USING SIMULATED ANNEALING FOR COMPUTING TEST CASE-AWARE COVERING ARRAYS

by

Uğur Koç

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

Sabancı University

January 2014

USING SIMULATED ANNEALING FOR COMPUTING TEST CASE-AWARE COVERING ARRAYS

Approved by:

Asst. Prof. Dr. Cemal Yılmaz .....
(Thesis Supervisor)

Assoc. Prof. Dr. Berrin Yanıkoğlu .....

Assoc. Prof. Dr. Bülent Çatay .....

Assoc. Prof. Dr. Erkay Savaş .....

Asst. Prof. Dr. Hüsnü Yenigün .....

Date of Approval: .....

USING SIMULATED ANNEALING FOR COMPUTING TEST CASE-AWARE COVERING ARRAYS

Uğur Koç

Computer Science and Engineering, MS Thesis, 2014

Thesis Supervisor: Asst. Prof. Cemal Yılmaz

Keywords: Software quality assurance, combinatorial interaction testing, covering arrays, test case-aware covering arrays, simulated annealing

Abstract

Exhaustive testing of highly configurable software systems is generally infeasible in practice. For this reason, efficient sampling of the configuration space is important to improve the coverage of testing. A t-way covering array is a list of systematically selected configurations covering all value combinations for every t-way option combinations and it aims to discover faults caused by interactions of configuration options. Despite its many successes, it can be difficult to use covering arrays in practice.

Once a traditional t-way covering array is constructed, the system is then tested by running its test cases in all the selected configurations. By doing so, traditional covering arrays assume that all test cases can run in all configurations of covering array.

Recent studies, however, show that test cases of configurable systems are likely to have assumptions about the underlying configurations, i.e., they are like to have some test case-specific inter-option constraints. When a configuration does not satisfy the test case-specific constraints of a test case, that test case simply skips the configuration, which

prevents the test case from testing all valid combinations of option settings appearing in the configuration an effect called a masking effect. A harmful consequence of masking effects is that they can make the developers to believe that they have tested certain option setting combinations while they in fact have not.

A solution approach is to use test case-aware covering arrays a novel type of combinatorial objects for testing that has been recently introduced. Test case-aware covering arrays take test case-specific inter-option constraints into account when computing combinatorial interaction test suites, such that no masking effects caused by overlooked constraints occur. Given a configuration space model augmented with test case-specific constraints, a test case-aware covering array is not just a set of configurations as is the case in traditional covering arrays, but a set of configurations each of which is associated with a set of test cases, indicating the test cases scheduled to be executed in the configuration.

Although it has been empirically demonstrated that test case-aware covering arrays, compared to traditional covering arrays, can significantly improve the quality of combinatorial interaction testing by avoiding masking effects, there is no efficient and effective algorithms to compute them, except for a couple of proof-of-concept algorithms. We conjecture that this greatly hurts the adaptation of test case-aware covering arrays in practice.

In this thesis, we have developed simulated annealing-based, efficient and effective algorithms to compute test case-aware covering arrays and a tool implementing these algorithms. We, furthermore, compare and contrast the performance of our algorithms by conducting large-scale experiments in which we used two highly configurable large software systems. The results of our empirical studies strongly suggest that the proposed algorithms are an efficient and effective way of computing test case-aware covering arrays and that they perform better than existing approaches.

BENZETİLMİŞ TAVLAMA ALGORİTMASINI KULLANARAK TEST DURUMLARINI DİKKATE ALAN KAPSAYAN DİZİLER HESAPLAMA

Uğur Koç

Bilgisayar Bilimleri ve Mühendisliği, Yükseklisans Tezi, 2014

Tez Danışmanı: Yar. Doç. Cemal Yılmaz

Anahtar Kelimeler: Yazılım kalite güvencesi, kombinatoryal etkileşim testi, kapsayan diziler, test durumlarını dikkate alan kapsayan diziler, benzetilmiş tavlama

Özet

Yapılandırılabilirliği yüksek yazılım sistemlerinin eksiksiz bir şekilde test edilmesi pratikte olanaksızdır. Bu nedenle, konfigürasyon uzayının verimli bir şekilde örneklendirilmesi testlerin kapsamını artırmak için önemlidir.

Bu amaca yönelik geliştirilen t-yollu kapsayan diziler (t-way covering arrays) (KAD), konfigürasyon seçeneklerinin bütün t-yollu kombinasyonları için bütün değer kombinasyonlarını kapsamak üzere sistematik bir şekilde oluşturulmuş bir konfigürasyon kümesidir. KAD'lar konfigürasyon seçeneklerinin etkileşimlerinden kaynaklanan hataları keşfetmeyi hedeflemektedir. Günümüzde, elde ettikleri birçok başarıya rağmen, pratikte KAD'ları kullanmak zor olabilir.

Bir t-yollu KAD oluşturduktan sonra, diziye seçilmiş tüm konfigürasyonlar sistemin her bir test durumu (test case) için test edilir. Böyle yaparak, geleneksel KAD'lar, test durumlarının hepsinin seçilmiş bütün konfigürasyonlarda çalışabileceğini varsayar.

Ancak yapılan son çalışmalar, yapılandırılabilirliği yüksek yazılım sistemlerinin test durumlarının üzerinde çalışacakları konfigürasyon hakkında varsayımlarının (kısıtlama) olmasının muhtemel olduğunu göstermektedir. Eğer bir konfigürasyon bir test durumunun varsayımlarına uymazsa, o test durumu o konfigürasyonu atlar ve bu da sadece o konfigürasyonda görünen değerlerinin o test durumunu tarafından test edilememesi sorununa yol açar. Bu soruna maskeleme etkisi denmektedir.

Bu sorunu çözmenin bir yöntemi, son zamanlarda geliştirilen test durumlarını dikkate alan kapsayan diziler (test-case-aware covering arrays) (T-KAD) kullanmaktır. T-KAD'lar test durumlarının konfigürasyon seçeneklerinin aldıkları değerlerle ilgili olan kısıtlamalarını hesaba katarak bu kısıtlamalarından kaynaklanan maskeleme etkilerinin oluşmasını önler. Test durumlarının kısıtlarıyla zenginleştirilmiş bir konfigürasyon uzay modeli icin hesaplanmıs bir T-KAD, geleneksel kapsayan dizilerde olduğu gibi sadece bir konfigürasyon kümesi değil, her bir konfigürasyonun bir dizi test durumuyla ilişkilendirildiği bir konfigürasyon kümesidir. Bu yapıda, bir konfigürasyonla ilişkilendirilmiş test durumları kümesi, o konfigürasyonda çalıştırılması gereken test durumlarını ifade eder.

Yapılan araştırmalarda, KAD'lar ile karşılaştırıldığında, T-KAD'ların maskeleme etkilerini ortadan kaldırarak kombinatoryal etkileşim testinin kalitesini önemli ölçüde arttırdığı gösterilmiş olmasına rağmen, kavram ispatı olarak geliştirilen birkaç algoritma haricinde, T-KAD hesaplamanın etkili ve verimli bir yöntemi yoktur. Bu sorunun, T-KAD'ların kombinatoryal etkileşim testine adapte olmasını engellediğini öngörmekteyiz.

Bu tezde, benzetilmiş tavlama-tabanlı etkili ve verimli T-KAD hesaplama algoritmaları ve bu algoritmaları uygulayan bir yazılım geliştirdik. Ayrıca, iki yapılandırılabilirliği yüksek yazılım sistemi kullanarak büyük çaplı deneyler yaparak geliştirdiğimiz algoritmaların performanslarını karşılaştırdık ve değerlendirdik. Deneylerimizin sonuçları, önerilen algoritmaların T-KAD hesaplamada verimli ve etkili bir yol olduğunu ve mevcut yaklaşımlara göre performansının daha yüksek olduğunu göstermektedir.

To the scientists who have ostracized or punished for seeking the truth.

ACKNOWLEDGMENTS

Foremost, I would like to express my sincere gratitude to my advisor Prof. Cemal Yılmaz for the continuous support of my master study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my master study.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Berrin Yanıkoğlu, Prof. Bülent Çatay, Prof. Erkay Savaş, and Prof. Hüsnü Yenigün for their encouragement, insightful comments, and hard questions.

My sincere thanks also goes to Gülşen Demiröz for her great research cooperation in thesis project, insightful comments and support of my master study.

I thank my fellow labmates in Software Research Group: Hanefi Mercan, Arsalan Javeed, Yusuf Külah, for the stimulating discussions, and research cooperations. Also I thank my friends in Sabanci University: Mehmet Ahat, Salim Sarımurat, Gizem Gezici, Zeynep Doğmuş, and Rahim Dehkharghani for all the fun we have had in the last two years.

Last but not the least, I would like to thank my family: my parents Ahmet Koç and Emine Sönmez, for giving birth to me at the first place, my brother and sisters Suat Alışkan, Aslı Filya and Suna Alışkan for supporting spiritually me throughout my life and my dear darling Meryem Yılmaz for being in my life and empowering me with her great love.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

ϕ_t	t-tuple.
Φ_t	set of all valid t-tuples.
λ_t	t-pair.
Λ_t	set of valid t-pairs.
τ	test case.
T	test suit (set of test cases).
Q	set of test case-specific constraints.
Ω_t	t-way covering array.
$\prod$	covering array generator.
Ψ_t	t-way test case-aware covering array.

LIST OF ABBREVIATIONS

CS	Computer Science.
SA	Simulated Annealing.
CIT	Combinatorial Interaction Testing.
CA	Covering Array.
TCA	Test Case-Aware Covering Array.
BS	Binary Search.
NS	Neighboring Strategy.
IS	Initialization Strategy.
SUT	Software Under Test.
SQA	Software Quality Assurance.
HIS	Hamming Distance Initial Set.
TIS	Traditional Covering Array as Initial Set.
RIS	Random Initial Set.
TCIS	Test Case-Aware Covering Array as Initial Set.
CRI	Change a Random Index.
CRT	Change a Random Tuple.
SMT	Schedule More Test Cases.
CMP	Cover at least one Missing t-Pair.
AVO	Alter Violating Option.

1

INTRODUCTION

Software is a fundamental component of modern life. People engage with software to overcome many tasks of daily life such as driving car, watching TV, shopping and learning. In many application domains, variety of requirements and diversity of environments force software systems to be highly configurable. For example, Apache Web Server has 172 user-configurable options to support customization for different requirements and environments.

While having highly configurable system promotes customization, it introduces testing problems with regret. Number of possible configurations grows exponentially with number of configurable factors; therefore, exhaustive testing of all possible configurations becomes practically infeasible. For example, with 172 configuration options, Apache Web Server has 1.8×10^{55} unique configurations. Testing all possible configurations for such a system takes longer than the age of universe, thus infeasible. For this reason, to improve the coverage of software testing, efficient sampling from the configuration space is a vital problem for software quality assurance.

Combinatorial interaction testing (CIT) is an effective method that has been commonly studied for this purpose [33]. CIT aims to improve the coverage of testing by revealing failures that are caused by the interactions of various system input parameters. As input, CIT approaches take a configuration space model which includes a set of configurable factors, their possible settings, and a set of system-wide inter-option constraints that explicitly or implicitly invalidate some configurations. They then systematically sample the configuration space based on some coverage criteria.

In CIT, a common criteria is to cover all t-way combinations of configuration options, where t is referred to as the coverage strength. Typically, this criteria is satisfied through the use of a combinatorial structure called t-way covering array (CA). A t-way CA is a set of systematically selected configurations covering all value combinations for every t-way option combinations. The goal is to discover faults that are caused by interactions of t (and fewer) configuration options. The results of many empirical studies strongly suggest that a majority of such failures in practice, are caused by the interactions of only a small number of configuration options. Thus, t-way covering arrays, where t is much smaller than the number of possible configurable factors, are an effective and efficient way of revealing such failures [2, 9, 13, 14]. CAs are currently being used in many application domains, and a wide variety of free and commercial tools exist to generate them. Despite its many successes, practical application is challenging for it.

Once a t-way CA is constructed, the system is then tested by running its test cases in all configurations of the covering array. By doing so, it is assumed that all test cases can run in all configurations of the CA. However, test cases of configurable systems are likely to have assumptions about the underlying configurations. Thus, it is not enough to satisfy the system-wide constraints to execute each test case for each configuration of the CA. When a configuration does not satisfy the assumptions of a test case, that test case simply skips that configuration and which causes the masking effects [15].

Figure 1.1 illustrates masking effects on a system that has four binary configuration options (o_1, o_2, o_3, and o_4) and a test suite containing three test cases (t_1, t_2 and t_3, as in Figure 1.1(a)). In this example, there is no system-wide constraint. However, test cases t_1 and t_2 have some self-specific constraints: t_1 can run only in configurations in which o_1=0, and t_2 can run only in configurations in which o_1=1. Test case t_3, on the other hand,

Configuration Space Model	
option	**settings**
o_1	{0, 1}
o_2	{0, 1}
o_3	{0, 1}
o_4	{0, 1}

(a)

3-way CA						
o_1	o_2	o_3	o_4	t_1	t_2	t_3
1	1	1	1	S	E	E
1	1	0	0	S	E	E
1	0	1	0	S	E	E
1	0	0	1	S	E	E
0	1	1	0	E	S	E
0	1	0	1	E	S	E
0	0	1	1	E	S	E
0	0	0	0	E	S	E

(b)

Figure 1.1: Input (a) and output (b) of CIT.

has no test case-specific constraints. For this configuration model, a 3-way covering array is created and then all the test cases are executed in all configurations of the covering array (Figure 1.1(b)). The literal E indicates that the test is executed, and the literal S indicates that the test skipped the configuration because of the unsatisfying option setting(s).

There are twenty valid 3-tuples to be tested by each of t_1 and t_2 and 32 valid 3-tuples for t_3. Now consider t_1; since t_1 skipped the first 4 configurations, the 3-way option setting combinations for options o_2, o_3, and o_4 that appear in the first four configurations, were actually not tested by t_1. These 4 combinations appear nowhere else in the covering array, thus t_1 never had a chance to test them. Similarly, t_2 never had a chance to test the four valid 3-way combinations that appear in the configurations skipped by t_2. As a result, eight out of 72 (11%) valid 3-way option setting combination-test case pairs were not tested at all, masked.

In order to avoid this kind of masking effects caused by existence of test case-specific constraints, a new combinatorial object -test case-aware covering array (TCA)- is introduced by Yilmaz et al. [32] and CIT became aware of the test case-specific constraints. As input, test case-aware CIT takes a configuration space model which includes a set of configurable factors, their possible settings, and a set of system-wide and test case-specific inter-option constraints that explicitly or implicitly invalidate some configurations system-widely or on test case bases (such as Figure 1.2(a)). Then, systematically sample the configuration space based on satisfying some coverage criteria and create a test case-aware covering array.

A test case-aware covering array is not just a set of configurations as is the case in traditional covering array, but a set of configurations, each of which is associated with a set of test cases, indicating that the test cases are scheduled to be executed in the configuration. Figure 1.2(b), as an example, presents a 3-way test case-aware covering array constructed for our hypothetical scenario.

Configuration Space Model	
option	**settings**
o_1	{0, 1}
o_2	{0, 1}
o_3	{0, 1}
o_4	{0, 1}
test suit: t_1, t_2, t_3	
test	**constraint**
t_1	o_1=0
t_2	o_1=1

(a)

3-way TCA				
o_1	o_2	o_3	o_4	scheduled tests
0	1	1	1	$\{t_1\}$
1	1	1	1	$\{t_2, t_3\}$
0	1	0	0	$\{t_1\}$
1	1	0	0	$\{t_2, t_3\}$
0	0	1	0	$\{t_1\}$
1	0	1	0	$\{t_2, t_3\}$
0	0	0	1	$\{t_1\}$
1	0	0	1	$\{t_2, t_3\}$
0	1	1	0	$\{t_1, t_3\}$
1	1	1	0	$\{t_2\}$
0	1	0	1	$\{t_1, t_3\}$
1	1	0	1	$\{t_2\}$
0	0	1	1	$\{t_1, t_3\}$
1	0	1	1	$\{t_2\}$
0	0	0	0	$\{t_1, t_3\}$
1	0	0	0	$\{t_2\}$

(b)

Figure 1.2: Input (a) and output (b) of test case-aware CIT.

A t-way test case-aware covering array has the following properties:

1. For each test case, every valid t-way combination of option settings occurs at least once in the set of configurations in which the test case is scheduled to be executed,
2. No test case is scheduled to be executed in a configuration which violates the test case-specific constraints of the test case, or the system-wide constraints.

Having stated the improvements of awareness of test case-specific constraints, except for a couple of proof-of-concept algorithms introduced by Yilmaz et al. [32], there is no effective algorithm or tool to generate test case-aware covering arrays. Although, CA and TCA generation problems are similar, since TCAs are more complex objects with test case-specific constraints, generating them is a more challenging problem compared to

generating a traditional CA. For example, for a configuration space model with 65 binary configuration options and 30 distinct test case-specific constraints, conventional greedy algorithms take seventeen days to generate a 3-way test case-aware covering array [32], whereas generating a 3-way CA for the same model is just a matter of minutes.

In this thesis, we focused on test case-aware covering array generation problem. We have assessed existing covering array generation methods and investigated their weaknesses to solve this problem. Finally, we focused on simulated annealing algorithm, which has been commonly used for covering array generation task as well, to compute TCAs.

We have developed simulated annealing-based, efficient and effective algorithms to compute test case-aware covering arrays and a tool implementing these algorithms. We, furthermore, compare and contrast the performance of our algorithms by conducting large-scale experiments in which we used two highly configurable large software systems. The results of our empirical studies strongly suggest that the proposed algorithms are an efficient and effective way of computing test case-aware covering arrays and that they perform better than existing approaches.

Our contribution can be summarized as follows:

- design of a new methodology to compute test case-aware covering arrays,
- a tool for test case-aware (and also traditional) covering array generation,
- discovery of new bounds for test case-aware covering array sizes, and
- significant cost reduction in test case-aware covering array generation.

The remainder of this article is organized as follows: chapter 2 provides background information; chapter 3 discusses the related work; chapter 4 presents the proof of optimality for Algorithm 3 introduced in [32]; chapter 5 introduces the proposed approach to compute test case-aware covering arrays; chapter 6 presents the empirical studies; chapter 7 discusses the potential external threats to validity; and chapter 8 presents concluding remarks and possible directions for future work.

2

BACKGROUND INFORMATION

This chapter provides background information about traditional covering arrays, masking effects, test case-aware covering arrays, and simulated annealing algorithm.

2.1. Combinatorial Interaction Testing

Combinatorial interaction testing aims to improve the coverage of testing by revealing failures that are caused by the interactions of various system input parameters. At a high level, CIT can be broken down into four major phases as shown in Figure 2.1.

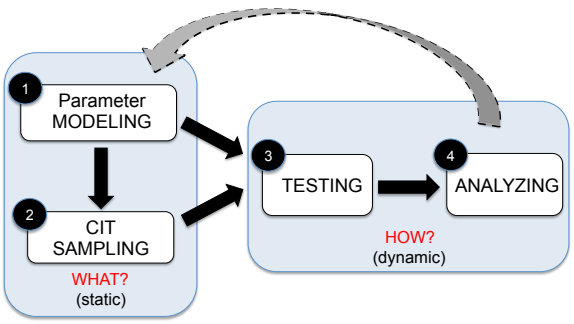

Figure 2.1: Four Phases of CIT

The first two of these phases, modeling and sampling, basically address the 'WHAT' of testing - what are the characteristics of the SUT, and what are the inputs against which it should be tested? Modeling involves determining what aspect of the system to model (i.e., input parameters, configuration options, sequences of operations). Sampling refers to the process or algorithm by which we determine a means to cover the model generated in the first phase (e.g., all pairs of all factors, etc.). Currently, these phases are typically static, done once at the beginning of the process.

The last two phases, testing and analysis, typically address the 'HOW' of testing actually running the tests and then examining the test results. These phases tend to be more process-driven than the first two phases, unfolding over a more extended period of time. In testing, developers may test in a batch mode, or test more incrementally or adaptively. And finally, developers analyze the test results, at a minimum to understand which test cases have passed and which have failed. In some cases, developers can use the testing and analysis phases to provide feedback to improve and refine later modeling and sampling activities.

2.2. Traditional Covering Arrays

CIT approaches take a *configuration space model* $M=<O, V, Q>$ as input. The model includes a set of configuration options $O = \{o_1, o_2, \ldots, o_n\}$, their possible settings $V = \{V_1, V_2, \ldots, V_n\}$, and a system-wide inter-option constraint Q (if any). In effect, the configuration space model implicitly defines a valid configuration space for testing.

Each option o_i ($1 \leq i \leq n$) in the configuration space model takes a value from a finite set of k_i distinct values $V_i=\{v_1, v_2, \ldots, v_{k_i}\}$ ($k_i=|V_i|$). Let s_{ij} be an option-value tuple of the form $<o_i, v_j>$, indicating that option o_i is set to value $v_j \in V_i$. Furthermore, let S_i be the set of all possible option-value tuples for option o_i, i.e., $S_i=\{<o_i, v_j>: v_j \in V_i\}$.

Definition 1. *A t-tuple* $\phi_t=\{s_{i_1 j_1}, s_{i_2 j_2}, \ldots, s_{i_t j_t}\}$ *is a set of option-value tuples for a combination of t distinct options, such that* $1 \leq t \leq n$, $1 \leq i_1 < i_2 < \cdots < i_t \leq n$, *and* $s_{i_p j_p} \in S_{i_p}$ *for* $p=1, 2, \ldots, t$.

Let $\hat{\Phi}_t$ be the set of all t-tuples for some $1 \leq t \leq n$. Not all the t-tuples in $\hat{\Phi}_t$ may be valid due to the system-wide constraint Q. Assume a deterministic function $valid(\phi_t, Q)$, such that $valid(\phi_t, Q)$ is true, if and only if, ϕ_t is a valid t-tuple under constraint Q. Otherwise, $valid(\phi_t, Q)$ is false. The set of all *valid* t-tuples Φ_t under constraint Q is then defined as: $\Phi_t = \{\phi_t : \phi_t \in \hat{\Phi}_t \wedge valid(\phi_t, Q)\}$.

Definition 2. *Given a configuration space model $M=<O, V, Q>$, a valid configuration c is a valid n-tuple, i.e., $c \in \Phi_n$, where $n = |O|$.*

Note that, in a valid configuration, each option defined in the configuration space model takes a valid value and the configuration (i.e., n-tuple) does not violate Q.

Definition 3. *Given a configuration space model $M=<O, V, Q>$, the valid configuration space C is the set of all valid configurations, i.e., $C=\{c : c \in \Phi_n\}$.*

CIT approaches systematically sample the valid configuration space and test only the selected configurations. The sampling is carried out by computing a t-way covering array [9], where t is often referred to as the *strength* of the covering array.

Definition 4. *A t-way covering array $CA(t, M=<O, V, Q>)$ is a set of valid configurations in which each valid t-tuple appears at least once, i.e., $CA(t, M=<O, V, Q>)=\{c_1, c_2, \ldots, c_N\}$, such that $\forall\, \phi_t \in \Phi_t\, \exists\, c_i \supseteq \phi_t$, where $c_i \in C$ for $i=1, 2, \ldots, N$.*

Once a covering array is computed, the system under test is validated by running its test suite in all the selected configurations. Since the amount of resources required for testing is a function of the covering array size (i.e., N), covering arrays are constructed so that all valid t-tuples are covered in minimum number of configurations.

2.3. Masking Effects

Definition 5. *A masking effect is an effect that prevents a test case from testing all valid combinations of option settings appearing in a configuration, which the test case is normally expected to test.*

The concept of masking effects has been introduced by Dumlu et al. [15]. A harmful consequence of masking effects is that they cause developers to develop false confidence in their test processes, believing them to have tested certain option setting combinations, when they in fact have not. One simple example of a masking effect (besides the ones caused by overlooked test case-specific constraints) would be an error that crashes a program early in the programs execution. The crash then prevents some configuration dependent behaviors, that would normally occur later in the programs execution, from being exercised. Unless the combinations controlling those behaviors are tested in a different configuration, or unless the error is fixed and the faulty configuration is re-tested, we cannot conclude that those configuration dependent behaviors have been tested.

Masking effects can be caused by many factors. System failures, unaccounted control dependencies among configuration options (i.e., option setting combinations that effectively cancel other option setting combinations), and incomplete or incorrect inter-option constraints can all perturb program executions in ways that prevent other configuration dependent behaviors from being tested.

2.4. Test Case-Aware Covering Arrays

Definition 6. *An inter-option constraint is a constraint among option settings, which explicitly or implicitly invalidates some combinations of option settings.*

System-wide inter-option constraints apply to all test cases and define the set of valid ways the system under test can be configured. A test case-specific constraint, on the other hand, applies only to the test case that it is associated with and determines the configurations in which the test case can run.

It is important to note that expressing test case-specific constraints as system-wide constraints and then generating traditional covering arrays, is not an adequate solution for handling test case-specific constraints. One reason is that constraints for different test cases may conflict with each other, in which case no solution will be found. For example, in our hypothetical scenario discussed in Chapter 1, the constraints of t_1 and t_2 conflict;

t_1 cannot run when the binary option o_1 has one setting and t_2 cannot run when the same option has the other setting. Globally enforcing these conflicting constraints will not generate any covering arrays. Another reason is that, even if the test case-specific constraints do not conflict, enforcing them on all test cases can prevent the test cases from exercising some valid option setting combinations that are invalidated by other test cases. For example, in our hypothetical scenario given in Chapter 1, enforcing the constraint of t_1 on t_3 prevents t_3 from testing any combinations with o_1=1, which are valid for t_3.

For these reasons, we need to account the test case-specific constraints individually. Test case-aware covering arrays have been introduced for this purpose by Yilmaz et al. [32]. As is the case with traditional covering arrays, test case-aware covering arrays take as input a configuration space model M. The model contains a set of configuration options O=$\{o_1, \ldots, o_n\}$, their settings V=$\{V_1, \ldots, V_n\}$, and a system-wide inter-option constraint Q_s. Unlike traditional covering arrays, the configuration space model of test case-aware covering arrays additionally includes a set of test cases T=$\{\tau_1, \tau_2, \ldots\}$. Each test case $\tau \in$ T, in addition to implicitly inheriting the system-wide constraint Q_s, can have a test case-specific constraint Q_τ. In the remainder of the paper, the collection of all test case-specific constraints is referred to as Q_{T}.

In the presence of test case-specific constraints, we define the set of valid t-tuples on a per-test case basis, since a valid t-tuple for a test case may be invalid for another test case. Let Φ_t^τ=$\{\phi_t : \phi_t \in \hat{\Phi}_t \wedge valid(\phi_t, Q_s \wedge Q_\tau)\}$ be the set of all valid t-tuples for a test case τ.

Definition 7. *A valid configuration c^τ for a test case $\tau \in$ T is a valid n-tuple for τ, i.e., $c^\tau \in \Phi_n^\tau$, where $n = |O|$.*

Definition 8. *The valid configuration space C^τ for a test case $\tau \in$ T is the set of all valid configurations for τ, i.e., C^τ=$\{c : c \in \Phi_n^\tau\}$.*

Test case-aware covering arrays aim to ensure that each test case has a fair chance to test all of its valid t-tuples. To this end, each test case is scheduled to be executed only in configurations which are valid for the test case so that no masking effects can occur.

Definition 9. *A t-pair is a pair of the form λ_t=<ϕ_t, τ>, such that $\phi_t \in \Phi_t^\tau$ and $\tau \in$ T.*

Definition 10. *A t-way test case-aware covering array* $TCA(t, M{=}<O, V, \mathrm{T}, Q_s, Q_\mathrm{T}>) = \{<c_1, \mathrm{T}_1>, \ldots, <c_N, \mathrm{T}_N>\}$ *is a set of rows of the form* $<c_i, \mathrm{T}_i>$*, where* $c_i \in C$ *and* $\mathrm{T}_i \subseteq \mathrm{T}$ *for* $i = 1, 2, \ldots, N$*, such that each valid t-pair appears at least once, i.e.,* $\forall\, \tau \in \mathrm{T} \wedge \phi_t \in \Phi_t^\tau\ \exists\, <c_i, \mathrm{T}_i> : \phi_t \subseteq c_i \wedge \tau \in \mathrm{T}_i$ *and* $\tau \in \mathrm{T}_i \rightarrow c_i \in C^\tau$.

In other words, for a given configuration space model, a t-way test case-aware covering array is a set of configurations, each of which is associated with a set of test cases, indicating the test cases scheduled to be executed in the configuration, such that 1) none of the selected configurations violate the system-wide constraint, 2) no test case is scheduled to be executed in a configuration that violates the test case-specific constraint of the test case, and 3) for each test case, every valid t-tuple appears at least once in the set of configurations in which the test case is scheduled to be executed. Figure 1.2b, as an example, presents a 3-way test case-aware covering array created for our hypothetical scenario depicted in Figure 1.1. Since none of the test case-specific constraints are violated in this covering array, each test case has a chance to test all of its valid 3-tuples; no masking effects caused by test skips can occur.

Compared to traditional t-way covering arrays, handling test case-specific constraints is likely to increase the number of configurations required, as the t-tuples being masked in traditional arrays may need to be covered in additional configurations. However, this does not necessarily imply an increase in the number of test runs required, as the test cases are executed only in configurations that contribute to the coverage. For example, comparing the 3-way test case-aware covering array in Figure 1.2b to the traditional 3-way covering array in Figure 1.1b, we observe that, while the number of configurations doubles, the number of test runs stays the same as each array requires a total of 24 test runs.

Therefore, when the goal is to test all valid t-pairs, then traditional t-way covering arrays will not guarantee the coverage in the presence of test case-specific constraints, whereas t-way test case-aware covering arrays, while guaranteeing a full coverage, will do so at the possible cost of increased number of configurations, but not necessarily increased number of test runs.

2.5. Simulated Annealing

Simulated Annealing (SA) is a stochastic optimization method emulated from metal annealing process [8, 19]. Physical annealing is the process of cooling high temperature molten metal at a significant rate to have frozen metal with minimum potential energy at the end. There are three control points in physical annealing process; beginning temperature T_0, cooling rate C_r and stopping temperature T_s. All of these parameters are important to reach to the minimum potential energy, and they also affect the annealing time.

At high temperatures particles are more susceptive to movement. Therefore, more drastic changes and high energy releases are likely to occur at early phases of annealing. As the process goes on, particles get stabilized and it becomes difficult to happen big structural changes. The process terminates when the temperature reaches to T_{end} or potential energy becomes 0. If C_r is not small enough frozen metal will contain imperfections caused by unreleased energy. Or vice versa, if the cooling rate is too small then the frozen metal will be too softened to work with.

SA mimics this process to solve optimization problems. Energy corresponds to cost and the state of metal with minimum potential energy corresponds to the optimal solution with minimum cost. T_0, C_r, and T_s are referred as annealing parameters and they are used to control the search process. Annealing parameters should be determined by the needs of the problem domain.

To avoid stacking in local minimums, SA algorithm applies some probability to create a chance for accepting the states that are worse that the current state. If the newly generated neighbor S' is more costly than S, SA invokes bolzman probability distribution function;

$$B(T) = -k_b \frac{\Delta E}{T} \tag{2.1}$$

and check for the following condition (5^{th} line of Algorithm 1);

$$Rand(0, 1) < e^{B(T)} \tag{2.2}$$

Algorithm 1 Simulated Annealing

Input T_0, C_r, T_s: Annealing Parameters

1: $T \leftarrow T_0$ $S \leftarrow S_0$ $S_{best} \leftarrow S_0$
2: **while** $E(S_{best}) \neq 0$ and $T_s < T$ **do**
3: $S' \leftarrow neighbour(S)$ # *generate a neighbour* S'
4: $\Delta E \leftarrow E(S') - E(S)$ # $E(S)$ *energy of state* S
5: **if** $\Delta E < 0$ or $Rand(0,1) < e^{-k\Delta E/T}$ **then**
6: $S \leftarrow S'$ # *change the state*
7: **if** $E(S) < E(S_{best})$ **then**
8: $S_{best} \leftarrow S$ # *save to the* S_{best}
9: **end if**
10: **end if**
11: $T- \leftarrow (T \times C_r)$ # *cool the system*
12: **end while**
13: **return** S_{best}

If the condition holds, SA continues with S' otherwise rejects S' and continues with S. This probabilistic decision keeps SA to get stacked in local minimums.

Since the temperature is higher at the early phases of the search, the condition (2.2) is more likely to hold, means, SA is more open to accept worse states. Therefore the temperature interval, (T_s, T_0), is important to effectively scan the search space. C_r helps to cool the system which will effect the acceptance of worse states.

Finally, if the optimum solution cannot be found, SA stops when $T = T_s$. In this case, which is the wort case for complexity analysis, there will be

$$(T_0 - T_s)/C_r \tag{2.3}$$

iterations with decreasing temperature and at the end the system will be cold.

Although, SA is not a deterministic and complete algorithm which exhaustively scan the entire search space, in practice it achieves to find the optimum solution and commonly used to solve NP-hard problems.

3

RELATED WORK

Traditional covering arrays aim at revealing interaction-related failures. The results of many empirical studies strongly suggest that a majority of such failures in practice are caused by the interactions of only a small number of configurable factors or input parameters and that traditional t-way covering arrays, where t is much smaller than then the number of possible configurable factors, are an effective and efficient way of revealing such failures [2, 9, 13, 14].

3.1. Covering Array Generation

Nie et al. classify the methods for generating covering arrays, which is an NP-hard problem, into 4 main categories [23]: random search-based methods [24], heuristic search-based methods [6, 10, 12, 17, 25], mathematical methods [18, 20, 30, 31], and greedy methods [3, 5, 7, 9, 13, 21, 27, 29].

Random search-based methods employ a random selection without replacement strategy [24]. Valid configurations are randomly selected from the configuration space in an iterative manner until all the required t-tuples have been covered by the selected configurations.

Mathematical methods for constructing covering arrays have also been studied [20,30,31]. Some mathematical methods are based on recursive construction methods, which build covering arrays for larger configuration space models (i.e., the ones with a larger number of configuration options) by using covering arrays built for smaller configuration space models [20, 30]. Other mathematical methods leverage mathematical programming, such as integer programming, to compute covering arrays [31].

Greedy algorithms operate in an iterative manner [3,5,7,9,13,21,27,29]. At each iteration, among the sets of configurations examined as candidates, the one that covers the most previously uncovered t-tuples is included in the covering array. The iterations terminate when all the required t-tuples have been covered.

Heuristic search-based methods, on the other hand, employ heuristic search techniques, such as hill climbing [12], tabu search [6], and simulated annealing [10, 28], or AI-based search techniques, such as genetic algorithms [17] and ant colony algorithms [25]. These methods maintain a set of configurations at any given time and iteratively apply a series of transformations to the set until the set constitutes a t-way covering array. These methods do not search all the search space exhaustively. Therefore, theoretically they are not sound. However, in practice some of these methods achieve to find a covering array in reasonable construction cost and size.

3.2. Constraint Handling

Handling system-wide inter-option constraints in the construction of traditional covering arrays have also been of interest. Cohen et al. study the nature of such constraints in configurable software systems and empirically demonstrate that ignoring such constraints leads to wasted testing efforts [11]. Mats et al. propose various techniques to efficiently handle system-wide constraints [22]. Bryce et al. introduce the concept of "soft constraints" to mark option setting combinations that are permitted, but undesirable to be included in a covering array [4].

Traditional covering arrays, while handling system-wide constraints, do not account for test case-specific constraints. In this work we, on the other hand, take test case-specific constraints into account when constructing combinatorial interaction test suites.

