Constructing multiple unique input/output sequences using metaheuristic optimisation techniques

Q. Guo, R.M. Hierons, M. Harman and K. Derderian

Abstract: Multiple unique input/output sequences (UIOs) are often used to generate robust and compact test sequences in finite state machine (FSM) based testing. However, computing UIOs is NP-hard. Metaheuristic optimisation techniques (MOTs) such as genetic algorithms (GAs) and simulated annealing (SA) are effective in providing good solutions for some NP-hard problems. In the paper, the authors investigate the construction of UIOs by using MOTs. They define a fitness function to guide the search for potential UIOs and use sharing techniques to encourage MOTs to locate UIOs that are calculated as local optima in a search domain. They also compare the performance of GA and SA for UIO construction. Experimental results suggest that, after using a sharing technique, both GA and SA can find a majority of UIOs from the models under test.

1 Introduction

Finite state machines (FSMs) have been used for modelling systems in various areas such as sequential circuits, software and communication protocols [1–9]. In FSM-based testing, a standard test strategy consists of two parts, namely, transition test and tail state verification. The former part aims to determine whether a transition of an implementation under test (IUT) produces the expected output while the latter checks that the IUT arrives at the specified state when a transition test is finished. Three techniques are proposed for state verification: unique input/output sequence (UIO), distinguishing sequence (DS) and characterising set (CS). Test sequence generation methods using the above are called the U-, D- and W-methods, respectively. In terms of fault coverage, the U-, D- and W-Methods exhibit no significant difference [6]. The use of UIOs has several advantages: (i) not all FSMs have a distinguishing sequence (DS), but nearly all FSMs have UIOs for each state [1]; (ii) the length of a UIO is no longer than that of a DS; (iii) while UIOs may be longer than a characterising set, in practice UIOs often lead to shorter test sequences. Aho *et al.* [1] showed how an efficient test sequence may be produced using UIOs for state verification. Shen *et al.* [7] extended the method by using *multiple UIOs* for each state and showed that this leads to a shorter test sequence. This paper considers the problem of finding multiple UIOs for a given FSM. Yang and Ural [8], Miller [9] and Hierons [10, 11] showed that overlap can be used in conjunction with (multiple) UIOs to further reduce the test sequence length.

Unfortunately, computing UIOs is NP-hard [4]. Lee and Yannakakis [4] note that adaptive distinguishing sequences and UIOs may be produced by constructing a state splitting tree. However, no rule is explicitly defined to guide the construction of an input sequence. Naik [12] proposes an approach to construct UIOs by introducing a set of inference rules. Some minimal length UIOs are found. These are used to deduce some other states' UIOs. A state's UIO is produced by concatenating a sequence to another state, whose UIO has been found, with this state's UIO sequence. Although it may reduce the time taken to find some UIOs, the inference rule inevitably increases a UIO's length, which consequently leads to longer test sequences.

Metaheuristic optimisation techniques (MOTs) such as genetic algorithms (GAs) [13] and simulated annealing (SA) [14, 15] have proven efficient in search and optimisation and have shown their effectiveness in providing good solutions to some NP-hard problems such as the Travelling Salesman Problem. When searching for optimal solutions in multi-modal functions, the use of *sharing* techniques is likely to lead to a population that contains several sub-populations that cover local optima [16]. This result is useful since in some search problems we wish to locate not only global optima, but also local optima.

In software engineering, MOTs have been introduced for the generation of test data. Applications can be found in structural coverage testing (branch coverage testing) [17, 18], worst case and best case execution time estimating [19], and exception detecting [20].

Previous work has shown that a GA may be used to find UIOs for an FSM [21]. The search used a fitness function based on the state splitting tree [Note 1]. In experiments the GA outperformed random search, especially on finding longer UIOs. However, a significant drawback was also noted. Some UIOs are missed with high probability. Solutions of all UIOs form multi-modals (local optima) in

IEE Proceedings online no. 20045001

doi: 10.1049/ip-sen:20045001

Paper first received 20th April 2004 and in revised form 1st February 2005

Q. Guo, R.M. Hierons and K. Derderian are with the Department of Information System and Computing, Brunel University, Uxbridge UB8 3PH, UK

M. Harman is with the Department of Computer Science, King's College London, Strand, London WC2R 2LS, UK

E-mail: Qiang.Guo@brunel.ac.uk

Note 1: The state splitting tree will be defined in Section 2.

the search space - a search might find only a few of these local optima and thus miss some UIOs. To find more UIOs, it is necessary to use some techniques to effectively detect local optima. This paper investigates the construction of UIOs using MOTs combined with sharing techniques. A rule is defined to calculate the similarity degree (SD) among candidates. The value of SD is used as a guide to degrade the fitness values of candidates that are highly similar to others. Degraded individuals are less likely to be selected for the reproduction, which helps to maintain the diversity in a genetic pool. The proposed approach of using a GA or a SA, with sharing, is evaluated using two FSMs. The results of this evaluation are also used to compare the effectiveness of GA and SA for the problem.

2 FSMs based testing

2.1 Finite state machines

A deterministic FSM M is defined as a quintuple (I,O,S,δ,λ), where I,O and S are finite and nonempty sets of input symbols, output symbols, and states, respectively; $\delta : S \times I \rightarrow S$ is the state transition function; and $\lambda : S \times I \rightarrow O$ is the output function. If the machine receives an input $a \in I$ when in state $s \in S$, then it moves to the state $\delta(s,a)$ and produces output $\lambda(s,a)$. Functions δ and λ can be extended to take input sequences in the usual way [22].

An FSM M can be viewed as a directed graph $G = (V,E)$, where the set of vertices V represents the state set S of M and the set of edges E represents the transitions. An edge has label a/o, where $a \in I$ and $o \in O$ are the corresponding transition's input and output. Figure 1 illustrates an FSM represented by its corresponding directed graph.

Two states s_i and s_j are said to be *equivalent* if and only if for every input sequence $\alpha \in I$ the machine produces the same output sequence, $\lambda(s_i,\alpha) = \lambda(s_j,\alpha)$. Machines M_1 and M_2 are *equivalent* if and only if for every state in M_1 there is an equivalent state in M_2, and vice versa. A machine is *minimal* (*reduced*) if and only if no two states are equivalent. It is assumed that any FSM being considered is minimal since any (deterministic) FSM can be converted into an equivalent (deterministic) minimal FSM [22]. An FSM is *completely specified* if and only if for each state s_i and input a, there is a specified next state $s_{i+1} = \delta(s_i,a)$ and a specified output $o_i = \lambda(s_i,a)$; otherwise, the machine is *partially specified*. A partially specified FSM can be converted to a completely specified one in two ways [22].

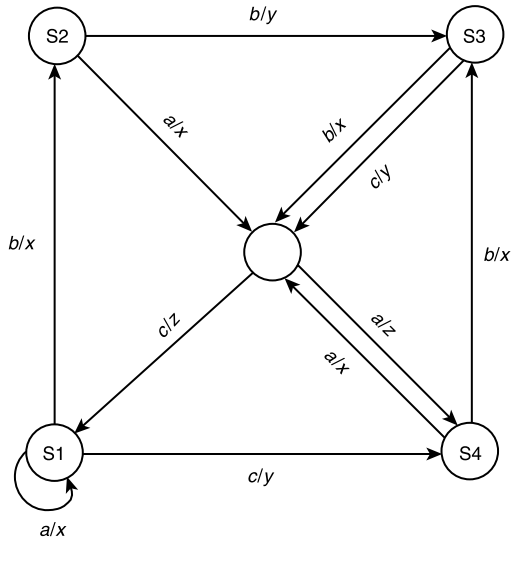


Fig. 1 *Finite state machine*

One way is to define an error state. When a machine is in state s and receives an input a such that there is no transition from s with input a, it moves to the error state with a given (error) output. The other way is to add a loop transition. When receiving an undefined input, the state of a machine remains unchanged. At the same time, the machine produces no output. An FSM is *strongly connected* if, given any ordered pair of states (s_i,s_j), there is a sequence of transitions that moves the FSM from s_i to s_j.

It is assumed throughout this paper that an FSM is deterministic, minimal, completely specified and strongly connected. A partially specified machine is converted to a completely specified one by adding an error state.

2.2 Conformance testing

Given a specification FSM M, for which we have its complete transition diagram, and an implementation M', for which we can only observe its I/O behaviour ('black box'), we want to test to determine whether the I/O behaviour of M' conforms to that of M. This is called *conformance testing* [23]. A test sequence that solves this problem is called a *checking sequence*. An I/O difference between the specification and implementation can be caused by either an incorrect output (an output fault) or an earlier incorrect state transfer (a state transfer fault). The latter can be detected by adding a final state check after a transition. A standard test strategy is:

1. *Homing*: Move M' to an initial state s.
2. *Output check*: Apply an input sequence α and compare the output sequences generated by M and M' separately.
3. *Tail state verification*: Using state verification techniques to check the final state.

The first step is known as homing a machine to a desired initial state. The second step checks whether M' produces the desired output. The last step checks whether M' is in the expected state $s' = \delta(s,\alpha)$ after the transition [22]. There are three main techniques used for state verification:

- distinguishing sequence (DS)
- unique input/output (UIO)
- characterising set (CS).

A distinguishing sequence is an input sequence that produces a different output for each state. Not every FSM has a DS.

A UIO sequence of state s_i is an input/output sequence x/y, that may be observed from s_i, such that the output sequence produced by the machine in response to x from any other state is different from y, i.e.$\lambda(s_i,x) = y$ and $\lambda(s_i,x) \neq \lambda(s_j,x)$ for any $i \neq j$. A DS defines a UIO. While not every FSM has a UIO for each state, some FSMs without a DS have a UIO for each state.

A characterising set W is a set of input sequences with the property that, for every pair of states (s_i, s_j), $j \neq i$, there is some $w \in W$ such that $\lambda(s_i,w) \neq \lambda(s_j,w)$. Thus, the output sequences produced by executing each $w \in W$ from s_j verifies s_j.

This paper focuses on the problem of generating multiple UIOs.

