Soft Computing (2020) 24:13929–13954
https://doi.org/10.1007/s00500-020-04769-z

METHODOLOGIES AND APPLICATION

An evaluation of Monte Carlo-based hyper-heuristic for interaction testing of industrial embedded software applications

Bestoun S. Ahmed[1] · Eduard Enoiu[2] · Wasif Afzal[2] · Kamal Z. Zamli[3]

Published online: 25 February 2020

Abstract
Hyper-heuristic is a new methodology for the adaptive hybridization of meta-heuristic algorithms to derive a general algorithm for solving optimization problems. This work focuses on the selection type of hyper-heuristic, called the exponential Monte Carlo with counter (EMCQ). Current implementations rely on the memory-less selection that can be counterproductive as the selected search operator may not (historically) be the best performing operator for the current search instance. Addressing this issue, we propose to integrate the memory into EMCQ for combinatorial t-wise test suite generation using reinforcement learning based on the Q-learning mechanism, called Q-EMCQ. The limited application of combinatorial test generation on industrial programs can impact the use of such techniques as Q-EMCQ. Thus, there is a need to evaluate this kind of approach against relevant industrial software, with a purpose to show the degree of interaction required to cover the code as well as finding faults. We applied Q-EMCQ on 37 real-world industrial programs written in Function Block Diagram (FBD) language, which is used for developing a train control management system at Bombardier Transportation Sweden AB. The results show that Q-EMCQ is an efficient technique for test case generation. Addition- ally, unlike the t-wise test suite generation, which deals with the minimization problem, we have also subjected Q-EMCQ to a maximization problem involving the general module clustering to demonstrate the effectiveness of our approach. The results show the Q-EMCQ is also capable of outperforming the original EMCQ as well as several recent meta/hyper-heuristic including modified choice function, Tabu high-level hyper-heuristic, teaching learning-based optimization, sine cosine algorithm, and symbiotic optimization search in clustering quality within comparable execution time.

Keywords Search-based software engineering (SBSE) · Fault finding · System reliability · Software testing · Hyper-heuristics

1 Introduction

Despite their considerable success, meta-heuristic algorithms have been adapted to solve specific problems based on some domain knowledge. Some examples of recent meta-heuristic algorithms include Sooty Tern optimization algorithm (STOA) (Dhiman and Kaur (2019)), farmland fertility algorithm (FF) (Shayanfar and Gharehchopogh (2018)), owl search algorithm (OSA) (Jain et al. (2018)), human mental search (HMS) (Mousavirad and Ebrahimpour-Komleh (2017)), and find-fix-finish-exploit-analyze (F3EA) (Kashan et al. (2019)). Often, these algorithms require significant expertise to implement and tune; hence, their standard versions are not sufficiently generic to adapt to changing search spaces, even for the different instances of the same problem. Apart from this need to adapt, the existing research on meta-heuristic algorithms has also not sufficiently explored the adoption of more than one meta-heuristic to perform the

Communicated by V. Loia.

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s00500-020-04769-z) contains supplementary material, which is available to authorized users.

✉ Bestoun S. Ahmed
bestoun@kau.se

Eduard Enoiu
Eduard.Enoiu@mdh.se

Wasif Afzal
Wasif.Afzal@mdh.se

Kamal Z. Zamli
kamalz@ump.edu.my

[1] Department of Mathematics and Computer Science, Karlstad University, Karlstad, Sweden

[2] Mälardalen University, Västerås, Sweden

[3] University Malaysia Pahang, Pekan, Malaysia

search (termed *hybridization*). Specifically, the exploration and exploitation of the existing algorithms are limited to use the (local and global) search operators derived from a single meta-heuristic algorithm as a basis. In this case, choosing a proper combination of search operators can be the key to achieve good performance as hybridization can capitalize on the strengths and address the deficiencies of each algorithm collectively and synergistically.

Hyper-heuristics have recently received considerable attention for addressing some of the above issues (Tsai et al. 2014; Sabar and Kendall 2015). Specifically, hyper-heuristic represents an approach of using (meta)-heuristics to choose (meta)-heuristics to solve the optimization problem at hand (Burke et al. 2003). Unlike traditional meta-heuristics, which directly operate on the solution space, hyper-heuristics offer flexible integration and adaptive manipulation of complete (low-level) meta-heuristics or merely the partial adoption of a particular meta-heuristic search operator through non-domain feedback. In this manner, hyper-heuristic can evolve its heuristic selection and acceptance mechanism in searching for a good-quality solution.

This work is focusing on a specific type of hyper-heuristic algorithm, called the exponential Monte Carlo with counter (EMCQ) Sabar and Kendall (2015); Kendall et al. (2014). EMCQ adopts a simulated annealing like Kirkpatrick et al. (1983) reward and punishment mechanism to adaptively choose the search operator dynamically during runtime from a set of available operators. To be specific, EMCQ rewards a good performing search operator by allowing its re-selection in the next iteration. Based on decreasing probability, EMCQ also rewards (and penalizes) a poor performing search operator to escape from local optima. In the current implementation, when a poor search operator is penalized, it is put in the Tabu list, and EMCQ will choose a new search operator from the available search operators randomly. Such memory-less selection can be counterproductive as the selected search operator may not (historically) be the best performing operator for the current search instance. For this reason, we propose to integrate the memory into EMCQ using reinforcement learning based on the Q-learning mechanism, called Q-EMCQ.

We have adopted Q-EMCQ for combinatorial interaction t-wise test generation (where t indicates the interaction strength). While there is already significant work on adopting hyper-heuristic as a suitable method for t-wise test suite generation [see, e.g., Zamli et al. (2016, 2017)], the main focus has been on the generation of minimal test suites. It is worthy of mentioning here that in this work, our main focus is not to introduce new bounds for the t-wise generated test suites. Rather we dedicate our efforts on assessing the effectiveness and efficiency of the generated t-wise test suites against real-world programs being used in industrial practice. Our goal is to push toward the industrial adoption of t-wise testing, which is lacking in numerous studies on the subject. We, nevertheless, do compare the performance of Q-EMCQ against the well-known benchmarks using several strategies, to establish the viability of Q-EMCQ for further empirical evaluation using industrial programs. In the empirical evaluation part of this paper, we rigorously evaluate the effectiveness and efficiency of Q-EMCQ for different degrees of interaction strength using real-world industrial control software used for developing the train control management system at Bombardier Transportation Sweden AB. To demonstrate the generality of Q-EMCQ, we have also subjected Q-EMCQ a maximization problem involving the general module clustering. Q-EMCQ gives the best overall performance on the clustering quality within comparable execution time as compared to competing hyper-heuristics (MCF and Tabu HHH) and meta-heuristics (EMCQ, TLBO, SCA, and SOS). Summing up, this paper makes the following contributions:

This paper makes the following contributions:

1. A novel Q-EMCQ hyper-heuristic technique that embeds the Q-learning mechanism into EMCQ, providing a memory of the performance of each search operator for selection. The implementation of Q-EMCQ establishes a unified strategy for the integration and hybridization of Monte Carlo-based exponential Metropolis probability function for meta-heuristic selection and acceptance mechanism with four low-level search operators consisting of cuckoo's Levy flight perturbation operator (Yang and Deb 2009), flower algorithm's local pollination, and global pollination operator (Yang 2012) as well as Jaya's search operator (Rao 2016).
2. An industrial case study, evaluating t-wise test suite generation in terms of cost (i.e., using a comparison of the number of test cases) and effectiveness (i.e., using mutation analysis).
3. Performance assessment of Q-EMCQ with contemporary meta/hyper-heuristics for maximization problem involving general module clustering problem.

2 Theoretical Background and an Illustrative Example

Covering array (CA) is a mathematical object to represent the actual set of test cases based on t-wise coverage criteria (where t represents the desired interaction strength). CA $(N; t, k, v)$, also expressed as CA $(N; t, v^k)$, is a combinatorial structure constructed as an array of N rows and k columns on v values such that every $N \times t$ sub-array contains all ordered subsets from the v values of size t at least once. Mixed covering array (MCA) $(N; t, k, (v_1, v_2, \dots v_k))$

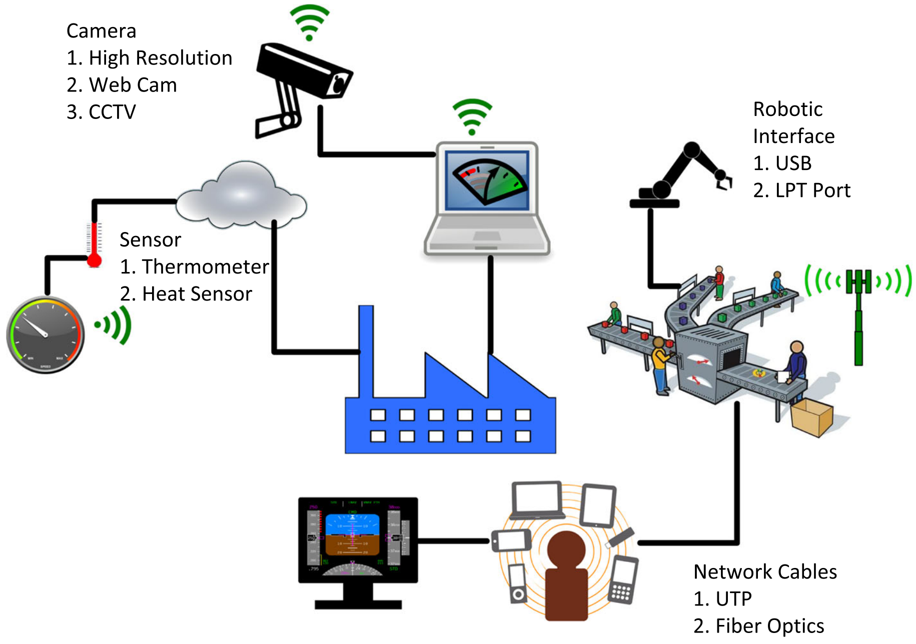


Fig. 1 Interconnected manufacturing system

or MCA $(N; t, k, v^k)$ may be adopted when the number of component values varies.

To illustrate the use of CA for t-wise testing, consider a hypothetical example of an integrated manufacturing system in Fig. 1. There are four basic elements/parameters of the system, i.e., Camera, Robotic Interface, Sensor, and Network Cables. The camera parameter takes three possible values (i.e., Camera = {High Resolution, Web Cam, and CCTV}), whereas the rest of the parameters take two possible values (i.e., Robotic Interface = {USB, HDMI}, Sensor = {Thermometer, Heat Sensor}, and Network Cables = {UTP, Fiber Optics}).

As an example, the mixed CA representation for MCA $(N; 3, 3^1 2^3)$ is shown in Fig. 2 with twelve test cases. In this case, there is a reduction of 50% test cases from the 24 exhaustive possibilities.

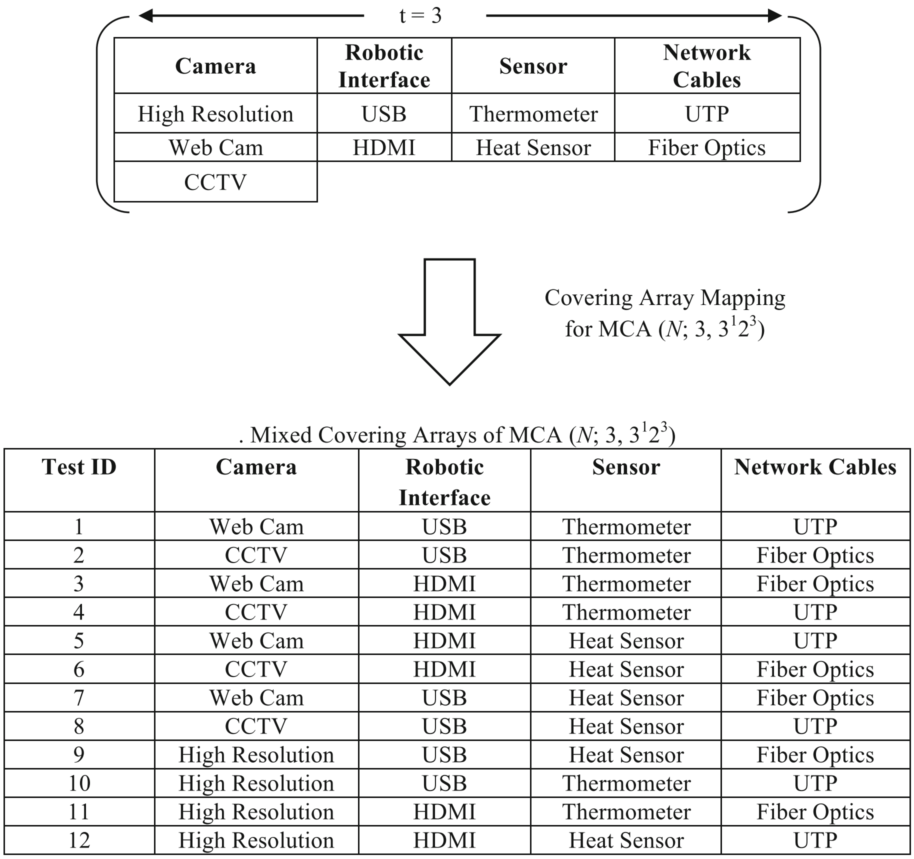
3 Related Work

In this section, we present the previous work performed on the combinatorial t-wise test generation and the evaluation of such techniques in terms of efficiency and effectiveness.

3.1 Combinatorial *t*-wise test suite generators

CA construction is an NP-complete problem (Lei and Tai 1998). CA construction is directly applied for t-wise test case reduction; thus, considerable research has been carried out to develop effective strategies for obtaining (near) optimal solutions. The existing works for CA generation can be classified into two main approaches: mathematical and greedy computational approaches. The mathematical approach often exploits the mathematical properties of orthogonal arrays to construct efficient CA (Mandl 1985). An example of strategies that originate from the extension of mathematical concepts called orthogonal array is recursive CA (Colbourn et al. 2006). The main limitation of the OA solutions is that these techniques restrict the selection of values, which are confined to low interaction (i.e., $t < 3$), thus limiting its applicability for only small-scale systems configurations. Greedy computational approaches exploit computing power to generate the required CA, such that each solution results from the greedy selection of the required interaction. The greedy computational approaches can be categorized further into one-parameter-at-a-time (OPAT) and one-test-at-a-time (OTAT) methods (Nie and Leung 2011). In-parameter-order (IPO) strategy (Lei and Tai 1998) is perhaps the pioneer strategy that adopts the OPAT approach (hence termed IPO-like). IPO strategy is later generalized into a number of variants IPOG (Lei et al. 2007), IPOG-D (Lei et al. 2008), IPOF (Forbes et al. 2008), and IPO-s (Calvagna and Gargantini 2009), whereas AETG (Cohen et al. 1997) is the first CA construction strategy that adopts the OTAT method (hence, termed as AETG-like (Williams and Probert 1996)). Many variants of AETG emerged later, including mAETG (Cohen 2004) and $mAETG_SAT$ (Cohen et al. 2007).

