Memetic Programming Algorithm with Automatically Defined Functions

Emad Mabrouk [1] Abdel-Rahman Hedar [2] Masao Fukushima [3]

November 11, 2010

Abstract

Applications of Artificial Intelligence (AI) are rapidly increasing especially in the infrastructure of every industry, and researchers continually try to develop new efficient AI algorithms or improve the current ones to maximize their benefits. In this paper, we introduce a new hybrid evolutionary algorithm, called the Memetic Programming (MP) algorithm, which hybridizes the Genetic Programming (GP) algorithm with a new set of local search procedures over a tree space. Specifically, in each generation of the MP algorithm, we use the GP strategy to generate a new population. Then, using some local search procedures over a tree space, we try to improve promising programs from the generated population. In addition, the MP algorithm can deal with the Automatically Defined Function (ADF) technique that enables the algorithm to exploit the modularities in problem environments. Through extensive numerical experiments, the proposed MP algorithm is shown to have promising performance compared to some recent versions of the GP algorithm, especially by using the ADF technique.

1 Introduction

Genetic Programming (GP) is one of the most well-known Artificial Intelligence (AI) algorithms [16, 17, 18, 19]. It is inspired from the biological process to produce computer programs as solutions for a given problem. It has received a lot of attention during the last two decades and it has shown promising performance in various applications [1, 2, 4, 6, 15, 20, 33, 34, 37]. Despite that, some important observations have been made about a disruption effect for its main operators, i.e., crossover and mutation [30, 31]. The main idea in those operators is to choose a node randomly from a tree and exchange the subtree below it by a new subtree generated randomly or cut from another tree. Therefore, altering a node high in the tree may result in serious disruption of the subtree below it. In fact, many researchers have attempted to edit GP operators to make changes in small scales [12, 20]. Moreover, some researchers claimed that the local search could be effective in improving the local structure of programs [12, 24].

The aim of this work is to introduce a hybrid evolutionary algorithm, called Memetic Programming (MP) algorithm, as an improvement of the GP algorithm. The proposed algorithm hybridizes GP with new local search procedures over a tree space to intensify promising programs generated by the GP algorithm. These local searches are used to generate trial programs in the neighborhood of the current one by changing it in small scales. In addition, the proposed algorithm can easily deals with the *Automatically Defined Function* (ADF) technique to exploit the modularities in problem environments [18]. We will show through numerical experiments that the proposed MP algorithm is more efficient in finding an optimal solution than the GP algorithm especially when using the ADF technique.

The MP algorithm inherits the main idea from the Memetic Algorithms (MAs) [8, 22, 27, 28, 29] which combine the global search characteristic of Evolutionary Algorithms (EAs) with local search techniques to improve individual solutions obtained in the global search phase. In the field of optimization, MAs are known as hybrid EAs that combine global and local searches by using an EA to perform exploration while using a local search method to intensify the results obtained by EA. In fact, MAs have achieved

[1]Dept. of Applied Mathematics and Physics, Graduate School of Informatics, Kyoto University, Kyoto 606-8501, JAPAN Email: hamdy@amp.i.kyoto-u.ac.jp

[2]Department of Computer Science, Faculty of Computers and Information, Assiut University, EGYPT Email: hedar@aun.edu.eg.

[3]Dept. of Applied Mathematics and Physics, Graduate School of Informatics, Kyoto University, Kyoto 606-8501, JAPAN Email: fuku@i.kyoto-u.ac.jp

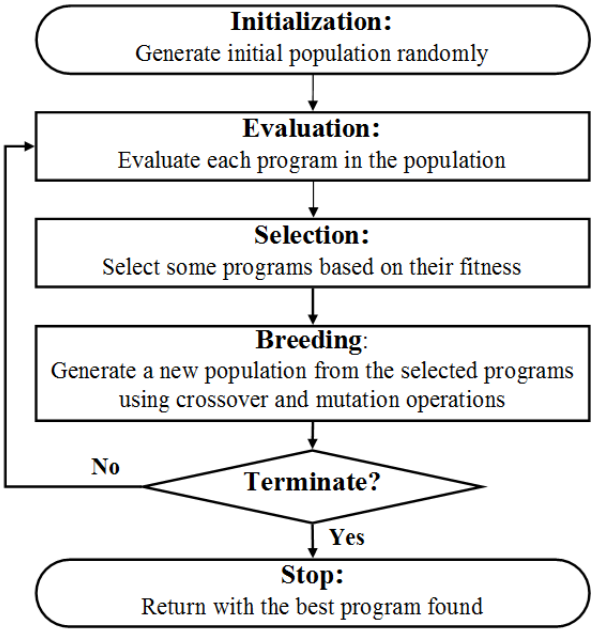

Figure 1: Flowchart of GP algorithm

a great success across a wide range of problem domains such as combinatorial optimization, nonconvex global optimization, and multi-objective optimization [21, 27, 28, 29].

