Learning Combinatorial Interaction Test Generation Strategies using Hyperheuristic Search

Yue Jia
University College London,
London, WC1E 6BT, UK
Email: yue.jia@ucl.ac.uk

Myra B. Cohen
University of Nebraska-Lincoln
Lincoln, NE 68588-0115, USA
Email: myra@cse.unl.edu

Mark Harman, Justyna Petke
University College London,
London, WC1E 6BT, UK
Email: {mark.harman, j.petke}@ucl.ac.uk

***Abstract*—The surge of search based software engineering research has been hampered by the need to develop customized search algorithms for different classes of the same problem. For instance, two decades of bespoke Combinatorial Interaction Testing (CIT) algorithm development, our exemplar problem, has left software engineers with a bewildering choice of CIT techniques, each specialized for a particular task. This paper proposes the use of a single hyperheuristic algorithm that learns search strategies across a broad range of problem instances, providing a single *generalist* approach. We have developed a Hyperheuristic algorithm for CIT, and report experiments that show that our algorithm competes with known best solutions across constrained and unconstrained problems: For all 26 real-world subjects, it equals or outperforms the best result previously reported in the literature. We also present evidence that our algorithm's strong generic performance results from its unsupervised learning. Hyperheuristic search is thus a promising way to relocate CIT design intelligence from human to machine.**

I. Introduction

Over the past decade, research in Search Based Software Engineering has been growing at a rapid pace [1]. One limitation of much of this work is that search algorithms must be customized for specific instances or equivalence classes of the problem. For example, the existence of constraints in the data set to be optimized often requires a different set of search operators [2]. Hyperheuristic search (HS) is a new class of optimisation algorithms that may provide a substantial improvement to this bespoke approach [1]. HS algorithms use dynamic adaptive optimisation to learn strategies without active supervision [3], [4]. Hyperheuristics have been successfully applied to many operational research problems outside of software engineering [4]. However, though they have been advocated as a possible solution to dynamic adaptive optimisation for software engineering [3], they have not, hitherto, been applied to any software engineering problem [5]–[7].

In this paper, we examine the feasibility of using HS in search based software engineering. To achieve our goal we select a mature and well studied search problem as our exemplar, Combinatorial Interaction Testing (CIT). CIT aims to generate samples that cover all possible value combinations between any set of t parameters, where t is fixed (usually between 2 and 6). Software product lines [8], operating systems, development environments and many other systems are typically governed by large configuration, parameter and feature spaces for which CIT has proved useful [9].

Over two decades of research has gone into the development of CIT test generation techniques, each of which is tailored and tuned to a specific problem [10]–[16]. For example, some CIT algorithms have been tuned and evaluated only on unconstrained problems [10], [15], [17], [18], while others have been specifically tuned for constrained interaction testing [2], [19], which prohibits certain configurations. Still other CIT approaches target specific problem structures, such as parameter spaces with few available parameter value choices [13], [20], or are tuned to work on a particular set of real-world problems [21]. Colbourn maintains a website of results from many different sources and techniques [22], both published and unpublished in the CIT literature, while the `pairwise.org` web portal contains almost 40 tools for pairwise instances of CIT alone. There is also a framework that collates the many different CIT algorithms [23], but none of these frameworks or portals can help the tester to choose *which algorithm* to apply to each CIT problem instance.

Even when attention is restricted to a single kind of CIT algorithm, such as simulated annealing, there remain further choices to be made: For general unconstrained problems, the single-mutation-at-a-time variant yields a smallest test suites [10], but for binary-valued problems a different simulated annealing variant would be recommended [24], while still another variant would be preferred for highly constrained problems [2], [25]. The tester is therefore presented a bewildering choice between different techniques and implementations from which to choose, each of which has its own special properties. It is unreasonable to expect practicing software testers to perform their own experiments to decide on the best CIT algorithm choice for each and every testing problem. CIT users in the research community also find the choices overwhelming. For example Lopez-Herrejon et al. ask "With all these pairwise testing approaches available the question now is: how do they compare?" [26]. We cannot expect each testing organisation to hire an algorithm designer to build bespoke CIT testing implementations for each testing scenario they face. We need a more general CIT approach.

To evaluate the feasibility of using HA as a generalist approach we introduce a simulated annealing hyperheuristic search based algorithm for CIT. Our hyperheuristic algorithm

learns the best CIT strategy to apply dynamically, as it is executed and the chosen strategy may change over time. This single algorithm can be applied to a wide range of CIT problem instances, regardless of their structure and characteristics.

For our new algorithm to be acceptable as a generic solution to the CIT problem, we need to demonstrate that it is effective and efficient across a wide range of CIT problem instances, when compared to other possible algorithm choices. To assess the effectiveness of CIT solutions we use test suite size, which Garvin et al. [2] show to have the greatest impact on overall test efficacy. To assess efficiency we report computational time (as is standard in CIT experiments), but we also deploy the algorithms in the cloud to provide a supplementary assessment of monetary cost (as has been done in other studies [27]).

We compare our hyperheuristic algorithm, not only against results from state-of-the-art search CIT techniques, but also against the best known results in the literature, garnered over 20 years of analysis of CIT. This is a particularly challenging 'quality comparison' for any algorithm, because some of these best known results are the product of many years of careful analysis by mathematicians, not machines.

We show that our hyperheuristic algorithm performs well on both constrained and unconstrained problems and across a wide range of parameter sizes and data sets. Like the best known results, some of these data sets have been designed using human ingenuity. Human design ensures that these benchmarks capture especially pathological 'corner cases' and problems with specific structures that are known to pose challenges to the CIT algorithms. Overall, our results provide evidence to support the claim that hyperheuristic search is a promising solution for CIT which suggests it may be useful on other search based problems.

The primary contributions of this paper are:

1. The formulation of CIT as a hyperheuristic search problem and the introduction of the first hyperheuristic algorithm (HHSA) for solving it. It is the first use of hyperheuristic learning in the software testing literature.

2. A comprehensive empirical study showing that HHSA is both effective and efficient. The study reports results across a wide range of 59 previously studied benchmarks. We also study 26 problem instances from two previous studies where each of the 26 CIT problems is drawn from a real-world configurable system testing problem.

3. A study using Amazon EC2 cloud to measure the real computational cost (in US dollars) of the algorithms studied. These results indicate that, with default settings, our hyperheuristic algorithm can produce competitive results to state-of-the-art tools at a reasonable cost; all pairwise interaction tests reported in the paper for all 26 real-world problems and the 44 pairwise benchmarks cost only $2.09.

4. A further empirical study is used to explore the nature of online learning employed by our algorithm. The results of this study show that the hyperheuristic search productively combines heuristic operators that would have proved to be unproductive in isolation and that our algorithm adapts its choice of operators based on the problem.

II. Preliminaries

In this section we will give a quick overview of the notation used throughout the paper. CIT seeks to select a set of N test cases that cover all possible value combinations between any set of t parameters. It produces the selected test set in a Covering Array (CA) notation, which is typically represented as follows in the literature:

$$CA(N; t, v_1^{k_1} v_2^{k_2} ... v_m^{k_m})$$

where N is the number of selected tests (array size), the sum of $k_1, ..., k_m$ is the number of parameters (or factors) in each test case (denoted by k), each v_i stands for the number of values for each of the k_i parameters in turn and t is the strength of the array; a t-way interaction test suite aims to cover all possible t-way combinations of values between any t parameters.

Suppose we want to generate a pairwise (aka 2-way) interaction test suite for an instance with 3 parameters, where the first and second parameter can take 4 different values and the third one can only take 3 different values. Then the problem can be formulated as: $CA(N; 2, 4^2 3^1)$ and the model of the problem is $4^2 3^1$. In order to test all combinations one would need $4 * 4 * 3 = 48$ test cases; pairwise coverage reduces this number to 16. We can also introduce the following constraint: the first value of the first and third parameters cannot be combined together. It turns out that adding multiple constraints can significantly reduce test suite size. These naturally occur in real-world problems, thus constrained CIT is well-fitted for industrial applications [28].

Many different algorithms have been introduced to generate covering arrays. Each algorithm is customised for specific problem instances. For example, there are many greedy algorithms, such as AETG [15] , IPOG [16] and PICT [29]. These methods either generate a new test case on-the-fly, seeking to cover the largest number of uncovered t-way interactions, or start with a small number of parameters and iteratively add new columns and rows to fill in the missing coverage.