One can find two recent trends in research for combinatorial interaction testing: handling of constraints (Ahmed et al. 2017) and the application of meta-heuristic algorithms. Many current studies focus on the use of meta-heuristic algorithms as part of the greedy computational approach for CA construction (Mahmoud and Ahmed 2015; Wu et al. 2015; Ahmed et al. 2012). Meta-heuristic-based strategies,



Camera	Robotic Interface	Sensor	Network Cables
High Resolution	USB	Thermometer	UTP
Web Cam	HDMI	Heat Sensor	Fiber Optics
CCTV			

. Mixed Covering Arrays of MCA (N; 3, $3^1 2^3$)

Test ID	Camera	Robotic Interface	Sensor	Network Cables
1	Web Cam	USB	Thermometer	UTP
2	CCTV	USB	Thermometer	Fiber Optics
3	Web Cam	HDMI	Thermometer	Fiber Optics
4	CCTV	HDMI	Thermometer	UTP
5	Web Cam	HDMI	Heat Sensor	UTP
6	CCTV	HDMI	Heat Sensor	Fiber Optics
7	Web Cam	USB	Heat Sensor	Fiber Optics
8	CCTV	USB	Heat Sensor	UTP
9	High Resolution	USB	Heat Sensor	Fiber Optics
10	High Resolution	USB	Thermometer	UTP
11	High Resolution	HDMI	Thermometer	Fiber Optics
12	High Resolution	HDMI	Heat Sensor	UTP

Fig. 2 Mixed CA Construction MCA (N; 3, 3^1, 2^3) for interconnected manufacturing system

which complement both the OPAT and OTAT methods, are often superior in terms of obtaining optimal CA size, but trade-offs regarding computational costs may exist. Meta-heuristic-based strategies often start with a population of random solutions. One or more search operators are iteratively applied to the population to improve the overall fitness (i.e., regarding greedily covering the interaction combinations). Although variations are numerous, the main difference between meta-heuristic strategies is on the defined search operators. Meta-heuristics such as genetic algorithm (e.g., GA) Shiba et al. (2004), ant colony optimization (e.g., ACO) Chen et al. (2009), simulated annealing (e.g., SA) Cohen et al. (2007), particle swarm optimization (e.g., PSTG Ahmed et al. (2012), DPSO) Wu et al. (2015), and cuckoo search algorithm (e.g., CS) Ahmed et al. (2015) are effectively used for CA construction.

In line with the development of meta-heuristic algorithms, the room for improvement is substantial to advance the field of search-based software engineering (SBSE) by the provision of hybridizing two or more algorithms. Each algorithm usually has its advantages and disadvantages. With hybridization, each algorithm can exploit the strengths and cover the weaknesses of the collaborating algorithms (i.e., either partly or in full). Many recent scientific results indicate that hybridization improves the performance of meta-heuristic algorithms (Sabar and Kendall 2015).

Owing to its ability to accommodate two or more search operators from different meta-heuristics (partly or in full) through one defined parent heuristic (Burke et al. 2013), hyper-heuristics can be seen as an elegant way to support hybridization. To be specific, the selection of a particular search operator at any particular instance can be adaptively decided (by the parent meta-heuristic) based on the feedback from its previous performance (i.e., learning).

In general, hyper-heuristic can be categorized as either selective or generative ones (Burke et al. 2010). Ideally, a selective hyper-heuristic can select the appropriate heuristics from a pool of possible heuristics. On the other hand, a generative hyper-heuristic can generate new heuristics from the existing ones. Typically, selective and generative hyper-heuristics can be further categorized as either constructive or perturbative ones. A constructive gradually builds a particular solution from scratch. On the other hand, a perturbative hyper-heuristic iteratively improves an existing solution by relying on its perturbative mechanisms.

In hyper-heuristic, there is a need to maintain a “domain barrier” that controls and filters out domain-specific information from the hyper-heuristic itself (Burke et al. 2013).

In other words, hyper-heuristic ensures generality to its approach.

Concerning related work for CA construction, Zamli et al. (2016) implemented Tabu search hyper-heuristic (Tabu HHH) utilizing a selection hyper-heuristic based on Tabu search and three measures (quality, diversity, and intensity) to assist the heuristic selection process. Although showing promising results, Tabu HHH adopted full meta-heuristic algorithms (i.e., comprising of teaching learning-based optimization (TLBO) Rao et al. (2011), particle swarm optimization (PSO) Kennedy and Eberhart (1995), and cuckoo search algorithm (CS) Yang and Deb (2009)) as its search operators. Using the three measures in HHH, Zamli et al. (2017) later introduced the new Mamdani fuzzy-based hyper-heuristic that can accommodate partial truth, hence allowing a smoother transition between the search operators. In other work, Jia et al. (2015) implemented a simulated annealing-based hyper-heuristic called HHSA to select from variants of six operators (i.e., single/multiple/smart mutation, simple/smart add and delete row). HHSA demonstrates good performance regarding test suite size and exhibits elements of learning in the selection of the search operator.

Complementing HHSA, we propose Q-EMCQ as another alternative SA variant. Unlike HHSA, we integrate the Q-learning mechanism to provide a memory of the performance of each search operator for selection. The Q-learning mechanism complements the Monte Carlo-based exponential Metropolis probability function by keeping track of the best performing operators for selection when the current fitness function is poor. Also, unlike HHSA, which deals only with CA (with constraints) construction, our work also focuses on MCA.

3.2 Case studies on combinatorial *t*-wise interaction test generation

The number of successful applications of combinatorial interaction testing in the literature is expanding. Few studies (Kuhn and Okum 2006; Richard Kuhn et al. 2004; Bell and Vouk 2005; Wallace and Richard Kuhn 2001; Charbachi et al. 2017; Bergström and Enoiu 2017; Sampath and Bryce 2012; Charbachi et al. 2017) are focusing on fault and failure detection capabilities of these techniques for different industrial systems. However, still, there is a lack of industrial applicability of combinatorial interaction testing strategies.

Some case studies concerning combinatorial testing have focused on comparing between different strengths of combinatorial criteria (Grindal et al. 2006) with random tests (Ghandehari et al. 2014; Schroeder et al. 2004) and the coverage achieved by such test cases. For example, Cohen et al. (1996) found that pairwise generated tests can achieve 90% code coverage by using the AETG tool. Other studies (Cohen et al. 1994; Dalal et al. 1998; Sampath and Bryce 2012) have reported the use of combinatorial testing on real-world systems and how it can help in the detection of faults when compared to other test design techniques.

Few papers examine the effectiveness (i.e., the ability of test cases to detect faults) of combinatorial tests of different t-wise strengths and how these strategies compare with each other. There is some empirical evidence suggesting that across a variety of domains, all failures could be triggered by a maximum of four-way interactions (Kuhn and Okum 2006; Richard Kuhn et al. 2004; Bell and Vouk 2005; Wallace and Richard Kuhn 2001). In one such case, 67% of failures are caused by one-parameter, two-way combinations cause 93% of failures, and 98% by three-way combinations. The detection rate for other studies is similar, reaching 100% fault detection by the use of four-way interactions. These results encouraged our interest in investigating a larger case study on how Q-EMCQ and different interaction strengths perform in terms of test efficiency and effectiveness for industrial software systems and study the degree of interaction involved in detecting faults for such programs.

4 Overview of the proposed strategy

The high-level view of Q-EMCQ strategy is illustrated in Fig. 3. The main components of Q-EMCQ consist of the algorithm (along with its selection and acceptance mechanism) and the defined search operators. Referring to Fig. 3, Q-EMCQ chooses the search operator much like a multiplexer via a search operator connector based on the memory on its previous performances (i.e., penalize and reward). However, it should be noted that the Q-learning mechanism is only summoned when there are no improvements in the prior iteration. The complete detailed working of Q-EMCQ is highlighted in the next subsections.

4.1 Q-learning Monte Carlo hyper-heuristic strategy

The exponential Monte Carlo with counter (EMCQ) algorithm from Ayob and Kendall (2003); Kendall et al. (2014) has been adopted in this work as the basis of Q-EMCQ selection and acceptance mechanism. EMCQ algorithm accepts poor solution (similar to simulated annealing (Kirkpatrick et al. 1983); the probability density is defined as:

$$\psi = e^{-\frac{\delta T}{q}} \tag{1}$$

where δ is the difference in fitness value between the current solution (S_i) and the previous solution (S_0) (i.e., $\delta = f(S_i) - f(S_0)$), T is the iteration counter, and q is a control parameter for consecutive non-improving iterations.

Similar to simulated annealing, probability density Ψ decreases toward zero as T increases. However, unlike sim-

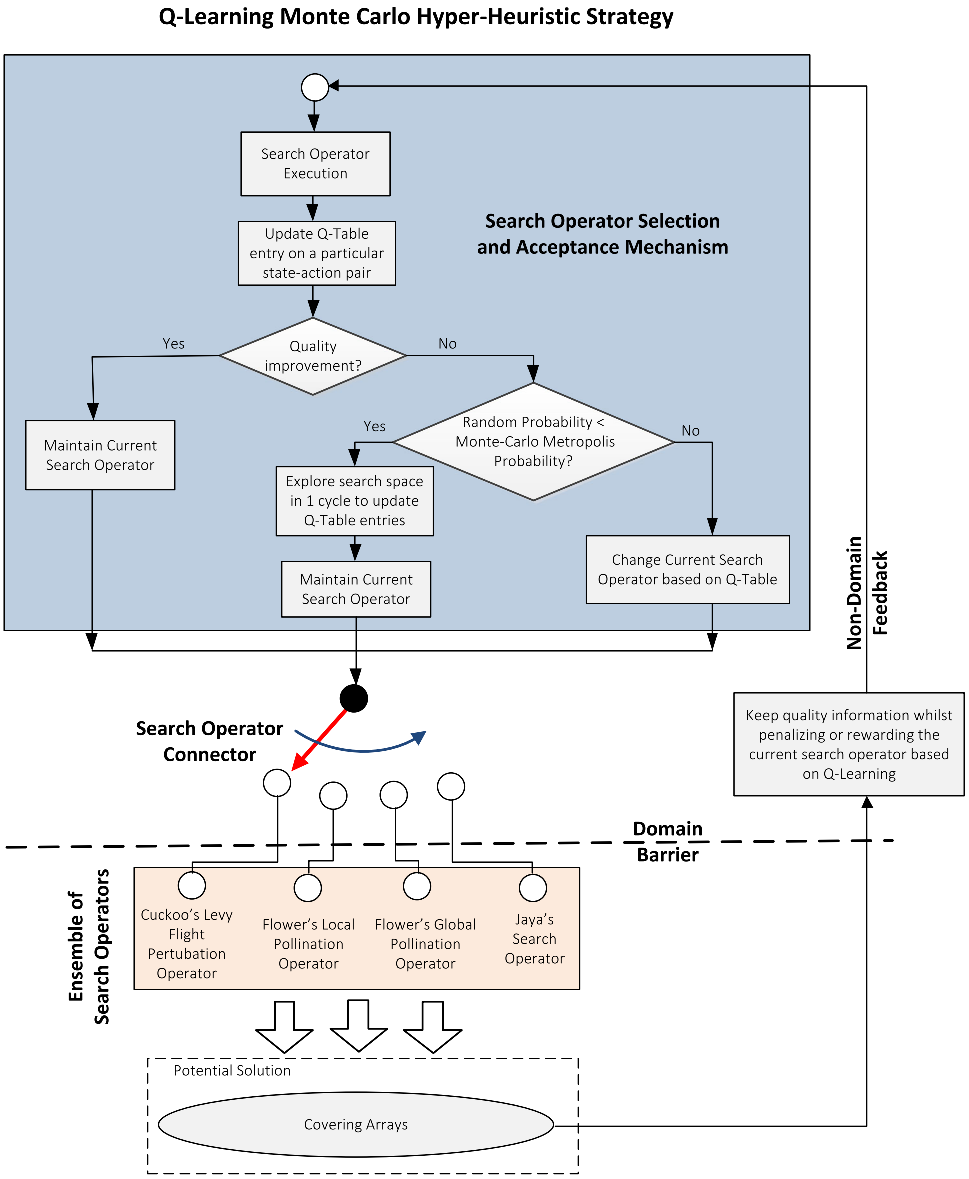

Q-Learning Monte Carlo Hyper-Heuristic Strategy
Search Operator Execution
Update Q-Table entry on a particular state-action pair
Search Operator Selection and Acceptance Mechanism
Quality improvement?
Yes
No
Maintain Current Search Operator
Random Probability < Monte-Carlo Metropolis Probability?
Yes
No
Explore search space in 1 cycle to update Q-Table entries
Maintain Current Search Operator
Change Current Search Operator based on Q-Table
Non-Domain Feedback
Search Operator Connector
Keep quality information whilst penalizing or rewarding the current search operator based on Q-Learning
Domain Barrier
Ensemble of Search Operators
Cuckoo's Levy Flight Pertubation Operator
Flower's Local Pollination Operator
Flower's Global Pollination Operator
Jaya's Search Operator
Potential Solution
Covering Arrays

Fig. 3 High-level view of the proposed hyper-heuristic strategy

ulated annealing, EMCQ does not use any specific cooling schedule; hence, specific parameters do not need to be tuned. Another notable feature is that EMCQ allows dynamic manipulation on its q parameter to increase or decrease the probability of accepting poor moves. q is always incremented upon a poor move and reset to 1 upon a good move to enhance the diversification of the solution.

Although adopting the same cooling schedule as EMCQ, Q-EMCQ has a different reward and punishment mechanism. For EMCQ, the reward is based solely on the previous performance (although sometimes the poor performing operator may also be rewarded based on some probability). Unlike EMCQ, when a poor search operator is penalized, Q-EMCQ chooses the historically best performing operator for the next search instance instead of the available search operators randomly.

Q-learning is a Markov decision process that relies on the current and forward-looking Q-values. It provides the reward and punishment mechanism (Christopher 1992) that dynamically keeps track of the best performing operator via online reinforcement learning. To be specific, Q-learning learns the optimal selection policy by its interaction with the environment. Q-learning works by estimating the best state–action pair through the manipulation of memory based on $Q(s, a)$ table. A $Q(s, a)$ table uses a state–action pair to index a Q-value (i.e., as cumulative reward). The $Q(s, a)$ table is updated dynamically based on the reward and punishment (r) from a particular state–action pair.

Let $S = [s_1, s_2, \ldots, s_n]$ be a set of states, $A = [a_1, a_2, \ldots, a_n]$ be a set of actions, α_t be the learning rate within [0, 1], γ be the discount factor within [0, 1], and r_t be the immediate reward/punishment acquired from executing action a, the $Q(st, at)$ as the cumulative reward at time (t) can be computed as follows:

$$Q_{(t+1)}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t(r_t + \gamma \max(Q_t(s_{(t+1)}, a_{(t+1)})) - Q_t(s_t, a_t)) \tag{2}$$

The optimal setting for t, γ, and r_t needs further clarification. When α_t is close to 1, a higher priority is given to the newly gained information for the Q-table updates. On the contrary, a small value of α_t gives higher priority to the existing information. To facilitate exploration of the search space (to maximize learning from the environment), the value of α_t during early iteration can be set a high value, but adaptively reduce toward the end of the iteration (to exploit the existing best known Q-value) as follows:

$$\alpha_t = 1 - 0.9 \times t/(MaxIteration) \tag{3}$$

The parameter γ works as the scaling factor for rewarding or punishing the Q-value based on the current action. When γ is close to 0, the Q-value is based on the current reward/punishment only. When γ is close to 1, the Q-value will be based on the current and the previous reward/punishment. It is suggested to set $\gamma = 0.8$ Samma et al. (2016).

The parameter r_t serves as the actual reward or punishment value. In our current work, the value of r_t is set based on:

$$\left.\begin{array}{ll} r_t = 1, & \text{if the current action improves fitness} \\ r_t = -1, & \text{otherwise} \end{array}\right\} \tag{4}$$

Based on the discussion above, Algorithm 1 highlights the pseudo-code for Q-EMCQ.