Seeding mechanisms in CIT approaches have been used to guarantee the inclusion of certain configurations in traditional covering arrays [9, 13, 16]. In this work, we use the seeding mechanism to construct test case-aware covering arrays.

3.3. Test Case-Aware Covering Array Generation

Since, test case-aware covering arrays have been introduced recently, there are only 3 proof-of-concept algorithms that also have been introduced with the object [32].

- Algorithm 1: Maintaining a separate configuration space model for each test case,
- Algorithm 2: Maintaining a single configuration space model,
- Algorithm 3: Minimizing number of test runs.

These algorithms fall into the category of greedy algorithms. However, while the existing greedy algorithms compute traditional covering arrays, they compute test case-aware covering arrays and each has different objective.

4

ALGORITHM 3: MINIMIZING NUMBER OF TEST RUNS

The algorithm presented in this section is introduced by Yilmaz et al. [32] and it aims to minimize the number of test runs.

Algorithm 2 Minimizing the number of test runs required.

Input M=<O, V, T, Q_s, Q_T>: Config. space model
Input t: Covering array strength

1: $\Psi_t^M \leftarrow \emptyset$
2: **for each** test case τ **in** T **do**
3: $\quad S_\tau \leftarrow \emptyset$
4: $\quad \Omega_t^{M_\tau} \leftarrow \prod(t, M_\tau, S_\tau)$
5: $\quad \Psi_t^M \leftarrow \Psi_t^M \bullet \Omega_t^{M_\tau}$
6: **end for**
7: **return** Ψ_t^M

Given a configuration space model M=<O, V, T, Q_s, Q_T>, strength t, and using an existing traditional covering array constructor $\prod$, this algorithm generates a t-way test case-aware covering array, Ψ_t, by concatenating |T| number of t-way covering arrays, $\Omega_t^{M_\tau}$, each of which is created for M_τ where $\tau \in \text{T} \wedge \forall\, \tau \in \text{T}$ and M_τ=<O, V, $\{\tau\}$, Q_τ, $\emptyset$> only has one system-wide constraint which is originally the test case specific constraint of τ. Therefore, there is only one test case scheduled to execute for each configuration (row) of Ψ_t.

4.1. Proof of Optimality

Assuming that, $\prod$ computes covering array that are optimum in size proof of optimality of this algorithm can be done as follows;

Statement; Algorithm 3 is optimum in the number of test runs.

1. **Basis** $|\mathrm{T}| = 1$; constructed $N \times k$ covering array is optimum minimum in N. There will be

$$N \times 1 = N \tag{4.1}$$

   test runs, so number of test runs is also N which is optimum, statement holds for $|\mathrm{T}| = 1$.

2. **Inductive step**; assuming that the statement holds for $|\mathrm{T}| = n$. For each test case $\tau \in \mathrm{T}$, $\prod$ will construct $N_i \times k$ covering array meaning $N_i \times 1 = N_i$ test runs. In total;

$$\sum_{i=1}^{n} N_i \tag{4.2}$$

   is the number of test runs, which is optimum.

   Then, for $|\mathrm{T}| = n + 1$;

$$\left(\sum_{i=1}^{n} N_i \right) + N_{n+1} \tag{4.3}$$

   The first term is optimum from (4.2) and the second term is optimum from (4.1). Thus, (4.3) is also optimum in number of test run, statement also holds for $|\mathrm{T}| = n + 1$.

5

APPROACH

This chapter presents architectural design, search levels, and initialization and neighboring strategies of the proposed approach.

5.1. Architectural Design

In order to generate test case-aware covering arrays, we have designed a nested search process with two levels. The outer level, referred to as the outer search, is the search of the minimal size for the test case-aware covering array. The inner level on the other hand, referred to as the inner search, is the actual search of the test case-aware covering array for the size determined by the outer search.

As input, the approach takes a configuration space model, M=<O, V, T, Q_s, Q_T>, coverage strength, t, and optionally a seed to start with. The output is a t-way test case-aware covering array.

The following two sections elaborate on the search levels of the approach.

5.2. Binary Search for The Outer Search

Many covering array generation approaches, which use heuristics, generate covering arrays for a given size. There are known array size bounds for a large number of configuration space models published by Nist [1]. However, no work has been done to discover test case-aware covering array size bounds yet. Indeed, since the test case-specific constraints are application specific, sizes for test case-aware covering arrays cannot be generalized as in the case of traditional covering arrays. Therefore, the array size cannot be given as an input. Due to this reason, we have designed the outer search.

We used binary search algorithm for the outer search (in Algorithm 3). The interval (lower and upper bounds, B_l and B_u) of the search is determined relatively to the published covering array sizes [1].

Algorithm 3 Binary Search for TCA size

Input M=<O, V, T, Q_s, Q_{T}>: Configuration space model
Input t: Covering array strength
Input <B_l, B_u>: Lower and upper bounds
Input S_0: Seed

1: $N \leftarrow (B_l + B_u)/2$
2: $S_0 \leftarrow initialize(M, N, t, S_0)$ $S \leftarrow S_0$
3: **while** $B_u \geq B_l$ **do**
4: $S \leftarrow run(M, N, t, S)$ # *running the inner search for* Ψ_t *of size* N
5: **if** $E(S) = 0$ **then**
6: $\Psi_t \leftarrow S$ # *keep as the best so far*
7: $B_u \leftarrow N - 1$
8: **else**
9: $B_l \leftarrow N + 1$
10: **end if**
11: $N \leftarrow (B_l + B_u)/2$ # *new size*
12: **end while**
13: **return** Ψ_t

First, the initial array size is computed as the average of the bounds. Then, the system is initialized as in the 2^{nd} line of Algorithm 3. This initialization operation is a crucial component of the approach which will affect the computation cost. Therefore, we have developed several initialization strategies which will be elaborated in Section 5.4.

Then, iterative search starts. At each iteration, first an inner search is performed for the computed array size, N (4^{th} line of Algorithm 3). For the next iteration, if the inner search can find the Ψ_t, then, N is the new upper bound, otherwise N is the new lower bound. The function, $E(S)$, computes the number of missing t-pairs as the cost of S. Finally, the outer search terminates when $B_u \geq B_l$ and returns the best found Ψ_t (e.g. $E(\Psi_t) = 0$).

5.3. Simulated Annealing for The Inner Search

The inner search is developed to compute test case-aware covering array, Ψ. We first assessed existing covering array generation methods which are also mentioned in Section 3.1.

Mathematical methods are not effective in constraint handling and they put unrealistic requirements for configuration space model, such as, having prime number of configuration options or having the same setting count for each option [23].

Random search-based methods are more flexible compared to mathematical methods and they have been commonly used to compute covering arrays [24]. However, compared to greedy or local search algorithms they are less effective.

Greedy algorithms work faster than local search algorithms but they produce covering arrays that are larger in size [23]. Throughout the search, greedy methods complete the array gradually. They make decisions only with the current local information of the selected configurations and do not account the needs of the proceeding steps. However, having a complete solution object and being aware of the needs of the system are important for the test case-aware covering array generation task.

Local search-based methods on the other hand, work with a complete solution object and are aware of the needs of the system. Compared to other methods, they are also more suitable and effective for constraint handling. Therefore among the others, local search-based methods are the most suitable ones for the task.

Stardom et al. [26] compared the performance of tabu search, genetic algorithm and simulated annealing, on the covering array generation task. Their empirical study have suggested that simulated annealing algorithm, which is a commonly used local search algorithm, was more effective in finding covering arrays that are smaller in size. For these reasons, we used simulated annealing (SA) algorithm, described in Chapter 2.5, to compute test case-aware covering arrays.

In our use of SA algorithm, components of the (inner) search are defined as follows:

The **state**, S, is a set of configurations each of which is associated with a set of test cases, indicating that the test cases are scheduled to be executed in the configuration.

$$S = \{< c_1, \mathrm{T}_1 >, \ldots, < c_N, \mathrm{T}_N >\}$$

where $c_i \in C$ and $\mathrm{T}_i \subseteq \mathrm{T}$ for $i = 1, 2, \ldots, N$.

The **cost** of S, $E(S)$, is the number of t-pairs, λ_t, that are not covered by S (missing t-pairs).

$$E(S) = |\Lambda_t^{missing}| \tag{5.1}$$

where

$$\Lambda_t^{missing} = \{\lambda_t =< \phi_t, \tau >: \tau \in \mathrm{T} \wedge \ \phi_t \in \Phi_t^{\tau} \wedge \neg\exists \ < c_i, \mathrm{T}_i > \in \ S \ : \phi_t \subseteq c_i \wedge \tau \in \mathrm{T}_i\}.$$

The **action**, is a transition performed on S (will be elaborated in Section 5.5).

Finally, the **goal** is to find an S with $E(S)$=0 which is a Ψ_t.

SA algorithm works with a complete solution in an iterative manner. At each iteration, a new state S', called neighbor of S, is generated by applying a simple transition to S. Then, if the cost of S' is smaller than the cost of S, S' is accepted to continue with.

$$\text{if } E(S') < E(S), \text{ then } \ S = S' \tag{5.2}$$

Otherwise, the decision (accept or reject) is made based on the probability criteria of the SA algorithm as described in Chapter 2.5.

In the inner search, the neighbor generation operation is the second crucial component of the approach which affects the construction cost as well as the size of Ψ_t. Therefore, we have developed several neighboring strategies which will be elaborated in Section 5.5

5.4. Initial Set Generation Strategies

Choosing a reasonable initial state, S_0, that covers higher number of t-pairs will shorten the search (construction) time. Because there will be lower number of t-pairs left to search for. Therefore, we have developed 4 initialization strategies (IS) each of which is applying different methods to cover more t-pairs.

If there is no seed given, each of the following initialization strategies starts with an initially empty set S_0 and perform configuration selection based on their objective until S_0 has N configurations each of which is associated with a set of test cases. However, if there is a seed provided as input, then the initialization strategies perform future selection for $(N-N_0)$ times where N_0 is the size of the given seed. None of them allow system-wide constraint violations to occur.

Random Initial Set (RIS): Let a random configuration be a random assignment of each option value. Then, this initialization strategy fills S_0 as follows;

1. Generate a system-wide valid configuration c_i at random,
2. Schedule all valid test cases T_i to c_i,
3. Add the $<c_i, T_i>$ to S_0,
4. Repeat from step 1 until S_0 has N configurations each having scheduled test cases.

RIS does not depend on t, and other than system-wide constraint validation check, it does not apply any criteria for configuration selection. For these reasons, required time for RIS is always negligible compared to search time. RIS has been commonly-used for the covering array generation problem [10, 12, 23, 26, 28].

Hamming distance Initial Set (HIS): This initialization strategy fills S_0 using hamming distance formula as follows:

Definition 11. *Hamming distance in between two objects is the number of elements in which they differ.*

Let $E = \{e_1, e_2, \ldots, e_n\}$ be the set of elements for the objects, then;

$$h(o_1, o_2) = \sum_{i=1}^{|E|} d(o_1(i), o_2(i)) \tag{5.3}$$

where $o(i)$ is the i^{th} element of o, and

$$d(e_1, e_2) = \begin{cases} 1 & \text{if } e_1 \neq e_2 \\ 0 & otherwise \end{cases}$$

Perform following steps;

1. Generate the first configuration at random and add it to S_0.
2. Generate 2 candidate system-wide valid configurations c_1, c_2 at random,
3. Pick one of the candidate configurations c_1, c_2 that has larger overall hamming distance according to

$$h_{overall}(c) = \sum_{j=1}^{|S_0|} h(S(j), c) \tag{5.4}$$

   where $S(j)$ is the j^{th} configuration of S_0,
4. Schedule valid test cases T_i to c_i,
5. Add the $<c_i, \mathrm{T}_i>$ to S_0,
6. Repeat from step 2 until S_0 has N configurations.

This strategy has also been used by Torres et al. [28] in combination with RIS. HIS does not depend on t and therefore required time for it is negligible compared to search time.

t-way covering array as Initial Set (TIS): This initialization strategy generates a t-way covering array with a published array size [1], N'. Then it schedules all valid test cases for each configuration and completes the remaining $(N - N')$ configurations as in HIS strategy.

Generating traditional covering array means ignoring the test case-specific constraints. By doing so, TIS first aims to cover all the t-pairs that are not constrained. Once a t-way covering array is generated and all the valid test cases are scheduled, what remain are the masked t-pairs which will be covered in the search process. Required time for TIS is not negligible and depends on t and the configuration space model.

Ψ_{t-1} **as Initial Set (TCIS):** This initialization strategy generates a Ψ_{t-1} with size N' and completes the remaining $(N - N')$ configurations as in HIS strategy.

The relation stated below, implies that; Ψ_t also covers all valid λ_{t-1}. TCIS aims to take the advantage of this relation by starting with an initial solution that is a subset of Ψ_t.

$$\Psi_{t-1} \subset \Psi_t \tag{5.5}$$

Required time for TCIS is not negligible and depends on t, test case-specific constraints, and the configuration space model on overall.

The following section describes each of the neighboring strategies.

5.5. Neighbor Generation Strategies

Neighbor generation operation of the inner search is crucial for the effectiveness of the approach. Unsuitable neighboring strategies may keep the algorithm converging to the optimum solution and last in cold termination of annealing, whereas some intelligent strategies may reduce the construction cost drastically.

For this reason, we have also developed 5 neighboring strategies (NS) which differ in intelligence levels. Each of the following neighboring strategies performs some transition based on their objectives to generate a neighbor form S. They do not let constraint violations to happen. Therefore, all the scheduled test and configurations are valid at any point of the search.

5.5.1. Change a Random Index - CRI

This strategy changes the value of a randomly chosen option in a randomly chosen configuration to another randomly chosen valid value from the domain of the option as in Algorithm 4.

CRI steps can be summarized as follows: 1) randomly pick a configuration, 2) randomly pick a option of that configuration, 3) change the value of the option to another valid value from the domain, 4) check for system-wide constraint violation for the altered configuration and if there is a constraint violation, then turn back to step 1 otherwise, 5) update the scheduled test case list for the altered configuration, and 6) return S', which contains the altered configuration as the neighbor.

Algorithm 4 Changing a Random Index - CRI

Input M=<O, V, T, Q_s, Q_T>: Configuration space model
Input S: Current Solution

1: $Random.shuffle(S)$ # *to avoid picking the same configuration again in the loop*
2: **for all** $c : c \exists S$ **do**
3: $idx \leftarrow Random.nextInt(0, O.size)$
4: $c.getOption(idx).value \leftarrow Random.Pickfrom\{v : v \in V_{idx}\}$
5: **if** $valid(Q_s, c)$ **then**
6: $T_c \leftarrow scheduleTestCases(c, T, Q_T)$
7: $S' \leftarrow keep(S, <c, T_c>)$
8: **return** S'
9: **end if**
10: $Rollback(c)$
11: **end for**
12: **return** S # *no valid neighbor found*

5.5.2. Change a Random t-Tuple - CRT

This strategy aims at inserting a randomly chosen missing t-tuple, $\phi_t^{missing}$, into S by changing the values of referred options to the values of missing t-tuple in a randomly chosen configuration.

$$\phi_t^{missing} \in \Phi_t^{missing}$$

where

$$\Phi_t^{missing} = \{\phi_t : \lambda_t =< \phi_t, \tau > \wedge \lambda_t \in \Lambda_t^{missing}\}$$

CRT steps can be summarized as follows: 1) randomly pick a $\phi_t^{missing}$, 2) randomly pick a configuration, 3) insert $\phi_t^{missing}$ into the selected configuration, 4) check for system-wide constraint violation for the altered configuration and if there is constraint violation, then turn back to step 1, 5) update scheduled test case list for the altered configuration and return S', which contains the altered configuration, as the neighbor.

Algorithm 5 Change a Random t-tuple - CRT

Input M=<$O, V,$ T, Q_s, Q_T>: Configuration space model
Input S: Current Solution

```
1:  Random.shuffle(S)   # to avoid picking the same configuration again in the loop
2:  for all φ_t : φ_t ∈ Φ_t^missing do
3:     for all c : c ∈ S do
4:        c.changeTuple(φ_t)
5:        if valid(Q_s, c) then
6:           T_c ← scheduleTestCases(c, T, Q_T)
7:           S' ← keep(S, <c, T_c>)
8:           return S'
9:        end if
10:       Rollback(c)
11:    end for
12: end for
13: return S                    # no valid neighbor found
```

The altered configuration may not satisfy the test case-specific constraints of any of the test cases, such as $<\phi_t^{missing}, \tau>$. In that case, the update operation will not provide any benefit.

5.5.3. Schedule More Test Cases - SMT

As in CRT, this strategy also aims at inserting a randomly chosen missing t-tuple, $\phi_t^{missing}$, into S by changing the values of referred option to the values of missing t-tuple in a randomly chosen configuration. Unlike CRT, SMT also requires the altered configuration to have a larger scheduled test case list after the update.

SMT steps can be summarized as follows: 1) randomly pick $\phi_t^{missing}$, 2) randomly pick a configuration, 3) insert $\phi_t^{missing}$ into the selected configuration, 4) update scheduled test case list for the altered configuration, 5) checks for system-wide constraint violation for the altered configuration, if there is, then turn back to step 2, 6) compare the sizes of old and new scheduled test case lists, and if new test case list has smaller size turns back to step 1, 7) return the S', which contains the altered configuration, as the neighbor.

Algorithm 6 Schedule More Test Cases - SMT

Input M=<O, V, T, Q_s, Q_T>: Configuration space model
Input S: Current Solution

```
1:  Random.shuffle(S)   # to avoid picking the same configuration again in the loop
2:  for all φ_t : φ_t ∈ Φ_t^missing do
3:     for all c : c ∈ S do
4:        c.changeTuple(φ_t)
5:        T'_c ← scheduleTestCases(c, T, Q_T)
6:        if valid(Q_s, c) and |T'_c| > |T_c| then
7:           S' ← keep(S, <c, T'_c>)
8:           return S'
9:        end if
10:       Rollback(c)
11:    end for
12: end for
13: return S                  # no valid neighbor found
```

5.5.4. Cover At Least One Missing t-Pair - CMP

As in CRT, this strategy also aims at inserting a randomly chosen missing t-tuple, $\phi_t^{missing}$, into S by changing the values of referred option to the values of missing t-tuple in a randomly chosen configuration. Unlike CRT, CMP requires the altered configuration to cover at least one missing pair, $\lambda_t^{missing}$.

$$\lambda_t^{missing} \in \Lambda_t^{missing}$$

CMP steps can be summarized as follows: 1) randomly pick a $\phi_t^{missing}$, 2) randomly pick a configuration, 3) insert $\phi_t^{missing}$ into the selected configuration, 4) find the test cases that are not covered for $\phi_t^{missing}$, 5) updates scheduled test case list for the altered configuration, 6) checks for system-wide constraint violation for the altered configuration and if there is turn back to step 1, 7) intersect the new scheduled test case list with missing test case list, and if the intersection is empty set turn back to step 1, 7) return the S', which contains the altered configuration, as the neighbor.

Algorithm 7 Cover At Least One Missing t-Pair - CMP

Input M=<O, V, T, Q_s, Q_T>: Configuration space model
Input S: Current Solution

1: $Random.shuffle(S)$ # *to avoid picking the same configuration again in the loop*
2: **for all** $\phi_t : \phi_t \in \Phi_t^{missing}$ **do**
3: $T_{missing} \leftarrow \{\tau : \lambda_t =<\phi_t, \tau> \wedge \lambda_t \in \Lambda_t^{missing}\}$
4: **for all** $c : c \in S$ **do**
5: $c.changeTuple(\phi_t)$
6: $T_c \leftarrow scheduleTestCases(c, T, Q_T)$
7: **if** $valid(Q_s, c)$ and $T_c \cap T_{missing} \neq \emptyset$ **then**
8: $S' \leftarrow keep(S, <c, T_c>)$
9: **return** S'
10: **end if**
11: $Rollback(c)$
12: **end for**
13: **end for**
14: **return** S # *no valid neighbour found*

5.5.5. Alter Violating Option - AVO

As in CRT, this strategy also aims at inserting a randomly chosen missing t-tuple, $\phi_t^{missing}$, into S by changing the values of referred option to the values of missing t-tuple in a randomly chosen configuration. Unlike CRT, AVO alters the selected configuration to schedule one randomly chosen uncovered test case.

AVO steps can be summarized as follows: 1) randomly pick a $\phi_t^{missing}$, 2) randomly pick a configuration, 3) insert $\phi_t^{missing}$ into the selected configuration, 4) randomly pick a test case, τ_u, that was uncovered for $\phi_t^{missing}$, 5) check for system-wide constraint violation for the altered configuration, and if there is turn back to step 2, 6) update the scheduled test case list for the altered configuration, 7) if τ_u is not in the scheduled test case list, then alter the row to change the violating options for τ_u and return S', which contains the altered configuration, as the neighbor.

Algorithm 8 Alter Violating Option - AVO

Input M=<O, V, T, Q_s, Q_T>: Configuration space model
Input S: Current Solution

1: $\phi_t \leftarrow Random.Pick from\ \Phi_t^{missing}$
2: $Random.shuffle(S)$ # *to avoid picking the same*
3: **for all** $c : c \in S$ **do**
4: $c.changeTuple(\phi_t)$
5: **if** $valid(Q_s, c)$ **then**
6: $\tau_u \leftarrow Random.Pick from\{\tau : \lambda_t =<\phi_t, \tau> \wedge \lambda_t \in \Lambda_t^{missing}\}$
7: **if** $!valid(Q_{\tau_u}, c)$ **then**
8: $c.changeViolatingOptions(\tau_u)$
9: **end if**
10: $T_c \leftarrow scheduleTestCases(c, T, Q_T)$
11: $S' \leftarrow keep(S, <c, T_c>)$
12: **return** S'
13: **end if**
14: $Rollback(c)$
15: **end for**
16: **return** S # *no valid neighbor found*

6

EXPERIMENTS

This chapter provides information about the experiments we have conducted to evaluate the proposed approach.

6.1. Subject Applications

In the experiments, we used two highly-configurable widely-used software systems as our subject applications: Apache v2.3.11-beta and MySQL v5.1. Apache is a HTTP server. MySQL is a database management system.

We chose these systems for several reasons. First, they share the key characteristics common to configurable software systems. They are highly configurable with dozens of configuration options supporting a wide variety of features. They have a large code base and substantial test code. Both systems enjoy a large developer community that actively updates and tests the systems. Second, like many configurable software systems, developers of these systems cannot exhaustively test the entire configuration space; the number of possible configurations is far beyond the resources available to run the test cases in a timely manner, e.g., for regression testing.

For the SUT versions we have used, out of 3789 and 738 test cases examined for Apache and MySQL respectively, 378 Apache test cases and 337 MySQL test cases had some test case-specific constraints. These test cases were clustered based on their self-specific constraints by Yilmaz et al. [32]. There are 17 test clusters for Apache and 30 test clusters for MySQL.

option	**settings**
case-filter	{enable, disable}
ssl	{enable, disable}
dav	{enable, disable}
echo	{enable, disable}
rewrite	{enable, disable}
case-filter-in	{enable, disable}
bucketeer	{enable, disable}
info	{enable, disable}
headers	{enable, disable}
vhost-alias	{enable, disable}
cgi	{enable, disable}
proxy-http	{enable, disable}
proxy	{enable, disable}
test cluster list: $\{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}, t_{14}, t_{15}, t_{16}, t_{17}\}$	
system-wide constraint: proxy-http = enable $\rightarrow$ proxy=enable	
cluster idx	**test case-specific constraint**
t_1	ssl=enable $\wedge$ proxy-http=enable
t_2	ssl=enable
t_3	rewrite=enable
t_4	headers=enable
t_5	proxy=enable
t_6	dav=enable
t_7	case-filter=enable
t_8	vhost-alias=enable
t_9	proxy-http=enable
t_{10}	proxy=enable $\wedge$ rewrite=enable $\wedge$ cgi=enable
t_{11}	echo=enable
t_{12}	ssl=enable $\wedge$ headers=enable
t_{13}	rewrite=enable $\wedge$ proxy=enable
t_{14}	ssl=enable $\wedge$ case-filter-in=enable
t_{15}	case-filter-in=enable
t_{16}	bucketeer=enable
t_{17}	info=enable

Table 6.1: Initial configuration space model for Apache.

option	**settings**
log-format	{row, statement, mixed}
sql-mode	{strict, traditional, ansi}
ext-charsets	{disable, complex, all}
innodb	{enable, disable}
libedit	{enable, disable}
log-bin	{enable, disable}
readline	{enable, disable}
ndbcluster	{enable, disable}
ssl	{enable, disable}
archive	{enable, disable}
blockhole	{enable, disable}
federated	{enable, disable}
test cluster list: $\{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}, t_{14}, t_{15}, t_{16}, t_{17}, t_{18}, t_{19}, t_{20}, t_{21}, t_{22}, t_{23}, t_{24}, t_{25}, t_{26}, t_{27}, t_{28}, t_{29}, t_{30}\}$	
system-wide constraint: ssl=disable $\wedge$ (libedit=enable $\rightarrow$ readline=disable)	
cluster idx	**test case-specific constraint**
t_1	log-bin=enable $\wedge$ sql-mode$\neq$ansi
t_2	ndbcluster=enable
t_3	innodb=enable
t_4	log-format$\neq$row $\wedge$ log-bin=enable $\wedge$ sql-mode$\neq$ansi
t_5	sql-mode$\neq$ansi
t_6	ext-charsets$\neq$disable $\wedge$ sql-mode$\neq$ansi
t_7	log-format$\neq$statement $\wedge$ log-bin=enable $\wedge$ ndbcluster=enable
t_8	innodb=enable $\wedge$ log-bin=enable $\wedge$ sql-mode$\neq$ansi
t_9	log-bin=enable $\wedge$ ndbcluster=enable
t_{10}	log-format$\neq$row $\wedge$ innodb=enable $\wedge$ log-bin=enable $\wedge$ sql-mode$\neq$ansi
t_{11}	log-format$\neq$row $\wedge$ ext-charsets$\neq$disable $\wedge$ log-bin=enable $\wedge$ sql-mode$\neq$ansi
t_{12}	federated=enable $\wedge$ log-bin=enable $\wedge$ sql-mode$\neq$ansi
t_{13}	innodb=enable $\wedge$ sql-mode$\neq$ansi
t_{14}	ndbcluster=enable $\wedge$ sql-mode$\neq$ansi
t_{15}	log-format$\neq$statement $\wedge$ innodb=enable $\wedge$ log-bin=enable $\wedge$ sql-mode$\neq$ansi
t_{16}	blackhole=enable $\wedge$ log-bin=enable $\wedge$ ndbcluster=enable
t_{17}	archive=enable $\wedge$ log-format$\neq$row $\wedge$ log-bin=enable $\wedge$ sql-mode$\neq$ansi
t_{18}	federated=enable $\wedge$ innodb=enable $\wedge$ log-bin=enable $\wedge$ sql-mode$\neq$ansi
t_{19}	log-format$\neq$row $\wedge$ blackhole=enable $\wedge$ log-bin=enable $\wedge$ sql-mode$\neq$ansi
t_{20}	log-format$\neq$statement $\wedge$ log-bin=enable $\wedge$ ndbcluster=enable $\wedge$ sql-mode$\neq$ansi
t_{21}	ext-charsets$\neq$disable $\wedge$ log-bin=enable $\wedge$ sql-mode$\neq$ansi
t_{22}	log-bin=enable $\wedge$ ndbcluster=enable $\wedge$ sql-mode$\neq$ansi
t_{23}	log-format$\neq$row $\wedge$ log-bin=enable $\wedge$ ndbcluster=enable
t_{24}	ext-charsets$\neq$disable $\wedge$ innodb=enable $\wedge$ sql-mode$\neq$ansi
t_{25}	innodb=enable $\wedge$ log-bin=enable $\wedge$ ndbcluster=enable
t_{26}	innodb=enable $\wedge$ ndbcluster=enable
t_{27}	archive=enable $\wedge$ innodb=enable
t_{28}	archive=enable
t_{29}	log-bin=enable
t_{30}	ext-charsets$\neq$all

Table 6.2: Initial configuration space model for MySQL.

6.2. Operation Model

We used the configuration space models given in Table 6.1 and 6.2. These models only contain configuration options that are referenced by the system-wide or test case-specific constraints. A configuration option that is referenced by a constraint (system-wide or test case-specific) is referred to as a *constrained option*, e.g., all the options in our initial configuration models were constrained options (100%). In order to vary the constrained option percentage (*cop* in short), we augmented the initial configuration space models by adding extra unconstrained binary options.

We then run the experiments for each combination of independent variables and strategies. In total, we have executed 2240 test case-aware covering array generation tasks.

$$\{2 \text{ SUTs}\} \times \{7 \text{ Models}\} \times \{4 \text{ IS}\} \times \{4 \text{ NS}\} \times \{2 \text{ t value}\} \times \{5 \text{ runs}\} = 2240$$

All the experiments were performed on a Casper computer with 31.3 GB of RAM, 8 Intel(R) Xeon(R) E630 @ 2.53GHz CPUs, and running CentOS 6.2 operating system on 64bit Kernel Linux 2.6.32 and GNOME 2.28.2.

6.3. Independent Variables

Strength of the test case-aware covering array, t, is an independent variable that we used to evaluate the performance of the approach. Test case-aware covering array computation time grows exponentially with the strength. The results of many empirical studies strongly suggest that a majority of option-related failures in practice are caused by the interactions among only a small number of configuration options [23]. Therefore in the experiments, we used $t = 2$ and $t = 3$ to see the behavior of our approach for varying strengths.

Subject application (SUT) is an independent variable to evaluate the behavior of the approach. We used two different subject applications; Apache, MySQL which are described in Section 6.1. Compared to Apache, MySQL has more constrained configuration space model (see Table 6.1 and 6.2) which will challenge the task.

Configuration space model of SUT is the last independent variable to evaluate the behavior of the approach. Initial configuration space models of the SUTs have 100% constrained options (e.g. each of the configuration options is referred at least one constraint). To vary the percentage of constraint options (constraint option percentage, *cop* in short), we augmented the initial configuration space models by adding extra unconstrained binary options. In particular, we used *cop*=20, 30, 40, 50, 60, 80, and 100 (e.g. 7 configuration space models for each SUT).

In addition, SA algorithm has three control parameters; T_0, C_r, and T_s. By conducting a small-scale experiment, values of those parameters determined as follows: $T_0 = 1$, $C_r = 0,1 \times 10^{-2}$, and $T_s = 0,1 \times 10^{-3}$

6.4. Evaluation Framework

In order to evaluate the proposed approach, we have investigated 1) the effect of coverage strength, t, 2) the consequence of *cop*, 3) the impact of subject application spectra, 4) the performance of the initialization strategies, 5) the performance of the neighboring strategies, and 6) the overall performance of the approach compared to existing algorithms (Algorithm 1 and 2 introduced in [32]).

For evaluation, we used the dependent variables that are described in the following section.

6.4.1. Dependent Variables

As evaluation metrics, we used the following measures:

Initialization time is the time to generate the initial set. The smaller initialization time is the better.

$$\text{Initialization time} = \text{initialization end time} - \text{initialization start time}$$

Search (annealing) time is the time to generate a test case-aware covering array for a given initial set. The smaller search time is the better.

$$\text{Search time} = \text{search end time} - \text{search start time}$$

Total time is the time to generate a test case-aware covering array. The smaller total time is the better.

$$\text{Total time} = \text{initialization time} + \text{search time}$$

Size of array is the number of the configurations in the test case-aware covering array, Ψ. The smaller the array size is the better.

$$\text{Size of array} = |\{< c_1, \mathrm{T}_1 >, \ldots, < c_N, \mathrm{T}_N >\}|$$

where $c_i \in C$ and $\mathrm{T}_i \subseteq \mathrm{T}$ for $i = 1, 2, \ldots, N$.

Initial miss count is number of the t-pairs that are not in the initial set. The smaller the initial miss count is the better.

$$\text{Initial miss count} = |\{\lambda_t =< \phi_t, \tau >: \tau \in \mathrm{T} \wedge \phi_t \in \Phi_t^{\tau} \wedge \neg\exists \ < c_i, \mathrm{T}_i > \in \ S_0 : \phi_t \subseteq c_i \wedge \tau \in \mathrm{T}_i\}|.$$

Initial miss percentage is the percentage of missing t-pairs in the initial set. The smaller the initial miss percentage is the better.

$$\text{Initial miss percentage} = \frac{\text{initial miss count}}{\text{number of valid t-pairs}} \times 100$$

IS ineffectiveness is a measure for the effectiveness of the initialization strategy. The smaller the IS ineffectiveness is the better.

$$\text{IS ineffectiveness} = \frac{\text{initialization time}}{\text{initial miss percentage}}$$

6.5. Data and Analysis

The results of the conducted experiments are set of 2-way and 3-way test case-aware covering arrays that are generated for each of the configuration space models of the subject applications using each combination of initialization and neighboring strategies. All row data from the experiments can be found in Appendix A.

In our analysis, we first compared the initialization strategies (Section 6.5.1). For this analysis, the data is grouped by initialization strategy and using box plots we depicted initialization time, initial miss percentages, and initial miss counts of initialization strategies. We then compared the neighboring strategies. Using box plots, we depicted search (annealing) time and test case-aware covering array sizes for the neighboring strategies (Section 6.5.2). Finally, we grouped the data by initialization and neighboring strategy combinations and using normal plots we compared total construction time and test case-aware covering array sizes (Section 6.5.3).

Box plots: Box plots depict groups of numerical data through their quartiles. The lower horizontal bar represents the first quartile, middle horizontal bar represents the median value (second quartile), and the upper horizontal bar represents the third quartile. Thus, 50% of the data falls into the box. Height of box shows the variance; the higher the box the higher the variance. Red small circles show the mean value for that data group. Black points that are outside of the boxes are outliers.

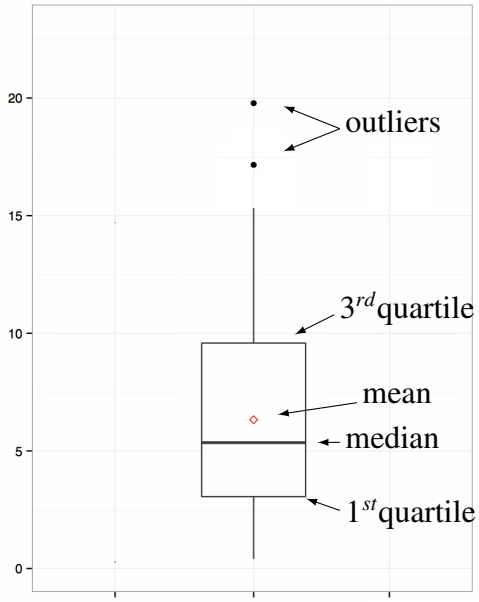

Figure 6.1: Sample box plot

6.5.1. Study 1: Comparing Initial Set Generation Strategies

In this study, we evaluated the initialization strategies described in Section 5.4. The desired case is to have the initial set that has minimum missing count with minimum initialization time.