Other approaches include metaheuristic search algorithms, such as simulated annealing [2], [10], [20] or tabu search [13]. These metaheuristics are usually divided into two phases or stages. In the first stage, binary search, for instance, is used to generate a random test suite, r of fixed size n. In the second stage, metaheuristic search is used to search for a test suite of size n, starting with r, that covers as many interactions as possible. And there are other unique algorithms, such as those that use constraint solving or logic techniques as the core of their approach [17], [23].

III. Hyperheuristic CIT Algorithm

There are two subclasses of hyperheuristic algorithms: generative and selective. Generative hyperheuristics combine low level heuristics to generate new higher level heuristics. Selective hyperheuristics select from a set of low level heuristics. In this paper we use a selective hyperheuristic algorithm. Selective hyperheuristic algorithms can be further divided into two classes, depending upon whether they are online or offline. Online hyperheuristics are unsupervised, learning strategies

while the algorithms are solving the problem. Offline ones require an additional training step prior to solving the problem. We use online selective hyperheuristics.

The hyperheuristic algorithm takes the set of navigation operators as input. A navigation operator is a lower level heuristic which transforms a given solution into a new solution in the search space. The algorithm layers the heuristic search into two levels that work together to produce the overall solution. The first (or outer) layer uses a normal metaheuristic search to find solutions directly from the solution space of the problem. The inner layer heuristic searches for the best candidate operators for the outer layer heuristics in the current problem state. As a result, the inner search adaptively identifies and exploits different strategies according to the characteristics of the problems it faces.

Our algorithm uses Simulated Annealing (SA) as the outer search. We choose SA because it has been successfully applied to CIT problems, yet, even within this class of algorithms, there is a wide choice of available approaches and implementations [2], [10], [20], [25], [30]. We use a reinforcement learning agent to perform the inner layer selection on heuristics. Our overall algorithm, Hyper Heuristic Simulated Annealing (HHSA), is depicted in Figure 1 and set out more formally as Algorithm 1.

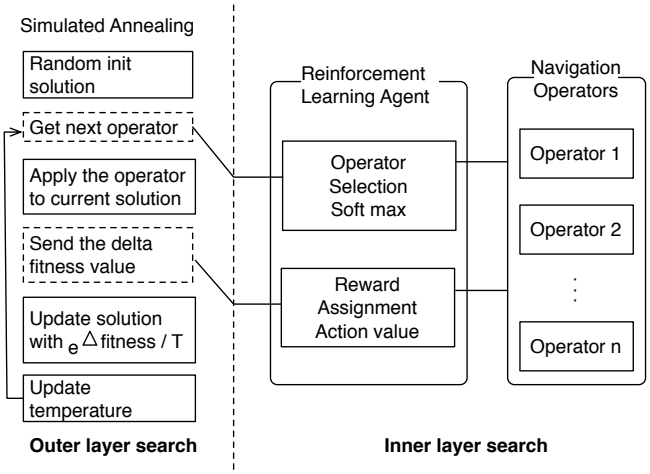

Fig. 1. Hyper Heuristic Simulated Annealing

A. The Outer Layer: Simulated Annealing

A standard Simulated Annealing (SA) algorithm is used as the outer layer search. The SA algorithm starts with a randomly generated $N \times k$ array as an initial solution. The fitness value for each solution is the number of uncovered t-tuples present in the current array. The fitness value is also used to represent the current state of the problem (i.e. how many tuples remain to be covered). This 'problem state' is used to understand how our algorithm learns throughout different stages of the problem.

In each iteration, the SA algorithm asks the reinforcement learning agent to choose a best operator for the current problem state. It then applies the operator and passes the change in fitness value (delta fitness) back to the agent. The SA algorithm accepts the new solution if its fitness is the same as or better than the fitness of the previous solution. Otherwise it uses a probability, $e^{\Delta fitness/T}$, for accepting the current solution based on the current temperature T.

As the SA algorithm proceeds, the temperature, T, is progressively decreased according to a cooling schedule. Decreasing the temperature reduces the probability of SA accepting a move that reduces fitness. The temperature and cooling schedule settings used in the SA are reported in section IV-B. The SA algorithm stops when the current array covers all t-tuples or it reaches a preset maximum number of non-improving moves.

Many real-world CIT models contain constraints (or dependencies) between parameters. To incorporate the constraints, the outer SA first preprocesses the constraints and identifies all invalid tuples which must not be covered. Since previous work [2], [25], [30] used a SAT-solver, MiniSAT, for constraint solving, we also use it in our implementation. Other constraint solvers could be used, but we wish to be able to compare effectiveness to these existing state-of-the-art CIT systems and the best results reported for them.

The outer SA checks constraint violations after applying each operator and proposes a repair if there are any violations. The constraint fixing algorithm is a simple greedy approach that checks each row of the covering array, one at a time. When the algorithm finds a term violation in a row, it attempts to fix the row by changing the value of the parameter which violates the term to a random valid value. The algorithm is set out formally as Algorithm 2.

If the outer SA fails to fix the array, it reapplies the current heuristic operator to generate a new solution.

Input : t, k, v, c, N, $MaxNoImprovment$
Output: covering array A
$A \leftarrow$ `initial_array` (t,k,v,N)
$no\_improvement \leftarrow 0$
$curr\_missing \leftarrow$ `countMissingTuples` (A)
while $curr\_missing \neq 0$ *and* $MaxNoImprovment \neq no\_improvement$ **do**
  $op \leftarrow$ `rl_agent_choose_action` ($curr\_missing$)
  A' = `local_move` (op, A)
  while `fix_cons_violation` (A', c) **do**
    A' = `local_move` (op, A)
  end
  $new\_missing \leftarrow$ `countMissingTuples` (A')
  $\Delta fitness$ = $curr\_missing - new\_missing$
  `rl_agent_set_reward` (op, $\Delta fitness$)
  if $e^{\Delta fitness/Temp} >$ `rand_0_to_1` () **then**
    if $\Delta fitness = 0$ **then**
      $no\_improvement \leftarrow no\_improvement + 1$
    else
      $no\_improvement \leftarrow 0$
    end
    $A \leftarrow A'$
    $curr\_missing \leftarrow new\_missing$
  end
  $Temp \leftarrow$ `cool` ($Temp$)
end

ALGORITHM 1: HHSA

We enclose the SA in a binary search procedure to determine the array size N. This outer binary search procedure is a commonly used solution to iteratively find covering arrays for

different values of N [10], [25], [30], until a smallest covering array of size N can be found. The outer binary search takes an upper and lower bound on the size of array as input, and returns the covering array with a smallest possible size.

The CASA tool for CIT [2], [25] uses a more sophisticated version of the binary search. It first tries the same size multiple times and then does a greedy one sided narrowing to improve the final array size. Our implementation also performs this 'CASA-style' greedy approach to finding the array size, but the use of this approach is tunable.

```
Input  : covering array A, constraints c
Output: has_violation
has_violation ← False
fix_time ← 0
foreach row R in A do
    recheck: foreach clause in c do
        foreach term in clause do
            if R has term then
                has_violation ← True
            else
                has_violation ← False
                break
            end
        end
        if has_violation then
            if fix_time = MaxFixTime then
                break
            end
            term =clause_get_random_term (clause )
            R = random_fix_term (R, term )
            fix_time = fix_time +1
            go to recheck
        end
    end
end
```

ALGORITHM 2: Constraint Violation Fixing

B. The Reinforcement Learning Agent

The goal of the inner layer is to select the best operator at the current problem state. This operator selection problem can be considered an n-Armed Bandit Problem, in which the n arms are the n available heuristics and the machine learner needs to determine which of these heuristics offers the best reward at each problem state. We designed a Reinforcement Learning (RL) agent for the the inner search, as RL agents are known to provide generally good solutions to this kind of so-called 'Bandit Problem' [31].

As shown in Figure 1, the RL agent takes a set of operators as input. In each annealing iteration, the RL agent repeatedly chooses the best fit operator, a, based on the expected reward of applying it at the current problem state. After applying the operator a, the RL agent receives a reward value from the outer layer SA algorithm, based on performance. At the end of the iteration, the RL agent updates the expected reward for the chosen operator with the reward value returned.

The goal of the RL agent is to maximise the expected total reward that accrues over the entire run of the algorithm. Because the reward returned by SA is the improvement of SA's fitness value, the RL agent will thus 'learn' to choose the operators that tend to maximise the SA's fitness improvement, adapting to changes in problem characteristics.