2.3 State splitting tree

A state splitting tree (SST) [4] is a rooted tree T that is used to construct adaptive distinguishing sequences or UIOs from an FSM. Each node in the tree has a predecessor (parent) and successors (children). A tree starts from a root node and terminates at discrete partitions: sets that contain one state only. The predecessor of the root node, which contains the set

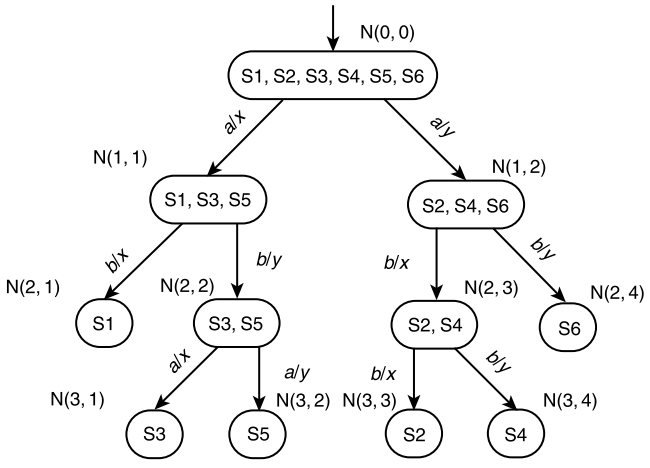


Fig. 2 *State splitting tree from an FSM*

of all states, is null. The nodes corresponding to a single state have an empty successor. These nodes are also known as terminals. A child node is connected to its parent node through an edge labelled with characters. The edge implies that the set of states in the child node is partitioned from that in the parent node upon receiving the labelled characters. The splitting tree is complete if the partition is a discrete partition.

An example is illustrated in Fig. 2, where an FSM (different from the one shown in Fig. 1) has six states, namely, $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$. The input set is $I = \{a, b\}$, while the output set is $O = \{x, y\}$. The root node is indicated by N(0,0) [Note 2], containing the set of all states. Suppose states $\{s_1, s_3, s_5\}$ produce x when responding to a, while $\{s_2, s_4, s_6\}$ produce y. Then $\{s_1, s_3, s_5\}$ and $\{s_2, s_4, s_6\}$ are distinguished by a. Two new nodes rooted from N(0,0) are then generated, indicated by N(1,1) and N(1,2). If we then apply b, the state reached from $\{s_1\}$ by a produces x, while the states reached from $\{s_3, s_5\}$ by a produce y. Thus ab distinguish $\{s_1\}$ from $\{s_3, s_5\}$. Two new nodes rooted from N(1,1) are generated, denoted by N(2,1) and N(2,2). The same operation can be applied to $\{s_2, s_4, s_6\}$. Repeating this process, we can get all discrete partitions as shown in Fig. 2. Note that for some FSMs this process might terminate without producing a complete set of discrete partitions since there need not exist such a tree [22]. A path from a discrete partition node to the root node forms a UIO for the state related to this node. When the splitting tree is complete, we can construct UIOs for each state.

Unfortunately, the problem of finding data to build up the state splitting tree is NP-hard. This provides the motivation for investigating the use of MOTs. The problem is discussed in the following Sections.

3 Metaheuristic optimisation techniques (MOTs)

3.1 Genetic algorithms

Genetic algorithms (GAs) [13, 16] are heuristic optimisation techniques that simulate natural processes, utilising selection, crossover and mutation. Since Holland's seminal work (1975) [24], they have been applied to a variety of learning and optimisation problems.

3.1.1 Simple GA: A simple GA starts with a randomly generated population, each element (chromosome) being a sequence of variables/parameters for the optimisation problem. The set of chromosomes represents the search space: the set of potential solutions. The representation format of variable values is determined by the system under evaluation. It can be represented in binary form, by real numbers, by characters, etc. The search proceeds through a number of iterations. Each iteration is treated as a generation. At each iteration, the current set of candidates (the population) is used to produce a new population. The quality of each chromosome is determined by a fitness function that depends on the problem considered. Those of high fitness have a greater probability of contributing to the new population.

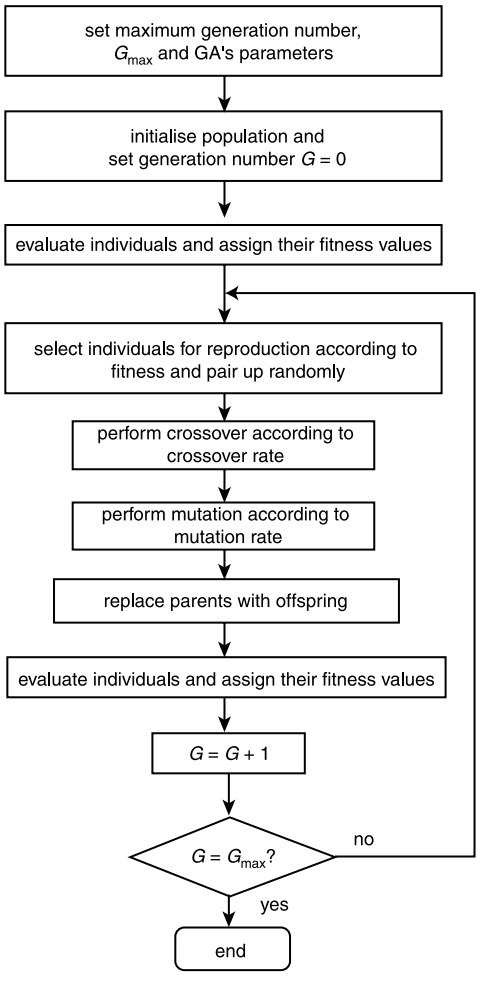


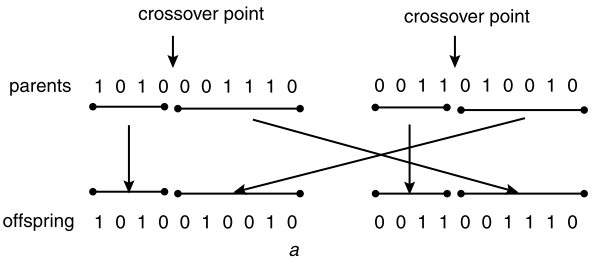
Fig. 3 *Flow chart for a simple GA*

Selection is applied to choose chromosomes from the current population and pairs them up as parents. *Crossover* and *mutation* are applied to produce new chromosomes. A new population is formed from new chromosomes produced on the basis of *crossover* and *mutation* and may also contain chromosomes from the previous population.

Figure 3 shows a flow chart for a simple GA. The following sub-Sections give a detailed explanation on *Selection*, *Crossover* and *Mutation*. All experiments in this work used roulette wheel selection and uniform crossover.

3.1.2 Encoding: A potential solution to a problem may be represented as a set of parameters. These parameters are joined together to form a string of values (often referred to as a *chromosome*). Parameter values can be represented in various forms such as binary form, real numbers, characters, etc. The representation format should make the computation effective and convenient.

Note 2: N(i, j): i indicates that the node is in the ith layer from the tree. j refers to the jth node in the ith layer.



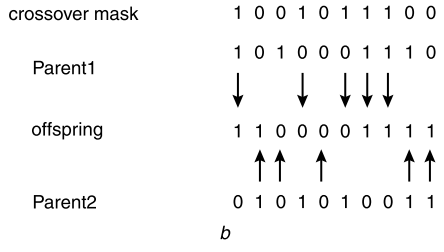


Fig. 4 *Crossover operation in a simple GA*

a Single-point crossover
b Uniform crossover

Fig. 5 *Mutation operation in a simple GA*

3.1.3 Reproduction: During the reproductive phase of a GA, individuals are selected from the population and recombined, producing children. Parents are selected randomly from the population using a scheme which favours the more fit individuals. Roulette wheel selection (RWS) and tournament selection (TS) are the two most popular selection regimes that are used for reproduction. RWS involves selecting individuals randomly but weighted as if they were chosen using a roulette wheel, where the amount of space allocated on the wheel to each individual is proportional to its fitness, while TS selects the fittest individual from a randomly chosen group of individuals.

Having selected two parents, their chromosomes are *recombined*, typically using the mechanisms of *crossover* and *mutation*. *Crossover* exchanges information between

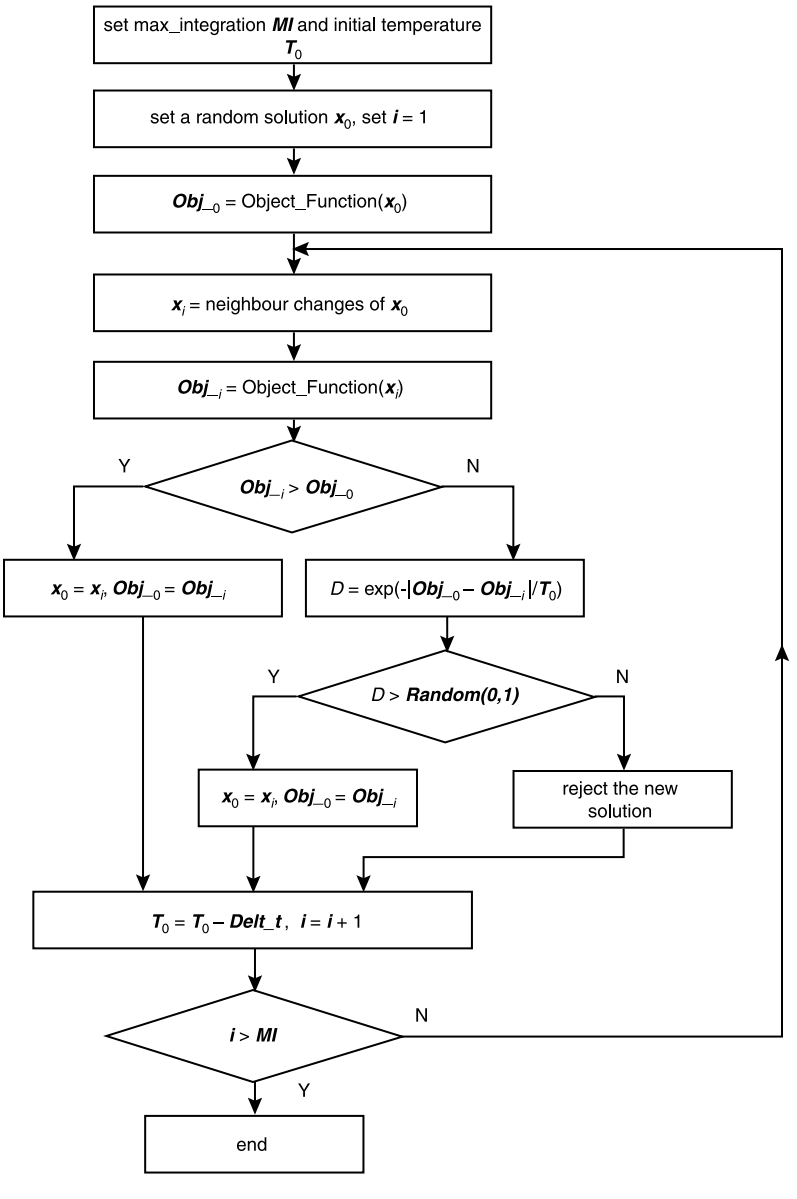


Fig. 6 *Simulated annealing algorithm*

parent chromosomes by exchanging parameter values to form children. It takes two individuals, and cuts their chromosome strings at some randomly chosen position, to produce two 'head' segments, and two 'tail' segments. The tail segments are then swapped over to produce two new full length chromosomes (see Fig. 4*a*). Two offspring inherit some genes from each parent. This is known as *single point crossover*. In *uniform crossover*, each gene in the offspring is created by copying the corresponding gene from one or other parent, chosen according to a randomly generated *crossover mask*. Where there is a 1 in the crossover mask, the gene is copied from the first parent, and where there is a 0 in the mask, the gene is copied from the second parent (see Fig. 4*b*). The process is repeated with the parents exchanged to produce the second offspring.