Q-EMCQ involves three main steps, denoted as Steps A, B, and C. Step A deals with the initialization of variables. Line 1 initializes the populations of the required t-wise interactions, $I = I_1, I_2, \ldots, I_M$. The value of M depends on the given inputs interaction strength (t), parameter (k), and its corresponding value (v). M captures the number of required interactions that need to be captured in the constructed CA. M can be mathematically obtained as the sum of products of each individual's t-wise interaction. For example, for $CA(9; 2, 3^4)$, M takes the value of $3 \times 3 + 3 \times 3 + 3 \times 3 + 3 \times 3 + 3 \times 3 + 3 \times 3 = 54$. If $MCA(9; 2, 3^2 2^2)$ is considered, then M takes the value of $3 \times 3 + 3 \times 2 + 3 \times 2 + 3 \times 2 + 3 \times 2 + 2 \times 2 = 37$. Line 2 defines the maximum iteration Θ_{max} and population size, N. Line 3 randomly initializes the initial population of solution $X = X_1, X_2, \ldots, X_M$. Line 4 defines the pool of search operators. Lines 6–14 explore the search space for 1 complete episode cycle to initialize the Q-table.

Step B deals with the Q-EMCQ selection and acceptance mechanism. The main loop starts in line 15 with Θmax as the maximum number of iteration. The selected search operator will be executed in line 17. The Q-table will be updated accordingly based on the quality/performance of the current state–action pairs (lines 18–24). Like EMCQ, the Monte Carlo Metropolis probability controls the selection of search operators when the quality of the solution improves (lines 25–30). This probability decreases with iteration (T). However, it may also increase as the Q-value can be reset to 1 (in the case of re-selection of any particular search operator (lines 29 and 34)). When the quality does not improve, the Q-learning gets a chance to explore the search space in one complete episode cycle (as line 33) to complete the Q-table entries. As an illustration, Fig. 4 depicts the snapshot of one entire Q-table cycle for Q-EMCQ along with a numerical example.

Referring to episode 1 in Fig. 4, assume that the initial settings are as follows: the current state s_t = *Lévy flight perturbation operator*, the next action a_t = *local pollination operator*, the current value stored in the Q-table for the current state $Q_{(t+1)}(s_t, a_t) = 1.25$ (i.e., grayed cell); the punishment $r_t = -1.00$; the discount factor $\gamma = 0.10$; and the current learning factor $\alpha_t = 0.70$. Then, the new value for $Q_{(t+1)}(s_t, a_t)$ in the Q-table is updated based on Eq. 2 as:

$$Q_{(t+1)}(s_t, a_t) = 1.25 + 0.70 \times [-1.00 + 0.10 \times Max(0.00, -1.01, 1.00, -1.05) - 1.25] = -0.26 \tag{5}$$

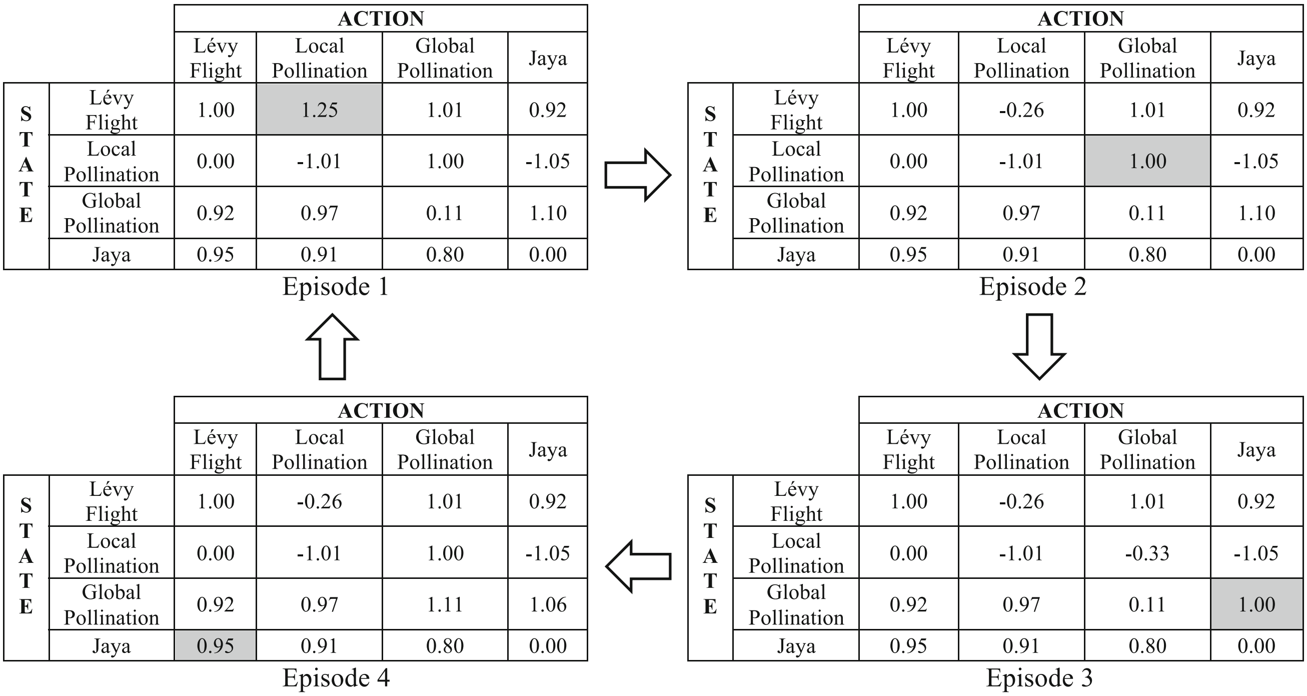
Concerning episode 2 in Fig. 4, the current settings are as follows: the current state s_t= *Local Pollination Operator*, the next action a_t= *Global Pollination Operator*, the current value stored in the Q-table for the current state $Q_{(t+1)}(s_t, a_t) = 1.00$ (i.e., grayed cell); the punishment $r_t = -1.00$; the discount factor $\gamma = 0.10$; and the current learning factor $\alpha_t = 0.70$. Then, the new value for $Q_{(t+1)}(s_t, a_t)$ in the Q-table is updated based on Eq. 2 as:

$$Q_{(t+1)}(s_t, a_t) = 1.00 + 0.70 \times [-1.00 + 0.10 \times \text{Max}(0.92, 0.97, 0.11, 1.00) - 1.00] = -0.33 \tag{6}$$

Considering episode 3 in Fig. 4, the current settings are as follows: the current state s_t = *Global Pollination Operator*, the next action a_t = *Jaya Operator*, the current value stored in the Q-table for the current state $Q_{(t+1)}(s_t, a_t) = 1.00$ (i.e., grayed cell); the reward $r_t = 1.00$; the discount factor $\gamma = 0.10$; and the current learning factor $\alpha t = 0.70$. Then, the new value for $Q_{(t+1)}(s_t, a_t)$ in the Q-table is updated based on Eq. 2 as:

$$Q_{(t+1)}(s_t, a_t) = 1.00 + 0.70 \times [1.00 + 0.10 \times \text{Max}(0.95, 0.91, 0.80, 0.00) - 1.00] = 1.06 \tag{7}$$

The complete exploration cycle for updating Q-values ends in episode 4 as the next action $a_t = s_{(t+1)}$ = *Lévy*



Episode 1

STATE	Lévy Flight	Local Pollination	Global Pollination	Jaya
Lévy Flight	1.00	1.25	1.01	0.92
Local Pollination	0.00	-1.01	1.00	-1.05
Global Pollination	0.92	0.97	0.11	1.10
Jaya	0.95	0.91	0.80	0.00

Episode 2

STATE	Lévy Flight	Local Pollination	Global Pollination	Jaya
Lévy Flight	1.00	-0.26	1.01	0.92
Local Pollination	0.00	-1.01	1.00	-1.05
Global Pollination	0.92	0.97	0.11	1.10
Jaya	0.95	0.91	0.80	0.00

Episode 3

STATE	Lévy Flight	Local Pollination	Global Pollination	Jaya
Lévy Flight	1.00	-0.26	1.01	0.92
Local Pollination	0.00	-1.01	-0.33	-1.05
Global Pollination	0.92	0.97	0.11	1.00
Jaya	0.95	0.91	0.80	0.00

Episode 4

STATE	Lévy Flight	Local Pollination	Global Pollination	Jaya
Lévy Flight	1.00	-0.26	1.01	0.92
Local Pollination	0.00	-1.01	1.00	-1.05
Global Pollination	0.92	0.97	1.11	1.06
Jaya	0.95	0.91	0.80	0.00

Fig. 4 Q-learning mechanism for 1 complete episode cycle

flight perturbation operator. It must be noted that throughout the Q-table updates, the Q-EMCQ search process is also working in the background (i.e., for each update, X_{best} is also kept and the population X is also updated accordingly).

A complete cycle update is not always necessary, especially during convergence. Lines 38–39 depict the search operator selection process as the next action (a_t) (i.e., between Lévy flight perturbation operator, local pollination operator, global pollination operator, and Jaya operator) based on the maximum reward defined in the state–action pair memory within the Q-table (unlike EMCQ where the selection process is random).

Complementing earlier steps, Step C deals with termination and closure. In line 39, upon the completion of the main Θmax loop, the best solution S_{best} is added to the final CA. If uncovered t-wise interaction exists, Step B is repeated until termination (line 41).

4.2 Cuckoo's Levy Flight Perturbation Operator

Cuckoo's Levy flight perturbation operator is derived from the cuckoo search algorithm (CS) Yang and Deb (2009). The complete description of the perturbation operator is summarized in Algorithm 2.

Cuckoo's Levy flight perturbation operator acts as the local search algorithm that manipulates the *Lévy flight* motion. For our *Lévy flight* implementation, we adopt the well-known Mantegna's algorithm Yang and Deb (2009). Within this algorithm, a *Lévy flight* step length can be defined as:

$$\text{Step} = u/[v]^{(1/\beta)} \tag{8}$$

where u and v are approximated from the normal Gaussian distribution in which

$$u \approx N(0, {\sigma_u}^2) \times \sigma_u \qquad v \approx N(0, {\sigma_v}^2) \times \sigma_v \tag{9}$$

For v value estimation, we use $\sigma_v = 1$. For u value estimation, we evaluate the gamma function (Γ) with the value of $\beta = 1.5$ Yang (2008) and obtain σ_u using

$$\sigma_u = \left| \frac{(\Gamma(1+\beta) \times \sin(\pi\beta/2))}{(\Gamma(1+\beta)/2) \times \beta \times 2^{(((\beta-1))/2))}} \right|^{(1/\beta)} \tag{10}$$

In our case, the gamma function (Γ) implementation is adopted from Press et al. (1992). The Lévy flight motion is essentially a random walk that takes a sequence of jumps, which are selected from a heavy-tailed probability function (Yang and Deb 2009). As a result, the motion will produce a series of "aggressive" small and large jumps (either positive or negative), thus ensuring largely diverse values. In our implementation, the Lévy flight motion performs a single value perturbation of the current population of solutions, thus rendering it as a local search operator.

As for the working of the operator, the initial X_{best} is set to X_0 in line 1. The loop starts in line 2. One value from a

Algorithm 1: Pseudo Code for Q-EMCQ

```
    Input: Interaction strength ($t$), parameter ($k$) and its corresponding value ($v$)
    Output: Final covering array, CA
    /* Step A: (Initialization)                                                    */
 1  Initialize the population of the required t-wise interactions, $I = \{I_0, I_1, \ldots, I_M\}$ based on $k$ and $v$ values
 2  Initialize $\Theta_{max}$ iteration and population size N
 3  Initialize the random population of solutions, $X = \{X_0, X_1, \ldots, X_N\}$
 4  Let the pool of search operator $H = \{H_0, H_1, \ldots, H_N\}$
 5  Set $Q_t(s_t, a_t) = 0$ for each state $S = [s_1, s_2, \ldots, s_n]$, and action $A = [a_1, a_2, \ldots, a_n]$
 6  for each state $S = [s_1, s_2, \ldots, s_n]$, and action $A = [a_1, a_2, \ldots, a_n]$ in random order do
 7      From the current state $s_t$, select the best action at from the Q-table
 8      if action $(a_t) == H_i^t$, update $X_i^t$ using $H_i^t$ search operator then
 9          Update the best solution obtained so far, $X_{best} = P_i^t$
10      Get immediate reward/punishment $r_t$ using Eq. 4
11      Get the maximum Q value for the next state $s_{t+1}$
12      Update $\alpha_t$ using Eq.3
13      Update Q-table entry using Eq. 2
14      Update the current state, $s_t = s_{t+1}$
    /* Step B: (Selection and Acceptance)                                          */
15  From the current state $s_t$, select the best action at from the Q-table
16  while $T < \Theta_{max}$ do
17      if action $(a_t) == H_i^t$, update $X_i^t$ using $H_i^t$ search operator then
18          Update the best solution obtained so far, $X_{best} = P_i^t$
19          $H_i^{t'} = H_i^t$
20          Get immediate reward/punishment $r_t$ using Eq. 4
21          Get the maximum Q value for the next state $s_{t+}$
22          Update $\alpha_t$ using Eq. 3
23          Update Q-table entry using Eq. 2
24          Update the current state, $s_t = s_{t+1}$
25      Compute $\delta = f(X_i^t) - f(X_i^{(t-1)})$
26      if $(\delta > 0)$ /* improving fitness, complete episode unnecessary           */
27       then
28          Set q=1 and maintain the best action $a_t = H_i^t$
29      else
30          Compute probability density $\varsigma$ using Eq. 1 /* worsening fitness      */
31          if $random(0, 1) < \varsigma$ then
32              $H_i^{t'} = H_i^t$
33              Redo Steps 6-14, starting with state st /* explore as one complete episode cycle */
34              Set q=1 and reselect the next action $at = H_i^{t'}$
35          else
36              From the current state $s_t$, select the best action at from the Q-table
37              q++
38      T++
    /* Step C: (Termination and Closure)                                           */
39  Add $X_{best}$ to covering array, CA
40  if there are uncovered $t - wise$ interaction in I then
41      Return to Step B
42  else
43      Terminate
```

particular individual X_i is selected randomly (column-wise) and perturbed using α with entry-wise multiplication ($\oplus$) and levy flight motion (L), as indicated in line 4. If the newly perturbed X_i has a better fitness value, then the incumbent is replaced and the value of X_{best} is also updated accordingly (in lines 5–11). Otherwise, X_i is not updated, but X_{best} will be updated based on its fitness against X_i.

4.3 Flower's Local Pollination Operator

As the name suggests, the flower's local pollination operator is derived from the flower algorithm Yang (2012). The complete description of the operator is summarized in Algorithm 3.

In line 1, XS_{best} is initially set to X_0. In line 2, two distinct peer candidates X_p and X_q are randomly selected from the

```
Algorithm 2: Pseudo Code for Cuckoo's Levy Flight Perturbation Operator
    Input: the population $X = \{X_0, X_1, \ldots, X_M\}$
    Output: $X_{best}$ and the updated population $X^{'} = \{X_0^{'}, X_1^{'}, \ldots, X_M^{'}\}$
 1  $X_{best} = X_0$
 2  for $i = 0$ to population size, $M$ do
 3      Generate a step vector Ł which obeys Levy Flight distribution
 4      Perturbate one value from random column wise, $X_i^{t+1} = X_i^t + \alpha \bigoplus \text{Ł}\ with\ \alpha = 1$
 5      if $f(X_i^{(t+1)}) > f(X_i^{(t)})$ then
 6          $X_i^{(t)} = X_i^{(t+1)}$
 7          if $(f(X_i^{(t+1)}) > f(X_{best}))$ then
 8              $X_{best} = X_i^{(t+1)}$
 9      else
10          if $(f(X_i^{(t)}) > f(X_{best})$ then
11              $X_{best} = X_i^{(t)}$
12  Return $S_{best}$
```

```
Algorithm 3: Flower's Local Pollination Operator
    Input: the population $X = \{X_0, X_1, \ldots, X_M\}$
    Output: $X_{best}$ and the updated population $X^{'} = \{X_0^{'}, X_1^{'}, \ldots, X_M^{'}\}$
 1  $X_{best} = X_0$
 2  for $i = 0$ to population size, $S - 1$ do
 3      Choose $X_p$ and $X_q$ randomly from X, where $j \neq k$
 4      Set $\gamma = random\ (0, 1)$
 5      Update the current population $X_i^{(t+1)} = X_i^{(t)} + \gamma(X_p^{(t)} - X_q^{(t)})$
 6      if $(f(X_i^{(t+1)}) > f(X_i^{(t)}))$ then
 7          $X_i^{(t)} = X_i^{(t+1)}$
 8          if $(f(X_i^{(t+1)}) > f(X_{best}))$ then
 9              $X_{best} = X_i^{(t+1)}$
10      else
11          if $(f(X_i^{(t)}) > f(X_{best}))$ then
12              $X_{best} = X_i^{(t)}$
13  Return $S_{best}$
```

current population X. The loop starts in line 2. Each X_i will be iteratively updated based on the transformation equation defined in lines 4–5. If the newly updated X_i has better fitness value, then the current X_i is replaced accordingly (in lines 6–7). The value of X_{best} is also updated if it has a better fitness value than that of X_i (in lines 8–10). When the newly updated X_i has poorer fitness value, no update is made to X_i, but X_{best} will be updated if it has better fitness than X_i (in lines 11–12).