Our RL agent uses an action-value method [31] to estimate the expected rewards for each of the operators available to it at a given problem state. That is, given a set of operators $A = \{a_1, a_2, \ldots, a_i\}$, let $R_i = \{r_{i1}, r_{i2}, \ldots, r_{ik}\}$, be the returned reward values of operator a_i at the k^{th} iteration at which a_i is applied.

Let R_{a_i} be the estimated reward for a_i, which is defined as the mean reward value, $\frac{r_{i1}+r_{i2}+\ldots+r_{ik}}{k}$, received from SA. To balance the twin learning objectives of exploration and exploitation, the RL agent uses a SOFTMAX selection rule [31].

The SOFTMAX selection rule is a greedy approach that gives the operator with the best estimated reward the highest selection probability. For each operator a_i, the selection probability is defined based on the Gibbs distribution: $\frac{e^{R_{a_i}/T}}{\sum_{j=1}^{n} e^{R_{a_j}/T}}$, which is commonly used for the SOFTMAX selection [31]. A higher value of temperature T makes the selection of all operators more equal while a lower value makes a greater difference in selection probability.

C. Search Space Navigation Operators

We have selected a set of six operators to investigate the performance and feasibility of this approach to adaptive learning for CIT. Like any general process, we choose operators that can be widely applicable and which the learner might be able to combine in productive ways. Since we must be general, we cannot exploit specific problem characteristics, leaving it to the learner to find ways to do this through the smart combination of the low level heuristics we define.

We have based our operator selection on the previous algorithms for CIT. None of the operators consider constraints directly, but some have been used for constrained and some for unconstrained problems. Like other machine learning approaches we need a combination of 'smart' heuristics and 'standard' heuristics, since each can act as an enabler for the other. The first three operators are ones we deem to be entirely *standard*; they do not require book keeping or search for particular properties before application. The second set contains ones that we deem to be somewhat *smart*; these are designed with domain knowledge and use information that one might expect could potentially help guide the outer search. The operators are as follows:

1. Single Mutation (Std): Randomly select a row r and a column c, change the value at r, c to a random valid value. This operator matches the neighbourhood transformation in the unconstrained simulated annealing algorithm [10].

2. Add/Del: (Std): Randomly delete a row r and add a new row r' randomly generated. While CASA also includes a row replacement operator, it does not just randomly generate a row.

3. Multiple Mutation (Std): Randomly select two rows, r_1 and r_2, and crossover each column of r_1 and r_2 with a probability of 0.2.

4. Single Mutation (Smart): Randomly select a missing tuple, m, which is the combination of columns $c_1, \ldots, c_n$. Go through each row in the covering array, if there exists a duplicated tuple constructed by the same combination of

columns $c_1, \ldots, c_n$, find a row containing the duplication randomly and change the row to cover the missing tuple m. Otherwise randomly select a row r and change the row to cover the missing tuple m.

5. Add/Del: (Smart): Randomly delete a row r, and add a new row r' to cover n missing tuples. We define n as the smaller value from $k/2$ (where k is the number of parameters) and the number of missing tuples. This is a simple form of constructing a new row used by AETG [15].

6. Multiple Mutation (Smart): Randomly select two rows, r_1 and r_2, and compare the frequency of each value at each column, f_{c1} and f_{c2}. With probability of 0.2, the column with higher frequency will be mutated to a random value.

IV. Experiments

To assess the usefulness of using HHSA as a general approach to CIT, we built a version and posed the following research questions:

RQ1 What is the *quality* of the test suites generated using the hyperheuristic approach?

One of the primary goals of CIT is to find a smallest test suite (defined by the covering array) that achieves the desired strength coverage. It is trivial to generate an arbitrarily large covering test suite – simply include one test case for each interaction to be covered. However, such a naïve approach to test generation would yield exponentially many test cases. All CIT approaches therefore work around the problem of finding a minimal size covering array for testing. The goal of CIT is to try to find a smallest test suite that achieves 100% t-way interaction coverage for some chosen strength of interaction t. In our experiment, we compare the size of the test suites generated by the HSSA in three different ways. We compare against the:

1. Best known results reported in the literature, produced by any approach, including analysis and construction by mathematicians, as is reported in [2].
2. Best known results produced by automated tools.
3. A state-of-the-art SA-based tool that was designed to run on unconstrained problems and a state-of-the-art SA-based tool that was designed to handle constrained problems well.

RQ2 How *efficient* is the hyperheuristic approach and what is the trade-off between the quality of the results and the running time?

Another important issue in CIT is the time to find a test suite that is as close to the minimal one as possible given time budgeted for the search. Depending on the application, one might want to sacrifice minimality for efficiency (or vice-versa). This question investigates whether HHSA can generate small test suites in reasonable time.

If the answers to the first two research questions are favourable to our hyperheuristic algorithm, then we will have evidence that it can be useful. However, usefulness on our set of problems, wide and varied though it is, may not be sufficient for our algorithm to be actually used. We seek to further explore whether its value is merely an artefact of the operators we chose for low level heuristics. We also want to check whether the algorithm is really 'learning'. If not, then it might prove to be insufficiently adaptive to changing problem characteristics. The next two research questions investigate learning.

RQ3 How efficient and effective is each search navigation operator *in isolation*?

In order to collect baseline results for each of the operators that HHSA can choose, we study the effects of each operator in isolation. That is, we ask how well each operator can perform on its own. We also study the effects of making a random choice from all operators at each stage.

Should it turn out that there is a single operator that performs very well across subjects, then there would be no need for further study; we could simply use the high performing operator in isolation. Similarly, should one operator prove to perform poorly and to be expensive then we might consider removing it from further study.

RQ4 Do we see evidence that the hyperheuristic approach is learning?

Should it turn out that HHSA performs well, finding competitively sized covering arrays in reasonable time, then we have evidence to suggest that the adaptive learning used by the hyperheuristic approach is able to learn which operator to deploy. However, is it *really learning*? This RQ investigates, in more detail, the learning strategies as the algorithm searches. We explore how the problem difficulty varies over time for each of the CIT problems we study, and then ask which operators are chosen at each stage of difficulty; is there evidence that the algorithm is selecting different operators for different types of problems?

A. Experimental Setup

In this section we present the experiments conducted[1].

Subjects Studied. There are five subject sets used in our experiments. The details are summarised below:

[Syn-2] contains 14 pairwise (2-way) synthetic models without constraints. These are shown in the leftmost column of Table I. These models are benchmarks that have been used both to compare mathematical constructions as well as search based techniques [2], [10], [11], [18], [32]. We take these from Table 7 from the paper by Garvin et al. [2].

[Syn-3] contains 15 3-way synthetic models without constraints. These are shown in the second column of Table I. These models are benchmarks that have been used for mathematical constructions and search [10], [33], [34]. We take these from Table 7 from the paper by Garvin et al. [2].

[Syn-C2] contains 30 2-way synthetic models with constraints (see Table I, rightmost two columns). These models were designed to simulate configurations with constraints in real-world programs, generated by Cohen et al. [35] and adopted in follow-up research by Garvin et al. [2], [25].

[1]Supplementary data, models and results, can be found on our website (http://cse.unl.edu/~myra/artifacts/HHSA).

TABLE I

SYNTHETIC SUBJECTS SYN-2, SYN-3 AND SYN-C2. THE FIRST SUBJECT SET CONTAINS 2-WAY UNCONSTRAINED SYNTHETIC MODELS FROM [2], [10], [11], [18], [32]. THE SECOND SUBJECT SET CONTAINS 3-WAY UNCONSTRAINED SYNTHETIC MODELS FROM [10], [33], [34]. THE LAST SET CONTAINS SYNTHETIC MODELS DESIGNED TO SIMULATE REAL-WORLD PROGRAMS [2], [25], [35].