Crossover is not usually applied to all pairs of individuals selected for mating. A random choice is made, where the likelihood of crossover being applied is typically between 0.6 and 1.0 [13]. If crossover is not applied, offspring are produced simply by duplicating the parents. This gives each individual a chance of appearing in the next generation.

Mutation is applied to each child individually after crossover, randomly altering each gene with a small probability. Figure 5 shows the fourth gene of the chromosome being mutated. Mutation prevents the genetic pool from premature convergence, namely, getting stuck in local maxima/minima. However, too high a mutation rate prevents the genetic pool from convergence. A probability value between 0.01 and 0.1 for mutation is suggested [13].

3.1.4 Sharing scheme:

A simple GA is likely to converge to a single peak, even in domains characterised by multiple peaks of equivalent fitness. Moreover, in dealing with multimodal functions with peaks of unequal value, GA is likely to converge to the best peak. To identify multiple optima in the domain, some mechanisms should be used to force a GA to maintain a diverse population of members throughout its search. Sharing is such a mechanism that is proposed to overcome the above limitations. Sharing, proposed by Holland [24] and expanded by Goldberg and Richardson [16], aims to reduce the fitness of individuals that have highly similar members within the population. This rewards individuals who uniquely exploit areas of the domain while discouraging redundant (highly similar) individuals in a domain. This causes population diversity pressure, which helps maintain population members at local optima.

The shared fitness of an individual i is given by $f_{(sh,i)} = f_{(i)}/m_{(i)}$, where $f_{(i)}$ is the raw fitness of the individual and $m_{(i)}$ is the peak count. The peak count is calculated by summing a sharing function over all members of the population $m_{(i)} = \sum_{j=i}^{N} sh(d_{(i,j)})$. The distance $d_{(i,j)}$ represents the distance between individual i and individual j in the population, determined by a similarity measurement. If the sharing function determines that the distance is within a fixed radius σ_{sh}, it returns a value determined by $sh(d_{(i,j)}) = 1 - (d_{(i,j)}/\sigma_{sh})^{\alpha_{sh}}$; otherwise it returns 0. α_{sh} is a constant that regulates the shape of the sharing function.

3.2 Simulated annealing

Simulated annealing (SA) was first proposed by Kirkpatrick, *et al.* [14]. The basic principle is to iteratively improve a given solution by performing local changes. Usually, changes that improve the solution are accepted, whereas those changes that make the solution worse are accepted with a probability that depends on the temperature.

Traditionally, SA works on minimising the cost or energy of solutions to find the global minimal solution. In this paper, in order to make a reasonable comparison with GA, SA is slightly modified where the maximising heuristic is adopted.

In order to enable local search to escape from local optima through downhill moves, Metropolis *et al.* [15] proposed an algorithm parametrised by a temperature t. A move that produces a reduction of δ in the fitness is accepted with probability $\min(1, e^{-\delta/t})$. Figure 6 illustrates the general SA scheme.

4 Apply MOTs to FSMs

4.1 Solution representation

When applying MOTs to an FSM, the first question that has to be considered is what representation is suitable. In this work, the potential solutions in a genetic pool are defined as strings of characters from the input set I. A Do not care character '♯' is also used to further maintain diversity [21]. When receiving this input, the state of an FSM remains unchanged and no output is produced. When a solution is about to be perturbed to generate a new one in its neighbourhood, some of the characters in this solution are replaced with characters randomly selected from the rest of the input set, including '♯'.

4.2 Fitness definition

A key issue is to define a fitness function to (efficiently) evaluate the quality of solutions. This function should embody two aspects: (i) solutions should create as many discrete units as possible; (ii) the solution should be as short as possible. The function needs to make a tradeoff between these two points. This work uses a function that rewards the early occurrence of discrete partitions and punishes the chromosome's length. An alternative would be to model the number of state partitions and the length of a solution as two objectives and then treat them as multi-object optimisation problems (for more information on multi-object optimisation problems with GA see, for example, [13]).

A fitness function is defined to evaluate the quality of an input sequence. While applying an input sequence to an FSM, at each stage of a single input, the state splitting tree constructed is evaluated by (1):

$$f_{(i)} = \frac{x_i e^{(\delta x_i)}}{l_i^{\gamma}} + \alpha \frac{(y_i + \delta y_i)}{l_i} \tag{1}$$

where i refers to the ith input character. x_i denotes the number of existing discrete partitions, while δx_i is the number of new discrete partitions caused by the ith input. y_i is the number of existing separated groups, while δy_i is the number of new groups. l_i is the length of the input sequence up to the ith element (*Do Not Care* characters are excluded). α and γ are constants. It can be noted that a partition that finds a new discrete unit creates new separated groups as well.

Equation (1) consists of two parts: exponential part, $f_{e(i)} = x_i e^{(\delta x_i)}/l_i^{\gamma}$, and linear part, $f_{l(i)} = \alpha(y_i + \delta y_i)/l_i$. It can be seen that the occurrence of discrete partitions makes x_i and δx_i increase. Consequently, $x_i e^{(\delta x_i)}$ is increased exponentially. Meanwhile, with the input sequence's length l_i increasing, l_i^{γ} is reduced exponentially (γ should be greater than 1). Suppose x_i and l_i change approximately at the same rate, that is $\delta x_i \approx \delta l_i$; as long as $e^{(\delta x_i)}$ has faster dynamics than l_i^{γ}, $f_{e(i)}$ increases exponentially, causing f_i to be increased exponentially. However, if, with the length of the input sequence increasing, no discrete partition is found,

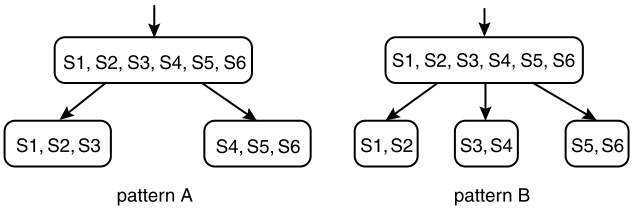


Fig. 7 *Two patterns of partitions*

$f_{e(i)}$ decreases exponentially, causing f_i to be decreased exponentially. $f_{e(i)}$ thus performs two actions: encouraging the early occurrence of discrete partitions and punishing the increment of an input sequence's length.

$f_{l(i)}$ also affects $f_{(i)}$ in a linear way. Compared to $f_{e(i)}$, it plays a less important role. This term rewards partitioning even when discrete classes have not been produced. Figure 7 shows two patterns with no discrete partition. We believe pattern B is better than A since B might find more discrete units in the forthcoming partitions.

Individuals that find discrete partitions at the first several inputs but fail to find more in the following steps may obtain higher fitness values than others. They are likely to dominate the population and cause the genetic pool to converge prematurely. To balance the evaluation, after all input characters have been examined, the final fitness value for an input candidate is defined as the average of (1):

$$F = \frac{1}{N}\sum_{i=1}^{N} f_{(i)} \tag{2}$$

where N is the sequence's length.

4.3 Sharing application

4.3.1 Similarity measurement:

Before reducing a candidate's fitness, a mechanism should be used to evaluate the similarities between two solutions. There are two standard techniques that are proposed to measure the distance between two individuals, namely Euclidian distance and Hamming distance. However, both methods are not suitable in this work since inputs for an FSM are ordered sequences. The order of characters plays a very important role in evaluating the similarity. This work defines a similarity degree (SD) to guide the degrade of a candidate's fitness value.

Definition 1: A valid partition (VP) is defined as a partition that gives rise to at least one new separated group when responding to an input character.

Figure 8 illustrates two patterns of partition. In the Figure, A is valid since the parent group is split into two new groups, while B is invalid since the current group is identical to its parent group and no new group is created. A UIO can be formed by a mixture of valid and invalid partitions.

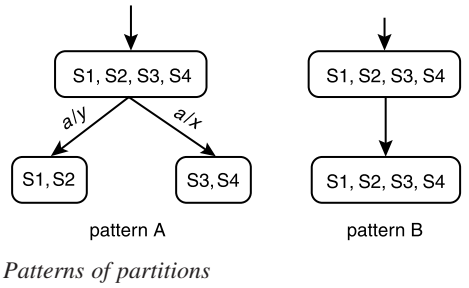


Fig. 8 *Patterns of partitions*

Definition 2: The max length of valid partition (MLVP) is the length up to an input that gives rise to the occurrence of the last valid partition.

Definition 3: The max discrete length (MDL) is the length up to an input character that gives rise to the occurrence of the last discrete partition.

Since a discrete partition defines a valid partition, MDL can never be greater than MLVP in a state splitting tree.

Definition 4: Similarity degree (SD) between two ordered sequences is defined as the length of a maximum length prefix sequence of these two sequences.

If elements in two ordered sequences are the same before the Nth character and different at the Nth, the SD is $N-1$ (♯ is excluded from the calculation).

4.3.2 Fitness degrade:

In order to prevent the population from converging at one or several peaks in the search space, at each iteration of computation, some candidates (that are not marked as degraded) that have high SD value should have the fitness value reduced by the mechanism as follows: (i) if a candidate's SD is greater than or equal to its MDL, its fitness value should be degraded to a very small value; else (ii) if $SD/MLVP$ passes a threshold value Θ, its fitness value is reduced to $(1 - SD/MLVP) \times V_{Org}$, where V_{Org} is its original value.

If a candidate's SD is greater than or equal to its MDL, it implies that, in terms of finding discrete partitions, this solution has been significantly represented by others and becomes redundant. Generally, the fitness value of a redundant candidate needs to be zero to keep it from reproduction. However, in the experiments, we set the value to 1% of its original value, allowing it to be selected with a low probability. If not, $(1 - SD/MLVP)$ controls the degree of decrement. The more information in a candidate that is represented in others, the more it is reduced. After a candidate's fitness value is reduced, it is marked as 'Degraded'.

