all the states are single in a dedicated block.

In each iteration of the loop on line 4, the algorithm takes a non–singleton block $B$ from $\Pi$. Singleton blocks are already divided as much as they can, so they become out of consideration. The operations between line 5 to line 15 are done only for creating a sequence that divides the block $B$ to as many blocks as possible. The approach we use for generating this sequence is very similar to the one of Algorithm 4 (Chassis-C). In fact, lines 7-12 of Algorithm 6 are identical to the lines 6-11 of Algorithm 4. However in Algorithm 4, the state pair set $\overline{Z}$ is global to the execution and gets updated during the loop. But in Algorithm 6 (Chassis-P), we compile $\overline{Z}$ in every iteration from scratch and it consists of the pairs of all the states from $B$.

During the design of the algorithm, it was possible to choose an approach similar to the one of Gill's algorithm (or the base algorithm Chassis) by picking the separating sequence of a state pair and not extending it further. Considering the fact that we do not mind the cardinality of the W–set generated with Chassis-P, this seems like a good method. However, having a single sequence divide a group of states to a large number of group of states is preferable while constructing a K–tree, in order to minimize the height of K–tree. The height is the most crucial property of a K–tree in SIS generation; because the height of the K–tree directly affects the depth of recursion in the formula. As this W–set generation method is designed specifically for K–tree generation, we chose to use an approach similar to the one of Algorithm 4 which creates longer sequences, which should divide blocks into large number of blocks at once.

At the very end of each main loop iteration, the new sub-partitions of $B$ is calculated and added to the partition $\Pi$. Eventually, the main loop must terminate. This is guaranteed; because, we create a sequence consisting of at least one shortest separating sequence of some state pairs of $B$. This shortest separating sequence must put the two states that constitute the state pair into different blocks. This shows that the block $B$ must be divided into at least two blocks in each iteration and hence, all blocks in $\Pi$ must reach to the cardinality of 1 eventually.

Algorithm 6 analyzes state pairs by using partitions of states, differently than Algorithm 4. However, the worst time complexity still can be referred from Algorithm 4. The reason is the following: Algorithm 6's worst case does create totally unbalanced partitions, such as a partition with only two blocks one of which has only one state and the other with $k-1$ states. This way, the algorithm will have to process all the pairs generated by the $k-1$ states, which will lead to the greatest amount of pairs processed in total. In the worst case of Algorithm 4, the algorithm separates only one state from the others and all the other states stay together. This scenario is exactly the same scenario as the worst case of Algorithm 6. That's why, these two algorithms share the same worst case time complexity of $O(pn^2+n^4)$.

## 4.5 A Note on the Complexity of W–set Algorithms

Algorithms 3–6 all use state pairs for tracking the current separation situation. As we briefly mentioned in the beginning of this chapter, the same information can be encoded and tracked in state partitions. In several sections of this chapter, we noted that the concept of state pairs getting separated and removed from a waiting list is internally equivalent (for our purposes) to a set of states getting partitioned and some blocks becoming singletons (possessing a single state). Therefore, it is also possible to design and implement the above-mentioned algorithms by using state partitioning technique.

For a given set of states of size $k$, tracking the state pairs and applying inputs to those state pairs has the complexity of $O(k^2)$; because $k$ states make up $k\times(k-1)/2$ pairs. Each state occurs in $k-1$ state pairs and during the application of inputs to those pairs, each state is redundantly checked $k-1$ times. In the state partitioning technique, this extra theoretical cost is avoided. This is why the same logic that Algorithms 3–6 implements can be implemented in a time complexity of $O(pn^2+n^3)$ rather than $O(pn^2+n^4)$ that we showed above. However, in practice, at least for the size of FSMs used in our experiments, the performance of the algorithms are competitive enough. On top of this, using pairs decreases the implementation challenges, which made us choose this technique.

# 5. CONTRIBUTIONS ON K–TREE GENERATION

In the previous chapter, we introduced several methods to generate W–sets. W–sets are widely used for generating state identification sequences in FSM-based testing (Hennie, 1964; Kohavi, 1978; Kohavi et al., 1974; Lee & Yannakakis, 1996; Rezaki & Ural, 1995). The elements of W–sets are sequences and these sequences are used for identifying states. However, it is known that instead of applying a sequence directly, adaptivity can also be used, which yields to a tree structure known as PADS. When we have a set of PADSs instead of a set of sequences, this is called a K–set (Jourdan et al., 2016). Jourdan et al. (2016) also introduces the concept *K–tree*. K–tree is defined as a rooted tree where the root and intermediate nodes are labeled by the elements of K–set and the leaf nodes are labeled by the states of the FSM. A path from the root to a leaf in a K–tree gives a sequence of K–set elements that the leaf state can be located by. The K–tree concept has been shown to help create short SISs by Jourdan et al. (2016), but a clear method of K–tree generation has not been recommended: The way that PADSs should be placed and organized in the K–tree is left open. In this chapter, we propose several heuristic methods to generate efficient K–trees. Efficiency here is defined by how short state identification sequences are created from the K–trees. These K–tree generation methods can be applied on any W–set/K–set given. Originally, K–trees are known to process K–sets only, but it is possible to convert a W–set to K–set implicitly by representing a sequence $w = x_1x_2 \dots x_k$ as a PADS in the form of a complete tree, where every node at depth $i$ in this tree is labeled by the input letter $x_i$ in the sequence $w$. An example of this conversion is given in Figure 5.1. Hence, we say that a K–tree can be built by using a W–set as well.

We propose a greedy K–tree building heuristic in this section. The pseudo-code is given in Algorithm 7.

**Algorithm 7:** Our K–tree building algorithm

```
input  : An FSM $M = (S, I, O, \delta, \lambda)$, a K–set $W$
output: A K–tree $K$ for $M$
1  $r = (S, -)$ // $r$: root node, has all states and is initially unlabeled
2  initialize the K–tree $K$ with root $r$
3  initialize the working queue $Q = [r]$
4  while $|Q| > 0$ do
5      remove the next node $V = (S_V, \ell_V)$ from $Q$
6      foreach PADS $w_i$ in $W$ do
7          if $w_i$ does not occur on the path from $r$ to $V$ then
8              calculate the partition $\Pi_i = \{B_i^1, B_i^2, \ldots, B_i^k\}$ such that
                 $\forall s, s' \in S_V ; s, s' \in B_i^j$ for some $B_i^j \iff \lambda(s, w_i) = \lambda(s', w_i)$
9              calculate[††] the score $c_i$ of $\Pi_i$ induced by $w_i$
10         end
11     end
12     let $w_*$ be the PADS in $W$ that induces the partition $\Pi_* = \{B_*^1, B_*^2, \ldots, B_*^k\}$, which has
         the lowest score $c_*$
13     set label $\ell_V = w_*$
14     foreach $B_*^x \in \Pi_*$ do
15         if $|B_*^x| > 1$ then
16             create a new child $V^x = (B_*^x, -)$ of $V$
17             insert $V_x$ into $Q$
18         else // $|B_*^x| = 1$
19             create a new child $V^x = (B_*^x, s)$ where $s$ is only state in $B_*^x$
20         end
21     end
22 end
```

This algorithm constructs a K–tree in a breadth-first manner. In addition to the labels of K–tree nodes, Algorithm 7 also keeps track of a set of states corresponding to a K–tree node. Therefore, there are two pieces of information for each node tracked by Algorithm 7. The set of states $S_V$ and the label $\ell_V$ of a node $V$ is represented as the pair $(S_V, \ell_V)$ in Algorithm 7.

It starts with a single root node $r = (S, -)$, corresponding to the set of all states, where "$-$" indicates that the label for the node is not decided yet. In every step, a non-singleton node (i.e, a node having more than one state) is processed. This node is labeled by a PADS $w_i \in K$ and new children nodes are created depending on the partitioning induced by $w_i$. This PADS $w_i$ is chosen by using the score functions proposed in the upcoming sections. These score functions calculate a score called *unseparability index* on the partitions that the candidate K–set elements create. The

[††]The calculation here is done by using any of the score functions we introduce in the upcoming sections of this chapter.

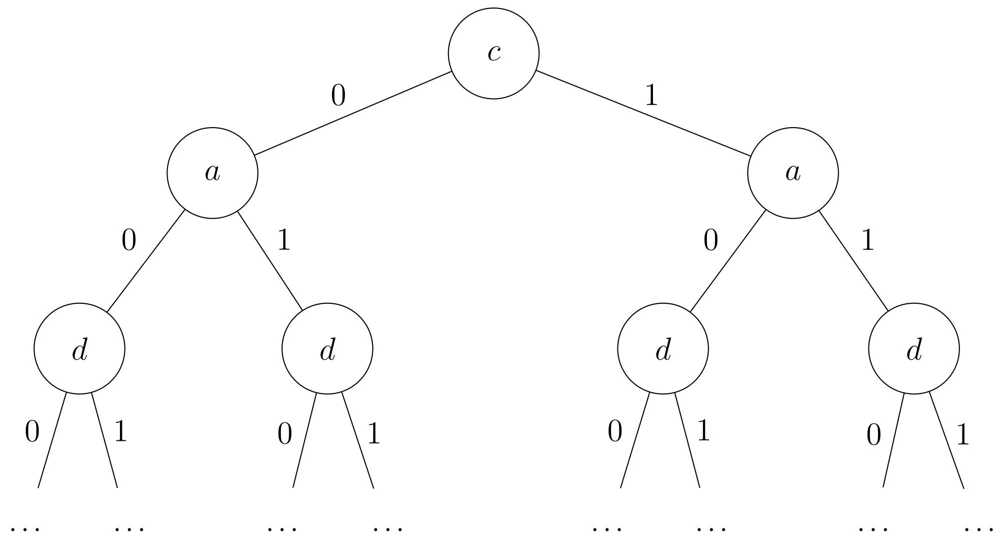

Figure 5.1 An example PADS that is equivalent to the sequence $w = cad$ where the output alphabet of the FSM is $O = \{0, 1\}$

K–set element that creates the partition with the lowest score is chosen. Beware that, by the definition of K–tree, having a K–set element twice on a path is redundant and hence unnecessary. Therefore, the algorithm considers the K–set elements that do not occur on the path from the root node to the current node, only. The process of dividing nodes into new children nodes continues until all leaf nodes are singletons.

The following sections propose the cost functions mentioned and explain the development processes of them. All these functions are defined with two parameters:

- $w$ : The PADS of consideration
- $\Pi = \{B_1, B_2, \ldots, B_k\}$ : The partition induced by $w$

The score calculated by the score functions can be called "the score of $w$" or "the score of $\Pi$". We use both notations in this document interchangeably. The experimental comparisons of these score functions are given in Chapter 7.

## 5.1 Cost Function on Partition Cardinality

There are only few parameters to work with in this problem at hand. One of the primitive ways to approach this problem is considering the number of blocks the partition $\Pi$ has. In the optimal case, we would want to have a partition that has singleton blocks only, hence being a perfect division. In this case, we could call $w$ "an ADS for the given set of states", which can separate all the possible state pairs single-handedly. In practice, running into such a partition is not probable in high state counts. However, this example is still good for showing that high block count is a desired property in a partition for our purpose.

Finally, we can conclude that the cardinality of $\Pi$ can be used within the score calculation. At this point, we would like to remind that the lower score should be better in the score functions we design, due to the way we chose to implement Algorithm 7. Because our observations show that high block count is desired, the value of the score function must be *inversely proportional* to the cardinality of $\Pi$.

As a result of all these conclusions drawn, we propose the cost function $\Phi_{car}$ as the following:

$$\Phi_{car}(w, \Pi) = \frac{1}{|\Pi|} \tag{5.1}$$

## 5.2 Cost Function on Largest Block Size

The cost function $\Phi_{CAR}$, which is introduced in the previous section, indicates a good and simple measure of how well a state set is partitioned. If the number of disconnected state sets (i.e, blocks) is higher, we can consider it a more successfully refined partition. However, there are some edge cases which this simple approach cannot cover. The most obvious unwanted case is when there are many small blocks and one huge block.

As an example, let us think two partitions $\Pi_1$ and $\Pi_2$, where the cardinalities of the blocks of $\Pi_1$ go as $6-6-2-2$ and the cardinalities of the blocks of $\Pi_2$ go as $1-1-1-1-12$. In this case, if we use $\Phi_{CAR}$ as our cost function, $c_1$ gets calculated as $\frac{1}{4}$ and $c_2$ gets calculated as $\frac{1}{5}$. As $c_2 < c_1$ , Algorithm 7 picks $\Pi_2$. As Jourdan et al. (2016) explains, the length of the SISs derived from a K–tree for a state grows *exponentially* with the depth of that state in the K–tree. Due to this, isolating few states in one step at the expense of leaving many states in the same block may not be desirable. By this observation, we decided to work in the direction to improve the worst case created by the previous cost function ($\Phi_{car}$).

The worst case (i.e. the longest SIS) will be due to the largest block size in the partitioning, due the exponential dependence between the length of the state identification sequence and the cardinality in the set of states. Therefore in order to improve the worst case, one has to consider minimizing the largest cardinality block. As Algorithm 7 chooses the alternative with a smallest cost, we can consider the cost of a partitioning $\Pi$ as the size of the largest block in $\Pi$ as follows:

$$\Phi_{lbs}(w,\Pi) = \max_{B \in \Pi} \{|B|\} \tag{5.2}$$

Although not strictly enforcing it, $\Phi_{lbs}$ also aims to provide that the cardinality of $\Pi$ is large. It is easy to see that, given a partition $\Pi$ with a total of $Y$ states, $\frac{Y}{\Phi_{lbs}(\Pi)}$ is a lower bound for the number of blocks in this partition $\Pi$. Thanks to this property of $\Phi_{lbs}$, it also bears some functionality of $\Phi_{car}$.

## 5.3 Cost Function on Sum-Squares of Block Sizes

From the previous sections, it is obvious that $\Phi_{lbs}$ is a good candidate that performs well in theory and it also covers for the pitfalls of $\Phi_{car}$. However, $\Phi_{lbs}$ has no mechanism to check beyond the biggest block. We may extend the example from the previous section, such that $\Pi_1$ has blocks with cardinalities $6-6-2-2$, $\Pi_2$ with $1-1-1-1-12$ and also $\Pi_3$ with $6-4-3-3$. While using $\Phi_{lbs}$ as the score function, Algorithm 7 chooses either $\Pi_1$ or $\Pi_3$, because both have the same score of 6. However, $\Pi_3$ is definitely better than $\Pi_1$, as $\Pi_3$ has a better distribution of block sizes when the largest is omitted. The problem grows even larger once we add $\Pi_4$ with the block sizes $6-1-1-1-1-1-1-1-1-1-1$. $\Pi_4$ is the best partitioning that one can reach with a $\Phi_{lbs}$ score of 6. $\Pi_4$ has the same $\Phi_{lbs}$ score with $\Pi_1$ and $\Pi_3$, although $\Pi_4$ is way better than $\Pi_1$ and $\Pi_3$. $\Phi_{car}$ seems to be able to detect that $\Pi_4$ is the best partition in this scenario, but it is already known to have some other issues, as explained in Section 5.2.

From the discussion above, it is obvious that we need a solution where the number of blocks in the partition is preferred to be as high as possible, and the block sizes are preferred to be distributed as evenly balanced as possible. Mathematically, this can be accomplished by applying a sum-square approach, as follows:

$$\Phi_{ssq}(w, \Pi) = \sum_{B \in \Pi} |B|^2 \tag{5.3}$$

Consider a set $S' \subseteq S$ of states and a partitioning $\Pi$ for $S'$, where $k = |\Pi|$ is the number of blocks in $\Pi$. The minimum (hence the best) score from $\Phi_{ssq}$ can be achieved when every block $B \in \Pi$ has exactly $|S'|/k$ states. So, we can tell that $\Phi_{ssq}$ satisfies the even distribution requirement.

From the aspect of the other requirement, which is optimizing for the number of blocks, $\Phi_{ssq}$ also calculates lower scores for the partitions that have higher block counts. Because, all the block cardinalities are squared then summed, and the sum-square of many small numbers is smaller than the sum-square of few greater numbers. Consequently, we may argue that this score function is competent in terms of both requirements we described above.

## 5.4 Cost Function on Sum-Squares of Block Sizes and Height of PADS

The sum-square approach that $\Phi_{ssq}$ uses gives pretty good results as we lay further in Chapter 7. Yet, there is one improvement we propose for $\Phi_{ssq}$. As one can notice, we declared the generic $\Phi$ function with two parameters: $w$ - the PADS that causes the partitioning and $\Pi$ - the partition. But, the three functions we proposed until this point do not utilize the characteristics of $w$, ever.