Subject Set: Syn-2		**Subject Set: Syn-3**		**Subject Set: Syn-C2**			**Subject Set: Syn-C2**		
Subjects	Model	Subjects	Model	Subjects	Unconstr. Param.	Constr. Param.	Subjects	Unconstr. Param.	Constr. Param.
S2-1	3^4	S3-1	3^6	C2-S1	$2^{86}3^34^15^56^2$	$2^{20}3^34^1$	C2-S16	$2^{81}3^34^26^1$	$2^{30}3^4$
S2-3	$5^13^82^2$	S3-2	4^6	C2-S2	$2_{86}3^34^35^16^1$	$2^{19}3^3$	C2-S17	$2^{128}3^34^25^16^3$	$22^{25}3^4$
S2-3	3^{13}	S3-3	$3^24^25^2$	C2-S3	$2^{27}4^2$	2^93^1	C2-S18	$2^{127}3^24^25^16^3$	$2^{23}3^44^1$
S2-4	$4^13^{39}2^{35}$	S3-4	5^6	C2-S4	$2^{51}3^44^25^1$	$2^{15}3^2$	C2-S19	$2^{172}3^94^95^36^4$	$2^{38}3^5$
S2-5	$5^14^43^{11}2^5$	S3-5	5^7	C2-S5	$2^{155}3^74^35^56^4$	$2^{32}3^64^1$	C2-S20	$2^{138}3^44^55^46^7$	$2^{42}3^6$
S2-6	$4^{15}3^{17}2^{29}$	S3-6	6^6	C2-S6	$2^{73}4^36^1$	$2^{26}3^4$	C2-S21	$2^{76}3^34^25^16^3$	$2^{40}3^6$
S2-7	$6^15^14^63^82^3$	S3-7	$6^64^22^2$	C2-S7	$2^{29}3^1$	$2^{13}3^2$	C2-S22	$2^{73}3^34^3$	$2^{31}3^4$
S2-8	$7^16^15^14^53^82^3$	S3-8	$10^16^24^33^1$	C2-S8	$2^{109}3^24^25^36^3$	$2^{32}3^44^1$	C2-S23	$2^{25}3^16^1$	$2^{13}3^2$
S2-9	4^{100}	S3-9	8^8	C2-S9	$2^{57}3^14^15^16^1$	$2^{30}3^7$	C2-S24	$2^{110}3^25^36^4$	$2^{25}3^4$
S2-10	6^{16}	S3-10	7^7	C2-S10	$2^{130}3^64^55^26^4$	$2^{40}3^7$	C2-S25	$2^{118}3^64^25^26^6$	$2^{23}3^44^1$
S2-11	7^{16}	S3-11	9^9	C2-S11	$2^{84}3^44^25^26^4$	$2^{28}3^4$	C2-S26	$2^{87}3^14^35^4$	$2^{28}3^4$
S2-12	8^{16}	S3-12	10^6	C2-S12	$2^{136}3^44^35^16^3$	$2^{23}3^4$	C2-S27	$2^{55}3^24^25^16^2$	$2^{17}3^3$
S2-13	8^{17}	S3-13	10^{10}	C2-S13	$2^{124}3^44^15^26^2$	$2^{22}3^4$	C2-S28	$2^{167}3^{16}4^25^36^6$	$2^{31}3^6$
S2-14	10^{20}	S3-14	12^{12}	C2-S14	$2^{81}3^54^36^3$	$2^{13}3^2$	C2-S29	$2^{134}3^75^3$	$2^{19}3^3$
		S3-15	14^{14}	C2-S15	$2^{50}3^44^15^26^1$	$2^{20}3^2$	C2-S30	$2^{72}3^44^16^2$	$2^{20}3^2$

[Real-1] contains real-world models from a recent benchmark created by Segall et al. [21], shown in Table II. There are 20 CIT problems in this subject set, generated by or for IBM customers. The 20 problems cover a wide range of applications, including telecommunications, healthcare, storage and banking systems.

[Real-2] contains 6 real-world constrained subjects shown in Table II, which have been widely studied in the literature [2], [25], [30], [35], [36]. The TCAS model was first presented by Kuhn et al. [36]. TCAS is a traffic collision avoidance system from the 'Siemens' suite [37]. The rest of the models in this subject set were introduced by Cohen et al. [30], [35]. SPIN-S and SPIN-V are two components for model simulation and model verification. GCC is a well known compiler system from the GNU Project. Apache is a web server application and Bugzilla is a web-based bug tracking system.

Methodology: All experiments but one are carried out on a desktop computer with a 6 core 3.2GHz Intel CPU and 8GB memory. To understand the trade-off between the quality of the results and the cost of the hyperheuristics approach, we use the Amazon EC2 Cloud. All experiments are repeated five times. We report the best and the average results over five runs.

B. HHSA Configuration

There are four parameters that impact the computational resources used by our hyperheuristic algorithm, HHSA: the initial temperature, the cooling rate, the cooling step function, and maximum number of non-improvements allowed before termination is forced. A higher initial temperature allows HHSA to spend more effort in exploring the search space. The cooling rate and cooling step function work together to control the cooling schedule for HHSA.

To understand the trade-off between the quality of the results and the efficiency of HHSA, we use three different configurations: HHSA-L (LOW), HHSA-M (MEDIUM) and HHSA-H (HIGH). The HHSA-L and HHSA-M configurations only apply the outer binary search to guide HHSA to search for a smallest test suite while the HHSA-H configuration additionally applies the greedy search conducted after the binary search. The settings are shown in Table III.

TABLE II

REAL-WORLD SUBJECT SETS. REAL-1 (TOP) CONTAINS 20 MODELS FROM [21]. REAL-2 (BOTTOM) CONTAINS 6 MODELS WITH CONSTRAINTS FROM [2], [25], [30], [35], [36].

Subjects	Unconstrained Parameters	Constrained Param.
Real-1: 2-way		
Concurrency	2^5	$2^43^15^2$
Storage1	$2^13^14^15^1$	4^{95}
Banking1	3^44^1	5^{112}
Storage2	3^46^1	-
CommProtocol	$2^{10}7^1$	$2^{10}3^{10}4^{12}5^{96}$
SystemMgmt	$2^53^45^1$	$2^{13}3^4$
Healthcare1	$2^63^25^16^1$	2^33^{18}
Telecom	$2^53^14^25^16^1$	$2^{11}3^14^9$
Banking2	$2^{14}4^1$	2^3
Healthcare2	$2^53^64^1$	$2^13^65^{18}$
NetworkMgmt	$2^24^15^310^211^1$	2^{20}
Storage3	$2^93^15^36^18^1$	$2^{38}3^{10}$
Proc.Comm1	$2^33^64^6$	2^{13}
Services	$2^33^45^28^210^2$	$3^{386}4^2$
Insurance	$2^63^15^16^211^113^117^131^1$	-
Storage4	$2^53^74^15^26^27^19^113^1$	2^{24}
Healthcare3	$2^{16}3^64^55^16^1$	2^{31}
Proc.Comm2	$2^33^{12}4^85^2$	1^42^{121}
Storage5	$2^53^85^36^28^19^110^211^1$	2^{151}
Healthcare4	$2^{13}3^{12}4^65^26^17^1$	2^{22}
Real-2: 2,3-way		
TCAS	$2^73^24^110^2$	2^3
Spin-S	$2^{13}4^5$	2^{13}
Spin-V	$2^{42}3^24^{11}$	$2^{47}3^2$
GCC	$2^{189}3^{10}$	$2^{37}3^3$
Apache	$2^{158}3^84^45^16^1$	$2^33^14^25^1$
Bugzilla	$2^{49}3^14^2$	2^43^1

TABLE III

SETTINGS FOR THE HHSA-L, HHSA-M AND HHSA-H CONFIGURATIONS.

Config.	Search	InitT	Co-Rate	Co-Step	MaxNo-Imp
HHSA-L	binary	0.3	0.98	2,000	50,000
HHSA-M	binary	0.3	0.998	10,000	50,000
HHSA-H	binary	0.3	0.998	10,000	50,000
	greedy	0.5	0.9998	10,000	100,000

We chose these settings after some experimentation so that all can be executed in reasonable time for one or more use-cases of CIT. In the low setting, the time taken is low, but the expected result quality is consequently equally low, whereas in the higher settings, we can explore if additional benefits are gained from the allocation of extra computational resources.

TABLE IV

SIZES AND TIMES (SECONDS) FOR SYN-2 (TOP) AND SYN-3 (BOTTOM). THE BEST COLUMN REPORTS THE BEST KNOWN RESULTS FROM [2]. THE SA AND CASA COLUMNS REPORT THE SIZE OF THE UNCONSTRAINED SA AND THE CASA ALGORITHM. THE SIZE FOR EACH HHSA VARIANT REPORTS THE BEST RESULT OVER FIVE RUNS. TIME IS THE AVERAGE RUNTIME (SECONDS). DIFF-BEST INDICATES THE DIFFERENCE BETWEEN THE SMALLEST HHSA VARIANT AND THE BEST COLUMN.