Since a discrete partition defines a valid partition, MDL can never be greater than MLVP $(MDL \leq MLVP)$. $(1 - SD/MLVP)$ is applied only when $SD < MDL$. Since $SD < MDL \leq MLVP$, $(1 - SD/MLVP)$ is positive. When SD is greater than or equal to MDL, a fitness is reduced to a small value but still positive. So, the fitness value of an individual is always positive.

Threshold value Θ might be varied from different systems. Since it is enabled only when SD is less than MDL, a value between 0 and 1 can be applied. In the first model under test, we used $2/3$ while in the second model we used $1/2$.

4.4 Extending simple SA

Simple SA works on a single solution. In order to find all possible UIOs, multi-run based SA needs to be applied. Several papers involve the studies on multi-run based SA [25, 26]. In this paper, population based simulated annealing (PBSA) is used. Each individual in the genetic pool refers to a solution and is perturbed according to the simple SA scheme. All individuals in a population follow the same temperature drop control. During the computation, individuals in the genetic pool are compared with others. Those individuals that have been significantly represented by others have the fitness value reduced according to the sharing scheme.

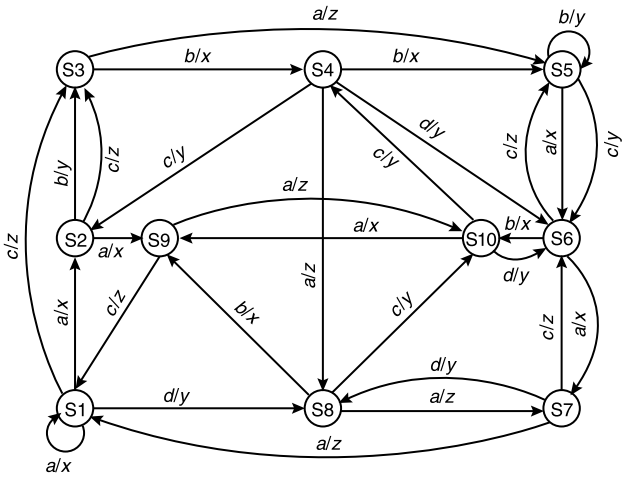


Fig. 9 *First FSM under test*

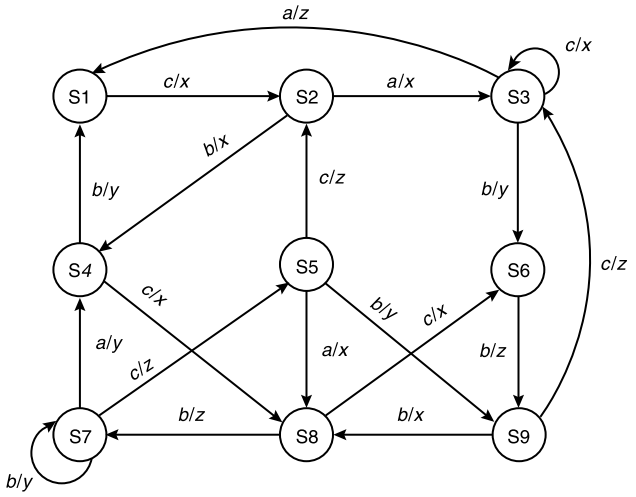


Fig. 10 *Second FSM under test*

Table 1: UIO list of Model 1

SQ	NS	SQ	NS	SQ	NS	SQ	NS	SQ	NS	SQ	NS	SQ	NS	SQ	NS
aaaa	8	aaa	5	bcca	5	accd	4	caab	4	bcc	3	aadc	2	ca	1
aaab	7	aabc	5	bccb	5	bab	4	cbaa	4	bcd	3	abd	2	aad	1
cca	7	aacc	5	cbcd	5	baca	4	cbb	4	caac	3	acbd	2	acb	1
ccb	7	abca	5	Ccc	5	bacb	4	cbcb	4	cabc	3	ba	2	ada	1
aaad	6	acaa	5	aabb	4	bbc	4	aab	3	cacc	3	caa	2	adba	1
aaca	6	acad	5	aabd	4	bca	4	aba	3	cbab	3	cabb	2	adbc	1
acca	6	baa	5	abcc	4	bcba	4	abb	3	cbd	3	cabd	2	adbd	1
accb	6	bba	5	abcd	4	bcbc	4	abc	3	aacb	2	caca	2	adc	1
accc	6	bbb	5	Aca	4	bcbd	4	bacc	3	ccb	2	cacb	2	cd	1
cbca	6	bcad	5	acba	4	bccc	4	bad	3	aacd	2	cad	2		
cbcc	6	bcbb	5	acbb	4	caaa	4	bb	3	aada	2	cb	2		

Table 2: UIO list of Model 2

SQ	NS	SQ	NS	SQ	NS	SQ	NS	SQ	NS	SQ	NS	SQ	NS	SQ	NS
cbb	7	cbca	6	bca	5	cacc	4	ab	4	ca	3	aa	3	a	2
bcb	7	bcc	6	ccb	4	caca	4	ccc	3	bcbc	3	cc	2	bc	1
bb	7	cacb	5	cbc	4	caa	4	cca	3	acc	3	bcba	2		
cbcc	6	vab	5	cba	4	acb	4	cb	3	aca	3	ba	2		

5 Experiments and discussions

Reference [21] has already investigated the construction of UIOs using a simple GA, finding that it outperformed random generation. This paper focuses on the studies of UIO distribution. In this Section we report the results of experiments that investigate the impact of sharing techniques when constructing multiple UIOs using GA and SA. Two models are used for experiments shown in Figs. 9 and 10, respectively. The first model has 10 states while the second has 9 states. Both FSMs use the same input and output sets. They are: $I = \{a, b, c, d\}$ and $O = \{x, y, z\}$. In order to compare the set of UIOs produced with a known complete set, the search was restricted to UIOs of length 4 or less. With such a restriction, $4^4 = 256$ input sequences can be constructed. There are 86 UIOs for Model 1 listed in Table 1 and 30 UIOs for Model 2 listed in Table 2. In the Tables, SQ stands for the input from a UIO sequence and NS refers to the number of states this sequence can identify. In all

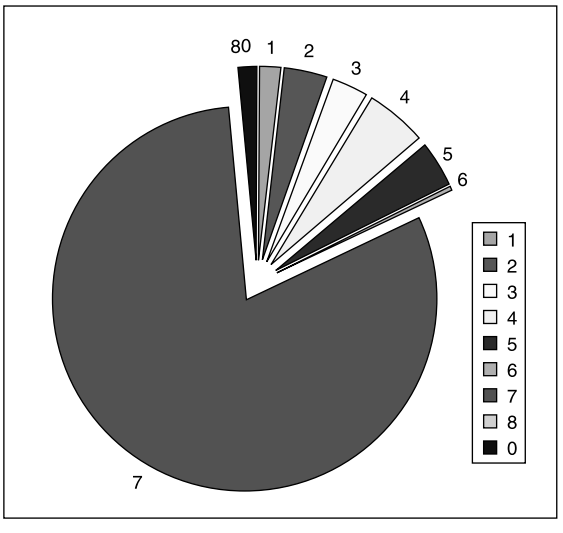


Fig. 11 *UIO distribution for GA without sharing in Model 1*
Legends indicate number of states that input sequences identify

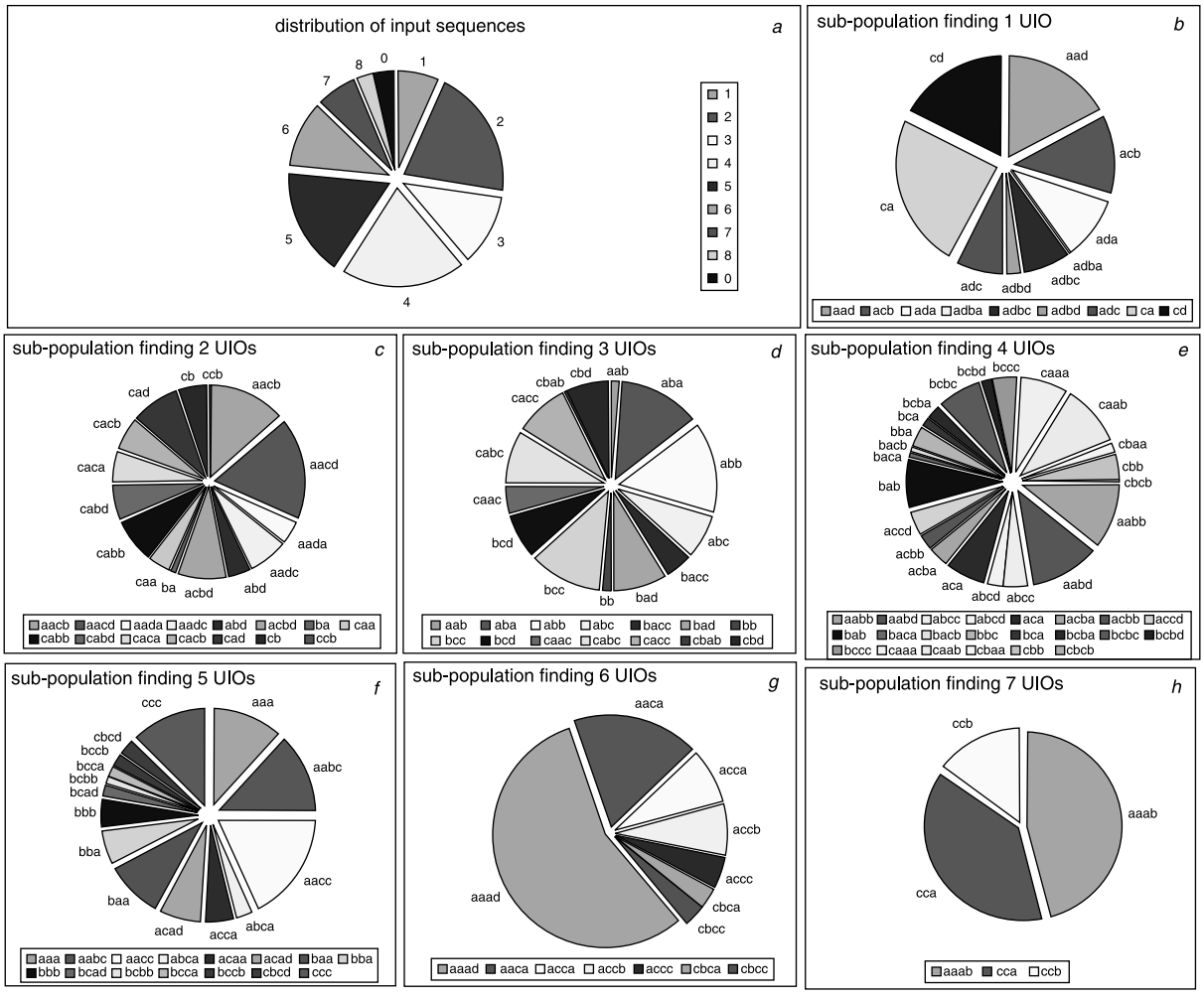


Fig. 12 *UIO distribution for GA with sharing in Model 1*

experiments, α and γ are set to 20 and 1.2, respectively. Section 5.4 explains the reason for choosing such values. The population size is set to 600 in all experiments.