4.4 Flower's global pollination operator

Flower's global pollination operator (Yang 2012) is summarized in Algorithm 4 and complements the local pollination operator described earlier.

Similar to cuckoo's Levy flight perturbation operator described earlier, the global pollination operator also exploits Levy flight motion to generate a new solution. Unlike the former operator, the transformation equation for flower's global pollination operator uses the Levy flight to update all the (column-wise) values for Z_i of interest instead of only perturbing one value, thereby making it a global search operator.

Considering the flow of the global pollination operator, X_{best} is initially set to X_0 in line 1. The loop starts in line 2. The value of X_i will be iteratively updated by using the transformation equation that exploits exploiting Levy flight motion (in lines 4–5). If the newly updated X_i has better fitness value, then the current X_i is replaced accordingly (in lines 6–7). The value of X_{best} is also updated if it has a better fitness value than that of X_i (in lines 8–10). If the newly updated X_i has poorer fitness value, no update is made to X_i. X_{best} will be updated if it has better fitness than X_i (in lines 8–10 and lines 11–12).

```
Algorithm 4: Flower's Global Pollination Operator
Input: the population $X = \{X_0, X_1, \ldots, X_M\}$
Output: $X_{best}$ and the updated population $X' = \{X'_0, X'_1, \ldots, X'_M\}$
$X_{best} = X_0$
for $i = 0$ to population size, $M$ do
    Set scaling factor $\rho = random(0, 1)$
    Generate a step vector Ł which obeys Levy Flight distribution
    Update the current population $X_i^{(t+1)} = X_i^{(t)} + \rho \cdot Ł \cdot (X_{best} - X_i^{(t)})$
    if $(f(X_i^{(t+1)}) > f(X_i^{(t)}))$ then
        $X_i^{(t)} = X_i^{(t+1)}$
        if $(f(X_i^{(t+1)}) > f(X_{best}))$ then
            $X_{best} = X_i^{(t+1)}$
    else
        if $(f(X_i^{(t)}) > f(X_{best}))$ then
            $X_{best} = X_i^{(t)}$
Return $X_{best}$
```

4.5 Jaya search operator

The Jaya search operator is derived from the Jaya algorithm Rao (2016). The complete description of the Jaya operator is summarized in Algorithm 5.

Unlike the search operators described earlier (i.e., keeping track of only X_{best}), the Jaya search operator keeps track of both X_{best} and X_{poor}. As seen in line 6, the Jaya search operator exploits both X_{best} and X_{poor} as part of its transformation equation. Although biased toward the global search for Q-EMCQ in our application, the transformation equation can also address local search. In the case when $\mathit{\Delta} X = X_{best} - X_{poor}$ is sufficiently small, the transformation equation offset (in line with the term $\mho(X_{best} - X_i) - \zeta(X_{poor} - X)$) will be insignificant relative to the current location of X_i allowing steady intensification.

As far as the flow of the Jaya operator is concerned, lines 1–2 set up the initial values for $X_{best} = X_0$ and $X_{poor} = X_{best}$. The loop starts from line 3. Two random values $\mho$ and ζ are generated to compensate and scale down the delta differences between X_i with X_{best} and X_{poor} in the transformation equation (in lines 4–5). If the newly updated X_i has a better fitness value, then the current X_i is replaced accordingly (in lines 7–8). Similarly, the value of X_{best} is also updated if it has a better fitness value than that of X_i (in lines 9–11). In the case in which the newly updated X_i has poorer fitness value, no update is made to X_i. If the fitness of the current X_i is better than that of X_{best}, X_{best} is assigned to X_i (in lines 12–13). Similarly, if the fitness of the current X_i is poorer than that of X_{poor}, X_{poor} is assigned to X_i (in lines 14–15).

5 Empirical study design

We have put our strategy under extensive evaluation. The goals of the evaluation experiments are threefold: (1) to investigate how Q-EMCQ fares against its own predecessor EMCQ, (2) to benchmark Q-EMCQ against well-known strategies for t-wise test suite generation, (3) to undertake the effectiveness assessment of Q-EMCQ using t-wise criteria in terms of achieving branch coverage as well as revealing mutation injected faults based on real-world industrial applications, (4) to undertake the efficiency assessment of Q-EMCQ by comparing the test generation cost with manual testing, and (5) to compare the performance of Q-EMCQ with contemporary meta-heuristics and hyper-heuristics.

In line with the goals above, we focus on answering the following research questions:

- RQ1: In what ways does the use of Q-EMCQ improve upon EMCQ?
- RQ2: How good is the efficiency of Q-EMCQ in terms of test suite minimization when compared to the existing strategies?
- RQ3: How good are combinatorial tests created using Q-EMCQ and 2-wise, 3-wise, and 4-wise at covering the code?
- RQ4: How effective are the combinatorial tests created using Q-EMCQ for 2-wise, 3-wise, and 4-wise at detecting injected faults?
- RQ5: How does Q-EMCQ with 2-wise, 3-wise, and 4-wise compare with manual testing in terms of cost?
- RQ6: Apart from minimization problem (i.e., t-wise test generation), is Q-EMCQ sufficiently general to solve (maximization) optimization problem (i.e., module clustering)?

Algorithm 5: Jaya Search Operator

```
Input: the population $X = \{X_0, X_1, \ldots, X_M\}$
Output: $X_{best}$ and the updated population $X^{'} = \{X_0^{'}, X_1^{'}, \ldots, X_M^{'}\}$
$X_{best} = X_0$
$X_{poor} = X_{best}$
for $i = 0$ to population size, $M$ do
    Set $\varphi = random(0, 1)$
    Set $\zeta = random(0, 1)$
    Update the current population $X_i^{(t+1)} = X_i^{(t)} + \varphi \cdot (X_{best} - X_i^{(t)}) - \zeta \cdot (X_{poor} - X_i^{(t)})$
    if $(f(X_i^{(t+1)}) > f(X_i^{(t)})$ then
        $X_i^{(t)} = X_i^{(t+1)}$
        if $(f(X_i^{(t+1)}) > f(X_{best}))$ then
            $X_{best} = X_i^{(t+1)}$
    else
        if $(f(X_i^{(t)}) > f(X_{best}))$ then
            $X_{best} = X_i^{(t)}$
        if $(f(X_i^{(t)}) < f(X_{poor}))$ then
            $X_{poor} = X_i^{(t)}$
Return $X_{best}$
```

5.1 Experimental Benchmark setup

We adopt an environment consisting of a machine running Windows 10, with a 2.9 GHz Intel Core i5 CPU, 16 GB 1867 MHz DDR3 RAM, and 512 GB flash storage. We set the population size of $N = 20$ with a maximum iteration value $\theta_{\max} = 2500$. While such a choice of population size and maximum iterations could result in more than 50,000 fitness function evaluations, we limit our maximum fitness function evaluation to 1500 only (i.e., the Q-EMCQ stops when the fitness function evaluation reaches 1500). This is to ensure that we can have a consistent value of fitness function evaluation throughout the experiments (as each iteration can potentially trigger more than one fitness function evaluation). For statistical significance, we have executed Q-EMCQ for 20 times for each configuration and reported the best results during these runs.

5.2 Experimental Benchmark Procedures

For RQ1, we arbitrarily select 6 combinations of covering arrays CA $(N; 2, 4^2 2^3)$, CA $(N; 3, 5^2 4^2 3^2)$, CA $(N; 4, 5^1 3^2 2^3)$, MCA $(N; 2, 5^1 3^3 2^2)$, MCA $(N, 3, 6^1 5^1 4^3 3^3 2^3)$ and MCA $(N, 4, 7^1 6^1 5^1 4^3 3^3 2^3)$. Here, the selected covering arrays span both uniform and non-uniform number of parameters. To ensure a fair comparison, we re-implement EMCQ using the same data structure and programming language (in Java) as Q-EMCQ before adopting it for covering array generation. Our EMCQ re-implementation also rides on the same low-level operators (i.e., cuckoo's Levy flight perturbation operator, flower algorithm's local pollination, and global pollination operator as well as Jaya's search operator). For this reason, we can fairly compare both test sizes and execution times.

For RQ2, we adopted the benchmark experiments mainly from Wu et al. (2015). In particular, we adopt two main experiments involving CA $(N; t, v^7)$ with variable values $2 \leq v \leq 5$, t varied up to 4 as well as CA $(N; t, 3^k)$ with variable number of parameters $3 \leq k \leq 12$, t varied up to 4. We have also compared our strategy with those published results for those strategies that are not freely available to download. Parts of those strategies depend mainly on meta-heuristic algorithms, specifically HSS, PSTG, DPSO, ACO, and SA. The other part of those strategies is dependent on exact computational algorithms, specifically PICT, TVG, IPOG, and ITCH. We represent all our results in the tables where each cell represents the smallest size (marked as bold) generated by its corresponding strategy. In the case of Q-EMCQ, we also reported the average sizes to give a better indication of its efficiency. We opt for generated size comparison and not time because all of the strategies of interest are not available to us. Even if these strategies are available, their programming languages and data structure implementations are not the same renderings as an unfair execution time comparison. Often, the size comparison is absolute and is independent of the implementation language and data structure implementation.

For answering RQ3–RQ5, we have selected a train control management system that has been in development for a couple of years. The system is a distributed control software with multiple types of software and hardware components for operation-critical and safety-related supervisory behavior of the train. The program runs on programmable logic controllers (PLCs), which are commonly used as real-time controllers used in industrial domains (e.g., manufacturing and avionics); 37 industrial programs have been provided for which we applied the Q-EMCQ approach for minimizing the t-wise test suite.

Concerning RQ6, we have selected three public domain class diagrams available freely in the public domains involving Credit Card Payment System (CCPS) Cheong et al. (2012), Unified Inventory University (UIU) Sobh et al. (2010), and Food Book (FB)[1] as our module case studies. Here, we have adopted the Q-EMCQ approach for maximizing the number of clusters so that we can have the best modularization quality (i.e., best clusters) for all given three systems' class diagrams.

For comparison purposes, we have adopted two groups of comparison. In the first group, we adopt EMCQ as well as modified choice function (Pour Shahrzad et al. 2018) and Tabu search HHH Zamli et al. (2016) implementations. It should be noted that all the hyper-heuristic rides on the same operators (i.e., Lévy flight, local pollination, global pollination, and Jaya). In the second group, we have decided to adopt the TLBO Praditwong et al. (2011), SCA Mirjalili (2016) and SOS Cheng and Prayogo (2014) implementations. Here, we are able to fairly compare the modularization quality as well as execution time as the data structure, language implementation and the running system environment are the same (apart from the same number of maximum fitness function evaluation). It should be noted that these algorithms (i.e., TLBO, SCA, SOS) do not have any parameter controls apart from population size and maximum iteration. Hence, their adoption does not require any parameter calibrations.

5.3 Case study object

As highlighted earlier, we adopt two case study objects involving the train control management system as well as the module clustering of class diagrams.

5.3.1 Train control management system

We have conducted our experiment on programs from a train control management system running on PLCs that have been developed for a couple of years. A program running on a PLC executes in a loop in which every cycle contains the reading of input values, the execution of the program without interruptions, and the update of the output variables. As shown in Fig. 5, predefined logical and/or stateful blocks (e.g., bistable latch SR, OR, XOR, AND, greater-than GT, and timer TON) and connections between blocks represent the behavior of a PLC program written in the Function Block Diagram (FBD) programming language (John and Tiegelkamp 2010). A hardware manufacturer supplies these blocks or is developed using custom functions. PLCs contain particular types of blocks, such as timers (e.g., TON) that provide the same functions as timing relays and are used to activate or deactivate a device after a preset interval of time. There are two different timer blocks: (1) on-delay timer (TON) and (2) off-delay timer (TOF). A timer block keeps track of the number of times its input is either true or false and outputs different signals. In practice, many other timing configurations can be derived from these basic timers. An FBD program is translated to a compliant executable PLC code. For more details on the FBD programming language and PLCs, we refer the reader to the work of John and Tiegelkamp (2010).

We experimented with 37 industrial FBD programs for which we applied the Q-EMCQ approach. These programs contain ten input parameters and 1209 lines of code on average per program.

To answer our research questions, we generated test cases using Q-EMCQ for 2-wise, 3-wise, and 4-wise and executed each program on these test cases to collect branch coverage and fault detection scores for each test suite as well as the number of test cases created. A *test suite* created for a PLC program contains a set of *test cases* containing inputs, expected and actual outputs together with timing constraints.

Test Case Generation and Manual Testing We used test suites automatically generated using Q-EMCQ. To do this, we asked an engineer from Bombardier Transportation Sweden AB, responsible for developing and testing the PLC programs used in this study, to identify the range parameter values for each input variable and constraints. We used the collected input parameter ranges for each input variable for generating combinatorial test cases using Q-EMCQ. These ranges and constraints were also used for creating manual test suites. We collected the number of test cases for each manual test suite created by engineers for each of the programs used in this case study. In testing these PLC programs, the testing processes are performed according to safety standards and certifications, including rigorous specification-based testing based on functional requirements expressed in natural language. As the programs considered in this study are manually tested and are part of a delivered project, we expect that the number of test cases created manually by experienced industrial engineers to be a realistic proxy measure of the level of efficiency needed to test these PLC programs thoroughly.

Measuring Branch Coverage Code coverage criteria are used in practice to assess the extent to which the PLC program has been covered by test cases (Ammann and Offutt 2008). Many criteria have been proposed in the literature, but in this study, we only focus on branch coverage criteria. For the PLC programs used in this study, the engineers developing software indicated that their certification process involves achieving high branch coverage. A branch coverage score

[1] https://bit.ly/2XDPOPB.