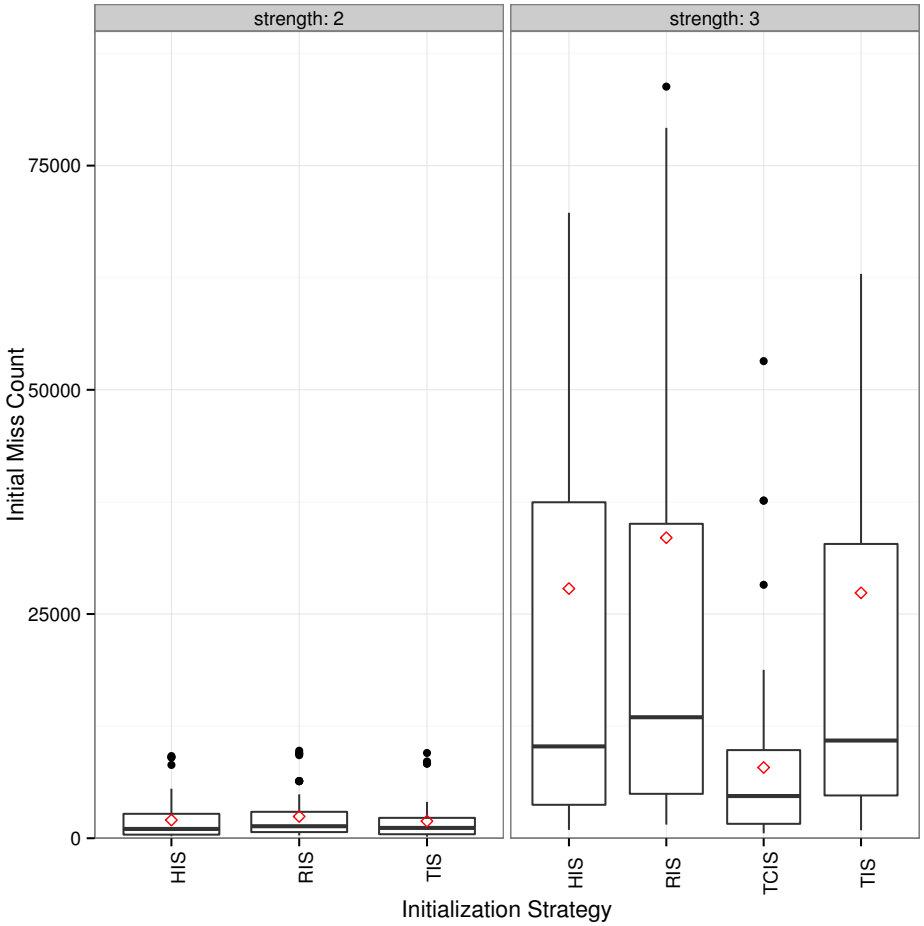

Figure 6.2: Comparing initial missing t-pair counts for initialization strategies at strength level

Figure 6.2 illustrates the missing pair counts of the initialization strategies for each strength. For $t = 3$, TCIS strategy is the best in the initial miss count, but it is not applicable for $t = 2$. For $t = 2$, performances of HIS and TIS strategies are closed to each other. RIS on the other hand, is always the worst in the initial miss count.

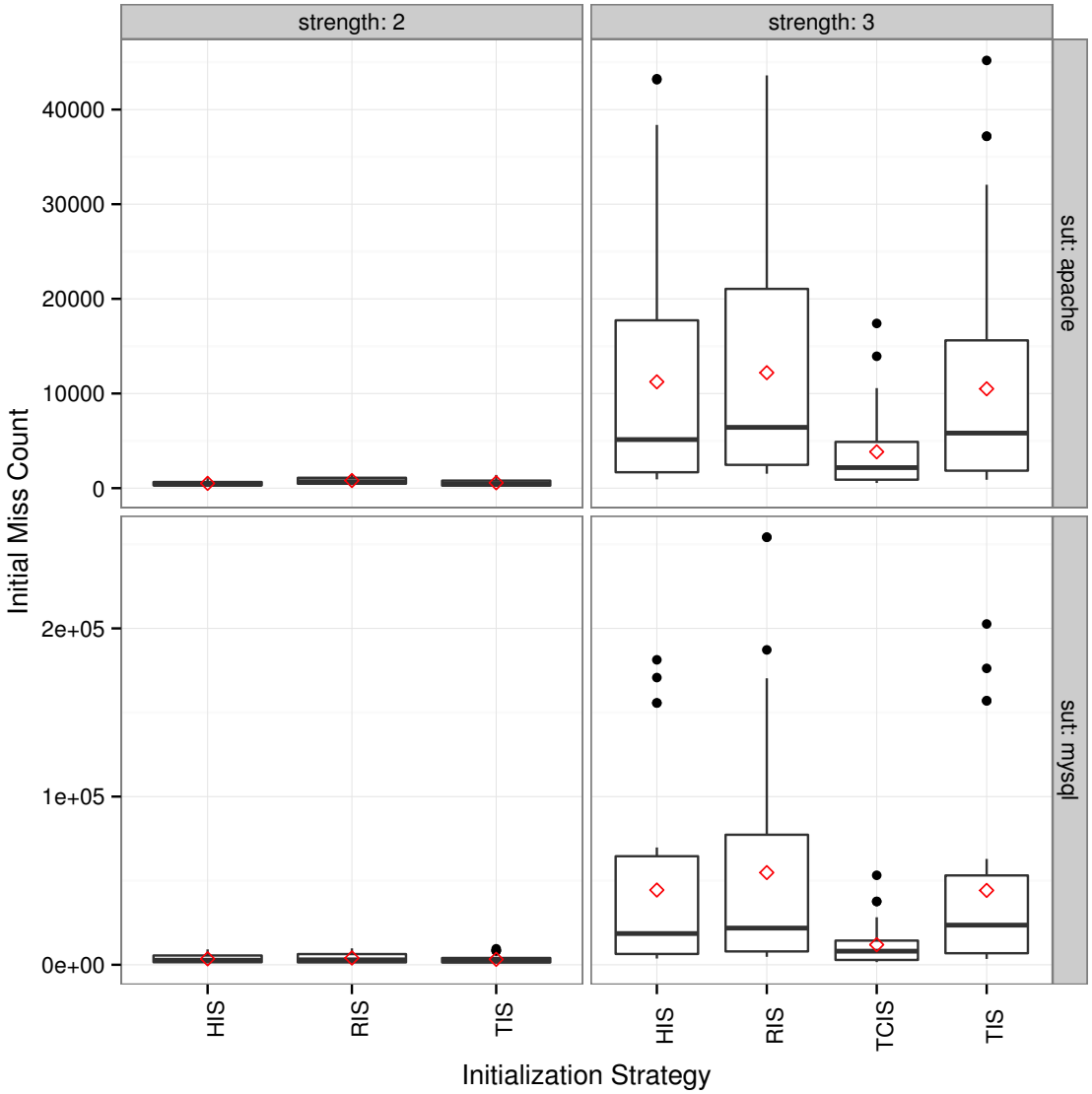

Figure 6.3: Comparing initial missing t-pair counts for initialization strategies at SUT by strength level

Figure 6.3 illustrates the missing pair counts for each of the initialization strategies for each SUT and strength. First of all, the graph has similar patters for each SUT. Which means in the experiments performance of our initialization strategies did not depend on the subject applications. For $t = 3$, TCIS strategy is the best in the initial miss count, but it is not applicable for $t = 2$. For $t = 2$, performances of HIS and TIS strategies are closed to each other. RIS on the other hand, is always the worst in the initial miss count.

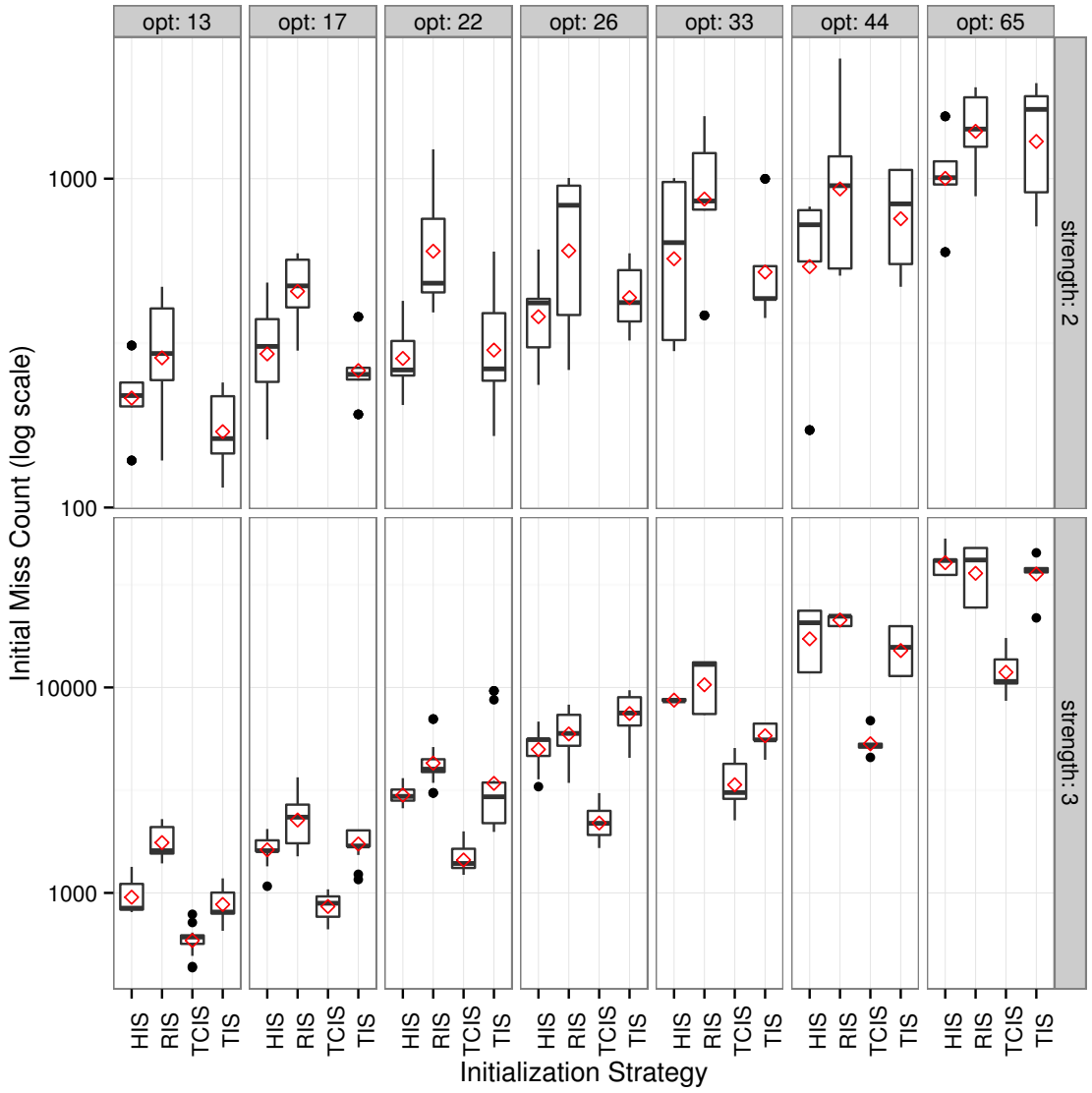

Figure 6.4: Comparing initial missing t-pair counts for initialization strategies detailed for Apache configuration space models

Figure 6.4 illustrates the missing pair counts of the initialization strategies for each strength and configuration space model of Apache. For $t = 3$, TCIS strategy is the best in the initial miss count, but it is not applicable for $t = 2$. For $t = 2$, performances of HIS and TIS strategies are closed to each other. RIS on the other hand, is always the worst in the initial miss count. Lastly, on overall the initial miss count is increasing when the number of configuration options grow.

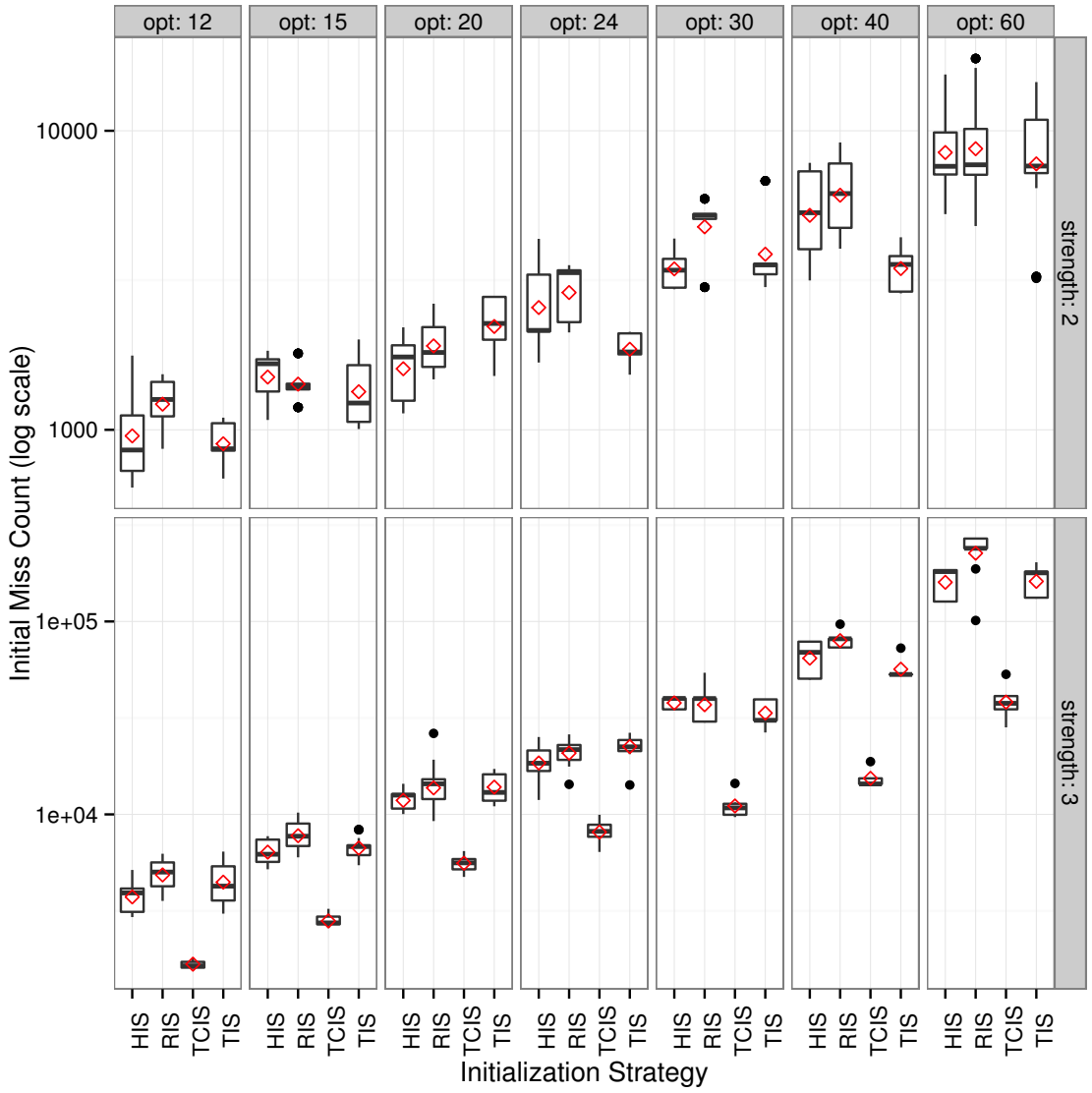

Figure 6.5: Comparing initial missing t-pair counts for initialization strategies detailed for MySQL Configuration space models

Figure 6.5 illustrates the missing pair counts of the initialization strategies for each strength and configuration space model of MySQL. For $t = 3$, TCIS strategy is the best in the initial miss count, but not applicable when $t = 2$. For $t = 2$, performances of HIS and TIS strategies are closed to each other. RIS on the other hand, is always the worst in the initial miss count. Lastly, on overall the initial miss count is increasing when the number of configuration options grow.

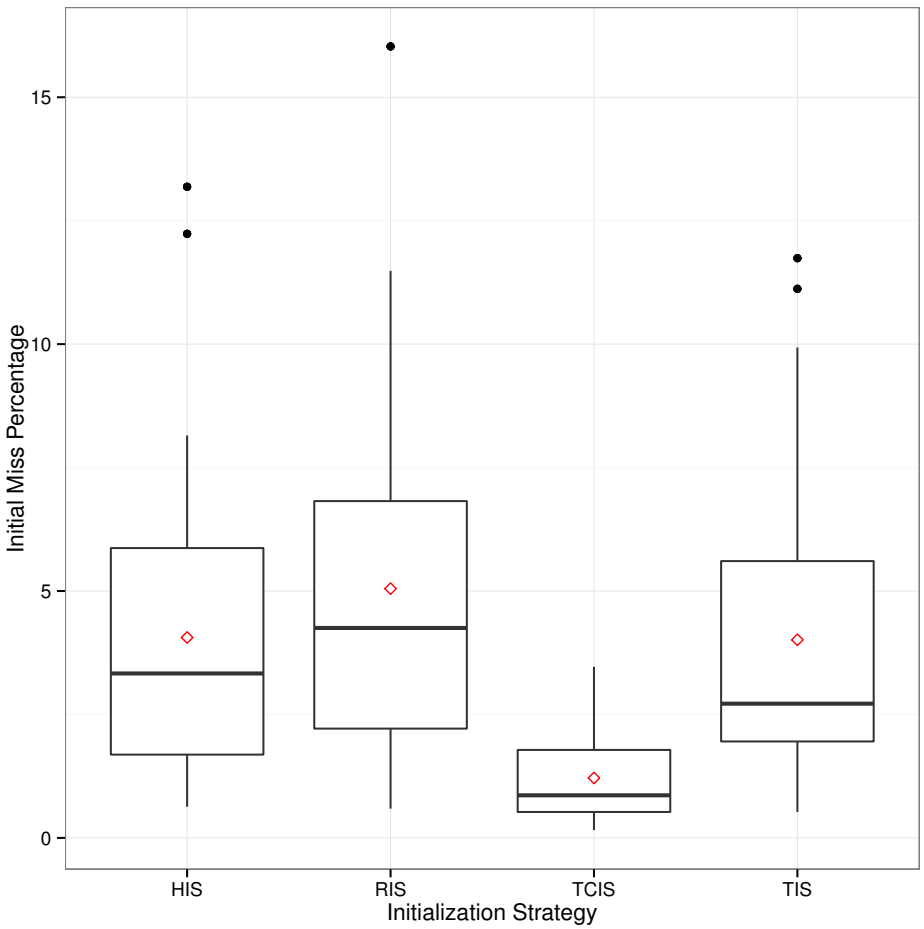

Figure 6.6: Comparing initial missing t-pair percentages for initialization strategies overall

Figure 6.6 illustrates the overall miss percentage of the initialization strategies. TCIS strategy is the best in the overall miss percentage and performances of HIS and TIS strategies are closed to each other. RIS on the other hand, is the worst in the overall miss percentage.

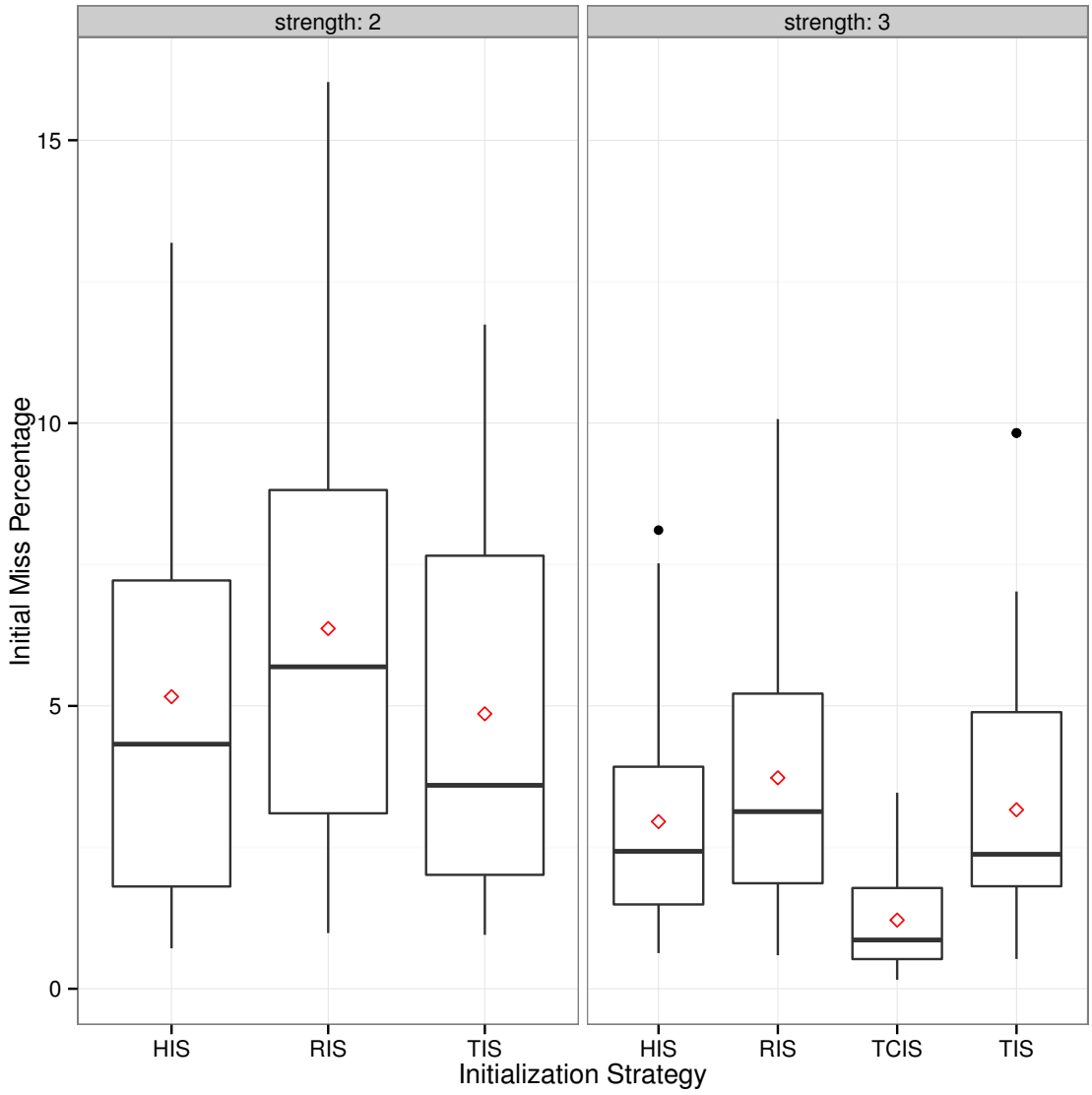

Figure 6.7: Comparing initial missing t-pair percentages for initialization strategies at strength level

Figure 6.7 illustrates the miss percentage of the initialization strategies for each strength. TCIS strategy is the best in the miss percentage for $t = 3$ but not applicable for $t = 2$. For $t = 2$, performances of HIS and TIS strategies are closed to each other. RIS on the other hand, is the worst in the miss percentage.

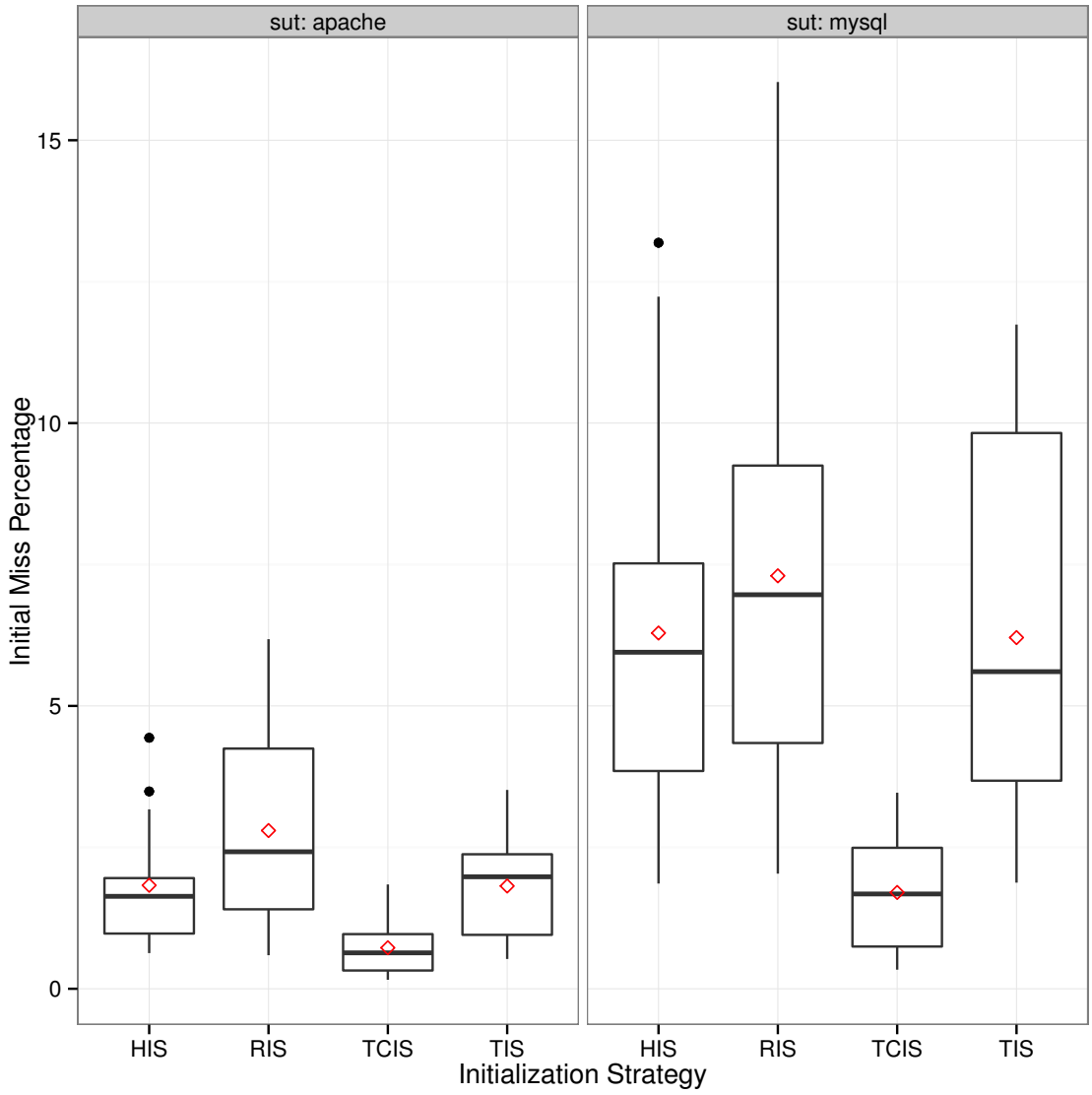

Figure 6.8: Comparing initial missing t-pair percentages for initialization strategies at SUT level

Figure 6.8 illustrates the miss percentage of the initialization strategies for each SUT. In addition to the Figure 6.6, it is necessary to say that the graph has similar patterns for each of the subject applications, indicating that; in the experiments, performance of our initialization strategies did not depend on the subject applications.

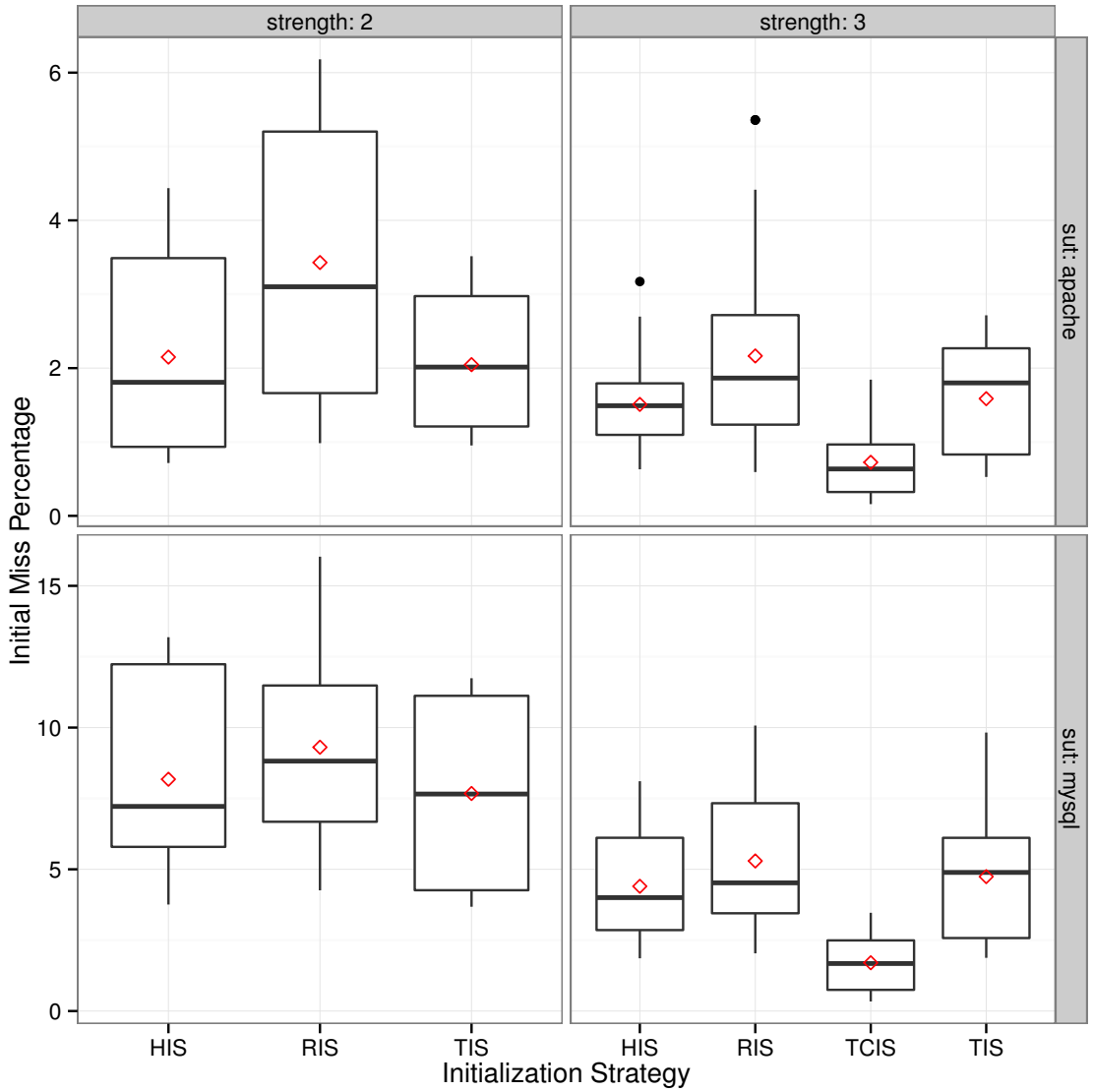

Figure 6.9: Comparing initial missing t-pair percentages for initialization strategies at SUT by strength level

Figure 6.9 illustrates the miss percentage of the initialization strategies for each strength and SUT. In addition to the Figure 6.7, it is necessary to say that the graph has similar patterns for each of the the subject applications, indicating that; in the experiments, performance of our initialization strategies did not depend on the subject applications for different strength levels.

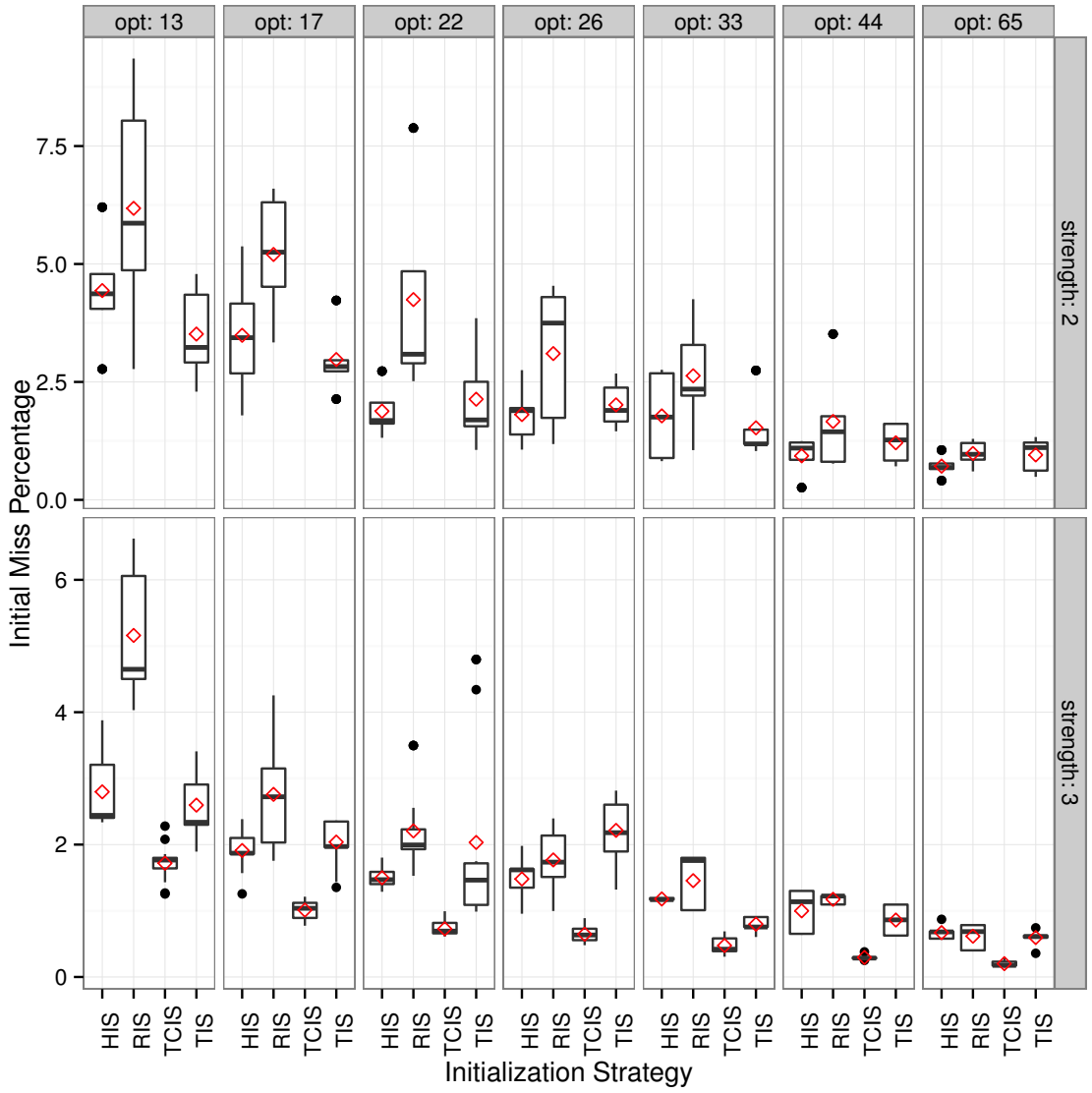

Figure 6.10: Comparing initial missing t-pair percentages for initialization strategies detailed for Apache configuration space models

Figure 6.10 illustrates the miss percentage of the initialization strategies for each strength and configuration space model of Apache. First of all, as the number of configuration options increase the initial miss percentage is decreasing for all strategies. For $t = 3$, TCIS strategy is the best in the miss percentage but it is not applicable for $t = 2$. For $t = 2$, performances of HIS and TIS strategies are closed to each other. RIS on the other hand, is always the worst in the initial miss percentage.

