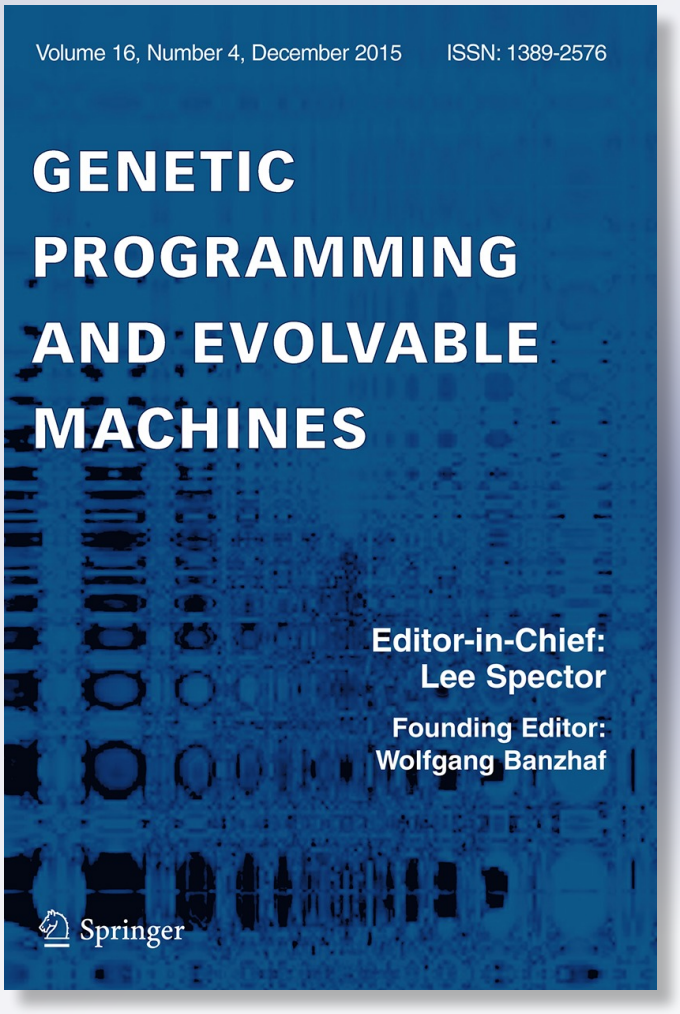
A learning automata-based memetic algorithm

M. Rezapoor Mirsaleh & M. R. Meybodi

Genetic Programming and Evolvable Machines

ISSN 1389-2576
Volume 16
Number 4

Genet Program Evolvable Mach (2015)
16:399-453
DOI 10.1007/s10710-015-9241-9



Springer

Genet Program Evolvable Mach (2015) 16:399–453
DOI 10.1007/s10710-015-9241-9

A learning automata-based memetic algorithm

M. Rezapoor Mirsaleh · M. R. Meybodi

Received: 9 January 2014 / Revised: 10 December 2014 / Published online: 21 January 2015

Abstract Combing a genetic algorithm (GA) with a local search method produces a type of evolutionary algorithm known as a memetic algorithm (MA). Combining a GA with a learning automaton (LA) produces an MA named GALA, where the LA provides the local search function. GALA represents chromosomes as object migration automata (OMAs), whose states represent the history of the local search process. Each state in an OMA has two attributes: the value of the gene (allele), and the degree of association with those values. The local search changes the degree of association between genes and their values. In GALA a chromosome's fitness is computed using only the value of the genes. GALA is a Lamarckian learning model as it passes on the learned traits acquired by its local search method to offspring by a modification of the genotype. Herein we introduce a modified GALA (MGALA) that behaves according to a Baldwinian learning model. In MGALA the fitness function is computed using a chromosome's fitness and the history of the local search recorded by the OMA states. In addition, in MGALA the learned traits are not passed to the offspring. Unlike GALA, MGALA uses all the information recorded in an OMA representation of the chromosome, i.e., the degree of association between genes and their alleles, and the value of a gene, to compute the fitness of genes. We used MGALA to solve two problems: object partitioning and graph isomorphism. MGALA outperformed GALA, a canonical MA, and an OMA-based method using computer simulations, in terms of solution quality and rate of convergence.

M. Rezapoor Mirsaleh (✉) · M. R. Meybodi
Soft Computing Laboratory, Computer Engineering and Information Technology Department, Amirkabir University of Technology, Tehran, Iran
e-mail: mrezapoorm@aut.ac.ir

M. R. Meybodi
e-mail: mmeybodi@aut.ac.ir

M. R. Meybodi
Institute for Research and Fundamental Sciences, School of Computer Science, Tehran, Iran

Keywords Learning automata (LA) · Local search · Memetic algorithm (MA) · Object migration automata (OMA)

1 Introduction

Exploration and exploitation are two main search goals. Exploration is important for ensuring global reliability: the whole of search space needs to be searched to provide a trustworthy estimate of the global optimum. Exploitation is important, because it focuses the search effort around the best solutions by searching their neighborhoods to find more accurate solutions [1]. Many search algorithms use a combination of a global search method and a local search method to achieve their goal. These algorithms are known as hybrid methods. The combination of a traditional genetic algorithm (GA) with local search methods that incorporate local improvement procedures can improve the performance of GAs. These hybrid methods are commonly known as memetic algorithms (MAs), or Baldwinian [2] or Lamarckian [3] evolutionary algorithms (EA). The particular local search method employed is the important aspect of these algorithms. In the Lamarckian approach the local search method is used as a refinement genetic operator that modifies the genetic structure of an individual and places it back in the genetic population [4]. Lamarckian evolution can increase the speed of search processes in genetic algorithms. However, it can damage schema processing by changing the genetic structure of individuals, which may lead to premature convergence [5, 6].

The Baldwinian learning approach improves the fitness of an individual by applying a local search, however, individual genotypes remain unchanged. Thus, it increases the individual's chances of remaining in subsequent generations. Similar to natural evolution, Baldwinian learning does not modify the genetic structure of an individual; but it does increase its chances of survival. Unlike the Lamarckian learning model, the Baldwinian approach does not allow parents to transfer what they have learned to their children [6]. The local search method is used as a part of the individual's evaluation process in the Baldwinian approach. The local search method uses local knowledge to create a new fitness that can be used by the global genetic operators to improve an individual's capability. In this method one or more individuals of a population that are similar in genotype gain similar fitness. These individuals, are probably near to each other in search space, and are equal in fitness after applying the local search. Therefore, the new search space will be a smooth surface, and will cover many of the local minima of the new search space. This fitness modification is known as the smoothing effect. The Baldwinian learning approach can be more effective, albeit slower, than Lamarckian approaches, since it does not alter the global search process of GAs [5].

Learning automata (LAs) are based on the general schemes of reinforcement learning algorithms. LAs enable agents to learn their interaction with an environment. They select actions via a stochastic process and apply them on a random, unknown environment. They can learn the best action by iteratively

performing and receiving stochastic reinforcement signals from the unknown environment. These stochastic responses from the environment show the favorability of the selected actions, and the LAs change their action selecting mechanism in favor of the most promising actions according to responses from the environment [7, 8].

GALA is a type of MA first reported by Rezapoor and Meybodi [9]. GALA combines a GA, used for its global search function (Exploration), with an LA, used for its local search function (Exploitation). Object migration automata (OMAs) represent chromosomes in GALA. Each state in an OMA has two attributes: the value of the gene, and the degree of association with its value. Information about the past history of the local search process shows the degree of association between genes and their values. GALA performs according to a Lamarckian learning model, because it modifies the genotype and only uses a chromosome's fitness to fitness function computation.

We present a new version of GALA, called modified GALA (MGALA), in the first part of this paper. MGALA behaves according to a Baldwinian learning model. Unlike GALA, which only uses the value of genes for fitness computation, MGALA uses all the information in the OMA representation of the chromosome (i.e., the degree of association between genes and their alleles, and the value of genes) to compute the fitness function. In the second part of the paper MGALA is used to solve two optimization problems: object partitioning and graph isomorphism. Computer simulations show that MGALA outperforms GALA, a canonical MA, and an OMA-based method, in terms of solution quality and in the rate of convergence.

Overall our paper is organized as follows: after this introduction Sect. 2 briefly describes learning automata and object migrating automata. GALA and its applications are described in Sect. 3. MGALA is introduced in Sect. 4. Two MGALA applications, those of solving the object partitioning problem and the graph isomorphism problem (GIP), are explained in Sects. 5 and 6, respectively. These two sections include implementation considerations, simulation results, and comparisons with other algorithms, which highlights MGALA's contributions to the field. Section 7 is the conclusion.

2 Learning automata and object migrating automata

2.1 Learning automata

A learning automaton (LA) [5] is an adaptive decision-making unit. It can be described as determination of an optimal action from a set of actions through repeated interactions with an unknown random environment. It selects an action based on a probability distribution at each instant and applies it on a random environment. The environment sends a reinforcement signal to automata after evaluating the input action. The learning automata process the response of environment and update its action probability vector. By repeating this process, the automaton learns to choose the optimal action so that the average penalty obtained

from the environment is minimized. The environment is represented by a triple $<\underline{\alpha}, \underline{\beta}, \underline{c}>$. $\underline{\alpha} = \{\alpha_1, \ldots, \alpha_r\}$ is the finite set of the inputs, $\underline{\beta} = \{0, 1\}$ is the set of outputs that can be taken by the reinforcement signal, and $\underline{c} = \{c_1, \ldots, c_r\}$ is the set of the penalty probabilities, where each element c_i of $\underline{c}$ is corresponds to one input action α_i. The input $\alpha(n)$ to the environment belongs to $\underline{\alpha}$ and may be considered to be applied to the environment at discrete time $t = n\,(n = 0, 1, 2, \ldots)$. The output $\beta(n)$ of the environment belongs to $\underline{\beta}$ and can take on one of two values 0 and 1. An $\beta(n) = 1$ is identified with a failure or an unfavorable response and $\beta(n) = 0$ with a success or favorable response of the environment. The element c_i of $\underline{c}$ which characterizes the environment may then be defined by $pr(\beta(n) = 1|\alpha(n) = \alpha_i) = c_i \quad (i = 1, 2, \ldots, r)$. When the penalty probabilities are constant, the random environment is said a stationary random environment. It is called a non stationary environment, if they vary with time. Figure 1 shows the relationship between the LA and the random environment.

There are two main families of learning automata [6]: fixed structure learning automata and variable structure learning automata. First, we formally define fixed structure learning automata and then some of the fixed structures learning automata such as Tsetline, Krinsky, and Krylov automata are described.

A fixed structure LA is represented by a quintuple $<\underline{\alpha}, \underline{\Phi}, \underline{\beta}, F, G>$. where:

- $\underline{\alpha} = \{\alpha_1, \ldots, \alpha_r\}$ is the set of actions that it must choose from.
- $\underline{\Phi} = \{\varphi_1, \ldots, \varphi_s\}$ is the set of internal states.
- $\underline{\beta} = \{0, 1\}$ is the set of inputs where 1 represents a penalty and 0 represents a reward.
- $F : \underline{\Phi} * \underline{\beta} \rightarrow \underline{\Phi}$ is a function that maps the current state and current input into the next state.
- $G : \underline{\Phi} \rightarrow \underline{\alpha}$ is a function that maps the current state into the current output. In other words, G determines the action taken by the automaton.

The operation of fixed learning automata could be described as follows: At the first step, the selected action $\alpha(n) = G[\Phi(n)]$ serves as the input to the environment, which in turn emits a stochastic response $\beta(n)$ at the time n. $\beta(n)$ is an element of $\underline{\beta} = \{0, 1\}$ and is the feedback response of the environment to the automaton. In the

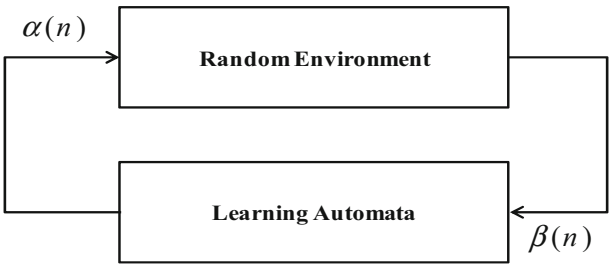


Fig. 1 The relationship between the LA and random environment

second step, the environment penalize (i.e., $\beta(n) = 1$) the automaton with the penalty probability c_i, which is the action dependent. On the basis of the response $\beta(n)$, the state of automaton is updated by $\Phi(n+1) = F[\Phi(n), \beta(n)]$. This process continues until the desired result is obtained.

In the following paragraphs, we describe some of the fixed structure learning automata such as Tsetline, Krinsky, and Krylov automata.

2.1.1 The two-state automata ($L_{2,2}$)

This automaton has two states, φ_1 and φ_2 and two actions α_1 and α_2. The automaton accepts input from a set of {0,1} and switches its states upon encountering an input 1 (unfavorable response) and remains in the same state on receiving an input 0 (favorable response). An automaton that uses this strategy is refereed as $L_{2,2}$ where the first subscript refers to the number of states and second subscript to the number of actions.

2.1.2 The Tsetline automata (the two-action automata with memory $L_{2N,2}$)

Tsetline suggested a modification of $L_{2,2}$ denoted by $L_{2N,2}$. This automaton has *2 N* states and two actions and attempts to incorporate the past behavior of the system in its decision rule for choosing the sequence of actions. While the automaton $L_{2,2}$ switches from one action to another on receiving a failure response from environment, $L_{2N,2}$ keeps an account of the number of success and failures received for each action. It is only when the number of failures exceeds the number of successes, or some maximum value *N*; the automaton switches from one action to another. The procedure described above is one convenient method of keeping track of performance of the actions α_1 and α_2. As such, *N* is called depth of memory associated with each action, and automaton is said to have a total memory of *2N*. For every favorable response, the state of automaton moves deeper into the memory of corresponding action, and for an unfavorable response, moves out of it. This automaton can be extended to multiple action automata. The state transition graph of $L_{2N,2}$ automaton is shown in Fig. 2.

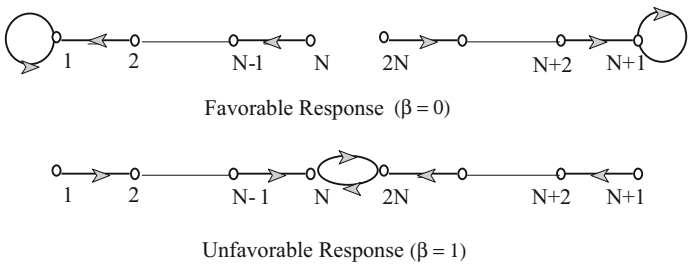


Fig. 2 The state transition graph for $L_{2N,2}$ (Tsetline automaton)

2.1.3 The Krinsky automata

This automaton behaves exactly like $L_{2N,2}$ automaton when the response of the environment is unfavorable, but for favorable response, any state φ_i (for $i = 1,\ldots,N$) passes to the state φ_1 and any state φ_i ($i = N + 1, 2N$) passes to the state φ_{N+1}. This implies that a string of N consecutive unfavorable responses are needed to change from one action to another. The state transition graph of Krinsky automaton is shown in Fig. 3.

2.1.4 The Krylov automata

This automaton has state transitions that are identical to the $L_{2N,2}$ automaton when the output of the environment is favorable. However when the response of the environment is unfavorable, a state φ_i ($i \neq 1, N, N+1, 2N$) passes to a state φ_{i+1} with probability 1/2 and to state φ_{i-1} with probability 1/2, as shown in Fig. 4. When $i = 1$ or $N + 1$, φ_i stays in the same state with probability 1/2 and moves to φ_{i+1} with the same probability. When $i = N$, φ_N moves to φ_{N-1} and φ_{2N} each with probability 1/2 and similarly, When $i = 2N$, φ_{2N} moves to φ_{2N-1} and φ_N each with probability 1/2. The state transition graph of Krylov automaton is shown in Fig. 4.

Object migration automaton (OMA) that is an example of fixed structure learning automata is described in the next section. Learning automata have a vast variety of applications in combinatorial optimization problems [8–10], computer networks [10–13], queuing theory [14, 15], signal processing [16, 17], information retrieval

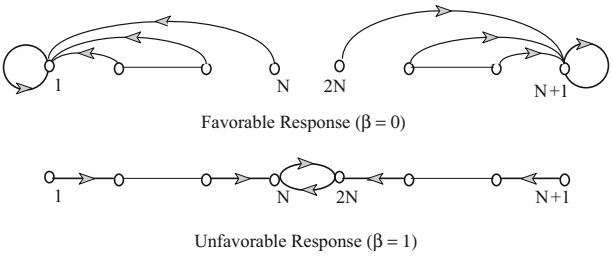


Fig. 3 The state transition graph for Krinsky automaton

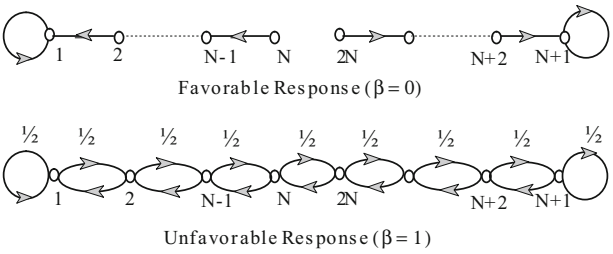


Fig. 4 The state transition graph for Krylov automaton

[18, 19], adaptive control [20–22], neural networks engineering [23, 24] and pattern recognition [25–27].