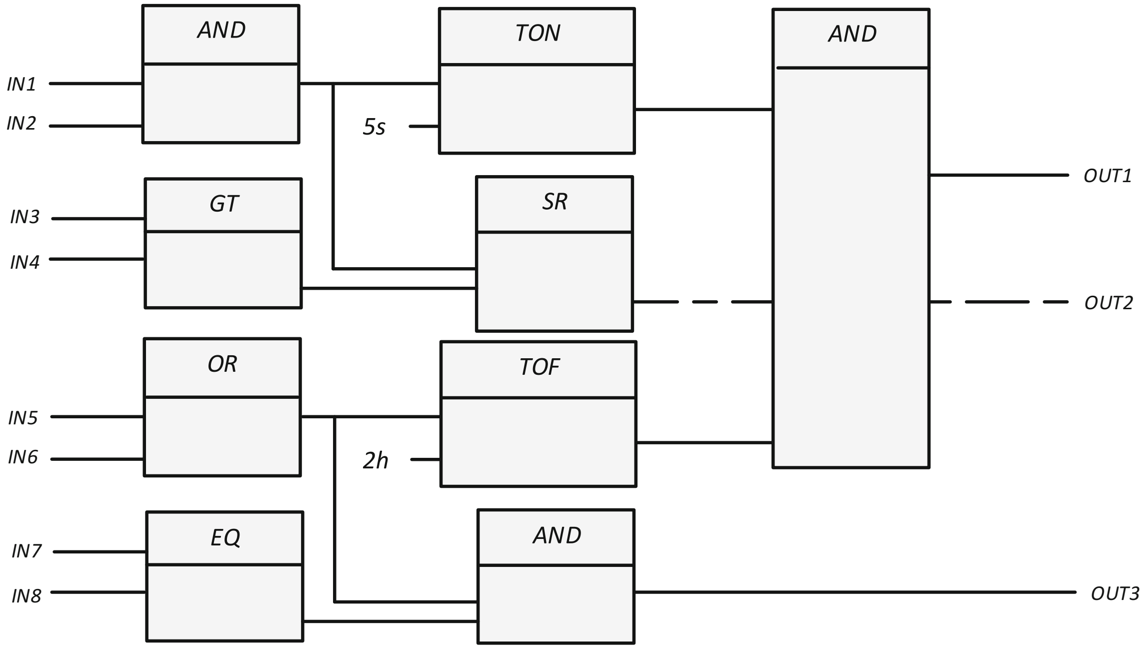


Fig. 5 An example of a PLC control program written using the FBD programming language

was obtained for each test suite. A test suite satisfies decision coverage if running the test cases causes each branch in the program to have the value *true* at least once and the value *false* at least once.

Measuring Fault Detection Fault detection was measured using mutation analysis by generating faulty versions of the PLC programs. Mutation analysis is used in our case study by creating faulty implementations of a program in an automated manner to examine the fault detection ability of a test case (DeMillo et al. 1978). A mutated program is a new version of the original PLC program created by making a small change to this original program. For example, in a PLC program, a mutated program is created by replacing an operator with another, negating an input variable, or changing the value of a constant to another interesting value. If the execution of a test suite on the mutated program gives a different observable behavior as the original PLC program, the test case kills that mutant. We calculated the mutation score using an output-only oracle against all the created mutated programs. For all programs, we assessed the mutation detection capability of each test case by calculating the ratio of mutated programs killed to the total number of mutated programs. Researchers (Just et al. (2014); Andrews et al. (2005)) investigated the relation between real fault detection and mutant detection, and there is some strong empirical evidence suggesting that if a test case can detect or kill most mutants, it can also be good at detecting naturally occurring faults, thus providing evidence that the mutation score is a fairly good proxy measure for fault detection.

In the creation of mutants, we rely on previous studies that looked at using mutation analysis for PLC software (Shin et al. 2012; Enoiu et al. 2017). We used the mutation operators proposed in Enoiu et al. (2017) for this study. The following mutation operators were used:

- *Logic Block Replacement Operator (LRO)* Replacing a logical block with another block from the same category (e.g., replacing an AND block with an XOR block in Fig. 5).
- *Comparison Block Replacement Operator (CRO)* Replacing a comparison block with another block from the same category (e.g., replacing a greater-than (GT) block with a greater-or-equal (GE) block in Fig. 5).
- *Arithmetic Block Replacement Operator (ARO)* Replacing an arithmetic block with another block from the same functional category (e.g., replacing a maximum (MAX) block with an addition (ADD) block).
- *Negation Insertion Operator (NIO)* Negating an input or output connection between blocks (e.g., a variable var becomes NOT(var)).
- *Value Replacement Operator (VRO)* Replacing a value of a constant variable connected to a block (e.g., replacing a constant value (var = 5) with its boundary values (e.g., var = 6, var = 4)).
- *Timer Block Replacement Operator (TRO).* Replacing a timer block with another block from the same timer category (e.g., replacing a timer-off (TOF) block with a timer-On (TON) block in Fig. 5).

To generate mutants, each of the mutation operators was systematically applied to each program wherever possible. In total, for all of the selected programs, 1368 mutants (faulty programs based on ARO, LRO, CRO, NIO, VRO, and TRO operators) were generated by automatically introducing a single fault into the program.

Measuring Cost Leung and White (1991) proposed the use of a cost model for comparing testing techniques by using direct and indirect testing costs. A direct cost includes the engineer's time for performing all activities related to testing, but also the machine resources such as the test environment and testing tools. On the other hand, indirect cost includes test process management and tool development. To accurately measure the cost effort, one would need to measure the direct and indirect costs for performing all testing activities. However, since the case study is performed a postmortem on a system that is already in use and for which the development is finished, this type of cost measurement was not feasible. Instead, we collected the number of test cases generated by Q-EMCQ as a proxy measure for the cost of testing. We are interested in investigating the cost of using the Q-EMCQ approach in the same context as manual testing. In this case study, we consider that costs are related to the number of test cases. The higher the number of test cases, the higher is the respective test suite cost. We assume this relationship to be linear. For example, a complex program will require more effort for understanding, and also more tests than a simple program. Thus, the cost measure is related to the same factor—the complexity of the software which will influence the number of test cases. Analyzing the cost measurement results is directly related to the number of test cases giving a picture of the same effort per created test case. In addition to the number of test cases measure, other testing costs are not considered, such as setting up the testing environment and tools, management overhead, and the cost of developing new tests. In this work, we restrict our analysis to the number of test cases created in the context of our industrial case study.

5.3.2 Module clustering of class diagrams

The details of the three class diagrams involved are:

- Credit Card Payment System (CCPS) Cheong et al. (2012) consists of 14 classes interlink with 20 two-way associations and 1 aggregation relationship (refer to Fig. 9a).
- Unified Inventory University (UIU) Sobh et al. (2010) consists of 19 classes interlink with 28 aggregations, 1 2-wise associations and 1 dependency relationship (refer to Fig. 10a).
- Food Book (FB)[2] consists of 31 interlinked classes with 25 2-wise associations, 7 generalizations, and 6 aggregations clustered into 3 packages (refer to Fig. 11a).

Module clustering problem involves partitioning a set of modules into clusters based on the concept of coupling (i.e., measuring the dependency between modules) and cohesion (i.e., measuring the internal strength of a module cluster). The higher the coupling, the less readable the piece of code will be, whereas the higher the cohesion, the better to code organization will be. To allow its quantification, Praditwong et al. (2011) define modularization quality(MQ) as the sum of the ratio of intra-edges and inter-edges in each cluster, called modularization factor (MF_k) for cluster k based on the use of module dependency graph such as the class diagram. Mathematically, MF_k can be formally expressed as in Eq. 11:

$$\mathrm{MF}_k = \begin{cases} 0 & \text{if } i = 0 \\ \frac{i}{i+\frac{1}{2}j} & \text{if } i > 0 \end{cases} \tag{11}$$

where i is the weight of intra-edges and j is that of inter-edges. The term $\frac{1}{2}j$ is to split the penalty of inter-edges across the two clusters that are connected by that edge. The MQ can then be calculated as the sum of MF_k as follows:

$$\mathrm{MQ} = \sum_{n}^{k=1} \mathrm{MF}_k \tag{12}$$

where n is the number of clusters, and it should be noted that maximizing MQ does not necessarily mean maximizing the clusters.

6 Case study results

The case study results can be divided into two parts: for answering RQ1–RQ5 and for answering RQ6.

6.1 Answering RQ1–RQ5

This section provides an analysis of the data collected in this case study, including the efficiency of Q-EMCQ and the effectiveness of using combinatorial interaction testing of different strengths for industrial control software. For each program and each generation technique considered in this study, we collected the produced test suites (i.e., 2-wise stands for Q-EMCQ generated test suites using pairwise combinations, 3-wise is short for test suites generated using Q-EMCQ and 3-wise interactions and 4-wise stands for generated test suites using Q-EMCQ and 4-wise interactions).

[2] https://bit.ly/2XDPOPB.

The overall results of this study are summarized in the form of boxplots in Fig. 7. Statistical analysis was performed using the R software (R-Project 2005).

As our observations are drawn from an unknown distribution, we evaluate if there is any statistical difference between 2-wise, 3-wise, and 4-wise without making any assumptions on the distribution of the collected data. We use a Wilcoxon–Mann–Whitney U-test (Howell 2012), a nonparametric hypothesis test for determining if two populations of data samples are drawn at random from identical populations. This statistical test was used in this case study for checking if there is any statistical difference among each measurement metric. Besides, the Vargha–Delaney test (Vargha and Delaney 2000) was used to calculate the standardized effect size, which is a nonparametric magnitude test that shows significance by comparing two populations of data samples and returning the probability that a random sample from one population will be larger than a randomly selected sample from the other. According to Vargha and Delaney (2000), statistical significance is determined when the obtained effect size is above 0, 71 or below 0, 29.

For each measure, we calculated the effect size of 2-wise, 3-wise, and 4-wise and we report in Table 5 the p values of these Wilcoxon–Mann–Whitney U-tests with statistically significant effect sizes shown in bold.

RQ1: In what ways does the use of Q-EMCQ improve upon EMCQ?

Table 1 highlights the results for both Q-EMCQ and EMCQ results involving the 3 combinations of mixed covering arrays MCA $(N; 2, 5^1 3^3 2^2)$, MCA $(N; 3, 5^2 4^2 3^2)$, and MCA $(N; 4, 5^1 3^2 2^3)$.

Referring to Table 1, we observe that Q-EMCQ has outperformed EMCQ as far as the average test suite size is concerned in all three MCAs. As for the time performances, EMCQ is better than Q-EMCQ, notably because there is no overhead as far as maintaining the Q-learning table.

To investigate the performance of Q-EMCQ and EMCQ further, we plot the convergence profiles for the 20 runs for the three covering arrays, as depicted in Fig. 6a to Fig. 6c. At a glance, visual inspection indicates no difference as far as average convergence is concerned. Nonetheless, when we zoom in all the figures (on the right of Fig. 6a to Fig. 6c), we notice that Q-EMCQ has better average convergence than EMCQ.

RQ2: How good is the efficiency of Q-EMCQ in terms of test suite minimization when compared to the existing strategies?

Tables 2 and 3 highlight the results of two main experiments involving CA $(N; t, v^7)$ with variable values $2 \le v \le 5$, t varied up to 4 as well as CA $(N; t, 3^k)$ with variable number of parameters $3 \le k \le 12$, t varied up to 4. In general, the authors of the strategies used in our experimental comparisons only provide the best solution quality, in terms of the size N, achieved by them. Thus, these strategies cannot be statistically compared with Q-EMCQ.

As seen in Tables 2 and 3, the solution quality attained by Q-EMCQ is very competitive with respect to that produced by the state-of-the-art strategies. In fact, Q-EMCQ is able to match or improve on 7 out of 16 entries in Table 2 (i.e., 43.75%) and 20 out of 27 entries in Table 3 (i.e., 74.07%), respectively. The closest competitor is that of DPSO which scores 6 out of 16 entries in Table 2 (i.e., 37.50%) and 19 out of 27 entries in Table 3 (i.e., 70.37%). Regarding the computational effort, as the strategies used in our comparisons adopt different running environments, data structures, and implementation languages, these algorithms cannot be directly compared with ours.

RQ3: How good are combinatorial tests created using Q-EMCQ for 2-wise, 3-wise and 4-wise at covering the code?

In Table 4, we present the mutation scores, code coverage results, and the number of test cases in each collected test suite (i.e., 2-wise, 3-wise, and 4-wise generated tests). This table lists the minimum, maximum, median, mean, and standard deviation values. To give an example, 2-wise created test suites found an average mutation score of 52%, while 4-wise tests achieved an average mutation score of 60%. This shows a considerable improvement in the fault-finding capability obtained by 4-wise test suites over their 2-wise counterparts. For branch coverage, combinatorial test suites are not able to reach or come close to achieving 100% code coverage on most of the programs considered in this case study.

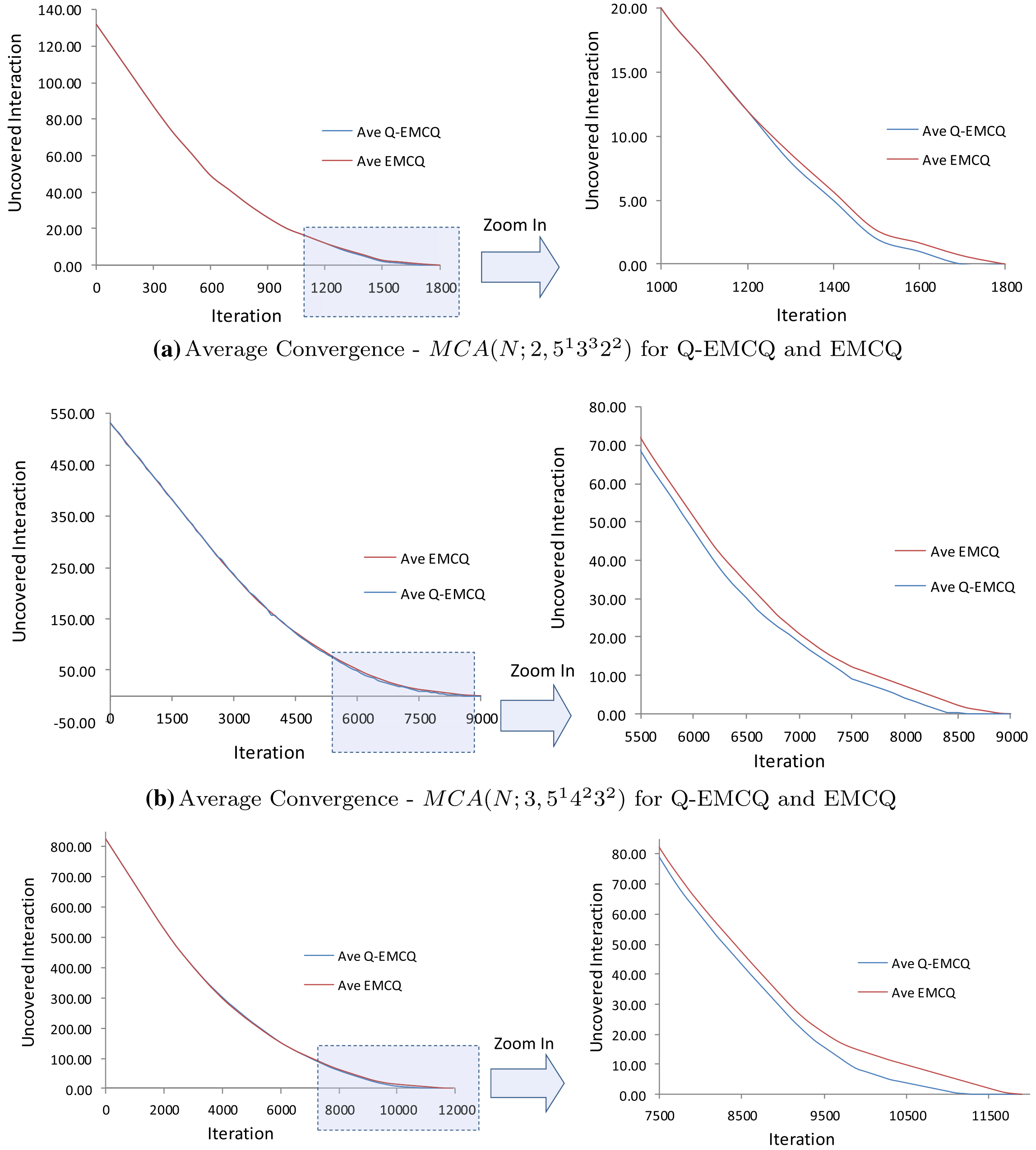
As seen in Fig. 7b, for the majority of programs considered, combinatorial test suites achieve at least 50% branch coverage. 2-wise test suites achieve lower branch coverage scores (on average 84%) than 3-wise test suites (on average 86%). The coverage achieved by combinatorial test suites using 4-wise is ranging between 50% and 100% with a median branch coverage value of 90%.