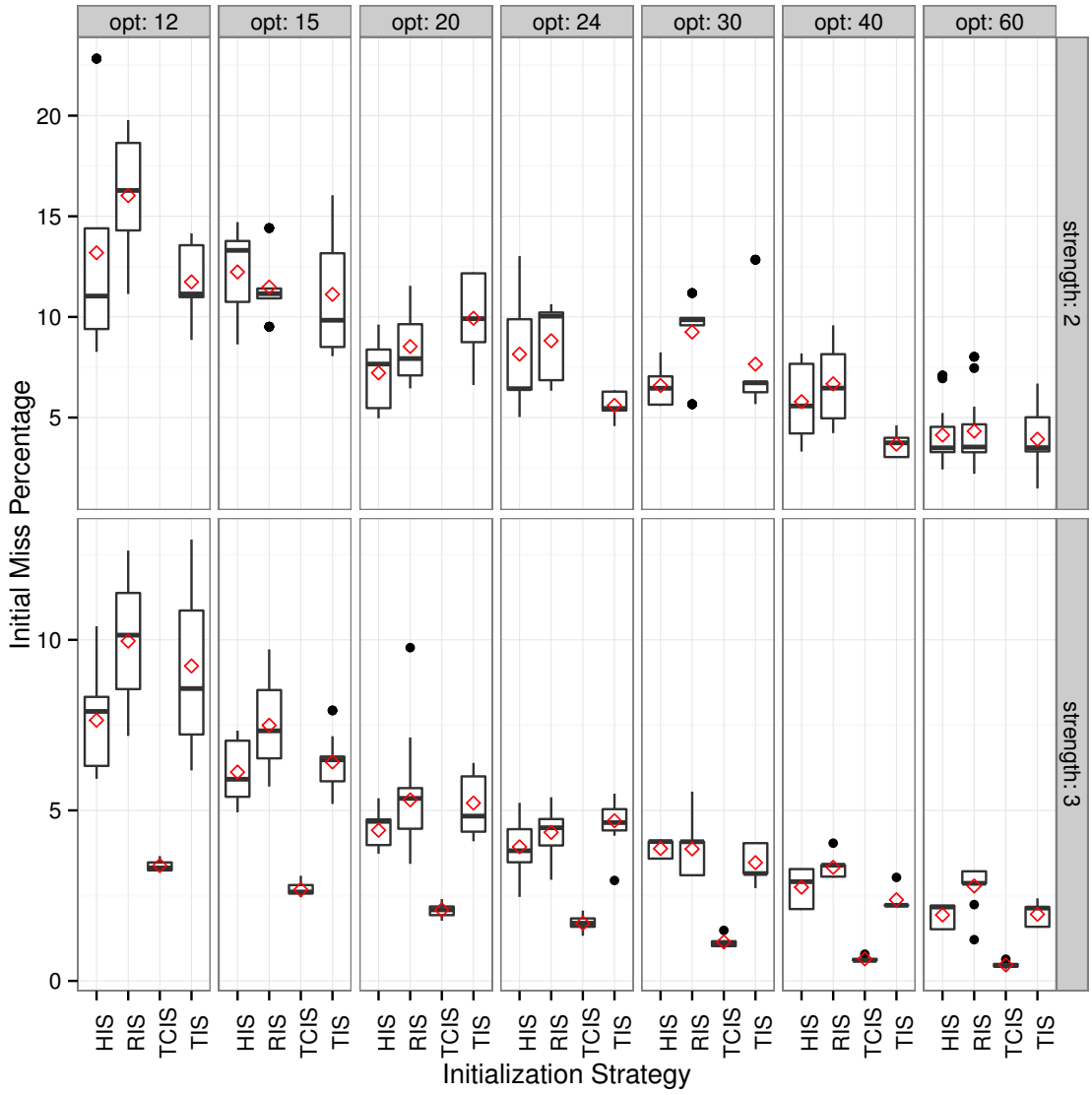

Figure 6.11: Comparing initial missing t-pair percentages for initialization strategies detailed for MySQL Configuration space models

Figure 6.11 illustrates the miss percentage of the initialization strategies for each strength and configuration space model of MySQL. In this graph has a similar pattern with Figure 6.10. For $t = 3$, TCIS strategy is the best in the miss percentage but it is not applicable for $t = 2$. For $t = 2$, performances of HIS and TIS strategies are closed to each other. RIS on the other hand, is always the worst in the initial miss percentage.

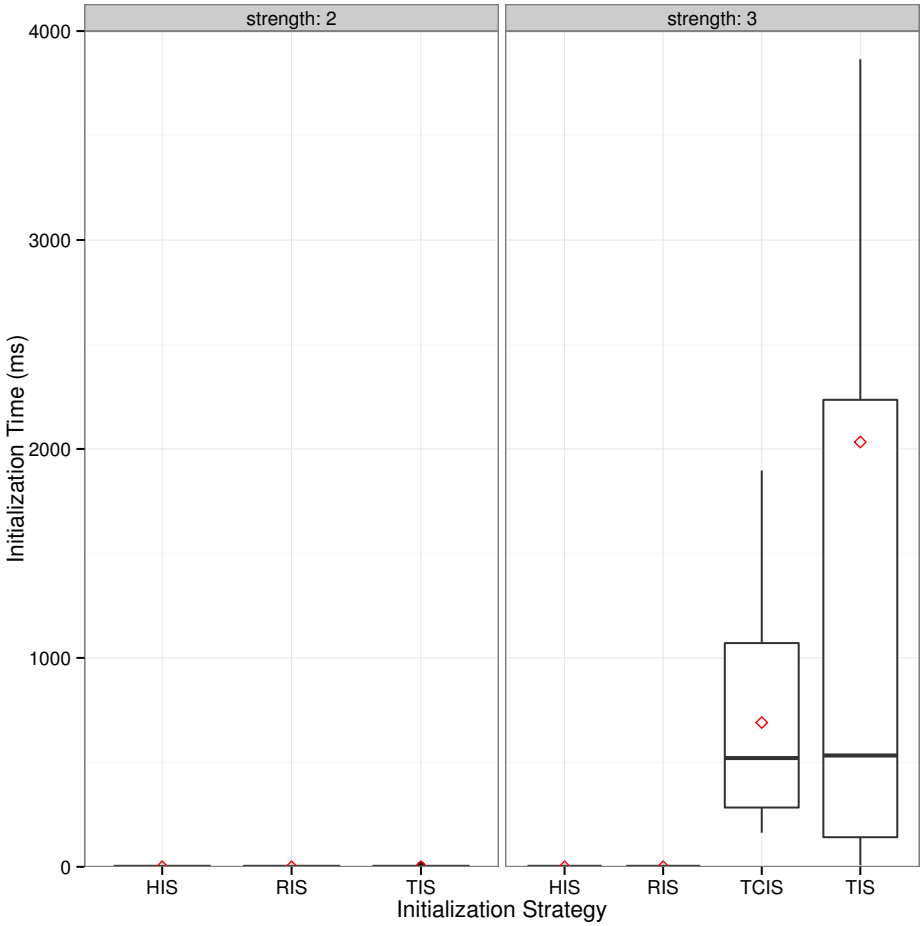

Figure 6.12: Comparing initialization times for initialization strategies at strength level

Figure 6.12 illustrates the initialization time of the initialization strategies for each strength. HIS and RIS strategies have negligible initialization time (always measured as 0 in the experiments). TIS strategy on the other hand, is always the most time consuming one for $t = 3$ as well as for $t = 2$ (although it is very close to 0). TCIS is not applicable for $t = 2$ and it required longer time compared to HIS and RIS.

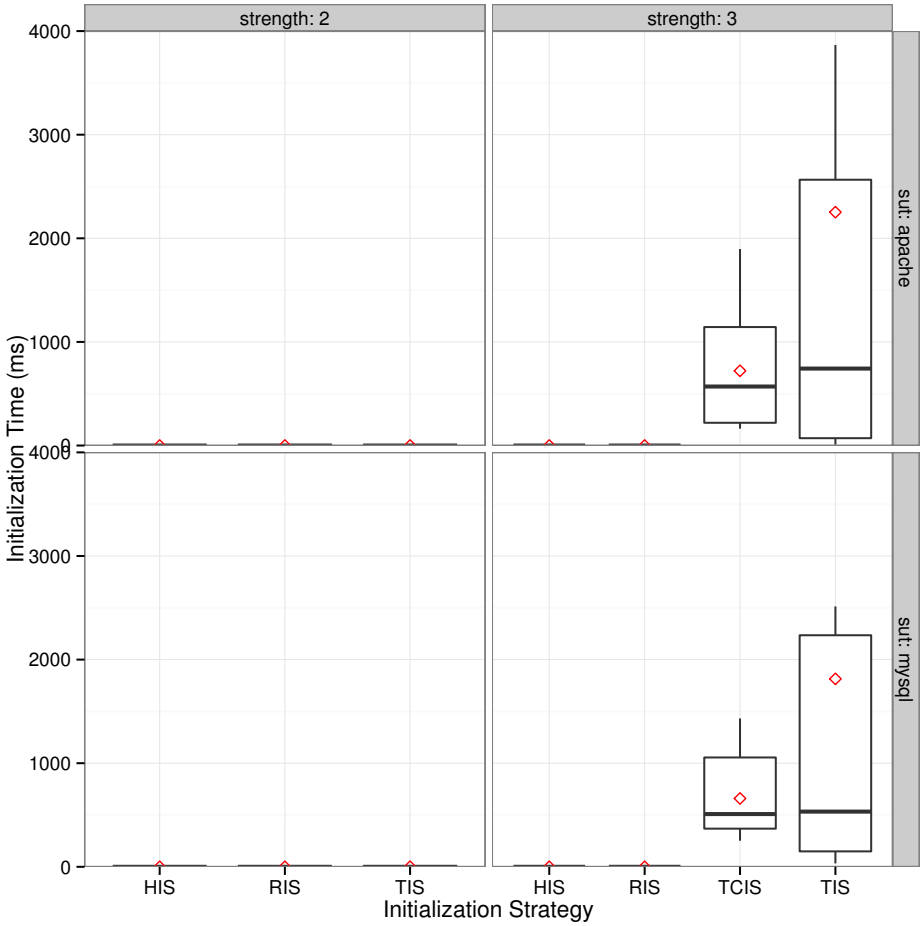

Figure 6.13: Comparing initialization times for initialization strategies at SUT by strength level

Figure 6.13 illustrates the initialization time of the initialization strategies for each SUT and strength. In addition to the Figure 6.12, it is necessary to say that the graph has similar patterns for each of the subject applications, indicating that; in the experiments, initialization times of the initialization strategies did not depend on the subject applications for different strength levels.

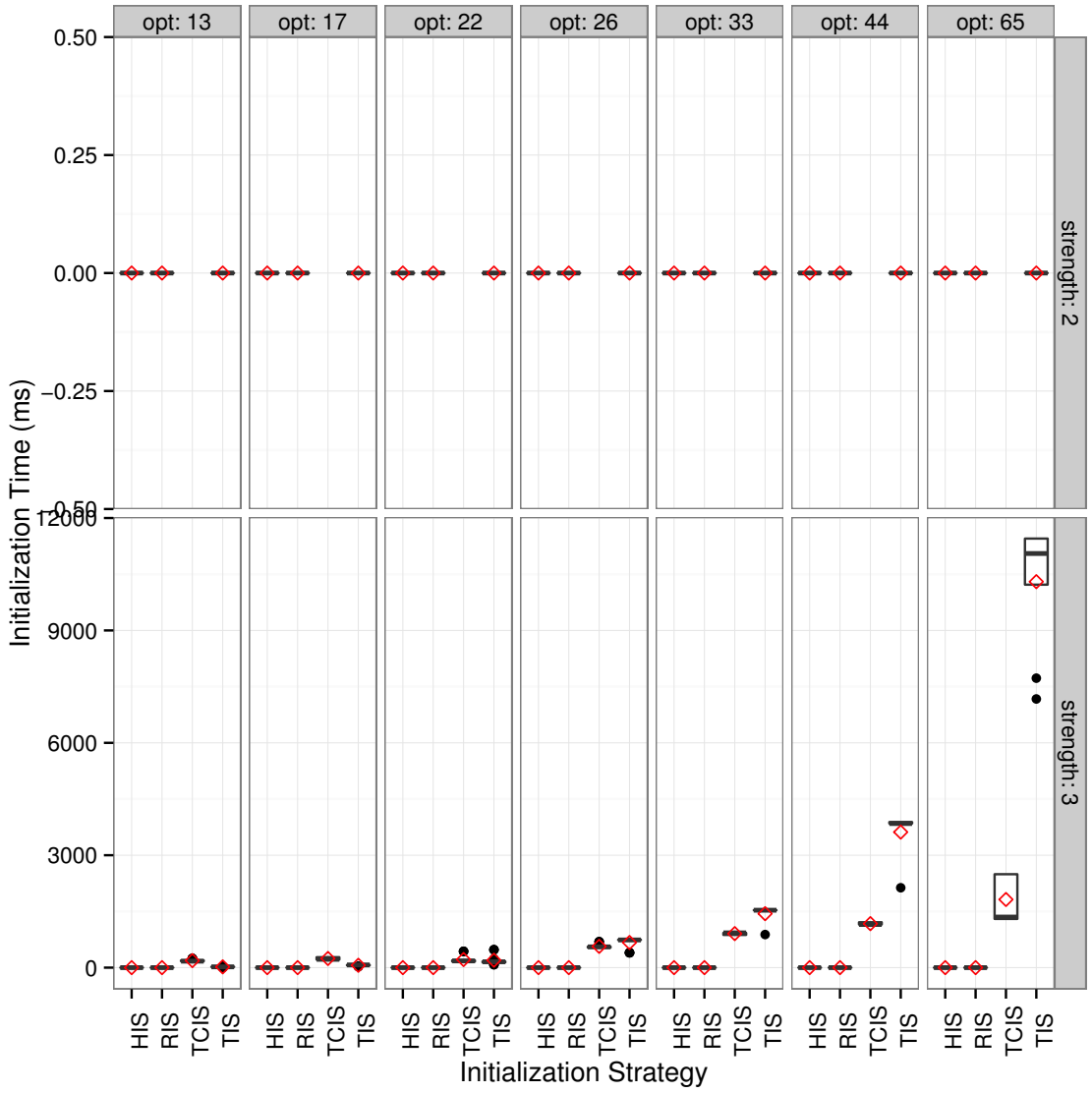

Figure 6.14: Comparing initialization times for initialization strategies detailed for Apache configuration space models

Figure 6.14 illustrates the initialization time of the initialization strategies for each strength and configuration space model of Apache. HIS and RIS strategies have negligible initialization time for every case. TCIS again is not applicable for $t = 3$ and it is not the worst for $t = 3$. TIS on the other hand, has negligible initialization time for $t = 2$, but for $t = 3$, it becomes the most time consuming strategy as the number of the configuration options grow.

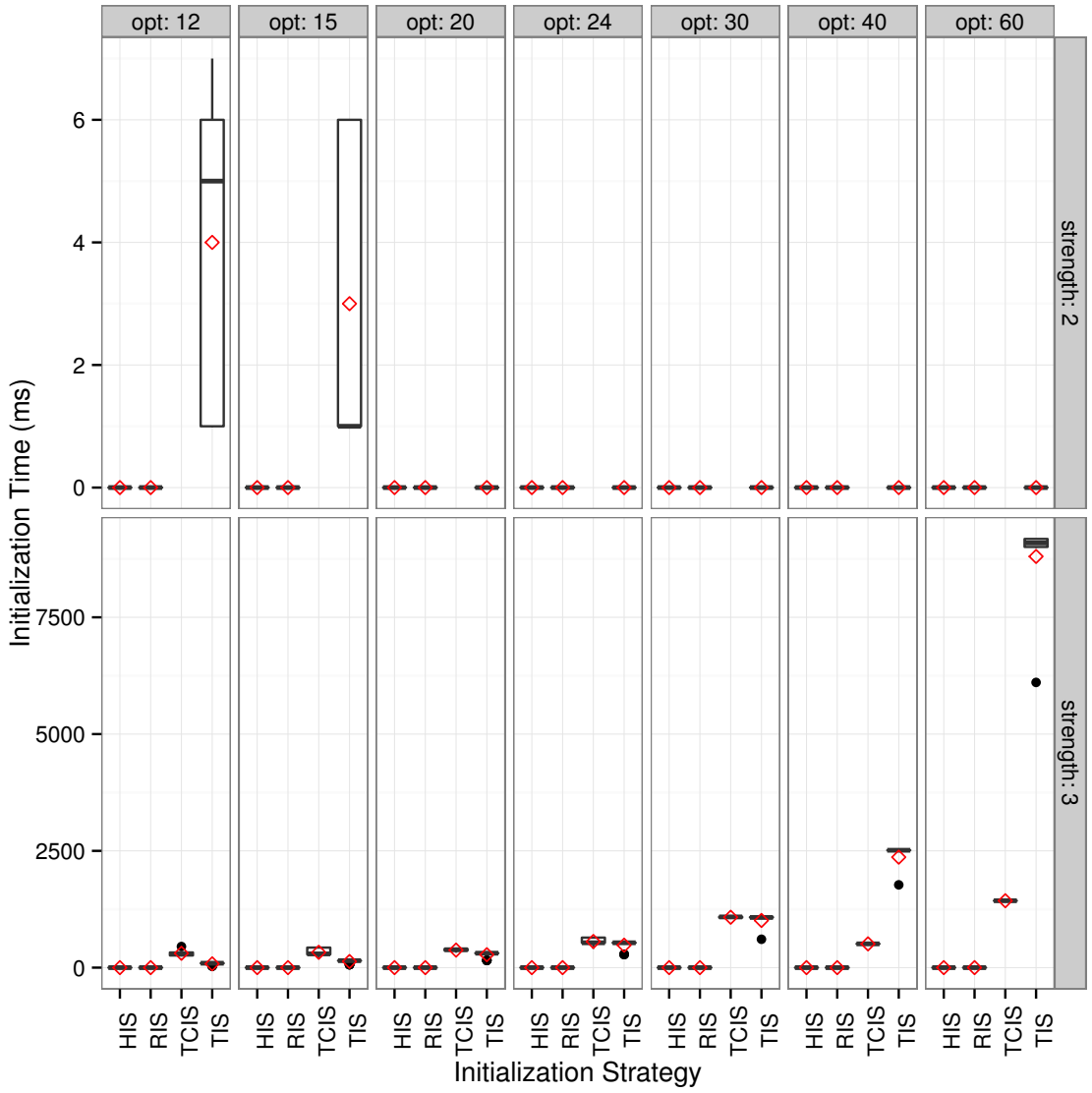

Figure 6.15: Comparing initialization times for initialization strategies detailed for MySQL Configuration space models

Figure 6.15 illustrates the initialization time of the initialization strategies for each strength and configuration space model of MySQL. HIS and RIS strategies have negligible initialization time for every case. TCIS again is not applicable for $t = 3$ and it is not the worst for $t = 3$. TIS on the other hand, has negligible initialization time for $t = 2$, but for $t = 3$, it becomes the most time consuming strategy as the number of the configuration options grow.

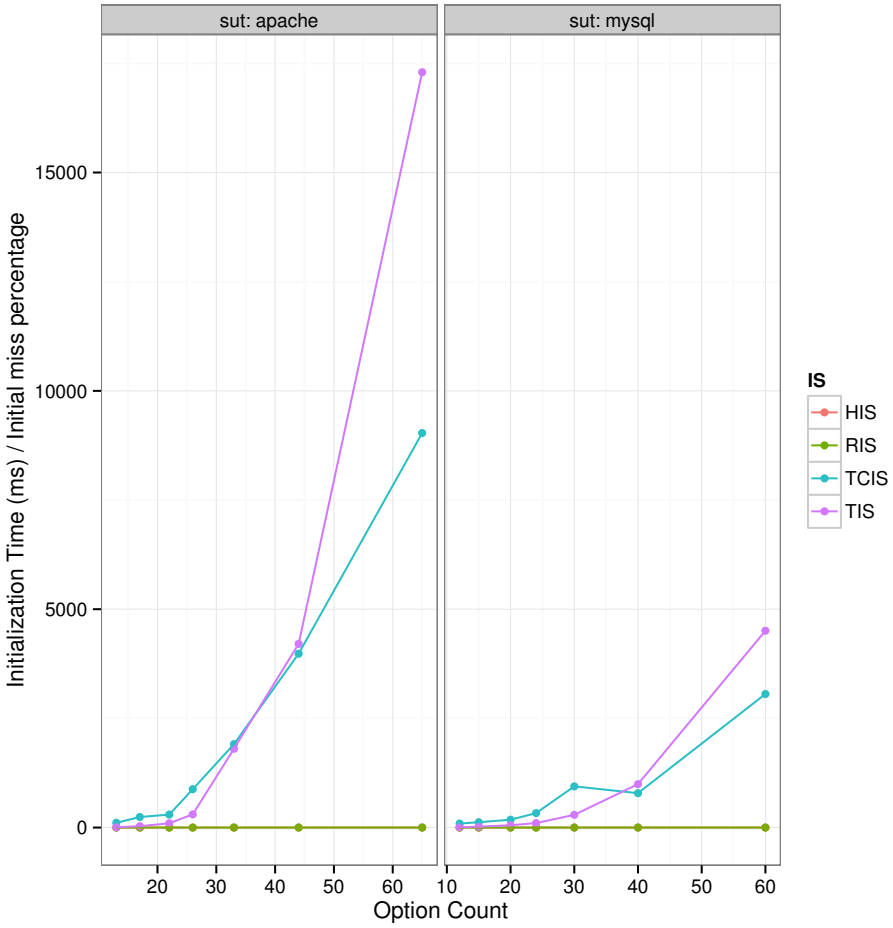

Figure 6.16: Comparing the ineffectiveness of initialization strategies

Figure 6.16 illustrates the IS ineffectiveness of initialization strategies for each configuration space model of subject applications. Since the initialization time of RIS and HIS strategies are negligible (always measured as 0 in the experiments), their IS ineffectiveness scores are the minimum (the best) and TIS strategies has performed the worst on overall and it is getting worse as the configuration space grows.

This study has shown that TCIS strategy is the best in the miss count and percentage, but compared to HIS and RIS strategies, it required longer time to compute test case-aware covering arrays. Although HIS and TIS strategies have similar miss counts and percentages, HIS strategy is better, since it is faster.

RIS strategy on the other hand, which is the most commonly used one for covering array generation [10, 12, 23, 26], is fast, because it does not apply any intelligence. However, it is the worst in the initial miss count and percentage.

TIS strategy is not the best or worst for any case. However, this strategy is important to account already in use testing objects. We have designed our approach to be capable of using a traditional covering array as an initial set, so that developers can seed their available covering arrays into our tool to generate test case-aware covering arrays. By this way, their important configurations and testing objects will not be wasted.

6.5.2. Study 2: Comparing Neighbor Generation Strategies

In this study, we evaluated the neighboring strategies described in Section 5.5 (except for the SMT strategy which has failed to find Ψ most of the time). The desired case is to have the test case-aware covering array of minimum size with minimum computation time.

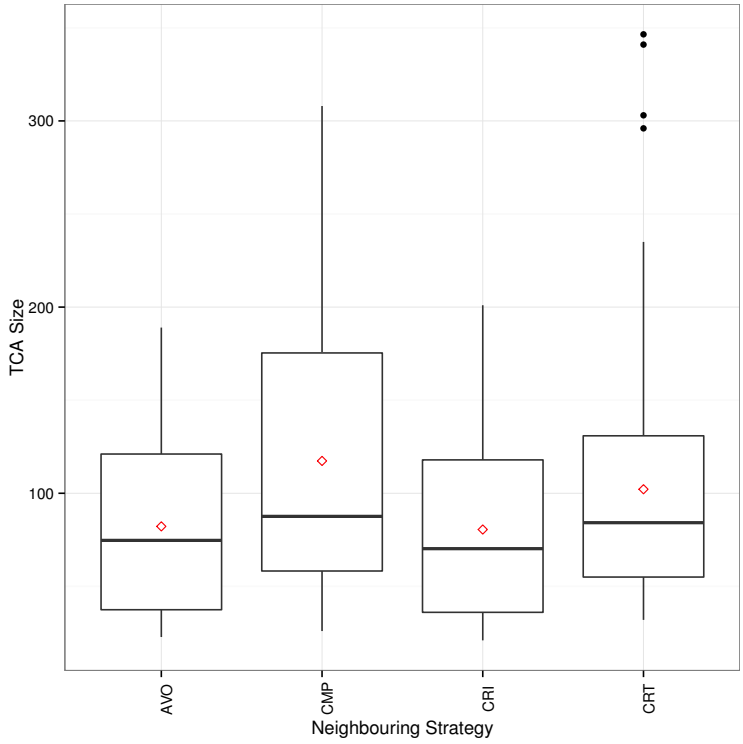

Figure 6.17: Comparing TCA sizes for neighboring strategies overall

Figure 6.17 illustrates the overall test case-aware covering array sizes for the neighboring strategies. Performances of AVO and CRI strategies are close to each other, and CMP strategy is the worst among others. Lastly, the height of boxes depict the performance of the strategy as the configuration space model grows. CMP is the most effected one from the number of configuration options.

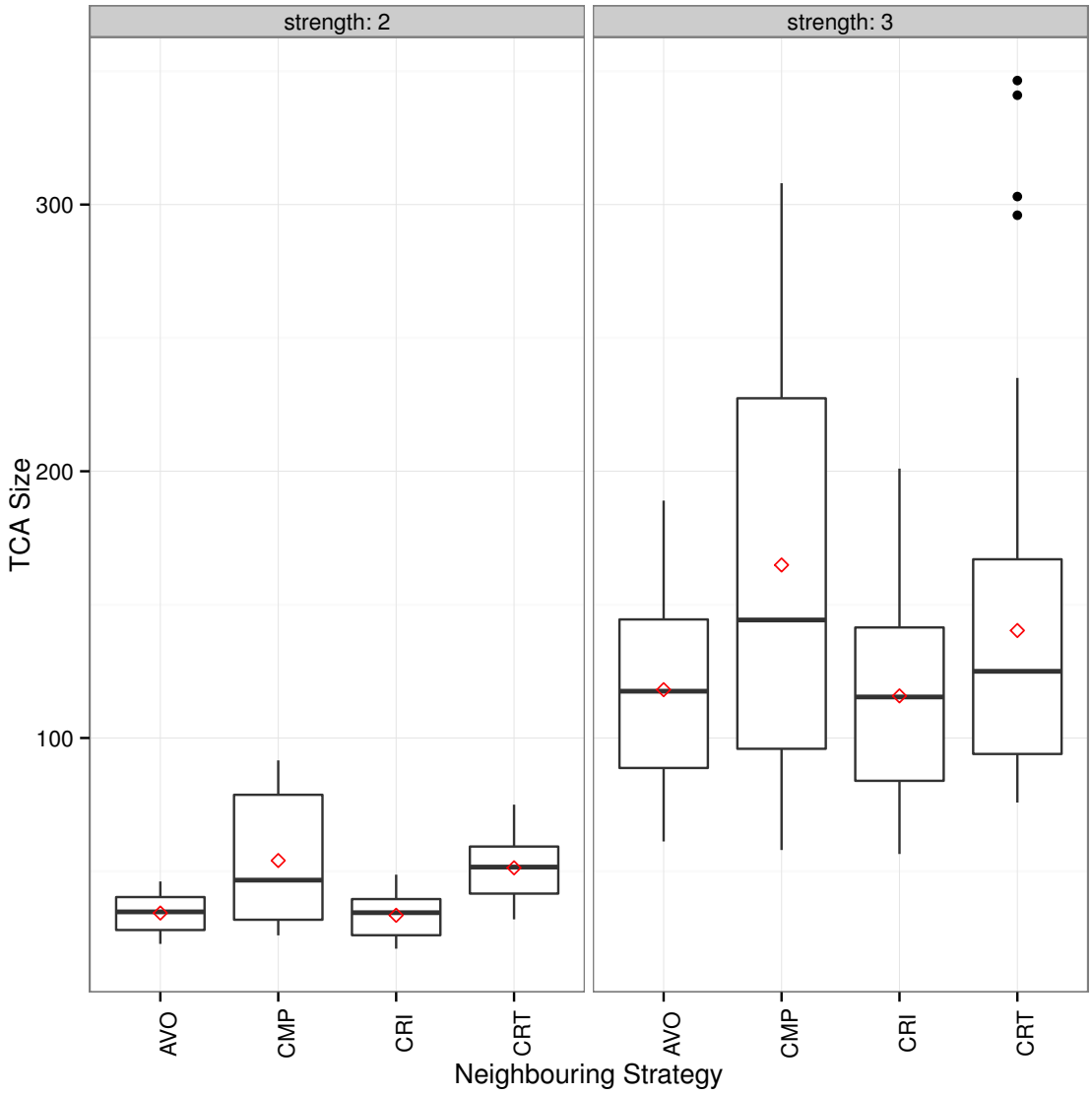

Figure 6.18: Comparing TCA sizes for neighboring strategies at strength level

In addition to Figure 6.17, Figure 6.18 illustrates sizes of the test case-aware covering arrays computed by the neighboring strategies for each strength. Performance of AVO and CRI strategies are close to each other, and CMP strategy is the worst among others.

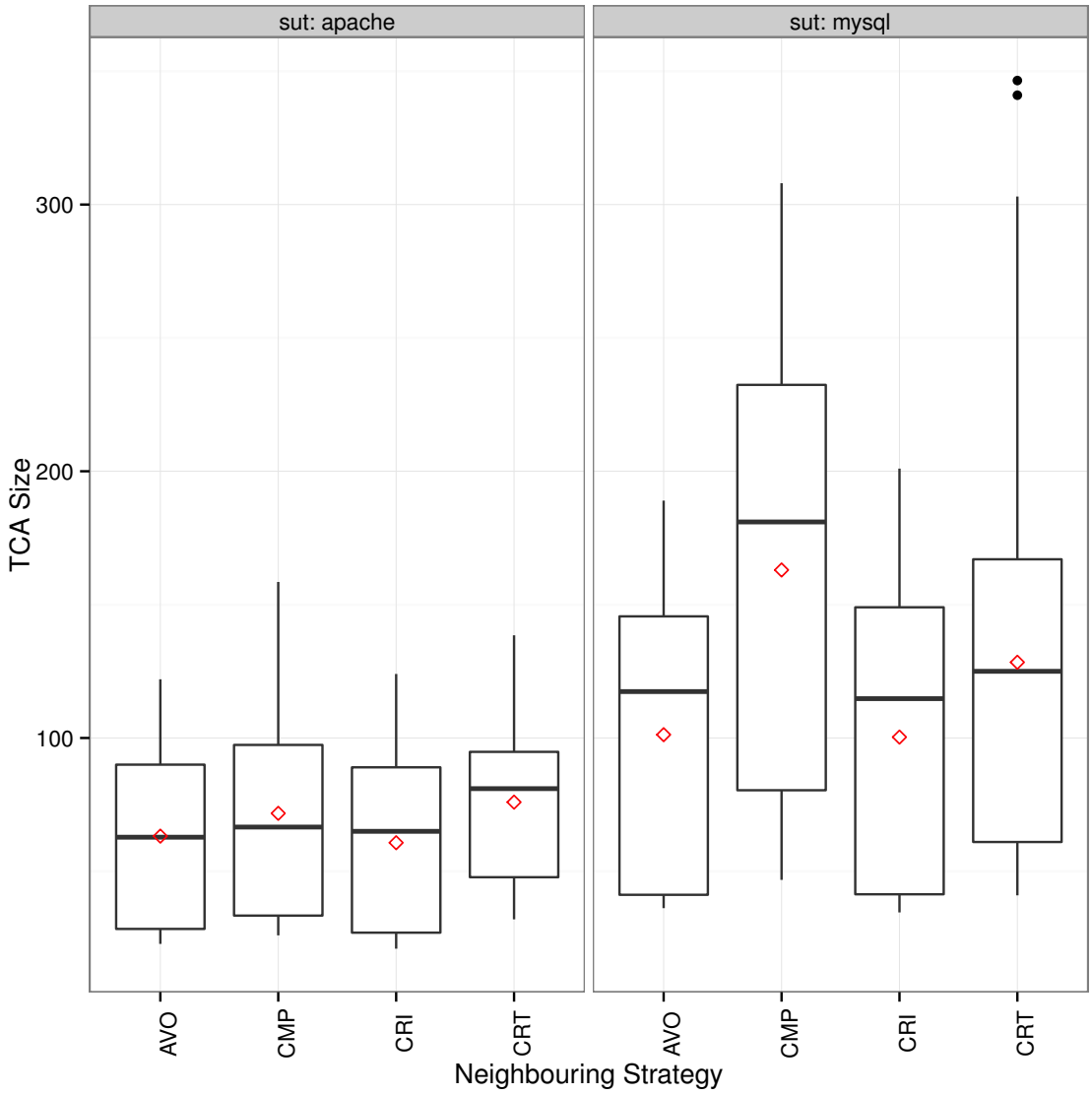

Figure 6.19: Comparing TCA sizes for neighboring strategies at SUT level

In addition to Figure 6.17, Figure 6.19 illustrates sizes of the test case-aware covering arrays computed by the neighboring strategies for each SUT. CRI strategy again is slightly better than AVO for Apache and they are similar for MySQL. CMP strategy on the other hand is the worst among others.

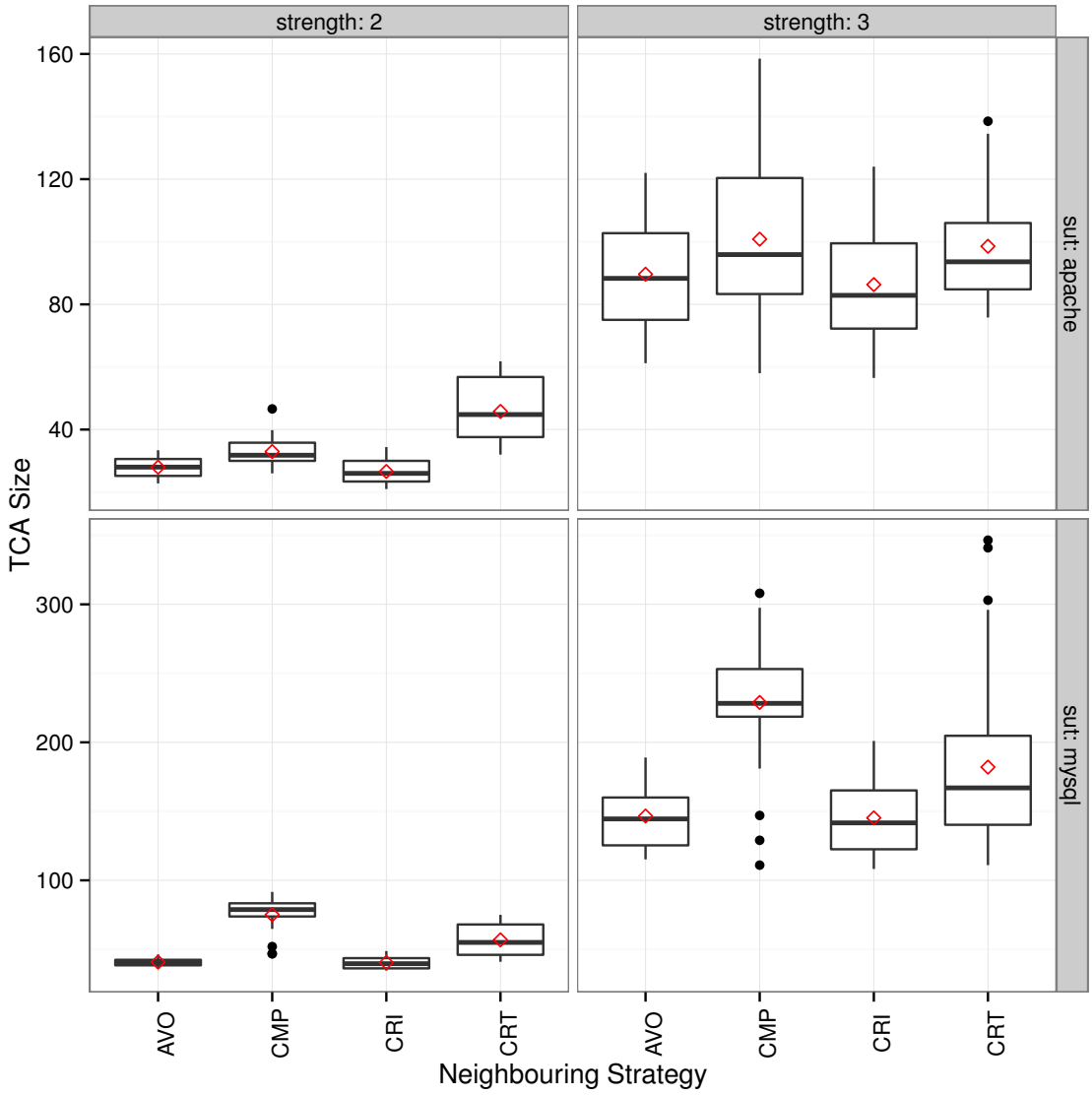

Figure 6.20: Comparing TCA sizes for neighboring strategies at SUT by strength level

Figure 6.20 illustrates sizes of the test case-aware covering arrays computed by the neighboring strategies for each SUT and strength. AVO and CRI strategies have similar performances except for Apache, $t = 3$. For Apache, $t = 2$ CRT strategy is the worst, and CMP strategy is the worst in the other cases.