5.1 Experiments on using GA

Experiments in this Section investigated the performance on the construction of UIOs using a simple GA. In [13], effects on choosing crossover and mutation rates have been studied. In this work, we simply follow the suggestions. The parameter settings are [Note 3]: $XRate = 0.75$; $MRate = 0.05$; $MGen = 300$. These settings remain unchanged throughout all experiments. The experiments used Model 1 first, and then Model 2. Threshold value θ is set to $2/3$ for Model 1 and $1/2$ for Model 2.

The first experiment studied the UIO distribution when using GA without sharing. The experiment was repeated 10 times. Figure 11 shows the UIO distribution of the best result. It can be seen that a majority of individuals move to a subpopulation that can identify seven states. The rest scatter among some other subpopulations that can identify 8, 6, 5, 4, 3, 2, 1 states. Owing to such an uneven distribution, some sequences that define UIOs of 6, 5, 4, 3, 2, 1 states are likely to be missed. Only 59 are found and so 27 were missed.

An experiment was then designed to investigate the use of the sharing technique. This experiment was repeated 10 times. The best result is shown in Fig. 12 (in B-H, sequences of legends indicate input sequences that define UIOs). It can be seen that, after applying sharing technique, the population is generally spread out, forming nine subpopulations. Each subpopulation contains UIO sequences that identify 0, 1, 2, 3, 4, 5, 6, 7, 8 states correspondingly. Only 4 UIOs were missed - the performance of the search had improved dramatically. However, the distributions in subpopulations do not form a good shape. Each subpopulation is dominated by one or several UIOs.

The impact of sharing techniques was further investigated by using Model 2. Figure 13 shows the best result from the five experiments. It can be seen that the distribution of input sequences is similar to that of GA with sharing in Model 1. Generally, the population is spread out, forming several subpopulations. However, each subpopulation is dominated by several individuals. We found that two UIOs were missed.

Experimental results above suggest that, when constructing UIOs using GA, without sharing technique, the population is likely to converge at several individuals that have high fitness values. The distribution of such a population causes some UIOs to be missed with high probability. This is consistent with the results of [21]. After applying the sharing technique, the population is encouraged to spread out and forms several subpopulations. These subpopulations are intended to cover all optima in the search space. The search quality significantly improved and more UIOs were found.

Note 3: XRate:Cross Rate; MRate:Mutation Rate; MGen:Max Generation

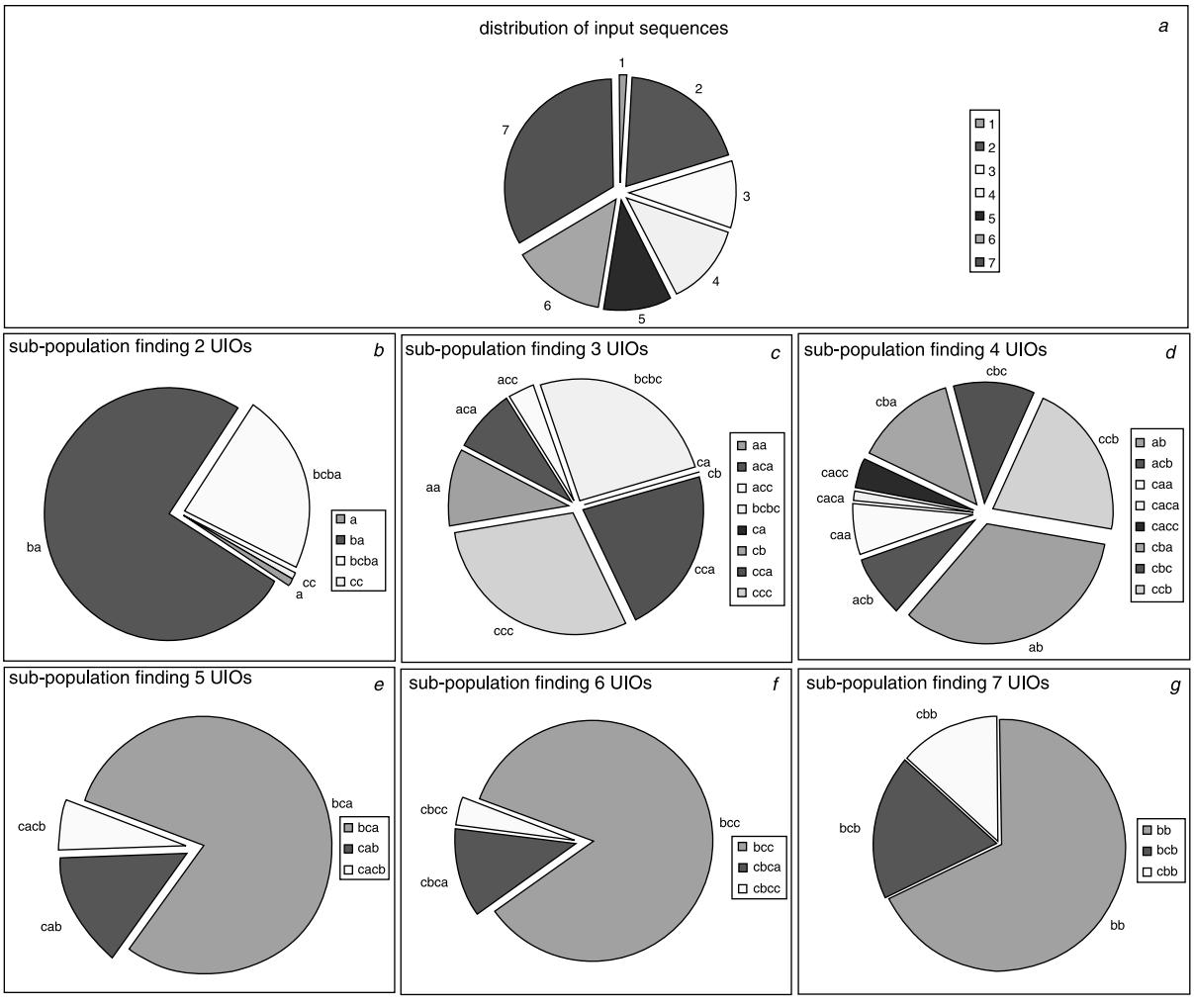


Fig. 13 *UIOs distribution for GA with sharing Model 2*

Convergence rates have also been studied when constructing UIOs for both models. Figures 14 and 15 show the average fitness values when constructing UIOs for Models 1 and 2, respectively. From the Figures it can be seen that, in Model 1, the genetic pool begins to converge after 200 generations, while in Model 2 genetic pool converges after 60 generations.

5.2 Experiments on using SA

Experiments in this Section aimed to study the performance of SA. As described above, a population based SA (PBSA) was used. Each individual in the genetic pool referred to a solution and was updated according to a simple SA's scheme. Individuals in the genetic pool were compared with others according to the sharing scheme. All individuals in a population followed the same temperature drop control. We also made a further restriction on the creation of a new solution. When an individual was required to generate a new solution, it was continuously perturbed in its neighbourhood until the new solution found at least one discrete partition. In order to make a comparison with GA, max generation is set to 300.

Two temperature drop control schema were considered. In the first experiment, the temperature was reduced by a normal exponential function $nT(i+1) = 0.99\,nT(i)$ (Fig. 16*a*), and a sharing technique was applied. The experiment was repeated 10 times and the best result is shown in Fig. 17.

From the Figure it can be seen that the general distribution and subpopulation distributions are quite