As seen in Fig. 7b, the use of combinatorial testing achieves between 84% and 88% branch coverage on average. Results for all programs (in Table 5) show that differences in code coverage achieved by 2-wise versus 3-wise and 4-wise test suites are not strong in terms of any significant statistical difference (with an effect size of 0.4). Even if automatically generated test suites are created by having the purpose of covering up to 4-wise input combinations, these test suites are not missing some of the branches in the code. The results are matching our expectations: combinatorial test suites achieve high code coverage to automatically generated

Table 1 Size and time comparison for Q-EMCQ and its predecessor EMCQ

MCA	Q-EMCQ				EMCQ			
	Size		Time (sec)		Size		Time (sec)	
	Best	Ave	Best	Ave	Best	Ave	Best	Ave
MCA (N; 2, $5^1 3^3 2^2$)	**15**	17.00	11.53	12.55	17	17.56	9.29	11.35
MCA (N; 3 , $5^1 4^2 3^2$)	**83**	86.10	53.93	8.14	84	86.50	42.92	46.49
MCA (N; 4, $5^1 3^2 2^3$)	**99**	111.50	107.15	134.10	03	112.80	91.05	10.36

The bold numbers show the best results obtained



(a) Average Convergence - $MCA(N;2,5^1 3^3 2^2)$ for Q-EMCQ and EMCQ

(b) Average Convergence - $MCA(N;3,5^1 4^2 3^2)$ for Q-EMCQ and EMCQ

(c) Average Convergence - $MCA(N;4,5^1 3^2 2^3)$ for Q-EMCQ and EMCQ

Fig. 6 Average convergences for Q-EMCQ and EMCQ for different CAs

Table 2 CA $(N; t, v^7)$ with variable values $2 \leq v \leq 5$, with t varied up to 4

		Meta-heuristic-based strategies						Other Strategies					
		Q-EMCQ		1HSS	1PSTG	1CS	1DPSO	1Jenny	1TConfig	1ITCH	1PICT	1TVG	1IPOG
T	*v*	Best	Ave										
2	2	7	7.00	7	**6**	**6**	7	8	7	**6**	7	7	8
	3	**14**	15.35	**14**	15	15	**14**	16	15	15	16	15	17
	4	**23**	24.6	25	26	25	24	28	28	28	27	27	28
	5	35	35.9	35	37	37	**34**	37	40	45	40	42	42
3	2	15	15.0	**12**	13	**12**	15	14	16	13	15	15	19
	3	49	50.1	50	50	49	49	51	55	**45**	51	55	57
	4	**112**	115.4	121	116	117	**112**	124	**112**	**112**	124	134	208
	5	**216**	220.1	223	225	223	**216**	236	239	225	241	260	275
4	2	**27**	32.2	29	29	**27**	34	31	36	40	32	31	48
	3	**148**	153.55	155	155	155	150	169	166	216	168	167	185
	4	482	485.05	500	487	487	**472**	517	568	704	529	559	509
	5	**1148**	1162.40	1174	1176	1171	**1148**	1248	1320	1750	1279	1385	1349

The bold numbers show the best results obtained

Table 3 CA $(N; t, 3^k)$ with variable number of parameters $3 \leq k \leq 12$, with t varied up to 4

		Meta-heuristic-based strategies						Other Strategies					
		Q-EMCQ		HSS	PSTG	CS	DPSO	Jenny	TConfig	ITCH	PICT	TVG	IPOG
T	*k*	Best	Ave										
2	3	**9**	9.80	**9**	**9**	**9**	**9**	**9**	10	**9**	10	10	11
	4	**9**	9.00	**9**	**9**	**9**	**9**	13	10	**9**	13	12	12
	5	**11**	11.35	12	12	**11**	**11**	14	14	15	13	13	14
	6	**13**	14.20	**13**	**13**	**13**	14	15	15	15	14	15	15
	7	**14**	15.00	15	15	**14**	15	16	15	15	16	15	17
	8	**15**	15.60	**15**	**15**	**15**	**15**	17	17	**15**	16	**15**	17
	9	**15**	16.30	17	17	16	**15**	18	17	**15**	17	**15**	17
	10	16	16.90	17	17	17	16	19	17	**15**	18	16	20
	11	17	17.75	17	17	18	17	17	20	**15**	18	16	20
	12	16	17.95	18	18	18	16	19	20	**15**	19	16	20
3	4	**27**	29.45	30	30	28	**27**	34	32	**27**	34	34	39
	5	**38**	41.25	39	39	**38**	41	40	40	45	43	41	43
	6	**33**	39.00	45	45	43	**33**	51	48	45	48	49	53
	7	**48**	50.80	50	50	48	**48**	51	55	**45**	51	55	57
	8	**51**	53.65	54	54	53	52	58	58	**45**	59	60	63
	9	**56**	57.85	59	58	58	**56**	62	64	75	63	64	65
	10	**59**	61.25	62	62	62	**59**	65	68	75	65	68	68
	11	**63**	64.45	66	64	66	**63**	65	72	75	70	69	76
	12	66	67.45	67	67	70	**65**	68	77	75	72	70	76
4	5	**81**	86.5	94	96	94	**81**	109	97	153	100	105	115
	6	**131**	133.5	132	133	132	**131**	140	141	153	142	139	181
	7	**150**	153.3	154	155	154	**150**	169	166	216	168	172	185
	8	173	175.15	174	175	173	**171**	187	190	216	189	192	203
	9	**167**	188.65	195	195	195	187	206	213	306	211	215	238
	10	207	209.45	212	210	211	**206**	221	235	336	231	233	241
	11	**221**	225.05	223	222	229	**221**	236	258	348	249	250	272
	12	238	240.35	244	244	253	**237**	252	272	372	269	268	275

The bold numbers show the best results obtained

Table 4 Results for each measure: mutation score, branch coverage score, and the cost in terms of the number of test cases

Measure	Test technique	Minimum	Maximum	Median	Mean	SD
Mutation score (%)	2-wise	0, 0	100, 0	48, 4	52, 3	34, 7
	3-wise	0, 0	100, 0	55, 5	57, 2	34, 3
	4-wise	0, 0	100, 0	63, 4	60, 9	34, 2
Branch coverage (%)	2-wise	50, 0	100, 0	85, 0	84, 1	14, 3
	3-wise	50, 0	100, 0	87, 5	86, 6	13, 0
	4-wise	50, 0	100, 0	90, 6	88, 3	13, 4
Cost (# test cases)	2-wise	6, 0	231, 0	8, 0	19, 6	41, 3
	3-wise	8, 0	732, 0	17, 0	50, 5	137, 3
	4-wise	16, 0	1462, 0	43, 5	105, 8	273, 0
	Manual	2, 0	62, 0	11, 0	17, 5	15, 3

We report the cost comparison between manual tests and t-wise ($t \leq 4$) generated tests. We report several statistics relevant to the obtained results: minimum, maximum, median, mean, and standard deviation (SD) values

test suites using combinatorial goals up to 4-wise achieve high branch coverage. Nevertheless, we confirm that there is a need to consider other test design aspects and higher t-wise strengths to achieve over 90% branch coverage. This underscores the need to study further how combinatorial testing can be improved in practice and what aspects can be taken into account to achieve better code coverage. The programs considered in this study are used in real-time systems to provide operational control in trains. The runtime behavior of such systems depends not only on the choice of parameters but also on providing the right choice of values at the right time points. By considering such information, combinatorial tests might be more effective at covering the code. This needs to be further studied by considering the extent to which t-wise can be used in combination with other types of information.

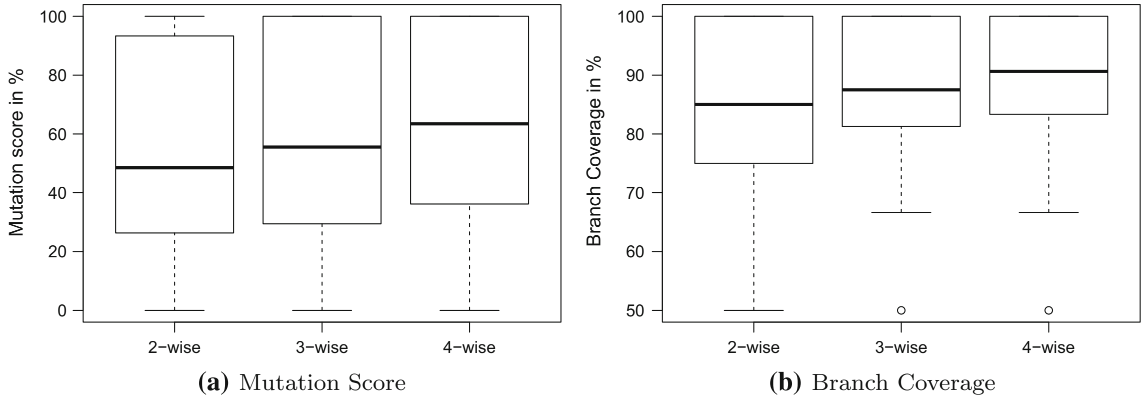
RQ4: How effective are tests generated using Q-EMCQ for 2-wise, 3-wise, and 4-wise at detecting injected faults?

To answer RQ4 regarding the effectiveness in terms of fault detection, we focused on analyzing the test suite quality of combinatorial testing. For all programs, as shown in Fig. 7a, the fault detection scores of pairwise generated test suites show an average mutation score of 52%, but they are not significantly worse than 3-wise (57% on average) and 4-wise (60%) test suites with no statistically significant differences (effect size of 0, 4 in Table 5). Hence, a test that is generated automatically using combinatorial techniques up to 4-wise is not a good indicator of test effectiveness in terms of mutation score. However, one hypothesis is emerging from this result: if 4-wise test suites are not achieving a high mutation score, there is a need to generate higher-strength test suites as well as find ways to improve the fault detection scores by using other test design techniques.

This is, to some extent, an entirely surprising result. Our expectation was that combinatorial testing of higher strength than 2-wise would yield high scores (over 90%) in terms of fault detection. Tests for 4-wise in testing FBD programs would intuitively be quite good test cases at detecting faults. However, the results of our study are not consistent with the results of other studies (Kuhn et al. 2010; Richard Kuhn et al. 2004; Kuhn and Reilly 2002) reporting the degree of interaction occurring in naturally occurring faults. Surprisingly, this expectation does not clearly hold for the results of this study. Our results indicate that combinatorial test cases with interactions up to 4-wise are not good indicators of test effectiveness in terms of fault detection. In addition, our results are not showing any statistically significant difference in mutation score between any t-wise strength considered in this study.

RQ5: How does Q-EMCQ for 2-wise, 3-wise, and 4-wise compare with manual testing in terms of cost?

As a baseline for comparing the cost of testing, we used test cases created by industrial engineers in Bombardier Transportation for all 37 programs included in this case study. These programs are part of a project delivered already to customers and thoroughly tested. Each test suite contains a set of test cases containing inputs, expected and actual outputs, and time information expressing timing constraints. As in this case study, we consider the number of test cases related to the cost of creating, executing, and checking the result of each test case; we use the number of test cases in a test suite manually created as a realistic measure of cost encountered in the industrial practice for the programs considered. We assume that the higher the number of test cases, the higher are the respective cost associated with each test



(a) Mutation Score **(b)** Branch Coverage

Fig. 7 Mutation score and achieved branch coverage comparison between 2-wise, 3-wise, and 4-wise generated test cases; boxes span from first to third quartile, black middle lines mark the median, and the whiskers extend up to 1.5x the interquartile range and the circle symbols represent outliers

Table 5 For mutation score and coverage, we calculated the effect size of 2-wise versus 3-wise and 4-wise

Measure	Method	Effect Size	*p* value
Mutation Score	2-wise 3-wise	0.444	0.411
	3-wise 4-wise	0.464	0.595
	2-wise 4-wise	0.412	0.193
Coverage	2-wise 3-wise	0.454	0.494
	3-wise 4-wise	0.447	0.432
	2-wise 4-wise	0.412	0.186
Cost (#test cases)	Manual 2-wise	0.536	< 0.565
	Manual 3-wise	0.376	< 0.054
	Manual 4-wise	**0.157**	< 0.001

In addition, for the cost measure, we calculated the effect size between manual testing and t-wise with $t \leq 4$. We also report the p values of a Wilcoxon–Mann–Whitney U-tests with significant effect sizes shown in bold

The bold number shows the best result obtained

suite. This section aims to answer RQ5 regarding the relative cost of performing testing concerning the number of test cases generated using Q-EMCQ in comparison with manually hand-crafted tests. As seen in Table 4, the number of test cases for 2-wise and 3-wise is consistently significantly lower than for 4-wise created tests. As seen in Table 5, the cost of performing testing using Q-EMCQ for 4-wise is consistently significantly higher (in terms of the number of test cases) than for manually created test suites; 3-wise and 4-wise generated test suites are longer (88 and 33 more test cases on average, respectively) over manual testing. There is enough evidence to claim that the results between 4-wise and manual test suites are statistically significant, with a p-value below the traditional statistical significance limit of 0, 05 and a standardized effect size of 0, 157. The effect is weaker for the result between $3 - wise$ and manual test suites with a p-value of 0,05 and an effect size of 0, 376.

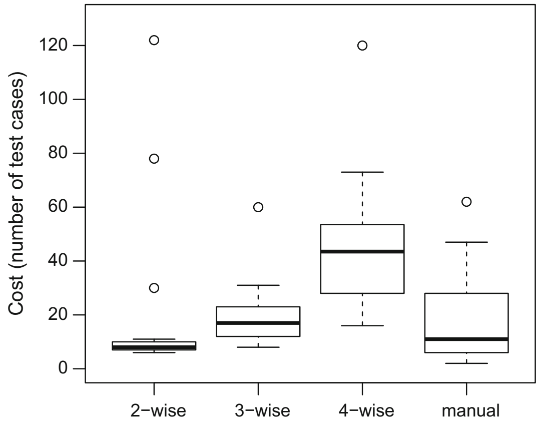


Fig. 8 Cost comparison in terms of number of test cases between 2-wise, 3-wise, 4-wise generated test cases and manual testing

As seen in Fig. 8, the use of 2-wise consistently results in shorter test suites for all programs than for 3-wise and 4-wise. It seems like 2-wise test suites are comparable with manual test suites in terms of the number of test cases. Examining Table 5, we see the same pattern in the statistical analysis: standardized effect sizes being higher than 0, 1, with p-value higher than the traditional statistical significance limit of 0, 05. The effect is the strongest for the 2-wise and 4-wise with a standardized effect size of 0, 08. It seems that 4-wise will create much more tests than 2-wise, which in practice can affect the cost of performing testing.

6.2 Answering RQ6

As highlighted earlier, the experiment for RQ6 investigates the performance of Q-EMCQ against some selected meta/hyper-heuristics.

RQ6: Apart from the minimization problem (i.e., *t*-wise test generation), is Q-EMCQ sufficiently general to solve (maximization) optimization problem (i.e., module clustering)?

As the general observation from the results in Table 6, we note that hyper-heuristics generally outperform meta-heuristics. This could be due to the fact hyper-heuristics can adaptively choose the right operator based on the need for the current search. However, in terms of execution times, general meta-heuristics appear to be slightly faster than their hyper-heuristic counterparts owing to the direct link from the problem domain to the actual search operators.