2.2 Object migration automata

Object migration automata were first proposed by Oommen and Ma [10]. OMAs are a type of fixed structure learning automata, and are defined by a quintuple $<\underline{\alpha}, \underline{\Phi}, \underline{\beta}, F, G>$.$\underline{\alpha} = \{\alpha_1, \ldots, \alpha_r\}$. is the set of allowed actions for the automaton. For each action α_k, there is a set of states $\{\varphi_{(k-1)N+1}, \ldots, \varphi_{kN}\}$, where N is the depth of memory. The states $\varphi_{(k-1)N+1}$ and φ_{kN} are the most internal state and the boundary state of action α_k, respectively. The set of all states is represented by $\underline{\Phi} = \{\varphi_1, \ldots, \varphi_s\}$, where $s = N * r$. $\underline{\beta} = \{0, 1\}$ is the set of inputs, where 1 represents an unfavorable response, and 0 represents a favorable response. $F : \underline{\Phi} * \underline{\beta} \rightarrow \underline{\Phi}$ is a function that maps the current state and current input into the next state, and $G : \underline{\Phi} \rightarrow \underline{\alpha}$ is a function that maps the current state into the current output. In other words, G determines the action taken by the automaton. W objects are assigned to actions in an OMA and moved around the states of the automaton, as opposed to general learning automata, in which the automaton can move from one action to another by environmental response. The state of objects is changed on the basis of the feedback response from the environment. If the object w_i is assigned to action α_k (i.e., w_i is in state ξ_i, where $\xi_i \in \left\{\varphi_{(k-1)N+1}, \ldots, \varphi_{kN}\right\}$), and the feedback response from the environment is 0, α_k is rewarded, and w_i is moved toward the most internal state $(\varphi_{(k-1)N+1})$ of that action. If the feedback from the environment is 1, then α_k is penalized, and w_i is moved toward the boundary state (φ_{kN}) of action α_k. The variable γ_k denotes the reverse of the state number of the object assigned to action α_k (i.e., degree of association between action α_k and its assigned object). By rewarding an action, the degree of association between that action and its assigned object will be increased. Conversely, penalizing an action causes the degree of association between that action and its assigned object to be decreased. An object associated with state $\varphi_{(k-1)N+1}$ has the most degree of association with action α_k, and an object associated with state φ_{kN} has the least degree of association with action α_k.

3 GALA

GALA, which is a hybrid model based on a GA and an LA, was introduced for the first time by Rezapoor and Meybodi [9]. Chromosomes are represented by OMAs in this model. In the OMA-based representation, there are n actions in each automaton corresponding to n genes in each chromosome. Furthermore, for each action, there are a fixed number of states N. The value of each gene, as a migratory object in the automata, is selected from the set $W = \{w_1,\ldots,w_m\}$ and assigned to states of corresponding action. After applying a local search, if the assignment of an object to the states of an action is promising, then the automaton is rewarded and the assigned

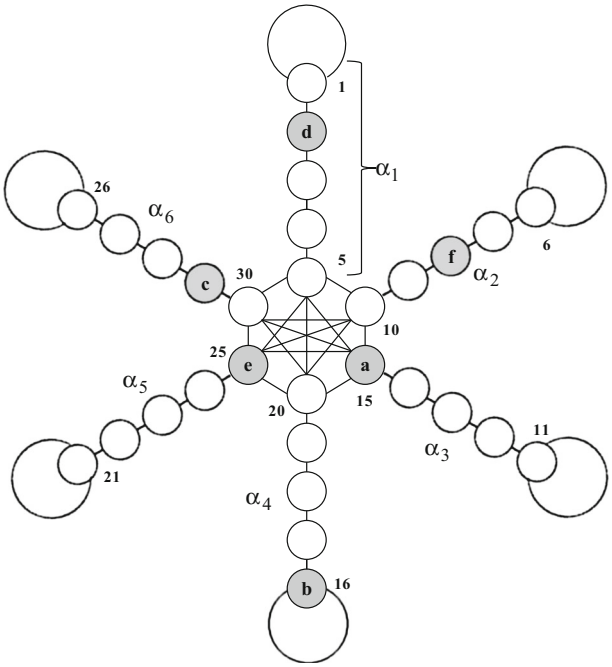


Fig. 5 The state transition graph of a Tsetline-based OMA

object moves toward the most internal state of that action; otherwise, the automaton is penalized and the assigned object moves toward the boundary state of that action. The rewarding and penalizing of an action changes the degree of association between an object and its action. Figure 5 shows a representation of chromosome "dfabec" using the Tsetline automaton-based OMA with six actions and a depth of memory of five.

In Fig. 5 there are six actions (genes), denoted by α_1, α_2, α_3, α_4, α_5, and α_6. Genes 1, 2, and 6 possess values 'd,' 'f,' and 'c,' located at internal states 2, 3, and 4 of their actions, respectively. The value of genes 3 and 5 are 'a' and 'e' respectively, and both of them are located at the boundary states of their actions. Consequently, there is a minimum degree of association between these actions and their corresponding object. The remaining action, gene 4, has a value of 'b' and is located at the most internal state of its action. That is, it has the maximum degree of association with action 4. Representation of chromosomes based on other fixed structure learning automata is also possible. In a Krinsky-based OMA representation, as shown in Fig. 3, the object will be associated with the most internal state (i.e., it gets the highest degree of association with the corresponding action) when it is rewarded, and moves according to the Tsetline automaton-based OMA when it is penalized. In the representation of a Krylov OMA shown in Fig. 4, the object moves either toward the most internal state, or toward the boundary state, with a probability 0.5 toward penalty, and moves according to the Tsetline automaton-based OMA upon reward.

3.1 Global search in GALA

The global search in GALA is based on a traditional genetic algorithm. A population of chromosomes is represented by an OMA. Chromosome i is denoted by $CR_i = [(CR_i.Action(1), CR_i.Object(1), CR_i.State(1)),\ldots,(CR_i.Action(n), CR_i.Object(n), CR_i.State(n))]$, where $CR_i.Action(k)$ is the kth action of CR_i, $CR_i.Object(k)$ is the object assigned to the k_{th} action (the value of the kth gene), and $CR_i.State(k)$ is the state of the object assigned to the k_{th} action (the degree of association between gene k and its value), specifying $1 \leq k \leq n$, $1 \leq CR_i.Object(k) \leq m$, and $(k-1)N + 1 \leq CR_i.State(k) \leq kN$. The initial population is created randomly and objects are located at the boundary state of their actions. At the beginning of each generation the best chromosome from the previous generation is moved to the population of the current generation. Next, the crossover operator is applied to the parent chromosomes at rate r_c (parents are selected according to the chromosome's fitness using a tournament mechanism), and then the mutation operator is applied at rate r_m.

3.2 Crossover operator

The crossover operator in GALA is applied as follows: Two chromosomes, CR_1 and CR_2, are selected by the selection mechanism as parent chromosomes. Two actions, r_1 and r_2, are also randomly selected from CR_1 and CR_2, respectively. Then for each action in the range of $[r_1,r_2]$ of CR_1, the assigned object is exchanged with an assigned object of the same action in chromosome CR_2. In the crossover operator, the previous states of the selected actions in CR_1 and CR_2 are transferred to the child chromosomes. The pseudo code for the crossover operator is shown in Fig. 6.

Figure 7 illustrates an example of a crossover operator. First, two actions are randomly selected in the parent chromosomes (e.g. actions 2 and 4 here), and then objects are assigned to all actions in the range of [2,4] in CR_1, and are exchanged with the objects of the corresponding actions in CR_2.

3.3 Mutation operator

The mutation operator is the same as in a traditional genetic algorithm. Two actions are selected randomly in the parent chromosome, and then their assigned objects are exchanged. The previous states of selected actions in the parent chromosome are transferred to the child chromosomes in this operator. Pseudo code for the mutation operator is shown in Fig. 8.

```
Procedure Crossover (CR1, CR2)
    Generate two random numbers r1 and r2 in [1, N] where r1 < r2;
    For i = r1 to r2 do
        Swap(CR1.Object(CR1.Action(i)), CR2.Object(CR2.Action(i)));
        Swap(CR1.State(CR1.Action(i)), CR2.State(CR2.Action(i)));
    End For
End Crossover
```

Fig. 6 Pseudo code for a crossover operator

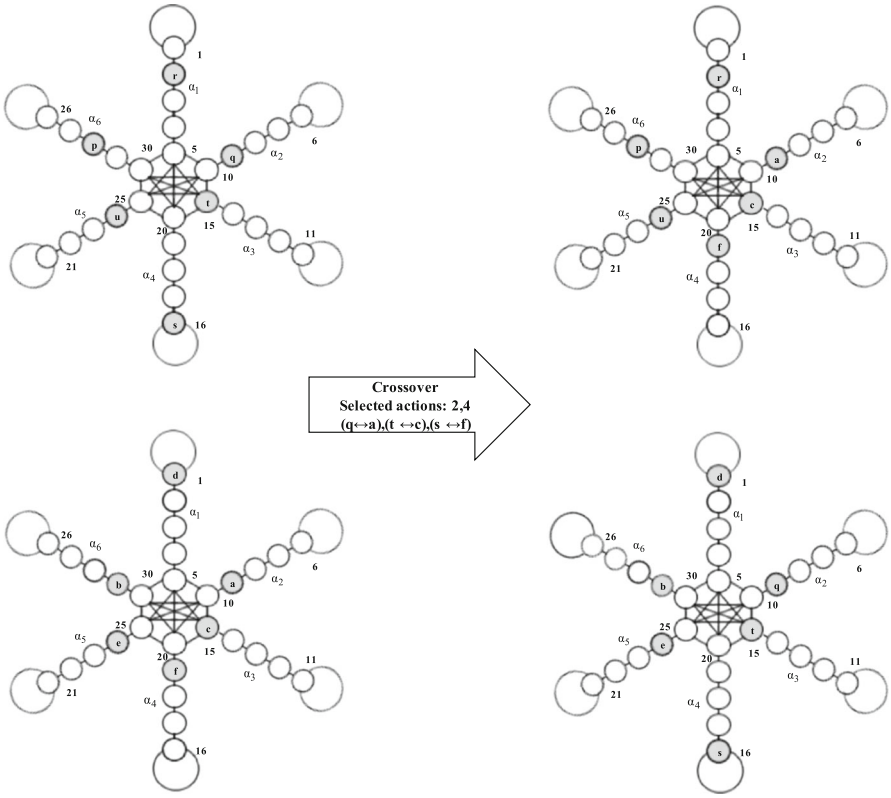


Fig. 7 An example of crossover operator

Procedure Mutation (CR)
Generate two random numbers r_1 and r_2 in $[1,n]$ where $r_1 < r_2$;
Swap($CR.Object(CR.Action(r_1))$, $CR.Object(CR.Action(r_2))$);
Swap($CR.State(CR.Action(r_1))$, $CR.State(CR.Action(r_2))$);
End Mutation

Fig. 8 Pseudo code for a mutation operator

Figure 9 illustrates an example of a mutation operator. First two actions in the parent chromosome are randomly selected (e.g. actions 1 and 2 here). The mutation operator exchanges both the state and the object assigned to action 1 with the state and the object assigned to action 2.

3.4 Local learning in GALA

Local learning in GALA is done using OMA representations of chromosomes. If the objects assigned to an action are the same before and after applying a given local

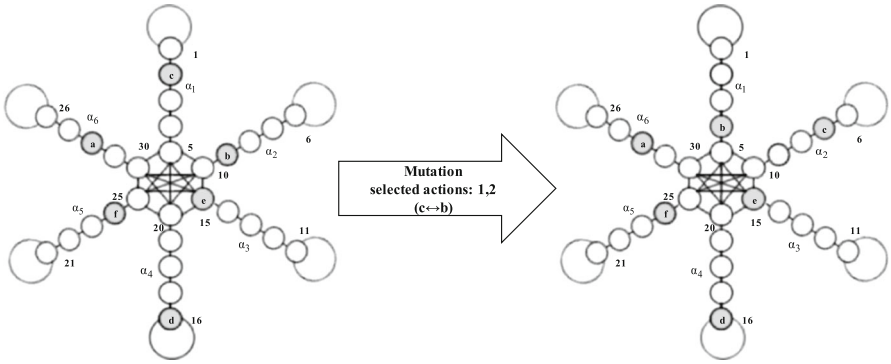


Fig. 9 A sample of mutation operator

```
Procedure Reward( CR, u )
    If (CR.State(u)-1) mod N <> 0 then
        Dec(CR.State(u)); //Decrement the state number of action (the state of action move toward the most internal
state)
    End If
End Reward
```

Fig. 10 Pseudo code for a reward function

search, then that action will be rewarded; otherwise, it will be penalized. It is worth noting that a local search only changes the state of the actions (according to the OMA connections), not the objects assigned to the actions (i.e., the action associated with an object will not change). By rewarding an action, the state of that action will move toward the most internal state according to the OMA connection. This causes the degree of association between an object, and its corresponding action, to be increased. The state of an action remains unchanged, if the object is located at its most internal state, such as the state of object D in action 4, shown in Fig. 11.

Figure 10 provides pseudo code for rewarding an action. Figure 11 illustrates an example of rewarding an action.

Penalizing an action causes the degree of association between an object and its corresponding action to be decreased. If an object is not in the boundary state of its action, then penalizing causes the object assigned to the action to move toward the boundary state. This means that the degree of association between the action and the corresponding object will be decreased (Fig. 12). If an object is in the boundary state of its action, then penalizing the action causes the object assigned to that action to change, and results in the creation of a new chromosome. How a new chromosome is created depends on the application. A new chromosome is always created in such a way that its fitness becomes greater than the fitness of the old chromosome. Figure 13 shows the effect of the penalty function on action 3 of a sample chromosome (assuming that chromosome "cbadfe" has better fitness than chromosome "cbedfa"). Pseudo code for the penalty function is shown in Fig. 14. The pseudo code for GALA is shown in Fig. 15.

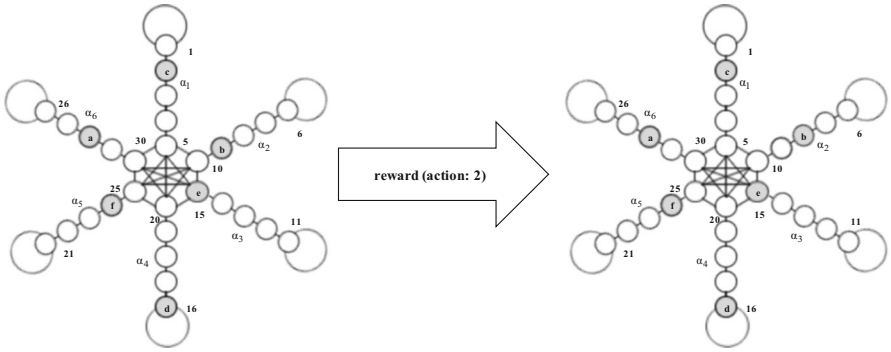


Fig. 11 An example of a reward function

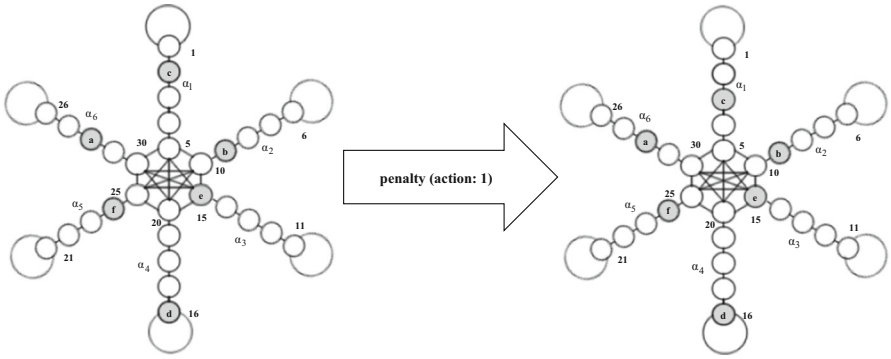


Fig. 12 An example of a penalty function

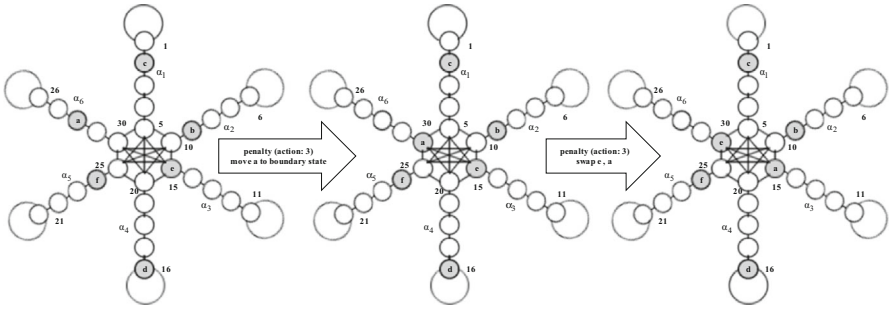


Fig. 13 Another example of a penalty function

3.5 Applications of GALA

GALA has been used in a variety of applications, including the GIP [11], join ordering problems in database queries [12–15], the traveling salesman problem

```
Procedure Penalize( CR, u )
  If (CR.State(u)) mod N <> 0 then
    Inc(CR.State(u)); //Increment the state number of action (the state of action move toward the boundary state)
  Else
    Find action U of chromosome CR in such a way that swapping the object assigned to u and object assigned to U
causes the fitness of CR be minimized.
    CR.State(U) = CR.Action(U)*N; //the state of action changed to the boundary state
    CR.State(u) = CR.Action(u)*N; //the state of action changed to the boundary state
    Swap(CR.Object(CR.Action(u)), CR.Object(CR.Action(U)));
  End If
End Penalize
```

Fig. 14 Pseudo code for the penalty function

```
Function GALA
t ← 0;
Init population P(t) with size ps; //Create an initial population P(0) of chromosomes P(0).CR1 ... P(0).CRps;
EvaluateFitness ();    // Evaluate fitness of all chromosomes of initial population
While (while termination criteria is not satisfied) do
   P(t+1).CR1 ← Best chromosome of P(t); //the best chromosome move to population of next generation
   For i = 2 to ps do
      Select parents CR1, CR2 based on tournament mechanism from P(t);
      NewCR←Crossover(CR1, CR2);
      NewCR←Mutation(NewCR);
      TempCR←LocalSearch(NewCR);
      For j=1 to n do
         If (TempCR.Object(TempCR.Action (j)) = NewCR.Object (NewCR.Action (j))) then
            Reward(NewCR, NewCR.Action (j));
         Else
            Penalize(NewCR, NewCR.Action (j));
         End If
         P(t+1).CRi ← NewCR;
      End For
   End For
t ← t + l;
End While
End GALA
```

Fig. 15 Pseudo code for GALA

[16–18], the Hamiltonian cycles problem [19], sorting problems in graphs [20], the graph bandwidth minimization problem [21–23], software clustering problems [24, 25], the single machine total weighted tardiness scheduling problem [26], data allocation problems in distributed database systems [27, 28], and the task graph scheduling problem [29, 30].