Subject	Best	SA	CASA	HHSA-L		HHSA-M		HHSA-H		Diff-Best	Diff-SA	Diff-CASA
				Size	Time	Size	Time	Size	Time			
S2-1	9	9	9	9	1	9	12	9	44	0	0	0
S2-2	15	15	15	15	1	15	14	15	120	0	0	0
S2-3	15	15	15	15	1	15	14	15	101	0	0	0
S2-4	21	21	22	22	6	21	92	21	1,086	0	0	-1
S2-5	21	21	23	22	1	22	21	21	241	0	0	-2
S2-6	30	30	30	31	4	29	212	29	961	-1	-1	-1
S2-7	30	30	30	30	1	30	41	30	177	0	0	0
S2-8	42	42	46	42	1	42	22	42	175	0	0	-4
S2-9	45	45	46	47	41	46	259	45	2,647	0	0	-1
S2-10	62	62	64	66	2	64	31	63	293	1	1	-1
S2-11	84	87	86	88	3	87	43	86	315	2	-1	0
S2-12	110	112	112	115	6	112	54	111	581	1	-1	-1
S2-13	111	114	114	117	7	115	62	113	644	2	-1	-1
S2-14	162	183	185	195	15	194	98	189	1,201	27	6	4
Overall	757	786	797	814	90	801	975	789	8,586	32	3	-8
S3-1	33	33	33	33	0	33	2	33	5	0	0	0
S2-2	64	64	96	64	0	64	1	64	1	0	0	-32
S3-3	100	100	100	101	1	100	31	100	153	0	0	0
S3-4	125	152	185	176	2	161	21	125	78	0	-27	-60
S3-5	180	201	213	211	3	205	40	202	473	22	1	-11
S3-6	258	300	318	316	4	315	56	308	875	50	8	10
S3-7	272	317	383	345	11	329	123	319	1,893	47	2	-64
S3-8	360	360	360	360	6	360	138	360	498	0	0	0
S3-9	512	918	942	958	39	1,000	187	994	6,966	446	40	16
S3-10	545	552	573	595	14	595	99	575	2,309	30	23	2
S3-11	729	1,426	1,422	1,520	112	1,637	351	1,600	7,206	791	94	98
S3-12	1,100	1,426	1,462	1,440	44	1,530	329	1,496	10,921	340	14	-22
S3-13	1,219	2,163	2,175	2,190	231	2,440	543	2,453	11,138	971	27	15
S3-14	2,190	4,422	4,262	4,760	831	5,080	1,634	5,080	17,679	2,570	338	498
S3-15	3,654	8,092	8,103	9,195	3,684	9,040	5,748	9,039	30,611	5,385	947	936
Overall	11,341	20,526	20,627	22,264	4,982	22,889	9,303	22,748	90,807	10,652	1467	1366

V. RESULTS

In this section we provide results aimed at answering each of our research questions.

A. RQ1: Quality of Hyperheuristic Search

We begin by looking at the set of unconstrained synthetic problems (Table IV) for 2- (top) and 3-way (bottom) CIT. We see the best reported solutions from the literature followed by a smallest CIT sample and its running time for each of the three settings of the HHSA. The best column follows the format of Table 7 from Garvin et al. [2] and includes results obtained by mathematical or constructive methods as well as search. We also include the size reported in that paper both for the unconstrained SA and CASA tools, which is optimized for constrained problems. Running times for SA and CASA tools are not reported, since we did not re-run them.

The size and time columns give a smallest size of the CIT sample found by HHSA, and the average running time in seconds over five runs. The Diff-Best column reports the difference between the best known results (first column) and HHSA's best results. We have also reported HHSA vs. SA (Diff-SA) and HHSA vs. CASA (Diff-CASA). A negative value indicates that HHSA found a smaller sample.

The sizes of test suites found by HHSA are very close to the benchmarks for all but one of the 2-way unconstrained synthetic models. In fact, in benchmark S2-6, both the medium and high settings of HHSA find a lower bound.

The last subject, S2-14 is interesting because it is pathological and has been studied extensively by mathematicians. The model 10^{20}, has 20 parameters, each with 10 values. The use of customizations for this particular problem, such as symmetry has led to both constructions and post optimizations. The discussion of this model consumes more than half a page in a recent dissertation which is credited with the bound[2] of 162 [38]. The best simulated annealing bound, of 183, is close to the high setting of HHSA (189).

There is a larger gap between the results generated by HHSA and best known results on 3-way synthetic models. On the smaller models, HHSA seems to generate sample sizes between the unconstrained SA technique and CASA. However, on the larger size models HHSA does not fare as well. We do see improvement as we increase from low to high, and these are all very large search spaces; we explore the cost-effectiveness trade-off in RQ2.

We now turn to the constrained synthetic models seen in Table V. In this table the column labelled 'Best' represents the best known results for CASA (the only tool on which these synthetic benchmarks have been reported to date). For the constrained problems HHSA performs as well or better than the best known results (except in one case) despite the fact that CASA is optimized for these subjects. HHSA requires 39 fewer rows overall than the best reported results.

[2]This bound was recently reduced by others to 155.

The last comparison we make is with the Real benchmarks. Table VI shows a comparison for all of our real subjects against a set of existing tools which were reported in the literature. Again we see that the HHSA performs as well or better than all of the other tools. For the Real-1 benchmarks, HHSA reduces the overall number of rows in our samples by 52, and for the open source applications HHSA reduces the 2-way by 3 rows, and the 3-way by 54 rows.

Summary of RQ1. *We conclude that the quality of results obtained by using HHSA is high. While we do not produce the best results on every model, we are quite competitive and for all of the real subjects we are as good as, or improve upon the best known results.*

TABLE VI
SIZES AND TIMES (SECONDS) FOR REAL-1 2-WAY (TOP), REAL-2 2-WAY (MIDDLE) AND REAL-2 3-WAY (BOTTOM). THE BEST KNOWN COLUMN SHOWS THE BEST RESULTS IN THE LITERATURE, AND THE TOOLS THAT PRODUCED THE RESULTS. REFERENCES WHERE THESE ARE REPORTED ARE LISTED. THE SIZE COLUMNS FOR EACH VARIANT REPORT THE BEST RESULT OVER FIVE RUNS.

Sub.	Best Known		HHSA-L		HHSA-M		HHSA-H		Diff
	Size	Tools	Size	Time	Size	Time	Size	Time	
Tools: A-ACTS, F-FoCuS, J-Jenny, P-PICT, C-CASA, T-Ttools									
Subject set: Real-1, 2-way [21]									
Con.	5	A,J	5	0	5	9	5	76	0
Sto.1	17	F	17	2	17	67	17	396	0
Ban.1	14	F	13	1	13	24	13	205	-1
Sto.2	18	F	18	1	18	23	18	100	0
Com.	16	F	16	3	16	86	16	898	0
Sys.	16	F	15	1	15	16	15	103	-1
Hea.1	30	A,F	30	2	30	49	30	193	0
Tel.	30	F	30	2	30	40	30	163	0
Ban.2	10	A	10	1	10	28	10	96	0
Hea.2	18	A,P,F	14	1	14	17	14	143	-4
Net.	115	F	110	2	110	63	110	229	-5
Sto.3	52	A,F	50	5	50	136	50	578	-2
Pro.1	28	J	23	1	22	14	22	123	-6
Ser.	102	F	100	10	100	266	100	1,008	-2
Ins.	527	A,P,F	527	13	527	411	527	1,549	0
Sto.4	130	P,F	117	3	117	80	117	345	-13
Hea.3	35	F	34	2	34	34	34	189	-1
Pro.2	32	A	28	5	27	54	27	66	-5
Sto.5	226	F	215	17	215	415	215	1,501	-11
Hea.4	47	F	46	3	46	45	46	230	-1
Overall	1,468	-	1,418	75	1,416	1,877	1,416	8,191	- 52
Subject set: Real-2, 2-way [39] [25]									
TCAS	100	C,T	100	6	100	166	100	578	0
SPIN-S	19	C	19	1	19	27	19	144	0
SPIN-V	32	C	33	11	31	212	31	1,725	-1
GCC	19	C	19	43	17	578	18	2,552	-2
Apache	30	C,T	31	71	30	656	30	3,676	0
Bugzilla	16	C,T	16	3	16	28	16	119	0
Overall	216	-	218	135	213	1,667	214	8,794	-3
Subject set: Real-2, 3-way [39] [2]									
TCAS	401	T	400	141	400	4,636	400	13,808	-1
SPIN-S	95	C	95	14	80	200	80	680	-15
SPIN-V	232	C	217	818	202	7,942	195	37,309	-37
GCC	94	C	102	7,562	94	83,324	-	-	0
Apache	177	C	193	25,258	176	191,630	-	-	-1
Bugzilla	59	C,T	61	156	59	1,769	60	1,726	0
Overall	1,058	-	1,068	33,949	1,011	289,501	-	-	-54