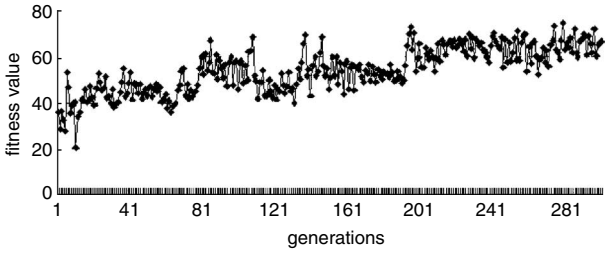


Fig. 14 *Average fitness values when constructing UIOs using GA: Model 1*

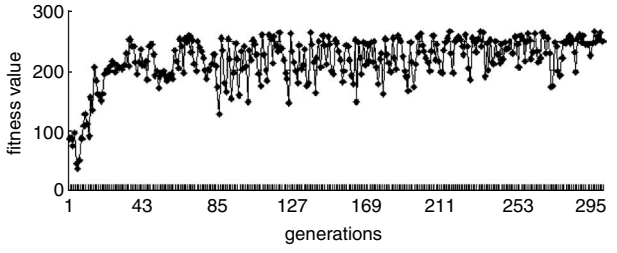


Fig. 15 *Average fitness values when constructing UIOs using GA: Model 2*

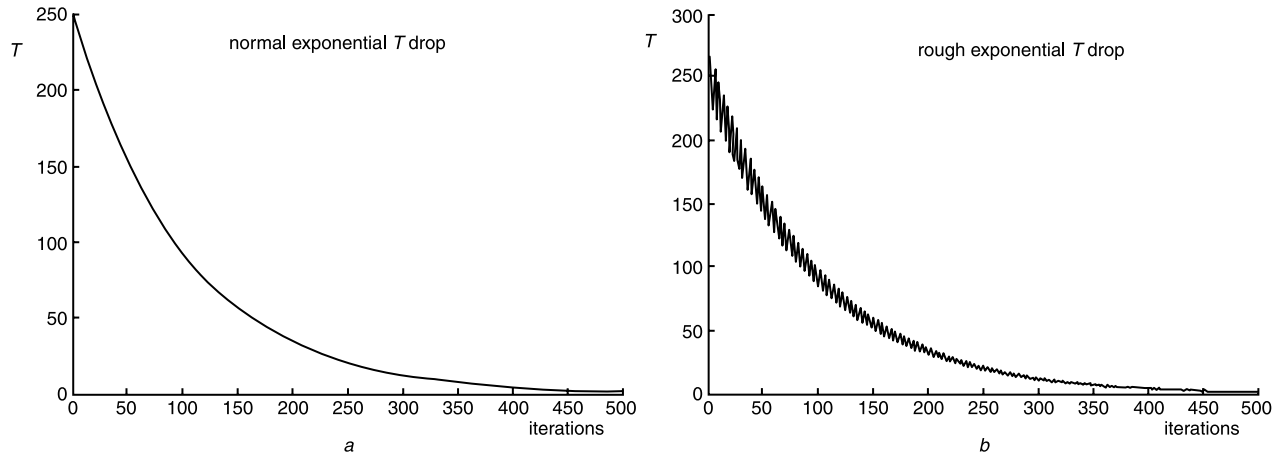


Fig. 16 *SA temperature drop schema*

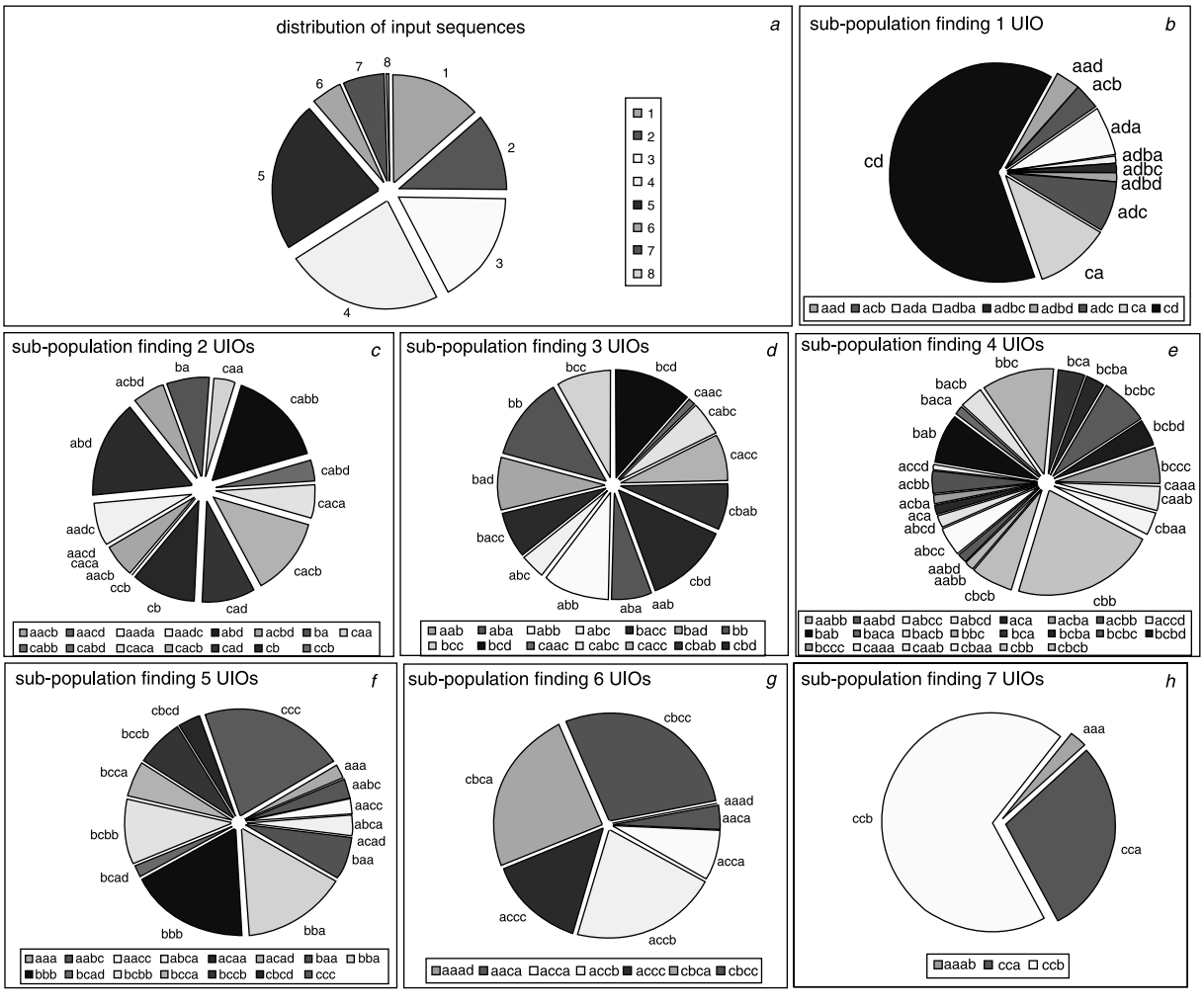


Fig. 17 *UIOs distribution for SA with normal exponential temperature drop in Model 1*

similar to that of GA with sharing. The population was formed with several subpopulations. Each subpopulation was dominated by several individuals. A total of eight UIOs were missed. Compared to the experiments studied in Section 5.1, this figure is quite high. To improve the search quality, the temperature drop scheme was changed to $nT(i+1) = 0.99\,nT(i) + nS(i+1)\,\sin(10\,\pi\,i)$, where $nS(i+1) = 0.95\,nS(i)$. The curve of the function is shown in Fig. 16*b*. Generally, the tendency of temperature control is still exponentially decreasing, but local bumps occur. The best result from 10 experiments is shown in Fig. 18. We find that the distribution of population and subpopulation have no significant changes. However, only two UIOs were missed. The performance is much better than the previous one.

Two SA methods were further studied by using Model 2. Figure 19 shows the best result on the normal temperature drop control, while Fig. 20 shows the best result for the

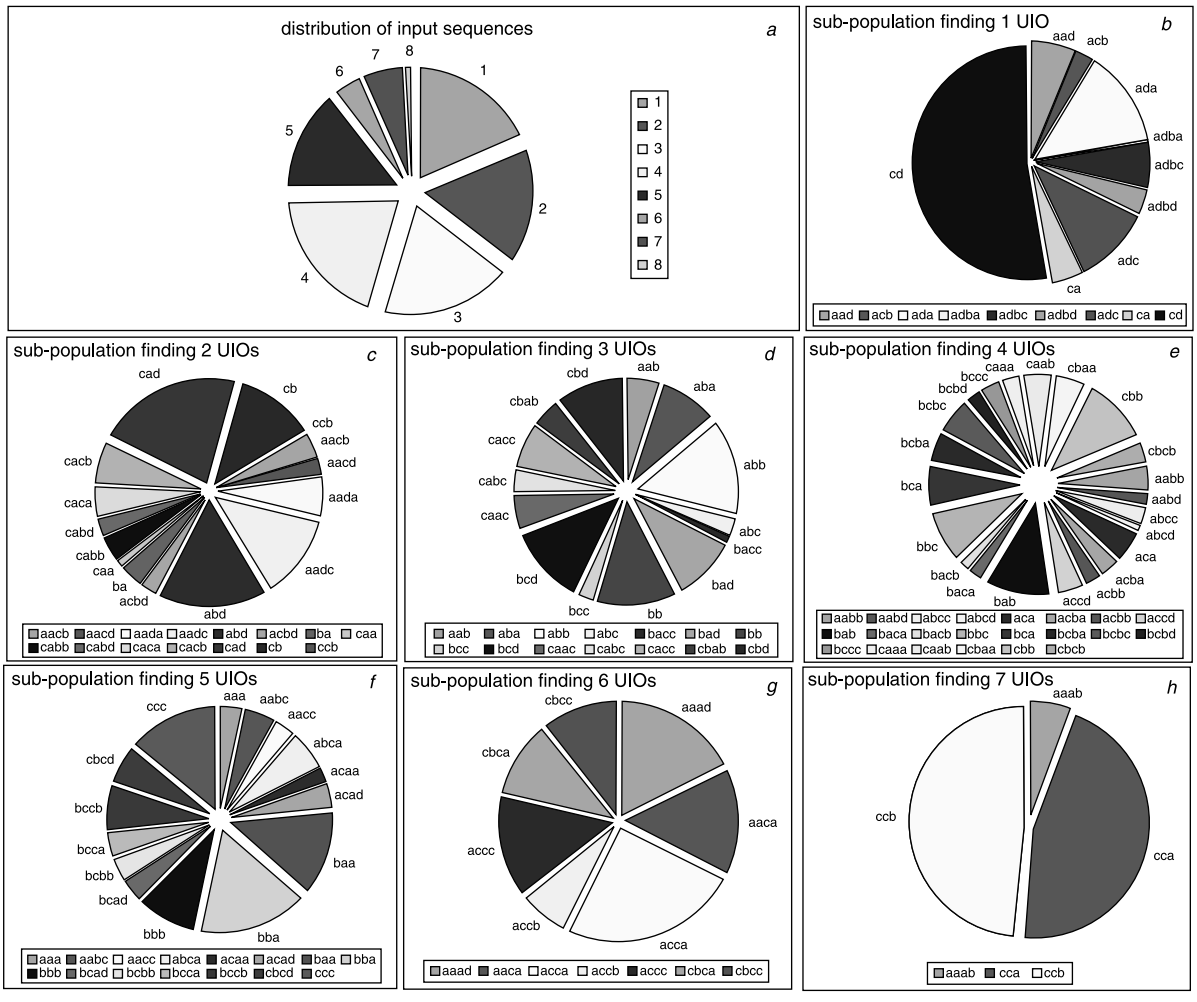


Fig. 18 *UIO distribution for SA with rough exponential temperature drop in Model 1*

rough temperature drop control. From these Figures it can be seen that, compared to the experiments using Model 1, the distribution of input sequences has no significant changes. Both temperature control schemes achieve a good performance. In the normal temperature drop experiment, three UIOs are missed, while two UIOs are missed in the rough temperature drop experiment.

Figures 21 and 22 present the average fitness values when constructing UIOs for Models 1 and 2 using the rough temperature drop control scheme, respectively. The Figures show that, when using Model 1, the genetic pool begins to converge after 250 generations while, when using Model 2, the genetic pool converges after 200 generations. Compared to Figs. 14 and 15, it can be seen that SA converges more slowly than GA.

5.3 General evaluation

Experimental results above suggest that, when constructing UIOs using GA and SA, without the sharing technique, the population is likely to converge at several individuals that have high fitness values. The distribution of such a population causes some UIOs to be missed with high probability. After applying the sharing technique, the population is encouraged to spread out and forms several subpopulations. These subpopulations are intended to cover all optima in the search space. The search quality significantly improved and more UIOs were found. Tables 3 and 4 give the number of UIOs that are missed for each experiment using GA, SA, GA with sharing and SA with sharing. From these Tables it can be seen that the sharing technique is effective in finding multiple UIOs. It can also be noted that the search performance is comparatively stable.

It has also been shown that, with the sharing technique, there is no significant difference between GA and SA on the search for UIOs. Both techniques force their populations to maintain diversity by forming subpopulations. In the two models under test, with the sharing technique, both GA and SA are effective. When applying SA, the rough exponential temperature drop seems to be better than the normal exponential temperature drop. Since the sharing technique tactically reduces some individuals' fitness values, some information might be lost during the computation. Local bumping in the rough exponential temperature drop gives the experiments a second chance for amendments, which might help to prevent such information from being lost. This could explain why the performance of the rough SA was consistently better than that of a simple SA.