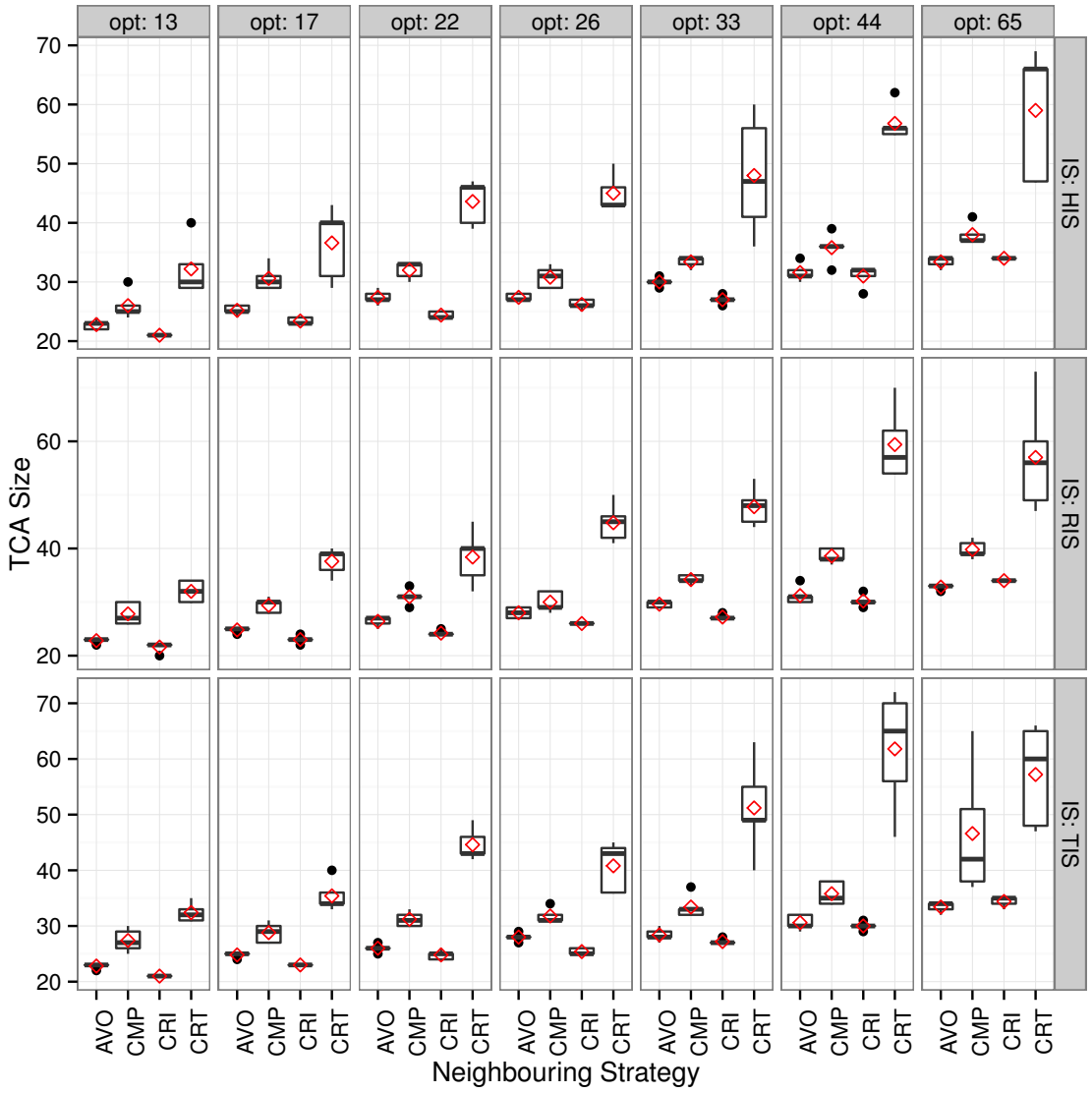

Figure 6.21: Comparing TCA sizes for neighboring strategies detailed for Apache configuration space models and $t = 2$

Figure 6.21 illustrates sizes of the test case-aware covering arrays computed by the neighboring strategies for each initialization strategy and configuration space model of Apache, when $t = 2$. As the number of configuration options increase CRT losses performance, and CRI losses it's advantage to AVO. Lastly, the graph has similar patterns for each of the initialization strategies.

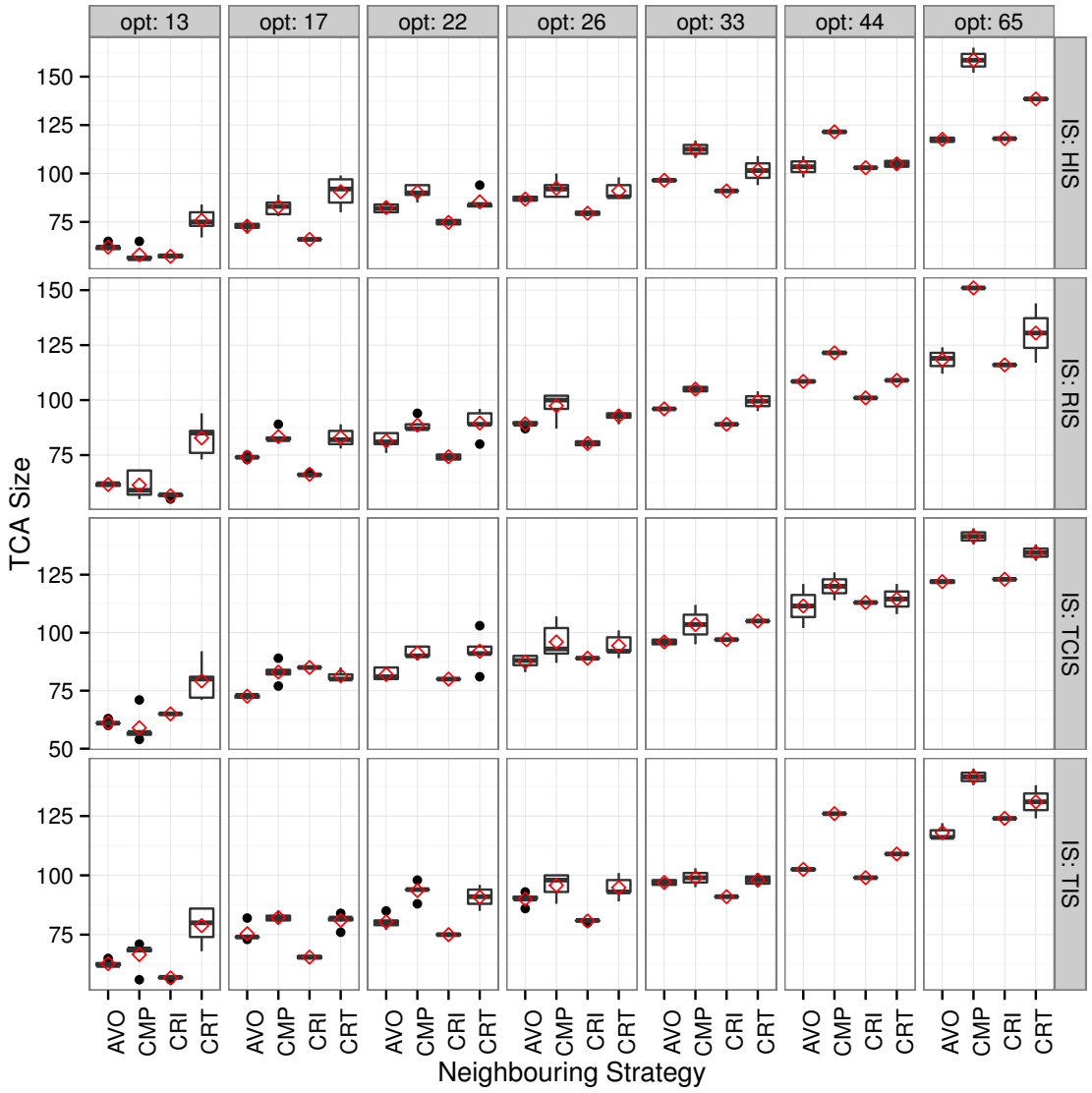

Figure 6.22: Comparing TCA sizes for neighboring strategies detailed for Apache configuration space models and $t = 3$

Figure 6.22 illustrates sizes of the test case-aware covering arrays computed by the neighboring strategies for each initialization strategy and configuration space model of Apache, when $t = 3$. The observation from this graph is similar with Figure 6.21, indicating that; strength did not have a significant effect on the performance of neighboring strategies. As the number of configuration options increase CRT losses performance, and CRI losses it's advantages to AVO.

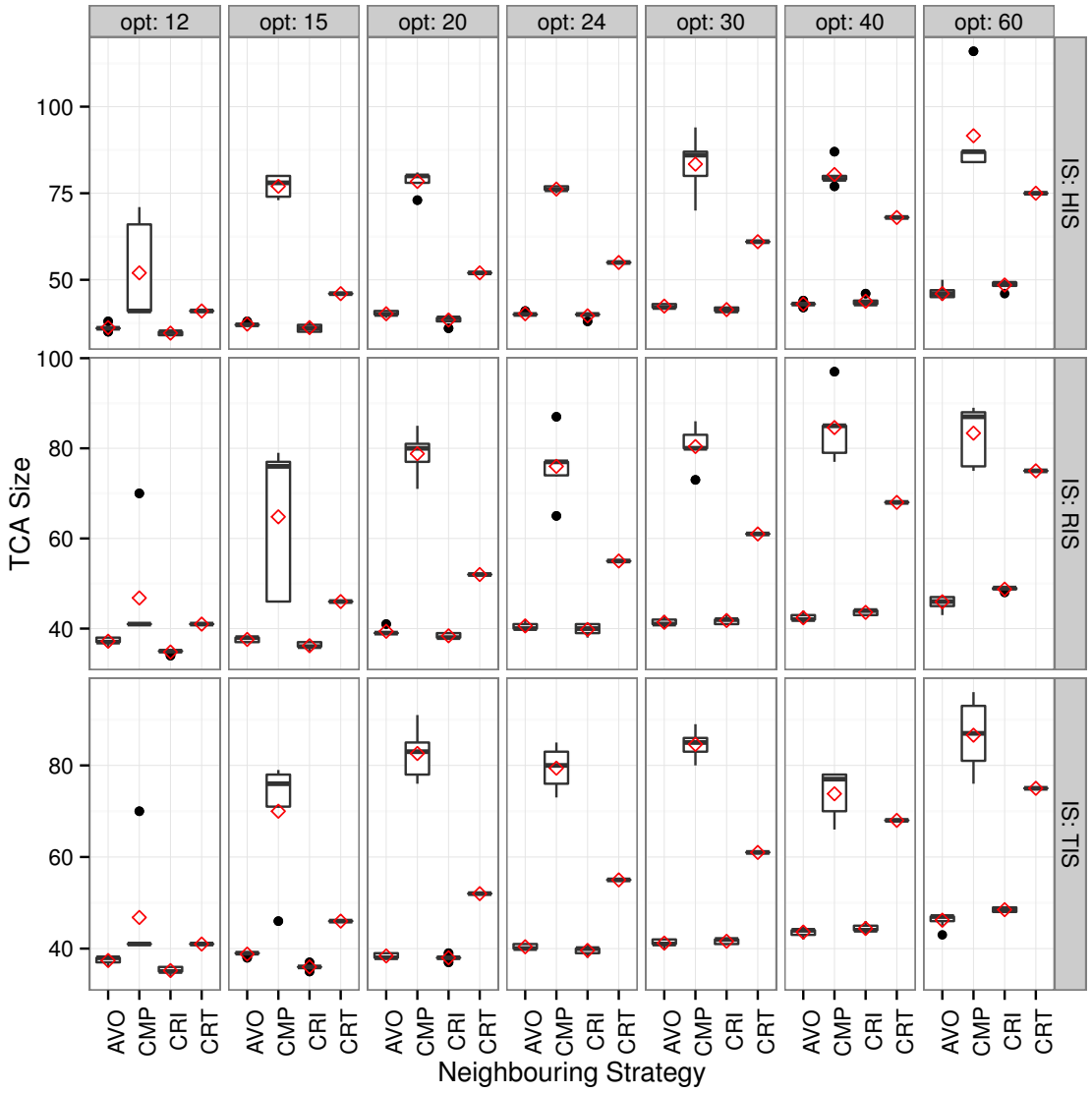

Figure 6.23: Comparing TCA sizes for neighboring strategies detailed for MySQL Configuration space models and $t = 2$

Figure 6.23 illustrates sizes of the test case-aware covering arrays computed by the neighboring strategies for each initialization strategy and configuration space model of MySQL, when $t = 2$. As the number of configuration options increase CRT losses performance, and CRI losses it's advantages to AVO.

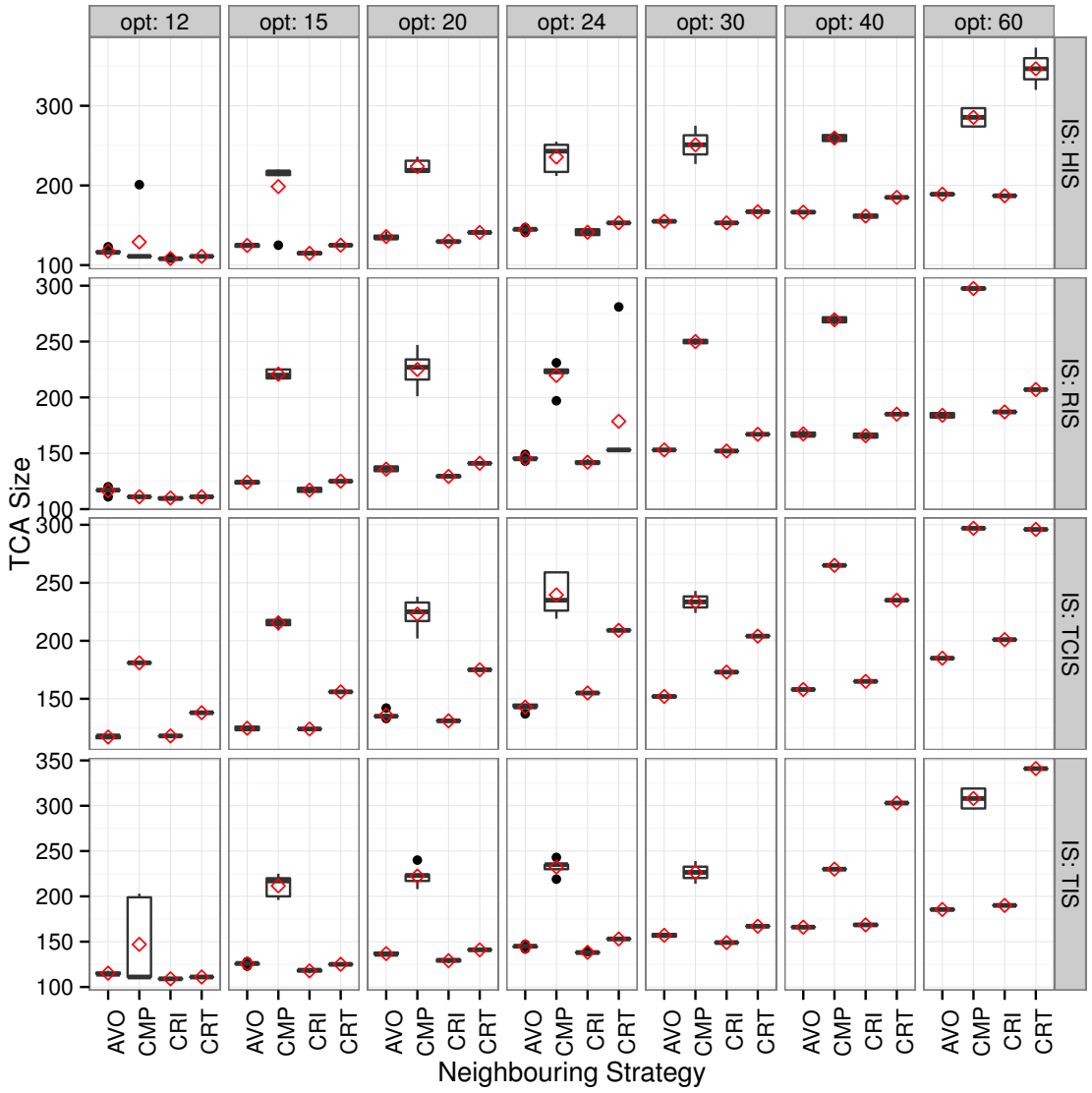

Figure 6.24: Comparing TCA sizes for neighboring strategies detailed for MySQL Configuration space models and $t = 3$

Figure 6.24 illustrates sizes of the test case-aware covering arrays computed by the neighboring strategies for each initialization strategy and configuration space model of MySQL, when $t = 3$. The observation from this graph is similar with Figure 6.23, indicating that; strength did not have a significant effect on the performance of neighboring strategies. As the number of configuration options increase, CRT losses performance, and CRI losses it's advantages to AVO.

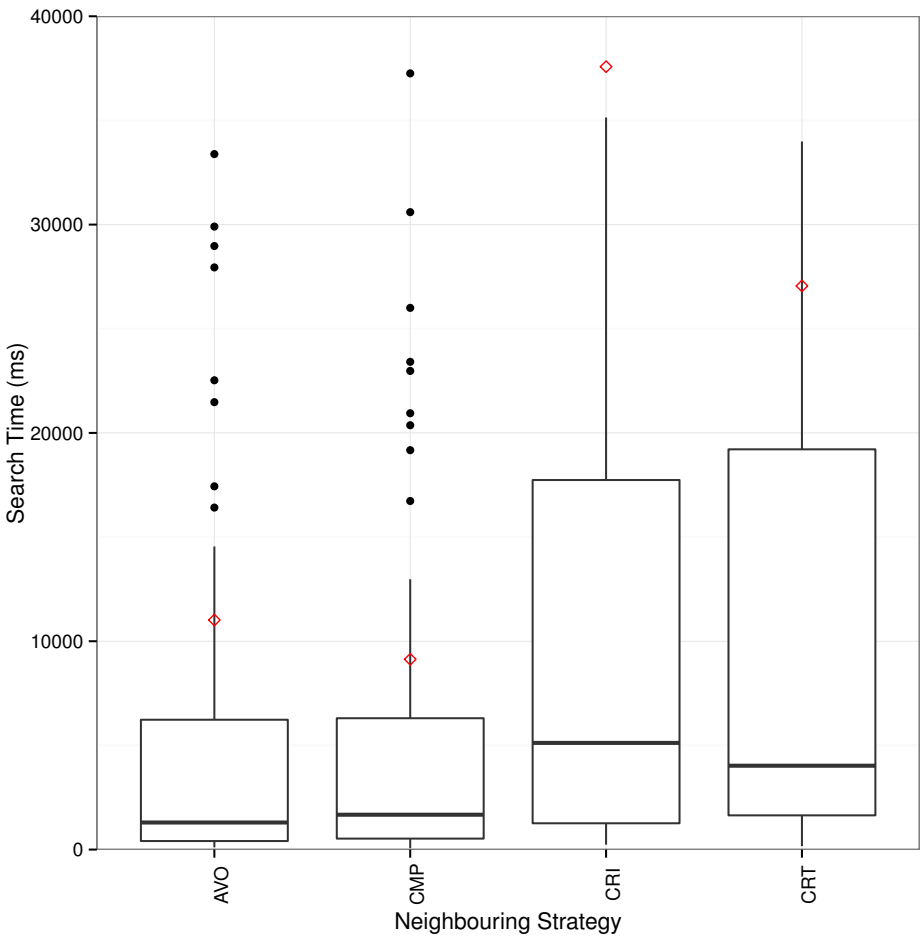

Figure 6.25: Comparing annealing times for neighboring strategies overall

Figure 6.25 illustrates the overall search time of the neighboring strategies. CMP strategy is the best in the search time. CRI strategy on the other hand, is the most time consuming one on overall.

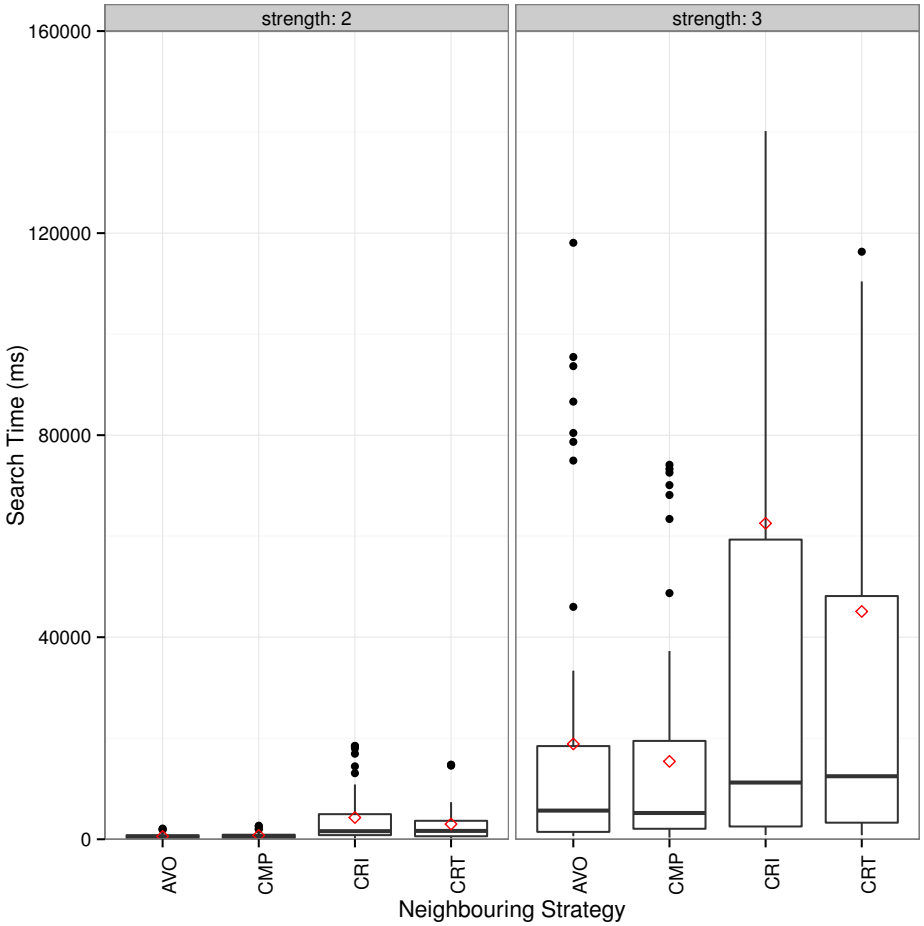

Figure 6.26: Comparing search times for neighboring strategies strength level

Figure 6.26 illustrates the search time of the neighboring strategies for each strength. For $t = 2$, AVO and CMP strategies have similar search times. For $t = 3$, CMP strategy is the fastest. CRI strategy on the other hand, is the most time consuming for both strength values.

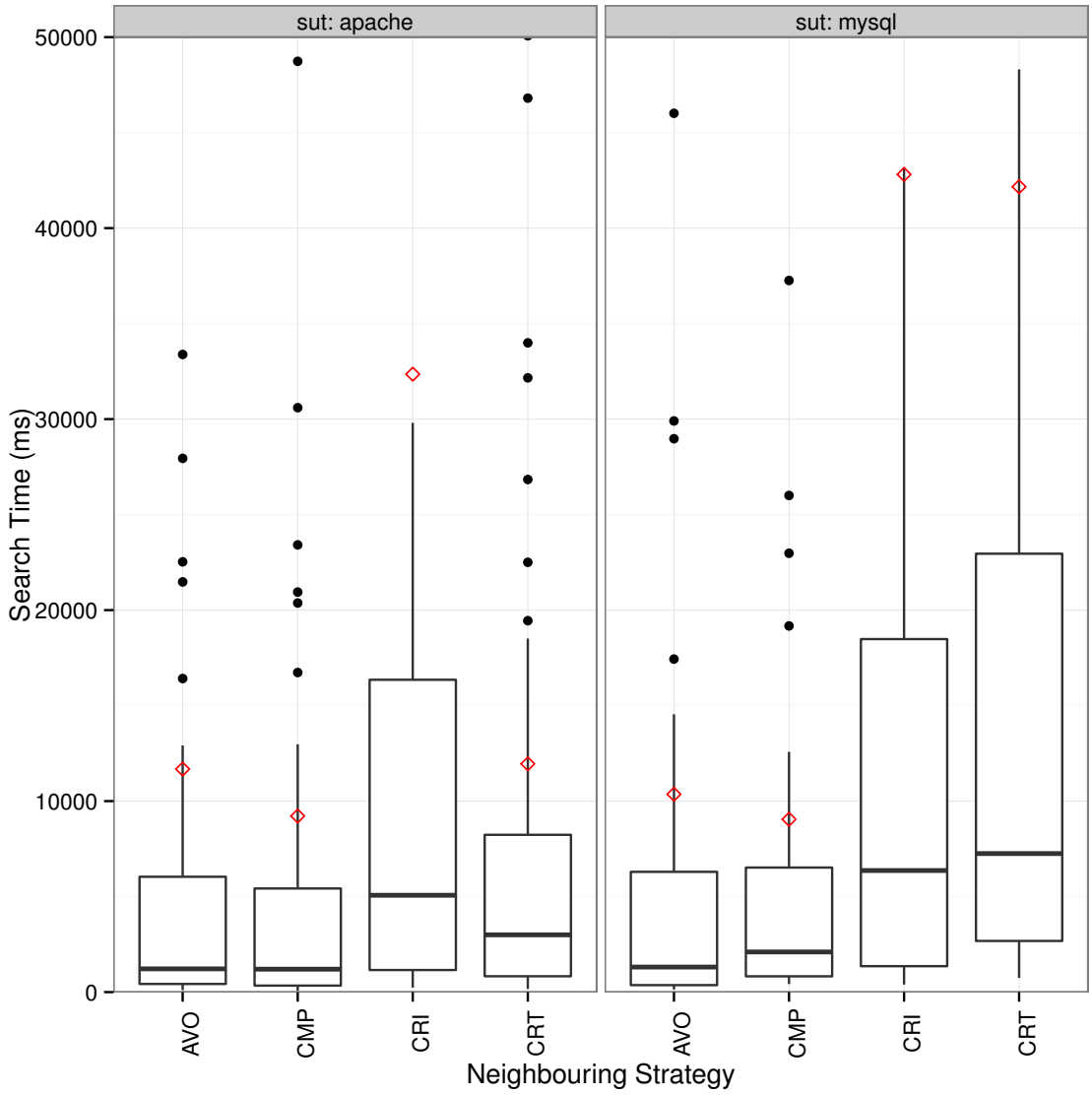

Figure 6.27: Comparing annealing times for neighboring strategies SUT level

Figure 6.27 illustrates the search time of the neighboring strategies for each subject application. For both of the subject applications, CMP strategy is the fastest and CRI is the most time consuming. There is a difference in the performances of CRI and CRT strategies for subject applications. For MySQL, CRI and CRT strategies have similar performance. However, CRT is better that CRI for Apache.

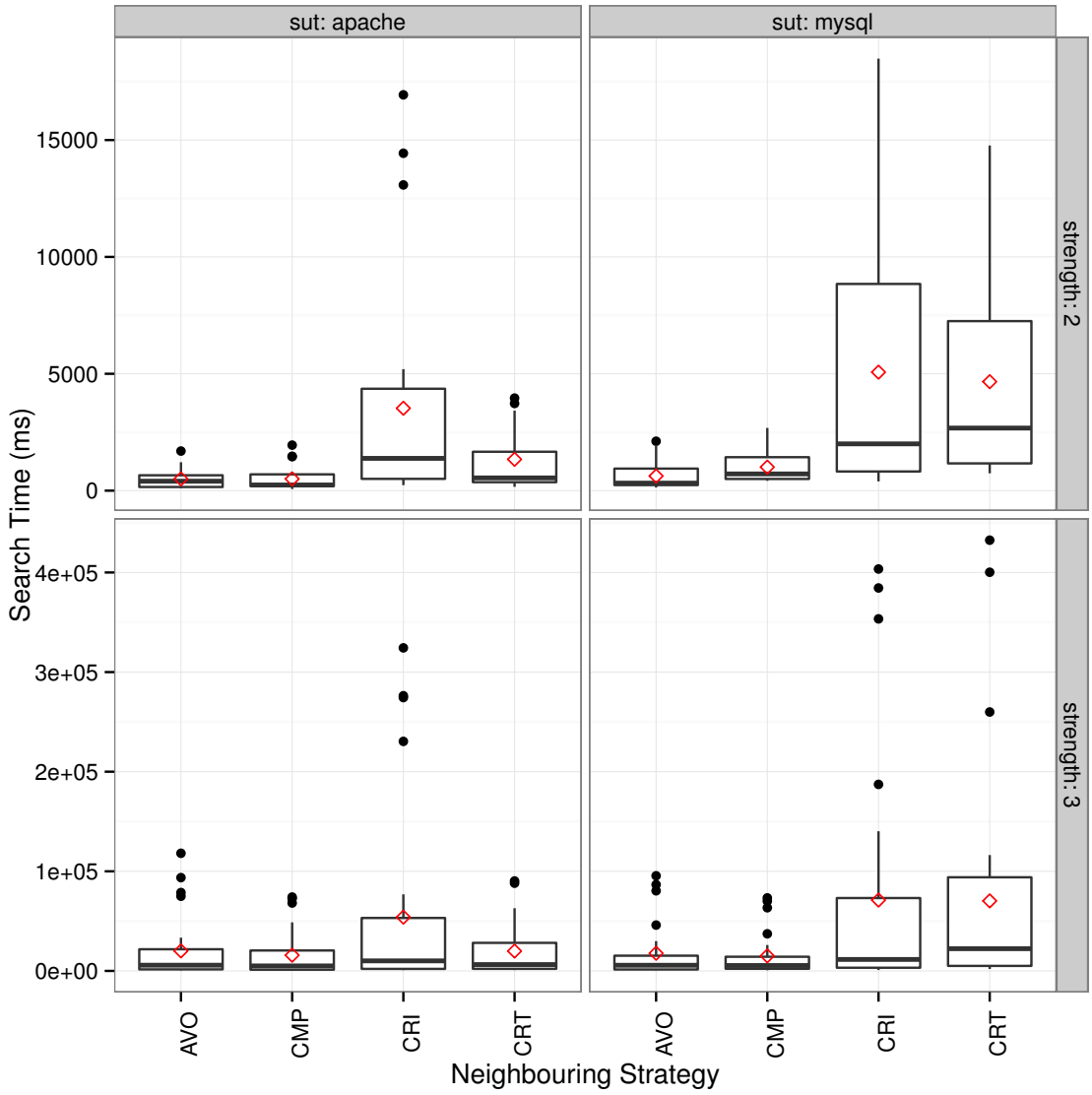

Figure 6.28: Comparing annealing times for neighboring strategies SUT by strength level

Figure 6.28 illustrates the search time of the neighboring strategies for each SUT and strength. Graph has similar pattern for strength levels, which means neighboring strategies behave similar for different strength values. However, for SUT levels the pattern is different, just like we observed in Figure 6.27.

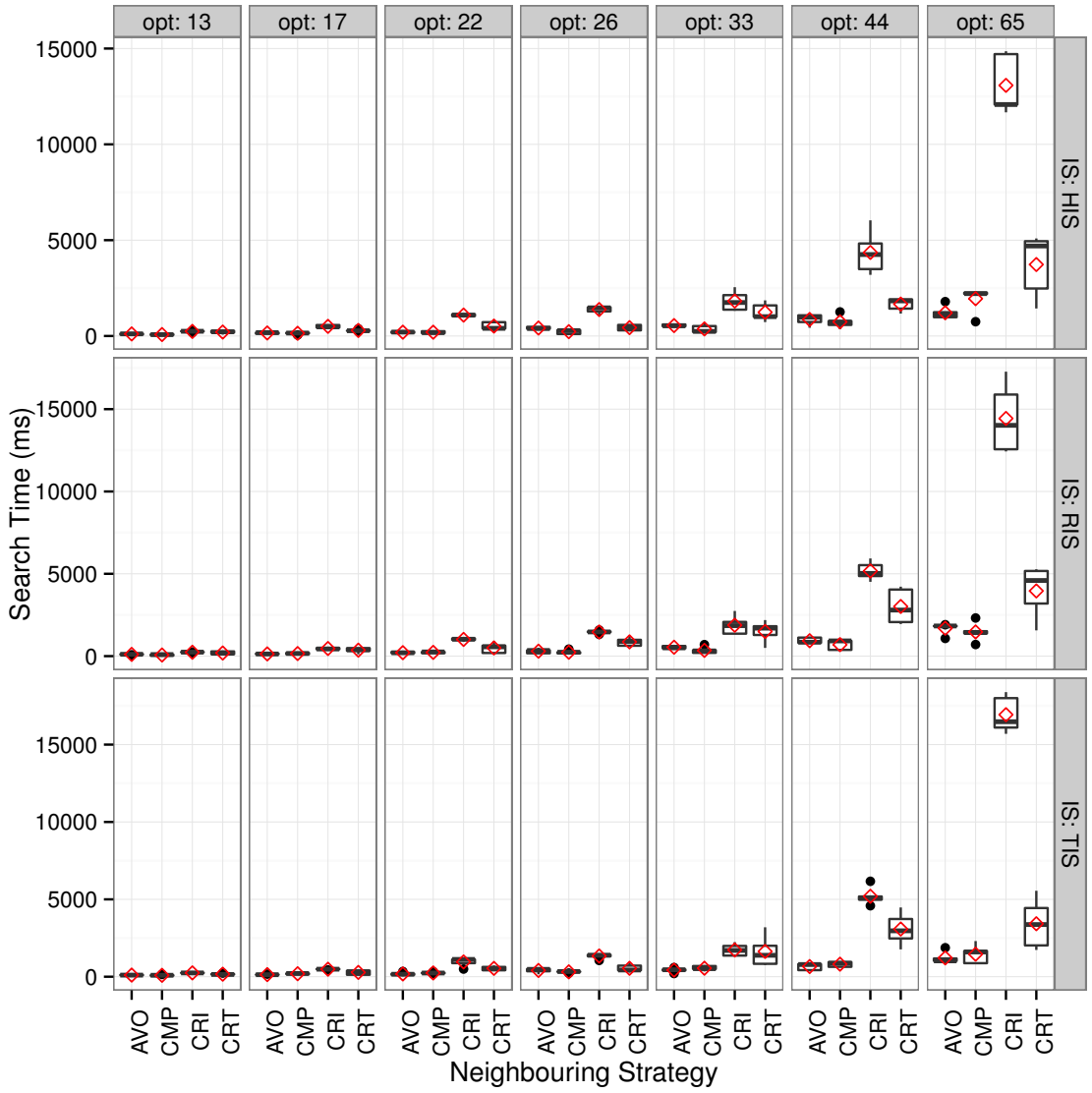

Figure 6.29: Comparing annealing times for neighboring strategies detailed for Apache configuration space models and $t = 2$

Figure 6.29 illustrates the search time of the neighboring strategies for each initialization strategy and configuration space model of Apache, for $t = 2$. On overall, as the number of the configuration options increase, the search time increases for every neighboring strategy, but CRI strategy is the most effected one among the others.