The rest of the paper is organized as follows. In the next section, we introduce more details about GP that represents the global search technique in MP. The basic procedures for stochastic local searches over a tree space are presented in Section 3. The proposed MP algorithm is described in Section 4. In Section 5, more explanations about the practical implementation of the MP algorithm are introduced. In Section 6, we report numerical results for two types of benchmark problems. Finally, conclusions make up Section 7.

2 Genetic Programming

The GP algorithm is an evolutionary algorithm (EA) inspired from the biological processes of natural selection and survival of the fittest [16, 17, 18, 19, 20, 23]. GP evolves a population of computer programs represented as trees to find an acceptable solution for a given problem. The first proposal of "tree-based" genetic programming was given by Cramer [5] in 1985. This work was popularized by Koza [16, 17, 18, 19], and subsequently, the feasibility of this approach in well-known application areas has been demonstrated [1, 4, 6, 15, 20, 33, 34, 37]. First, the GP algorithm generates a population of random computer programs. Each program in the population is evaluated and some good programs are selected and recombined using the mutation and the crossover operators to breed a new population of programs. This process of selection and recombination to breed new programs is repeated until the best program is found. Fig. 1 represents a flowchart of the GP algorithm.

The computer programs treated in GP are represented as trees in which leaf nodes are called terminals and internal nodes are called functions. Depending on the problem at hand, the user defines the domains of terminals and functions. In the coding process, the tree structure of a solution should be transformed to an executable code. Fig. 2 shows two examples of individuals represented as trees, along with their executable codes. Actually, the tree-based representation enables the GP to evolve solutions conveniently, and to cover a wide range of applications.

The crossover and mutation are fundamental operators in the GP algorithm and have received much attention in the literature. A huge amount of papers and books have been published to examine effective settings for these operators and to improve the behavior of the GP algorithm, see for example [20, 35, 37] and the references therein. These efforts have popularized the GP algorithm and expanded the range of GP applications. The crossover operator mates two selected trees (parents) and yields two new trees (offsprings), while the mutation operator is applied to one tree and yields a new mutated tree. Fig. 3 illustrates an example of applying the crossover and mutation operators in GP.