4 Modified GALA (MGALA)

Modified GALA (MGALA) is a new version of GALA. Like GALA each chromosome is represented by an object migration automaton (OMA) whose states keep information about the past history of the local search process. Each state in the OMA has two attributes: the value of the corresponding gene, and the degree of association of the gene with its value. In MGALA the fitness function is computed using the past history of the local search kept in the OMA states, as well as the chromosome's fitness. Unlike GALA, which only uses the value of the genes for

fitness computation, MGALA uses all the information in the OMA representation of the chromosome (i.e., the degree of association between genes and their values) to compute the fitness function. Hence, unlike GALA, which behaves according to a Lamarckian learning model it behaves according to a Baldwinian learning model. MGALA's various components will be described in the remainder of this section.

4.1 Fitness function

The fitness function in MGALA is not only dependent on genotype information, but also on phenotype information. We use the fitness function $f^{'}(CR) = \sum_{i=1}^{n} f_i(1+\gamma_i)$ for maximization problems, and $f^{'}(CR) = \sum_{i=1}^{n} f_i(1-\gamma_i)$ for minimization problems in the selection of chromosomes. In these functions f_i is the fitness of the *ith* gene, and γ_i is the degree of association between action α_i and its assigned object. The depth of memory is N, so we have $\frac{1}{N} \leq \gamma_i \leq 1$. The parent chromosomes and the chromosomes of the next generation are selected based on the defined fitness function using a tournament mechanism.

4.2 Mutation operator

Depending on whether the state of the selected actions changes or not, we define two types of mutation operators in MGALA. In the first type the states of the selected actions remain unchanged (i.e., the degree of association between actions and their assigned objects are saved). In second class the states of the selected actions are changed (i.e., the degree of association between actions and their assigned objects are lost).

MGALA has three mutation operators defined in this paper: SS-Mutation, XS-Mutation, and LS-Mutation. Specifically:

- The mutation operator in which the previous state of selected actions can be saved is referred to as the SS-Mutation.
- The mutation operator in which the previous state of selected actions can be exchanged is referred to as the XS-Mutation.
- The mutation operator in which the previous state of selected actions can be lost is referred to as the LS-Mutation.

The SS-Mutation and XS-Mutation are examples of the first type of mutation operator described above, and the LS-Mutation is an example of the second type. These mutation operators are described in more detail below.

4.2.1 SS-Mutation

Assuming actions 1 and 2 are the selected actions, the SS-Mutation exchanges the objects assigned to the states of actions 1 and 2. Figure 16 shows an example of an SS-Mutation. Pseudo code for our SS-Mutation is shown in Fig. 17.

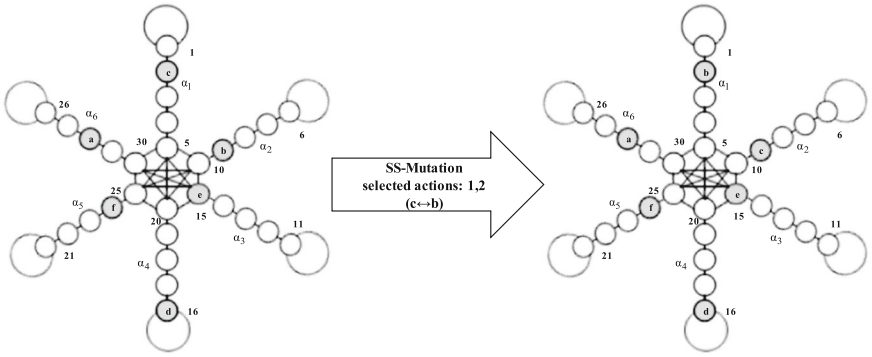


Fig. 16 An example of an SS-Mutation

Procedure SS-Mutation (*CR*)
Generate two random numbers r_1 and r_2 in *[1,n]* where $r_1 < r_2$;
Swap(*CR.Object(CR.Action*(r_1)), *CR.Object(CR.Action*(r_2)));
End SS-Mutation

Fig. 17 Pseudo code for the SS-Mutation

4.2.2 XS-Mutation

The XS-Mutation is the same as the mutation that GALA uses. The states of the actions, along with their assigned objects, are exchanged. Figure 18 shows an example of an XS-Mutation. Assuming actions 1 and 2 are the selected actions, the XS-Mutation exchanges the object and the state of action 1 with those of action 2. Pseudo code for our XS-Mutation is shown in Fig. 19.

4.2.3 LS-Mutation

The LS-Mutation is the same as that used in GALA, except that the state of each action is changed to become its corresponding boundary state. Figure 20 shows an example of an LS-Mutation operator. Assuming that actions 1 and 2 are the selected actions, the LS-Mutation operator causes: 1) The object assigned to the state of action 1 is exchanged with the assigned object of the action; and 2) The state of each action changes to become its corresponding boundary state. Pseudo code for our LS-Mutation operator is given in Fig. 21.

4.3 Crossover operator

Similar to the mutation operators, we define two different types of crossover operators for MGALA. In the first type, the states of selected actions remain unchanged, and in the second type the states of the actions change to the boundary states.

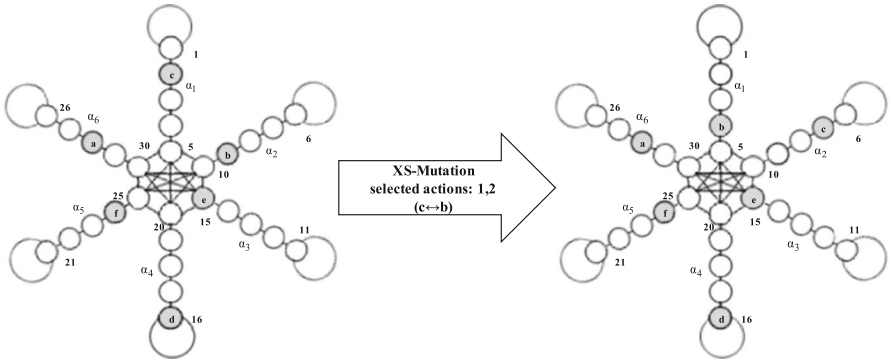


Fig. 18 An example of an XS-Mutation

Procedure XS-Mutation (*CR*)
Generate two random numbers r_1 and r_2 in *[1,n]* where $r_1 < r_2$;
Swap(*CR.Object(CR.Action*(r_1)*), CR.Object(CR.Action*(r_2)*)*);
Swap(*CR.State(CR.Action*(r_1)*), CR.State(CR.Action*(r_2)*)*);
End XS-Mutation

Fig. 19 Pseudo code for the XS-Mutation

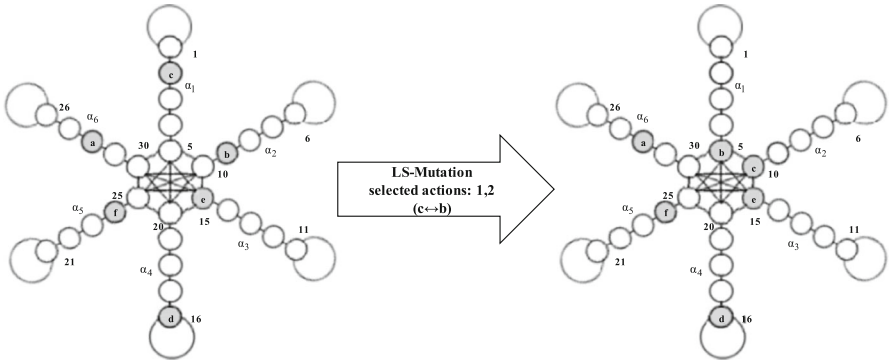


Fig. 20 An example of an LS-Mutation

Procedure LS-Mutation (*CR*)
Generate two random numbers r_1 and r_2 in *[1,n]* where $r_1 < r_2$;
Swap(*CR.Object(CR.Action*(r_1)*), CR.Object(CR.Action*(r_2)*)*);
CR.State(CR.Action(r_1)*) = CR.Action*(r_1)**N*; //the state of action changed to the boundary state
*CR.State(CR.Action(r2)) = CR.Action(r2)*N*; //the state of action changed to the boundary state
End LS-Mutation

Fig. 21 Pseudo code for the LS-Mutation

Three crossover operators, SS-Crossover, XS-Crossover, and LS-Crossover, are defined in MGALA:

- The crossover operator in which the previous state of selected actions can be saved is referred to as the SS-Crossover.
- The crossover operator in which the previous state of selected actions can be exchanged is referred to as the XS-Crossover.
- The crossover operator in which the previous state of selected actions can be lost is referred to as the LS-Crossover.

The SS-Crossover and XS-Crossover are examples of crossover operators of the first type, and the LS-Crossover is an example of the second type. These crossover operators are described in further detail below.

4.3.1 SS-Crossover

Figure 22 shows an example of an SS-Crossover. Assuming actions 2 and 4 are selected randomly from the parent chromosomes, the SS-Crossover exchanges the

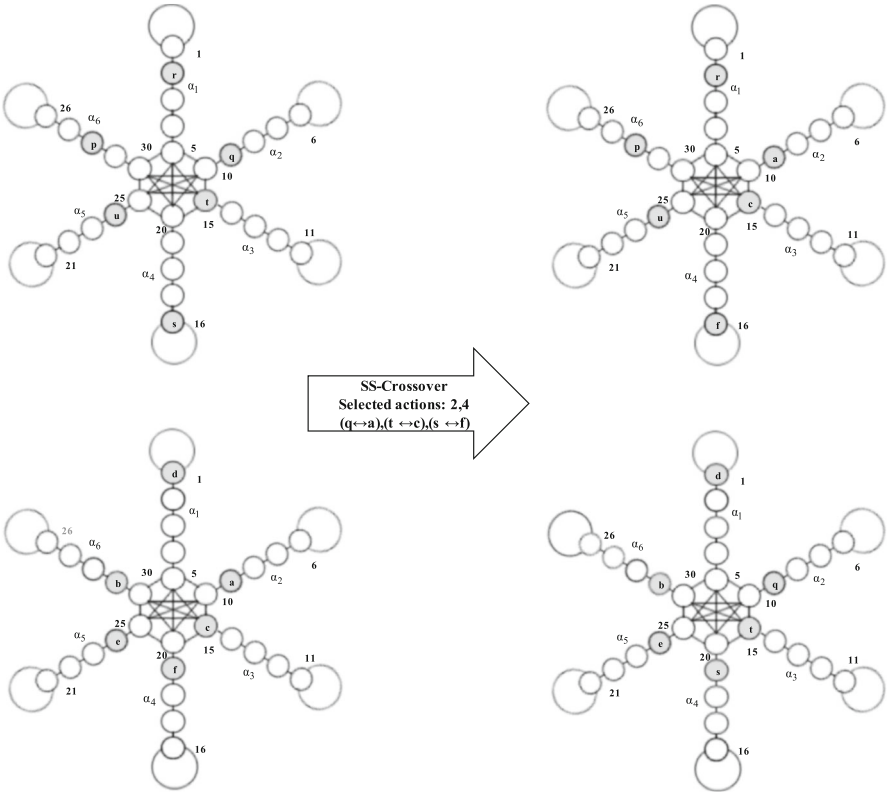


Fig. 22 An example of an SS-Crossover

```
Procedure SS-Crossover ( CR1, CR2 )
    Generate two random numbers r1 and r2 in [1,n] where r1 < r2;
    For i = r1 to r2 do
        Swap(CR1.Object(CR1.Action(i)), CR2.Object(CR2.Action(i)));
    End For
End SS-Crossover
```

Fig. 23 Pseudo code for the SS-Crossover

assigned object of each action in the range of [2,4] of CR_1, with the assigned object of the same action in the range of [2,4] of CR_2. Note that in the SS-Crossover the states of the actions remain unchanged. Pseudo code for the SS-Crossover is shown in Fig. 23.

4.3.2 XS-Crossover

The XS-Crossover is the same as the crossover used in GALA. Figure 24 shows an example of an XS-Crossover. Assuming actions 2 and 4 are selected randomly from the parent chromosomes, the XS-Crossover exchanges both the assigned object and the state of each action in the range of [2,4] of CR_1, with the assigned object and the state of the same action in the range of [2,4] of CR_2. Figure 25 shows the pseudo code for the XS-Crossover.

4.3.3 LS-Crossover

The LS-Crossover is the same as the crossover used in GALA, except that the state of each action is changed to become its corresponding boundary state. Figure 26 shows an example of an LS-Crossover. Assume that actions 2 and 4 are randomly selected in the parent chromosomes, then the LS-Crossover causes: (1) Each object assigned to the actions in the range of [2,4] of CR_1 is exchanged with the assigned object of the same action in the range of [2,4] of CR_2; and (2) The state of each action changes to become its boundary state. Figure 27 shows the pseudo code of the LS-Crossover.

5 The equipartitioning problem

Let $\underline{A} = \{A_1, \ldots, A_W\}$ be a set of W objects. We want to partition $\underline{A}$ into R classes $\{P_1,\ldots,P_R\}$ such that the objects used more frequently are located together in the same class. We assume that the joint access probabilities of the objects are unknown. This problem is called the object partitioning problem. A special case of the object partitioning problem, referred to as the equal partitioning problem (EPP), is where the objects are equipartitioned. In an EPP each class has exactly $M = W/R$ objects. For solving the EPP with MGALA we define a chromosome to have

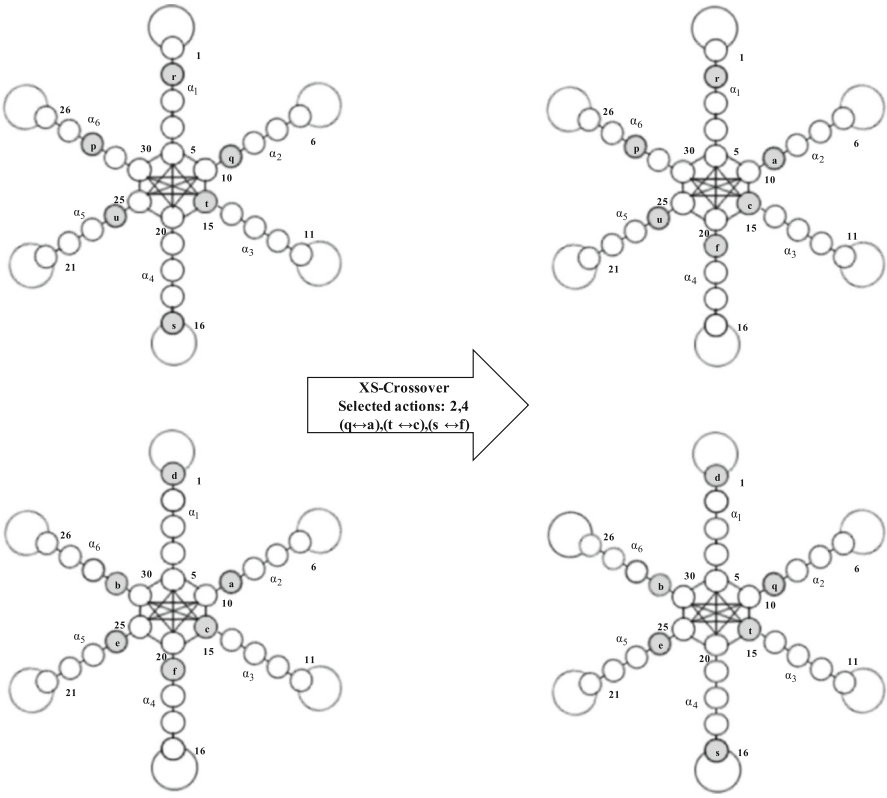


Fig. 24 An example of an XS-Crossover

Procedure XS-Crossover (CR_1, CR_2)
Generate two random numbers r_1 and r_2 in *[1,n]* where $r_1 < r_2$;
For i = r_1 to r_2 **do**
Swap($CR_1.Object(CR_1.Action(i))$, $CR_2.Object(CR_2.Action(i))$);
Swap($CR_1.State(CR_1.Action(i))$, $CR_2.State(CR_2.Action(i))$);
End For
End XS-Crossover

Fig. 25 Pseudo code for the XS-Crossover

W genes (actions) and the value of the genes are selected from a set of classes $\{P_1,\ldots,P_R\}$ (as migratory objects in the OMA) such that each class is assigned to W/R of genes (actions). Objects are initially assigned to the boundary state of actions. Figure 28 shows a chromosome based on a Tsetline OMA representation for 6 objects and 2 classes (called class α and class β) with $N = 5$. In this figure objects 1, 3, and 4 are assigned to class α, and objects 2, 5, and 6 are assigned to class β.