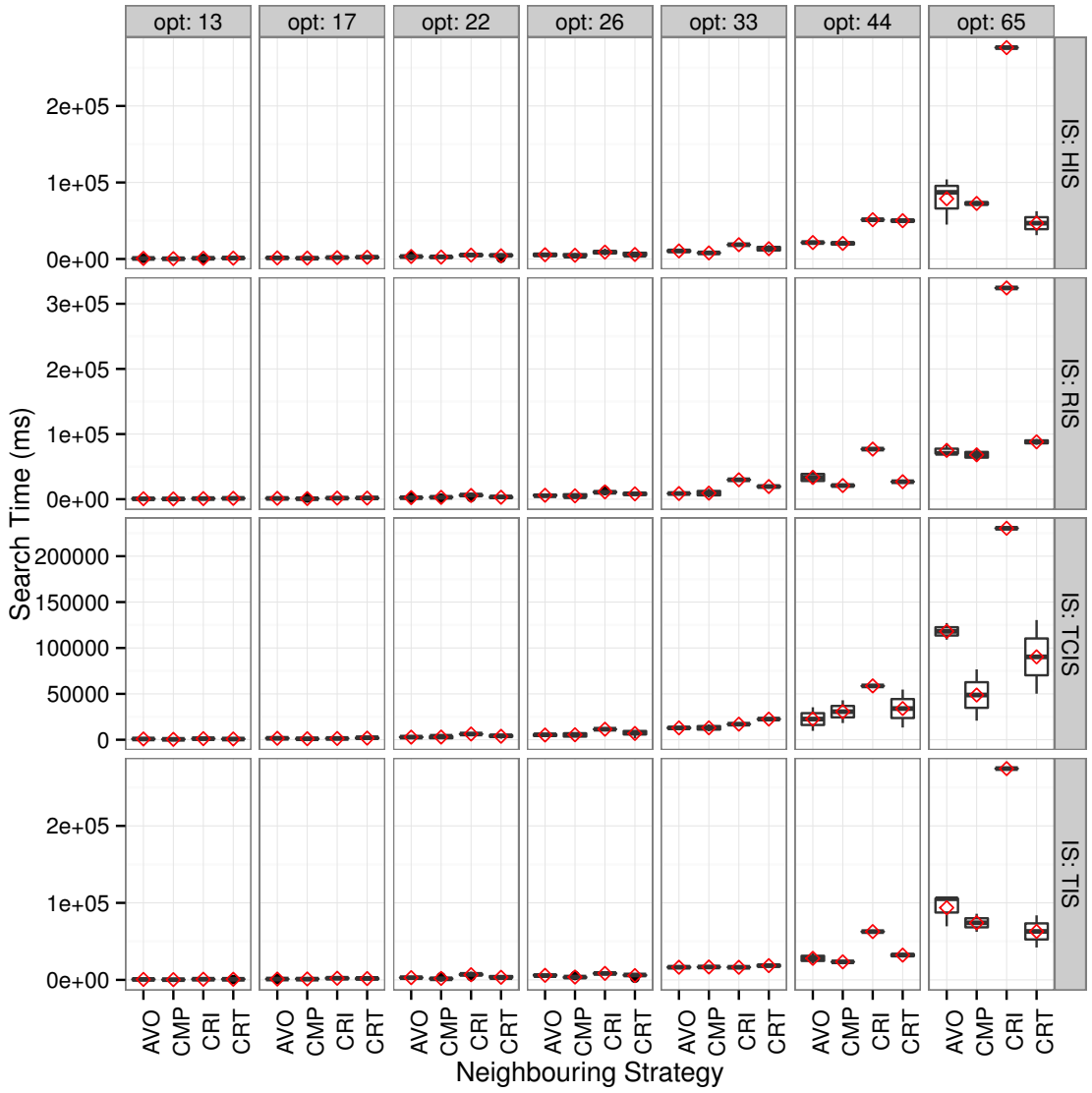

Figure 6.30: Comparing annealing times for neighboring strategies detailed for Apache configuration space models and $t = 3$

Figure 6.30 illustrates the search time of the neighboring strategies for each initialization strategy and configuration space model of Apache, when $t = 3$. On overall, as the number of the configuration options increase, the search time increases for every neighboring strategy, but CRI strategy is the most effected one among the others.

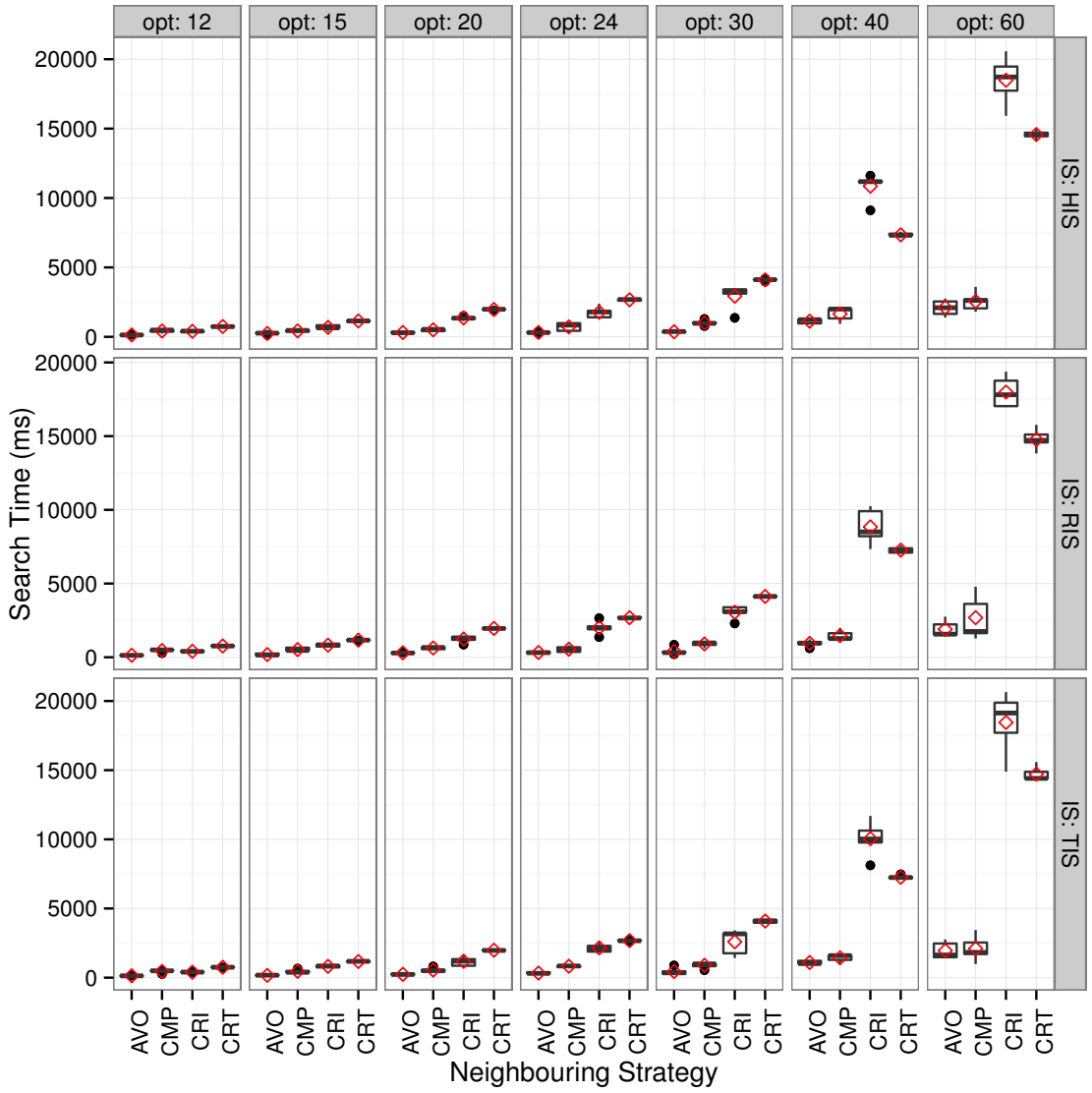

Figure 6.31: Comparing annealing times for neighboring strategies detailed for MySQL Configuration space models and $t = 2$

Figure 6.31 illustrates the search time of the neighboring strategies for each initialization strategy and configuration space model of MySQL, when $t = 2$. On overall, as the number of the configuration options increase, the search time increases for every neighboring strategy, but CRI strategy is the most effected one among the others.

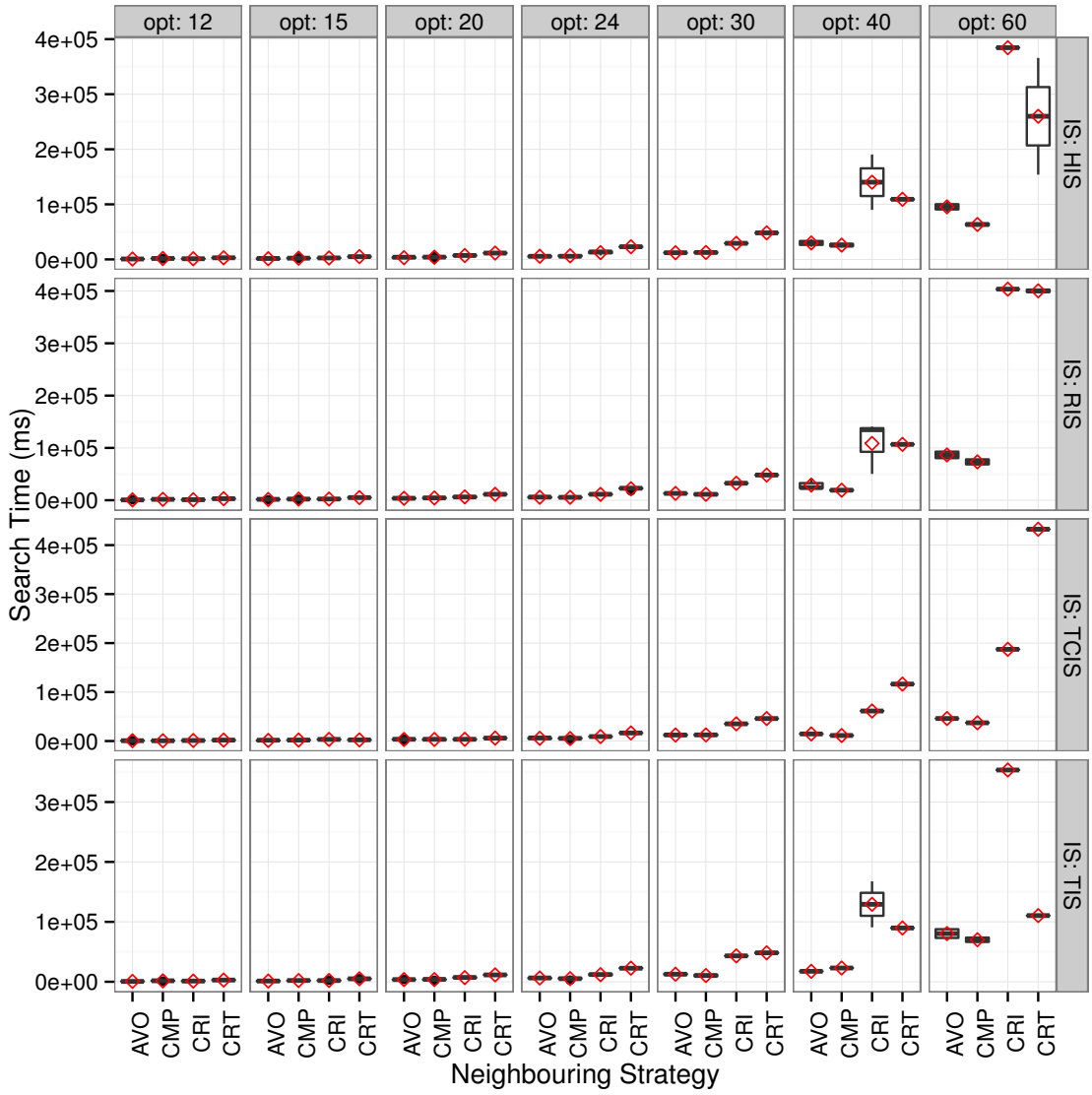

Figure 6.32: Comparing annealing times for neighboring strategies detailed for MySQL Configuration space models and $t = 3$

Figure 6.32 illustrates the search time of the neighboring strategies for each initialization strategy and configuration space model of MySQL, when $t = 3$. On overall, as the number of the configuration options increase, the search time increases for every neighboring strategy, but CRI strategy is the most effected one among the others.

This study has shown that, CRI strategy, which is commonly used by researchers [10, 12, 23,26,28], is also successful in computing Ψ that are smaller in size. CRI does not require any intelligence or state specific knowledge and perform a random transition blindly. Although CRI strategy is the fastest per iteration, compared to other strategies, it requires much more iterations to complete the task. On overall, CRI is the most time consuming one among the neighboring strategies.

CRT strategy which has been first experienced by Torres et al. [28], has generated TCAs larger in size compared to AVO. When we inspect the execution, we observed that CRT gets stacked in finding a suitable place for remaining a few missing pairs whose constraints contradict each other. Therefore, binary search increases the size of the search for just a few missing pairs in each iteration.

CMP strategy was fast but generated TCAs larger in size compared to AVO. When we inspect the execution, we observed that CMP has the same problem with CRT.

AVO strategy on the other hand, was the best in Ψ generation task. AVO succeeded to overcome the problem of stacking in finding suitable places for remaining a few missing pairs by altering constraint violating options of the row which host the last transition of the neighboring.

6.5.3. Study 3: Overall Comparison

On overall, we compared combination of the initialization and the neighboring strategies each other as well as with Algorithm 1 and Algorithm 2 which are introduced by Yilmaz et al. [32] (Algorithm 3 is out of scope of our objective for this study).

Following seven figures, (Figure 6.33,6.34,6.35,6.36,6.38,6.39) illustrate the effectiveness of neighboring strategies. In these graphs, y-axis is the total time for Ψ generation task and x-axis is the size of Ψ. Computing Ψ of minimum size in the minimum time is the desired case.

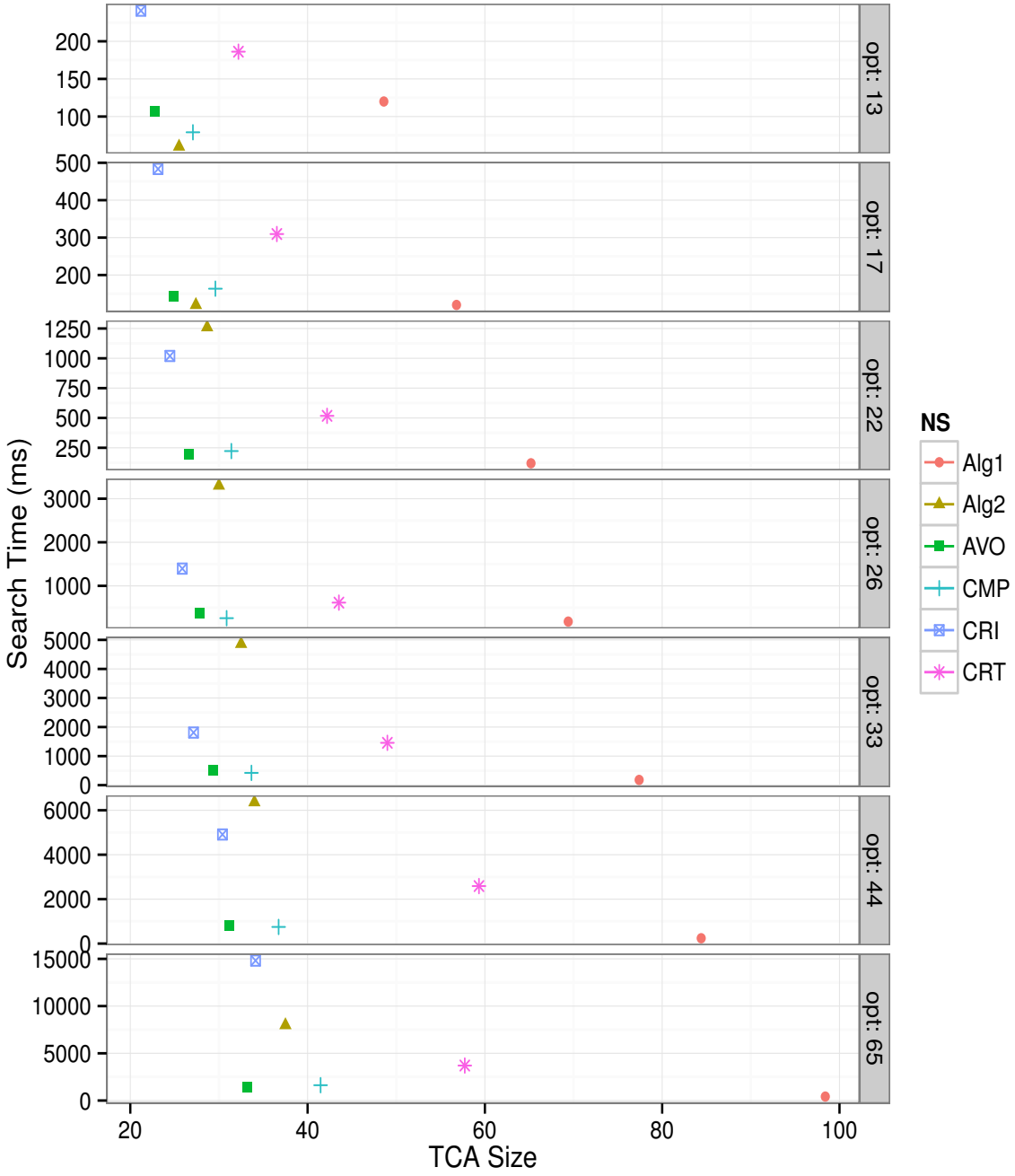

Figure 6.33: Comparing search times and TCA sizes of neighboring strategies for Apache configuration space models and $t = 2$

Figure 6.33 illustrates the search time and TCA size for Apache, $t = 2$. In this figure, it can be observed that as the configuration space model grows, AVO strategy gains advantage to the others. On overall, AVO strategy has generated smaller test case-aware covering arrays with minimum search times.

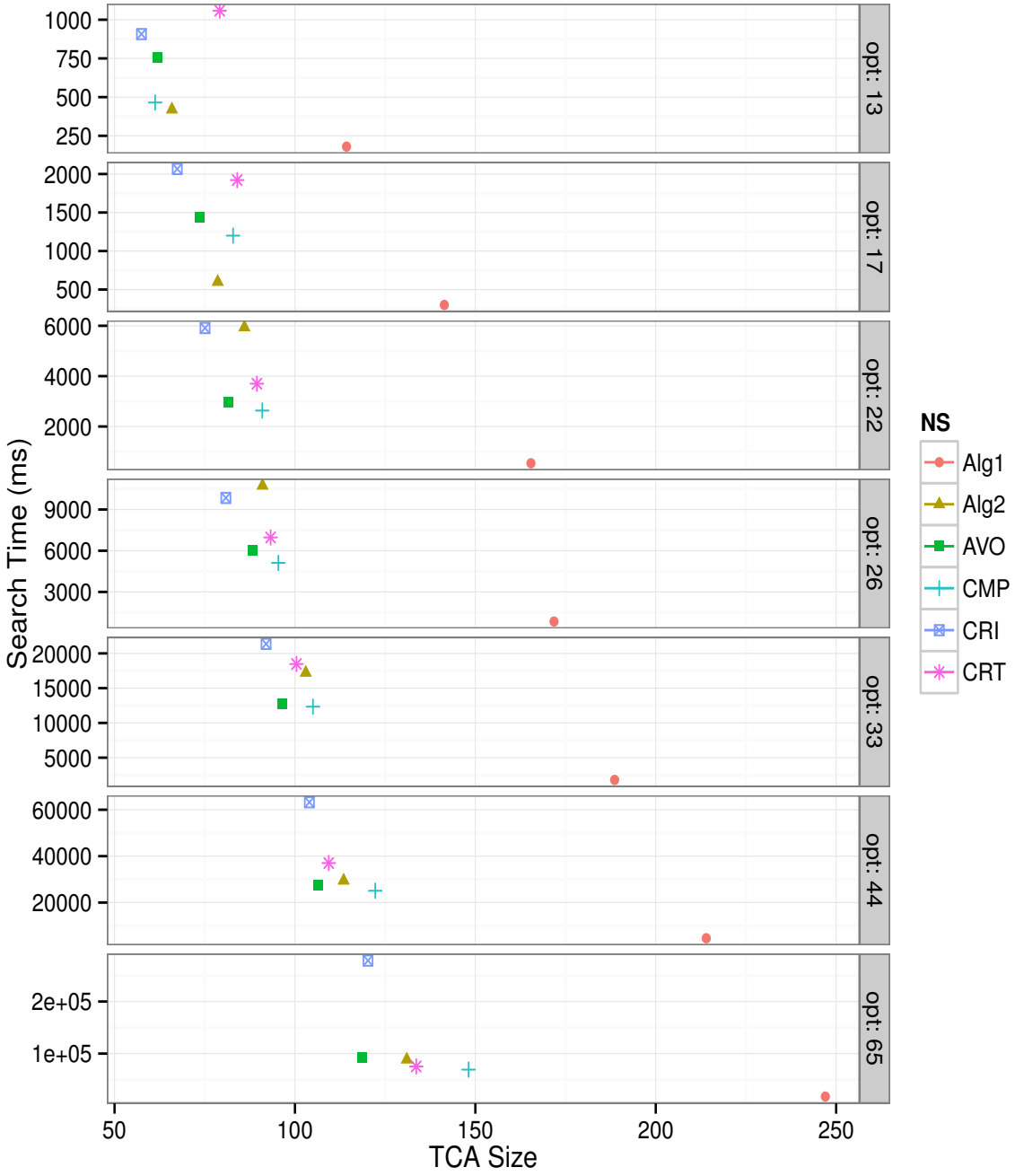

Figure 6.34: Comparing search times and TCA sizes of neighboring strategies for Apache configuration space models and $t = 3$

Figure 6.34 illustrates the search time and TCA size for Apache, $t = 3$. In this figure, it can be observed that as the configuration space model grows, AVO strategy gains advantage to the others. On overall, AVO strategy has generated smaller test case-aware covering arrays with minimum search times.

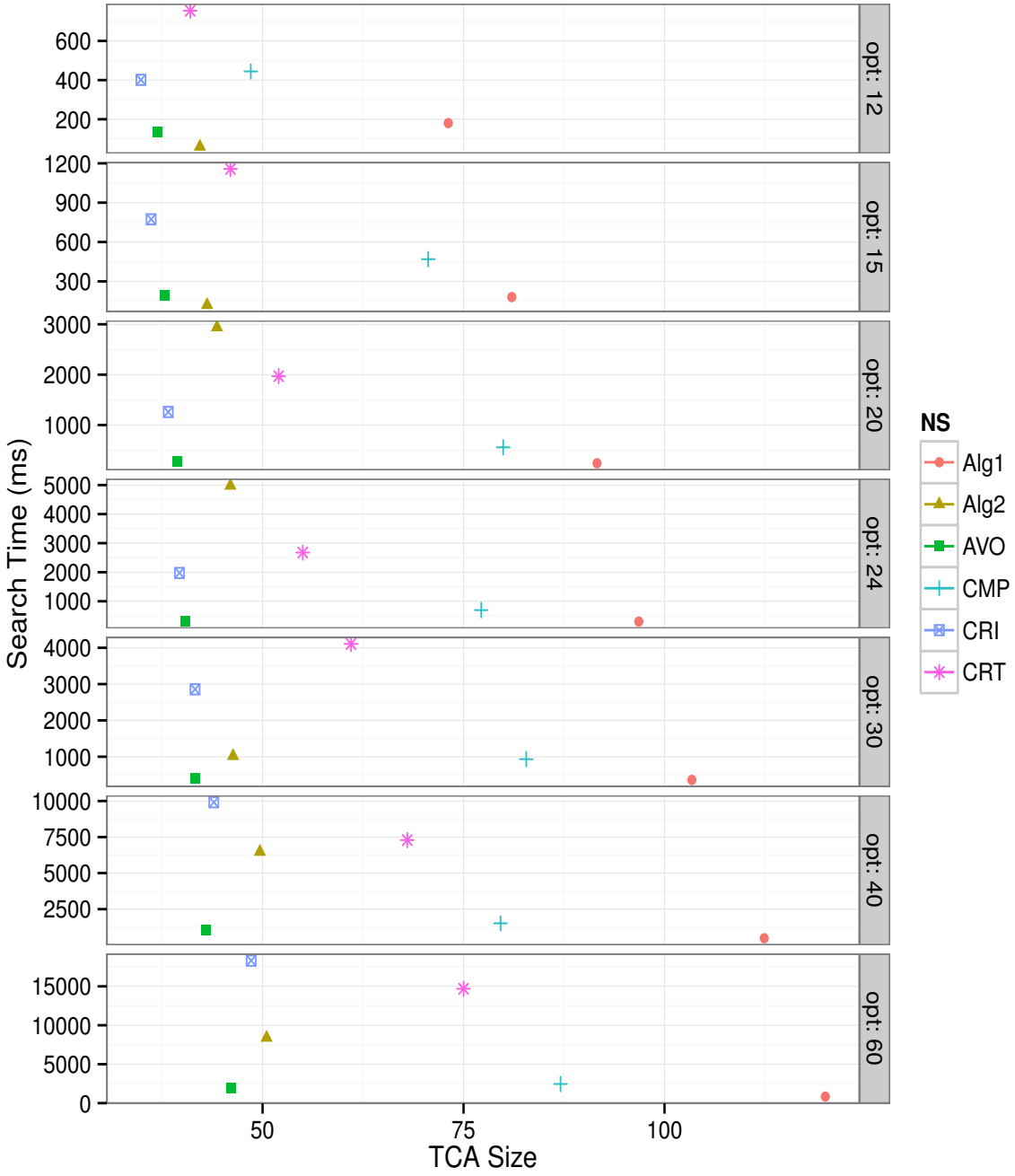

Figure 6.35: Comparing search times and TCA sizes of neighboring strategies for MySQL Configuration space models and $t = 2$

Figure 6.35 illustrates the search time and TCA size for Mysql, $t = 2$. In this figure, it can be observed that as the configuration space model grows, AVO strategy gains advantage to the others. On overall, AVO strategy has generated smaller test case-aware covering arrays with minimum search times.

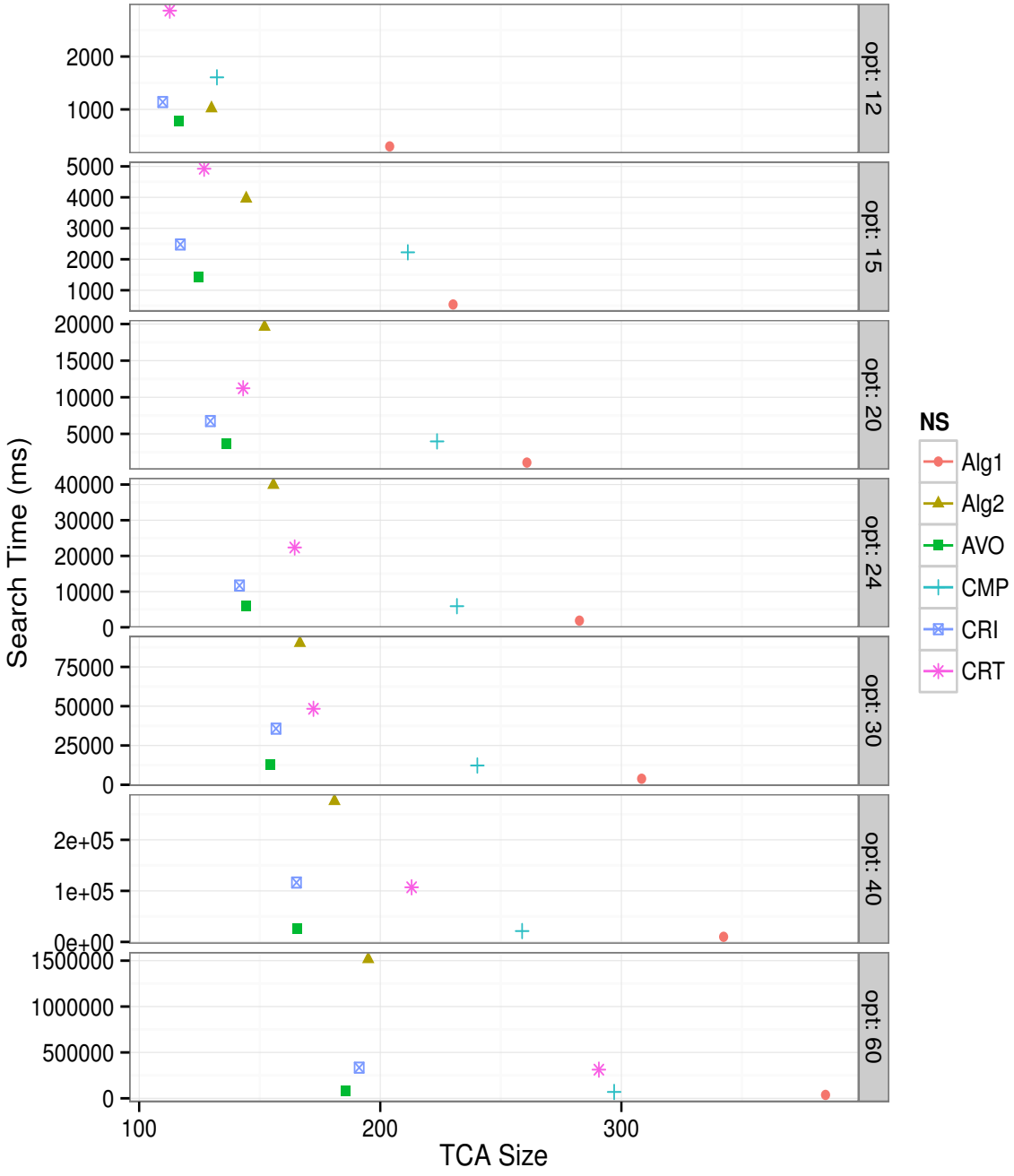

Figure 6.36: Comparing search times and TCA sizes of neighboring strategies for MySQL Configuration space models and $t = 3$

Figure 6.36 illustrates the search time and TCA size for Mysql, $t = 3$. In this figure, it can be observed that as the configuration space model grows, AVO strategy gains advantage to the others. On overall, AVO strategy has generated smaller test case-aware covering arrays with minimum search times.

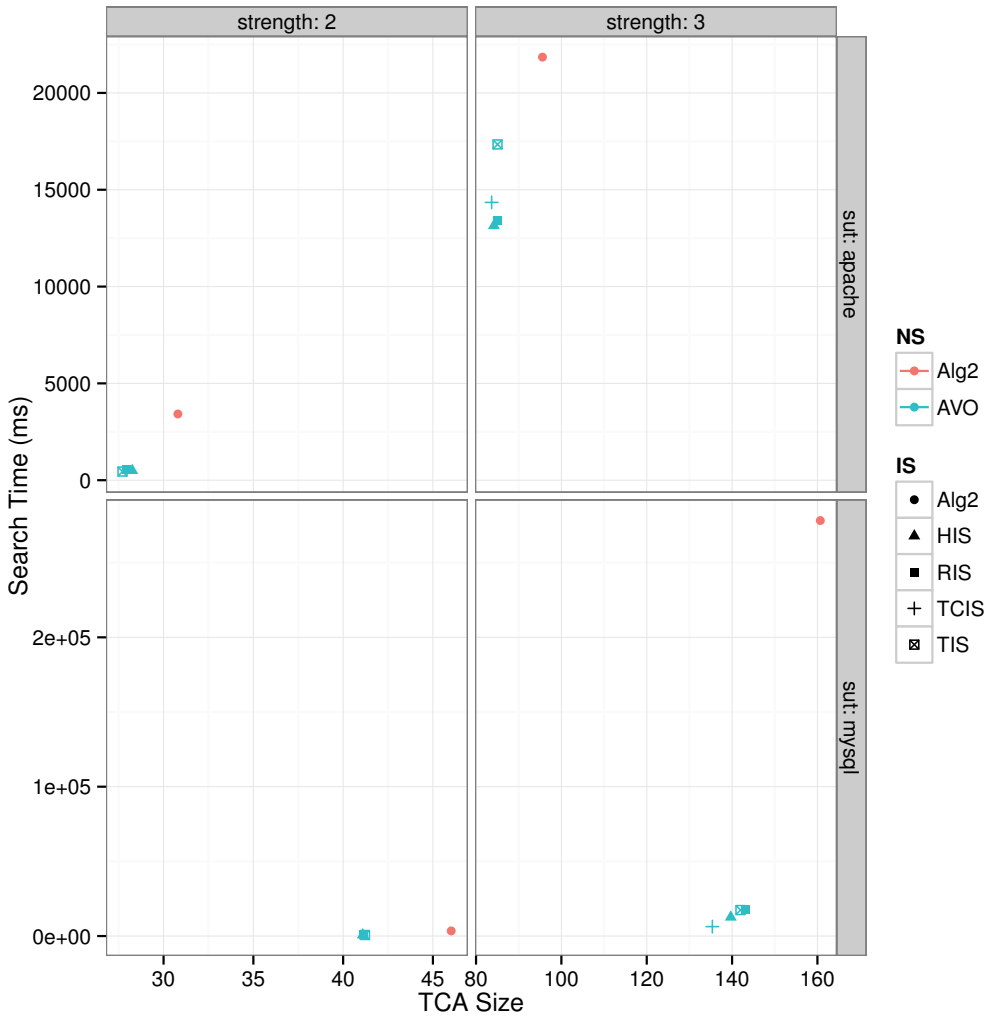

Figure 6.37: Comparing search times and TCA sizes for AVO strategy and Algorithm 2 at SUT by strength level

In Figures 6.37, 6.38, and 6.39, we compared search time and TCA sizes of ISxAVO combinations with Algorithm 2 only (Algorithm 1 is easy to beat in test case-aware covering array size). It can be observed that; for both of the subject applications and for both of the strength levels, AVO strategy has achieved to generate smaller TCAs with a fraction of construction cost compared to Algorithm 2. When $t = 3$, TCIS strategy were slightly better than the others for $t = 3$. When $t = 2$ however, initialization strategies did not have a significant effect on search time and TCA size.