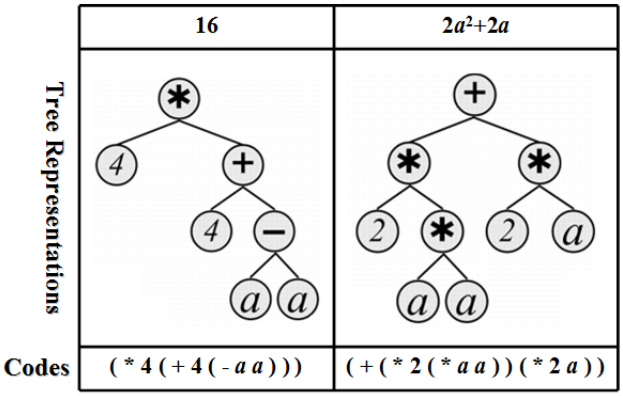

Figure 2: Examples of GP representation

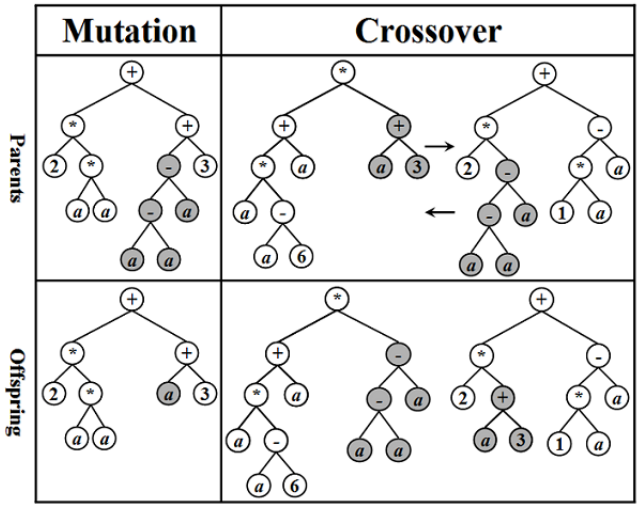

Figure 3: Mutation and crossover operations in GP

On the other hand, a number of authors have pointed out that crossover and mutation operators suffer from some drawbacks [25, 12, 30, 31]. Altering a node high in a tree may result in serious disruption of the subtree rooted at that node, which means GP may lose promising solutions easily. To cope with the difficulty, there have been many attempts to edit GP operators to make changes in small scales, for example by using natural language processing [12, 20]. In particular, the importance of local search has been well recognized, and methods of improving the local structure of individuals have been developed [9, 11, 24, 25, 26].

3 Local Searches over Tree Space

In this section, some local search operators over a tree space are introduced. These operators aim to generate a new tree in a neighborhood of the current tree. We discuss two types of local searches; *static structure search* and *dynamic structure search* [9, 10, 25, 26]. Static structure search aims to explore the neighborhood of a tree by altering its nodes without changing its structure. On the other hand, dynamic structure search changes the structure of a tree by expanding its terminal nodes or cutting its subtrees. *Shaking* operator is shown as a static structure search, while *Grafting* and *Pruning* operators are introduced as a dynamic structure search.

Before proceeding to the description of Shaking, Grafting and Pruning procedures, we introduce some basic notations. For a tree X, we use $|X|$ to denote the size of X (the number of nodes in X), $l(x)$ the number of leaf nodes in X, and $d(X)$ the depth of X (the number of links in the path from the root of X to its farthest node).

3.1 Shaking Search

Shaking search is used to generate a new tree $\tilde{X}$ from the current one X, by altering some nodes of X without changing its structure. The altered nodes are replaced by alternative values, i.e., a terminal node is replaced by a new terminal value and a function node is replaced by a new function of the same number of arguments as the original one. Procedure 1 gives the formal description of shaking search, where λ is a positive integer that must be determined before calling the procedure.

Procedure 1 $\tilde{X} = \texttt{Shaking}(X, \lambda)$

Step 1. *Set $\tilde{X} := X$ and set the counter $j := 1$.*

Step 2. *While $j \leq \lambda$, repeat Steps 2.1-2.3.*

2.1 *Choose a node t_j from $\tilde{X}$ randomly.*

2.2 *Generate an alternative randomly from the set of functions and terminals.*

2.3 *Update $\tilde{X}$ by replacing the chosen node t_j by the new alternative and set $j := j + 1$.*

Step 3. *Return.*

A neighborhood $N_S(X)$ of a tree X, associated with shaking search, is defined by

$$N_S(X) = \left\{ \tilde{X} | \tilde{X} = \texttt{Shaking}(X, \lambda), \lambda = 1, \ldots, |X| \right\}. \tag{1}$$

3.2 Grafting Search

Grafting search is introduced to increase the variability of the search process. Using grafting search, a new tree $\tilde{X}$ is generated from a tree X by replacing some of its terminals by branches of depth $\zeta \geq 1$. Therefore, X and $\tilde{X}$ have different tree structures since $|\tilde{X}| > |X|$, $l(\tilde{X}) > l(X)$, and $d(\tilde{X}) \geq d(X)$. The formal description of grafting search is shown in Procedure 2, where μ and ζ are two positive integers that must be determined before calling the procedure.

Procedure 2 $\tilde{X} = \texttt{Grafting}(X, \mu, \zeta)$

Step 1. *Set $\tilde{X} := X$ and set the counter $j := 1$.*

Step 2. *While $j \leq \mu$, repeat Steps 2.1-2.3.*

2.1 *Generate a branch B_j of depth ζ randomly.*

2.2 *Choose a terminal node t_j from $\tilde{X}$ randomly.*

2.3 *Update $\tilde{X}$ by replacing the node t_j by the branch B_j and set $j := j + 1$.*

Step 3. *Return.*

A neighborhood $N_G(X)$ of a tree X, associated with grafting search, is defined by

$$N_G(X) = \Big\{\tilde{X} | \tilde{X} = \texttt{Grafting}(X, \mu, \zeta), \quad \mu = 1, \ldots, l(X), \zeta = 1, \ldots, \zeta_{max}\Big\}, \tag{2}$$

where ζ_{max} is a predetermined positive integer.