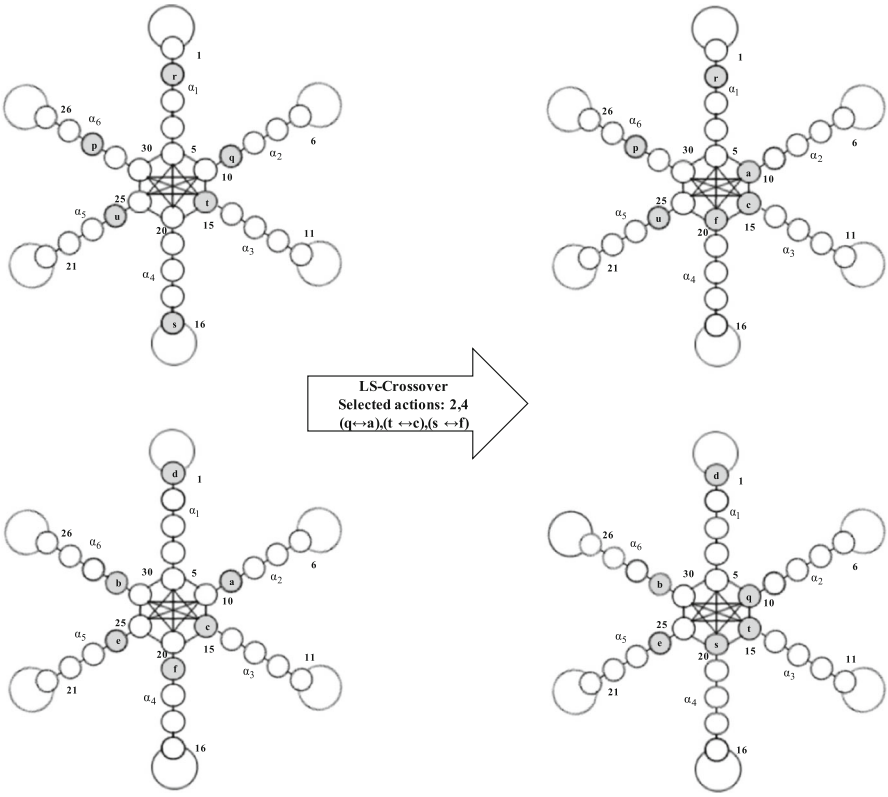


Fig. 26 An example of an LS-Crossover

```
Procedure LS-Crossover ( CR1, CR2 )
    Generate two random numbers r1 and r2 in [1,n] where r1 < r2;
    For i = r1 to r2 do
        Swap(CR1.Object(CR1.Action(i)), CR2.Object(CR2.Action(i)));
        CR.State(CR.Action(i)) = CR.Action(i)*N; //the state of action changed to the boundary state
    End For
End LS-Crossover
```

Fig. 27 Pseudo code for the LS-Crossover

5.1 Local search for EPP

Suppose a query [which is a pair of objects (A_i, A_j)] has been accessed. If the assigned objects of actions α_i and α_j are the same, then both actions α_i and α_j are rewarded and their states change according to OMA connections. If the assigned objects of actions α_i and α_j are different, then they are penalized and their states change according to OMA connections. Pseudo code for our local search of the EPP is shown in Fig. 29.

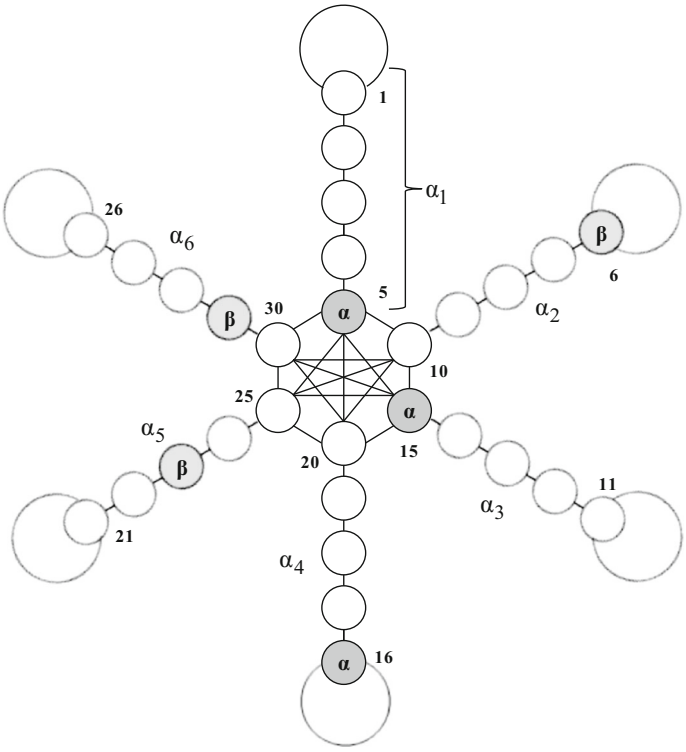


Fig. 28 A chromosome representation of the EPP with $W = 6$, $R = 2$, and $N = 5$

5.2 Experimental results

We studied the efficiency of our MGALA algorithm in solving the EPP by comparing its results to those obtained for an OMA method reported in [10] and to the GALA algorithm. Queries were chosen randomly from a pool of queries for all experiments. The pool of queries was generated in such a way that the sum of probabilities that object A_i in partition π_i is jointly accessed with other objects in partition π_i is p, and with objects in partition π_j ($j \neq i$) is $1 - p$, that is:

$$\sum_{A_j \in \pi_i} \Pr\left[A_i, A_j \, accessed \, together\right] = p \tag{1}$$

Therefore, if $p = 1$, then queries will only involve objects in the same partition. As the value of p decreases, the queries will become decreasingly informative about the solution of the EPP [10]. For all experiments an initial population of chromosomes of size 1 was randomly created, the size of each chromosome was set equal to the number of objects, the mutation rate was 0.05, the selection mechanism was (1,1), p was 0.9, and the depth of memory was 2. The algorithm terminates when all the objects in only chromosome are located in the most internal state of

```
Procedure LocalSearch( CR )
    Access query (A_i, A_j) from the pool of queries;
    If ( CR.Object(CR.Action(i)) = CR.Object(CR.Action(j)) )
        Reward(CR, CR.Action(i));
        Reward(CR, CR.Action(j));
    Else
        Penalty(CR, CR.Action(i));
        Penalty(CR, CR.Action(j));
    End if
End LocalSearch;
```

Fig. 29 Pseudo code for our local search of the EPP

their actions. For all experiments a Tsetline-based OMA was used for chromosome representation. Each reported result was averaged over 30 runs. We performed a parametric test (T test) and two non-parametric tests (wilcoxon rank sum test and permutation test) at the 95 % significance level to provide statistical confidence. The T tests were performed after ensuring that the data followed a normal distribution (by using the Kolmogorov–Smirnov test).

5.2.1 Experiment 1

In this experiment we compared the results obtained from MGALA with the results of two other algorithms, an OMA-based algorithm reported in [10] and the GALA version of the algorithm, for the EPP, in terms of the number of iterations (number of accessed queries) required by the algorithm. MGALA was tested with three different mutation operators: SS-Mutation, XS-Mutation, and LS-Mutation.

Table 1 presents the results of the different algorithms for 14 different cases with respect to the average number of iterations and their standard deviation. From the results reported in Table 1 we report the following:

- The MGALA algorithm outperforms both the OMA and GALA algorithms.
- For cases ($W = 9$ and $R = 3$), ($W = 12$ and $R = 2$) and ($W = 8$ and $R = 2$), MGALA using the LS-Mutation performs the best, and for the other cases MGALA with the SS-Mutation displays the best performance.
- The OMA algorithm displays the worst performance compared with the GALA and MGALA algorithms.
- As the number of classes (R) decreases, the number of iterations required by all algorithms increase. This is because a low value for R means a higher number of objects will be placed in each class, leading to a situation where more actions have the same class number (migratory objects). This causes the probability that a mutation operator swaps two objects between two actions to decrease.
- MGALA with the XS-Mutation displays the same performance as GALA. This is because: (1) In the MGALA and GALA algorithms, the selection mechanism is considered to be (1,1), that is, MGALA, like GALA, has no selection

Table 1 Comparison of the number of iterations required by different algorithms

W	R	OMA		GALA		MGALA					
						SS-Mutation		XS-Mutation		LS-Mutation	
		Avg.	Std.	Avg.	Std.	Avg.	Std.	Avg.	Std.	Avg.	Std.
4	2	18	8.50E+00	10	1.50E+00	10	2.10E+00	10	1.50E+00	12	2.00E+00
6	3	33	1.63E+01	18	2.20E+00	18	1.80E+00	18	2.20E+00	21	3.50E+00
	2	115	5.63E+01	98	6.30E+00	85	6.50E+00	98	6.30E+00	102	9.80E+00
9	3	209	9.86E+01	194	2.16E+01	186	1.26E+01	194	2.16E+01	180	1.72E+01
12	6	70	2.56E+01	36	6.50E+00	34	6.70E+00	36	6.50E+00	41	1.29E+01
	4	243	5.86E+01	181	1.23E+01	179	9.50E+00	181	1.23E+01	190	2.18E+01
	3	569	1.26E+02	500	2.53E+01	486	2.56E+01	500	2.53E+01	512	3.13E+01
	2	1,993	3.65E+02	1,884	1.52E+01	1,869	8.56E+01	1,884	1.52E+01	1,823	9.84E+01
15	5	317	9.53E+01	198	1.95E+01	188	1.53E+01	198	1.95E+01	212	6.84E+01
	3	2,000	1.56E+02	1,902	1.25E+02	1,892	8.86E+01	1,902	1.25E+02	1,911	1.09E+02
18	9	111	3.43E+01	62	6.20E+00	58	9.50E+00	62	6.20E+00	62	1.67E+01
	6	334	1.25E+02	211	1.52E+01	201	1.25E+01	211	1.52E+01	227	5.23E+01
	3	5,169	5.69E+02	4,443	4.52E+01	3,896	9.64E+01	4,443	4.52E+01	4,563	1.73E+02
	2	10,096	2.36E+02	9,898	6.85E+01	9,865	1.22E+02	9,898	6.85E+01	9,755	1.32E+02

mechanism in this mode; and (2) the XS-Mutation operator used by MGALA is the same as the mutation operator used by GALA.

Table 2 shows the p values of the two-tailed T test, the p values of the two-tailed wilcoxon rank sum test and the p values of the two-tailed permutation test. From the results reported in Table 2 we report the following:

- For all three kinds of statistical tests (wilcoxon, permutation and T test), the difference between the performance of the OMA and the performance of the other algorithms is statistically significant (p value <0.05) in most cases.

5.2.2 *Experiment 2*

This experiment's goal was to evaluate the accuracy of the solution produced by MGALA. Before we introduce the concept of accuracy for MGALA, we will provide some preliminaries. For this purpose we use an example of equipartitioning with the 4 objects A_1, A_2, A_3, and A_4, and two classes, α and β. A Tsetline based on an OMA with a depth of memory 2 will be used for the chromosome representation shown in Fig. 30. We also assume that the initial population is of size one, and that the only chromosome in the initial population is created randomly, and has its migratory objects in the boundary state of the actions.

For this example there are the three possible object equipartitioning schemes specified below:

- $\alpha\alpha\beta\beta$: Objects A_1, A_2 are in class α and objects A_3, A_4 are in class β.
- $\alpha\beta\alpha\beta$: Objects A_1, A_3 are in class α and objects A_2, A_4 are in class β.
- $\alpha\beta\beta\alpha$: Objects A_1, A_4 are in class α and objects A_2, A_3 are in class β.

The chromosome in Fig. 30 can be also represented by $(\bar{\alpha}\underline{\alpha}\underline{\beta}\bar{\beta})$ which corresponds to the following situation:

- Object A_1 is in partition α and the migratory object is located in the boundary state of action 1.
- Object A_2 is in partition α and the migratory object is located in the internal state of action 2.
- Object A_3 is in partition β and the migratory object is located in the internal state of action 3.
- Object A_4 is in partition β and the migratory object is located in the boundary state of action 4.

Each possible equipartitioning scheme may correspond to any of the 16 possible chromosomes out of all 48 possible chromosomes. For example chromosomes $(\bar{\alpha}\bar{\alpha}\bar{\beta}\bar{\beta})$, $(\bar{\alpha}\bar{\alpha}\underline{\beta}\underline{\beta})$, $(\bar{\alpha}\bar{\alpha}\underline{\beta}\bar{\beta})$, $(\bar{\alpha}\bar{\alpha}\underline{\beta}\underline{\beta})$, $(\bar{\alpha}\underline{\beta}\bar{\beta}\bar{\beta})$, $(\bar{\alpha}\underline{\alpha}\bar{\beta}\underline{\beta})$, $(\bar{\alpha}\underline{\alpha}\underline{\beta}\bar{\beta})$, $(\bar{\alpha}\underline{\alpha}\underline{\beta}\underline{\beta})$, $(\underline{\alpha}\bar{\alpha}\bar{\beta}\bar{\beta})$, $(\underline{\alpha}\bar{\alpha}\bar{\beta}\underline{\beta})$, $(\underline{\alpha}\bar{\alpha}\underline{\beta}\bar{\beta})$,$(\underline{\alpha}\bar{\alpha}\underline{\beta}\underline{\beta})$, $(\underline{\alpha}\underline{\alpha}\bar{\beta}\bar{\beta})$, $(\underline{\alpha}\underline{\alpha}\bar{\beta}\underline{\beta})$, $(\underline{\alpha}\underline{\alpha}\underline{\beta}\bar{\beta})$, and $(\underline{\alpha}\underline{\alpha}\underline{\beta}\underline{\beta})$ show the equipartitioning $\alpha\alpha\beta\beta$.

Table 2 The results of statistical tests for OMA algorithm and other algorithms

W	R	GALA			MGALA								
					SS-Mutation			XS-Mutation			LS-Mutation		
		p value			*p* value			*p* value			*p* value		
		T test	Permutation	Wilcoxon	*T* test	Permutation	Wilcoxon	*T* test	Permutation	Wilcoxon	*T* test	Permutation	Wilcoxon
4	2	2.05E−05	0.00E+00	1.49E−04	2.50E−05	0.00E+00	1.78E−04	2.05E−05	0.00E+00	1.17E−04	7.57E−04	0.00E+00	1.77E−03
6	3	2.57E−05	0.00E+00	6.37E−04	2.47E−05	0.00E+00	6.37E−04	2.57E−05	0.00E+00	6.04E−04	4.68E−04	0.00E+00	3.76E−03
	2	1.11E−01	1.10E−01	4.25E−01	7.06E−03	5.00E−03	1.03E−02	1.11E−01	1.18E−01	4.51E−01	2.23E−01	2.21E−01	5.89E−01
9	3	4.22E−01	4.20E−01	5.74E−01	2.15E−01	1.98E−01	3.33E−01	4.22E−01	4.35E−01	5.84E−01	1.23E−01	1.06E−01	2.04E−01
12	6	9.36E−08	0.00E+00	8.35E−08	3.27E−08	0.00E+00	3.35E−08	9.36E−08	0.00E+00	1.31E−07	5.65E−06	0.00E+00	3.83E−06
	4	3.94E−06	0.00E+00	1.17E−05	2.07E−06	0.00E+00	5.46E−06	3.94E−06	0.00E+00	7.74E−06	6.84E−05	0.00E+00	4.49E−05
	3	6.23E−03	1.00E−02	6.67E−03	1.35E−03	0.00E+00	7.30E−04	6.23E−03	6.00E−03	7.79E−03	2.24E−02	1.30E−02	3.64E−02
	2	1.13E−01	9.80E−02	8.11E−02	8.06E−02	7.30E−02	1.01E−01	1.13E−01	9.40E−02	8.37E−02	2.00E−02	2.00E−02	3.58E−02
15	5	2.38E−07	0.00E+00	1.20E−07	4.60E−08	0.00E+00	8.70E−08	2.38E−07	0.00E+00	1.31E−07	3.32E−05	0.00E+00	3.06E−05
	3	1.19E−02	1.00E−03	2.71E−02	2.62E−03	0.00E+00	1.72E−03	1.19E−02	1.90E−02	2.76E−02	1.61E−02	1.50E−02	1.70E−02
18	9	1.72E−08	0.00E+00	3.66E−07	5.40E−09	0.00E+00	1.25E−07	1.72E−08	0.00E+00	4.44E−07	9.75E−08	0.00E+00	3.13E−07
	6	9.93E−06	0.00E+00	1.05E−05	2.88E−06	0.00E+00	2.08E−06	9.93E−06	0.00E+00	8.29E−06	1.68E−04	0.00E+00	9.79E−05
	3	1.15E−07	0.00E+00	8.12E−09	7.55E−13	0.00E+00	1.77E−09	1.15E−07	0.00E+00	8.12E−09	5.01E−06	0.00E+00	2.47E−07
	2	1.29E−04	0.00E+00	1.37E−04	4.86E−05	0.00E+00	3.71E−05	1.29E−04	0.00E+00	1.68E−04	1.38E−07	0.00E+00	2.11E−07