Going to specific comparison from hyper-heuristic group in Table 6 and Figs. 9b, 10b and 11b, Q-EMCQ and MCF outperform all other hyper-heuristics as far as the best MQ (with 2.226, 2.899, and 4.465) for Credit Card Payment System, Unified University Inventory, and Food Book, respectively. In terms of average, Q-EMCQ has a better performance than that of MCF. Putting Q-EMCQ and MCF aside, Tabu HHH outperforms EMCQ in both average and best MQ. On the positive note, EMCQ outperforms all other hyper-heuristics as far as execution times are concerned.

Considering the comparison with the meta-heuristics, Q-EMCQ still manages to outperform all algorithms. In the case of the Credit Card Payment System, TLBO manages to match the best of MQ for Q-EMCQ, although with poorer average MQ. This is expected as the Credit Card Payment System consists of only 14 classes as compared to 19 and 31 classes in the Unified Inventory System, and Food Book, respectively. In terms of execution time, SCA has the best time performance overall for Unified Inventory University (with 37.782 secs) and Food Book (with 56.798 secs), while TLBO gives the best performance for Credit Card Payment System (with 33.531 secs). Here, SOS gives the poorest execution time.

Table 6 Comparing Q-EMCQ with contemporary meta/hyper-heuristics, EMCQ[52, 9], modified choice function Pour Shahrzad et al. (2018), Tabu HHH Zamli et al. (2016), TLBO Rao et al. (2011), SCA Mirjalili (2016), SOS Cheng and Prayogo (2014)

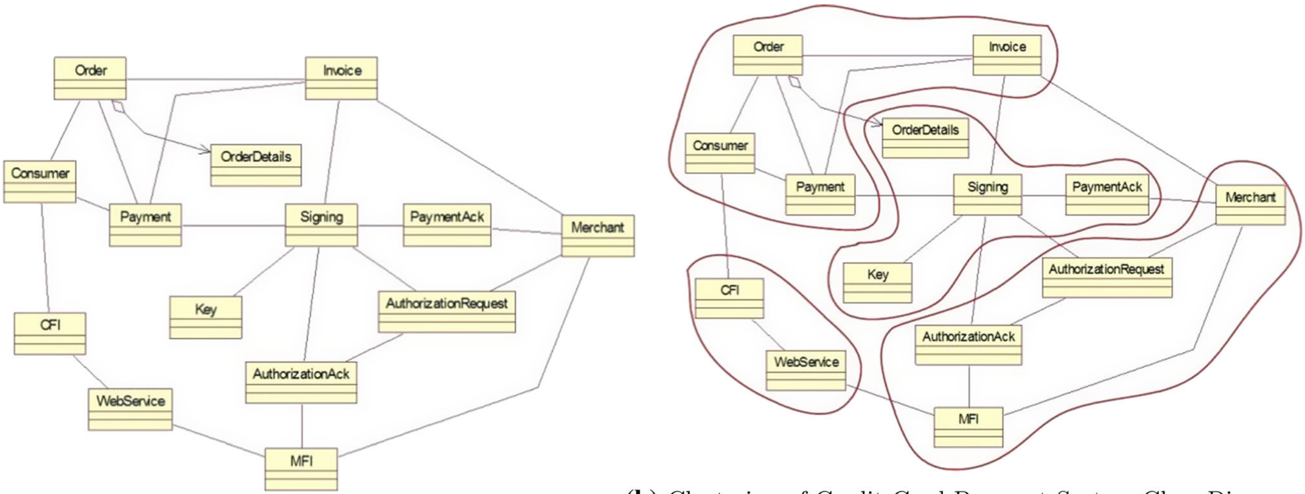
Case Studies	Hyper-heuristics								Meta-heuristics					
	Q-EMCQ		EMCQ		Mod. Choi. Func.		Tabu HHH		TLBO		SCA		SOS	
	Ave MQ/ Ave Time (sec)	Best MQ	Ave MQ/ Ave Time (sec)	Best MQ	Ave MQ/ Ave Time (sec)	Best MQ	Ave MQ/ Ave Time (sec)	Best MQ	Ave MQ/ Ave Time (sec)	Best MQ	Ave MQ/ Ave Time (sec)	Best MQ	Ave MQ/ Ave Time (sec)	Best MQ
Card Payment Sys.	**2.031/** 40.876	**2.226**	1.976/ 39.016	2.171	2.002/ 43.232	**2.226**	**2.110/** 54.876	**2.226**	1.999/ **33.531**	**2.226**	1.957/ 36.953	2.078	1.983/ 45.080	2.117
Unified Inventory University	2.315/ 41.673	**2.899**	2.288/ 39.312	2.721	2.543/ 47.673	**2.899**	**2.392/** 54.673	**2.899**	2.135/ 40.080	2.623	2.140/ **37.782**	2.588	2.118/ 47.282	2.581
Food Book	**3.43/** 63.5300	**4.465**	3.018/ 58.850	4.076	3.32/ 65.232	**4.465**	3.001/ 70.5300	4.377	3.012/ 68.130	3.741	2.991/ **56.798**	3.551	2.881/ 68.250	3.305

The bold numbers show the best results obtained

7 Discussion

Reflecting on the work undertaken, certain observations can be elaborated as lessons learned. In particular, we can group our observations into two parts: The first part relates to the design of Q-EMCQ and its operators, whereas the second part relates to its performance in the industrial case study.

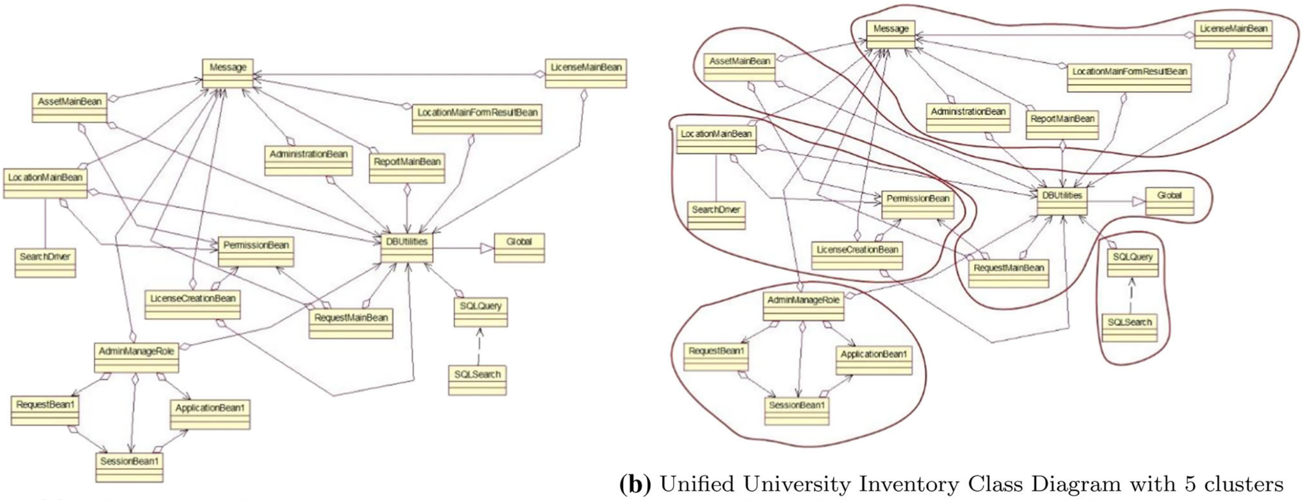
Concerning the first part, we foresee Q-EMCQ as a general hybrid meta-heuristic. Conventional hybrid meta-heuristics



(a) Credit Card Payment System Class Diagram

(b) Clustering of Credit Card Payment System Class Diagram with 4 clusters and Best MQ = 2.226 generated by Q-EMCQ

Fig. 9 Clustering of Case Study 1—Credit Card Payment System



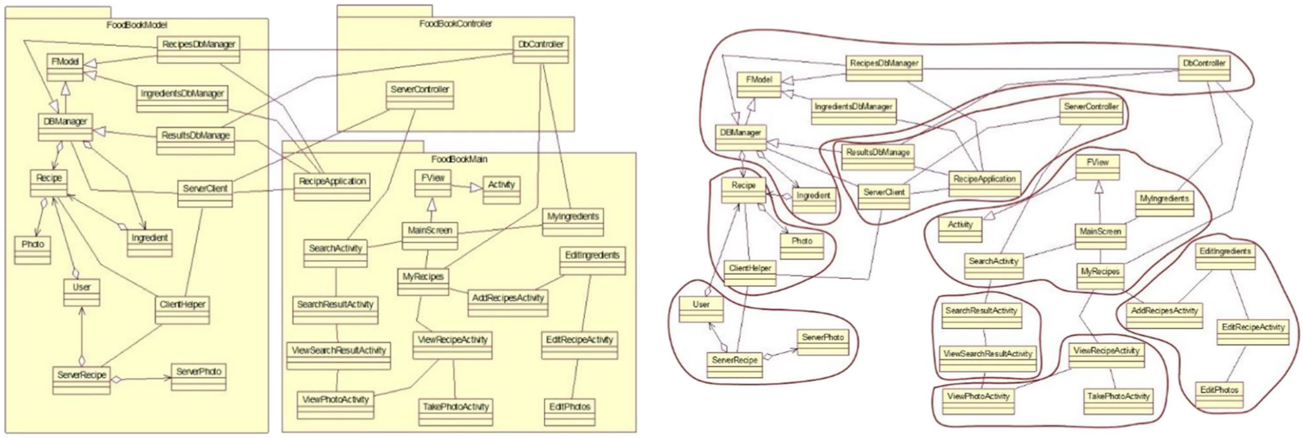
(a) Unified University Inventory Class Diagram

(b) Unified University Inventory Class Diagram with 5 clusters and Best MQ = 2.899 generated by Q-EMCQ

Fig. 10 Clustering of Case Study 2—Unified University Inventory

are often tightly coupled (whereby two or more operators are interleaved together) and too specific for a particular problem. In addition, the selection of a particular operator during the searching process does not consider the previous performances of that operator. Contrary to conventional hybrid meta-heuristic, apart from being adaptive, Q-EMCQ design is highly flexible. Two aspects of Q-EMCQ can be treated as "pluggable" components. First, the current Monte Carlo heuristic selection and acceptance mechanism can be replaced with other selection and acceptance mechanisms. Second, the individual search operators can also be replaced with other operators (taking into consideration whether it is for local or global search). For instance, the cuckoo's perturbation operator can easily be substituted by the simulated annealing's neighborhood search operator.

Unlike pure meta-heuristic approaches, Q-EMCQ also does not require any specific tuning apart from calibrating maximum iteration and population size. Notably, cuckoo as a standalone algorithm requires the calibration of three control parameters: maximum iteration, population size, and probability (p_a) for replacing poor eggs. Similarly, flower as a standalone algorithm requires the calibration of three control parameters: maximum iteration, population size, and probability (p) for local or global pollination. Unlike the cuckoo and flower algorithms, the Jaya algorithm does not require additional parameters (other than maximum iteration



(a) Food Book Class Diagram Programmer's Package with 3 clusters and MQ = 1.524

(b) Food Book Class Diagram with 8 clusters and Best MQ = 4.466 generated by Q-EMCQ

Fig. 11 Clustering of Case Study 3—Food Book Class Diagram

and population size). Adopted as individual search operators, the cuckoo's probability (p_a) and the flower's probability (p) are completely abandoned within the design of Q-EMCQ.

Similar to its predecessor EMCQ, the selection of the search operators at any instance of the searching process is adaptively performed based on the Monte Carlo heuristic selection and acceptance mechanism. However, unlike EMCQ, Q-EMCQ also keeps the memory of the best performing operators via the Q-learning table. The effect of maintaining the memory can be seen as far as average convergence is concerned. In the early iteration stage, Q-EMCQ behaves like EMCQ as far average convergence is concerned. However, toward the end of the iteration stage, while EMCQ relies solely on the random selection of operators, Q-EMCQ uses historical performance to perform the selection. For this reason, Q-EMCQ has better average convergence than EMCQ.

As far as comparative benchmark experiments with other strategies are concerned, we note that Q-EMCQ and DPSO give the best results overall (see 2 and 3). On the negative note, the approach taken by DPSO is rather problem-specific. On the contrary, our experiments with maximization problems (e.g., module clustering) indicate that the Q-EMCQ approach is sufficiently general (refer to Table 6) although with small-time penalty to maintain the Q-learning mechanism. Here, DPSO has introduced two new control parameters as probabilities (pro1 and pro2) in addition to the existing social parameters (c_1 and c_2) and inertia weight (w) to balance between exploration and exploitation in the context of its application for t-wise test generation. In this manner, adopting DPSO to other optimization problems can be difficult owing to the need to calibrate and tune all these control parameters accordingly.

On the other side of the spectrum, PICT and IPOG appear to perform the poorest (with no results matching any of the best sizes). A more subtle observation is the fact that meta-heuristic and hyper-heuristics-based strategies appear to outperform general computational-based strategies.

As part of our study, we used the number of test cases to estimate the cost in terms of creation, execution, and result checking. While the cost of creating and executing a test for creating combinatorial tests can be low compared to manual testing, the cost of evaluating the test result is usually human-intensive. Our study suggests that combinatorial test suites for 4-wise contain 100 created test steps (number of tests) on average. By considering generating optimized or shorter test suites, one could improve the cost of performing combinatorial testing. We note here that the cost of testing is heavily influenced by the human cost of checking the test result. In this paper, we do not take into account the time of checking the results per test case. In practice, this might not be the real situation. A test strategy, which requires every input parameter in the program to be used in a certain combination, could contain test cases that are not specified in requirements. This might increase the cost of checking the test case result. A more accurate cost model would be needed to obtain more confidence in the results.

The results of this paper show that 2- to 4-wise combinations of values are not able to detect more than 60% of injected faults (52% on average for 2-wise, 57% on average for 3-wise, and 60% on average for 4-wise) and are not able to cover more than 88% of the code (84% on average for 2-wise, 86% on average for 3 – 2-wise, and 88% on average for 4-wise). Surprisingly, these results are not consistent with the results of other studies (Kuhn et al. 2010; Richard Kuhn et al. 2004; Kuhn and Reilly 2002) reporting the degree of interaction occurring in real faults occurring in industrial systems.

While not conclusive, the results of this study are interesting because they suggest that the degree of interaction involved in faults might not be as low as previously thought. As a direct result, testing all 4-wise combinations might not provide reasonable assurance in terms of fault detection. There is a need to consider ways of studying the use of higher-strength algorithms and tailoring these to the programs considered in this study, which are used in real-time software systems to provide control capabilities in trains. The behavior of such a program depends not only on the choice of parameters but also on providing the right choice of continuous values. By considering the state of the system of the timing information, combinatorial tests might be more effective at detecting faults. Bergström and Enoiu (2017) indicated that the use of timing information in combinatorial testing for base-choice criterion results in higher code coverage and fault detection. This needs to be further studied by considering the extent to which t-wise can be used in combination with the real-time behavior of the input parameters.

8 Limitations

Our results regarding effectiveness are not based on naturally occurring faults. In our study, we automatically seeded mutants to measure the fault detection capability of the written tests. While it is possible that faults are naturally happening in the industry would yield different results, there are some evidence (Just et al. 2014) to support the use of injected faults as substitutes for real faults. Another possible risk of evaluating test suites based on mutation analysis is the *equivalent mutant* problem in which these faults cannot show any externally visible deviation. The mutation score in this study was calculated based on the ratio of killed mutants to mutants in total (including equivalent mutants, as we do not know which mutants are equivalent). Unfortunately, this fact introduces a threat to the validity of this measurement. In addition, the results are based on a case study in one company using 37 PLC programs. Even if this number can be considered quite small, we argue that having access to real industrial programs created by engineers working in the safety-critical domain can be representative. More studies are needed to generalize these results to other systems and domains.