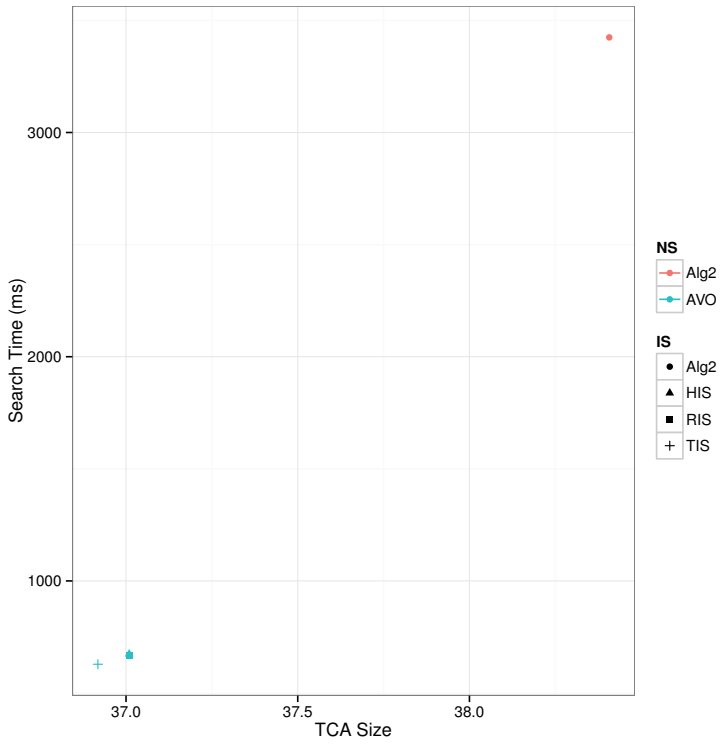

Figure 6.38: Comparing search times and TCA sizes for AVO strategy and Algorithm 2 for t = 2

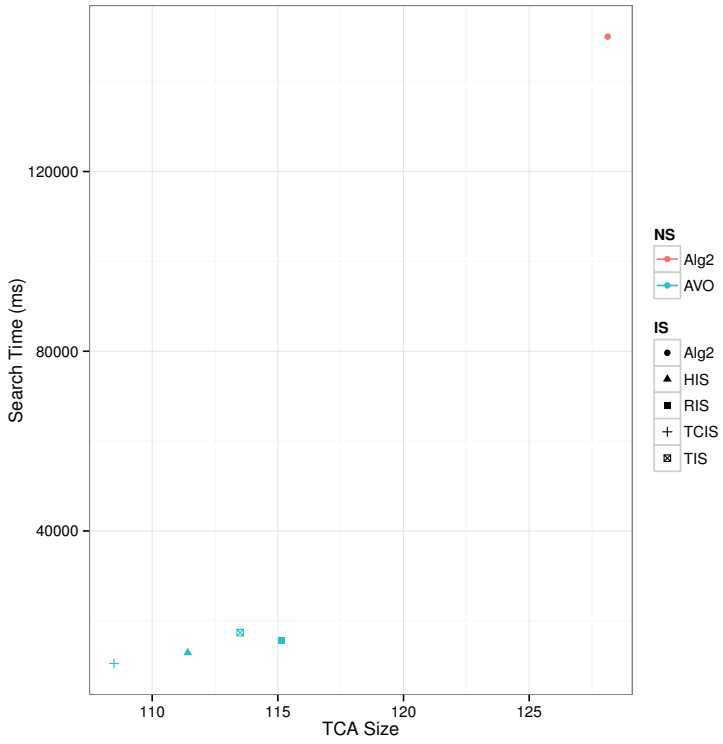

Figure 6.39: Comparing search times and TCA sizes for AVO strategy and Algorithm 2 for t = 3

This study has shown that; for small problems CRI strategy generates the smallest test case-aware covering arrays. However, it losses performance as the configuration space model grows. Another down side of CRI strategy is that; it is the most time consuming one among others.

CRT and CMP strategies are faster than CRI, however they generate test case-aware covering arrays that are larger in size compared to CRI and AVO.

AVO strategy on the other hand, gains advantage as the configuration space model grows. For large configuration space models, AVO strategy generated smallest test case-aware covering arrays with a fraction of computation cost compared to the others. On overall, AVO strategy has overruled the other strategies as well as existing algorithms (Algorithm 1 & 2).

subject app.	t	init. strategy	init. time(ms)	annealing time (ms)	total time (ms)	init. time percentage	anneal time percentage
apache	2	HIS	0.00	1348.21	1348.21	0.00	100.00
apache	2	RIS	0.00	1506.06	1506.06	0.00	100.00
apache	2	TIS	0.00	1547.79	1547.85	0.00	100.00
mysql	2	HIS	0.00	2888.85	2888.88	0.00	100.00
mysql	2	RIS	0.00	2801.71	2801.71	0.00	100.00
mysql	2	TIS	1.00	2833.34	2834.56	0.04	99.96
apache	3	HIS	0.00	25646.28	25646.30	0.00	100.00
apache	3	RIS	0.00	29797.34	29797.35	0.00	100.00
apache	3	TCIS	721.02	26874.60	27771.20	3.23	96.77
apache	3	TIS	2252.16	27379.12	29631.87	7.60	92.40
mysql	3	HIS	0.00	46476.69	46476.88	0.00	100.00
mysql	3	RIS	0.00	50692.71	50692.77	0.00	100.00
mysql	3	TCIS	659.78	38521.86	39328.17	2.05	97.95
mysql	3	TIS	1813.84	38452.11	40266.44	4.51	95.49

Table 6.3: Initialization time, annealing time, and time percentages

$$\text{init. time percentage} = \frac{\text{initialization time}}{\text{total time}} \times 100$$

$$\text{anneal time percentage} = \frac{\text{annealing time}}{\text{total time}} \times 100$$

Table 6.3 presents initialization time, annealing time, and time percentages for each initialization strategy and strength value. In the experiments, HIS and RIS strategies always had negligible initialization time. Therefore, initialization time percentage for them always measured as 0%. For $t = 2$, required time for TIS also negligible but it is the most time consuming one for $t = 3$. For $t = 2$, TCIS is not applicable and for $t = 3$.

6.6. Discussion

In these studies, the goal was to generate smaller test case-aware covering arrays with minimum construction cost. Therefore, we have compared only with Algorithm 1 and Algorithm 2 [32]. Algorithm 1, which aims at maintaining a separate configuration space model for each test case, generates test case-aware covering arrays larger in size but it is fast. Algorithm 2, which aims at maintaining a single configuration space model, on the other hand, generates test case-aware covering arrays smaller in size but it is very time consuming.

These testing objects, test case-aware covering arrays, are computed for ones and then used for many times for testing in general. For example, they can be used in daily test task. Thus, having smaller test suits is important to decrease the cost of testing. In this case, spending the necessary time to compute test case-aware covering arrays that are minimal in size is worthy. AVO and CRI strategies are better for this case.

However, if cost of testing is negligible, e.g. configuring the system has a cheap cost, then spending time to compute test case-aware covering arrays that are minimal in size is not needed. Algorithm 1 can be preferred for this case.

7

THREATS TO VALIDITY

In this thesis, we are primarily concerned with threats to external validity since they limit our ability to generalize the results of our studies to industrial practice.

First potential threat is the completeness of the used algorithms. Simulated annealing is not a complete algorithm; it does not exhaustively search the entire search space. Therefore, in theory, our approach may fail to find test case-aware covering arrays, even though there exist. To overcome this thread, in our approach, we have designed two level of search. While the inner search for the test case-aware covering arrays, the outer search helps to relax the problem in the case of failures. In the experiments we have conducted, our algorithms achieved to find test case-aware covering array in each execution. Also, in practice, simulated annealing is practically effective in traditional covering array generation task [23, 26].

Another potential thread is the appropriateness of the outer search (binary search) interval. In this thesis we have determined the search interval based on the published traditional covering array sizes. There is no guarantee to find the test case-aware covering array with a size in the determined interval. However in the experiments we have conducted, our algorithms always achieved to find a solution in the determined interval.

Another potential threat is that the proposed approach assumes that all test case-specific constraints are known a priori. In the presence of missing or incorrect constraints, as test cases can still skip some configurations due to unsatisfied constraints, the test case-aware covering arrays may suffer from masking effects. In such cases, the feedback driven adaptive combinational testing process we introduced in a prior work [15] can be used to iteratively detect and remove masking effects.

Another potential threat is that we have only studied two software systems; Apache and MySQL. However, both Apache and MySQL are widely-used non-trivial applications with large configuration spaces and both have been used in other related works in the literature [15, 16, 32].

A related threat concerns the representativeness of the configuration space models and the test suites used in the experiments. Although these configuration space models and test suites were culled from the actual configuration space models and test suites of our subject applications, they only represent two sets of data points. To reduce the threats concerning the representativeness of the configuration space models, we varied the percentage of constrained options in the models (Section 6.1).

Finally, we have not directly evaluated the cost-effectiveness of test case-aware covering arrays, i.e., evaluating the effectiveness, such as failure-detection capabilities, as a function of cost, such as total testing time. However, our empirical results reported in a prior work [15] strongly suggest that, as masking effects are removed, the number of failures observed and the structural code coverage obtained in testing monotonically increase.

8

CONCLUSION AND FUTURE WORK

In this thesis, we have focused on test case-aware covering array generation problem. We have developed simulated annealing-based, efficient and effective algorithms to compute test case-aware covering arrays and a tool implementing these algorithms.

To evaluate the effectiveness of our algorithms and tool, we conducted large-scale experiments on two widely-used highly-configurable software systems, namely Apache and MySQL. The results of our empirical studies strongly suggest that the proposed algorithms are an efficient and effective way of computing test case-aware covering arrays and that they perform better than existing approaches.

This study, first of all, has shown that local search-based methods, as in traditional covering arrays, can be used to compute test case-aware covering arrays. We have used simulated annealing algorithm, and it has achieved to compute test case-aware covering arrays.

In this study, as well as introducing novel initialization strategies, we have also leveraged existing initialization strategies, which have been used for computing traditional covering arrays (e.g. HIS and RIS). HIS (hamming distance initialization) strategy among them, was effective in generating initial sets that covers high number of t-pairs with rela-

tively negligible computation time. RIS (random initialization) strategy however, which is the most commonly used one for traditional covering arrays, was fast (again negligible computation time) but generated initial sets were covering fewer t-pairs compared to the others.

We have introduced and evaluated 2 more initialization strategies, namely TIS (traditional covering array as initial set) and TCIS ((t-1)-way test case-aware covering array as initial set). TCIS strategy has generated the best initial sets among others. TIS strategy is not the best or worst for any case. However, we have designed our approach capable of using a traditional covering array as an initial set, to account the already in use traditional covering arrays. Developers can seed their available covering arrays into our tool to generate test case-aware covering arrays. By doing so, their important configurations and testing objects will not be wasted.

For the neighbor generation task, we again introduced novel strategies, and leveraged existing neighboring strategies that are used in traditional covering arrays (e.g. CRI and CRT). It turned out that; CRI (change a random index) strategy, which is the most commonly used one for traditional covering arrays, is the most time consuming one. CRT (change a random t-tuple) on the other hand, computed test case-aware overing arrays larger in size and it was also relatively time consuming (compared to AVO and CMP).

Our novel neighboring strategy AVO (alter violating option) on the other hand, was the most effective one in the size of test case-aware covering arrays as well as in computation time. On overall, AVO strategy overruled the other strategies and also existing test case-aware covering array generation algorithms by achieving to compute test case-aware covering arrays that are smaller in size and with minimum computation time.

As future work, we first plan to focus on other local search algorithms such as genetic algorithm or tabu search for test case-aware covering array construction. We will then work on cost&test-case aware covering arrays that support a general cost model, in which the overall cost of testing can be specified at the granularity of option settings and test cases.

A

EMPIRICAL RESULTS

This appendix contains the row data from the experiment we have conducted. There is one table for each combination of the initialization and neighboring strategy. The headers are as follows:

sut : subject application, **t** : strength of the test case-aware covering array,

opt : option count of the configuration space model, **cop** : constrained option percentage,

 # of ϕ_t : number of valid t-tuples, **# of** λ_t : number of t-pairs,

size of Λ_t : number of valid t-pairs, **InitMiss** : initial miss count,

IT_{sa} : iteration count of SA, IT_{bs} : iteration count of binary search,

T_{init} : initialization time, T_{anneal} : annealing time, T_{total} : total time,

size : size of the computed test case-aware covering array,

T_{imp} : computation time improvement compared to Algorithm 2,

$$T_{imp} = \frac{\text{Algorithm 2 Time} - T_{total}}{\text{Algorithm 2 Time}} \times 100$$

N_{imp} : test case-aware covering array size improvement compared to Algorithm 2.

$$N_{imp} = \frac{\text{Algorithm 2 N} - \text{size}}{\text{Algorithm 2 N}} \times 100$$

sut	opt	t	cop	# of ϕ_t	# of λ_t	size of Λ_t	InitMiss	T_{init}	IT_{sa}	IT_{bs}	T_{anneal}	T_{total}	size	T_{imp}	N_{imp}
apache	13	2	100	311	5598	5013	222.4	0	380666.8	5.6	108.2	108.2	22.8	-80.33	10.59
apache	17	2	76	543	9774	8989	313.6	0	350162.4	5.6	156.4	156.4	25.2	-30.33	8.03
apache	22	2	59	923	16614	15579	293	0	261832.8	5.8	199.2	199.2	27.4	84.19	4.43
apache	26	2	50	1299	23382	22147	400.4	0	407511.6	6	415.6	415.6	27.4	87.41	8.67
apache	33	2	39	2111	37998	36413	648.4	0	306322.2	6	543.6	543.6	30	88.81	7.69
apache	44	2	30	3783	68094	65959	616	0	268755.4	6	861.6	861.6	31.6	86.45	7.06
apache	65	2	20	8319	149742	146557	1048.2	0	164065	6	1210.8	1210.8	33.4	84.83	10.93
mysql	12	2	100	307	9517	7756	1022.8	0	382986.4	6	136.2	136.4	36.2	-127.33	14.22
mysql	15	2	80	475	14725	12496	1528.8	0	457254.2	6.8	239.2	239.2	37.2	-99.33	13.69
mysql	20	2	60	835	25885	22876	1651.2	0	404729.4	6.4	312.8	312.8	40.2	89.36	9.32
mysql	24	2	50	1195	37045	33412	2723.4	0	269857.6	6	314.2	314.2	40.2	93.69	12.61
mysql	30	2	40	1855	57505	52936	3489.6	0	171163.2	6	365.4	365.4	42.4	64.18	8.48
mysql	40	2	30	3275	101525	95396	5524.6	0	368577.6	6.8	1127.6	1127.6	43	82.6	13.43
mysql	60	2	20	7315	226765	217516	9160.733	0	316671.9	6.53	2113.93	2114	46	74.83	8.91
apache	13	3	100	2266	40788	34530	931.4	0	470519.2	6.4	696.6	696.6	62	-65.86	5.92
apache	17	3	76	5410	97380	85842	1679.6	0	471161	7	1457	1457	72.8	-142.83	7.38
apache	22	3	59	12280	221040	200652	2991.2	0	476031.2	7.2	3266.6	3266.6	82.4	45.01	4.19
apache	26	3	50	20752	373536	344268	5131.4	0	509832.4	7.4	5567.6	5567.6	86.8	48.16	4.62
apache	33	3	39	43586	784548	735890	8686	0	394371	7	10506	10506	96.5	38.99	6.31
apache	44	3	30	105868	1905624	1816596	17732.5	0	373803.5	7.5	21476.5	21476.5	103.5	27.1	8.81
apache	65	3	20	349314	6287652	6087954	43257.33	0	456139	7.67	78679	78679.33	117.67	11.16	10.18
mysql	12	3	100	2176	67456	49557	3726.6	0	406093	7.6	690.2	690.2	117.4	32.33	9.69
mysql	15	3	80	4338	134478	105077	6438	0	457101.2	7.6	1288	1288	124.6	67.47	13.71
mysql	20	3	60	10448	323888	269077	11793	0	484311.6	7.8	3123	3123.4	135.8	84.08	10.66
mysql	24	3	50	18168	563208	482453	18572.4	0	530470.2	8	5689.2	5689.2	144.4	85.74	7.24
mysql	30	3	40	35668	1105708	976677	37461	0	549401	8	12122	12122	155	86.57	7
mysql	40	3	30	84888	2631528	2397077	64549	0	588212	8.5	29904	29904.5	166.5	89.16	8.01
mysql	60	3	20	287328	8907168	8368277	155667.5	0	560529.5	8.5	95481.5	95482.5	189	93.7	3.08

Table A.1: Statistics and improvements for HISxAVO combination

sut	opt	t	cop	# of ϕ_t	# of λ_t	size of Λ_t	Init.Miss	T_{init}	IT_{sa}	IT_{bs}	T_{anneal}	T_{total}	size	T_{imp}	N_{imp}
apache	13	2	100	311	5598	5013	222.4	0	249895.8	5.2	77.2	77.2	26	-28.67	-1.96
apache	17	2	76	543	9774	8989	313.6	0	313112.2	5.6	141.8	141.8	30.6	-18.17	-11.68
apache	22	2	59	923	16614	15579	293	0	270053.4	5.4	197.4	197.4	32	84.33	-11.61
apache	26	2	50	1299	23382	22147	400.4	0	203894.6	5.6	226	226	30.8	93.15	-2.67
apache	33	2	39	2111	37998	36413	648.4	0	224004.2	5.4	369.6	369.6	33.4	92.4	-2.77
apache	44	2	30	3783	68094	65959	616	0	253012.2	6	738.2	738.2	35.8	88.39	-5.29
apache	65	2	20	8319	149742	146557	1048.2	0	232632.6	6	1947.2	1947.2	38	75.6	-1.33
mysql	12	2	100	307	9517	7756	1022.8	0	588438.6	6.6	420.2	420.2	52	-600.33	-23.22
mysql	15	2	80	475	14725	12496	1528.8	0	521728	6.2	427.4	427.4	77	-256.17	-78.65
mysql	20	2	60	835	25885	22876	1651.2	0	445656.6	6.2	500.2	500.2	78.4	82.99	-76.86
mysql	24	2	50	1195	37045	33412	2723.4	0	460355	6.6	715.4	715.4	76.2	85.63	-65.65
mysql	30	2	40	1855	57505	52936	3489.6	0	459366.2	6.6	1004.4	1004.4	83.4	1.53	-80.01
mysql	40	2	30	3275	101525	95396	5524.6	0	416347.4	6.6	1652	1652.2	80.4	74.5	-61.87
mysql	60	2	20	7315	226765	217516	9015	0	340424.2	6	2554.4	2554.4	91.6	69.59	-81.39
apache	13	3	100	2266	40788	34530	931.4	0	307145.4	6.2	422	422	58	-0.48	11.99
apache	17	3	76	5410	97380	85842	1679.6	0	382196.8	7	1200.4	1200.4	82.8	-100.07	-5.34
apache	22	3	59	12280	221040	200652	2991.2	0	348007.4	7	2460.4	2460.4	90.4	58.58	-5.12
apache	26	3	50	20752	373536	344268	5131.4	0	417335	7.2	4879.4	4879.4	92.4	54.57	-1.54
apache	33	3	39	43586	784548	735890	8686	0	242685	7	7917	7917	112.5	54.02	-9.22
apache	44	3	30	105868	1905624	1816596	17732.5	0	228399	7	20370	20370	121.5	30.86	-7.05
apache	65	3	20	349314	6287652	6087954	38375.5	0	368872.5	7.5	72580	72580	158.5	18.04	-20.99
mysql	12	3	100	2176	67456	49557	3726.6	0	767752.8	8	1654.2	1654.2	129	-62.18	0.77
mysql	15	3	80	4338	134478	105077	6438	0	640153.6	7.8	2184.4	2184.4	198.6	44.84	-37.53
mysql	20	3	60	10448	323888	269077	11793	0	520511	7.8	3985.4	3985.6	223.8	79.69	-47.24
mysql	24	3	50	18168	563208	482453	18572.4	0	549158.6	8.2	6520.6	6520.8	235.6	83.66	-51.35
mysql	30	3	40	35668	1105708	976677	37461	0	460745	8	12580	12580	251	86.06	-50.6
mysql	40	3	30	84888	2631528	2397077	64549	0	488942	8	26002.5	26003	259.5	90.58	-43.37
mysql	60	3	20	287328	8907168	8368277	155667.5	0	317638.5	8	63413	63413.5	285.5	95.82	-46.41

Table A.2: Statistics and improvements for HISxCMP combination

sut	opt	t	cop	# of ϕ_t	# of λ_t	size of Λ_t	Init.Miss	T_{init}	IT_{sa}	IT_{bs}	T_{anneal}	T_{total}	size	T_{imp}	N_{imp}
apache	13	2	100	311	5598	5013	222.4	0	494488	5	239.6	239.6	21	-299.33	17.65
apache	17	2	76	543	9774	8989	313.6	0	797972.2	6	503.4	503.4	23.4	-319.5	14.6
apache	22	2	59	923	16614	15579	293	0	982455.4	6	1090.8	1090.8	24.4	13.43	14.89
apache	26	2	50	1299	23382	22147	400.4	0	892462.8	6	1381.6	1381.6	26.2	58.13	12.67
apache	33	2	39	2111	37998	36413	648.4	0	733980.2	5.4	1823	1823	27	62.49	16.92
apache	44	2	30	3783	68094	65959	616	0	978577.8	6	4359.6	4359.6	31	31.45	8.82
apache	65	2	20	8319	149742	146557	1048.2	0	1303805	6	13078.6	13078.6	34	-63.89	9.33
mysql	12	2	100	307	9517	7756	1022.8	0	789837.4	6	398	398	34.6	-563.33	18.01
mysql	15	2	80	475	14725	12496	1528.8	0	977166.2	6.4	679.8	679.8	36.2	-466.5	16.01
mysql	20	2	60	835	25885	22876	1651.2	0	999679.4	6.2	1357.2	1357.2	38.4	53.84	13.38
mysql	24	2	50	1195	37045	33412	2723.4	0	996863.6	6.2	1767	1767	39.6	64.52	13.91
mysql	30	2	40	1855	57505	52936	3489.6	0	875646.6	6	2927.4	2927.4	41.4	-187	10.64
mysql	40	2	30	3275	101525	95396	5524.6	0	1860645	6.8	10849.6	10849.6	43.8	-67.43	11.82
mysql	60	2	20	7315	226765	217516	8172.5	0	2278217	6.75	18482.25	18482.25	48.5	-120.03	3.96
apache	13	3	100	2266	40788	34530	1095.25	0	912794	6.75	864	864	57.25	-105.71	13.13
apache	17	3	76	5410	97380	85842	1493	0	822211.5	7	1794.25	1794.25	66	-199.04	16.03
apache	22	3	59	12280	221040	200652	3067	0	1027364	7.25	5068.75	5068.75	74.75	14.67	13.08
apache	26	3	50	20752	373536	344268	4966.25	0	943202.2	7.25	8952.5	8952.5	79.5	16.64	12.64
apache	33	3	39	43586	784548	735890	8600	0	846758	7	18656	18656	91	-8.34	11.65
apache	44	3	30	105868	1905624	1816596	20640	0	1021841	7	51312	51312	103	-74.18	9.25
apache	65	3	20	349314	6287652	6087954	43151	0	1564790	8	276248	276248	118	-211.93	9.92
mysql	12	3	100	2176	67456	49557	4017.75	0	1364804	7.75	1177.5	1177.5	108.25	-15.44	16.73
mysql	15	3	80	4338	134478	105077	6416.25	0	1410570	8	2621.25	2621.25	114.75	33.81	20.53
mysql	20	3	60	10448	323888	269077	12233	0	1399346	8	7020.75	7020.75	130	64.22	14.47
mysql	24	3	50	18168	563208	482453	20462.25	0	1336877	8	12296.75	12297	141.25	69.18	9.26
mysql	30	3	40	35668	1105708	976677	40553	0	1256262	8	29231	29231	153	67.61	8.2
mysql	40	3	30	84888	2631528	2397077	69742.5	0	2047554	8.5	140211.5	140211.5	161.5	49.19	10.77
mysql	60	3	20	287328	8907168	8368277	181310	0	1703960	8	384452	384452	187	74.65	4.1

Table A.3: Statistics and improvements for HISxCRI combination

sut	opt	t	cop	# of ϕ_t	# of λ_t	size of Λ_t	Init.Miss	T_{init}	IT_{sa}	IT_{bs}	T_{anneal}	T_{total}	size	T_{imp}	N_{imp}
apache	13	2	100	311	5598	5013	222.4	0	641414.8	5.6	206.2	206.2	32.2	-243.67	-26.27
apache	17	2	76	543	9774	8989	313.6	0	624010.2	5.2	288.6	288.6	36.6	-140.5	-33.58
apache	22	2	59	923	16614	15579	293	0	621481.4	5.4	516.8	516.8	43.6	58.98	-52.08
apache	26	2	50	1299	23382	22147	400.4	0	414801.4	5.2	435.6	435.6	45	86.8	-50
apache	33	2	39	2111	37998	36413	648.4	0	700899.8	6	1238.6	1238.6	48	74.51	-47.69
apache	44	2	30	3783	68094	65959	616	0	536213.4	5.6	1665	1665	56.8	73.82	-67.06
apache	65	2	20	8319	149742	146557	1048.2	0	382431.6	6	3729.8	3729.8	59	53.26	-57.33
mysql	12	2	100	307	9517	7756	1022.8	0	1918424	7	741.4	741.4	41	-1135.67	2.84
mysql	15	2	80	475	14725	12496	1528.8	0	2079812	7	1143.2	1143.2	46	-852.67	-6.73
mysql	20	2	60	835	25885	22876	1651.2	0	2181092	7	1965.8	1966	52	33.13	-17.3
mysql	24	2	50	1195	37045	33412	2723.4	0	2172204	7	2669.4	2669.6	55	46.39	-19.57
mysql	30	2	40	1855	57505	52936	3489.6	0	2199595	7	4103.6	4103.6	61	-302.31	-31.66
mysql	40	2	30	3275	101525	95396	5524.6	0	2240140	7	7347.4	7347.4	68	-13.39	-36.9
mysql	60	2	20	7315	226765	217516	9015	0	2095232	7	14572.4	14572.4	75	-73.48	-48.51
apache	13	3	100	2266	40788	34530	931.4	0	601583.2	6.8	1098.6	1098.6	75.8	-161.57	-15.02
apache	17	3	76	5410	97380	85842	1679.6	0	582387.2	6.8	2213.4	2213.4	90.6	-268.9	-15.27
apache	22	3	59	12280	221040	200652	2991.2	0	585350.8	7.6	4201.4	4201.4	85.4	29.27	0.7
apache	26	3	50	20752	373536	344268	5131.4	0	463016.2	7.2	5888.8	5888.8	91	45.17	0
apache	33	3	39	43586	784548	735890	8686	0	523874	7.5	13442.5	13442.5	101.5	21.94	1.46
apache	44	3	30	105868	1905624	1816596	17732.5	0	684156	8	50070.5	50070.5	105	-69.96	7.49
apache	65	3	20	349314	6287652	6087954	38375.5	0	238235.5	7.5	46806.5	46807	138.5	47.15	-5.73
mysql	12	3	100	2176	67456	49557	3726.6	0	1541308	8	2906.4	2906.4	111	-184.94	14.62
mysql	15	3	80	4338	134478	105077	6438	0	1567024	8	5032.8	5032.8	125	-27.09	13.43
mysql	20	3	60	10448	323888	269077	11793	0	1600183	8	11507.6	11507.8	141	41.35	7.24
mysql	24	3	50	18168	563208	482453	18572.4	0	1868423	9	22953.8	22954.2	153	42.47	1.72
mysql	30	3	40	35668	1105708	976677	37461	0	1882042	9	48203	48203.5	167	46.58	-0.2
mysql	40	3	30	84888	2631528	2397077	64549	0	1901488	9	109154.5	109154.5	185	60.44	-2.21
mysql	60	3	20	287328	8907168	8368277	170758	0	1143204	8.5	259940.5	259941	346.5	82.86	-77.69

Table A.4: Statistics and improvements for HISxCRT combination

sut	opt	t	cop	# of ϕ_t	# of λ_t	size of Λ_t	Init.Miss	T_{init}	IT_{sa}	IT_{bs}	T_{anneal}	T_{total}	size	T_{imp}	N_{imp}
apache	13	2	100	311	5598	5013	309.8	0	402652.4	5.8	105.8	105.8	22.8	-76.33	10.59
apache	17	2	76	543	9774	8989	467.6	0	295715.8	5.2	135.6	135.6	24.8	-13	9.49
apache	22	2	59	923	16614	15579	661.4	0	285094.8	5.8	209.8	209.8	26.4	83.35	7.92
apache	26	2	50	1299	23382	22147	686.8	0	307398.8	5.6	301.2	301.2	28	90.87	6.67
apache	33	2	39	2111	37998	36413	957.8	0	309057.8	6	540.2	540.2	29.6	88.88	8.92
apache	44	2	30	3783	68094	65959	1096	0	297188.8	6	935.2	935.2	31.2	85.3	8.24
apache	65	2	20	8319	149742	146557	1442.4	0	166194.6	6	1693.2	1693.2	32.8	78.78	12.53
mysql	12	2	100	307	9517	7756	1243.2	0	388149.2	6	133.6	133.6	37.2	-122.67	11.85
mysql	15	2	80	475	14725	12496	1435	0	358579.6	6.4	182	182	37.6	-51.67	12.76
mysql	20	2	60	835	25885	22876	1951.6	0	326929.8	6.2	299.4	299.4	39.4	89.82	11.12
mysql	24	2	50	1195	37045	33412	2945.8	0	286246.8	6	322.2	322.2	40.6	93.53	11.74
mysql	30	2	40	1855	57505	52936	4895.8	0	223609.2	6.2	406	406	41.4	60.2	10.64
mysql	40	2	30	3275	101525	95396	6369.6	0	337268.4	6.6	942.8	942.8	42.4	85.45	14.64
mysql	60	2	20	7315	226765	217516	9262.471	0	283407.9	6.41	1894.65	1894.71	45.94	77.44	9.03
apache	13	3	100	2266	40788	34530	1850.4	0	445818.2	6.4	679	679	61.6	-61.67	6.53
apache	17	3	76	5410	97380	85842	2332.8	0	425497.4	7	1333.6	1333.6	74	-122.27	5.85
apache	22	3	59	12280	221040	200652	4441.2	0	370486.2	7	2425.8	2425.8	81.4	59.16	5.35
apache	26	3	50	20752	373536	344268	6422	0	510030.8	7.4	6038.8	6038.8	89.2	43.77	1.98
apache	33	3	39	43586	784548	735890	10336.5	0	381132	7	8700.5	8700.5	96	49.47	6.8
apache	44	3	30	105868	1905624	1816596	21054	0	505648	7.5	33385	33385	108.5	-13.32	4.41
apache	65	3	20	349314	6287652	6087954	37360	0	416046.3	7.67	74967.33	74967.33	118.33	15.35	9.67
mysql	12	3	100	2176	67456	49557	4991	0	442790	7.2	695.6	695.6	116.4	31.8	10.46
mysql	15	3	80	4338	134478	105077	7945.2	0	563733.4	7.8	1461.2	1461.2	124	63.1	14.13
mysql	20	3	60	10448	323888	269077	14038.2	0	594030.2	8	3650.6	3650.6	135.8	81.39	10.66
mysql	24	3	50	18168	563208	482453	21817.6	0	536938	8	5658.4	5658.4	145.6	85.82	6.47
mysql	30	3	40	35668	1105708	976677	35060	0	507239	8	13067.5	13068	153	85.52	8.2
mysql	40	3	30	84888	2631528	2397077	79213.67	0	539031.3	8.33	28971.67	28971.67	167.33	89.5	7.55
mysql	60	3	20	287328	8907168	8368277	254203	0	480272	8	86638.5	86639	184	94.29	5.64

Table A.5: Statistics and improvements for RISxAVO combination

sut	opt	t	cop	# of ϕ_t	# of λ_t	size of Λ_t	Init.Miss	T_{init}	IT_{sa}	IT_{bs}	T_{anneal}	T_{total}	size	T_{imp}	N_{imp}
apache	13	2	100	311	5598	5013	309.8	0	220253.2	5	69	69	27.8	-15	-9.02
apache	17	2	76	543	9774	8989	467.6	0	323429.8	5.4	156	156	29.4	-30	-7.3
apache	22	2	59	923	16614	15579	661.4	0	327662	5.8	226.4	226.4	31	82.03	-8.13
apache	26	2	50	1299	23382	22147	686.8	0	235782.6	5.6	235.2	235.2	30	92.87	0
apache	33	2	39	2111	37998	36413	957.8	0	216386.4	5.6	348.2	348.2	34.2	92.84	-5.23
apache	44	2	30	3783	68094	65959	1096	0	246599.2	5.6	697.4	697.4	38.6	89.03	-13.53
apache	65	2	20	8319	149742	146557	1442.4	0	178588.6	6	1469.6	1469.6	39.8	81.58	-6.13
mysql	12	2	100	307	9517	7756	1243.2	0	646519	6.8	456.6	456.6	46.8	-661	-10.9
mysql	15	2	80	475	14725	12496	1435	0	564273.2	6.4	519.4	519.4	64.8	-332.83	-50.35
mysql	20	2	60	835	25885	22876	1951.6	0	531726.4	6.6	626.4	626.4	78.8	78.69	-77.76
mysql	24	2	50	1195	37045	33412	2945.8	0	364495.2	6.2	546.2	546.2	76	89.03	-65.22
mysql	30	2	40	1855	57505	52936	4895.8	0	399278.6	6.2	900.2	900.2	80.4	11.75	-73.54
mysql	40	2	30	3275	101525	95396	6369.6	0	412241.2	6.4	1465.2	1465.2	84.6	77.39	-70.32
mysql	60	2	20	7315	226765	217516	9449.364	0	319473.3	6.45	2683.82	2683.82	83.36	68.05	-65.07
apache	13	3	100	2266	40788	34530	1850.4	0	288288	6.4	428.8	429	61.4	-2.14	6.83
apache	17	3	76	5410	97380	85842	2332.8	0	294880	6.6	956	956	83.4	-59.33	-6.11
apache	22	3	59	12280	221040	200652	4441.2	0	408990.8	7	2812	2812	88.6	52.66	-3.02
apache	26	3	50	20752	373536	344268	6422	0	436631.4	7.4	5143.2	5143.2	97.4	52.11	-7.03
apache	33	3	39	43586	784548	735890	10336.5	0	355295	7.5	9298	9298	105	46	-1.94
apache	44	3	30	105868	1905624	1816596	21054	0	235957.5	7	20944.5	20944.5	121.5	28.91	-7.05
apache	65	3	20	349314	6287652	6087954	36098	0	307892	7	68156	68156	151	23.04	-15.27
mysql	12	3	100	2176	67456	49557	4991	0	788958.8	8	1689.8	1689.8	111	-65.67	14.62
mysql	15	3	80	4338	134478	105077	7945.2	0	634176.8	7.8	2158.4	2158.4	220.8	45.49	-52.91
mysql	20	3	60	10448	323888	269077	14038.2	0	573831	7.8	4311.4	4311.4	225	78.03	-48.03
mysql	24	3	50	18168	563208	482453	21817.6	0	442600.8	8	5655.6	5655.6	219.6	85.83	-41.07
mysql	30	3	40	35668	1105708	976677	35060	0	468674	8	11303	11303	250	87.47	-50
mysql	40	3	30	84888	2631528	2397077	77310.5	0	341781.5	8	19171	19171	269.5	93.05	-48.9
mysql	60	3	20	287328	8907168	8368277	254203	0	301067.5	8	73313	73313	297.5	95.17	-52.56