B. RQ2: Efficiency of Hyperheuristic HHSA

Table VII summarizes the average execution time in seconds per subject within each group of benchmarks, using the three configurations of HHSA. The average execution time for the experiments with low configuration is about 17 minutes. Despite the overall average of 17 minutes, the majority of the executions require fewer than 5 minutes. The 3-way experiments running GCC and Apache in the Real-2 benchmarks take the longest (1.6 hours on average). The high setting for this subject set was not finished after 3 days so we terminated it (indicated by '-'). HHSA-M is about 12 times slower overall than HHSA-L. However, most of the subjects still run within 10 minutes. The runtime for HHSA-H is about 7 times slower than for HHSA-M and takes at most 1.5 hours for the majority of the subjects.

On the right side of this table we see the 'Time Ratio' between the HHSA-L vs. HHSA-M and HHSA-M vs. HHSA-H, as well as the 'Size Improvement' which indicates how much smaller the second variant is. As we can see, while it costs us 12 times more to run the HHSA-M variant, it reduces our sample sizes by almost 3%.

Moving from HHSA-M to HHSA-H improves our results by another 1%, while the cost is 7 times more in algorithm runtime. If we also consider the time to run test suites for this sample (see [2]), then this may make a practical difference. Consider if it takes overnight to run a full test suite for each configuration in our sample. The extra computation time for construction may pay off.

We next examine the practical implications of running the different variants of our algorithm. For this experiment we run all of the 2-way subjects in the Amazon EC2 (Elastic Compute Cloud) with the High-CPU On-Demand Instance (c1.medium) [40], and record not only the time, but the actual cost for obtaining our results.

We run the CASA tool as a baseline and the HHSA-L and HHSA-M settings. The results are shown in Table VIII. The times shown represent the average total time for all programs in the respective benchmarks. Note that the runtimes reported in Table VIII are much slower than the times reported in Table IV-VI. This is due to the fact that the computational power of the Amazon EC2 instances used in these experiments are slower than the desktop machine used for prior experiments. The HHSA-L setting took about 8 tenths of an hour to run all of the benchmarks, but cost only 13 cents. CASA took more time than the HHSA-L variant (2.9 hours) and cost $0.49. The HHSA-M required the longest runtime (12.7 hours), but still only cost us $2.09.

Summary of RQ2. *We conclude that the HHSA algorithm is efficient when run at the lowest level (HHSA-low). When run at the higher levels we see a cost-quality trade-off. In practice, the monetary cost of running these algorithms is very small.*

C. RQ3: Search Navigation Operator Comparison

We now examine how efficient and effective each of the search navigation operators are in isolation. We built seven versions of the simulated annealing algorithm, all using the HHSA-M settings. Each of the first six versions contains a single operator. For the seventh version, HH-Random, we include all operators, but the operator to use at each stage is chosen at random (with no intelligence).

TABLE V

SIZES AND TIMES (SECONDS) FOR SYN-C2. THE BEST COLUMN REPORTS THE BEST RESULTS FROM CASA. THE SIZE COLUMNS FOR EACH HHSA VARIANT REPORTS THE BEST RESULT OVER FIVE RUNS. THE DF COLUMN IS THE DIFFERENCE BETWEEN THE BEST HHSA SETTING AND THE BEST.

Sub.	Best	HHSA-L		HHSA-M		HHSA-H		Df	Sub.	Best	HHSA-L		HHSA-M		HHSA-H		Df
		Size	Time	Size	Time	Size	Time				Size	Time	Size	Time	Size	Time	
CS1	38	39	16	37	563	36	3,093	-2	CS16	19	24	27	24	177	24	689	5
CS2	30	30	30	30	391	30	1,074	0	CS17	39	41	16	36	575	36	2,648	-3
CS3	18	18	2	18	24	18	130	0	CS18	43	44	31	41	397	39	5,779	-4
CS4	20	20	7	20	164	20	448	0	CS19	47	50	96	46	1,134	44	10,685	-3
CS5	47	49	59	45	894	44	8,731	-3	CS20	53	55	90	52	1,286	50	12,622	-3
CS6	24	24	16	24	149	24	1,248	0	CS21	36	36	23	36	411	36	2,513	0
CS7	9	9	3	9	74	9	364	0	CS22	36	36	12	36	345	36	2,234	0
CS8	39	41	22	38	875	37	5,362	-2	CS23	12	12	2	12	11	12	188	0
CS9	20	20	27	20	253	20	682	0	CS24	44	46	18	41	283	40	3,909	-4
CS10	43	46	53	43	611	40	8,902	-3	CS25	49	51	37	47	748	46	6,399	-3
CS11	41	43	21	39	222	38	3,096	-3	CS26	30	31	17	28	348	27	1,927	-3
CS12	40	40	32	37	952	36	4,097	-4	CS27	36	36	8	36	151	36	671	0
CS13	36	36	45	36	598	36	3,309	0	CS28	50	53	77	50	902	48	10,709	-2
CS14	36	37	20	36	304	36	1,780	0	CS29	27	30	32	26	528	26	2,995	-1
CS15	30	30	11	30	239	30	628	0	CS30	17	19	12	17	158	16	1,405	-1
									Ov.	1,009	1,046	862	990	13,767	970	108,317	-39

TABLE VII

RUNNING TIMES (SECONDS) OF THE THREE LEVELS OF HHSA. EACH TIME REPRESENTS THE AVERAGE FOR EACH INDIVIDUAL MODEL WITHIN THE BENCHMARK. TIME RATIO SHOWS THE AVERAGE RATIO AND PERCENTAGE OVER FIVE RUNS (RESPECTIVELY) BETWEEN THE L/M AND M/H. SIZE IMPR. SHOWS THE IMPROVEMENT RATIO FOR THE AVERAGE AND THE (BEST) RESULTS. '-' INDICATES THAT NO RESULT WAS OBTAINED AFTER 3 DAYS. THE AVERAGE ROW REPORTS THE AVERAGE VALUES OVER ALL INDIVIDUAL MODELS OVER THE 6 SUBJECT SETS.

Subject Sets	HHSA-L Time	HHSA-M Time	HHSA-H Time	HHSA-L vs. HHSA-M		HHSA-M vs. HHSA-H	
				Time Ratio	Size Impr. (best)	Time Ratio	Size Impr. (best)
Syn-2	6	70	613	11	2.6% (1.6%)	9	1.6% (1.5%)
Syn-C2	29	459	3,611	16	6.1% (5.4%)	8	1.8% (2.0%)
Syn-3	332	620	6,054	2	-0.6% (-2.8%)	10	0.4% (0.6%)
Real-1	4	94	409	25	0.3% (0.1%)	4	0.1% (0.0%)
Real-2	23	278	1,466	12	2.5% (2.3%)	5	0.9% (-0.5%)
Real-2(3way)	5,658	48,250	-	9	6.1% (5.3%)	-	-
Average	1,009	8,295	2,431	12	2.8% (2.0%)	7	1.0% (0.7%)

TABLE VIII

SIZES, TIMES (SECONDS) AND DOLLAR COSTS FOR RUNNING EACH OF THE BENCHMARK SETS TO COMPLETION IN THE AMAZON EC2 CLOUD WITH THE HIGH-CPU ON-DEMAND INSTANCE (C1.MEDIUM) [40]. THE TIME AND COST COLUMNS FOR EACH HHSA VARIANT REPORT THE AVERAGE RESULTS. THE SIZE COLUMN REPORTS BOTH AVERAGE AND (BEST) RESULTS.