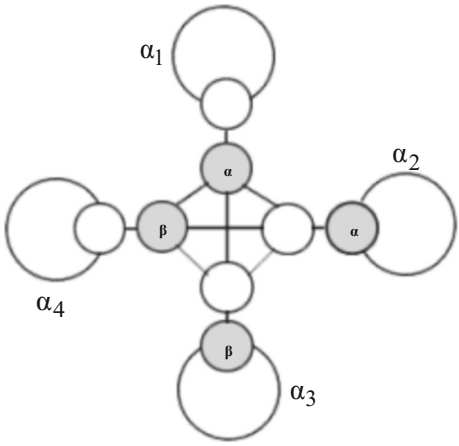


Fig. 30 A representation of the EPP with $W = 4$, $R = 2$, and $N = 2$

We group chromosomes corresponding to a given equipartitioning into two different sets: a converged chromosomes set (CCS), and a non-converged chromosomes set (NCCS). The CCS includes all the chromosomes for which all objects of class α or class β are in the internal states. All other chromosomes are in the NCCS. For example, chromosomes $(\bar{\alpha}\bar{\alpha}\bar{\beta}\bar{\beta})$, $(\bar{\alpha}\bar{\alpha}\bar{\beta}\underline{\beta})$, $(\bar{\alpha}\bar{\alpha}\underline{\beta}\bar{\beta})$, $(\bar{\alpha}\underline{\alpha}\bar{\beta}\bar{\beta})$, $(\bar{\alpha}\underline{\alpha}\underline{\beta}\bar{\beta})$, $(\underline{\alpha}\bar{\alpha}\bar{\beta}\bar{\beta})$, $(\underline{\alpha}\bar{\alpha}\bar{\beta}\underline{\beta})$, and $(\underline{\alpha}\bar{\alpha}\underline{\beta}\bar{\beta})$, of equipartitioning $\alpha\alpha\beta\beta$, are in the NCCS of equipartitioning $\alpha\alpha\beta\beta$, and $(\bar{\alpha}\bar{\alpha}\underline{\beta}\underline{\beta})$, $(\bar{\alpha}\underline{\alpha}\underline{\beta}\underline{\beta})$, $(\underline{\alpha}\bar{\alpha}\underline{\beta}\underline{\beta})$, $(\underline{\alpha}\underline{\alpha}\bar{\beta}\bar{\beta})$, $(\underline{\alpha}\underline{\alpha}\bar{\beta}\underline{\beta})$, $(\underline{\alpha}\underline{\alpha}\underline{\beta}\bar{\beta})$, and $(\underline{\alpha}\underline{\alpha}\underline{\beta}\underline{\beta})$ are in the CCS of equipartitioning $\alpha\alpha\beta\beta$. MGALA converges to one of the following six sets: either the NCCS of equipartitioning $\alpha\alpha\beta\beta$, or the CCS of equipartitioning $\alpha\alpha\beta\beta$, or the NCCS of equipartitioning $\alpha\beta\alpha\beta$, or the CCS of equipartitioning $\alpha\beta\alpha\beta$, or the NCCS of equipartitioning $\alpha\beta\beta\alpha$, or the CCS of equipartitioning $\alpha\beta\beta\alpha$. Let *p1*, *p2*, *p3*, *p4*, *p5*, and *p6* be the probabilities of the current chromosome to be in one of the above-mentioned sets, respectively. Furthermore, assume that eighty percent of the queries accessed from the pool of queries are (A_1, A_2) or (A_3, A_4). For MGALA to produce the correct solution it must generate a population containing one of following chromosomes: $(\bar{\alpha}\bar{\alpha}\bar{\beta}\bar{\beta})$, $(\bar{\alpha}\bar{\alpha}\bar{\beta}\underline{\beta})$, $(\bar{\alpha}\bar{\alpha}\underline{\beta}\bar{\beta})$, $(\bar{\alpha}\bar{\alpha}\underline{\beta}\underline{\beta})$, $(\bar{\alpha}\underline{\alpha}\bar{\beta}\bar{\beta})$, $(\bar{\alpha}\underline{\alpha}\bar{\beta}\underline{\beta})$, $(\bar{\alpha}\underline{\alpha}\underline{\beta}\bar{\beta})$, $(\bar{\alpha}\underline{\alpha}\underline{\beta}\underline{\beta})$, $(\underline{\alpha}\bar{\alpha}\bar{\beta}\bar{\beta})$, $(\underline{\alpha}\bar{\alpha}\bar{\beta}\underline{\beta})$, $(\underline{\alpha}\bar{\alpha}\underline{\beta}\bar{\beta})$,$(\underline{\alpha}\bar{\alpha}\underline{\beta}\underline{\beta})$, $(\underline{\alpha}\underline{\alpha}\bar{\beta}\bar{\beta})$, $(\underline{\alpha}\underline{\alpha}\bar{\beta}\underline{\beta})$, $(\underline{\alpha}\underline{\alpha}\underline{\beta}\bar{\beta})$, and $(\underline{\alpha}\underline{\alpha}\underline{\beta}\underline{\beta})$. So, MGALA must converge to one of the sets of NCCS of equipartitioning, or to one of the sets of CCS of equipartitioning $\alpha\alpha\beta\beta$. The algorithm is more accurate if the probability of converging to a set of NCCS of equipartitioning, or to a set of CCS of equipartitioning $\alpha\alpha\beta\beta$ (that is $p_1 + p_2$) are higher than 0.8.

Next we undertook the experimentation phase of the study. Input queries were generated in such a way that eighty percent of the queries accessed from the pool of queries were (A_1, A_2) or (A_3, A_4). Note that the initial population is considered to be of size one, that the only chromosome in the initial population is created randomly, and that it has its migratory objects at the boundary state of the actions. That is chromosome $(\bar{\alpha}\bar{\alpha}\bar{\beta}\bar{\beta})$, which belongs to the NCCS of equipartitioning $\alpha\alpha\beta\beta$, or $(\bar{\alpha}\bar{\beta}\bar{\alpha}\bar{\beta})$, which belongs to the NCCS of equipartitioning $\alpha\beta\alpha\beta$, or $(\bar{\alpha}\bar{\beta}\bar{\beta}\bar{\alpha})$, which belongs to the NCCS of equipartitioning $\alpha\beta\beta\alpha$ are the only options. Therefore,

initial values for p_1, p_3, and p_5 are considered to be zero, and initial values for p_2, p_4, and p_6 are considered to be 1/3. Figure 31a–c show the evolution of p_1, p_2, p_3, p_4, p_5, and p_6 for the three different mutation operators SS-Mutation, XS-Mutation, and LS-Mutation.

These figures show that $p_1 + p_2$ approaches a value close to 0.9 for all mutation operators. That is, when eighty percent of the queries accessed from the pool of queries are A_1, A_2 or A_3, A_4, MGALA converges to the correct equipartitioning ($\alpha\alpha\beta\beta$) with a probability close to 0.9. The mutation rate was set to 0.05 for this experiment.

Table 3 presents the MGALA algorithm results for different mutation operators and different query percentages (accessed from the pool of queries) being (A_1, A_2) or (A_3, A_4), with respect to the accuracy of the solution and its standard deviation. The percentage varies from 40 to 100 by increments of 10 in Table 3. From these results we conclude the following:

- For all mutation operators the accuracy of the solution generated by MGALA increases as the percentage of queries (A_1, A_2) or (A_3, A_4) in the input increases.
- MGALA has the highest accuracy when the LS-Mutation operator is used.

Table 4 shows the results of statistical tests. From the results reported in Table 4 we report the following:

- For all three kinds of statistical tests (wilcoxon, permutation and T test), the difference between the performance of the MGALA algorithm when it uses the LS-Mutation operator and the performance of the MGALA algorithm when it uses other mutation operators is not statistically significant (p value >0.05).

5.2.3 Experiment 3

The goal of this experiment was to study the impact of the parameter N (depth of memory) on the number of iterations required by the MGALA algorithm to find optimal equipartitioning. The depth of memory was varied from 2 to 10 by increments of 2. MGALA was tested with three different mutation operators. Tables 5, 6 and 7 show the results obtained for the MGALA algorithm with the SS-Mutation, XS-Mutation, and LS-Mutation for 14 different cases, with respect to the average number of iterations required and their standard deviation.

- For lower values of R MGALA requires a higher number of iterations to converge to optimal equipartitioning.
- For higher values of W/R MGALA requires a higher number of iterations to converge to optimal equipartitioning.
- For higher values of W/R MGALA requires a higher depth of memory to converge to optimal equipartitioning.
- The MGALA algorithm with a depth of memory $N = 2$ performs better than MGALA with a depth of memory of $N \neq 2$.

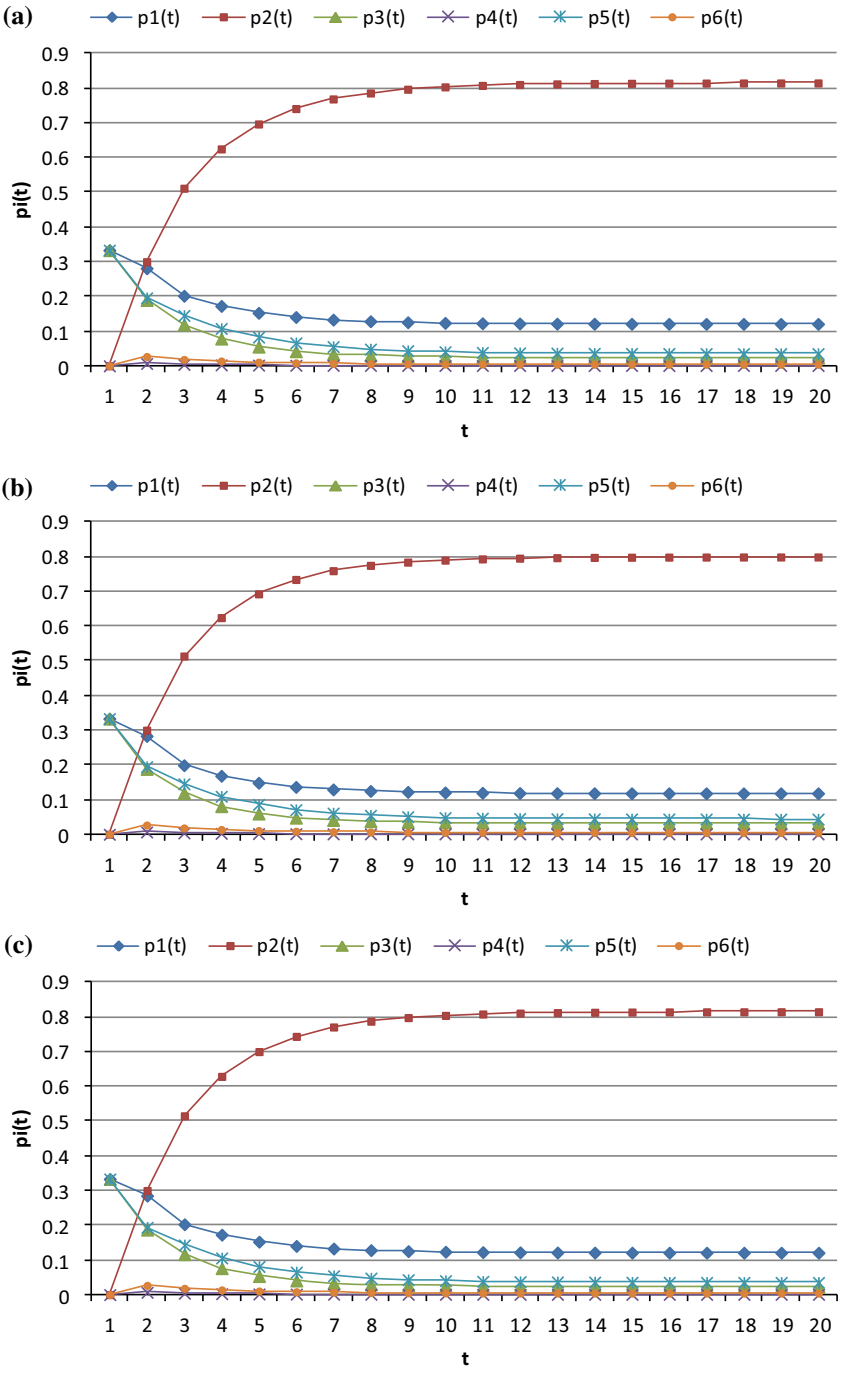


Fig. 31 The evolution of p_1, p_2, p_3, p_4, p_5, and p_6 in MGALA with $W = 4$, $R = 2$, and $N = 2$, for the SS-Mutation (a), XS-Mutation (b), and LS-Mutation (c) operator for the EPP

Table 3 Accuracy of MGALA with respect to percentage of queries (A_1, A_2) or (A_3, A_4) and mutation operators

The percentage of queries (A_1, A_2) or (A_3, A_4)	SS-Mutation		XS-Mutation		LS-Mutation	
	Avg.	Std.	Avg.	Std.	Avg.	Std.
40	0.41	5.10E−02	0.42	4.10E−02	0.42	4.40E−02
50	0.54	7.20E−02	0.56	4.20E−02	0.56	3.30E−02
60	0.69	7.70E−02	0.7	5.50E−02	0.69	6.50E−02
70	0.8	7.80E−02	0.81	6.70E−02	0.8	6.90E−02
80	0.9	6.50E−02	0.91	6.30E−02	0.9	7.80E−02
90	0.92	6.60E−02	0.94	7.20E−02	0.92	8.50E−02
100	0.95	7.20E−02	0.96	6.50E−02	0.94	5.60E−02

Table 4 The results of statistical tests for MGALA algorithm with LS-Mutation operator vs. MGALA algorithm with SS-Mutation and XS-Mutation operators with respect to percentage of queries (A_1, A_2) or (A_3, A_4)

The percentage of queries (A_1, A_2) or (A_3, A_4)	SS-Mutation			XS-Mutation		
	P value			P value		
	T test	Permutation	Wilcoxon	*T* test	Permutation	Wilcoxon
40	4.09E−01	5.19E−01	4.46E−01	1.00E+00	7.35E−01	8.65E−01
50	1.99E−01	3.44E−01	6.90E−01	1.00E+00	6.23E−01	5.69E−01
60	5.67E−01	1.31E−01	1.91E−01	5.25E−01	7.50E−02	9.05E−02
70	5.98E−01	7.15E−01	8.13E−01	5.73E−01	1.88E−01	3.67E−01
80	5.50E−01	2.99E−01	3.75E−01	5.89E−01	4.60E−02	6.79E−02
90	2.71E−01	3.49E−01	2.01E−01	3.34E−01	2.69E−01	3.55E−01
100	5.77E−01	7.28E−01	4.87E−01	2.12E−01	3.29E−01	2.52E−01

These conclusions are because higher values of *W/R* mean a higher number of objects are placed in each class. This leads to a situation where more actions have the same class number (migratory objects), and also decreases the probability that a mutation operator swaps two objects between two actions.

Tables 8, 9 and 10 show the p values of the two-tailed T test, the p values of the two-tailed wilcoxon rank sum test and the p values of the two-tailed permutation test for MGALA algorithm with SS-Mutation, XS-Mutation and LS-Mutation respectively. From the results reported in these tables we report the following:

- For all three kinds of statistical tests (wilcoxon, permutation and T test), the difference between the performance of the MGALA algorithm with depth of memory $N = 2$ and the performance of the MGALA algorithm with a depth of memory of $N \neq 2$ is statistically significant (p value <0.05) in most cases for all three types of operators.