State identification sequences (SIS) are calculated for a state from a given K–tree. The path from the root of the K–tree to the leaf node labeled by the state is considered for this calculation. The PADSs labeling the nodes on this path are used to construct the SIS. Therefore, the heights of the PADSs used in the construction affect the length of the SISs. Hence, minimizing the heights of PADSs selected during K–tree generation might become another objective of our score function. However, by both intuition and experimenting, we realised that the distribution of the partition is still a lot more important than the heights of PADSs. In other words, minimizing the magnitude of the sum-square in the score must be the main objective. Only when two alternatives have the same sum-square value, one can consider minimizing the heights of the PADSs.

$$\Phi_{ssh}(w,\Pi) = z \times (\underbrace{\sum_{B \in \Pi} |B|^2}_{\Phi_{ssq}}) + h(w) \tag{5.4}$$

The score function $\Phi_{ssh}$ in Equation 5.4 combines these two objectives. The notation $h(w)$ in this equation denotes the height of the PADS $w$. The sum-square component (which is the result of $\Phi_{ssq}$ at the same time) is multiplied by a constant $z$. This constant $z$ is used in order to prioritize the sum-square component, as reasoned in the previous paragraph. This constant $z$ must be selected large enough to make sure that even a smallest decrease in the sum-square component is prioritized over the largest possible decrease in the PADS height component.

# 6. COMPETITIVE W–SET EXPERIMENTS

In Chapter 4, we presented four new algorithms for generating W–sets for a given FSM. These algorithms go by names Chassis (Algorithm 3), Chassis-C (Algorithm 4), Chassis-CT (Algorithm 5) and Chassis-P (Algorithm 6). The first three of these are designed to be used for general purposes. Hence, they try to minimize the cardinality and total number of input letters for the W–sets. The last of them (Chassis-P) does not aim any minimization in these two parameters, as we explained in Chapter 4. Because of this, in this chapter, we present the results of experiments that we made on Chassis, Chassis-C and Chassis-CT, but not Chassis-P. We give comparative results with many different FSM configurations. As a baseline, we also implemented the W–set algorithm from Gill (1962) and the W–set algorithm from Bulut et al. (2019). These two algorithms will be referred as Gill and BJT in the rest of this document. However, please note that we could not run BJT at configurations higher than 512 states due to the exponential time complexity of this algorithm with respect to $n$ (Bulut et al., 2019).

The experiment in this chapter is performed on a machine running on 64 bit Ubuntu 16.04.7 equipped with 16GB RAM and an Intel Xeon E5-1650 clocked at 3.20GHz. All the algorithms are implemented in C++ and compiled with `gcc 5.4.0`. The standard flag `-std=c++0x`, the optimization flag `-O3` and the release mode flag `-DNDEBUG` are used for compilation.

We determined a total of 54 different configurations $(n,p,q)$, where the number of states $n \in \{128, 256, 512, 1024, 2048, 4096\}$, the number of input letters $p \in \{2, 16, 128\}$ and the number of output letters $q \in \{2, 16, 128\}$. For each configuration, we generated 500 random FSMs. The randomness of the FSMs is achieved by setting each $\delta(s,i)$ to a uniformly random state from $S$ and each $\lambda(s,i)$ to a uniformly random output letter from $O$.

The table below shows the results of the W–set experiments. For each configuration and algorithm, average cardinality and total number of input letters of the W–set are given, along with the average run-time of the algorithms in microseconds.

| | | | BJT | | | Gill | | | Chassis | | | Chassis-C | | | Chassis-CT | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ | $p$ | $q$ | Crd | Tlen | $t(\mu s)$ | Crd | Tlen | $t(\mu s)$ | Crd | Tlen | $t(\mu s)$ | Crd | Tlen | $t(\mu s)$ | Crd | Tlen | $t(\mu s)$ |
| 128 | 2 | 2 | 7.00 | 21.68 | 2253 | 7.75 | 24.16 | 159 | 6.67 | 20.84 | 353 | 2.25 | 18.23 | 368 | 2.25 | 14.94 | 382 |
| | | 16 | 2.29 | 3.97 | 2650 | 2.16 | 4.28 | 61 | 2.09 | 4.05 | 100 | 1.99 | 5.07 | 109 | 1.99 | 3.97 | 125 |
| | | 128 | 2.00 | 2.39 | 2770 | 2.00 | 2.41 | 67 | 2.00 | 2.39 | 61 | 1.39 | 2.80 | 66 | 1.39 | 2.38 | 77 |
| | 16 | 2 | 15.65 | 18.60 | 2373 | 17.00 | 26.93 | 160 | 12.88 | 16.98 | 288 | 2.39 | 17.09 | 329 | 2.39 | 14.13 | 341 |
| | | 16 | 4.84 | 4.95 | 2824 | 7.31 | 7.43 | 59 | 5.68 | 5.79 | 105 | 1.99 | 5.03 | 105 | 1.99 | 3.97 | 117 |
| | | 128 | 2.25 | 2.25 | 2850 | 2.39 | 2.39 | 65 | 2.38 | 2.38 | 58 | 1.39 | 2.80 | 64 | 1.39 | 2.38 | 75 |
| | 128 | 2 | 17.45 | 17.87 | 5657 | 26.81 | 27.27 | 150 | 18.82 | 19.26 | 283 | 2.39 | 17.08 | 312 | 2.39 | 14.16 | 327 |
| | | 16 | 3.83 | 3.83 | 4478 | 7.33 | 7.33 | 58 | 5.70 | 5.70 | 101 | 1.99 | 5.03 | 99 | 1.99 | 3.97 | 110 |
| | | 128 | 2.26 | 2.26 | 3682 | 2.39 | 2.39 | 63 | 2.38 | 2.38 | 56 | 1.39 | 2.80 | 61 | 1.39 | 2.38 | 74 |
| 256 | 2 | 2 | 7.94 | 25.25 | 39985 | 8.23 | 27.24 | 338 | 7.58 | 24.67 | 1375 | 2.30 | 21.58 | 1338 | 2.30 | 17.16 | 1356 |
| | | 16 | 2.57 | 4.94 | 56049 | 2.50 | 4.98 | 139 | 2.39 | 4.78 | 438 | 2.00 | 5.78 | 481 | 2.00 | 4.44 | 469 |
| | | 128 | 2.00 | 2.85 | 58864 | 2.00 | 3.22 | 160 | 2.00 | 2.87 | 302 | 1.64 | 3.55 | 347 | 1.64 | 2.86 | 356 |
| | 16 | 2 | 16.71 | 22.30 | 40202 | 17.60 | 33.34 | 359 | 13.85 | 20.26 | 1236 | 2.32 | 20.20 | 1294 | 2.32 | 16.44 | 1316 |
| | | 16 | 6.41 | 6.63 | 56821 | 9.85 | 10.11 | 142 | 7.25 | 7.47 | 510 | 2.00 | 5.74 | 464 | 2.00 | 4.43 | 445 |
| | | 128 | 2.90 | 2.90 | 59406 | 3.30 | 3.30 | 160 | 3.11 | 3.11 | 298 | 1.64 | 3.55 | 345 | 1.64 | 2.86 | 348 |
| | 128 | 2 | 44.55 | 45.55 | 51496 | 50.57 | 54.75 | 401 | 28.92 | 31.14 | 1616 | 2.32 | 20.17 | 1262 | 2.32 | 16.40 | 1283 |
| | | 16 | 4.27 | 4.27 | 63808 | 9.94 | 9.94 | 158 | 7.34 | 7.34 | 541 | 2.00 | 5.74 | 476 | 2.00 | 4.43 | 475 |
| | | 128 | 2.79 | 2.79 | 63256 | 3.30 | 3.30 | 171 | 3.11 | 3.11 | 302 | 1.64 | 3.55 | 307 | 1.64 | 2.86 | 332 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 512 | 2 | 2 | 8.22 | 27.62 | 674591 | 8.29 | 30.70 | 877 | 7.99 | 27.51 | 6147 | 2.30 | 24.41 | 5563 | 2.30 | 19.19 | 5651 |
|  |  | 16 | 3.01 | 6.01 | 960855 | 3.27 | 6.56 | 370 | 2.96 | 5.93 | 1909 | 2.00 | 6.56 | 1960 | 2.00 | 5.00 | 2009 |
|  |  | 128 | 2.00 | 3.00 | 1014089 | 2.00 | 3.94 | 395 | 2.00 | 3.06 | 1294 | 1.87 | 3.92 | 1519 | 1.87 | 3.06 | 1568 |
|  | 16 | 2 | 17.74 | 26.39 | 683222 | 18.02 | 34.03 | 980 | 14.40 | 23.29 | 5688 | 2.35 | 23.17 | 5496 | 2.35 | 18.77 | 5552 |
|  |  | 16 | 11.27 | 12.27 | 963605 | 14.60 | 19.76 | 420 | 7.99 | 9.27 | 2552 | 2.00 | 6.52 | 1953 | 2.00 | 5.00 | 2000 |
|  |  | 128 | 3.34 | 3.34 | 1081518 | 4.60 | 4.60 | 447 | 4.04 | 4.04 | 1532 | 1.87 | 3.92 | 1670 | 1.87 | 3.06 | 1741 |
|  | 128 | 2 | 63.20 | 67.32 | 732090 | 88.60 | 133.57 | 1149 | 33.26 | 39.02 | 8166 | 2.35 | 23.16 | 5326 | 2.35 | 18.74 | 5369 |
|  |  | 16 | 15.61 | 16.10 | 999389 | 18.30 | 18.97 | 445 | 12.49 | 12.98 | 2938 | 2.00 | 6.52 | 1992 | 2.00 | 5.00 | 2012 |
|  |  | 128 | 3.08 | 3.08 | 1035910 | 4.60 | 4.60 | 412 | 4.04 | 4.04 | 1466 | 1.87 | 3.92 | 1584 | 1.87 | 3.06 | 1594 |
| 1024 | 2 | 2 | – | – | – | 8.50 | 33.78 | 2703 | 8.21 | 30.02 | 29017 | 2.32 | 27.46 | 24613 | 2.32 | 21.34 | 25150 |
|  |  | 16 | – | – | – | 3.97 | 7.98 | 1085 | 3.38 | 6.80 | 8562 | 2.00 | 7.33 | 9504 | 2.00 | 5.43 | 9564 |
|  |  | 128 | – | – | – | 2.01 | 4.01 | 1027 | 2.00 | 3.21 | 5773 | 1.98 | 4.23 | 7760 | 1.98 | 3.22 | 7811 |
|  | 16 | 2 | – | – | – | 18.85 | 34.90 | 2864 | 15.06 | 26.60 | 27916 | 2.39 | 26.13 | 24342 | 2.39 | 20.72 | 25103 |
|  |  | 16 | – | – | – | 18.96 | 34.05 | 1283 | 8.44 | 10.54 | 11330 | 2.00 | 7.31 | 9431 | 2.00 | 5.42 | 9499 |
|  |  | 128 | – | – | – | 6.63 | 6.93 | 1013 | 5.57 | 5.87 | 6756 | 1.98 | 4.22 | 7641 | 1.98 | 3.21 | 7631 |
|  | 128 | 2 | – | – | – | 128.01 | 253.46 | 3278 | 29.43 | 39.11 | 34535 | 2.38 | 26.11 | 24636 | 2.38 | 20.77 | 25307 |
|  |  | 16 | – | – | – | 39.36 | 52.66 | 1331 | 12.98 | 14.60 | 13988 | 2.00 | 7.31 | 9463 | 2.00 | 5.42 | 9545 |
|  |  | 128 | – | – | – | 6.71 | 6.94 | 1018 | 5.64 | 5.88 | 6716 | 1.98 | 4.22 | 7513 | 1.98 | 3.21 | 7547 |

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2048 | 2 | 2 | – | – | – | 9.11 | 36.38 | 8587 | 8.68 | 33.53 | 167172 | 2.35 | 30.58 | 124961 | 2.34 | 23.39 | 126727 |
| | | 16 | – | – | – | 4.01 | 8.25 | 3366 | 3.85 | 7.92 | 41677 | 2.00 | 8.05 | 46237 | 2.00 | 5.99 | 46368 |
| | | 128 | – | – | – | 2.01 | 4.02 | 3059 | 2.01 | 3.66 | 27180 | 2.00 | 4.70 | 32602 | 2.00 | 3.66 | 33160 |
| | 16 | 2 | – | – | – | 19.12 | 35.37 | 9362 | 15.97 | 29.77 | 148622 | 2.37 | 29.14 | 123177 | 2.37 | 22.92 | 124318 |
| | | 16 | – | – | – | 18.19 | 34.19 | 4265 | 10.11 | 12.73 | 58263 | 2.00 | 8.04 | 45522 | 2.00 | 5.98 | 45463 |
| | | 128 | – | – | – | 12.55 | 15.16 | 3005 | 6.53 | 7.50 | 36178 | 2.00 | 4.70 | 31436 | 2.00 | 3.66 | 32350 |
| | 128 | 2 | – | – | – | 128.65 | 256.65 | 10087 | 25.08 | 38.88 | 149263 | 2.37 | 29.14 | 123467 | 2.37 | 22.91 | 124141 |
| | | 16 | – | – | – | 104.06 | 204.94 | 4538 | 11.35 | 13.94 | 59930 | 2.00 | 8.04 | 46804 | 2.00 | 5.98 | 46781 |
| | | 128 | – | – | – | 13.98 | 16.79 | 3018 | 6.85 | 7.83 | 36256 | 2.00 | 4.70 | 31607 | 2.00 | 3.66 | 32472 |
| 4096 | 2 | 2 | – | – | – | 10.73 | 42.89 | 29993 | 9.76 | 38.70 | 835729 | 2.34 | 33.62 | 649338 | 2.33 | 25.25 | 652553 |
| | | 16 | – | – | – | 4.02 | 9.69 | 12109 | 3.89 | 8.80 | 190763 | 2.00 | 8.74 | 222821 | 2.00 | 6.47 | 222886 |
| | | 128 | – | – | – | 2.05 | 4.10 | 10271 | 2.04 | 4.06 | 119195 | 2.00 | 5.06 | 157004 | 2.00 | 4.00 | 157079 |
| | 16 | 2 | – | – | – | 20.41 | 40.01 | 32970 | 17.81 | 35.54 | 662494 | 2.51 | 32.59 | 641890 | 2.50 | 24.89 | 645212 |
| | | 16 | – | – | – | 16.67 | 32.67 | 13211 | 8.25 | 11.85 | 220593 | 2.00 | 8.72 | 219938 | 2.00 | 6.48 | 219987 |
| | | 128 | – | – | – | 16.44 | 32.02 | 10240 | 5.57 | 6.91 | 142010 | 2.00 | 5.06 | 155354 | 2.00 | 4.00 | 155261 |
| | 128 | 2 | – | – | – | 128.20 | 256.20 | 34730 | 23.40 | 41.11 | 665746 | 2.51 | 32.59 | 648834 | 2.50 | 24.89 | 649394 |
| | | 16 | – | – | – | 128.09 | 256.06 | 13803 | 9.22 | 12.86 | 219487 | 2.00 | 8.72 | 218505 | 2.00 | 6.48 | 218668 |
| | | 128 | – | – | – | 51.93 | 99.17 | 10373 | 5.89 | 7.26 | 143043 | 2.00 | 5.06 | 154178 | 2.00 | 4.00 | 154194 |

Table 6.1 Quality and run-time results of W–set algorithms

In Table 6.1, cells corresponding to the algorithm BJT with $n > 512$ are left blank. Because, due to the exponential time complexity, we could not afford to test BJT beyond state size of 512. The new algorithms that we propose, namely Chassis, Chassis-C and Chassis-CT all perform slower than Gill and faster than BJT as the results indicate. These experimental time results are in correlation with our expectations; as BJT works in exponential time complexity and Gill is implemented in a much less complex way (i.e. by tracking sets of states, instead of sets of pairs of states) than our algorithms. Please recall that, Chassis, Chassis-C and Chassis-CT can be altered to run faster, as we noted on Section 4.5. All three of our algorithms run in similar speeds where the fastest is Chassis-C. The fact that Chassis-CT takes nearly the same time as Chassis-C is a sensible result; because Chassis-CT is identical to Chassis-C apart from the trimming process. This trimming process takes quite an insignificant time, thanks to the shallow W–sets Chassis-C is capable of producing. We even observe smaller times in Chassis-CT than Chassis-C occasionally, due to small measurement errors.