Subjects	CASA			HHSA-L			HHSA-M		
	Time (s)	Cost$	Size (best)	Time (s)	Cost $	Size (best)	Time (s)	Cost$	Size (best)
Syn-S2	5,777	0.26	808 (793)	220	0.01	820 (810)	2,350	0.11	805 (800)
Syn-C2	4,440	0.20	1,053 (1,011)	2,029	0.09	1,067 (1,049)	34,736	1.59	1,005 (991)
Real-1	119	0.01	1,451 (1,422)	185	0.01	1,421 (1,417)	4,660	0.21	1,417 (1,416)
Real-2	265	0.01	233 (223)	383	0.02	222 (217)	3,971	0.18	216 (213)
Overall	10,601	0.49	3,545 (3,449)	2,817	0.13	3,530 (3,493)	45,717	2.09	3,443 (3,420)

The results for operator comparison are shown in Table IX. Each operator is listed in a row (Op1-Op6). The numbers correspond to their earlier descriptions (Section III-C).

The next row is HH-Random, followed by the HHSA-M variant. The best operators on their own appear to be the "mutation" operators. Operator 4 (multiple mutation) seems to work relatively well on its own as does Operator 1 (single mutation). The HH-Rand variant performs second best indicating that the combination of operators is helping the search, and it runs relatively fast. However, without the guidance from learning it appears not do as well as the HHSA-M algorithm.

Summary of RQ3. *We conclude that there is a difference between effectiveness of each of the operators and that combining them contributes to a better quality solution. No single operator provides best results.*

TABLE IX

"NAVIGATION OPERATOR" COMPARISON. OP1 TO OP6 USE THE STANDARD SA (WITH HHSA-M SETTINGS) WITH AN INDIVIDUAL SEARCH OPERATOR. HH-RAND MAKES A RANDOM CHOICE AT EACH EVALUATION. THE SIZE COLUMNS REPORT THE AVERAGE RESULTS OVER FIVE RUNS FOLLOWED BY THE (BEST) RESULT. TIME IS IN SECONDS.

Subjects	Syn-S2		Syn-C2		Real-1		Real-2 (2-way)	
	Size (best)	Time	Size (best)	Time	Size (best)	Time	Size (best)	Time
Op1	841 (820)	68	1,117 (1,079)	862	1,461 (1,440)	35	227 (223)	117
Op2	1,333 (1,306)	113	1,376 (1,339)	2,033	1,500 (1,481)	111	263 (257)	248
Op3	1,235 (1,121)	359	3,298 (2,249)	5,055	2,715 (2,168)	90	726 (542)	868
Op4	816 (804)	208	1,070 (1,045)	1,639	1,420 (1,417)	159	227 (221)	179
Op5	981 (967)	254	1,198 (1,172)	2,884	1,432 (1,424)	294	237 (233)	282
Op6	880 (851)	383	1,042 (1,022)	3,133	1,454 (1,436)	97	221 (218)	562
HH-Rand	812 (806)	321	1,024 (1,014)	2,903	1,419 (1,417)	113	218 (216)	441
HHSA-M	806 (801)	975	1,003 (990)	13,767	1,418 (1,416)	1,877	216 (213)	1,667

D. RQ4: Does the Hyperheuristic Algorithm Learn?

To determine if the operators that are selected by the hyperheuristic SA algorithm are *learned*, we examine Table X and Figure 2. We first look at the graphs. The `x-axis` represents the different *problem states* which correspond to the number of missing tuples that the problem has left to cover. On the left part of the graph, there are many tuples still uncovered,

and towards the right, very few are uncovered. We plot the reward scores from our HHSA algorithm for each operator at each stage (a higher reward score means the operator is more likely to be selected). We show this data for one synthetic and one real subject (due to space limitations), S2-8 (top), and TCAS (bottom). As we can see, early on when the problem is easier, most of the operators are close to the same reward value with one or two standing out (Operator 4 in S2-8 and Operator 5 in TCAS). This changes as we have fewer tuples to cover; most of the operators move towards a negative reward with a few remaining the most useful. Not only do we see different "stages" of operator selection, but we also see two different patterns.

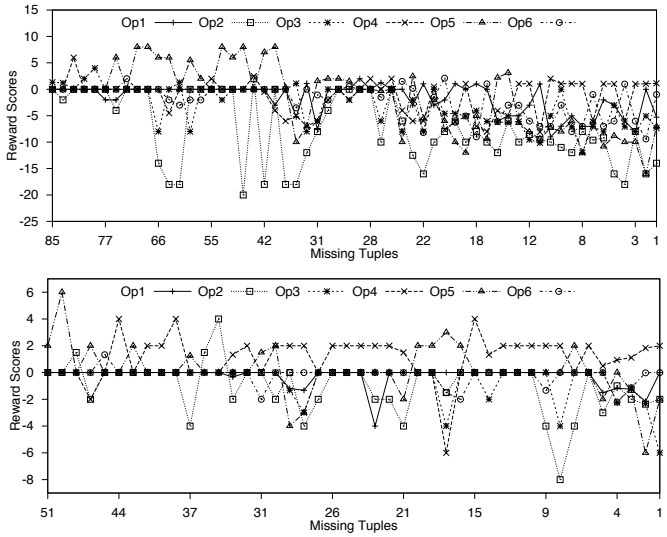

Fig. 2. Subject: S2-8 (top) and TCAS (bottom). X-axis shows number of tuples left to cover. Y-axis shows the HHSA algorithm's reward scores.

We examine this further by breaking down data from each benchmark set into stages (see Table X). We evenly split our data by iteration into an early (S1), middle (S2) and late (S3) stage of the search. For each, we select the pairs of operators that are selected most often across each benchmark set. For instance, Op1+Op4 is selected most often at stage S1 for 6 subjects in the set Syn-2. In stage 1 we see that Op4+Op5 is selected most often overall, while in stage 2 it is Op1+Op4 and in stage 3 it is Op1+Op6. Within each benchmark we see different patterns. For instance, in the first stage Op1+Op5 is selected most often by the Syn-C2 (constrained synthetic) which is different from the others. In stage 2 again we see that the Syn-C2 has a different pattern of operator selection with Op1+Op4 being selected 14 times. In other sets such as the Real 1 we see that the Op4+Op6 combination is chosen most often.

Summary of RQ4. *We see evidence that the Hyperheuristic algorithm is learning both at different stages of search and across different types of subjects.*

TABLE X
LEARNING STRATEGIES. THREE STAGES OF THE ALGORITHM HHSA-L (S1-EARLY), (S2-MIDDLE) AND (S3-LATE) SHOWING THE PAIRS OF OPERATORS CHOSEN THE MOST OFTEN BY STAGE AND SUBJECT SET.

Strategies		Syn-2	Syn-C2	Real-1	Real-2	Ov.
Stage	Operators					
S1	Op1 + Op4	6	2	2	0	10
	Op1 + Op5	0	11	1	1	13
	Op4 + Op5	6	13	12	4	35
	Op4 + Op6	1	0	2	1	4
	Op5 + Op6	1	4	3	0	8
S2	Op1 + Op3	0	1	0	0	1
	Op1 + Op4	0	14	1	2	17
	Op1 + Op5	1	2	2	1	6
	Op1 + Op6	6	6	2	1	15
	Op3 + Op4	0	1	1	0	2
	Op3 + Op5	1	0	3	0	4
	Op3 + Op6	0	1	0	0	1
	Op4 + Op5	0	1	0	0	1
	Op4 + Op6	5	3	7	1	16
	Op5 + Op6	1	1	4	1	7
S3	Op1 + Op3	2	3	2	1	8
	Op1 + Op4	1	2	3	0	6
	Op1 + Op5	0	0	1	1	2
	Op1 + Op6	3	10	6	3	22
	Op2 + Op3	0	0	1	0	1
	Op3 + Op5	0	0	1	0	1
	Op3 + Op6	7	3	3	1	14
	Op4 + Op6	0	10	1	0	11
	Op5 + Op6	1	2	2	0	5

VI. CONCLUSIONS

In this paper we have presented a hyperheuristic algorithm (HHSA) for constructing CIT samples. We have shown that the algorithm is general and *learns* as the problem set changes through a large empirical study on a broad set of benchmarks. We have shown that the algorithm is effective when we compare it across the benchmarks and other algorithms and results from the literature.

We have also seen that the use of different tunings for the algorithm (low, medium and high) will provide a quality-cost trade-off with the higher setting producing better results, but taking longer to run. When we examine the practicality we see that the monetary cost for running the algorithm is quite small when using today's cloud ($2.09).