3.3 Pruning Search

In contrast to grafting search, pruning search generates an altered tree $\tilde{X}$ from a tree X by cutting some of its branches of depth ζ. One may note that X and $\tilde{X}$ have different tree structures since $|\tilde{X}| < |X|$, $l(\tilde{X}) < l(X)$, and $d(\tilde{X}) \leq d(X)$. Procedure 3 gives the formal description of pruning search, where ν and ζ are two positive integers that must be determined before calling the procedure.

Procedure 3 $\tilde{X} = \texttt{Pruning}(X, \nu, \zeta)$

Step 1. *Set $\tilde{X} := X$ and set the counter $j := 1$.*

Step 2. *While $\zeta \leq d(\tilde{X})$ and $j \leq \nu$, repeat Steps 2.1-2.3.*

2.1 *Choose a branch B_j of depth ζ in $\tilde{X}$ randomly.*

2.2 *Choose a terminal node t_j randomly from the set of terminals.*

2.3 *Update $\tilde{X}$ by replacing the branch B_j by t_j and set $j := j + 1$.*

Step 3. *Return.*

A neighborhood $N_P(X)$ of a tree X, associated with pruning search, is defined by

$$N_P(X) = \Big\{\tilde{X} | \tilde{X} = \texttt{Pruning}(X, \nu, \zeta), \quad \nu = 1, \ldots, f(X), \zeta = 1, \ldots, d(X)\Big\}, \tag{3}$$

where $f(X) := |X| - l(X)$ represents the number of functions in X.

In fact, Procedures 1-3 will behave as stochastic searches due to the random choices in Step 2 in each procedure. In other words, by calling any of these procedures several times for the same tree X and using the same parameters, one may get a different $\tilde{X}$ for each call. Fig. 4 shows examples of generating a new tree $\tilde{X}$ from a tree X by applying Procedures 1-3 with $\lambda = 2, \mu = 2, \nu = 3$ and $\zeta = 1$.

4 Proposed Algorithm

In this section a new hybrid evolutionary algorithm is presented as an improvement to the GP algorithm. The proposed algorithm hybridizes GP with the local search procedures introduced in the previous section to improve individuals generated by using GP operators. First, a new local search algorithm over a tree space called Local Search Programming algorithm is discussed. Next, the proposed Memetic Programming (MP) algorithm is described. The term "memetic" comes from memetic algorithms since the MP algorithm inherits the basic idea from memetic algorithms, while the term "programming" comes from GP since MP deals with computer programs represented by trees.

The first idea of the MP algorithm was proposed by the authors [25] in a short conference paper. In the present paper, some procedures of the proposed algorithm are enhanced and described in more detail, and more extensive numerical experiments are reported.

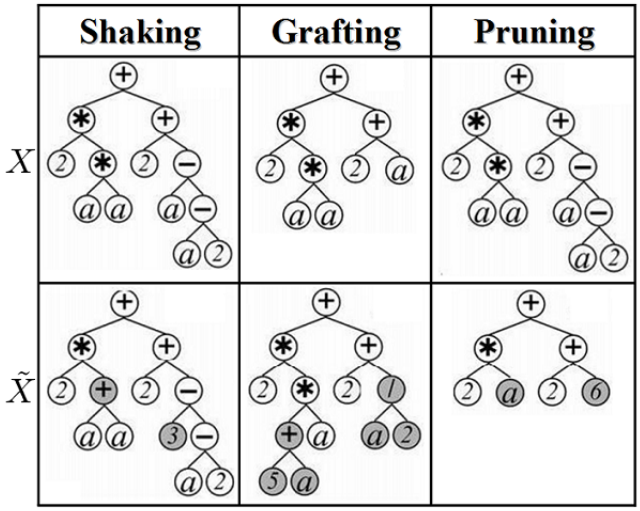

Figure 4: Generating new trees using shaking, grafting and pruning procedures.

4.1 Local Search Programming

A local search algorithm starts with an initial solution, and subsequently applies some operators to generate new solutions in a neighborhood of the current one. This process iterates until no better solution can be found in the neighborhood, and then the algorithm is terminated. In this subsection, a new local search algorithm over a tree space, called Local Search Programming (LSP) algorithm, is proposed to find the best program in the neighborhood of the current program X. The proposed LSP algorithm mainly uses the local search procedures described in Section 3. The details of the proposed algorithm are shown below.