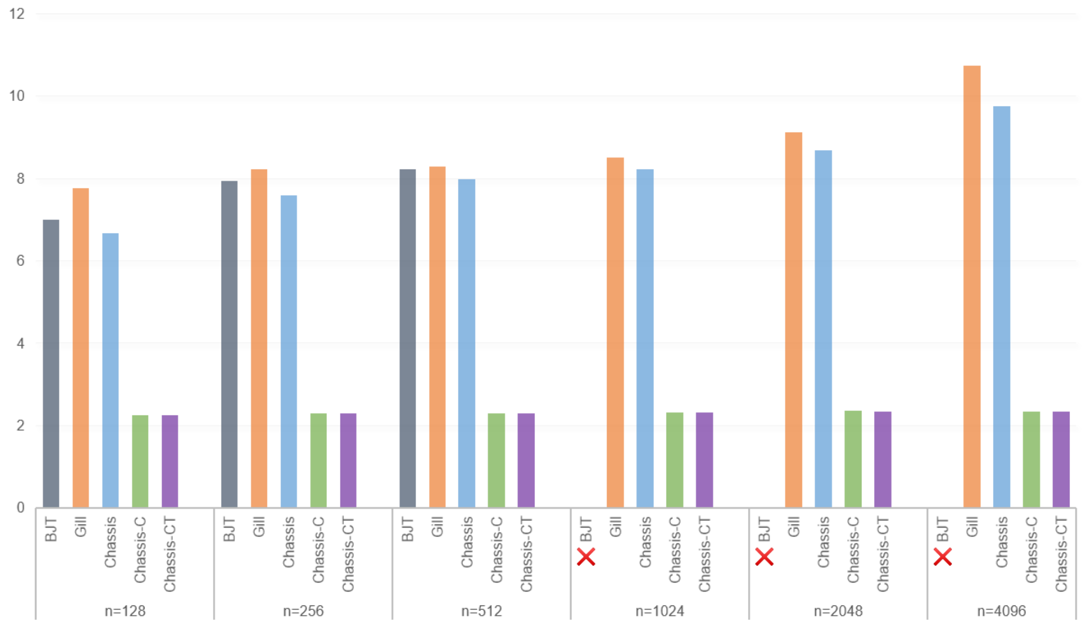

Figure 6.1 Average cardinality of the W–sets that each algorithm generates, for $p = 2$ and $q = 2$

In terms of quality (e.g, cardinality and total number of input letters in W–sets), the performances of our algorithms Chassis, Chassis-C and Chassis-CT show an increasing pattern; such that Chassis-CT is the best among three and Chassis is the worst. This is the expected result considering the development order and the analysis power of the algorithms.

Figures 6.1–6.4 summarize the quality results of the experiments we made on the W–set algorithms. For $p = q = 2$, our algorithms, especially Chassis-C and Chassis-

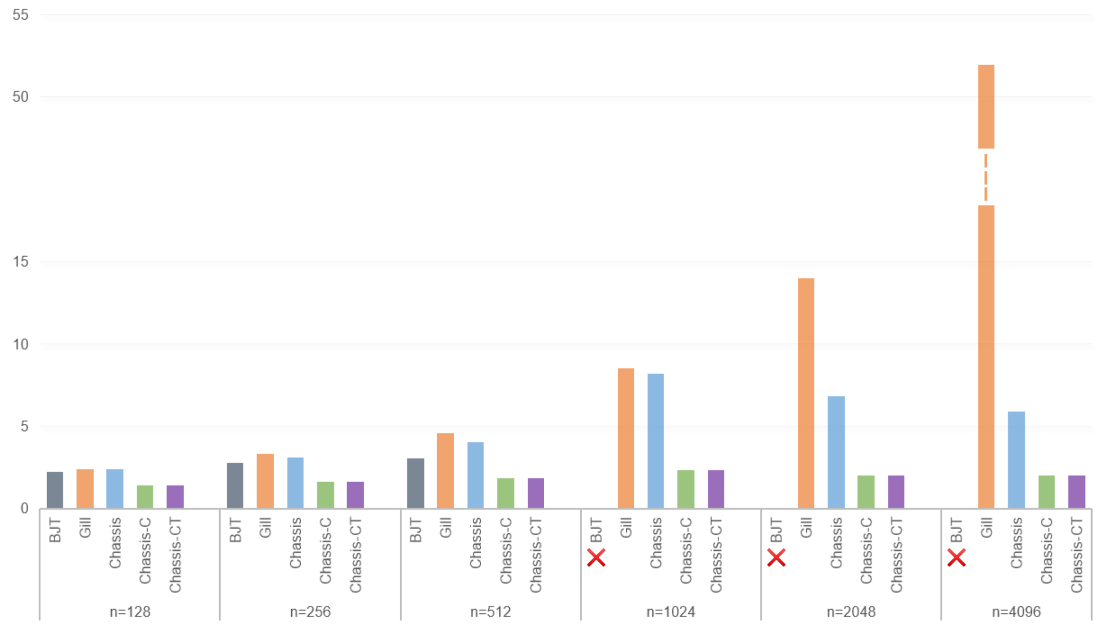

Figure 6.2 Average cardinality of the W–sets that each algorithm generates, for $p = 128$ and $q = 128$

CT, generate better W–sets in terms of both cardinality and total number of input letters. Even though CHASSIS-C and CHASSIS-CT might find worse W-sets for some small number of FSMs, they are much better than the other algorithms in general. CHASSIS also seems to present better results than GILL and comparable results to BJT. On the most extreme case $p = q = 128$, algorithms show similar quality on small state sizes. When $p$ and $q$ are as large as this, separability is quite common and finding small W–sets is of high possibility. This is the reason why all algorithms produce good results and become indistinguishable for $n = 128, p = q = 128$. However, as state size grows, CHASSIS-CT and BJT become prominent. This is an expected result for BJT, as it works in a breadth first manner for finding the shortest sequences in all cases. Still, CHASSIS-CT is able to produce competitive results against BJT, which soon becomes practically infeasible due to the exponential time complexity. Consequently, CHASSIS-CT becomes the only algorithm that produces results of high quality by maintaining practical feasibility in high state sizes.

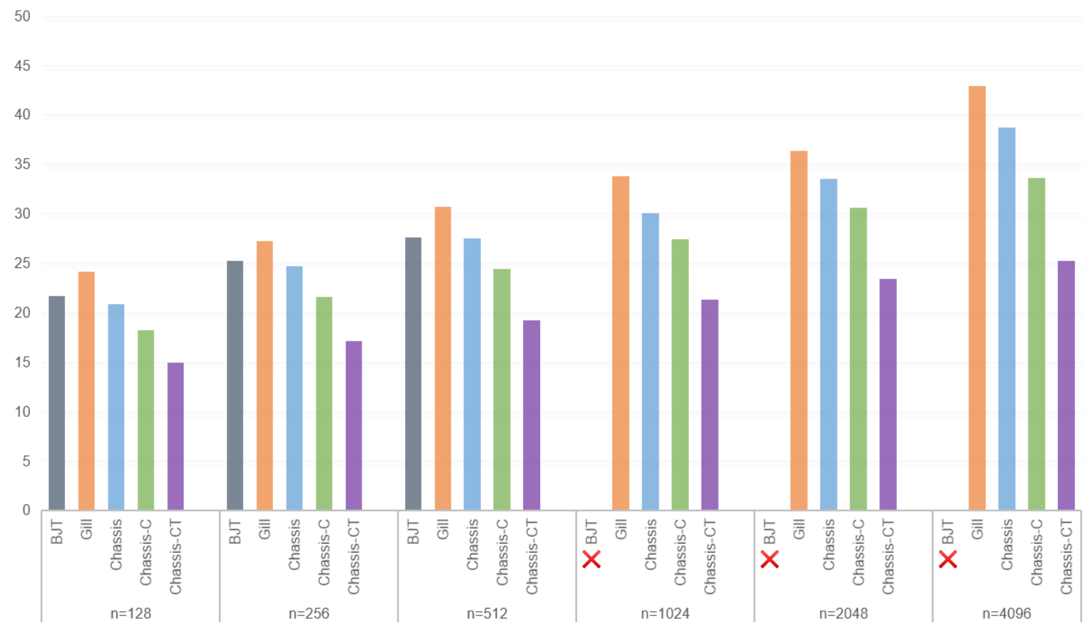

Figure 6.3 Average total number of input letters of the W–sets that each algorithm generates, for $p = 2$ and $q = 2$

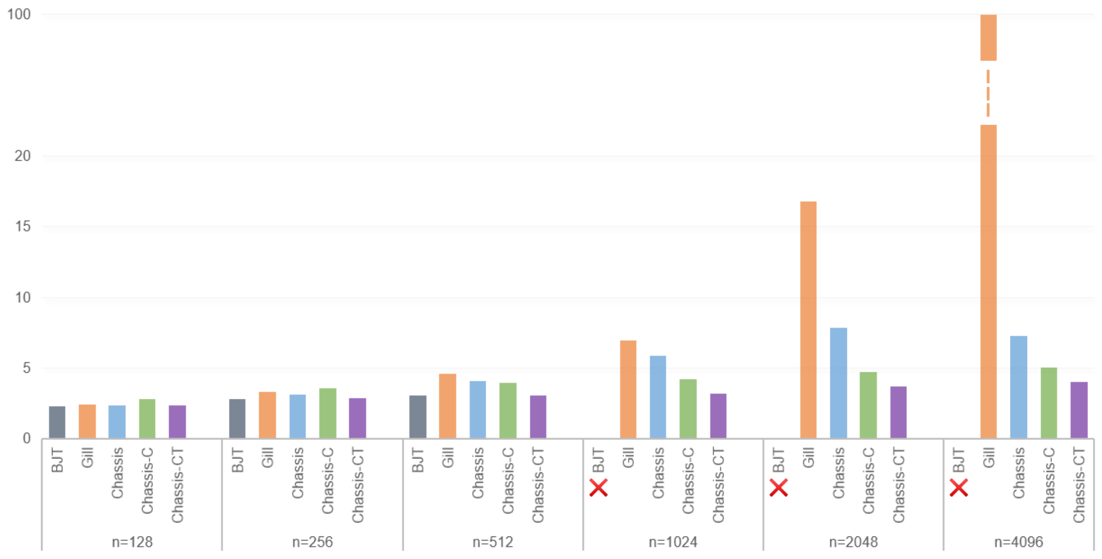

Figure 6.4 Average total number of input letters of the W–sets that each algorithm generates, for $p = 128$ and $q = 128$

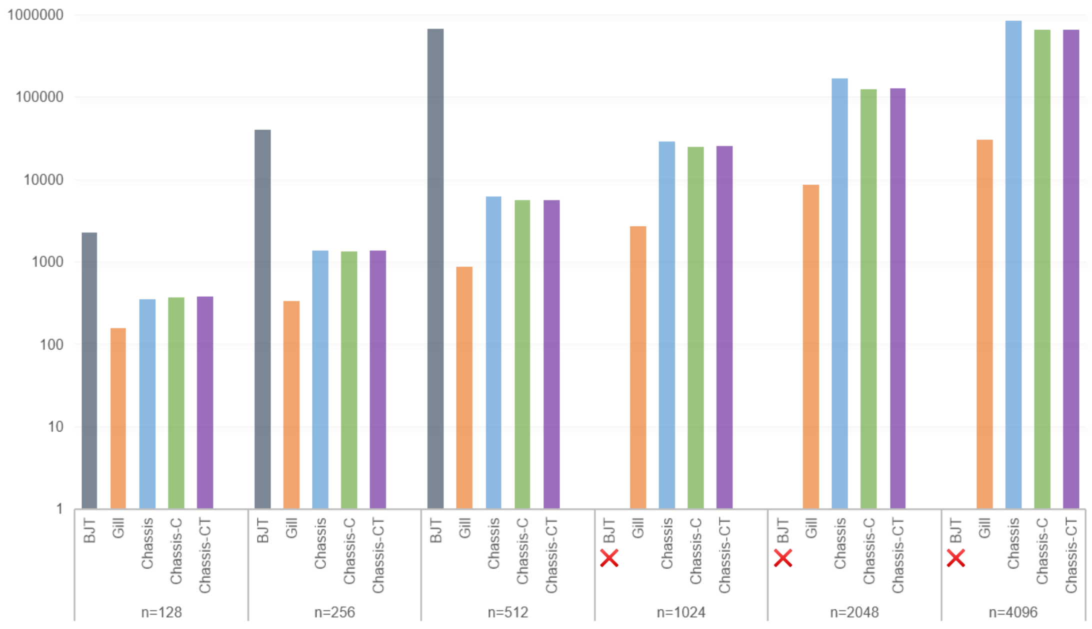

Figure 6.5 Average running time (in $\mu$s) of each W–set algorithm given in logarithmic scale, for $p = 2$ and $q = 2$

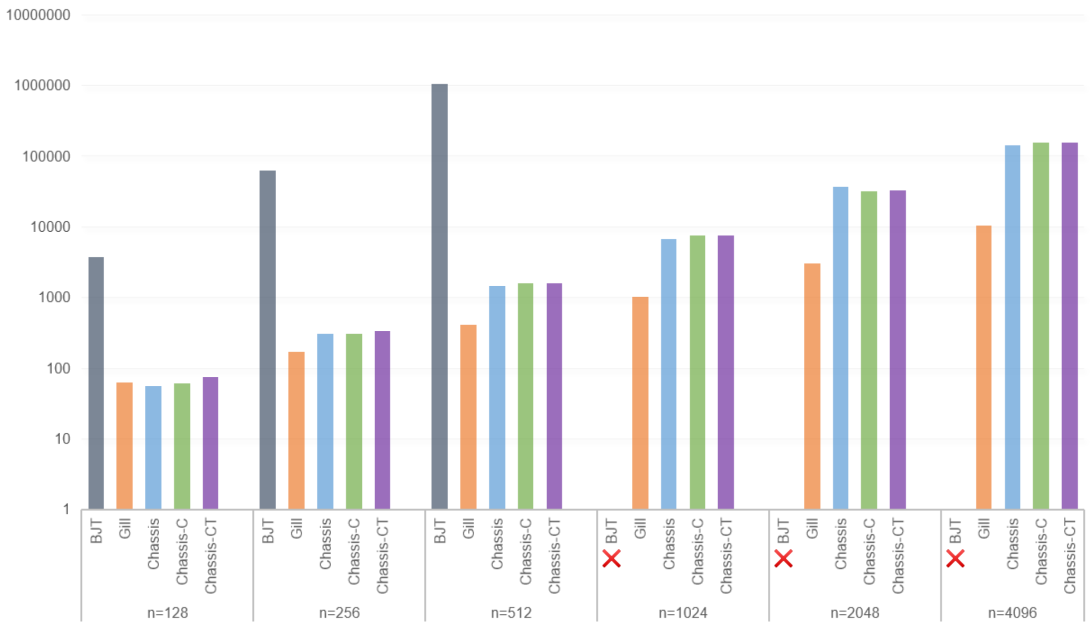

Figure 6.6 Average running time (in $\mu$s) of each W–set algorithm given in logarithmic scale, for $p = 128$ and $q = 128$

# 7. K–TREE AND STATE IDENTIFICATION SEQUENCE EXPERIMENTS

In Chapter 6, we objectively analyzed our new W–set algorithms and compared them to two algorithms from the literature. In these comparisons, we did not consider them in the K–tree generation context; they were compared in their generic features (i.e, total number of input letters and cardinality). In this chapter, we focus on experimenting the work that we did on K–tree generation. We also make a comparative study where the K–tree based SIS generation techniques are compared to other W–set based SIS generation techniques.

The experiments in this chapter are performed on a machine running on 64 bit Ubuntu 14.04.6 equipped with 128GB RAM and an Intel Xeon E5-2690 clocked at 2.90GHz. The programs are implemented in C++ and compiled with `gcc 4.8.4`. The standard flag `-std=c++0x`, the optimization flag `-O3` and the release mode flag `-DNDEBUG` are used for all compilations.

For all experiments in this chapter, we again used random FSMs. The FSM configurations that we use are the following: the number of states $n \in \{128, 256, 512, 1024, 2048, 4096\}$, the number of input letters $p \in \{2, 16, 128\}$ and the number of output letters $q \in \{2, 16, 128\}$. For each configuration, we generated 500 random FSMs. The randomness of the FSMs is achieved by setting each $\delta(s, i)$ to a uniformly random state from $S$ and each $\lambda(s, i)$ to a uniformly random output letter from $O$.