Table 5 Number of iterations required by MGALA with SS-Mutation operator for different cases and different depths of memory

Algorithm	N		W = 4	W = 6		W = 9	W = 12			
			R = 2	R = 3	R = 2	R = 3	R = 6	R = 4	R = 3	R = 2
MGALA	2	Avg.	10	18	85	186	34	179	486	1,869
		Std.	2.10E+00	1.80E+00	6.50E+00	1.26E+01	6.70E+00	9.50E+00	2.56E+01	8.56E+01
	4	Avg.	14	20	90	178	45	175	512	1,835
		Std.	2.30E+00	1.80E+00	7.10E+00	8.20E+00	3.50E+00	1.84E+01	5.62E+01	3.82E+01
	6	Avg.	18	29	98	205	58	189	423	1,663
		Std.	3.50E+00	6.80E+00	1.53E+01	2.42E+01	1.53E+01	1.53E+01	8.83E+01	1.28E+02
	8	Avg.	28	44	102	196	79	190	486	1,721
		Std.	1.20E+00	5.60E+00	1.27E+01	1.42E+01	8.40E+00	2.42E+01	4.43E+01	5.78E+01
	10	Avg.	59	49	125	224	96	215	459	1,858
		Std.	6.50E+00	7.30E+00	1.42E+01	2.18E+01	9.10E+00	2.51E+01	1.84E+01	5.64E+01

Algorithm	N		W = 15		W = 18			
			R = 5	R = 3	R = 9	R = 6	R = 3	R = 2
MGALA	2	Avg.	188	1,892	58	201	3,896	9,865
		Std.	1.53E+01	8.86E+01	9.50E+00	1.25E+01	9.64E+01	1.22E+02
	4	Avg.	365	1,770	86	216	3,684	9,635
		Std.	4.43E+01	8.52E+01	1.53E+01	2.43E+01	1.02E+02	1.56E+02
	6	Avg.	458	1,756	97	189	3,356	8,569
		Std.	1.42E+02	2.35E+02	1.57E+01	1.08E+01	1.16E+02	1.86E+02
	8	Avg.	380	1,656	102	225	3,015	4,563
		Std.	6.54E+01	1.41E+02	1.41E+01	2.81E+01	3.25E+02	1.85E+02
	10	Avg.	412	1,669	136	304	3,388	5,698
		Std.	2.84E+01	8.64E+01	1.28E+01	3.64E+01	2.63E+02	2.39E+02

Table 6 Number of iterations required by MGALA with XS-Mutation operator for different cases and different depths of memory

Algorithm	N		W = 4	W = 6		W = 9	W = 12			
			R = 2	R = 3	R = 2	R = 3	R = 6	R = 4	R = 3	R = 2
MGALA	2	Avg.	10	18	98	194	36	181	500	1,884
		Std.	1.50E+00	2.20E+00	6.30E+00	2.16E+01	6.50E+00	1.23E+01	2.53E+01	1.52E+01
	4	Avg.	12	23	112	215	55	198	523	1,698
		Std.	1.80E+00	3.50E+00	1.29E+01	2.54E+01	7.80E+00	1.95E+01	3.78E+01	1.25E+02
	6	Avg.	14	32	108	223	53	205	534	1,365
		Std.	2.50E+00	8.60E+00	1.29E+01	2.79E+01	9.50E+00	2.37E+01	5.83E+01	1.25E+02
	8	Avg.	21	39	123	245	69	218	496	1,236
		Std.	3.20E+00	8.60E+00	1.96E+01	3.26E+01	1.56E+01	3.53E+01	4.11E+01	6.53E+01
	10	Avg.	48	51	145	219	85	226	509	1,456
		Std.	6.50E+00	8.20E+00	1.26E+01	3.42E+01	1.51E+01	4.23E+01	5.64E+01	1.13E+02

Algorithm	N		W = 15		W = 18			
			R = 5	R = 3	R = 9	R = 6	R = 3	R = 2
MGALA	2	Avg.	198	1,902	62	211	2,443	9,898
		Std.	1.95E+01	1.25E+02	6.20E+00	1.52E+01	4.52E+01	6.85E+01
	4	Avg.	255	2,015	89	226	3,698	10,125
		Std.	2.43E+01	2.02E+02	2.55E+01	3.89E+01	1.25E+02	2.70E+02
	6	Avg.	273	2,115	87	199	4,569	9,632
		Std.	6.94E+01	1.46E+02	1.62E+01	3.21E+01	2.57E+02	3.27E+02
	8	Avg.	289	1,896	112	269	4,896	9,986
		Std.	7.51E+01	1.05E+02	1.96E+01	3.54E+01	2.40E+02	3.56E+02
	10	Avg.	256	2,365	163	278	2,369	6,896
		Std.	4.75E+01	4.23E+02	2.94E+01	3.59E+01	2.05E+02	4.02E+02

Table 7 Number of iterations required by MGALA with LS-Mutation operator for different cases and different depths of memory

Algorithm	N		W = 4	W = 6		W = 9	W = 12			
			R = 2	R = 3	R = 2	R = 3	R = 6	R = 4	R = 3	R = 2
MGALA	2	Avg.	12	21	102	180	41	190	512	1,823
		Std.	2.00E+00	3.50E+00	9.80E+00	1.72E+01	1.29E+01	2.18E+01	3.13E+01	9.84E+01
	4	Avg.	16	22	88	195	89	165	446	1,823
		Std.	3.20E+00	3.10E+00	1.27E+01	3.11E+01	1.22E+01	1.83E+01	3.94E+01	1.25E+02
	6	Avg.	18	35	85	176	125	154	485	1,912
		Std.	3.50E+00	6.50E+00	1.23E+01	1.32E+01	1.94E+01	2.56E+01	3.22E+01	1.28E+02
	8	Avg.	32	48	95	201	147	168	456	1,996
		Std.	2.90E+00	3.50E+00	1.05E+01	2.54E+01	1.85E+01	2.32E+01	5.24E+01	1.55E+02
	10	Avg.	64	68	114	223	212	186	412	1,785
		Std.	8.60E+00	5.40E+00	1.43E+01	3.26E+01	1.94E+01	1.43E+01	3.54E+01	1.21E+02

Algorithm	N		W = 15		W = 18			
			R = 5	R = 3	R = 9	R = 6	R = 3	R = 2
MGALA	2	Avg.	212	1,911	62	227	4,563	9,755
		Std.	6.84E+01	1.09E+02	1.67E+01	5.23E+01	1.73E+02	1.32E+02
	4	Avg.	238	1,698	78	156	3,568	8,856
		Std.	4.52E+01	1.25E+02	1.35E+01	2.15E+01	3.25E+02	4.53E+02
	6	Avg.	256	1,554	125	263	3,998	7,805
		Std.	5.52E+01	1.24E+02	1.35E+01	3.21E+01	4.01E+02	5.23E+02
	8	Avg.	325	1,396	179	325	3,368	5,639
		Std.	3.62E+01	1.89E+02	2.53E+01	4.52E+01	3.60E+02	4.68E+02
	10	Avg.	378	1,298	245	369	2,856	3,568
		Std.	4.39E+01	1.39E+02	2.64E+01	4.89E+01	2.35E+02	4.13E+02

Table 8 The results of statistical tests for MGALA with SS-Mutation and depth of memory 2 vs. MGALA algorithm with SS-Mutation and other depths of memory

Algorithm	N	*p* value	W = 4	W = 6		W = 9	W = 12			
			R = 2	R = 3	R = 2	R = 3	R = 6	R = 4	R = 3	R = 2
MGALA	4	*T* test	9.77E−08	1.75E−04	8.06E−03	6.80E−03	8.63E−09	1.90E−20	2.85E−02	5.65E−02
		Permutation	0.00E+00	0.00E+00	5.00E−03	0.00E+00	0.00E+00	0.00E+00	4.20E−02	2.40E−02
		Wilcoxon	9.06E−08	1.09E−03	3.94E−03	3.76E−03	9.67E−09	1.81E−01	7.24E−02	1.44E−02
	6	*T* test	1.29E−11	1.96E−09	1.84E−04	6.61E−04	1.11E−08	4.96E−03	7.78E−04	4.71E−08
		Permutation	0.00E+00	0.00E+00	1.00E−03	0.00E+00	4.00E−03	0.00E+00	0.00E+00	0.00E+00
		Wilcoxon	1.62E−09	4.37E−09	7.20E−05	3.11E−03	1.06E−08	1.91E−02	2.53E−04	3.08E−08
	8	*T* test	3.71E−27	8.74E−21	3.80E−07	7.31E−03	3.87E−20	2.77E−02	1.00E+00	1.18E−08
		Permutation	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	9.95E−01
		Wilcoxon	2.87E−11	2.87E−11	6.26E−08	1.12E−02	2.87E−11	1.68E−03	8.59E−01	2.10E−08
	10	*T* test	1.06E−26	5.96E−20	1.86E−14	4.11E−09	2.10E−23	4.29E−08	5.99E−05	5.61E−01
		Permutation	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	5.27E−01	0.00E+00
		Wilcoxon	2.87E−11	2.87E−11	8.15E−11	7.77E−09	2.87E−11	1.93E−08	5.97E−05	4.51E−01

Algorithm	N	*p* value	W = 15		W = 18			
			R = 5	R = 3	R = 9	R = 6	R = 3	R = 2
MGALA	4	*T* test	6.58E−19	7.55E−06	2.21E−09	5.41E−03	4.16E−09	6.07E−07
		Permutation	3.12E−01	0.00E+00	3.00E−03	4.00E−03	1.00E−03	0.00E+00
		Wilcoxon	2.87E−11	9.53E−07	3.06E−09	1.56E−02	9.67E−09	2.68E−07
	6	*T* test	3.12E−11	6.00E−03	1.89E−12	4.24E−04	2.65E−18	4.02E−24
		Permutation	8.00E−03	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
		Wilcoxon	2.87E−11	3.71E−02	5.49E−11	2.92E−04	2.87E−11	2.87E−11
	8	*T* test	1.11E−15	1.50E−08	1.43E−14	1.89E−04	1.28E−14	9.45E−42
		Permutation	2.70E−02	0.00E+00	1.00E−02	0.00E+00	0.00E+00	0.00E+00

Table 8 continued

Algorithm	N	*p* value	W = 15		W = 18			
			R = 5	R = 3	R = 9	R = 6	R = 3	R = 2
		Wilcoxon	3.02E−11	5.09E−08	5.23E−11	3.18E−04	2.49E−10	2.87E−11
	10	*T* test	2.67E−26	8.88E−11	5.17E−22	6.08E−15	7.92E−11	2.41E−36
		Permutation	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
		Wilcoxon	2.87E−11	5.32E−10	2.87E−11	3.18E−11	1.62E−09	2.87E−11

Table 9 The results of statistical tests for MGALA with XS-Mutation and depth of memory 2 vs. MGALA algorithm with XS-Mutation and other depths of memory

Algorithm	N	*p* value	W = 4	W = 6		W = 9	W = 12			
			R = 2	R = 3	R = 2	R = 3	R = 6	R = 4	R = 3	R = 2
MGALA	4	*T* test	6.25E−05	2.92E−07	9.82E−06	1.74E−03	3.76E−11	3.60E−04	9.69E−03	5.98E−09
		Permutation	0.00E+00	0.00E+00	3.80E−02	0.00E+00	1.00E−02	0.00E+00	0.00E+00	9.00E−03
		Wilcoxon	3.26E−03	1.19E−06	1.29E−05	2.21E−03	1.69E−10	1.95E−04	9.88E−03	1.49E−08
	6	*T* test	2.78E−08	1.64E−09	6.59E−04	1.01E−04	6.40E−09	3.14E−05	6.54E−03	5.89E−20
		Permutation	0.00E+00	1.00E−02	7.40E−02	0.00E+00	0.00E+00	0.00E+00	0.00E+00	4.00E−03
		Wilcoxon	5.53E−08	3.50E−09	2.89E−03	1.49E−04	1.37E−08	1.81E−05	9.47E−03	2.87E−11
	8	*T* test	1.18E−16	1.37E−13	2.72E−07	7.33E−08	1.41E−11	7.87E−06	6.53E−01	2.09E−30
		Permutation	2.00E−01	0.00E+00	0.00E+00	0.00E+00	8.20E−01	0.00E+00	0.00E+00	6.10E−01
		Wilcoxon	3.34E−11	2.87E−11	3.52E−07	2.20E−07	6.07E−11	3.97E−06	7.17E−01	2.87E−11
	10	*T* test	7.28E−24	3.00E−19	1.87E−17	2.06E−03	3.71E−16	4.86E−06	4.32E−01	7.39E−19
		Permutation	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	4.08E−01
		Wilcoxon	2.87E−11	2.87E−11	3.69E−11	1.72E−03	3.18E−11	5.60E−07	7.45E−01	2.87E−11

Algorithm	N	*p* value	W = 15		W = 18			
			R = 5	R = 3	R = 9	R = 6	R = 3	R = 2
MGALA	4	*T* test	6.30E−11	1.42E−02	4.35E−06	5.88E−02	3.77E−30	1.11E−04
		Permutation	0.00E+00	0.00E+00	2.00E−03	0.00E+00	0.00E+00	2.00E−03
		Wilcoxon	4.20E−10	2.66E−02	8.29E−06	2.46E−01	2.87E−11	5.41E−04
	6	*T* test	3.65E−06	1.28E−06	1.05E−08	7.44E−02	2.73E−28	1.46E−04
		Permutation	0.00E+00	0.00E+00	0.00E+00	1.00E−03	0.00E+00	0.00E+00
		Wilcoxon	3.70E−06	6.28E−07	7.77E−09	6.46E−02	1.48E−03	2.92E−04
	8	*T* test	5.01E−07	8.42E−01	6.87E−14	4.32E−09	7.00E−31	1.94E−01
		Permutation	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00

Table 9 continued

Algorithm	N	*p* value	W = 15		W = 18			
			R = 5	R = 3	R = 9	R = 6	R = 3	R = 2
		Wilcoxon	2.95E−08	5.06E−01	3.64E−10	2.33E−09	5.32E−10	5.44E−01
	10	*T* test	9.55E−07	3.20E−06	1.53E−17	2.56E−10	6.38E−02	5.16E−27
		Permutation	0.00E+00	0.00E+00	2.00E−03	0.00E+00	0.00E+00	0.00E+00
		Wilcoxon	2.47E−07	2.58E−06	2.87E−11	2.13E−09	2.87E−11	2.87E−11

Table 10 The results of statistical test for MGALA algorithm with LS-Mutation and depth of memory 2 versus MGALA algorithm with LS-Mutation and other depths of memory

Algorithm	N	*p* value	W = 4	W = 6		W = 9	W = 12			
			R = 2	R = 3	R = 2	R = 3	R = 6	R = 4	R = 3	R = 2
MGALA	4	*T* test	2.72E−06	2.51E−01	4.67E−05	2.81E−02	4.69E−15	4.29E−05	6.58E−08	1.00E+00
		Permutation	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	9.40E−02	9.88E−01	0.00E+00
		Wilcoxon	8.51E−07	1.60E−01	3.59E−05	1.10E−02	2.87E−11	3.16E−05	1.11E−07	8.53E−01
	6	*T* test	5.46E−09	2.77E−11	1.98E−06	3.21E−01	2.31E−18	2.31E−06	2.61E−03	5.28E−03
		Permutation	0.00E+00	0.00E+00	2.00E−03	0.00E+00	0.00E+00	1.30E−02	7.00E−03	2.00E−03
		Wilcoxon	5.22E−09	8.56E−11	4.58E−06	5.06E−01	2.87E−11	4.74E−06	3.03E−03	9.27E−03
	8	*T* test	8.00E−24	2.47E−23	1.23E−02	7.86E−04	1.59E−21	7.15E−04	2.36E−05	1.58E−05
		Permutation	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
		Wilcoxon	2.87E−11	2.87E−11	1.12E−02	1.52E−03	2.87E−11	1.48E−03	1.29E−05	8.58E−06
	10	*T* test	2.84E−24	6.33E−27	7.03E−04	5.50E−07	5.50E−27	4.08E−01	2.09E−12	1.93E−01
		Permutation	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	1.95E−01	0.00E+00
		Wilcoxon	2.87E−11	2.87E−11	1.49E−04	1.86E−06	2.87E−11	4.69E−01	2.26E−10	1.41E−01

Algorithm	N	*p* value	W = 15		W = 18			
			R = 5	R = 3	R = 9	R = 6	R = 3	R = 2
MGALA	4	*T* test	9.30E−02	1.02E−07	3.21E−04	1.48E−07	4.88E−15	2.48E−11
		Permutation	0.00E+00	0.00E+00	2.40E−02	0.00E+00	1.31E−01	0.00E+00
		Wilcoxon	6.57E−02	2.28E−07	3.09E−04	4.44E−07	2.87E−11	1.15E−10
	6	*T* test	1.04E−02	1.31E−12	5.62E−16	3.21E−03	8.65E−08	2.19E−18
		Permutation	0.00E+00	0.00E+00	3.77E−01	0.00E+00	0.00E+00	0.00E+00
		Wilcoxon	6.67E−03	3.88E−11	2.87E−11	6.52E−03	1.10E−08	2.87E−11
	8	*T* test	8.06E−09	1.50E−13	3.64E−19	1.46E−08	3.44E−16	9.50E−29

Table 10 continued

Algorithm	N	*p* value	W = 15		W = 18			
			R = 5	R = 3	R = 9	R = 6	R = 3	R = 2
		Permutation	4.00E−03	0.00E+00	1.00E−03	1.30E−02	0.00E+00	0.00E+00
		Wilcoxon	3.80E−08	3.51E−11	2.87E−11	1.49E−08	3.51E−11	2.87E−11
	10	*T* test	4.88E−12	6.94E−18	3.31E−24	9.77E−12	3.54E−24	2.74E−35
		Permutation	4.31E−01	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
		Wilcoxon	5.32E−10	2.87E−11	2.87E−11	1.04E−10	2.87E−11	2.87E−11

6 The graph isomorphism problem

A graph is described by $G = (E, V)$ where V is the set of vertices and $E \subset V * V$ is the set of edges. Two graphs $G = (E_1, V_1)$ and $H = (E_2, V_2)$ are isomorphic if, and only if, their adjacency matrices $M(G)$ and $M(H)$ differ only by permutations of rows and columns, i.e., $M(G)$ and $M(H)$ are related with a permutation σ, according to Eq. (2).

$$M(H) = P * M(G) * P^T \rightarrow \left[P * M(G) * P^T\right]_{i,j} = [M(H)]_{\sigma(i),\sigma(j)}, \tag{2}$$

where P is the permutation matrix of σ. If we define the difference between two graph as

$$J(\sigma) = \left|\left|M(H) - P * M(G) * P^T\right|\right|, \tag{3}$$

where $||\cdot||$ is the matrix norm, defined as $||M|| = \sum_i \sum_j |m_{ij}|$, the GIP can then be formulated as a search optimization problem in searching for a permutation σ that minimizes $J(\sigma)$.

The mapping error of vertex k in graph G to vertex $\sigma(k)$ in graph H is defined as

$$J_k(\sigma) = \sum_{m=1}^{n} \left| [M(H)]_{k,m} - [M(G)]_{\sigma(k),\sigma(m)} \right| + \sum_{m=1}^{n} \left| [M(H)]_{m,k} - [M(H)]_{\sigma(m),\sigma(k)} \right|, \tag{4}$$

where n is the number of vertices of G and H. For undirected graphs the mapping error can be computed according to Eq. (5), as given below:

$$J_k(\sigma) = 2 * \sum_{m=1}^{n} \left| [M(H)]_{k,m} - [M(H)]_{\sigma(k),\sigma(m)} \right| \tag{5}$$

Consequently, the mapping error of graphs G and H can be computed using Eq. (6).

$$J(\sigma) = \sum_{k=1}^{n} J_k(\sigma) \tag{6}$$

In this paper we use $f_g = C_{max} - J(\sigma)$ as genetic fitness, where C_{max} is the maximum of $J(\sigma)$ [31].