Algorithm 4 *Local Search Programming*

1. ***Initialization****: Choose an initial program X, set $X_{best} := X$ and set the counter $k := 0$. Choose the values of* `nFails` *and* `nTrs`*.*

2. *While $k \leq$* `nFails`*, repeat Steps 2.1-2.5.*

   2.1 ***Static Structure Search***

   2.1.1 *Generate a set $\mathbb{Y}$=$\{Y_i |\ Y_i$=*`Shaking`*(X, i), $i = 1, \cdots,$*`nTrs`*$\}$.*

   2.1.2 *Let Y_{best} be the best program in the set $\mathbb{Y}$.*

   2.1.3 *If Y_{best} is better than X, then set $X := Y_{best}$ and go back to Step 2.1.1. Otherwise, set $k := k + 1$ and proceed to Step 2.2.*

   2.2 *If X is better than X_{best}, then set $X_{best} := X$.*

   2.3 *If $k >$* `nFails`*, then go to Step 3. Otherwise, proceed to Step 2.4.*

   2.4 ***Dynamic Structure Search***

   2.4.1 *Select* `Grafting` *or* `Pruning` *procedure randomly and denote the selected one by* `R`*.*

   2.4.2 *Generate a set $\mathbb{Z}$=$\{Z_i |\ Z_i$=*`R`*(X, i, ζ), $i = 1, \cdots,$*`nTrs`*$\}$.*

   2.4.3 *Replace X by the best program in the set $\mathbb{Z}$.*

   2.5 *If X is better than X_{best}, then set $X_{best} := X$. Go back to Step 2.1.*

3. ***Termination****: Stop and return with X_{best}, the best program found.*

In the initialization step, Step 1, the algorithm starts with a program X that is generated randomly or received from another algorithm. In addition, X_{best} and a counter k take their initial values. In fact, the counter k is used to count the number of non-improvements during the search process. In Step 1, the user must choose two positive integers `nFails` and `nTrs`. Specifically, `nFails` is the maximum number of non-improvements and `nTrs` represents the number of trial programs generated in the neighborhood

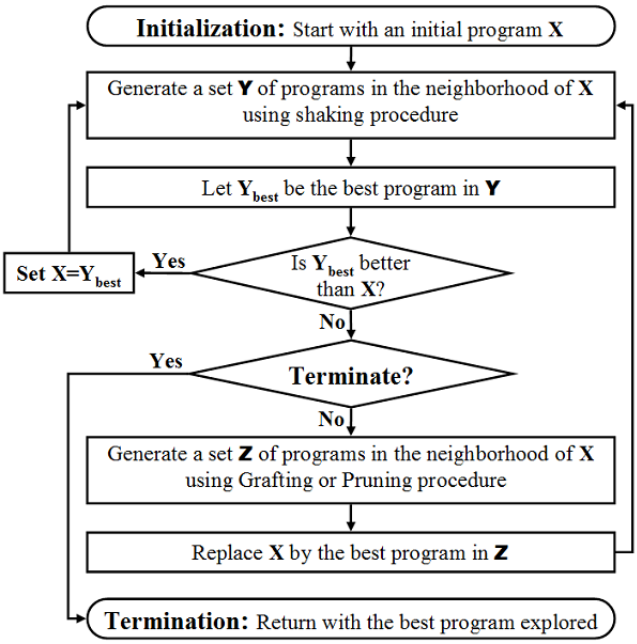

Figure 5: The flowchart of LSP

of the current program using static and dynamic structure searches. For the shaking procedure, we put $\texttt{nTrs} := \min(\texttt{nTrs}, |X|)$, and for the pruning procedure, we put $\texttt{nTrs} := \min(\texttt{nTrs}, d(X))$.

Step 2 consists of five substeps 2.1-2.5. In Step 2.1, an inner loop using the shaking procedure is iterated until it finds a better program near to X. In Step 2.1.1, the algorithm generates a set $\mathbb{Y}$ of trial programs using the shaking procedure. In Steps 2.1.2 and 2.1.3, if the best program, Y_{best}, in $\mathbb{Y}$ is better than X, then it replaces the current program X and the algorithm goes back to Step 2.1.1. Otherwise, the algorithm updates the counter k and proceeds to Step 2.2 to update X_{best} if better programs have already been explored.