## 7.1 Score Function Experiments

We introduced Algorithm 7 for K–tree generation in Chapter 5. In this algorithm, a cost function must be used for deciding which W–set/K–set element should be picked for a particular K–tree node. In Sections 5.1–5.4, we proposed four score functions to be used within Algorithm 7. These score functions aim to assess the quality of a candidate partition that might be created for the K–tree node of consideration.

For the upcoming experiments, we needed to choose one of these score functions. Because, there are a huge amount of different FSM configurations and several SIS generation methods to test. In this case, repeating all experiments four times for each score function would be infeasible. Hereby, we designed an intermediate experiment for finding the most effective score function and used only the winner function in the subsequent experiments. The result of this experiment is given in this section.

The score functions in this experiment are compared by the average SIS length calculated from the K–tree that the score function generates. The shorter SIS length is better in this scenario. For K–tree generation, one needs to have a W–set generated for the FSM. For this W–set generation task, we picked Chassis-CT algorithm. The reason is the superior quality results that it possesses, as shown in Chapter 6. Once K–tree is ready, we use the technique that Jourdan et al. (2016) proposes to calculate the SIS lengths from the given K–tree.

For the score function $\Phi_{ssh}$, we mentioned in Section 5.4 that a parameter $z$ should be picked. For this experiment and on-wards, we pick $z$ as the cube of the total number of states in the partition $\Pi$, which is an argument of our score functions. Formally, $z = (\sum\limits_{B \in \Pi} |B|)^3$ .

| $n$ | $p$ | $q$ | $\Phi_{car}$ | $\Phi_{lbs}$ | $\Phi_{ssq}$ | $\Phi_{ssh}$ |
|---|---|---|---|---|---|---|
| 128 | 2 | 2 | 40.0761 | 41.4280 | 40.0071 | 39.9971 |
| | | 16 | 6.5976 | 6.6605 | 6.6013 | 6.6013 |
| | | 128 | 4.8931 | 4.9945 | 4.8855 | 4.8855 |
| | 16 | 2 | 31.7427 | 32.3980 | 31.7112 | 31.7112 |
| | | 16 | 4.3792 | 4.4118 | 4.3837 | 4.3837 |
| | | 128 | 3.0846 | 3.1236 | 3.0903 | 3.0903 |
| | 128 | 2 | 29.6765 | 30.2327 | 29.6442 | 29.6442 |
| | | 16 | 4.0239 | 4.0521 | 4.0273 | 4.0273 |
| | | 128 | 2.7341 | 2.7667 | 2.7400 | 2.7400 |
| 256 | 2 | 2 | 44.4586 | 45.5338 | 44.4162 | 44.4161 |
| | | 16 | 8.7503 | 8.8394 | 8.7535 | 8.7535 |
| | | 128 | 4.9836 | 4.9915 | 4.9770 | 4.9770 |
| | 16 | 2 | 35.7845 | 36.4945 | 35.7561 | 35.7561 |
| | | 16 | 5.6709 | 5.7356 | 5.6724 | 5.6724 |
| | | 128 | 3.1266 | 3.1306 | 3.1214 | 3.1214 |
| | 128 | 2 | 33.7946 | 34.4210 | 33.7707 | 33.7707 |
| | | 16 | 5.1572 | 5.2098 | 5.1582 | 5.1582 |
| | | 128 | 2.8524 | 2.8570 | 2.8489 | 2.8489 |
| 512 | 2 | 2 | 50.9905 | 52.2253 | 50.9384 | 50.9380 |
| | | 16 | 9.3826 | 9.3966 | 9.3672 | 9.3672 |
| | | 128 | 2.8554 | 2.8554 | 2.8554 | 2.8554 |
| | 16 | 2 | 39.8251 | 40.3661 | 39.7841 | 39.7841 |
| | | 16 | 5.9272 | 5.9383 | 5.9252 | 5.9252 |
| | | 128 | 2.3939 | 2.3939 | 2.3939 | 2.3939 |
| | 128 | 2 | 37.6010 | 38.1334 | 37.5705 | 37.5705 |
| | | 16 | 5.4212 | 5.4301 | 5.4197 | 5.4197 |
| | | 128 | 2.3179 | 2.3179 | 2.3179 | 2.3179 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 1024 | 2 | 2 | 57.4760 | 58.6102 | 57.4419 | 57.4413 |
| | | 16 | 9.1692 | 9.1772 | 9.1740 | 9.1740 |
| | | 128 | 3.6140 | 3.6153 | 3.6140 | 3.6140 |
| | 16 | 2 | 44.2634 | 44.8843 | 44.2577 | 44.2577 |
| | | 16 | 6.2031 | 6.2059 | 6.2032 | 6.2032 |
| | | 128 | 2.7347 | 2.7350 | 2.7347 | 2.7347 |
| | 128 | 2 | 41.6493 | 42.2245 | 41.6434 | 41.6434 |
| | | 16 | 5.6913 | 5.6935 | 5.6914 | 5.6914 |
| | | 128 | 2.5889 | 2.5891 | 2.5889 | 2.5889 |
| 2048 | 2 | 2 | 63.4012 | 64.6209 | 63.3859 | 63.3853 |
| | | 16 | 9.8294 | 9.8952 | 9.8313 | 9.8313 |
| | | 128 | 3.9674 | 3.9710 | 3.9675 | 3.9675 |
| | 16 | 2 | 48.2013 | 48.9618 | 48.1855 | 48.1855 |
| | | 16 | 6.5608 | 6.5916 | 6.5617 | 6.5617 |
| | | 128 | 2.8156 | 2.8185 | 2.8157 | 2.8157 |
| | 128 | 2 | 45.2429 | 45.9208 | 45.2283 | 45.2283 |
| | | 16 | 5.9681 | 5.9931 | 5.9689 | 5.9689 |
| | | 128 | 2.6479 | 2.6502 | 2.6479 | 2.6479 |
| 4096 | 2 | 2 | 70.2362 | 71.1661 | 70.2074 | 70.2063 |
| | | 16 | 12.5508 | 12.5775 | 12.5507 | 12.5507 |
| | | 128 | 3.1108 | 3.1200 | 3.1108 | 3.1108 |
| | 16 | 2 | 54.0321 | 55.1123 | 54.0222 | 54.0222 |
| | | 16 | 8.0509 | 8.0698 | 8.0525 | 8.0525 |
| | | 128 | 2.5784 | 2.5816 | 2.5784 | 2.5784 |
| | 128 | 2 | 50.3250 | 51.2519 | 50.3169 | 50.3169 |
| | | 16 | 7.1188 | 7.1336 | 7.1199 | 7.1199 |
| | | 128 | 2.4843 | 2.4869 | 2.4843 | 2.4843 |

Table 7.1 Average length of State Identification Sequences generated by using different score functions

By looking at Table 7.1 and Figures 7.1-7.2, it is clear that $\Phi_{ssq}$ and $\Phi_{ssh}$ are two algorithms that provide better quality. The better of these two is not very obvious at the first glance. However, a more keen comparison reveals that $\Phi_{ssh}$ produces shorter SISs by a very small margin. For this reason, the rest of the experiments that this chapter presents are conducted by using the score function $\Phi_{ssh}$.

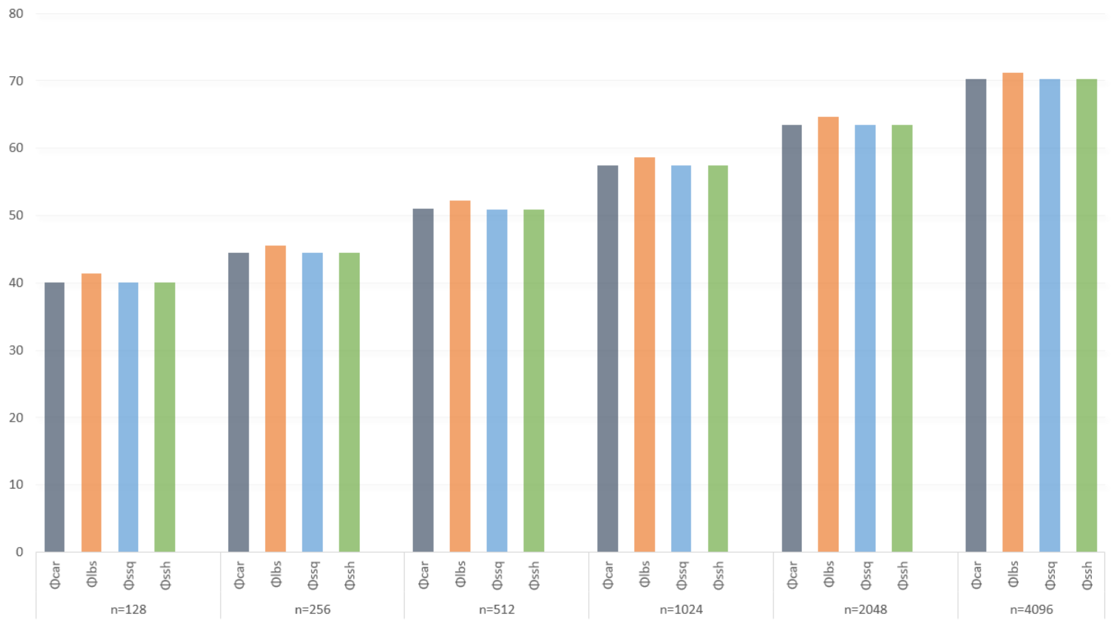

Figure 7.1 Average length of State Identification Sequences generated by each score function, for $p = 2$ and $q = 2$

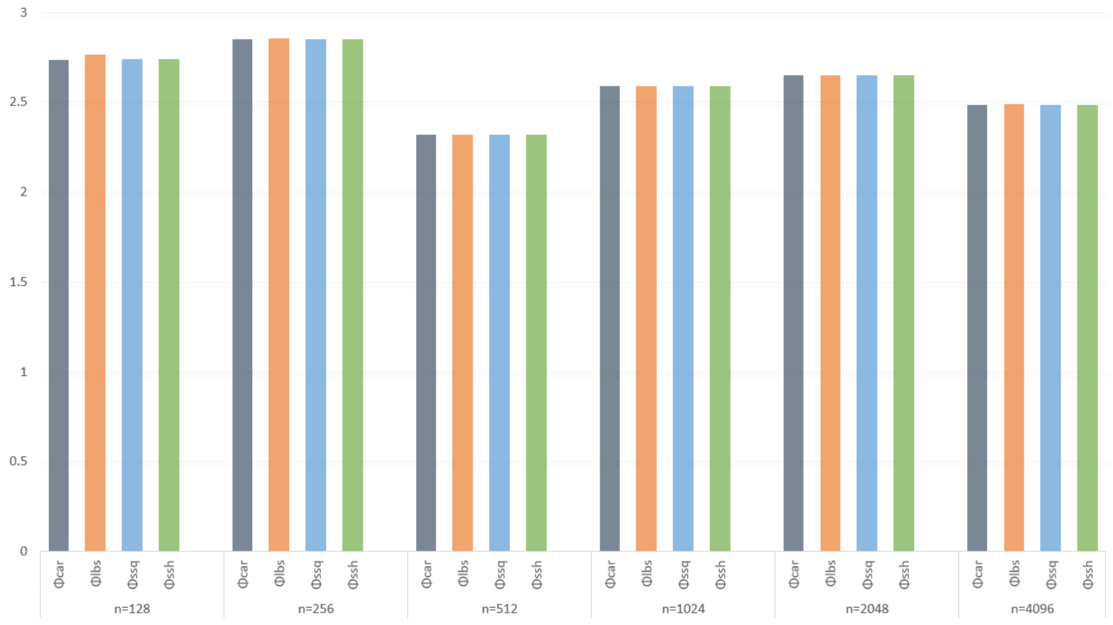

Figure 7.2 Average length of State Identification Sequences generated by each score function, for $p = 128$ and $q = 128$

## 7.2 Experiments on the Use of Implied K–trees

The ultimate purpose of this chapter (Chapter 7) is to make an experiment that compares the known state identification sequence (SIS) generation techniques. Some of these techniques directly work on W–sets, whereas the technique that Jourdan et al. (2016) suggests requires a K–tree. Another technique that Kohavi (1978) and Kohavi et al. (1974) suggest is not very clear on the data structure and implementation details. However, we will assume that these papers put forth a structure identical to a K–tree for the usage of their technique. After all, we may say that we have two SIS generation techniques that require building a K–tree.

Given any W–set or K–set, Algorithm 7 can build a K–tree. However, the way that Chassis-P (Algorithm 6) creates partitions and trace them independently is identical to the one of our K-tree building algorithm, Algorithm 7. Therefore, Chassis-P already creates a specific sequence for every candidate K-tree node. In other words, Chassis-P itself implies a K-tree. Therefore, we can even go ahead and argue that Chassis-P creates a K-tree implicitly. At the same time, what Chassis-P produces is a W–set on its own and this W–set can be fed to Algorithm 7 to produce another K–tree. Our K–tree algorithm, Algorithm 7, considers all available W–set/K–set elements at each step and makes use of a heuristic score function for picking the best sequence possible. Therefore, we suggest that using the natural K–tree implied by the progress of Chassis-P might in fact be worse than reiterating its resulting W–set by Algorithm 7. The same situation applies for the algorithm that Soucha & Bogdanov (2020) suggests for K–set creation (to be called Soucha in this document). Soucha also follows an approach similar to Algorithm 7 and may be said to imply a K–tree. We also wonder whether applying Algorithm 7 on the resulting K–set of Soucha is better or worse than using the implied K–tree by this algorithm.

For testing these above-mentioned questions, we designed an experiment where first, the implied K–trees of Chassis-P and Soucha are considered (see columns "Raw" in Table 7.2). Second, we also used Algorithm 7 to generate K–trees with the resulting W–sets and K–sets of these two algorithms (see columns "Algorithm 7" in Table 7.2). Once K–trees are ready, we use the technique that Jourdan et al. (2016) proposes to calculate the SIS lengths from the given K–trees and present these results below.

| | | | Chassis-P | | Soucha | |
|---|---|---|---|---|---|---|
| $n$ | $p$ | $q$ | Raw | Algorithm 7 | Raw | Algorithm 7 |
| 128 | 2 | 2 | 30.7055 | 30.5116 | 27.2898 | 27.2088 |
| | | 16 | 3.6973 | 3.6989 | 3.3274 | 3.3276 |
| | | 128 | 1.7924 | 1.7906 | 1.6801 | 1.6801 |
| | 16 | 2 | 24.8563 | 24.9214 | 18.8454 | 18.9010 |
| | | 16 | 3.1735 | 3.1737 | 2.6012 | 2.6012 |
| | | 128 | 1.7178 | 1.7178 | 1.6357 | 1.6357 |
| | 128 | 2 | 23.4052 | 23.4843 | 16.4660 | 16.5093 |
| | | 16 | 3.0439 | 3.0441 | 2.4077 | 2.4077 |
| | | 128 | 1.7025 | 1.7025 | 1.6357 | 1.6357 |
| 256 | 2 | 2 | 34.6929 | 34.4259 | 31.4371 | 31.2237 |
| | | 16 | 4.1669 | 4.1707 | 3.8584 | 3.8584 |
| | | 128 | 2.0488 | 2.0484 | 1.9548 | 1.9548 |
| | 16 | 2 | 27.9491 | 28.1818 | 23.0730 | 23.1450 |
| | | 16 | 3.5448 | 3.5452 | 3.0723 | 3.0723 |
| | | 128 | 1.9711 | 1.9711 | 1.8770 | 1.8770 |
| | 128 | 2 | 26.4721 | 26.7112 | 20.5365 | 20.6118 |
| | | 16 | 3.4061 | 3.4065 | 2.8509 | 2.8509 |
| | | 128 | 1.9550 | 1.9550 | 1.8770 | 1.8770 |
| 512 | 2 | 2 | 38.8786 | 38.3884 | 35.7593 | 35.4582 |
| | | 16 | 4.6685 | 4.6711 | 4.3956 | 4.3957 |
| | | 128 | 2.2047 | 2.2047 | 2.1220 | 2.1220 |
| | 16 | 2 | 31.3023 | 31.5534 | 26.6935 | 26.7859 |
| | | 16 | 3.9463 | 3.9464 | 3.5804 | 3.5803 |
| | | 128 | 2.1144 | 2.1144 | 2.0169 | 2.0169 |
| | 128 | 2 | 29.5964 | 29.8595 | 24.3309 | 24.4262 |
| | | 16 | 3.7889 | 3.7890 | 3.3567 | 3.3566 |
| | | 128 | 2.0954 | 2.0954 | 2.0124 | 2.0124 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 1024 | 2 | 2 | 43.1536 | 42.6713 | 40.4096 | 39.8640 |
| | | 16 | 5.0353 | 5.0379 | 4.8878 | 4.8879 |
| | | 128 | 2.2595 | 2.2595 | 2.1997 | 2.1997 |
| | 16 | 2 | 34.5141 | 34.8498 | 30.2084 | 30.3215 |
| | | 16 | 4.2405 | 4.2407 | 4.0384 | 4.0383 |
| | | 128 | 2.1646 | 2.1646 | 2.0861 | 2.0861 |
| | 128 | 2 | 32.5722 | 32.9146 | 27.6665 | 27.7842 |
| | | 16 | 4.0659 | 4.0660 | 3.8050 | 3.8049 |
| | | 128 | 2.1434 | 2.1434 | 2.0617 | 2.0617 |
| 2048 | 2 | 2 | 47.4192 | 46.8336 | 45.2098 | 44.3971 |
| | | 16 | 5.4205 | 5.4231 | 5.3397 | 5.3399 |
| | | 128 | 2.3442 | 2.3442 | 2.2966 | 2.2966 |
| | 16 | 2 | 37.6385 | 38.0556 | 33.7499 | 33.8843 |
| | | 16 | 4.5110 | 4.5118 | 4.4116 | 4.4114 |
| | | 128 | 2.2297 | 2.2297 | 2.1647 | 2.1647 |
| | 128 | 2 | 35.3364 | 35.7559 | 31.0186 | 31.1442 |
| | | 16 | 4.3035 | 4.3042 | 4.1670 | 4.1669 |
| | | 128 | 2.2043 | 2.2043 | 2.1365 | 2.1365 |
| 4096 | 2 | 2 | 51.6908 | 50.9676 | 49.4912 | 48.5964 |
| | | 16 | 5.8670 | 5.8709 | 5.7367 | 5.7369 |
| | | 128 | 2.4625 | 2.4625 | 2.4283 | 2.4283 |
| | 16 | 2 | 40.8922 | 41.2724 | 37.2678 | 37.4097 |
| | | 16 | 4.8678 | 4.8691 | 4.6773 | 4.6772 |
| | | 128 | 2.3405 | 2.3405 | 2.2909 | 2.2909 |
| | 128 | 2 | 38.0769 | 38.4608 | 34.0794 | 34.2312 |
| | | 16 | 4.6151 | 4.6165 | 4.4810 | 4.4809 |
| | | 128 | 2.3105 | 2.3105 | 2.2567 | 2.2567 |