Table A.6: Statistics and improvements for RISxCMP combination

sut	opt	t	cop	# of ϕ_t	# of λ_t	size of Λ_t	Init.Miss	T_{init}	IT_{sa}	IT_{bs}	T_{anneal}	T_{total}	size	T_{imp}	N_{imp}
apache	13	2	100	311	5598	5013	309.8	0	636917.6	5	235.4	235.4	21.6	-292.33	15.29
apache	17	2	76	543	9774	8989	467.6	0	717408.6	5.8	462	462	23	-285	16.06
apache	22	2	59	923	16614	15579	661.4	0	967024.4	6	1014	1014	24.2	19.52	15.59
apache	26	2	50	1299	23382	22147	686.8	0	879546.8	6	1465.4	1465.4	26	55.59	13.33
apache	33	2	39	2111	37998	36413	957.8	0	736939.8	5.2	1877.4	1877.4	27.2	61.37	16.31
apache	44	2	30	3783	68094	65959	1096	0	1189651	6	5171.4	5171.4	30.2	18.69	11.18
apache	65	2	20	8319	149742	146557	1442.4	0	1349022	6	14435	14435	34	-80.89	9.33
mysql	12	2	100	307	9517	7756	1243.2	0	785944	6	406.4	406.4	34.8	-577.33	17.54
mysql	15	2	80	475	14725	12496	1435	0	1178351	6.8	819.4	819.4	36.2	-582.83	16.01
mysql	20	2	60	835	25885	22876	1951.6	0	929693	6	1235.6	1235.6	38.4	57.97	13.38
mysql	24	2	50	1195	37045	33412	2945.8	0	1049890	6.4	2002	2002	39.8	59.8	13.48
mysql	30	2	40	1855	57505	52936	4895.8	0	919147.8	6	3051.2	3051.2	41.8	-199.14	9.78
mysql	40	2	30	3275	101525	95396	6369.6	0	1584873	6.4	8843.6	8843.6	43.6	-36.48	12.22
mysql	60	2	20	7315	226765	217516	9757.75	0	2215976	7	17990	17990	48.75	-114.17	3.47
apache	13	3	100	2266	40788	34530	1524.5	0	1011290	7	915	915	56.5	-117.86	14.26
apache	17	3	76	5410	97380	85842	2502.5	0	696652.5	7	1634	1634	66.25	-172.33	15.71
apache	22	3	59	12280	221040	200652	4378	0	1104850	7.25	5713.75	5713.75	74.25	3.81	13.66
apache	26	3	50	20752	373536	344268	4835.75	0	1166920	7.25	11218.5	11218.5	80.25	-4.46	11.81
apache	33	3	39	43586	784548	735890	12946	0	1424028	8	29807	29807	89	-73.1	13.59
apache	44	3	30	105868	1905624	1816596	22887	0	1416228	8	76894	76894	101	-161.01	11.01
apache	65	3	20	349314	6287652	6087954	43606	0	1551500	8	324326	324326	116	-266.22	11.45
mysql	12	3	100	2176	67456	49557	4735.5	0	1133572	7.25	936.5	936.5	110	8.19	15.38
mysql	15	3	80	4338	134478	105077	7622	0	1260647	7.5	2248	2248	117	43.23	18.98
mysql	20	3	60	10448	323888	269077	15198.5	0	1374265	8	6364.5	6364.5	129.25	67.56	14.97
mysql	24	3	50	18168	563208	482453	17968	0	1236984	8	11238.75	11238.75	142	71.83	8.78
mysql	30	3	40	35668	1105708	976677	54208	0	1496891	8	32492	32492	152	63.99	8.8
mysql	40	3	30	84888	2631528	2397077	83809	0	1431516	8.33	108570.7	108571.3	165.67	60.65	8.47
mysql	60	3	20	287328	8907168	8368277	187225	0	2030246	8	403490	403490	187	73.4	4.1

Table A.7: Statistics and improvements for RISxCRI combination

sut	opt	t	cop	# of ϕ_t	# of λ_t	size of Λ_t	Init.Miss	T_{init}	IT_{sa}	IT_{bs}	T_{anneal}	T_{total}	size	T_{imp}	N_{imp}
apache	13	2	100	311	5598	5013	309.8	0	543437.2	5.4	188.2	188.2	32	-213.67	-25.49
apache	17	2	76	543	9774	8989	467.6	0	608635.4	5.6	361.8	361.8	37.6	-201.5	-37.23
apache	22	2	59	923	16614	15579	661.4	0	527393	5.6	497.4	497.4	38.4	60.52	-33.94
apache	26	2	50	1299	23382	22147	686.8	0	723120.4	5.8	865.4	865.4	44.8	73.78	-49.33
apache	33	2	39	2111	37998	36413	957.8	0	724444.6	5.8	1497.6	1497.6	47.8	69.19	-47.08
apache	44	2	30	3783	68094	65959	1096	0	814249.6	6	3016.2	3016.2	59.4	52.58	-74.71
apache	65	2	20	8319	149742	146557	1442.4	0	571141.2	6	3959.6	3959.6	57	50.38	-52
mysql	12	2	100	307	9517	7756	1243.2	0	1976324	7	765.8	765.8	41	-1176.33	2.84
mysql	15	2	80	475	14725	12496	1435	0	2088811	7	1158.4	1158.4	46	-865.33	-6.73
mysql	20	2	60	835	25885	22876	1951.6	0	2175147	7	1954.2	1954.2	52	33.53	-17.3
mysql	24	2	50	1195	37045	33412	2945.8	0	2167684	7	2682.8	2682.8	55	46.13	-19.57
mysql	30	2	40	1855	57505	52936	4895.8	0	2203473	7	4125.4	4125.4	61	-304.45	-31.66
mysql	40	2	30	3275	101525	95396	6369.6	0	2216421	7	7267.6	7267.6	68	-12.15	-36.9
mysql	60	2	20	7315	226765	217516	9449.364	0	2107645	7	14767	14767	75	-75.8	-48.51
apache	13	3	100	2266	40788	34530	1850.4	0	620690.2	6.8	1288.8	1288.8	82.8	-206.86	-25.64
apache	17	3	76	5410	97380	85842	2332.8	0	482824.8	6.8	1681.2	1681.2	83	-180.2	-5.6
apache	22	3	59	12280	221040	200652	4441.2	0	433434.6	7	2997.4	2997.4	89.6	49.54	-4.19
apache	26	3	50	20752	373536	344268	6422	0	644166.8	7.4	8234.8	8234.8	92.8	23.33	-1.98
apache	33	3	39	43586	784548	735890	10336.5	0	711966	7.5	19446	19446	99.5	-12.93	3.4
apache	44	3	30	105868	1905624	1816596	21054	0	434419	7	26839.5	26839.5	109	8.9	3.96
apache	65	3	20	349314	6287652	6087954	36098	0	418820	7	88061	88061	130.5	0.56	0.38
mysql	12	3	100	2176	67456	49557	4991	0	1547523	8	2889.8	2889.8	111	-183.31	14.62
mysql	15	3	80	4338	134478	105077	7945.2	0	1562646	8	5034.6	5034.6	125	-27.14	13.43
mysql	20	3	60	10448	323888	269077	14038.2	0	1584334	8	11333.8	11333.8	141	42.23	7.24
mysql	24	3	50	18168	563208	482453	21817.6	0	1802850	8.8	21916	21916	178.6	45.07	-14.73
mysql	30	3	40	35668	1105708	976677	35060	0	1900220	9	48133	48133	167	46.66	-0.2
mysql	40	3	30	84888	2631528	2397077	77310.5	0	1878724	9	106773	106773	185	61.31	-2.21
mysql	60	3	20	287328	8907168	8368277	170329	0	1874380	9	400229.5	400229.5	207	73.61	-6.15

Table A.8: Statistics and improvements for RISxCRT combination

sut	opt	t	cop	# of ϕ_t	# of λ_t	size of Λ_t	Init.Miss	T_{init}	IT_{sa}	IT_{bs}	T_{anneal}	T_{total}	size	T_{imp}	N_{imp}
apache	13	2	100	311	5598	5013	176.2	0	408641.6	5.8	106.2	106.2	22.8	-77	10.59
apache	17	2	76	543	9774	8989	267.4	0	295106	5.2	137.2	137.2	24.8	-14.33	9.49
apache	22	2	59	923	16614	15579	332.4	0	236684	5.4	180.4	180.4	26	85.68	9.31
apache	26	2	50	1299	23382	22147	446	0	358450.2	5.8	404.6	404.6	28	87.74	6.67
apache	33	2	39	2111	37998	36413	556.2	0	257570	5.8	425	425	28.4	91.26	12.62
apache	44	2	30	3783	68094	65959	798	0	221062.4	6	657.8	657.8	30.6	89.66	10
apache	65	2	20	8319	149742	146557	1396.4	0	158008.6	6	1217.2	1217.4	33.4	84.74	10.93
mysql	12	2	100	307	9517	7756	910.6	4	397127.8	6	135	139.6	37.4	-132.67	11.37
mysql	15	2	80	475	14725	12496	1389.6	3	274171.4	6	164.6	168.4	38.8	-40.33	9.98
mysql	20	2	60	835	25885	22876	2272.2	0	276334.6	6	239.2	239.2	38.4	91.86	13.38
mysql	24	2	50	1195	37045	33412	1873	0	272632.6	6	331.6	331.6	40.4	93.34	12.17
mysql	30	2	40	1855	57505	52936	4052.4	0	210000	6.2	442.2	442.4	41.2	56.63	11.07
mysql	40	2	30	3275	101525	95396	3508.8	0	382812.2	6.6	1095	1095.2	43.6	83.1	12.22
mysql	60	2	20	7315	226765	217516	8559.333	0	289044.8	6.47	1954.6	1954.73	46.2	76.73	8.51
apache	13	3	100	2266	40788	34530	885.6	24	441600.4	6.2	621	645.4	62.8	-53.67	4.7
apache	17	3	76	5410	97380	85842	1845.4	70.8	358256.4	6.8	1075.4	1146.6	75.4	-91.1	4.07
apache	22	3	59	12280	221040	200652	3971.6	219	386753	7	2915.8	3135.6	80.4	47.21	6.51
apache	26	3	50	20752	373536	344268	8185.6	742.8	562557.6	7.4	5925.6	6668.6	90	37.91	1.1
apache	33	3	39	43586	784548	735890	6110.5	1533	645304.5	8	16416.5	17950	97	-4.24	5.83
apache	44	3	30	105868	1905624	1816596	15617	3865.5	429540.5	7.5	27945	31811.5	102.5	-7.98	9.69
apache	65	3	20	349314	6287652	6087954	32054	9892	530802	8	93665	103557.3	118	-16.93	9.92
mysql	12	3	100	2176	67456	49557	4868.6	94	398857.8	7.4	634.8	729.4	115.2	28.49	11.38
mysql	15	3	80	4338	134478	105077	6871.4	149	402752.4	7.6	1176.4	1326.2	125.6	66.51	13.02
mysql	20	3	60	10448	323888	269077	13602.6	309.4	595388.2	8	3697.4	4007	136.8	79.58	10
mysql	24	3	50	18168	563208	482453	23588.2	532.8	554527.8	8	6297	6830	144.6	82.88	7.11
mysql	30	3	40	35668	1105708	976677	35099	1077	516745	8	12605	13682	157	84.84	5.8
mysql	40	3	30	84888	2631528	2397077	53125	2513	353213	8	17436	19949	166	92.77	8.29
mysql	60	3	20	287328	8907168	8368277	156986	9092.5	412228	8	80433	89526.5	185.5	94.1	4.87

Table A.9: Statistics and improvements for TISxAVO combination

sut	opt	t	cop	# of ϕ_t	# of λ_t	size of Λ_t	Init.Miss	T_{init}	IT_{sa}	IT_{bs}	T_{anneal}	T_{total}	size	T_{imp}	N_{imp}
apache	13	2	100	311	5598	5013	176.2	0	294545.6	5.2	91	91	27.4	-51.67	-7.45
apache	17	2	76	543	9774	8989	267.4	0	381383.6	5.8	192.6	192.6	28.8	-60.5	-5.11
apache	22	2	59	923	16614	15579	332.4	0	297769.8	5.8	243.2	243.2	31.2	80.7	-8.82
apache	26	2	50	1299	23382	22147	446	0	274381.4	6	314.6	314.8	31.8	90.46	-6
apache	33	2	39	2111	37998	36413	556.2	0	347215.4	6	549.2	549.4	33.4	88.7	-2.77
apache	44	2	30	3783	68094	65959	798	0	297958.4	6	809.8	809.8	35.8	87.27	-5.29
apache	65	2	20	8319	149742	146557	1396.4	0	205586.8	6	1455	1455.2	46.6	81.76	-24.27
mysql	12	2	100	307	9517	7756	910.6	4	639474.6	6.8	451.4	456	46.8	-660	-10.9
mysql	15	2	80	475	14725	12496	1389.6	3	525047	6.2	456.4	459.8	70	-283.17	-62.41
mysql	20	2	60	835	25885	22876	2272.2	0	482448	6.4	544.2	544.6	82.6	81.48	-86.33
mysql	24	2	50	1195	37045	33412	1873	0	506223.2	6.8	825.4	825.6	79.4	83.42	-72.61
mysql	30	2	40	1855	57505	52936	4052.4	0	418506.6	6.4	891.2	891.2	84.6	12.63	-82.6
mysql	40	2	30	3275	101525	95396	3508.8	0	392861	6.4	1432	1432	73.8	77.9	-48.58
mysql	60	2	20	7315	226765	217516	8321.6	0	289403.2	6	2101.4	2102	86.6	74.98	-71.49
apache	13	3	100	2266	40788	34530	885.6	24	231584.8	6.4	338.6	363	66.6	13.57	-1.06
apache	17	3	76	5410	97380	85842	1845.4	70.8	388557.6	6.8	1220.6	1292.2	82.2	-115.37	-4.58
apache	22	3	59	12280	221040	200652	3971.6	219	255760.8	7	1886.6	2106	93.6	64.55	-8.84
apache	26	3	50	20752	373536	344268	8185.6	742.8	321731	7	3690.8	4433.8	95.8	58.72	-5.27
apache	33	3	39	43586	784548	735890	6110.5	1533	615754.5	8	16731.5	18265.5	99	-6.07	3.88
apache	44	3	30	105868	1905624	1816596	15617	3865.5	326480.5	7	23414	27280.5	126	7.4	-11.01
apache	65	3	20	349314	6287652	6087954	37179	11252.5	332146	7.5	74139.5	85392.5	141.5	3.58	-8.02
mysql	12	3	100	2176	67456	49557	4868.6	94	743973.4	7.8	1507	1601.8	147	-57.04	-13.08
mysql	15	3	80	4338	134478	105077	6871.4	149	617443.6	7.8	2048.8	2198.4	211.6	44.48	-46.54
mysql	20	3	60	10448	323888	269077	13602.6	309.4	486418.4	7.8	3671.8	3981.6	222.4	79.71	-46.32
mysql	24	3	50	18168	563208	482453	23588.2	532.8	404433.8	8	5207.6	5740.8	232.4	85.61	-49.29
mysql	30	3	40	35668	1105708	976677	35099	1077	398004	8	10401.5	11478.5	226.5	87.28	-35.9
mysql	40	3	30	84888	2631528	2397077	53125	2513	412257	8	22977	25491	230	90.76	-27.07
mysql	60	3	20	287328	8907168	8368277	156986	9092.5	346807	8	70111	79204.5	308	94.78	-57.95

Table A.10: Statistics and improvements for TISxCMP combination

sut	opt	t	cop	# of ϕ_t	# of λ_t	size of Λ_t	Init.Miss	T_{init}	IT_{sa}	IT_{bs}	T_{anneal}	T_{total}	size	T_{imp}	N_{imp}
apache	13	2	100	311	5598	5013	176.2	0	512865	5	247.2	247.2	21	-312	17.65
apache	17	2	76	543	9774	8989	267.4	0	783367	6	484.8	484.8	23	-304	16.06
apache	22	2	59	923	16614	15579	332.4	0	889921.4	5.8	954	954	24.8	24.29	13.5
apache	26	2	50	1299	23382	22147	446	0	848884.6	6	1339.6	1339.6	25.4	59.41	15.33
apache	33	2	39	2111	37998	36413	556.2	0	728444.4	5.2	1729.8	1729.8	27.2	64.41	16.31
apache	44	2	30	3783	68094	65959	798	0	1191289	6	5198.4	5198.4	30	18.26	11.76
apache	65	2	20	8319	149742	146557	1396.4	0	1439833	6	16934.6	16935	34.4	-112.22	8.27
mysql	12	2	100	307	9517	7756	910.6	4	817114	6	394.8	399.2	35.2	-565.33	16.59
mysql	15	2	80	475	14725	12496	1389.6	3	1160432	6.8	816.2	819.4	36	-582.83	16.47
mysql	20	2	60	835	25885	22876	2272.2	0	919037.6	6.2	1182.4	1183	38	59.76	14.28
mysql	24	2	50	1195	37045	33412	1873	0	1102777	6.4	2150.8	2150.8	39.6	56.81	13.91
mysql	30	2	40	1855	57505	52936	4052.4	0	753257	6	2590.6	2590.6	41.6	-153.98	10.21
mysql	40	2	30	3275	101525	95396	3508.8	0	1779191	6.6	10047.8	10047.8	44.4	-55.06	10.61
mysql	60	2	20	7315	226765	217516	9507	0	2383119	7	18450	18450	48.5	-119.64	3.96
apache	13	3	100	2266	40788	34530	937.5	7.25	909386.5	6.75	823.25	830.75	56.75	-97.8	13.88
apache	17	3	76	5410	97380	85842	1391	32.25	941710.5	7	2109.25	2142	65.5	-257	16.67
apache	22	3	59	12280	221040	200652	4479	119.5	1258394	7.75	6699.5	6819.25	75	-14.8	12.79
apache	26	3	50	20752	373536	344268	5523.75	398.25	909019.2	7	8431.5	8830.25	80.75	17.78	11.26
apache	33	3	39	43586	784548	735890	4446	883	853362	7	16352	17236	91	-0.09	11.65
apache	44	3	30	105868	1905624	1816596	15678	2132	991824	8	62720	64853	99	-120.14	12.78
apache	65	3	20	349314	6287652	6087954	45188	7726	1332155	7	274403	282130	124	-218.57	5.34
mysql	12	3	100	2176	67456	49557	3480.5	33	1387036	7.5	1155.75	1189	109	-16.57	16.15
mysql	15	3	80	4338	134478	105077	6269.75	64	1227830	7.5	2165.25	2229.75	117.75	43.69	18.46
mysql	20	3	60	10448	323888	269077	15638	155.25	1399981	8	6903.25	7059.25	129	64.02	15.13
mysql	24	3	50	18168	563208	482453	19178	283.25	1240620	8	11792.25	12076	138.25	69.73	11.19
mysql	30	3	40	35668	1105708	976677	26581	607	2024322	9	43190	43797	149	51.47	10.6
mysql	40	3	30	84888	2631528	2397077	62933.5	2142.5	1686342	8.5	129266.5	131409.5	168.5	52.38	6.91
mysql	60	3	20	287328	8907168	8368277	202616	6106	1942255	8	353434	359540	190	76.29	2.56

Table A.11: Statistics and improvements for TISxCRI combination

sut	opt	t	cop	# of ϕ_t	# of λ_t	size of Λ_t	Init.Miss	T_{init}	IT_{sa}	IT_{bs}	T_{anneal}	T_{total}	size	T_{imp}	N_{imp}
apache	13	2	100	311	5598	5013	176.2	0	450799.4	5.2	164.2	164.2	32.4	-173.67	-27.06
apache	17	2	76	543	9774	8989	267.4	0	483387.8	5.4	278.2	278.2	35.4	-131.83	-29.2
apache	22	2	59	923	16614	15579	332.4	0	647497.6	5.4	539	539	44.6	57.22	-55.56
apache	26	2	50	1299	23382	22147	446	0	458105.6	5.4	544.8	545	40.8	83.48	-36
apache	33	2	39	2111	37998	36413	556.2	0	763819.2	5.6	1634	1634.2	51.2	66.37	-57.54
apache	44	2	30	3783	68094	65959	798	0	894580.8	6	3082.4	3082.4	61.8	51.53	-81.76
apache	65	2	20	8319	149742	146557	1396.4	0	380039.4	6	3423.2	3423.4	57.2	57.1	-52.53
mysql	12	2	100	307	9517	7756	910.6	4	1938987	7	752.4	757	41	-1161.67	2.84
mysql	15	2	80	475	14725	12496	1389.6	3	2111891	7	1165.6	1169	46	-874.17	-6.73
mysql	20	2	60	835	25885	22876	2272.2	0	2197869	7	1984.6	1984.6	52	32.5	-17.3
mysql	24	2	50	1195	37045	33412	1873	0	2176270	7	2677	2677	55	46.24	-19.57
mysql	30	2	40	1855	57505	52936	4052.4	0	2180095	7	4090	4090	61	-300.98	-31.66
mysql	40	2	30	3275	101525	95396	3508.8	0	2221193	7	7253.2	7253.2	68	-11.93	-36.9
mysql	60	2	20	7315	226765	217516	8321.6	0	2109837	7	14713.8	14713.8	75	-75.16	-48.51
apache	13	3	100	2266	40788	34530	885.6	24	447485	6	809.4	834	78.8	-98.57	-19.58
apache	17	3	76	5410	97380	85842	1845.4	70.8	452617	6.6	1703.6	1775	81	-195.83	-3.05
apache	22	3	59	12280	221040	200652	3971.6	219	461081.8	7	3375.8	3595.4	90.8	39.47	-5.58
apache	26	3	50	20752	373536	344268	8185.6	742.8	426304.2	7	5583.8	6326.6	94.8	41.09	-4.18
apache	33	3	39	43586	784548	735890	6110.5	1533	660822	7.5	18511	20044.5	98	-16.4	4.85
apache	44	3	30	105868	1905624	1816596	15617	3865.5	467834.5	7	32159.5	36026	109	-22.29	3.96
apache	65	3	20	349314	6287652	6087954	37179	11252.5	300294.5	7	62948	74201	131	16.21	0
mysql	12	3	100	2176	67456	49557	4868.6	94	1519969	8	2847.2	2941.6	111	-188.39	14.62
mysql	15	3	80	4338	134478	105077	6871.4	149	1550189	8	4999.8	5149.2	125	-30.03	13.43
mysql	20	3	60	10448	323888	269077	13602.6	309.4	1594780	8	11504.4	11814.2	141	39.78	7.24
mysql	24	3	50	18168	563208	482453	23588.2	532.8	1853397	9	22702	23235.6	153	41.77	1.72
mysql	30	3	40	35668	1105708	976677	35099	1077	1894681	9	48310.5	49387.5	167	45.27	-0.2
mysql	40	3	30	84888	2631528	2397077	53125	2513	1525607	9	89746	92260	303	66.57	-67.4
mysql	60	3	20	287328	8907168	8368277	176223	9186	585590	8	110438	119625	341	92.11	-74.87

Table A.12: Statistics and improvements for TISxCRT combination

sut	opt	t	cop	# of ϕ_t	# of λ_t	size of Λ_t	Init.Miss	T_{init}	IT_{sa}	IT_{bs}	T_{anneal}	T_{total}	size	T_{imp}	N_{imp}
apache	13	3	100	2266	40788	34530	549.8	183.4	525810.2	7	830	1014	61.2	-141.43	7.13
apache	17	3	76	5410	97380	85842	896.8	245.8	479027.6	7	1552.8	1799.2	72.6	-199.87	7.63
apache	22	3	59	12280	221040	200652	1450.2	219.4	399883.4	7	2837.2	3057.4	82	48.53	4.65
apache	26	3	50	20752	373536	344268	2210.4	569.4	486934	7.2	5165	5735	87.4	46.6	3.96
apache	33	3	39	43586	784548	735890	2563	891	556659.5	7.5	12913.5	13805.5	96	19.83	6.8
apache	44	3	30	105868	1905624	1816596	5321.5	1181	313801.5	7.5	22526.5	23708	111.5	19.52	1.76
apache	65	3	20	349314	6287652	6087954	10570	1896.5	564367.5	8	118089.5	119986	122	-35.49	6.87
mysql	12	3	100	2176	67456	49557	1717.4	284	459770.2	8	732.6	1017.4	117	0.25	10
mysql	15	3	80	4338	134478	105077	2910.8	333.6	482762.6	7.8	1310.6	1645	124.6	58.46	13.71
mysql	20	3	60	10448	323888	269077	5314.2	380	476870.8	7.8	3245.2	3625.8	136	81.52	10.53
mysql	24	3	50	18168	563208	482453	8574	553.8	514217.8	8	5575.6	6130.2	142.8	84.64	8.27
mysql	30	3	40	35668	1105708	976677	13004.5	1088.5	467140.5	8	12325	13414	152	85.14	8.8
mysql	40	3	30	84888	2631528	2397077	18772	509	485867	8	14551	15061	158	94.54	12.71
mysql	60	3	20	287328	8907168	8368277	53200	1432	396138	8	46012	47445	185	96.87	5.13

Table A.13: Statistics and improvements for TCISxAVO combination

sut	opt	t	cop	# of ϕ_t	# of λ_t	size of Λ_t	Init.Miss	T_{init}	IT_{sa}	IT_{bs}	T_{anneal}	T_{total}	size	T_{imp}	N_{imp}
apache	13	3	100	2266	40788	34530	588.8	183.4	336887	6.6	462.8	646.2	59	-53.86	10.47
apache	17	3	76	5410	97380	85842	808.6	245.8	319970	6.8	1105	1351.2	83	-125.2	-5.6
apache	22	3	59	12280	221040	200652	1500.2	219.4	380330.8	7	2937.8	3157.6	91.2	46.84	-6.05
apache	26	3	50	20752	373536	344268	2127	569.4	459558.4	7.4	5424.8	5994.8	96	44.18	-5.49
apache	33	3	39	43586	784548	735890	3971	891	471804.5	7.5	12976	13868	103.5	19.47	-0.49
apache	44	3	30	105868	1905624	1816596	4788	1181	454823.5	7.5	30597.5	31779	120	-7.87	-5.73
apache	65	3	20	349314	6287652	6087954	9685	1896.5	247520.5	7.5	48733	50630	141.5	42.83	-8.02
mysql	12	3	100	2176	67456	49557	1606	332	323037	7	643	975	181	4.41	-39.23
mysql	15	3	80	4338	134478	105077	2760.75	306.75	631172.5	8	2084	2391.5	215.5	39.61	-49.24
mysql	20	3	60	10448	323888	269077	5702.4	380	434135.4	7.6	3229.6	3610.4	223	81.6	-46.71
mysql	24	3	50	18168	563208	482453	8145.8	553.8	410978.6	8	5185.4	5740	239.6	85.61	-53.92
mysql	30	3	40	35668	1105708	976677	10247	1088.5	452617.5	8	12544	13633.5	233.5	84.89	-40.1
mysql	40	3	30	84888	2631528	2397077	14385	509	319568	8	11341	11851	265	95.71	-46.41
mysql	60	3	20	287328	8907168	8368277	37646	1432	289072	8	37259	38692	297	97.45	-52.31

Table A.14: Statistics and improvements for TCISxCMP combination

sut	opt	t	cop	# of ϕ_t	# of λ_t	size of Λ_t	Init.Miss	T_{init}	IT_{sa}	IT_{bs}	T_{anneal}	T_{total}	size	T_{imp}	N_{imp}
apache	13	3	100	4316	40788	34530	620	244	935532	7	1155	1355	65	-222.62	1.37
apache	17	3	76	5410	97380	85842	891	213	987189	7	1379	4568	85	-661.33	-8.14
apache	22	3	59	12280	221040	200652	1323	163	801941	7	6255	6418	80	-8.05	6.98
apache	26	3	50	20752	373536	344268	1654	537	912094	7	11457	11995	89	-11.69	2.2
apache	33	3	39	43586	784548	735890	4620	954	185313	7	16985	19740	97	-14.63	5.83
apache	44	3	30	105868	1905624	1816596	5191	1131	1138749	8	58660	59811	113	-103.02	0.44
apache	65	3	20	349314	6287652	6087954	17402	1350	1028290	7	230422	231923	123	-161.88	6.11
mysql	12	3	100	2176	67456	49557	1606	452	1273037	8	1143	1595	118	-56.37	9.23
mysql	15	3	80	4338	134478	105077	2739	429	1198219	8	3390	3820	124	3.54	14.13
mysql	20	3	60	10448	323888	269077	5770	382	1113313	8	3574	5957	131	69.64	13.82
mysql	24	3	50	18168	563208	482453	6388	532	1502242	8	7135	7668	155	80.78	0.43
mysql	30	3	40	35668	1105708	976677	9687	1052	919072	8	15143	17276	173	80.86	-3.8
mysql	40	3	30	84888	2631528	2397077	14385	509	1114567	8	21341	21851	165	92.08	8.84
mysql	60	3	20	287328	8907168	8368277	37646	1432	1489072	8	47259	48692	201	96.79	-3.08

Table A.15: Statistics and improvements for TCISxCRI combination

sut	opt	t	cop	# of ϕ_t	# of λ_t	size of Λ_t	Init.Miss	T_{init}	IT_{sa}	IT_{bs}	T_{anneal}	T_{total}	size	T_{imp}	N_{imp}
apache	13	3	100	2266	40788	34530	636.6	183.4	481224.6	6.4	829	1012.8	79.2	-141.14	-20.18
apache	17	3	76	5410	97380	85842	892.6	245.8	552116.2	6.8	1768.4	2014.4	81.4	-235.73	-3.56
apache	22	3	59	12280	221040	200652	1458.6	219.4	512567.6	7	3785.6	4005.6	92	32.57	-6.98
apache	26	3	50	20752	373536	344268	2450.8	569.4	491292.2	7.4	6844.8	7415	94.4	30.96	-3.74
apache	33	3	39	43586	784548	735890	3302	928	857159	8	22503	23432	105	-36.07	-1.94
apache	44	3	30	105868	1905624	1816596	6042	1181	448046	7.5	33992.5	35174	114.5	-19.4	-0.88
apache	65	3	20	349314	6287652	6087954	13934.5	1896.5	412340.5	7	90301	92198	134.5	-4.11	-2.67
mysql	12	3	100	2176	67456	49557	1606	252	1133211	8	1941	2193	138	-115	-6.15
mysql	15	3	80	4338	134478	105077	2578	283	914175	8	2417	2701	156	31.79	-8.03
mysql	20	3	60	10448	323888	269077	6455	319	1393041	8	6018	6338	175	67.7	-15.13
mysql	24	3	50	18168	563208	482453	8024	643	1268640	8	6517	7161	209	82.05	-34.26
mysql	30	3	40	35668	1105708	976677	10807	1065	483153	8	15945	17011	204	81.15	-22.4
mysql	40	3	30	84888	2631528	2397077	14385	509	919368	8	16341	17851	235	93.53	-29.83
mysql	60	3	20	287328	8907168	8368277	28258	1432	1398541	9	182430	183863	296	87.88	-51.79

Table A.16: Statistics and improvements for TCISxCRT combination

BIBLIOGRAPHY

[1] Advanced Combinatorial Testing System (ACTS), 2012. `http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html`.

[2] R. Brownlie, J. Prowse, and M. S. Phadke. Robust testing of AT&T PMX/StarMAIL using OATS. AT&T Technical Journal, 71(3):41–7, 1992.

[3] R. C. Bryce and C. J. Colbourn. Constructing interaction test suites with greedy algorithms. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 440–443, New York, NY, USA, 2005. ACM.

[4] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, 48(10):960 – 970, 2006. Advances in Model-based Testing.

[5] R. C. Bryce and C. J. Colbourn. The density algorithm for pairwise interaction testing: Research articles. *Softw. Test. Verif. Reliab.*, 17:159–182, September 2007.

[6] R. C. Bryce and C. J. Colbourn. One-test-at-a-time heuristic search for interaction test suites. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1082–1089, New York, NY, USA, 2007. ACM.

[7] R. C. Bryce and C. J. Colbourn. A density-based greedy algorithm for higher strength covering arrays. *Softw. Test. Verif. Reliab.*, 19:37–53, March 2009.

[8] V. Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985.

[9] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–44, 1997.

[10] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling. Augmenting simulated annealing to build interaction test suites. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 394–, Washington, DC, USA, 2003. IEEE Computer Society.

[11] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 129–139, New York, NY, USA, 2007. ACM.

[12] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 38–48, Washington, DC, USA, 2003. IEEE Computer Society.

[13] J. Czerwonka. Pairwise testing in the real world: Practical extensions to test-case scenarios. In *Proc. of the 24th Pacific Northwest Software Quality Conference*, pages 285–294, 2006.

[14] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proc. of the Int'l Conf. on Software Engineering*, pages 285–294, 1999.

[15] E. Dumlu, C. Yilmaz, M. B. Cohen, and A. Porter. Feedback driven adaptive combinatorial testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 243–253, New York, NY, USA, 2011. ACM.

[16] S. Fouché, M. B. Cohen, and A. Porter. Towards incremental adaptive covering arrays. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers*, ESEC-FSE companion '07, pages 557–560, New York, NY, USA, 2007. ACM.

[17] S. Ghazi and M. Ahmed. Pair-wise test coverage using genetic algorithms. In *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, volume 2, pages 1420 – 1424 Vol.2, dec. 2003.

[18] A. Hartman. Software and hardware testing using combinatorial covering suites. In M. C. Golumbic and I. B.-A. Hartman, editors, *Graph Theory, Combinatorics and Algorithms*, volume 34 of *Operations Research/Computer Science Interfaces Series*, pages 237–266. Springer US, 2005.

[19] S. Kirkpatrick, M. Vecchi, et al. Optimization by simmulated annealing. *science*, 220(4598):671–680, 1983.

[20] N. Kobayashi. *Design and evaluation of automatic test generation strategies for functional testing of software*. Osaka University, Osaka, Japan, 2002.

[21] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. Ipog-ipog-d: efficient test generation for multi-way combinatorial testing. *Softw. Test. Verif. Reliab.*, 18:125–148, September 2008.

[22] G. Mats, O. Jeff, and M. Jonas. Handling constraints in the input space when using combination strategies for software testing. Technical Report HS- IKI -TR-06-001, University of Skvde, School of Humanities and Informatics, 2006.

[23] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43:11:1–11:29, February 2011.

[24] P. J. Schroeder, P. Bolaki, and V. Gopu. Comparing the fault detection effectiveness of n-way and random test suites. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 49–59, Washington, DC, USA, 2004. IEEE Computer Society.

[25] T. Shiba, T. Tsuchiya, and T. Kikuno. Using artificial life techniques to generate test cases for combinatorial testing. In *Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '04, pages 72–77, Washington, DC, USA, 2004. IEEE Computer Society.

[26] J. Stardom. *Metaheuristics and the Search for Covering and Packing Arrays [microform]*. Canadian theses. Thesis (M.Sc.)–Simon Fraser University, 2001.

[27] K.-C. Tai and Y. Lei. A test generation strategy for pairwise testing. *Software Engineering, IEEE Transactions on*, 28(1):109 –111, jan 2002.

[28] J. Torres-Jimenez and E. Rodriguez-Tello. New bounds for binary covering arrays using simulated annealing. *Information Sciences*, 185(1):137–152, 2012.

[29] Y.-W. Tung and W. Aldiwan. Automating test case generation for the new generation mission software system. In *Aerospace Conference Proceedings, 2000 IEEE*, volume 1, pages 431 –437 vol.1, 2000.

[30] A. W. Williams. Determination of test configurations for pair-wise interaction coverage. In *Proceedings of the IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems: Tools and Techniques*, TestCom '00, pages 59–74, Deventer, The Netherlands, The Netherlands, 2000. Kluwer, B.V.

[31] A. W. Williams and R. L. Probert. Formulation of the interaction test coverage problem as an integer program. In *Proceedings of the IFIP 14th International Conference on Testing Communicating Systems XIV*, TestCom '02, pages 283–, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[32] C. Yilmaz. Test case-aware combinatorial interaction testing. *Software Eng., IEEE Trans. on*, PP(99):1, 2012.

[33] C. Yilmaz, S. Fouche, M. Cohen, A. A. Porter, G. Demiroz, and U. Koc. Moving forward with combinatorial interaction testing. *Computer*, 99(PrePrints):1, 2013.