6.1 The local search in the graph isomorphism problem

If two graphs are isomorphic to each other, then the weight and the number of input and output edges of isomorphic vertices must be equal. This is taken into consideration in the design of the local search procedure [31]. Pseudo code for our local search method is given in Fig. 32. This local search method consists of the following steps:

1. The vertices are partitioned into a number of subsets of equal weight.
2. The worst gene of the current chromosome is selected (line 2 of Fig. 32).

```
1. Procedure LocalSearch( CR )
2. g1 = Worst Gene(CR);
3. g2 = Select a Random Gene From Same Subset(g1);
4. Swap(CR(g1), CR(g2));
5. End LocalSearch;
```

Fig. 32 Pseudo code for the local search in the GIP

3. The value of the selected gene is swapped with the value of a random gene selected from the same subset (lines 3 and 4 of Fig. 32).

6.2 Experimental results

In this section, several experiments are described that studied the effect of different MGALA parameters on the performance of the GIP. For this purpose we used a database with 10,000 coupled pairs of isomorphic graphs with different sizes [32]. We classified these graphs into three groups: small graphs ($n < 50$), medium graphs ($50 \leq n < 100$), and large graphs ($100 \leq n < 200$). MGALA results are compared with the results obtained from an algorithm based on a GA [31], an algorithm reported by Ullmann [33], and the VF and VF2 algorithms [34]. The source codes for these algorithms are available at http://amalfi.dis.unina.it/graph. Every result reported is the average of 30 runs. For all experiments an initial population of size 100 was created randomly, the chromosome size was set equal to the size of the graph, the mutation rate and crossover rate were both set to 0.05, and the selection mechanism was $(\mu + \lambda)$. Each algorithm terminates when either solution has been found or the number of generations exceeds 10,000. A Tsetline-based OMA is used for all experiments to represent chromosomes. We use RT to refer to running time, FE to refer to the number of fitness evaluations for the runs converged to the solution, and NR to refer to the number of runs not converged to the solution. All experiments were performed on three classes of graphs: small graphs (SG), medium graphs (MG) and large graphs (LG). We performed a parametric test (*T* test) and two non-parametric tests (wilcoxon rank sum test and permutation test) at the 95 % significance level to provide statistical confidence. The *T* tests were performed after ensuring that the data followed a normal distribution (by using the Kolmogorov–Smirnov test).

6.2.1 Experiment 1

Experiment 1 aimed to find the optimal memory depth for different classes of graphs for the MGALA algorithm. For this purpose we studied the effect of parameter *N* (depth of memory) on the FE, RT, and NR. The MGALA results were then compared with the results obtained for the Canonical Memetic Algorithm (CMA). Note that MGALA is equivalent to the CMA when $N = 0$. For this experiment the graph density was set to 0.5, and weights for vertices and edges were chosen from

Table 11 performance of MGALA with respect to depth of memory

Algorithm	Depth of memory	SG					MG					LG				
		FE		RT		NR	FE		RT		NR	FE		RT		NR
		Avg.	Std.	Avg.	Std.		Avg.	Std.	Avg.	Std.		Avg.	Std.	Avg.	Std.	
CMA	0	68	4.60E+00	1.07	5.80E−02	14	373	1.27E+02	10.09	4.20E+00	18	963	8.99E+01	36.07	5.60E+00	27
MGALA	2	85	5.30E+00	1.09	4.40E−02	0	902	8.93E+01	20.2	3.50E+00	7	2,634	2.36E+02	86.54	9.60E+00	23
	4	86	3.50E+00	1.08	5.30E−02	0	990	7.86E+01	23.01	3.20E+00	0	3,346	2.68E+02	117.16	1.23E+01	9
	6	84	4.60E+00	1.1	2.30E−02	0	969	8.65E+01	21.42	4.30E+00	0	3,085	3.24E+02	101.62	1.03E+01	7
	8	81	6.20E+00	1.07	4.30E−02	0	896	7.61E+01	20.04	3.60E+00	0	3,351	2.68E+02	109.87	9.50E+00	6
	10	87	5.40E+00	1.13	3.30E−02	0	957	6.84E+01	21.17	4.30E+00	0	3,473	2.45E+02	114.09	1.32E+01	0
	12	87	7.10E+00	1.14	6.50E−02	0	926	5.99E+01	20.42	3.10E+00	0	3,246	3.26E+02	106.58	8.60E+00	0
	14	86	5.30E+00	1.14	5.40E−02	0	933	6.53E+01	20.54	3.30E+00	0	3,314	2.36E+02	108.14	7.60E+00	0
	16	89	4.20E+00	1.15	3.60E−02	0	927	7.16E+01	20.49	2.30E+00	0	3,529	2.89E+02	115.5	1.03E+01	0
	18	90	4.10E+00	1.18	4.70E−02	0	903	8.03E+01	19.86	3.50E+00	0	3,289	3.25E+02	107.75	1.24E+01	0
	20	84	5.20E+00	1.1	5.60E−02	0	977	6.82E+01	21.48	3.10E+00	0	3,397	2.59E+02	111.11	1.13E+01	0

Table 12 The results of statistical test for MGALA algorithm with depth of memory $N = 2$ versus MGALA algorithm with other depths of memory

Algorithm	Depth of memory	SG			MG			LG		
		T test	Permutation	Wilcoxon	*T* test	Permutation	Wilcoxon	*T* test	Permutation	Wilcoxon
MGALA	4	3.96E−01	7.11E−01	5.59E−01	3.48E−04	0.00E+00	3.77E−04	8.60E−12	0.00E+00	4.40E−10
	6	4.41E−01	3.37E−01	3.33E−01	6.20E−03	3.00E−03	6.82E−03	1.02E−06	0.00E+00	6.05E−07
	8	1.18E−02	5.00E−03	1.01E−02	7.81E−01	7.59E−01	6.05E−01	7.26E−12	0.00E+00	8.15E−11
	10	1.58E−01	1.22E−01	1.52E−01	1.21E−02	6.00E−03	2.87E−02	4.82E−14	0.00E+00	2.87E−11
	12	2.26E−01	2.16E−01	1.83E−01	2.31E−01	2.39E−01	3.99E−01	3.44E−09	0.00E+00	7.12E−09
	14	4.71E−01	6.96E−01	7.96E−01	1.36E−01	1.42E−01	2.58E−01	5.10E−12	0.00E+00	1.15E−10
	16	3.00E−03	5.00E−03	5.32E−03	2.41E−01	2.45E−01	4.38E−01	9.61E−14	0.00E+00	2.87E−11
	18	3.16E−04	0.00E+00	2.60E−04	9.64E−01	9.89E−01	9.59E−01	7.95E−10	0.00E+00	3.66E−09
	20	4.67E−01	2.12E−01	2.40E−01	1.01E−03	1.00E−03	1.11E−03	1.05E−12	0.00E+00	5.77E−11

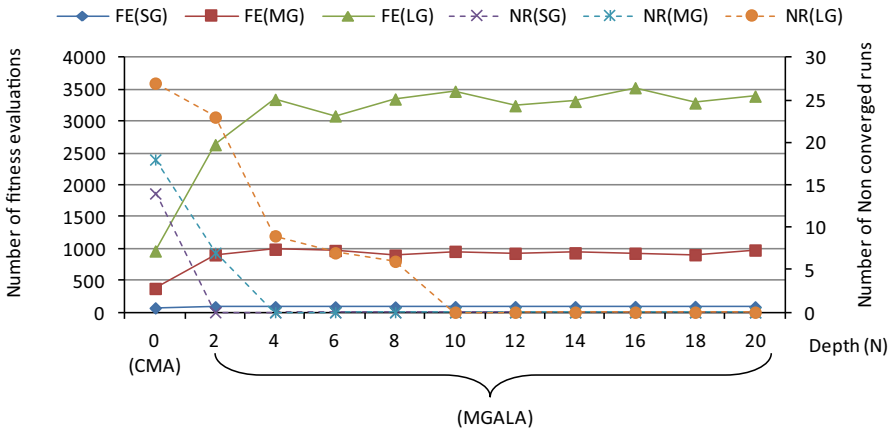


Fig. 33 Number of fitness evaluations (FE) and number of non-converged runs (NR) vs. depth of memory for different classes of graphs

[0,100]. Table 11 lists the RT, FE, NR, and the standard deviation for the different depths of memory employed. From the results we conclude the following:

- For all classes of graphs the minimum value for RT and FE are obtained when $N = 0$,
- For all classes of graphs the maximum value for NR is obtained when $N = 0$, and
- For all classes of graphs the NR is inversely proportional to depth of memory.

Table 12 shows that, according to results of wilcoxon test, permutation test and *T* test, the MGALA algorithm with a memory depth of $N = 2$ performs better than the MGALA algorithm for $N \neq 2$ for large graphs (LG).

Figure 33 shows the impact of the depth of memory on the FE and NR for different classes of graphs. Changes in the FE are minor for a depth of memory greater than 4 in all classes of graphs. Also, this figure shows that for all classes of graphs, a depth of memory greater than 10 causes convergence ($NR = 0$) in all runs.

6.2.2 *Experiment 2*

This experiment investigated the effect of graph edge and vertex weights on MGALA performance. We studied the effect of weights on the FE, RT, and NR for different classes of graphs using MGALA and CMA (MGALA when $N = 0$). For this experiment the density of all graphs was set to 0.5 and N was set to 10. The experiment was repeated for five different weight ranges: [0,20], [0,40], [0,60], [0,80], and [0,100]. The experiment was also repeated with unweighted graphs (graphs whose edge weights are chosen from {0,1}, and whose nodes have no weights). Table 13 gives the RT, FE, NR, and standard deviation for the different weight parameters.

Table 13 Performance of MGALA with respect to the weight parameter

Algorithm	Weight	SG					MG					LG				
		FE		RT		NR	FE		RT		NR	FE		RT		NR
		Avg.	Std.	Avg.	Std.		Avg.	Std.	Avg.	Std.		Avg.	Std.	Avg.	Std.	
CMA	[0,100]	68	6.50E+00	0.99	5.30E−02	14	373	2.55E+01	10.09	3.50E+00	18	963	1.12E+02	36.07	6.50E+00	27
MGALA	Unweighted	102	1.52E+01	1.29	5.60E−02	1	2,601	1.96E+02	59.14	8.50E+00	10	10,791	6.59E+02	355.84	2.65E+01	26
	[0,20]	99	8.20E+00	1.27	6.30E−02	0	883	1.09E+02	19.54	5.20E+00	0	3,723	2.37E+02	121.33	1.43E+01	3
	[0,40]	82	9.40E+00	1.04	2.50E−02	0	823	9.86E+01	18.22	4.60E+00	0	4,262	3.25E+02	139.73	1.26E+01	1
	[0,60]	76	6.20E+00	0.97	2.40E−02	0	892	1.15E+02	19.68	5.60E+00	0	3,665	2.36E+02	119.69	1.25E+01	0
	[0,80]	81	1.16E+01	1.03	3.60E−02	0	835	8.53E+01	18.44	3.50E+00	0	4,052	3.21E+02	130.51	9.50E+00	0
	[0,100]	74	5.20E+00	0.99	4.70E−02	0	815	7.56E+01	15.09	2.60E+00	0	3,535	2.37E+02	114.94	7.50E+00	0

Table 14 The results of statistical tests for MGALA algorithm with weight parameter [0,100] vs. MGALA algorithm with other values of weight parameter

Algorithm	Weight of graph	SG			MG			LG		
		T test	Permutation	Wilcoxon	*T* test	Permutation	Wilcoxon	*T* test	Permutation	Wilcoxon
MGALA	Unweighted	1.87E−10	0.00E+00	1.86E−10	8.58E−29	0.00E+00	2.87E−11	2.78E−31	0.00E+00	2.87E−11
	[0,20]	1.63E−14	0.00E+00	3.18E−11	8.68E−03	7.00E−03	3.59E−03	4.56E−03	3.00E−03	3.67E−03
	[0,40]	3.23E−04	0.00E+00	7.10E−04	7.27E−01	7.45E−01	9.18E−01	8.48E−11	0.00E+00	3.31E−10
	[0,60]	1.86E−01	1.29E−01	1.24E−01	4.74E−03	1.00E−03	6.24E−03	4.18E−02	2.90E−02	2.32E−02
	[0,80]	5.28E−03	1.00E−03	8.14E−03	3.44E−01	3.60E−01	3.40E−01	8.34E−08	0.00E+00	1.11E−07

From these experimental results we conclude the following for MGALA:

- For all classes of graphs RT and FE are minimized when the weights for vertices and edges are chosen from [0,100] and are maximized for the unweighted graph. This is because GIP only uses the properties of local search method. If the weights of the graph have higher values, then the vertices of the graph are partitioned into a higher number of subsets with a lesser number of members. The effect of this is to cause just the vertices of the subset, which have same weight as the worst gene, to have a chance for exchanging with the worst gene in the local search method. Consequently, the local search selects an alternative vertex accurately in weighted graphs.
- For all classes of graphs the NR is inversely proportional to the weights of the vertices and edges.
- For all classes of graphs the maximum value of NR is obtained when CMA is used.

Table 14 shows that, for all three kinds of statistical tests (wilcoxon, permutation and *T* test), the difference between the performance of the MGALA algorithm when weights for vertices and edges are chosen from [0,100], and the performance of the MGALA algorithm when it uses other weight parameter values that are statistically significant (p value <0.05) for most graphs.

Figure 34 shows the FE for different graph classes and weights. The FE for all classes of graphs when the weights are chosen from ranges greater than [0,20] are almost the same. Figure 34 also shows that with all graph classes the NR is equal to zero when the weights are chosen from ranges greater than [0,60]. Consequently, the FE and NR are both minimized when the weights are chosen from ranges greater than [0,60] (ranges [0,80] or [0,100]).

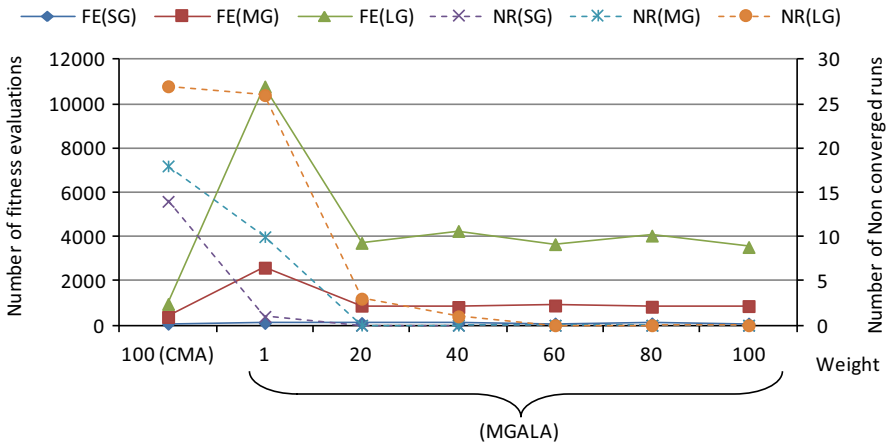


Fig. 34 Number of fitness evaluations (FE) and number of non-converged runs (NR) vs. the weight parameter for different classes of graphs

Table 15 performance of MGALA with respect to graph density

Algorithm	Density (D)	SG					MG					LG				
		FE		RT		NR	FE		RT		NR	FE		RT		NR
		Avg.	Std.	Avg.	Std.		Avg.	Std.	Avg.	Std.		Avg.	Std.	Avg.	Std.	
CMA	0.5	64	6.80E+00	0.87	6.00E−02	14	373	3.65E+01	10.09	1.62E+00	18	963	8.86E+01	36.06	3.65E+00	27
MGALA	0.1	85	7.30E+00	1.03	2.30E−02	0	2,428	2.63E+02	56.05	3.62E+00	0	13,166	1.16E+02	438.07	5.32E+00	12
	0.2	93	6.50E+00	1.01	2.50E−02	0	1,114	1.36E+02	25.32	3.63E+00	0	8,396	4.56E+02	276.62	2.65E+00	8
	0.3	91	8.40E+00	1.21	3.80E−02	0	1,063	1.20E+02	28.35	4.56E+00	1	4,470	3.56E+02	146.42	2.78E+00	5
	0.4	89	6.40E+00	1.81	1.60E−02	0	818	6.95E+01	17.65	2.37E+00	0	4,116	3.66E+02	233.25	2.69E+00	5
	0.5	96	7.60E+00	1.27	2.90E−02	0	725	5.62E+01	16.35	2.76E+00	0	3,357	2.36E+02	110.52	2.31E+00	4
	0.6	77	5.60E+00	1.04	3.60E−02	0	790	5.84E+01	17.66	1.69E+00	0	3,369	2.38E+02	111.31	2.57E+00	0
	0.7	78	6.10E+00	1.02	3.10E−02	0	809	6.53E+01	18.18	1.67E+00	0	3,530	3.13E+02	115.86	3.56E+00	1
	0.8	87	5.60E+00	1.09	3.80E−02	0	807	7.62E+01	17.81	2.36E+00	0	3,284	3.45E+02	107.11	1.38E+00	0
	0.9	73	7.10E+00	0.98	1.20E−02	0	723	4.56E+01	18.17	2.49E+00	0	3,259	2.89E+02	105.92	1.25E+00	0
	1	71	4.20E+00	0.95	1.50E−02	0	710	4.16E+01	19.93	1.94E+00	0	3,124	2.64E+02	103.01	1.09E+00	0

Table 16 The results of statistical tests for MGALA algorithm with density 1 versus MGALA algorithm with other values of density parameter

Algorithm	Density of graph	SG			MG			LG		
		T test	Permutation	Wilcoxon	*T* test	Permutation	Wilcoxon	*T* test	Permutation	Wilcoxon
MGALA	0.1	5.30E−10	0.00E+00	9.76E−10	2.19E−25	0.00E+00	2.87E−11	1.69E−46	0.00E+00	2.87E−11
	0.2	1.28E−15	0.00E+00	2.87E−11	1.35E−15	0.00E+00	2.87E−11	7.77E−31	0.00E+00	2.87E−11
	0.3	1.80E−12	0.00E+00	9.92E−11	2.43E−15	0.00E+00	2.87E−11	2.26E−16	0.00E+00	2.87E−11
	0.4	1.60E−13	0.00E+00	1.04E−10	4.82E−08	0.00E+00	1.47E−07	8.13E−13	0.00E+00	1.15E−10
	0.5	9.18E−16	0.00E+00	3.02E−11	2.50E−01	2.41E−01	7.48E−02	1.16E−03	2.00E−03	1.41E−03
	0.6	5.92E−05	0.00E+00	3.83E−05	1.17E−06	0.00E+00	3.96E−07	7.35E−04	1.00E−03	7.49E−04
	0.7	1.55E−05	0.00E+00	5.66E−06	1.06E−07	0.00E+00	1.87E−07	7.52E−06	0.00E+00	3.45E−06
	0.8	3.21E−13	0.00E+00	4.73E−11	1.15E−06	0.00E+00	3.96E−07	5.30E−02	4.10E−02	5.10E−02
	0.9	1.95E−01	1.10E−01	1.33E−01	2.58E−01	2.53E−01	2.04E−01	6.90E−02	5.20E−02	7.24E−02

6.2.3 Experiment 3

Experiment 3 studied the effect of graph density (D) on MGALA performance. The density of a graph is defined as $= \frac{2|E|}{|V|(|V|-1)}$, which is the probability of the existence of an edge between any two vertices. For this experiment the weights of vertices and edges were chosen from [0,100], and N was set to 10. The impact of graph density on the FE, RT, NR, and the standard deviation for different classes of graphs using both MGALA and CMA are reported in Table 15. From these results we conclude the following:

- For all classes of graph RT and FE are minimized when the graph density is 1.
- For all classes of graphs NR decreases as the graph density increases.
- For all classes of graphs a maximum value for NR is obtained when CMA is used.