Table 7.2 Average length of State Identification Sequences generated by using Chassis-P and Soucha directly or reiterated by Algorithm 7

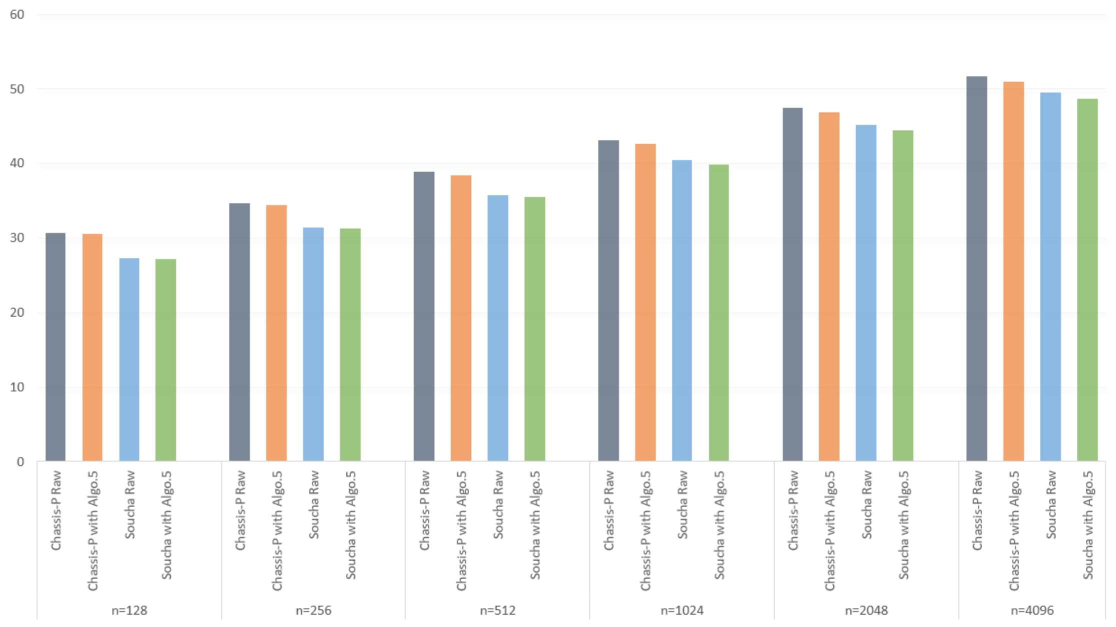

Figure 7.3 Average length of State Identification Sequences generated by CHASSIS-P AND SOUCHA, without and with Algorithm 7, for $p = 2$ and $q = 2$

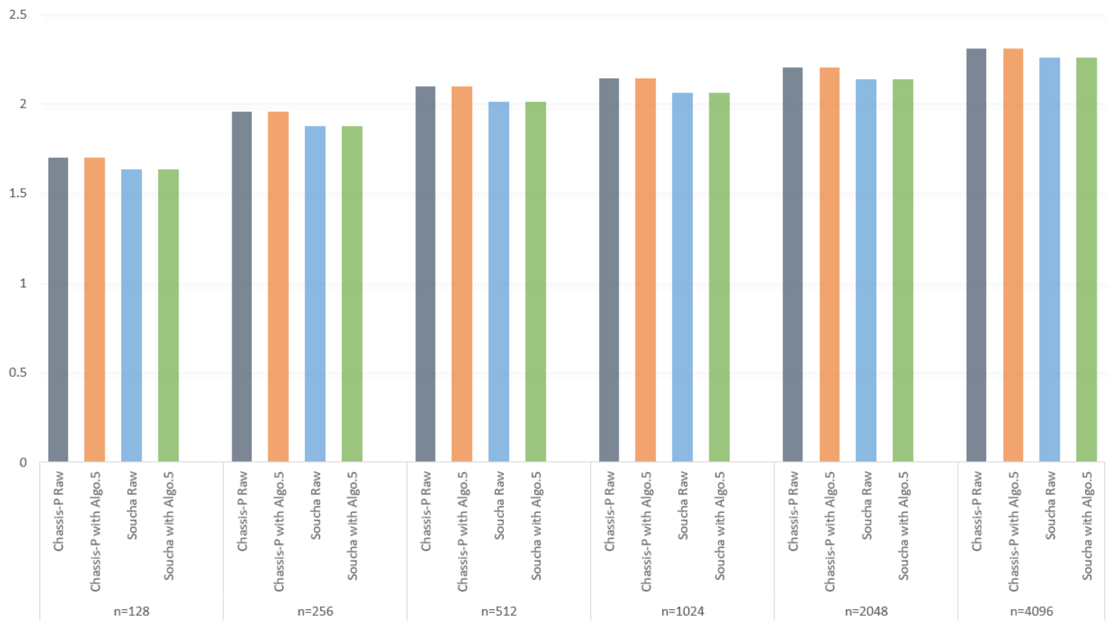

Figure 7.4 Average length of State Identification Sequences generated by CHASSIS-P AND SOUCHA, without and with Algorithm 7, for $p = 128$ and $q = 128$

By these results, we see that applying Algorithm 7 yields to similar results as skipping it, for these two algorithms. Therefore, it was up to our decision to use or not use it. All other W–set algorithms that we have considered in this paper have to be used with a K–tree building algorithm, that is Algorithm 7. In order to ensure uniformity in the methodology and implementation, we choose to use CHASSIS-P and SOUCHA with Algorithm 7 as well.

## 7.3 State Identification Sequence Experiments

In this section, we aim to assess the performance of the SIS construction method given in Jourdan et al. (2016) (referred as JUY in this document) by combining several findings from the rest of this chapter. We realize this assessment by comparing JUY's quality results with other SIS generation methods in the literature. These methods indeed use W–sets/K–sets and several W–set/K–set generation algorithms are proposed and mentioned in this document. We will use some of these in the experiments of this section. Using all methods and parameters would force us to make too many experiments. Because of this, we picked only some of the W–set/K–set generation algorithms and we fixed some parameters as explained in sections 7.1-7.2. The chosen W–set/K–set generation methods are: Gill, Chassis-CT, Chassis-P and Soucha. The SIS generation algorithms that we use for comparison are: Hennie (Hennie, 1964), LeeYann (Lee & Yannakakis, 1996), Hennie+ (discussed in Hennie (1964), proven in Jourdan et al. (2016)) and Kohavi (Kohavi, 1978; Kohavi et al., 1974). We recite that the algorithm Kohavi is our interpretation of the original proposed technique, as we stated in the beginning of Section 7.2. Such interpretation was a must, because the original proposals lack any clear formal definition. The interpretation that we made on the algorithm is indeed a favorable one, by assuming that they intended to use a K–tree constructed similarly to Jourdan et al. (2016). For the algorithms Kohavi and JUY, we build a K–tree by using Algorithm 7 proposed in this document. Such a K–tree is constructed even for W–set/K–set methods Chassis-P and Soucha which imply a K–tree naturally, as we concluded in Section 7.2. When the K–set algorithm Soucha is used, the only SIS generation algorithm that can be used is JUY; because K–sets consist of PADSs and other SIS generation algorithms do not bear the notion of PADS. The results of the experiment are given below.

|  |  |  | Gill |  |  |  |  | Chassis-CT |  |  |  |  | Chassis-P |  |  |  |  | Soucha |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ | $p$ | $q$ | Hennie | LeeYann | Hennie+ | Kohavi | JUY | Hennie | LeeYann | Hennie+ | Kohavi | JUY | Hennie | LeeYann | Hennie+ | Kohavi | JUY | JUY |
| 128 | 2 | 2 | 2.87e019 | 2.66e019 | 3.91e004 | 9.42e003 | 8.00e002 | 1.45e005 | 1.42e005 | 2.34e002 | 4.64e001 | 4.00e001 | 1.34e024 | 1.22e024 | 3.10e007 | 4.00e001 | 3.02e001 | 2.71e001 |
|  |  | 16 | 1.31e005 | 1.28e005 | 1.40e001 | 1.23e001 | 1.23e001 | 9.27e002 | 9.20e002 | 6.08e001 | 7.21e000 | 6.60e000 | 6.07e004 | 5.95e004 | 7.46e001 | 5.34e000 | 3.70e000 | 3.33e000 |
|  |  | 128 | 1.22e003 | 1.20e003 | 1.14e001 | 1.09e001 | 1.07e001 | 3.08e002 | 3.06e002 | 7.33e000 | 5.41e000 | 4.89e000 | 5.85e002 | 5.78e002 | 7.57e000 | 2.56e000 | 1.79e000 | 1.68e000 |
|  | 16 | 2 | 1.63e040 | 1.39e040 | 3.26e009 | 3.61e005 | 6.56e003 | 2.12e004 | 2.08e004 | 4.27e002 | 3.84e001 | 3.17e001 | 9.00e037 | 7.77e037 | 2.63e009 | 3.41e001 | 2.46e001 | 1.88e001 |
|  |  | 16 | 1.06e021 | 9.78e020 | 9.76e002 | 8.51e001 | 4.24e001 | 4.10e002 | 4.07e002 | 2.77e001 | 5.02e000 | 4.38e000 | 1.04e005 | 1.01e005 | 4.31e001 | 4.85e000 | 3.17e000 | 2.60e000 |
|  |  | 128 | 5.99e004 | 5.87e004 | 7.85e000 | 7.39e000 | 7.39e000 | 1.54e002 | 1.53e002 | 4.58e000 | 3.60e000 | 3.09e000 | 2.54e002 | 2.51e002 | 4.86e000 | 2.50e000 | 1.72e000 | 1.64e000 |
|  | 128 | 2 | 2.49e062 | 1.98e062 | 2.62e018 | 1.73e011 | 1.43e007 | 1.68e004 | 1.65e004 | 3.46e002 | 3.61e001 | 2.96e001 | 7.15e037 | 6.17e037 | 3.03e009 | 3.26e001 | 2.31e001 | 1.65e001 |
|  |  | 16 | 8.36e020 | 7.68e020 | 8.51e002 | 7.39e001 | 3.68e001 | 3.27e002 | 3.24e002 | 2.23e001 | 4.67e000 | 4.03e000 | 8.10e004 | 7.92e004 | 3.46e001 | 4.72e000 | 3.04e000 | 2.41e000 |
|  |  | 128 | 4.62e004 | 4.53e004 | 6.36e000 | 5.99e000 | 5.99e000 | 1.21e002 | 1.20e002 | 4.00e000 | 3.25e000 | 2.74e000 | 2.02e002 | 2.00e002 | 4.28e000 | 2.49e000 | 1.70e000 | 1.64e000 |
| 256 | 2 | 2 | 5.87e020 | 5.67e020 | 8.26e004 | 7.05e004 | 1.53e003 | 1.12e006 | 1.10e006 | 5.02e002 | 5.00e001 | 4.44e001 | 7.74e036 | 7.27e036 | 1.06e009 | 4.60e001 | 3.41e001 | 3.11e001 |
|  |  | 16 | 8.98e006 | 8.88e006 | 2.62e001 | 2.23e001 | 2.23e001 | 2.18e003 | 2.17e003 | 1.03e002 | 9.39e000 | 8.75e000 | 1.71e006 | 1.69e006 | 1.77e002 | 6.04e000 | 4.17e000 | 3.86e000 |
|  |  | 128 | 3.52e003 | 3.50e003 | 6.82e000 | 6.62e000 | 6.50e000 | 1.21e003 | 1.20e003 | 1.95e001 | 5.48e000 | 4.98e000 | 3.43e003 | 3.41e003 | 2.01e001 | 3.08e000 | 2.05e000 | 1.95e000 |
|  | 16 | 2 | 3.88e048 | 3.57e048 | 1.64e009 | 2.06e007 | 5.63e004 | 7.40e004 | 7.34e004 | 7.02e002 | 4.21e001 | 3.58e001 | 2.22e065 | 1.99e065 | 8.85e012 | 3.95e001 | 2.78e001 | 2.30e001 |
|  |  | 16 | 1.51e029 | 1.44e029 | 3.67e004 | 5.62e002 | 1.28e002 | 9.46e002 | 9.43e002 | 4.34e001 | 6.34e000 | 5.67e000 | 3.27e007 | 3.22e007 | 1.29e002 | 5.43e000 | 3.54e000 | 3.07e000 |
|  |  | 128 | 2.35e010 | 2.30e010 | 1.40e001 | 1.29e001 | 1.29e001 | 5.46e002 | 5.44e002 | 9.94e000 | 3.56e000 | 3.12e000 | 1.87e003 | 1.85e003 | 1.06e001 | 3.01e000 | 1.97e000 | 1.88e000 |
|  | 128 | 2 | 1.01e140 | 8.01e139 | 4.18e033 | 9.76e010 | 1.77e006 | 5.94e004 | 5.90e004 | 5.60e002 | 4.00e001 | 3.38e001 | 1.71e065 | 1.53e065 | 3.84e013 | 3.80e001 | 2.63e001 | 2.05e001 |
|  |  | 16 | 1.21e029 | 1.15e029 | 3.20e004 | 5.56e002 | 1.26e002 | 7.69e002 | 7.66e002 | 3.50e001 | 5.83e000 | 5.16e000 | 2.62e007 | 2.58e007 | 1.03e002 | 5.29e000 | 3.41e000 | 2.85e000 |
|  |  | 128 | 1.87e010 | 1.83e010 | 1.13e001 | 1.04e001 | 1.04e001 | 4.31e002 | 4.30e002 | 8.30e000 | 3.29e000 | 2.85e000 | 1.48e003 | 1.47e003 | 8.92e000 | 2.99e000 | 1.96e000 | 1.88e000 |
| 512 | 2 | 2 | 6.82e023 | 6.70e023 | 4.22e004 | 6.96e004 | 2.56e003 | 9.37e006 | 9.31e006 | 1.20e003 | 5.63e001 | 5.09e001 | 2.94e047 | 2.84e047 | 7.43e010 | 5.20e001 | 3.80e001 | 3.53e001 |
|  |  | 16 | 5.91e008 | 5.88e008 | 4.72e001 | 3.93e001 | 3.91e001 | 4.99e003 | 4.98e003 | 1.56e002 | 1.00e001 | 9.37e000 | 1.58e009 | 1.57e009 | 3.45e002 | 6.80e000 | 4.67e000 | 4.40e000 |
|  |  | 128 | 1.60e004 | 1.59e004 | 3.12e000 | 2.93e000 | 2.91e000 | 3.79e003 | 3.79e003 | 4.61e001 | 3.11e000 | 2.86e000 | 8.64e003 | 8.61e003 | 4.70e001 | 3.27e000 | 2.20e000 | 2.12e000 |
|  | 16 | 2 | 4.01e057 | 3.84e057 | 3.49e011 | 3.30e009 | 8.78e005 | 3.45e005 | 3.43e005 | 1.65e003 | 4.63e001 | 3.98e001 | 5.02e106 | 4.64e106 | 5.93e014 | 4.52e001 | 3.11e001 | 2.67e001 |
|  |  | 16 | 2.95e049 | 2.85e049 | 3.36e004 | 4.47e001 | 2.01e001 | 2.15e003 | 2.15e003 | 6.10e001 | 6.58e000 | 5.93e000 | 8.94e009 | 8.87e009 | 5.73e002 | 6.10e000 | 3.95e000 | 3.58e000 |
|  |  | 128 | 3.41e017 | 3.36e017 | 2.84e001 | 2.19e001 | 2.18e001 | 1.60e003 | 1.60e003 | 2.06e001 | 2.60e000 | 2.39e000 | 5.36e003 | 5.34e003 | 2.15e001 | 3.16e000 | 2.11e000 | 2.02e000 |
|  | 128 | 2 | 1.32e264 | 1.09e264 | 2.12e014 | 1.09e009 | 6.81e005 | 2.73e005 | 2.72e005 | 1.31e003 | 4.40e001 | 3.76e001 | 9.79e106 | 9.06e106 | 2.43e017 | 4.35e001 | 2.94e001 | 2.43e001 |
|  |  | 16 | 1.25e060 | 1.20e060 | 1.87e009 | 1.92e003 | 2.21e002 | 1.75e003 | 1.74e003 | 4.88e001 | 6.08e000 | 5.42e000 | 7.05e009 | 6.99e009 | 4.52e002 | 5.95e000 | 3.79e000 | 3.36e000 |
|  |  | 128 | 2.63e017 | 2.60e017 | 2.23e001 | 1.73e001 | 1.72e001 | 1.25e003 | 1.24e003 | 1.66e001 | 2.52e000 | 2.32e000 | 4.17e003 | 4.16e003 | 1.74e001 | 3.14e000 | 2.10e000 | 2.01e000 |
| 1024 | 2 | 2 | 1.36e029 | 1.35e029 | 1.93e005 | 9.34e005 | 9.57e003 | 5.51e007 | 5.49e007 | 3.46e003 | 6.27e001 | 5.74e001 | 1.29e062 | 1.26e062 | 4.63e013 | 5.85e001 | 4.23e001 | 3.97e001 |
|  |  | 16 | 1.10e011 | 1.09e011 | 7.45e001 | 6.67e001 | 6.60e001 | 3.44e004 | 3.44e004 | 1.71e002 | 9.76e000 | 9.17e000 | 7.52e008 | 7.50e008 | 8.01e002 | 7.44e000 | 5.04e000 | 4.89e000 |
|  |  | 128 | 8.11e004 | 8.09e004 | 4.22e000 | 4.03e000 | 4.03e000 | 9.91e003 | 9.90e003 | 9.71e001 | 3.88e000 | 3.61e000 | 1.65e005 | 1.65e005 | 9.90e001 | 3.44e000 | 2.26e000 | 2.20e000 |
|  | 16 | 2 | 7.73e064 | 7.57e064 | 1.38e015 | 8.36e012 | 1.99e007 | 1.62e006 | 1.62e006 | 3.85e003 | 5.05e001 | 4.43e001 | 4.57e181 | 4.31e181 | 5.17e015 | 5.07e001 | 3.44e001 | 3.02e001 |
|  |  | 16 | 4.11e067 | 4.02e067 | 2.06e002 | 2.92e001 | 2.92e001 | 1.28e004 | 1.28e004 | 5.86e001 | 6.72e000 | 6.20e000 | 4.55e013 | 4.53e013 | 4.30e003 | 6.66e000 | 4.24e000 | 4.04e000 |
|  |  | 128 | 1.05e023 | 1.04e023 | 6.52e001 | 2.86e001 | 2.79e001 | 3.81e003 | 3.81e003 | 3.84e001 | 2.95e000 | 2.73e000 | 1.15e005 | 1.15e005 | 4.03e001 | 3.34e000 | 2.16e000 | 2.09e000 |
|  | 128 | 2 | – | – | 5.99e011 | 3.91e011 | 9.32e006 | 1.24e006 | 1.23e006 | 2.86e003 | 4.79e001 | 4.16e001 | 3.82e184 | 3.59e184 | 3.75e016 | 4.88e001 | 3.25e001 | 2.77e001 |
|  |  | 16 | 2.03e145 | 1.93e145 | 6.74e005 | 2.38e001 | 2.38e001 | 9.98e003 | 9.96e003 | 4.67e001 | 6.20e000 | 5.69e000 | 3.54e013 | 3.52e013 | 3.41e003 | 6.49e000 | 4.07e000 | 3.80e000 |
|  |  | 128 | 1.09e023 | 1.08e023 | 5.34e001 | 2.39e001 | 2.33e001 | 2.91e003 | 2.91e003 | 2.98e001 | 2.81e000 | 2.59e000 | 8.76e004 | 8.75e004 | 3.15e001 | 3.32e000 | 2.14e000 | 2.06e000 |

| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2048 | 2 | 2 | 3.12e038 | 3.10e038 | 2.67e006 | 3.93e007 | 5.54e004 | 1.15e009 | 1.15e009 | 1.35e004 | 6.85e001 | 6.34e001 | 3.85e084 | 3.80e084 | 5.75e015 | 6.49e001 | 4.65e001 | 4.43e001 |
| | | 16 | 5.33e011 | 5.32e011 | 1.24e002 | 1.17e002 | 1.10e002 | 2.51e004 | 2.51e004 | 1.35e002 | 1.05e001 | 9.83e000 | 1.73e012 | 1.72e012 | 2.19e003 | 8.14e000 | 5.42e000 | 5.34e000 |
| | | 128 | 7.38e005 | 7.37e005 | 6.55e000 | 6.37e000 | 6.37e000 | 2.28e004 | 2.28e004 | 1.93e002 | 4.51e000 | 3.97e000 | 2.44e006 | 2.44e006 | 1.96e002 | 3.92e000 | 2.34e000 | 2.30e000 |
| | 16 | 2 | 1.48e074 | 1.47e074 | 1.97e018 | 3.39e016 | 7.15e008 | 6.36e006 | 6.36e006 | 7.92e003 | 5.45e001 | 4.82e001 | 5.77e302 | 5.52e302 | 1.04e016 | 5.64e001 | 3.76e001 | 3.37e001 |
| | | 16 | 7.28e070 | 7.20e070 | 1.49e002 | 5.13e001 | 5.13e001 | 1.06e004 | 1.06e004 | 5.10e001 | 7.21e000 | 6.56e000 | 2.52e024 | 2.51e024 | 5.28e005 | 7.32e000 | 4.51e000 | 4.41e000 |
| | | 128 | 1.60e054 | 1.59e054 | 5.06e002 | 7.46e000 | 6.57e000 | 8.30e003 | 8.30e003 | 7.15e001 | 3.38e000 | 2.82e000 | 8.24e005 | 8.23e005 | 7.67e001 | 3.79e000 | 2.23e000 | 2.16e000 |
| | 128 | 2 | – | – | 4.86e014 | 1.29e015 | 3.88e008 | 4.73e006 | 4.72e006 | 5.87e003 | 5.15e001 | 4.52e001 | – | – | 1.16e015 | 5.41e001 | 3.53e001 | 3.10e001 |
| | | 16 | – | – | 4.91e001 | 4.05e001 | 4.05e001 | 8.44e003 | 8.43e003 | 4.08e001 | 6.62e000 | 5.97e000 | 1.84e024 | 1.83e024 | 3.83e005 | 7.11e000 | 4.30e000 | 4.17e000 |
| | | 128 | 9.91e060 | 9.81e060 | 3.66e002 | 6.06e000 | 5.41e000 | 6.12e003 | 6.12e003 | 5.32e001 | 3.22e000 | 2.65e000 | 6.07e005 | 6.06e005 | 5.72e001 | 3.76e000 | 2.20e000 | 2.14e000 |
| 4096 | 2 | 2 | 4.64e052 | 4.63e052 | 1.13e008 | 3.25e009 | 3.56e005 | 1.35e010 | 1.35e010 | 8.38e004 | 7.53e001 | 7.02e001 | 1.13e114 | 1.12e114 | 7.85e018 | 7.13e001 | 5.06e001 | 4.85e001 |
| | | 16 | 6.29e013 | 6.28e013 | 1.15e002 | 1.23e002 | 9.37e001 | 5.52e004 | 5.52e004 | 1.75e002 | 1.33e001 | 1.26e001 | 1.17e014 | 1.17e014 | 7.20e003 | 8.81e000 | 5.87e000 | 5.74e000 |
| | | 128 | 1.06e007 | 1.06e007 | 1.13e001 | 1.11e001 | 1.11e001 | 4.98e004 | 4.98e004 | 3.96e002 | 3.87e000 | 3.11e000 | 1.01e007 | 1.01e007 | 4.02e002 | 4.27e000 | 2.46e000 | 2.43e000 |
| | 16 | 2 | 2.18e088 | 2.17e088 | 2.24e018 | 1.61e022 | 1.28e011 | 3.58e009 | 3.58e009 | 4.75e005 | 5.98e001 | 5.40e001 | – | – | 1.68e014 | 6.25e001 | 4.08e001 | 3.73e001 |
| | | 16 | 6.75e063 | 6.72e063 | 1.32e002 | 9.73e001 | 9.67e001 | 2.32e004 | 2.32e004 | 6.89e001 | 8.77e000 | 8.05e000 | 1.41e034 | 1.41e034 | 2.08e007 | 7.89e000 | 4.87e000 | 4.68e000 |
| | | 128 | 7.87e063 | 7.84e063 | 6.33e000 | 5.38e000 | 5.38e000 | 1.78e004 | 1.78e004 | 1.42e002 | 3.34e000 | 2.58e000 | 3.79e006 | 3.79e006 | 1.55e002 | 4.16e000 | 2.34e000 | 2.29e000 |
| | 128 | 2 | – | – | 1.72e020 | 1.60e021 | 7.28e010 | 2.49e009 | 2.49e009 | 3.31e005 | 5.61e001 | 5.03e001 | – | – | 3.05e016 | 5.97e001 | 3.80e001 | 3.41e001 |
| | | 16 | – | – | 1.13e002 | 7.31e001 | 7.31e001 | 1.79e004 | 1.79e004 | 5.32e001 | 7.83e000 | 7.12e000 | 2.13e034 | 2.13e034 | 1.14e007 | 7.63e000 | 4.61e000 | 4.48e000 |
| | | 128 | 9.06e214 | 8.93e214 | 4.84e000 | 4.39e000 | 4.39e000 | 1.24e004 | 1.24e004 | 9.95e001 | 3.24e000 | 2.48e000 | 2.65e006 | 2.65e006 | 1.09e002 | 4.13e000 | 2.31e000 | 2.26e000 |

Table 7.3 Average state identification sequence lengths of several techniques, tested with different W–set/K–set generation algorithms

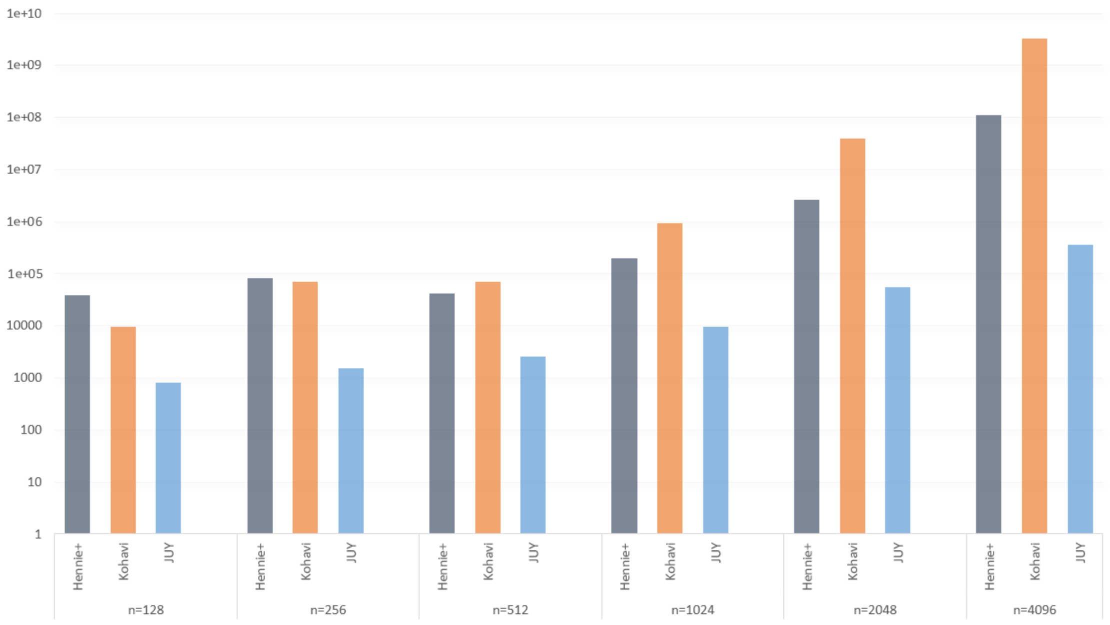

Figure 7.5 Average length of SISs generated by various techniques, with a W–set generated by GILL, given in logarithmic scale, for $p = 2$ and $q = 2$

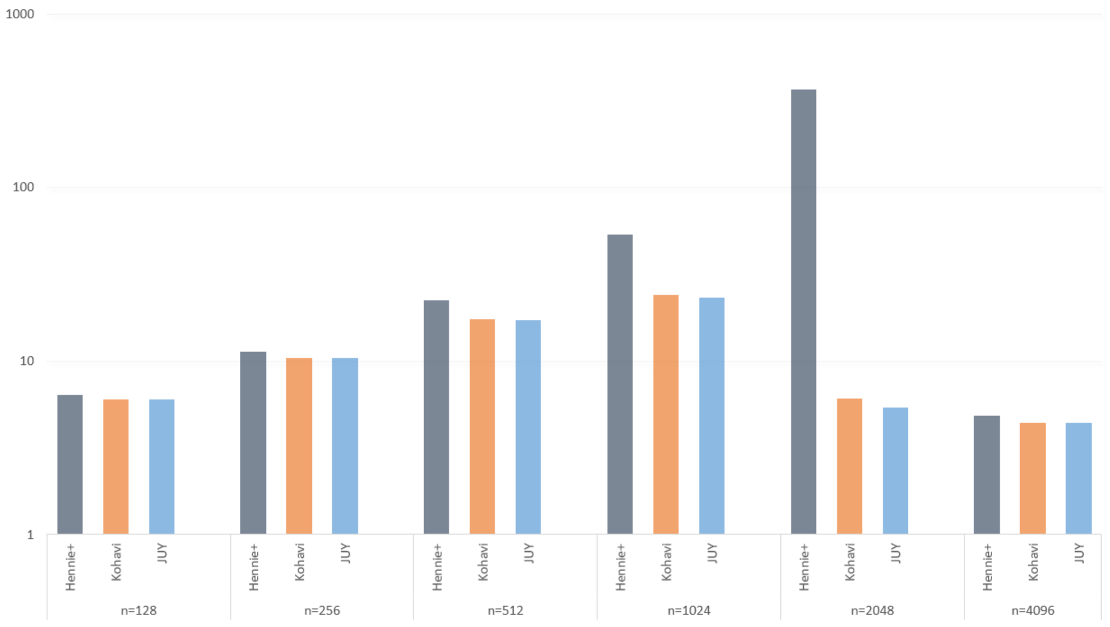

Figure 7.6 Average length of SISs generated by various techniques, with a W–set generated by GILL, given in logarithmic scale, for $p = 128$ and $q = 128$