Finally, we have examined the various stages of learning of our algorithm and see that the different heuristic operators are more effective at different stages (early, middle, late) and that they vary across programs and benchmarks. It is this ability to learn and adapt that we believe is the most important aspect of this search.

As future work we will look at alternative tunings for the algorithm to scale to very large problems (a very low setting) and to find even smaller sample sizes (a very high setting). We will also incorporate new operators and alternative algorithms for the outer layer, such as genetic algorithms.

VII. ACKNOWLEDGMENTS

The FITTEST project FP7/ICT/257574 supports Yue Jia. Mark Harman is supported by the EPSRC, EP/J017515/1 (DAASE), EP/I033688/1 (GISMO), EP/I010165/1 (RE-COST) and EP/G060525/2 (Platform Grant). DAASE also completely supports Justyna Petke. Myra Cohen is supported in part by NSF award CCF-1161767.

REFERENCES

[1] M. Harman, "The current state and future of search based software engineering," in *2007 Future of Software Engineering (FOSE'07)*, 2007, pp. 342–357. [Online]. Available: http://dx.doi.org/10.1109/FOSE.2007.29

[2] B. Garvin, M. Cohen, and M. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," *Empirical Software Engineering*, vol. 16, no. 1, pp. 61–102, 2011. [Online]. Available: http://dx.doi.org/10.1007/s10664-010-9135-7

[3] M. Harman, E. Burke, J. Clark, and X. Yao, "Dynamic adaptive search based software engineering," in *International symposium on Empirical software engineering and measurement (ESEM'12)*, 2012, pp. 1–8.

[4] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and R. Qu, "Hyper-heuristics: A survey of the state of the art," *Journal of the Operational Research Society*, no. 64, pp. 1695–1724, 2013.

[5] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test-case generation," *IEEE Transactions on Software Engineering*, pp. 742–762, 2010.

[6] M. Harman, A. Mansouri, and Y. Zhang, "Search based software engineering: Trends, techniques and applications," *ACM Computing Surveys*, vol. 45, no. 1, p. Article 11, November 2012.

[7] O. R "A survey on search-based software design," *Computer Science Review*, vol. 4, no. 4, pp. 203 – 249, 2010.

[8] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines," *IEEE Trans. Software Eng.*, vol. 40, no. 7, pp. 650–670, 2014. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2014.2327020

[9] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys*, vol. 43, no. 2, pp. 1–29, February 2011.

[10] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in *The 25th International Conference on Software Engineering*, ser. ICSE'03, Washington, DC, USA, 2003, pp. 38–48.

[11] J. Stardom, *Metaheuristics and the Search for Covering and Packing Arrays*. Thesis (M.Sc.)–Simon Fraser University, 2001. [Online]. Available: http://books.google.co.uk/books?id=BIkmYAAACAAJ

[12] G. Dueck, "New optimization heuristics - the great deluge algorithm and the record-to-record travel," *Journal of Computational Physics*, vol. 104, pp. 86–92, 1993.

[13] K. Nurmela, "Upper bounds for covering arrays by tabu search," *Discrete Applied Mathematics*, vol. 138, no. 1-2, pp. 143–152, 2004.

[14] R. C. Bryce and C. J. Colbourn, "One-test-at-a-time heuristic search for interaction test suites," in *Genetic and Evolutionary Computation Conference, GECCO'07*, 2007, pp. 1082–1089.

[15] 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*, vol. 23, no. 7, pp. 437–444, 1997.

[16] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG-IPOG-D: efficient test generation for multi-way combinatorial testing," *Software Testing Verification and Reliability*, vol. 18, pp. 125–148, September 2008.

[17] B. Hnich, S. Prestwich, E. Selensky, and B. Smith, "Constraint models for the covering test problem," *Constraints*, vol. 11, pp. 199–219, 2006.

[18] Y. Lei and K. Tai, "In-parameter-order: a test generation strategy for pairwise testing," in *The 3rd International High-Assurance Systems Engineering Symposium*, 1998, pp. 254–261.

[19] A. Calvagna and A. Gargantini, "A formal logic approach to constrained combinatorial testing," *Journal of Automated Reasoning*, vol. 45, pp. 331–358, December 2010.

[20] J. Martinez-Pena, J. Torres-Jimenez, N. Rangel-Valdez, and H. Avila-George, "A heuristic approach for constructing ternary covering arrays using trinomial coefficients," in *The 12th Ibero-American conference on Advances in artificial intelligence*, ser. IBERAMIA'10, 2010, pp. 572–581.

[21] I. Segall, R. Tzoref-Brill, and E. Farchi, "Using binary decision diagrams for combinatorial test design," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11)*, 2011, pp. 254–264. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001451

[22] C. J. Colbourn, "Covering array tables," Available at http://www.public.asu.edu/∼ccolbou/src/tabby/ catable.html, 2012.

[23] A. Calvagna, A. Gargantini, and P. Vavassori, "Combinatorial interaction testing with CITLAB," in *The Sixth International Conference on Software Testing, Verification and Validation (ICST'13)*, 2013, pp. 376–382.

[24] J. Torres-Jimenez and E. Rodriguez-Tello, "New bounds for binary covering arrays using simulated annealing," *Information Sciences*, vol. 185, no. 1, pp. 137–152, 2012.

[25] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "An improved meta-heuristic search for constrained interaction testing," in *Proceedings of the 2009 1st International Symposium on Search Based Software Engineering (SSBSE'09)*, 2009, pp. 13–22. [Online]. Available: http://dx.doi.org/10.1109/SSBSE.2009.25

[26] R. E. Lopez-Herrejon, J. Ferrer, J. F. Chicano, E. N. Haslinger, A. Egyed, and E. Alba, "Towards a benchmark and a comparison framework for combinatorial interaction testing of software product lines," *The Computing Research Repository (CoRR)*, vol. abs/1401.5367, 2014.

[27] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *The 34th International Conference on Software Engineering, ICSE'12*, 2012, pp. 3–13.

[28] J. Petke, S. Yoo, M. B. Cohen, and M. Harman, "Efficiency and early fault detection with lower and higher strength combinatorial interaction testing," in *The 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, August 2013, pp. 26–36.

[29] J. Czerwonka, "Pairwise testing in real world," in *Pacific Northwest Software Quality Conference*, October 2006, pp. 419–430.

[30] 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*, ser. ISSTA'07. New York, NY, USA: ACM, 2007, pp. 129–139. [Online]. Available: http://doi.acm.org/10.1145/1273463.1273482

[31] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998. [Online]. Available: http://www.cs.ualberta.ca/%7Esutton/book/ebook/the-book.html

[32] Y.-W. Tung and W. Aldiwan, "Automating test case generation for the new generation mission software system," in *IEEE Aerospace Conference 2000*, vol. 1, 2000, pp. 431–437.

[33] M. Chateauneuf and D. L. Kreher, "On the state of strength-three covering arrays," *Journal of Combinatorial Designs*, vol. 10, no. 4, pp. 217–238, 2002. [Online]. Available: http://dx.doi.org/10.1002/jcd.10002

[34] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling, "Augmenting simulated annealing to build interaction test suites," in *The 14th International Symposium on Software Reliability Engineering*, ser. ISSRE'03, Washington, DC, USA, 2003, pp. 394–405. [Online]. Available: http://dl.acm.org/citation.cfm?id=951952.952381

[35] M. Cohen, M. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 633–650, 2008.

[36] D. Richard Kuhn and V. Okum, "Pseudo-exhaustive testing for software," in *The 30th Annual IEEE/NASA Software Engineering Workshop (SEW'06)*, 2006, pp. 153–158. [Online]. Available: http://dx.doi.org/10.1109/SEW.2006.26

[37] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[38] P. Nayeri, "Post-optimization: Necessity analysis for combinatorial arrays," Ph.D. Thesis, Department of Computer Science and Engineering, Arizona State University, April 2011.

[39] A. Calvagna and A. Gargantini, "T-wise combinatorial interaction test suites construction based on coverage inheritance," *Software Testing, Verification and Reliability*, vol. 22, no. 7, pp. 507–526, 2012. [Online]. Available: http://dx.doi.org/10.1002/stvr.466

[40] Amazon, "EC2 (Elastic Compute Cloud)," Available at http://aws.amazon.com/ec2/.