The results on convergence rates imply that, when constructing UIOs, GA converges faster than SA. A simple GA works on population based exploration. New solutions (children) inherit information from previous solutions (parents) through a crossover operator, while SA generates a new solution based on the search in the neighbourhood of an existing solution. A simple GA is more exploitative than a SA. That might explain why GA converges faster than SA.

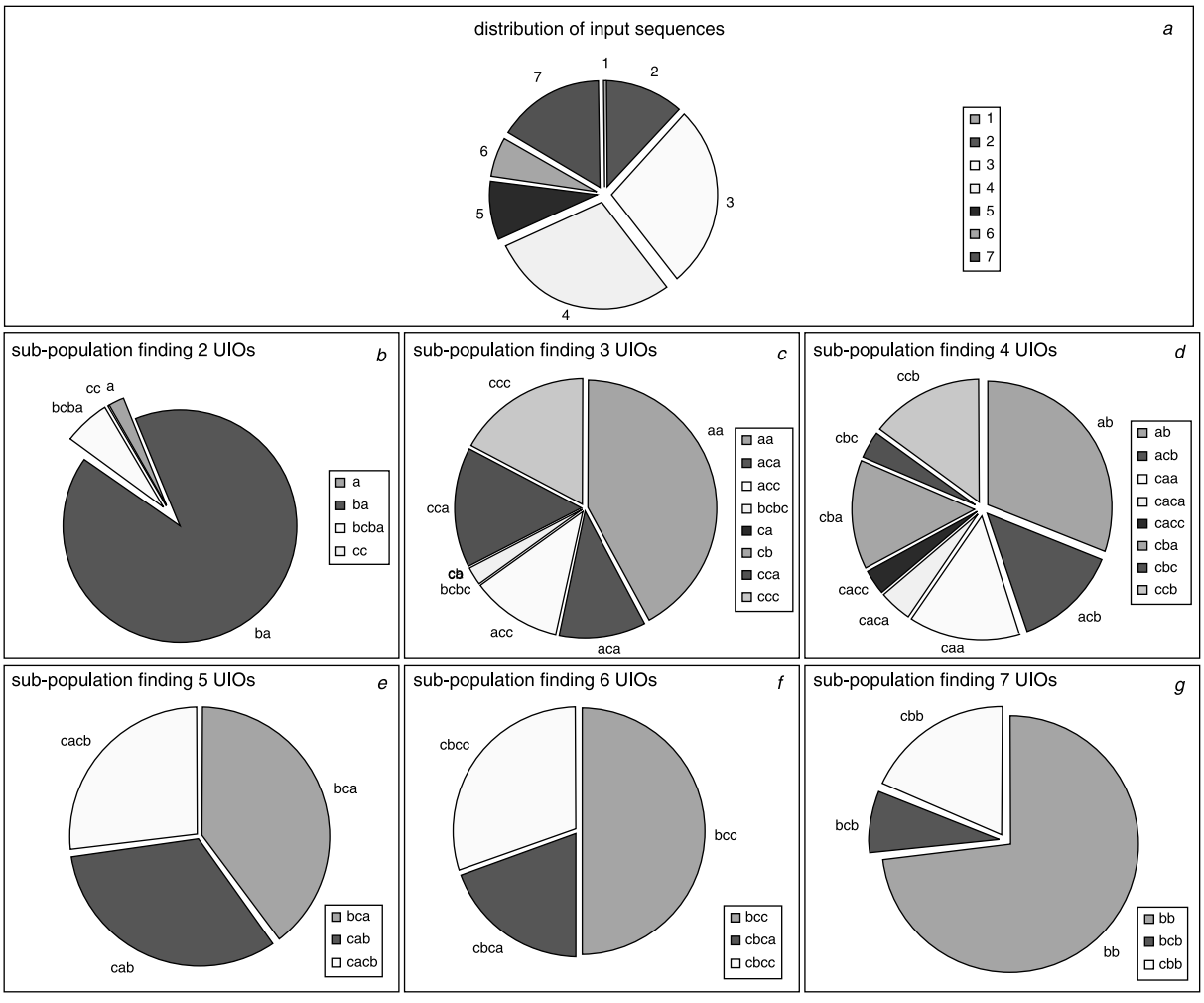


Fig. 19 *UIO distribution for SA with normal exponential temperature drop in Model 2*

5.4 Parameters settings

Parameter settings on crossover and mutation rates follow the suggestions from [13]; population size used in all experiments is fixed to 600. Using a larger size for a population may increase the chance of finding more UIOs, but it increases the computational cost as well. This paper did not investigate the effects on varying crossover rate, mutation rate and the population size. Future work will address these issues.

Parameter settings on α and γ affect the performance of computation significantly. γ is defined to control the dynamic behaviour of the exponential part in the fitness function, while α adjusts the weight of the linear part. To counteract the effect of $x_i e^{\delta x_i}$, γ must be set to a value that is greater than 1. However, it can also be noted that too big a value of γ causes the calculation of an individual's fitness a continuous decrement even when some discrete partitions are found. Therefore, a comparatively small value that is greater than 1 should be considered. In this work, we found that setting γ between 1.2 and 1.5 achieves better performance than other values.

α is defined to reward the partitions when no discrete class has been found. Normally, at the beginning of computation when no discrete class is found, the *linear part* plays the major role in the calculation of the fitness value. However, with the computation going further and some discrete classes being found, the *exponential part* takes over the role and becomes the major factor. Individuals that switch the role too slowly might obtain low fitness values and become unlikely to be selected for reproduction. This effect might cause some patterns (some UIOs) to be missed. For example, Fig. 23 shows two patterns of state splitting trees in the first model. In pattern A (corresponding to *aaaa*), there are five discrete units in the fourth layer (corresponding to the first three inputs) and three units in the fifth layer (corresponding to the first four inputs). In pattern B (corresponding to *ccac*), there is 1 discrete unit in the third layer (corresponding to the first two inputs) and six units in the fourth layer (corresponding to the first three inputs). *ccac* acquires a much higher fitness value than that of *aaaa*. *aaaa* is therefore likely to be missed during the computation. To compensate this effect, a comparatively high α value might be helpful since it enhances the effect of linear action. In this work, we set α to 20. We have also tested values that are below 15 and found that no experiment discovered the pattern A (*aaaa*).

Threshold value θ decides whether the fitness value of a candidate can be reduced. A value between 0 and 1 can be applied. The setting of θ should be suitable and may vary in different systems. If θ is set too low, candidates that are not fully represented by others may be degraded, causing some UIOs to be missed. For instance, *abcab* and *abaac* are two candidates. If θ is set less than 0.4, compared with *abcab*, the fitness value of *abaac* can be degraded. However, it can be seen that *abaac* is not fully represented by *abcab*. *abaac* might be missed in the computation due to inappropriate operations; at the same time, too high a value of θ might

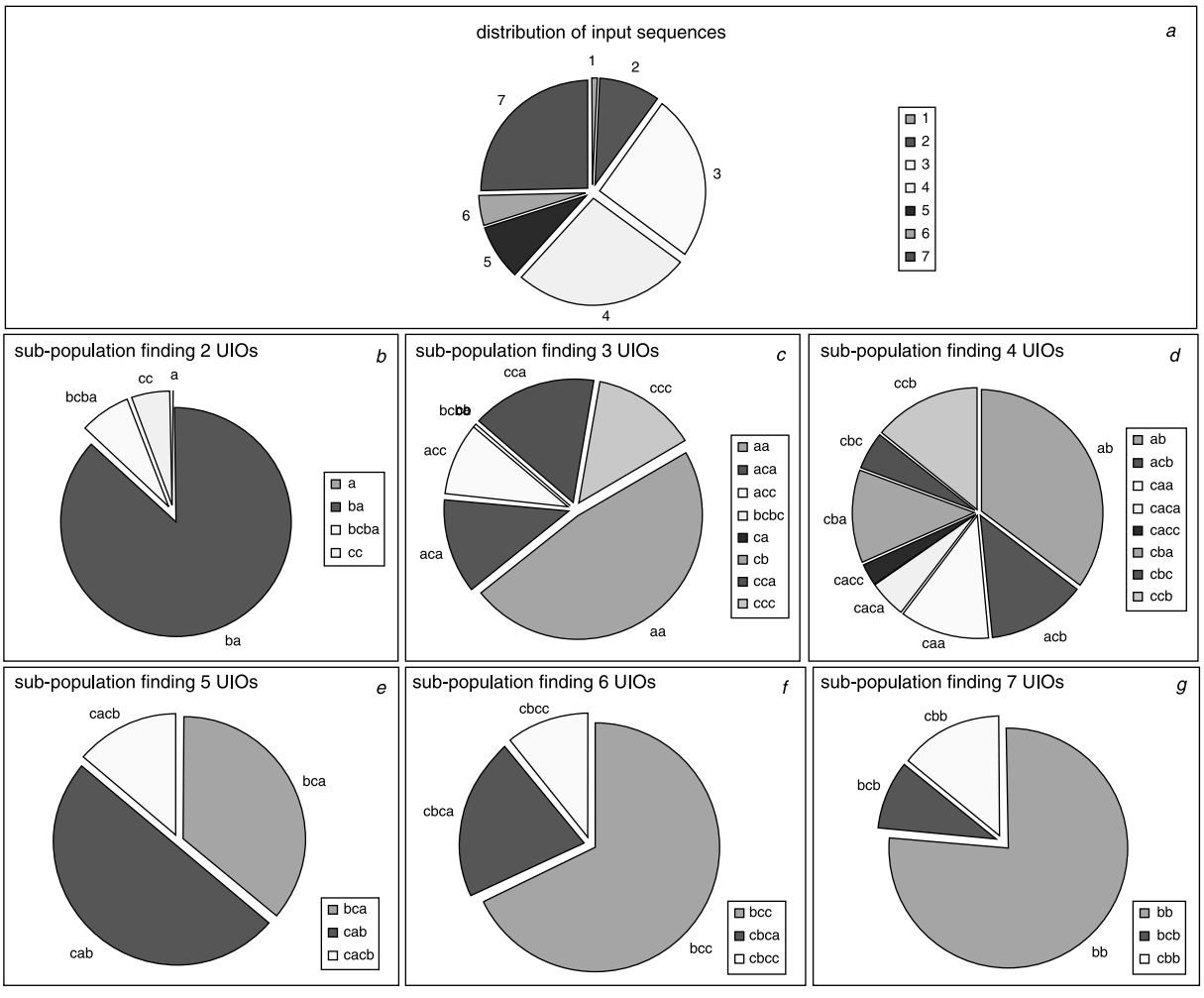


Fig. 20 *UIO distribution for SA with rough exponential temperature drop in Model 2*