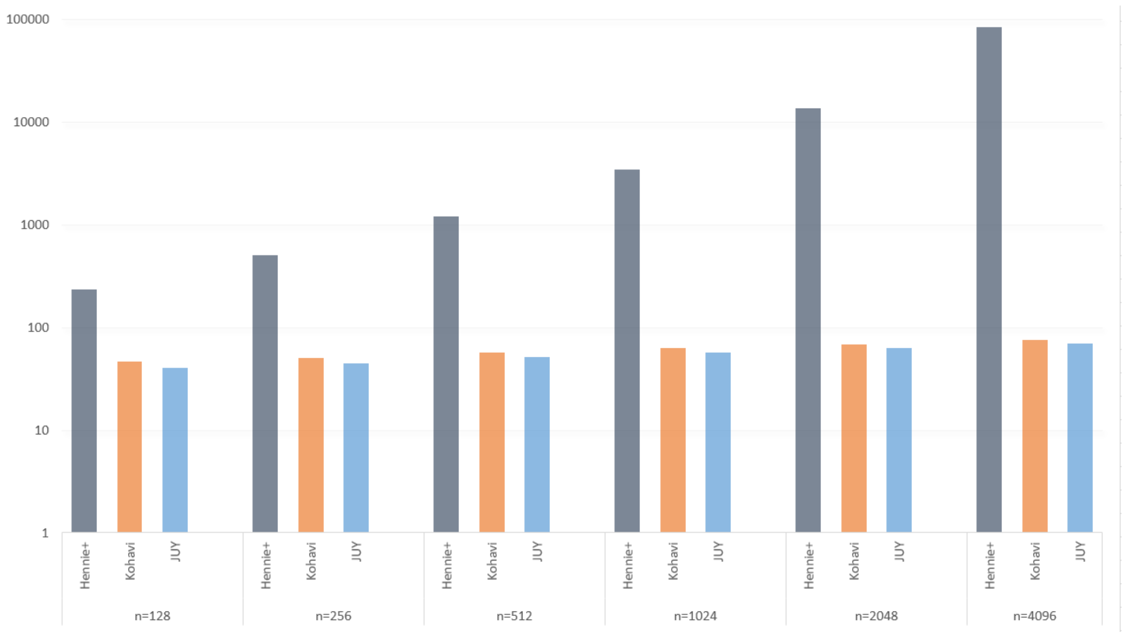

Figure 7.7 Average length of SISs generated by various techniques, with a W–set generated by CHASSIS-CT, given in logarithmic scale, for $p = 2$ and $q = 2$

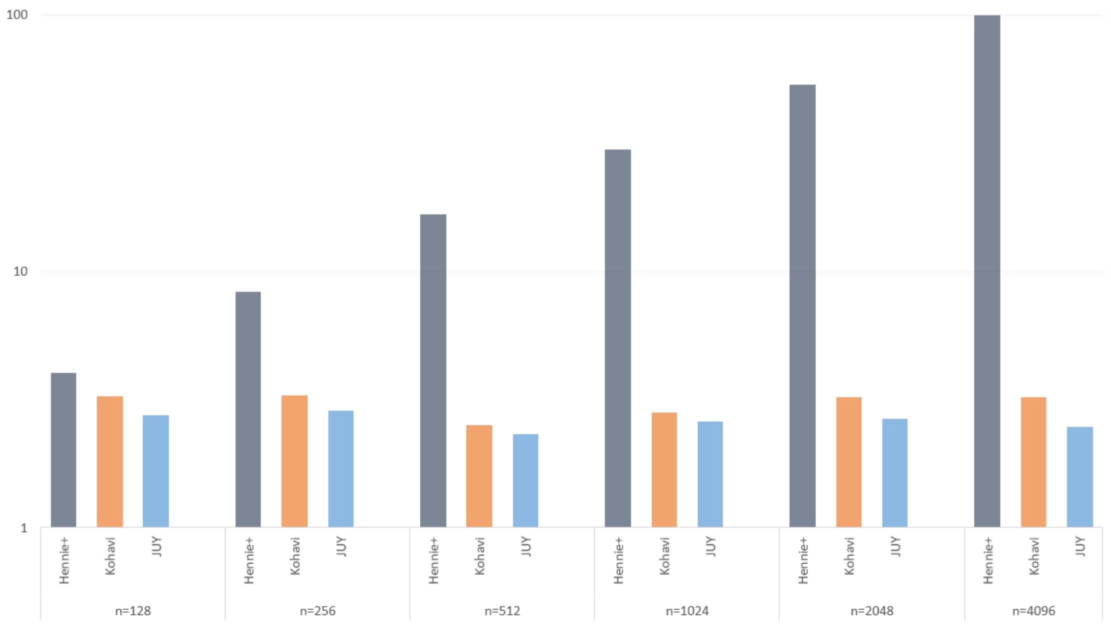

Figure 7.8 Average length of SISs generated by various techniques, with a W–set generated by CHASSIS-CT, given in logarithmic scale,for $p = 128$ and $q = 128$

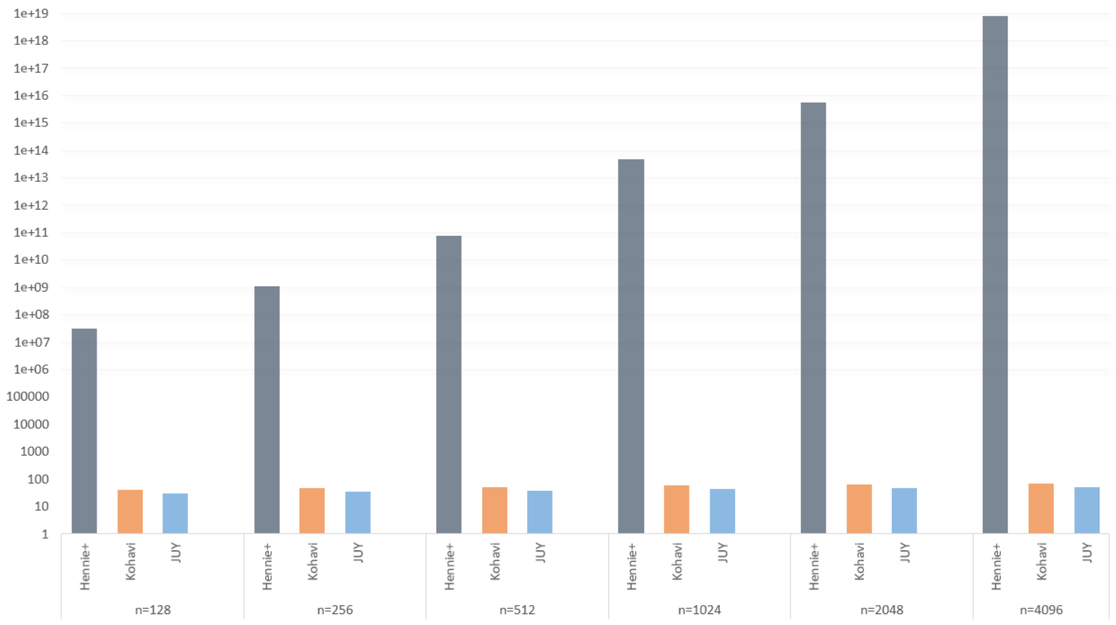

Figure 7.9 Average length of SISs generated by various techniques, with a W–set generated by CHASSIS-P, given in logarithmic scale, for $p = 2$ and $q = 2$

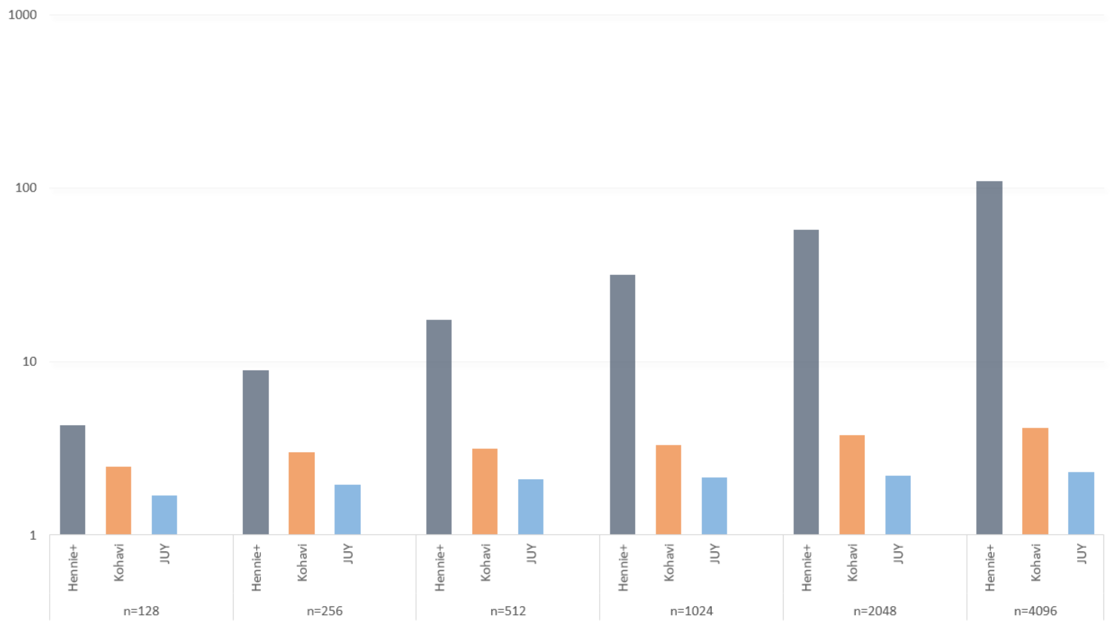

Figure 7.10 Average length of SISs generated by various techniques, with a W–set generated by CHASSIS-P, given in logarithmic scale, for $p = 128$ and $q = 128$

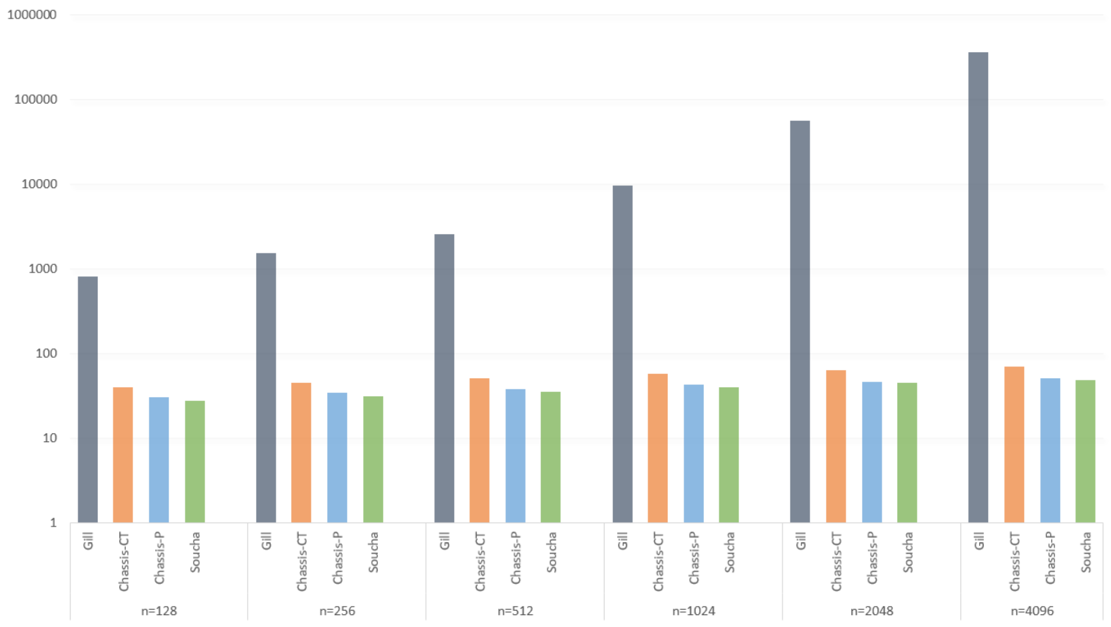

Figure 7.11 Average length of SISs generated by JUY, with W–sets/K–sets generated by several algorithms, given in logarithmic scale, for $p = 2$ and $q = 2$

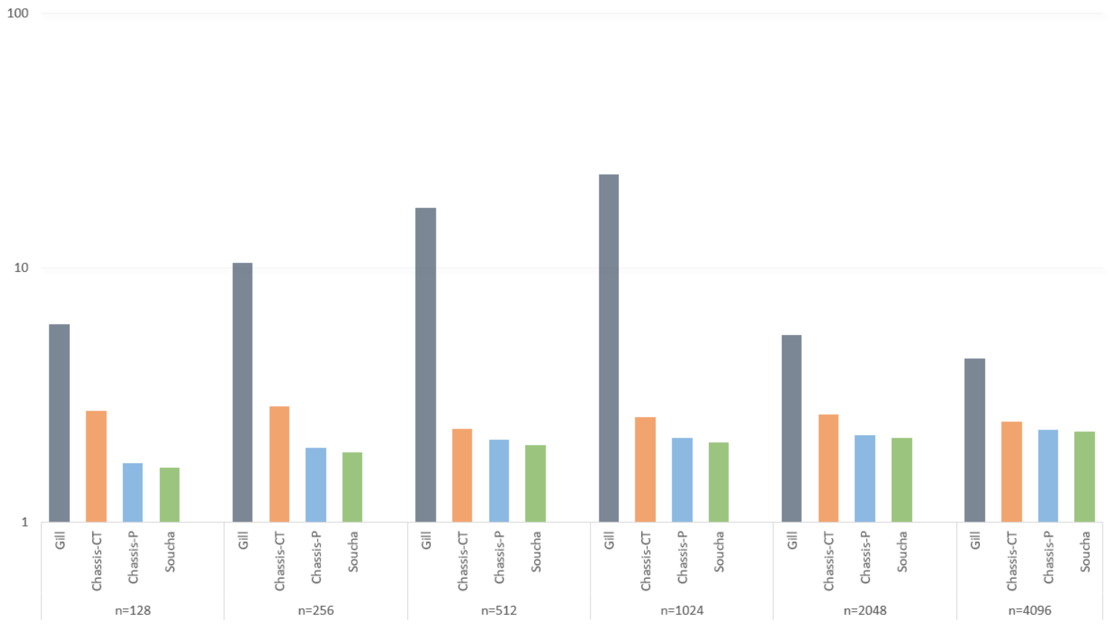

Figure 7.12 Average length of SISs generated by JUY, with W–sets/K–sets generated by several algorithms, given in logarithmic scale, for $p = 128$ and $q = 128$

With any W–set/K–set generation algorithm, we observe that Hennie and LeeYann perform significantly worse than the other three techniques. Because of the very high magnitude of SISs they produce, we were not able to place them in the figures 7.5–7.12, even though we drew these figures in logarithmic scale. Some cells of Table 7.3 that belong to Hennie and LeeYann are also left blank; because those length results caused the 64-bit floating point number to overflow.

In figures 7.5–7.10, we present the lengths of the SISs generated by Hennie+, Kohavi and JUY, with W–set algorithms Gill, Chassis-CT and Chassis-P separately. Such a figure was not possible for Soucha, because it produces a K–set and only JUY is able to process K–sets. In these figures, it is clear that JUY outperforms the other two techniques in every scenario. Kohavi seems to perform better than Hennie+ in most scenarios, with the exception where the W–set algorithm is Gill and $p$,$q$ values are low. The difference between JUY and Kohavi is generally not very huge; however with a W–set generated by Gill, this is not valid. Gill is an algorithm that produces W–sets with high cardinality, as shown in Chapter 6. Although JUY and Kohavi uses the same K–tree, Kohavi is prone to calculate a significantly longer SIS as the K–tree depth starts to grow large. As JUY does not have this deficit, it performs better than Kohavi, especially when paired with Gill, which produces high cardinality W–sets that result in K–trees with greater height. The results of Hennie+ become better when using Chassis-CT, because Hennie+ works directly on the W–set without producing K–tree and is affected by the structure of the W–set greatly. Considering the fact that Chassis-CT produces very minimal W–sets -as shown in Chapter 6- observing the best results of Hennie+ with Chassis-CT is meaningful.

In figures 7.11 and 7.12, we compare the effects of W–set/K–set methods on the SIS generation process. The only SIS generation technique that can work with all W–set/K–set algorithms is JUY, that is why we use JUY as the base of this comparison. For two different $p$,$q$ configurations, the results are clear that Soucha helps produce the best (shortest) SISs, followed closely by Chassis-P. Chassis-CT also performs much better compared to Gill. This result is expected, considering that Chassis-CT was developed for general W–set optimization, whereas Chassis-P and Soucha were considered for K–tree generation.