Table 16 shows that, for all three kinds of statistical tests (wilcoxon, permutation and T test), the difference between the performance of the MGALA algorithm when the density is 1, and the performance of the MGALA algorithm when it uses other density parameter values, is statistically significant (p value <0.05) for most graphs.

Figure 35 shows the impact of graph density on the FE for different classes of graphs. The FE remains almost fixed for graph densities >0.5 for all classes of graphs. Figure 35 also shows that for all classes of graphs, all runs converge (NR) to zero when the graph density is >0.6.

6.2.4 Experiment 4

The experimental goal here was to study the impact of different mutation and crossover operators on MGALA performance. For this experiment the density of all

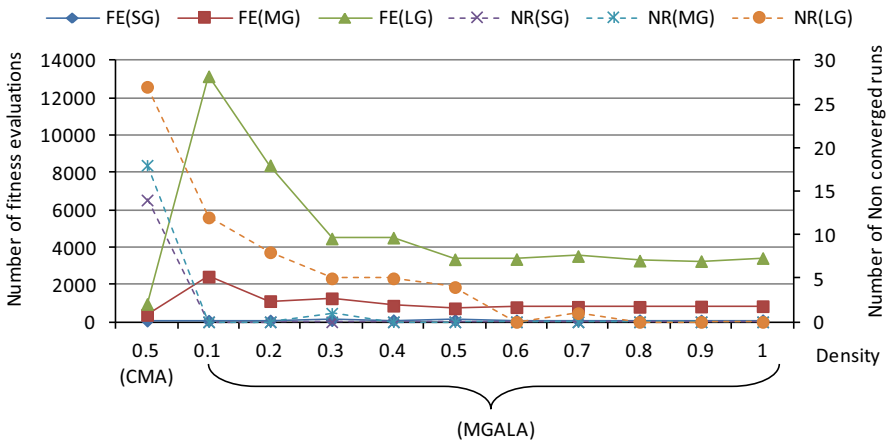


Fig. 35 Number of fitness evaluations (FE) and number of non-converged runs (NR) vs. density of graph for different classes of graphs

Table 17 Performance of MGALA for different mutation and crossover operators

Algorithm	Operators type	SG					MG					LG				
		FE		RT		NR	FE		RT		NR	FE		RT		NR
		Avg.	Std.	Avg.	Std.		Avg.	Std.	Avg.	Std.		Avg.	Std.	Avg.	Std.	
CMA	Canonical operators	68	5.63E+00	1.1	5.60E−02	14	373	2.56E+01	10.09	1.36E+00	18	963	9.86E+01	36.06	3.69E+00	27
MGALA	SS-Mutation, SS-Crossover	90	9.45E+00	1.15	2.40E−02	0	951	5.63E+01	21	1.98E+00	0	3,375	1.25E+02	109.16	8.96E+00	1
	XS-Mutation, XS-Crossover	86	8.35E+00	1.11	3.50E−02	0	927	4.85E+01	20.25	3.25E+00	0	3,752	2.36E+02	120.95	1.37E+01	4
	LS-Mutation, LS-Crossover	92	7.36E+00	1.13	3.70E−02	0	921	7.54E+01	20.13	3.45E+00	0	4,043	2.40E+02	128.65	9.54E+00	0

Table 18 The results of statistical tests for MGALA algorithm with SS-Mutation, SS-Crossover vs. MGALA algorithm with other operators

Algorithm	Operators type	SG			MG			LG		
		T test	Permutation	Wilcoxon	*T* test	Permutation	Wilcoxon	*T* test	Permutation	Wilcoxon
MGALA	XS-Mutation, XS-Crossover	9.29E−02	1.22E−01	1.98E−01	8.74E−02	8.30E−02	8.63E−02	1.64E−08	0.00E+00	1.26E−08
	LS-Mutation, LS-Crossover	3.68E−01	3.52E−01	3.18E−01	9.13E−02	9.00E−02	9.78E−02	4.64E−14	0.00E+00	3.88E−11

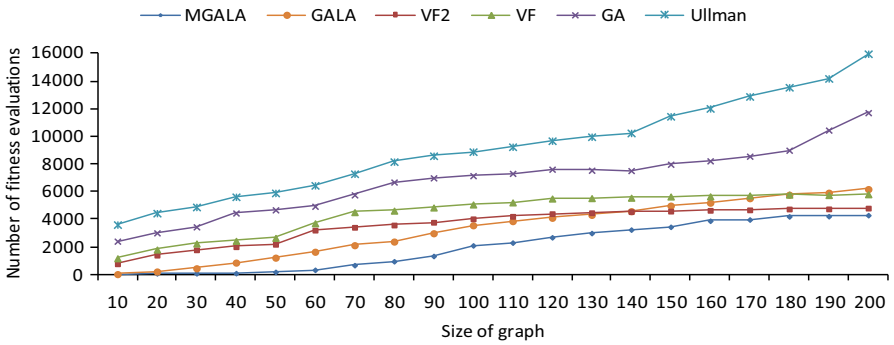


Fig. 36 Number of fitness evaluations vs. graph size for different algorithms

graphs was set to 0.5, and the depth of memory was set to 10. Weights for vertices and edges were selected from [0,100]. Table 17 lists the RT, FE, NR, and the standard deviations from different mutation and crossover operators. These results lead us to conclude the following:

- For all classes of graphs a minimum NR value is obtained when the LS-Mutation and LS-Crossover operators are used.
- For large size graphs (LG), with respect to the FE and RT, MGALA with the SS-Mutation and SS-Crossover operators outperforms both MGALA with the XS-Mutation and XS-Crossover operators, as well as MGALA with the LS-Mutation and LS-Crossover operators.
- For medium size graphs (MG), with respect to FE and RT, MGALA with the LS-Mutation and LS-Crossover operators outperforms both MGALA with the SS-Mutation and SS-Crossover operators, as well as MGALA with the XS-Mutation and XS-Crossover operators.
- For small size graphs (SG), with respect to FE and RT, MGALA with the XS-Mutation and XS-Crossover operators outperforms both MGALA with the SS-Mutation and SS-Crossover operators, as well as MGALA with the LS-Mutation and LS-Crossover operators.
- For all classes of graphs the maximum value for NR is obtained when CMA is used.

According to Table 18, the MGALA algorithm with the SS-Mutation and SS-Crossover operators performs better than the MGALA algorithm with other mutation operators for large graphs.

6.2.5 Experiment 5

MGALA was compared with five other algorithms in this experiment (GA [31], Ullmann [33], VF and VF2 [34], and GALA [11]) for the GIP, in terms of the number of fitness evaluations required. The result of this experiment is shown in

Fig. 36. Each result was the average of 30 runs. The graph size was varied from 10 to 200 by an increment of 10. The results clearly show the superiority of MGALA.

7 Conclusions

A new memetic algorithm called MGALA is proposed for optimization purposes in this paper. MGALA, which is a newly revised version of GALA, is obtained from a combination of a GA and an LA, in which the LA plays the role of providing the local search. Unlike GALA, which uses Lamarckian learning, MGALA uses a Baldwinian learning model to improve its convergence rate and the quality of its solution. In this model, chromosomes are represented by OMAs, and the OMA states keep information about the history of the local search process. Each state in the OMA has two attributes: the value of the gene (allele), and the degree of association with the value of the gene. The local search changes the degree of association between genes and their values. Unlike GALA, which only uses the value of the genes for its fitness computation, MGALA uses all the information recorded in the OMA representation of the chromosome (i.e., the degree of association between genes and their allele, and the values of the genes) to compute the fitness of genes. In other words, MGALA's fitness function is computed using a chromosome's fitness (as genotype information) and the history of the local search kept in the states of the OMA (as phenotype information). The EPP and GIP applications were used to investigate the performance of MGALA. MGALA was also compared with some other well-known algorithms for the EPP and GIP application. Our experimental results showed the superiority of the proposed algorithm in terms of quality of solution and in the rate of convergence. This line of research could be extended in several directions, such as in applying GALA and/or MGALA in solving optimization problems where the environment is dynamic. Extension examples include the dynamic shortest path problem and the dynamic traveling salesman problem, improving the proposed algorithms by designing new mutation or crossover operators, and designing new object migrating automata to be used for chromosome representation. Another direction that may be pursued is the development of a mathematical framework for analyzing the proposed algorithm.

References

1. M. Weber, F. Neri, V. Tirronen, Distributed differential evolution with explorative–exploitative population families. Genet. Progr. Evolvable Mach. **10**, 343–371 (2009)
2. K.W. Ku, M.-W. Mak, Empirical analysis of the factors that affect the Baldwin effect, in *Parallel Problem Solving from Nature—PPSN V* (1998), pp. 481–490
3. G.M. Morris, D.S. Goodsell, R.S. Halliday, R. Huey, W.E. Hart, R.K. Belew, A.J. Olson, Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function. J. Comput. Chem. **19**, 1639–1662 (1998)
4. C. Xianshun, O. Yew-Soon, L. Meng-Hiot, T. Kay Chen, A multi-facet aurvey on memetic computation. IEEE Trans. Evol. Comput. **15**, 591–607 (2011)

5. N. Krasnogor, J. Smith, A tutorial for competent memetic algorithms: model, taxonomy, and design issues. Evol. Comput. IEEE Trans. **9**, 474–488 (2005)
6. K. Downing, Reinforced genetic programming. Genet. Progr. Evolvable Mach. **2**, 259–288 (2001)
7. K.S. Narendra, M.A.L. Thathachar, *Learning automata: an introduction* (Prentice-Hall, Inc, Upper Saddle River, 1989)
8. M.A.L. Thathachar, P.S. Sastry, Varieties of learning automata: an overview. IEEE Trans. Syst. Man Cybern. B Cybern. **32**, 711–722 (2002)
9. M. Rezapoor, M.R. Meybodi, A hybrid algorithm for solving graph isomorphism problem, in *Proceedings of the Second International Conference on Information and Knowledge Technology (IKT2005), Tehran, Iran* (2005)
10. B.J. Oommen, D.C.Y. Ma, Deterministic learning automata solutions to the equipartitioning problem. IEEE Trans. Comput. **37**, 2–13 (1988)
11. M. Rezapoor, M.R. Meybodi, Improving GA+ LA algorithm for solving graph isomorphic problem, in *Proceedings of the 11th Annual CSI Computer Conference of Iran, Tehran, Iran* (2006), pp. 474–483
12. K. Asghari, A. Safari Mamaghani, F. Mahmoudi, M.R. Meybodi, A relational databases query optimization using hybrid evolutionary algorithm. J. Comput. Robot. **1**, 28–39 (2008)
13. K. Asghari, A. Safari Mamaghani, M.R. Meybodi, An evolutionary approach for query optimization problem in database, in *Procceding of Internatinal Joint Conferance on Computers, Information and System Sciences, and Engineering (CISSE2007)* (University of Bridgeport, England, 2007)
14. A. Safari Mamaghani, K. Asghari, M.R. Meybodi, F. Mahmoodi, A new method based on genetic algorithm for minimizing join operations cost in data base, in *Proceedings of 13th Annual CSI Computer Conference of Iran, Kish Island, Iran* (2008)
15. A. Safari Mamaghani, K. Asghari, F. Mahmoudi, and M. R. Meybodi, A novel hybrid algorithm for joint ordering problem in database queries, in *Proceedings of 6th WSEAS international Conference on Computational Intelligence, Man-Machine Systems and Cybernetics, Tenerife, Spain* (2007), pp. 104–109
16. B. Zaree, M. R. Meybodi, An evolutionary method for solving symmetric TSP, in *Proceedings of the Third International Conference on Information and Knowledge Technology (IKT2007), Mashhad, Iran* (2007)
17. B. Zaree, M.R. Meybodi, M. Abbaszadeh, A hybrid method for solving traveling salesman problem. IEEE/ACIS International Conference on Computer and Information Science, ICIS **2007**, 394–399 (2007)
18. B. Zaree, K. Asghari, M.R. Meybodi, A hybrid method based on clustering for solving large traveling salesman problem, in *Proceedings of 13th Annual CSI Computer Conference of Iran, Kish Island, Iran* (2008)
19. K. Asghari, M.R. Meybodi, Searching for Hamiltonian cycles in graphs using evolutionary methods, in *Proceedings of the second Joint Congress on Fuzzy and Intelligent Systems, Tehran, Iran* (2008)
20. B. Zaree, M.R. Meybodi, A hybrid method for sorting problem, in *Proceedings of the Third International Conference on Information and Knowledge Technology (IKT2007), Mashhad, Iran* (2007)
21. A. Safari Mamaghani, M.R. Meybodi, Hybrid algorithms (learning automata + genetic algorithm) for solving graph bandwidth minimization problem, in *Proceedings of the second Joint Congress on Fuzzy and Intelligent Systems, Tehran, Iran* (2008)
22. A.S. Mamaghani, M.R. Meybodi, A learning automaton based approach to solve the graph bandwidth minimization problem, in *International Conference on Application of Information and Communication Technologies (AICT)* (2011), pp. 1–5
23. A. Isazadeh, H. Izadkhah, A. Mokarram, A learning based evolutionary approach for minimization of matrix bandwidth problem. Appl. Math. **6**, 51–57 (2012)
24. A.S. Mamaghani, M.R. Meybodi, Clustering of software systems using new hybrid algorithms, in *Ninth IEEE International Conference on Computer and Information Technology* (2009), pp. 20–25
25. A. Safari Mamaghani, M.R. Meybodi, Hybrid evolutionary algorithms for solving software clustering problem, in *Proceedings of the second Joint Congress on Fuzzy and Intelligent Systems, Tehran, Iran* (2008)
26. K. Asghari, M.R. Meybodi, Solving single machine total weighted tardiness scheduling problem using learning automata and genetic algorithm, in *Proceedings of the 3rd Iran Data Mining Conference(IDMC'09), Tehran Iran* (2009)

27. A.S. Mamaghani, M. Mahi, M.R. Meybodi, A learning automaton based approach for data fragments allocation in distributed database systems, in *IEEE 10th International Conference on Computer and Information Technology (CIT)* (2010), pp. 8–12
28. A.S. Mamaghani, M. Mahi, M.R. Meybodi, M.H. Moghaddam, A novel evolutionary algorithm for solving static data allocation problem in distributed database systems, in *Second International Conference on Network Applications Protocols and Services (NETAPPS)* (2010), pp. 14–19
29. V. Majid Nezhad, H. Motee Gader, E. Efimov, A novel hybrid algorithm for task graph scheduling. Int. J. Comput. Sci. Issues **8**, 32–38 (2011)
30. A. Bansal, R. Kaur, Task graph scheduling on multiprocessor system using genetic algorithm. Int. J. Eng. Res. Technol. **1**, 1–5 (2012)
31. W. Yuan-Kai, F. Kuo-Chin, H. Jorng-Tzong, Genetic-based search for error-correcting graph isomorphism. IEEE Trans. Syst. Man Cybern. B Cybern. **27**, 588–597 (1997)
32. P. Foggia, C. Sansone, M. Vento, A database of graphs for isomorphism and sub-graph isomorphism benchmarking, in *Proceedings of the 3rd IAPR TC-15 International Workshop on Graph-based Representations* (2001), pp. 176–187
33. J.R. Ullmann, An algorithm for subgraph isomorphism. J. ACM **23**, 31–42 (1976)
34. L.P. Cordella, P. Foggia, C. Sansone, M. Vento, A (sub) graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. **26**, 1367–1372 (2004)