Finally, our general clustering problem has also dealt with small-scale problems (the largest class diagram is only 31 classes). As the classes get larger, enumeration of the possible solution grows in a factorial manner. With such growth, there could be a potential clustering mismatch. In this case, maximizing MQ can be seen as two conflicting sides of the same coin. On one side of the coin, there is a need to get the largest MQ for better modularization. On the other side of the coin, automatically maximizing MQ for a large set of classes may be counterproductive (in terms of disrupting the overall architectural package structure of the classes). In fact, some individual clusters may not be intuitive to programmers at all. For these reasons, there is a need to balance between getting the good enough MQ (i.e., which may not be the best one) and simultaneously obtaining a meaningful set of clusters.

9 Conclusions

We present Q-EMCQ, a Q-learning-based hyper-heuristic exponential Monte Carlo with a counter strategy for combinatorial interaction test generation, and show the evaluation results obtained from a case study performed at Bombardier Transportation, a large-scale company focusing on developing industrial control software. The 37 programs considered in this study have been in development and are used in different train products all over the world. The evaluation shows that the Q-EMCQ test generation method is efficient in terms of generation time and test suite size. Our results suggest that combinatorial interaction test generation can achieve high branch coverage. However, these generated test suites do not show high levels of fault detection in terms of mutation score and are more costly (i.e., in terms of the number of created test cases) than manual test suites created by experienced industrial engineers. The obtained results are useful for both practitioners, tool developers, and researchers. Finally, to complement our current work, we have also demonstrated the generality of Q-EMCQ via addressing the maximization problem (i.e., involving the clustering of class diagrams). For future work, we can focus on exploring the adoption of Q-EMCQ for large embedded software both for t-wise test generation as well as its modularization.

Acknowledgements Open access funding provided by Karlstad University. The work reported in this paper is funded by Fundamental Research Grant from the Ministry of Higher Education Malaysia titled: An Artificial Neural Network-Sine Cosine Algorithm-based Hybrid Prediction Model for the production of Cellulose Nanocrystals from Oil Palm Empty Fruit Bunch (RDU1918014). Wasif Afzal is supported by The Knowledge Foundation through 20160139 (TestMine) & 20130085 (TOCSYC) and the European Union's Horizon 2020 research and innovation programme, grant agreement No 871319. Eduard Enoiu is funded from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No. 737494 and the Swedish Innovation Agency, Vinnova (MegaM@Rt2).

Compliance with ethical standards

Conflict of interest All authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

Ahmed BS, Zamli KZ, Lim CP (2012) Application of particle swarm optimization to uniform and variable strength covering array construction. Appl Soft Comput 12(4):1330–1347

Ahmed BS, Abdulsamad TS, Potrus MY (2015) Achievement of minimized combinatorial test suite for configuration-aware software functional testing using the cuckoo search algorithm. Inf Softw Technol 66(C):13–29

Ahmed BS, Zamli KZ, Afzal W, Bures M (2017) Constrained interaction testing: a systematic literature study. IEEE Access 5:25706–25730

Ammann P, Offutt J (2008) Introduction to software testing. Cambridge University Press, Cambridge

Andrews JH, Briand LC, Labiche Y (2005) Is mutation an appropriate tool for testing experiments? In: Proceedings of the 27th international conference on Software engineering, ACM, pp 402–411

Ayob M, Kendall G (2003) A Monte Carlo hyper-heuristic to optimise component placement sequencing for multi head placement machine. In: Placement machine, INTECH'03, Thailand, pp 132–141

Bell KZ, Vouk MA (2005) On effectiveness of pairwise methodology for testing network-centric software. In: Enabling technologies for the new knowledge society: ITI 3rd international conference on information and communications technology, IEEE, pp 221–235

Bergström H, Enoiu EP (2017) Using timed base-choice coverage criterion for testing industrial control software. In: International conference on software testing, verification and validation workshops (ICSTW), pp 216–219

Burke E, Kendall G, Newall J, Hart E, Ross P, Schulenburg S (2003) Hyper-heuristics: an emerging direction in modern search technology. Springer, Boston, pp 457–474

Burke EK, Hyde M, Kendall G, Ochoa G, Özcan E, Woodward JR (2010) A classification of hyper-heuristic approaches. Springer, Boston, pp 449–468

Burke EK, Gendreau M, Hyde M, Kendall G, Ochoa G, Özcan E, Rong Q (2013) Hyper-heuristics: a survey of the state of the art. J Op Res Soc 64(12):1695–1724

Calvagna A, Gargantini A (2009) Ipo-s: Incremental generation of combinatorial interaction test data based on symmetries of covering arrays. In: 2009 International conference on software testing, verification, and validation workshops, pp 10–18

Charbachi P, Eklund L, Enoiu E (2017) Can pairwise testing perform comparably to manually handcrafted testing carried out by industrial engineers? In: International conference on software quality, reliability and security companion (QRS-C), pp 92–99

Cheng M-Y, Prayogo D (2014) Symbiotic organisms search: a new metaheuristic optimization algorithm. Comput Struct 139:98–112

Chen X, Gu Q, Li A, Chen D (2009) Variable strength interaction testing with an ant colony system approach. In: Proceedings of the 2009 16th Asia-pacific software engineering conference. APSEC '09, IEEE computer society, Washington, pp 160–167

Cheong CP, Fong S, Lei P, Chatwin C, Young R (2012) Designing an efficient and secure credit card-based payment system with web services based on ansi x9.59–2006. J Inf Process Syst 8(3):495–520

Christopher JCH (1992) Watkins and Peter Dayan technical note: Q-learning. Mach Learn 8(3):279–292

Cohen DM, Dalal SR, Kajla A, Patton GC (1994) The automatic efficient test generator (AETG) system. In: International symposium on software reliability engineering, IEEE, pp 303–309

Cohen MB (2004) Designing test suites for software interaction testing. Technical report, The University of Auckland, Ph.D. Thesis

Cohen MB, Dwyer MB, Shi J (2007) 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, ACM, New York, pp 129–139

Cohen DM, Dalal SR, Parelius J, Patton GC (1996) The combinatorial design approach to automatic test generation. IEEE Softw 13(5):83

Cohen DM, Dalal SR, Fredman ML, Patton GC (1997) The AETG system: an approach to testing based on combinatorial design. IEEE Trans Softw Eng 23(7):437–444

Colbourn CJ, Martirosyan SS, Mullen GL, Shasha D, Sherwood GB, Yucas JL (2006) Products of mixed covering arrays of strength two. J Comb Des 14(2):124–138

Dalal SR, Jain A, Karunanithi N, Leaton JM, Lott CM (1998) Model-based testing of a highly programmable system. In: International symposium on software reliability engineering, IEEE, pp 174–179

DeMillo RA, Lipton RJ, Sayward FG (1978) Hints on test data selection: help for the practicing programmer. In: Computer, vol 11, IEEE

Dhiman G, Kaur A (2019) Stoa: a bio-inspired based optimization algorithm for industrial engineering problems. Eng Appl Artif Intell 82:148–174

Enoiu E, Sundmark D, Čaušević A, Pettersson P (2017) A comparative study of manual and automated testing for industrial control software. In: International conference osoftware testing, verification and validation (ICST), IEEE, pp 412–417

Forbes M, Lawrence J, Lei Y, Kacker RN, Kuhn DR (2008) Refining the in-parameter-order strategy for constructing covering arrays. J Res Natl Inst Stand Technol 113(5):287–297

Ghandehari LS, Czerwonka J, Lei Y, Shafiee S, Kacker R, Kuhn R (2014) An empirical comparison of combinatorial and random testing. In: International conference on software testing, verification and validation workshops (ICSTW), IEEE, pp 68–77

Grindal M, Lindström B, Offutt J, Andler SF (2006) An evaluation of combination strategies for test case selection. Empir Softw Eng 11(4):583–611

Howell D (2012) Statistical methods for psychology. Cengage Learning, Boston

Jain M, Maurya S, Rani A, Singh V (2018) Owl search algorithm: a novel nature-inspired heuristic paradigm for global optimization. J Intell Fuzzy Syst 34:1573–1582

Jia Y, Cohen MB, Harman M, Petke J (2015) Learning combinatorial interaction test generation strategies using hyperheuristic search. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, vol 1, pp 540–550

John KH, Tiegelkamp M (2010) IEC 61131–3: programming industrial automation systems: concepts and programming languages, requirements for programming systems, decision-making aids. Springer, Berlin

Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G (2014) Are mutants a valid substitute for real faults in software testing? In: International symposium on foundations of software engineering, ACM

Kashan AH, Tavakkoli-Moghaddam R, Gen M (2019) Find-fix-finish-exploit-analyze (f3ea) meta-heuristic algorithm: an effective algorithm with new evolutionary operators for global optimization. Comput Ind Eng 128:192–218

Kendall G, Sabar NR, Ayob M (2014) An exponential Monte Carlo local search algorithm for the berth allocation problem. In: 10th International conference of the practice and theory of automated timetabling, pp 544–548

Kennedy J, Eberhart R (1995) Particle swarm optimization. In: Proceedings of the IEEE international conference on neural networks, vol 4, pp 1942–1948

Kirkpatrick S, Gelatt CD, Vecchi MP (1983) Optimization by simulated annealing. Science 220(4598):671–680

Kuhn DR, Kacker RN, Lei Y (2010) Sp 800-142. Practical combinatorial testing

Kuhn DR, Okum V (2006) Pseudo-exhaustive testing for software. In: 30th Annual IEEE/NASA software engineering workshop, SEW'06, IEEE, pp 153–158

Kuhn DR, Reilly MJ (2002) An investigation of the applicability of design of experiments to software testing. In: Proceedings of the 27th annual NASA Goddard/IEEE on software engineering workshop, IEEE, pp 91–95

Lei Y, Raghu Kacker D, Kuhn R, Okun V, Lawrence J (2008) Ipog–ipog-d: efficient test generation for multi-way combinatorial testing. Softw Test Verif Reliab 18(3):125–148

Lei Y, Kacker R , Kuhn DR, Okun V, Lawrence J (2007) Ipog: a general strategy for t-way software testing. In: Proceedings of the 14th annual IEEE international conference and workshops on the engineering of computer-based systems. ECBS '07, IEEE computer society, Washington, pp 549–556

Lei Y, Tai KC (1998) In-parameter-order: a test generation strategy for pairwise testing. In: Proceedings of third IEEE international high-assurance systems engineering symposium (Cat. No. 98EX231), pp 254–261

Leung HKN, White L (1991) A cost model to compare regression test strategies. In: Proceedings of the conference on Software maintenance, IEEE, pp 201–208

Mahmoud T, Ahmed BS (2015) An efficient strategy for covering array construction with fuzzy logic-based adaptive swarm optimization for software testing use. Expert Syst Appl 42(22):8753–8765

Mandl R (1985) Orthogonal latin squares: an application of experiment design to compiler testing. Commun ACM 28(10):1054–1058

Mirjalili S (2016) Sca: a sine cosine algorithm for solving optimization problems. Knowl Based Syst 96:120–133

Mousavirad SJ, Ebrahimpour-Komleh H (2017) Human mental search: a new population-based metaheuristic optimization algorithm. Appl Intell 47(3):850–887

Nie C, Leung H (2011) A survey of combinatorial testing. ACM Comput Surv 43(2):11:1–11:29

Pour Shahrzad M, Drake John H, Burke Edmund K (2018) A choice function hyper-heuristic framework for the allocation of maintenance tasks in danish railways. Comput Oper Res 93:15–26

Praditwong K, Harman M, Yao X (2011) Software module clustering as a multi-objective search problem. IEEE Trans Softw Eng 37(2):264–282

Press WH, Teukolsky SA, Vetterling WT, Flannery BP (1992) Numerical recipes in C: the art of scientific computing, 2nd edn. Cambridge University Press, New York

Rao R (2016) Jaya: a simple and new optimization algorithm for solving constrained and unconstrained optimization problems. Int J Ind Eng Comput 7(1):19–34

Rao RV, Savsani VJ, Vakharia DP (2011) Teaching-learning-based optimization: a novel method for constrained mechanical design optimization problems. Comput Aided Des 43(3):303–315

Richard Kuhn D, Wallace DR, Gallo AM (2004) Software fault interactions and implications for software testing. IEEE Trans Softw Eng 30(6):418–421

R-Project (2005) R: a language and environment for statistical computing. The R foundation for statistical computing, http://www.R-project.org

Sabar NR, Kendall G (2015) Population based monte carlo tree search hyper-heuristic for combinatorial optimization problems. Inf Sci 314(Supplement C):225–239

Samma H, Lim CP, Saleh JM (2016) A new reinforcement learning-based memetic particle swarm optimizer. Appl Softw Comput 43(C):276–297

Sampath S, Bryce RC (2012) Improving the effectiveness of test suite reduction for user-session-based testing of web applications. Inf Softw Technol 54(7):724–738

Schroeder PJ, Bolaki P, Gopu V (2004) Comparing the fault detection effectiveness of n-way and random test suites. In: International symposium on empirical software engineering, IEEE, pp 49–59

Shayanfar H, Gharehchopogh FS (2018) Farmland fertility: a new metaheuristic algorithm for solving continuous optimization problems. Appl Soft Comput 71:728–746

Shiba T, Tsuchiya T, Kikuno T (2004) Using artificial life techniques to generate test cases for combinatorial testing. In: Proceedings of the 28th annual international computer software and applications conference, COMPSAC '04, IEEE computer society, Washington, vol 01, pp 72–77

Shin D, Jee E, Bae DH (2012) Empirical evaluation on FBD model-based test coverage criteria using mutation analysis. In: Model driven engineering languages and systems. Springer

Sobh K, Oliveira D, Liu B, Mayantz M, Zhang YM, Alhazmi A, de Bled R, Al-Sharawi A (2010) Software design document, testing, deployment and configuration management, and user manual of the UUIS—a team 4 COMP5541-W10 project approach. CoRR, abs/1005.0169

Tsai CW, Huang WC, Chiang MH, Chiang MC, Yang CS (2014) A hyper-heuristic scheduling algorithm for cloud. IEEE Trans Cloud Comput 2(2):236–250

Vargha A, Delaney HD (2000) A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J Educ Behav Stat 25(2):101–132

Wallace DR, Richard Kuhn D (2001) Failure modes in medical device software: an analysis of 15 years of recall data. Int J Reliab Qual Saf Eng 8(04):351–371

Williams AW, Probert RL (1996) A practical strategy for testing pairwise coverage of network interfaces. In: Proceedings of seventh international symposium on software reliability engineering, pp 246–254

Wu H, Nie C, Kuo FC, Leung H, Colbourn CJ (2015) A discrete particle swarm optimization for covering array generation. IEEE Trans Evolut Comput 19(4):575–591

Yang XS, Deb S (2009) Cuckoo search via levy flights. In: 2009 World congress on nature biologically inspired computing (NaBIC), pp 210–214

Yang X-S (2008) Nature-inspired metaheuristic algorithms. Luniver Press, Frome

Yang X-S (2012) Flower pollination algorithm for global optimization. Springer, Berlin, pp 240–249

Zamli KZ, Alkazemi BY, Kendall G (2016) A tabu search hyper-heuristic strategy for *t*-way test suite generation. Appl Soft Comput 44(C):57–74

Zamli KZ, Din F, Kendall G, Ahmed BS (2017) An experimental study of hyper-heuristic selection and acceptance mechanism for combinatorial *t*-way test suite generation. Inf Sci 399(C):121–153