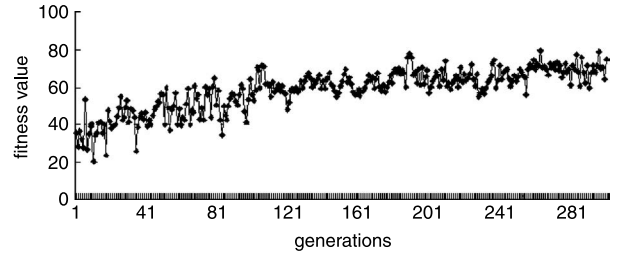


Fig. 21 *Average fitness values when constructing UIOs using SA (rough T drop): Model 1*

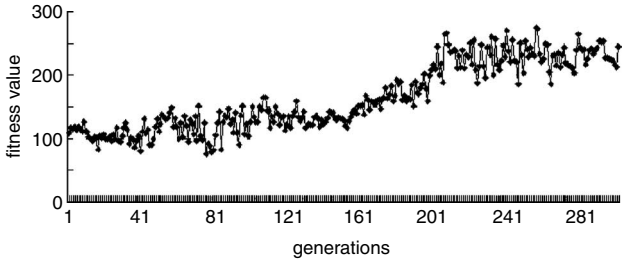


Fig. 22 *Average fitness values when constructing UIOs using SA (rought T drop): Model 2*

make the operation of fitness degrade ineffective. If θ is set to 1, no degrade action occurs. In our experiments, $2/3$ was used in the first model, while $1/2$ was selected for the second model; these values were chosen after some initial experiments.

Table 3: Missing UIOs of the first model

	GA	SA	GA/S	SA/N	SA/R
1	27	56	4	14	2
2	29	49	6	18	4
3	30	62	6	15	4
4	31	37	7	11	3
5	29	55	5	9	6
6	28	48	8	13	5
7	30	44	7	10	2
8	27	51	8	13	6
9	29	39	4	15	4
10	32	46	7	10	3
Avg	29.2	48.7	6.2	12.8	4.1

6 Conclusion

This paper investigated the use of metaheuristic optimisation techniques (MOTs), with sharing, in the generation of unique input/output sequences (UIOs) from a finite state machine (FSM). A fitness function, based on properties of a state splitting tree, guided the search for UIOs. A sharing

Table 4: Missing UIOs in Model 2

	GA	SA	GA/S	SA/N	SA/R
1	7	15	2	3	2
2	7	17	4	4	2
3	9	21	4	3	2
4	7	19	2	3	3
5	7	20	3	3	2
Avg	7.2	18.4	3	3.2	2

GA:simple GA without sharing; SA:simple SA without sharing; GA/S:GA with sharing; SA/N:SA with sharing using normal T drop; SA/R:SA with sharing using rough T drop

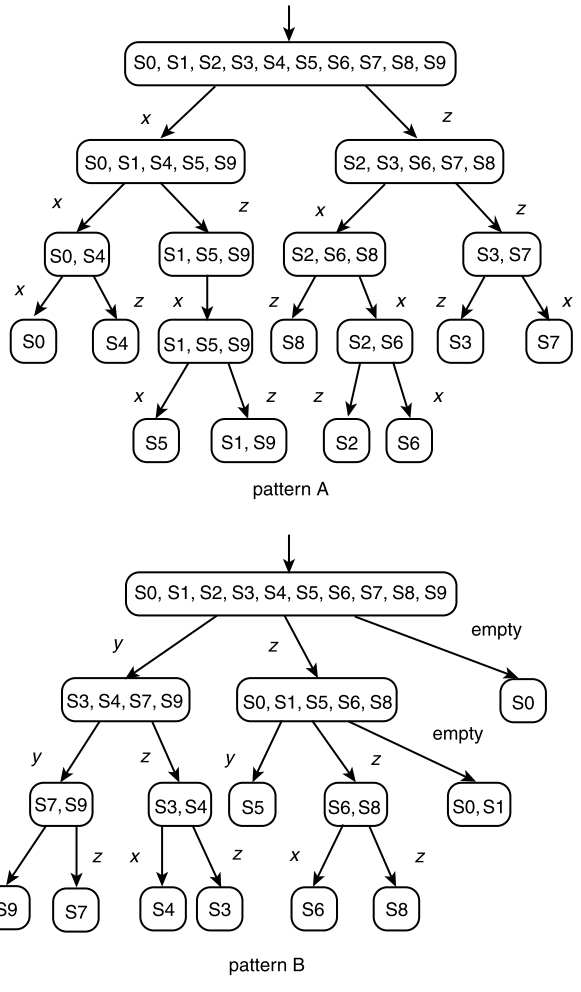


Fig. 23 *Two patterns of state splitting tree in the first model*

a State splitting tree for 'aaaa'

b State splitting tree for 'ccac'

technique was introduced to maintain the diversity in a population by defining a mechanism that measures the similarity of two sequences.

Two FSMs were used to evaluate the effectiveness of a genetic algorithm (GA), GA with sharing and simulated annealing (SA) with sharing. Experimental results show that, in terms of UIO distributions, there is no significant difference between GA with sharing and SA with sharing. Both outperformed GA without sharing. With the sharing technique, both GA and SA can force a population to form several subpopulations, and these are likely to cover many local optima. By finding more local optima, the search identifies more UIOs. However, a problem was also noted. All subpopulations were dominated by one or several individuals. More work needs to be carried out to study this problem.

7 References

1 Aho, A.V., Dahbura, A.T., Lee, D., and Uyar, M.U.: 'An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours', *IEEE Trans. Commun.*, 1991, **39**, (3), pp. 1604–1615

2 Hierons, R.M., and Ural, H.: 'UIO sequence based checking sequences for distributed test architectures', *Inf. Softw. Technol.*, 2003, **45**, pp. 793–803

3 Huang, C.M., Chiang, M.S., and Jiang, M.Y.: 'UIO: a protocol test sequence generation method using the transition executability analysis (TEA)', *Comput. Commun.*, 1998, **21**, pp. 1462–1475

4 Lee, D., and Yannakakis, M.: 'Testing finite state machines: state identification and verification', *IEEE Trans. Comput.*, 1994, **43**, (3), pp. 306–320

5 Pomeranz, I., and Reddy, S.M.: 'Functional test generation for full scan circuits'. Proc. Conf. on Design, Automation and Test in Europe, 2000, pp. 396–403

6 Sidhu, D.P., and Leung, T.K.: 'Formal methods for protocol testing: a detailed study', *IEEE Trans. Softw. Eng.*, 1989, **15**, (4), pp. 413–426

7 Shen, Y.N., Lombardi, F., and Dahbura, A.T.: 'Protocol conformance testing using multiple UIO Sequences', *IEEE Trans. Commun.*, 1992, **40**, (8), pp. 1282–1287

8 Yang, B., and Ural, H.: 'Protocol conformance test generation using multiple UIO sequences with overlapping'. ACM SIGCOMM: Communications, Architectures, and Protocols, Twente, The Netherlands, North-Holland, The Netherlands, 24-27 Sep, 1990, pp. 118–125

9 Miller, R.E., and Paul, S.: 'On the generation of minimal-length conformance tests for communication protocols', *IEEE/ACM Trans. Netw.*, 1993, **1**, (1), pp. 116–129

10 Hierons, R.M.: 'Extending test sequence overlap by invertibility', *Comput. J.*, 1996, **39**, (4), pp. 325–330

11 Hierons, R.M.: 'Testing from a finite-state machine: extending invertibility to sequences', *Comput. J.*, 1997, **40**, (4), pp. 220–230

12 Naik, K.: 'Efficient computation of unique input/output sequences in finite-state machines', *IEEE/ACM Trans. Netw.*, 1997, **5**, (4), pp. 585–599

13 Goldberg, D.E.: 'Genetic algorithms in search, optimization, and machine learning' (Addison-Wesley, Reading, MA, 1989)

14 Kirkpatrick, S., Gelatt, C.D., Jr., and Vecchi, M.P.: 'Optimization by simulated annealing', *Science*, 1983, **220**, (4598), pp. 671–680

15 Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., and Teller, E.: 'Equations of state calculations by fast computing machines', *J. Chem. Phys.*, 1953, **21**, pp. 1087–1092

16 Goldberg, D.E., and Richardson, J.: 'Genetic algorithms with sharing for multimodal function optimization'. Proc. 2nd Int. Conf. on Genetic Algorithms, Lawrence Erlbaum Associates, Hillsdale, NJ, 1987, pp. 41–49

17 Jones, B.F., Eyres, D.E., and Sthamer, H.H.: 'A strategy for using genetic algorithms to automate branch and fault-based testing', *Comput. J.*, 1998, **41**, (2), pp. 98–107

18 Michael, C.C., McGraw, G., and Schatz, M.A.: 'Generating software test data by evolution', *IEEE Trans. Softw. Eng.*, 2001, **27**, (12), pp. 1085–1110

19 Wegener, J., Sthamer, H., Jones, B.F., and Eyres, D.E.: 'Testing real-time systems using genetic algorithms', *Softw. Qual.*, 1997, **6**, (2), pp. 127–135

20 Tracey, N., Clark, J., Mander, K., and McDermid, J.: 'Automated test-data generation for exception conditions', *Softw. Pract. Exp.*, 2000, **30**, (1), pp. 61–79

21 Guo, Q., Hierons, R.M., Harman, M., and Derderian, K.: 'Computing unique input/output sequences using genetic algorithms'. Formal Approaches to Testing (FATES'03), *Lect. Notes Comput. Sci.*, 2004, **2931**, pp. 164–177

22 Lee, D., and Yannakakis, M.: 'Principles and methods of testing finite state machines – a survey', *Proc. IEEE*, 1996, **84**, (8), pp. 1090–1122

23 ITU-T, 'Recommendation Z.500 framework on formal methods in conformance testing', International Telecommunication Union, Geneva, Switzerland, 1997

24 Holland, J.H.: 'Adaptation in natural and artificial systems' (University of Michigan Press, Ann Arbor, MI, 1975)

25 Atkinson, A.C.: 'A segmented algorithm for simulated annealing', *Stat. Comput.*, (2), pp. 221–230

26 McGookin, E.W., Murray-Smith, D.J., and Li, Y.: 'Segmented simulated annealing applied to sliding mode controller design'. Proc. 13th World Congress of IFAC, San Francisco, USA, Vol. D, pp. 333–338