In Figure 7.6, all three SIS techniques show a suspicious trend. Normally, the SIS lengths are expected to grow greater as the state count of the FSM increases. In Figure 7.6, Kohavi and JUY show an inverse trend starting from $n = 2048$: the resulting SISs become smaller. Hennie+ joins them at $n = 4096$. We designed and performed another set of experiments in order to analyze this strange behaviour of the SIS techniques, which is seen only when paired with Gill's W–set generation algorithm. In these experiments, we focused on FSMs with $p = q = 128$ and we analyzed the characteristics of the W–sets that Gill's algorithm creates, and how these characteristics shift as the state count goes from 1024 to 2048 and also from 2048 to 4096. The first very clear observation is: "The sequences in the W–set that Gill's algorithm creates consist of sequences with length 1 or 2 only, for $p = q = 128$". After this observation, we analyzed the ratio of sequences with length 2.

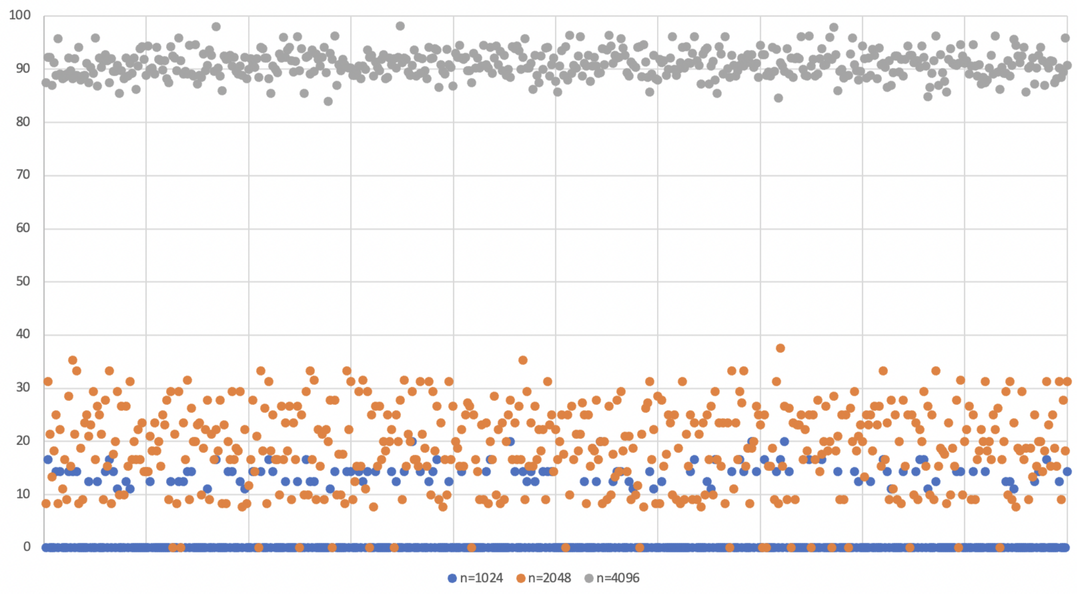

Figure 7.13 Ratios of the number of sequences of length 2 in the W–sets that are created by Gill's algorithm, for varying $n$ values, $p = q = 128$

In Figure 7.13, we presented the ratios of the number of sequences of length 2 to all sequences in the W–sets by Gill's algorithm. We wish to emphasize that, sequences of lengths 3 or more are not observed ever within this experiment with $p = q = 128$. Hence, the inverse of this figure might be considered as the density of sequences of length 1. From the figure, we observe that W–sets for FSMs with $n = 1024$ have sequences of single letters mostly always. This phenomenon changes slightly for $n = 2048$. Once we hit $n = 4096$, most of the sequences in the W–sets are of two letters $(80\% - 100\%)$.

We believe that, this increase in the density of the sequences with 2 letters explain the improvement in the SIS lengths. Longer sequences divide the states into partitions with higher cardinalities, and this situation yields better SISs naturally. The reason that Kohavi and JUY react earlier to this at $n = 2048$ is that they work with a K–tree and our K–tree algorithm (Algorithm 7) has a heuristic process to pick the sequences giving a better partitioning earlier in the process. Even if there are a few sequences of length 2, Kohavi and JUY are able to take advantage of the existence of such sequences by using them earlier in the K-tree. Hennie+, on the other hand, processes the W–set sequences in the order they are produced. Therefore it still hits sequences with length 1 mostly at $n = 2048$ and can only improve when the frequency of 2-length sequences vastly increases at $n = 4096$.

# 8. THREATS TO VALIDITY

During the implementation of the methods mentioned in this document, we implemented several techniques to tackle any errors in the code base.

First of all, all the FSMs that we use for this work must be complete, minimal and strongly connected. This requirement is mentioned in Chapter 2. We ensure the completeness of all the FSMs that we produce by assigning the results of the transition and output functions ($\delta$ and $\lambda$ respectively) for every state-input pair.

For every random FSM that we generate, we apply Tarjan's algorithm (Tarjan, 1972) that we implemented in our project to check the strongly connectedness of the FSM. If the FSM is not strongly connected, we try to pick a random state from one of the connected components and direct one of its transitions (picked randomly again) to a random state from another connected component. We apply this technique for a maximum of 20 times and apply Tarjan's algorithm after every iteration. If the FSM becomes strongly connected in any step, we stop. If the FSM cannot become strongly connected after 20 trials of re-connection, we discard the FSM and start generating a new one.

Once we have an FSM that is complete and strongly connected, ensured by the methods given above, we check the minimality of the FSM. Lemma 1 shows that each state pair of a minimal FSM must have a separating sequence. For checking this, we create the separating pair graph of the FSM, as defined in Definition 1. Every state pair in the original FSM has a corresponding node in this separating pair graph. If a state pair is separable, its corresponding node in the separating pair graph must have a path to the node *Separated* in the graph. If all the nodes in the separating pair graph have paths to *Separated*, then we say that the FSM is minimal. For checking this situation, we apply an inverse breadth-first search from *Separated* and see whether we can discover all the nodes in the graph. If all nodes are discovered, the FSM is ensured to be minimal. In order to check whether we created the transitions in the separating pair graph correctly, we run a procedure to check whether each node in the separating pair graph is built correctly by reversing the building algorithm.

In Chapter 4, we propose new W–set algorithms and we also have few W–set algorithms from the literature, as we mention in the experimental chapter, Chapter 6. We implemented these W–set algorithms and these implementations need a validation. For this purpose, we also implemented a checking procedure to validate whether the resulting input sequence sets are indeed W–sets. This procedure applies the input sequences from the set to all state pairs in the FSM and checks whether each pair can be separated by at least one of the input sequences. As this is the definition of a W–set -as given in Chapter 2-, this validation works for ensuring a set is a W–set.

In Algorithm 7, we present a way for building K–trees. For validating the implementation of this algorithm in our code base, we again constructed a checking procedure which utilizes the definition of a K–tree. In a K–tree, each leaf indicates a single state from the FSM. If we take all the nodes from the root to this leaf node, the collection consisting of the sequences/PADSs on this path must separate this state from all the other states. Therefore, we find such collections for each state of the FSM and check whether each state is separated from all the other states by its collection. This check tells whether the tree we created is a K–tree.

# 9. CONCLUSION

Proper testing of software - especially in big projects - saves a lot of resources and human power and make the software a lot more durable and a lot less error-prone. Therefore, automated testing and debugging have been fields of constant development and research. Although being a sub-branch of automated testing, the development of FSM-based testing has been slower and more gradual. Still, we know that FSM-based testing has good potential and broad use cases. While the generation and usage of ADSs and PDSs are well researched and presented, not all FSMs have these sequences, unfortunately. Therefore, more complicated methods like W–set based methods have to be developed and improved. These methods were known to produce very long test sequences. The K–tree concept that Jourdan et al. (2016) proposed is finally able to reduce the magnitude of the length of SIS to 10s from $10^{20}$s, albeit not practically presented by the original paper. The original paper also did not present any direction on how to build the K–tree from scratch and how well it performs for FSMs that have more than a few states.

In this work, we studied the W–set based testing topic in general. However, we also looked at K–set based methods, which can be generically included in the *W–set based testing* topic. We developed an algorithm (Algorithm 7) that builds a K–tree from a given W–set or K–set. This algorithm constructs a K–tree in breadth-first manner and optimizes the distribution of W–set/K–set elements in the K–tree such that the SIS generated from the K–tree becomes minimal. We presented the tests and the results are promising. We showed that Jourdan et al. (2016)'s technique generates SISs no longer than 100 in any case (for FSMs with up to 4096 states), when paired with a good W–set and our K–tree building algorithm (Algorithm 7). This sets a new bar on the standard length of the state identification sequences. In the future, new heuristics on K–tree generation and novel W–set generation algorithms can be proposed to further improve this SIS quality. In this document, we performed experiments only up to 4096 states. It can be also investigated how the K–tree will expand if the state count is vastly increased. We know by experience that the height of the K–tree is a critical factor in the length of the SIS generated. Therefore, this investigation may be beneficial in knowing whether these novel techniques are cardinality-proof. In such a future work with higher state counts and K–tree heights, we expect that the superiority of Jourdan et al. (2016)'s K–tree technique will be even more obvious.

Another part of our contribution in this work was on W–set generation methods. W–set is a set of sequences where every state pair in an FSM can be separated by at least one of the items of this set. By this property, W–sets are used for not only K–tree building but also many other SIS methods. Therefore, development of W–set algorithms that can build compact W–sets are greatly important. In this work, we proposed four W–set construction algorithms. Three of these were general-purpose algorithms, which served the purpose of building compact W–sets, in terms of the cardinality and in terms of the average length of the sequences in W–sets. By this, we aimed to create algorithms that can be used for any past or future W–set based SIS generation techniques. We performed experiments and compared the performance of these W–set algorithms to algorithms existing in the literature. Our algorithms -especially Chassis-CT- yielded a much better performance in terms of the compactness, with acceptable execution speeds. The last of these four W–set algorithms was intended for specifically K–tree based SIS generation method, where the compactness of the W–set itself is totally discarded. In this technique, the elements of the W–set were decided in a way that they would fit perfectly in the nodes of the K–tree to be constructed. This indeed happened. The last W–set algorithm that we proposed (Chassis-P) performed the best among all W–set algorithms in terms of the quality of the resulting K–trees. It only performed worse than Soucha (a K–set algorithm) by a very small margin, in K–tree SIS results. However, this is natural; because K–sets are more efficient data structures than W–sets (similar to the ADSs vs. PDSs) and the small margin Soucha gains over our W–set algorithm Chassis-P may easily be explained by this difference.

A possible improvement of our implementation can be to use excessive sequence pruning while generating SISs. To explain what we mean by excessive sequence pruning, first let us consider a node of the K-tree labeled by a K-set element $Y$ and set of states $S' \subseteq S$. Furthermore let $s \in S'$ be a state and $w$ be the sequence of inputs suggested to be used in the SIS generation for $s$ by the K-set element $Y$ at this node. Finally, let $w'$ be the shortest prefix of $w$ such that $\forall s' \in S$, $\lambda(s, w') = \lambda(s', w')$ if and only if $\lambda(s, w) = \lambda(s', w)$. In other words, $w'$ is the shortest prefix of $w$ that has the same distinguishing power as $w$ for $s$ when one considers the states in $S$. In this case, we can in fact use $w'$ instead of $w$ for the generation of SIS of $s$ for this particular K-tree node. Our algorithms do use this observation.

However, it is possible to take this observation one step further in the following way. Let $w''$ be the shortest prefix of $w$ such that $\forall s' \in \mathbf{S}'$ (that is, instead of the entire set of states $S$, only the states labeling the corresponding K-tree node are considered) $\lambda(s, w'') = \lambda(s', w'')$ if and only if $\lambda(s, w) = \lambda(s', w)$. In other words, $w''$ is the shortest prefix of $w$ that has the same distinguishing power as $w$ for $s$ when one considers only the states in $S'$. In this case, we can in fact use $w''$ instead of $w$ for the generation of SIS of $s$ for this particular K-tree node. This improved approach is not used in our implementations which would possibly yield even shorter SISs.

# BIBLIOGRAPHY

Aho, A. V., Dahbura, A. T., Lee, D., & Uyar, M. U. (1991). An optimization technique for protocol conformance test generation based on uio sequences and rural chinese postman tours. *IEEE transactions on communications*, *39*(11), 1604–1615.

Banks, A. & Porcello, E. (2017). *Learning React: functional web development with React and Redux.* " O'Reilly Media, Inc.".

Binder, R. (2000). *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Professional.

Bulut, K., Jourdan, G., & Türker, U. C. (2019). Minimizing characterizing sets. In Arbab, F. & Jongmans, S. (Eds.), *Formal Aspects of Component Software - 16th International Conference, FACS 2019, Amsterdam, The Netherlands, October 23-25, 2019, Proceedings*, volume 12018 of *Lecture Notes in Computer Science*, (pp. 72–86). Springer.

Chow, T. S. (1978). Testing software design modeled by finite-state machines. *IEEE transactions on software engineering*, (3), 178–187.

Friedman, A. D. & Menon, P. R. (1971). *Fault detection in digital circuits.* Prentice Hall.

Gargantini, A. (2004). Conformance testing. In Broy, M., Jonsson, B., Katoen, J., Leucker, M., & Pretschner, A. (Eds.), *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *Lecture Notes in Computer Science*, (pp. 87–111). Springer.

Gill, A. (1962). *Introduction to the Theory of Finite-state Machines.* Electronic science series. McGraw-Hill.

Gonenc, G. (1970). A method for the design of fault detection experiments. *IEEE transactions on Computers*, *100*(6), 551–558.

Güniçen, C., İnan, K., Türker, U. C., & Yenigün, H. (2014). The relation between preset distinguishing sequences and synchronizing sequences. *Formal Aspects of Computing*, *26*(6), 1153–1167.

Haydar, M., Petrenko, A., & Sahraoui, H. (2004). Formal verification of web applications modeled by communicating automata. In *International Conference on Formal Techniques for Networked and Distributed Systems*, (pp. 115–132). Springer.

Hennie, F. C. (1964). Fault detecting experiments for sequential circuits. In *Proceedings of the 1964 Proceedings of the Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, SWCT '64, (pp. 95–110)., USA. IEEE Computer Society.

Holzmann, G. J. & Lieberman, W. S. (1991). *Design and validation of computer protocols*, volume 512. Prentice hall Englewood Cliffs.

Jourdan, G., Ural, H., & Yenigün, H. (2016). Reducing locating sequences for testing from finite state machines. In Ossowski, S. (Ed.), *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, (pp. 1654–1659). ACM.

Kohavi, Z. (1978). *Switching and Finite Automata Theory.* McGraw-Hill computer science series. Tata McGraw-Hill.

Kohavi, Z., Rivierre, J., & Kohavi, I. (1974). Checking experiments for sequential machines. *Information Sciences*, *7*, 11 – 28.

Lee, D. & Yannakakis, M. (1994). Testing finite-state machines: State identification and verification. *IEEE Transactions on computers*, *43*(3), 306–320.

Lee, D. & Yannakakis, M. (1996). Principles and methods of testing finite state machines—a survey. *Proceedings of the IEEE*, *84*, 1090 – 1123.

Miao, H., Liu, P., & Mei, J. (2010). An improved algorithm for building the characterizing set. In *2010 4th IEEE International Symposium on Theoretical Aspects of Software Engineering*, (pp. 67–74). IEEE.

Rezaki, A. & Ural, H. (1995). Construction of checking sequences based on characterization sets. *Computer Communications*, *18*(12), 911 – 920.

Sabnani, K. & Dahbura, A. (1988). A protocol test generation procedure. *Computer Networks and ISDN systems*, *15*(4), 285–297.

Sandberg, S. (2004). Homing and synchronizing sequences. In Broy, M., Jonsson, B., Katoen, J., Leucker, M., & Pretschner, A. (Eds.), *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *Lecture Notes in Computer Science*, (pp. 5–33). Springer.

Soucha, M. & Bogdanov, K. (2020). State identification sequences from the splitting tree. *Information and Software Technology*, *123*, 106297.

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM journal on computing*, *1*(2), 146–160.

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

Vasilevskii, M. (1973). Failure diagnosis of automata. *Cybernetics*, *9*(4), 653–665.

Vuong, S. T. (1989). The uiov-method for protocol test sequence generation. In *Proc. 2nd IFIP Int. Workshop on Protocol Test Systems (IWPTS'89)*, (pp. 161–175).