In Step 2.3, when the algorithm reaches the maximum number of non-improvements `nFails`, it will stop and return with X_{best}. Otherwise, it proceeds to Step 2.4 to diversify the search process using a new program with different structure by applying either the grafting or the pruning procedure, which is chosen randomly. In Step 2.4.2, a set $\mathbb{Z}$ of trial programs is generated by the selected procedure. In Step 2.5, the algorithm replaces X by the best program in $\mathbb{Z}$ and goes back to Step 2.1. Finally, when the termination condition is satisfied, the algorithm stops at Step 3 and returns with the best program found. Fig. 5 shows the flowchart of the proposed LSP algorithm.

In Algorithm 4, the main loop starts in Step 2.1 by generating `nTrs` programs using the shaking procedure. If there is no improvement occurred in the current program, then the counter k is increased by 1, and the algorithm keeps working and generating new programs according to the following two processes. First, the algorithm proceeds to generate two sets of `nTrs` programs; the first `nTrs` programs are generated using the grafting or pruning procedures in Step 2.4, and the remaining `nTrs` programs are generated using the shaking procedure in Step 2.1. Second, the algorithm increases the counter k by one if no improvement occurred in the current program. These two processes are repeated as long as the value of the counter k does not exceed the maximum value `nFails`. Therefore, the number of fitness evaluations needed during a single run of the LSP algorithm varies depending on the improvement of the current program. If the algorithm completely fails to improve the current program, then the number of fitness evaluations consumed during the run of the algorithm is

$$minFit_{LSP} = (2\,\texttt{nFails} + 1)\,\texttt{nTrs}. \tag{4}$$

In fact, the value $minFit_{LSP}$ represents the minimum value of the number of fitness evaluations needed during the run of the LSP algorithm. However, if the algorithm succeeds to improve the current program, then the number of fitness evaluations needed during the run of the LSP algorithm will exceed the value $minFit_{LSP}$.

4.2 Memetic Programming

The main target of the MP algorithm is to improve the behavior of the GP algorithm by reducing the disruption effect of the crossover and mutation operators. Performing a local search for some promising programs during the search process can improve these programs. Moreover, if the search process succeeds to reach the area near an optimal solution, then a simple local search algorithm can capture that optimal solution easily. On the contrary, if the GP algorithm continues to be applied without the help of local search, there is a high probability of losing such promising solutions due to the disruption effect of crossover and mutation operators. In the MP algorithm, we use the LSP algorithm described in the previous subsection to improve some programs chosen from the current population based on their fitness values. The proposed MP algorithm is stated as follows:

Algorithm 5 *(MP Algorithm)*

1. ***Initialization.***
   - **1.1.** *Generate a random population of programs and evaluate the fitness value for each program. Set the initial values of controlling parameters needed in the search process.*
   - **1.2.** *Select some promising programs according to their fitness values.*
   - **1.3.** *Apply the LSP algorithm, Algorithm 4, to the selected programs.*
   - **1.4.** *Update the controlling parameters.*
2. ***Main Loop.*** *Repeat the following Steps 2.1-2.6 until a termination condition is satisfied. If a termination condition is satisfied, proceed to Step 3.*
   - **2.1.** *Select a set of parents from the current population, according to their fitness values.*
   - **2.2.** *Generate a new population using crossover and mutation operators, and evaluate the fitness value for each program in the new population.*
   - **2.3.** *Select a set of promising programs according to their fitness values.*
   - **2.4.** *Apply the LSP algorithm, Algorithm 4, to the selected programs.*
   - **2.5.** *Update the controlling parameters.*
   - **2.6.** *Return to Step 2.1 to breed a new population.*
3. *Stop with the best program found.*

The controlling parameters set in the Step 1.1 may store the number of generations that have been performed, the number of fitness evaluations that have been used, the fitness value of the best program found so far, and the number of consecutive non-improvements. These information can be used to terminate the algorithm in a suitable time. Therefore, the termination conditions in the MP algorithm may consist of one or more of the following events; reaching the highest fitness value, reaching the maximum number of fitness evaluations, or reaching the maximum number of consecutive non-improvements for the best program during the course of generating populations.

Step 2.2, in the main loop, generates a new population using the crossover and mutation operators as follows: First, pick up an operator randomly from the set of reproduction (copy), crossover and mutation operators. Second, pick up one or two program(s), depending on the selected operator, randomly from the pool set generated in Step 2.5. Third, get new offsprings by applying the selected operator for the selected program(s). Fourth, replace the parent(s) in the current population by the new offsprings. Repeat these steps until all programs in the population are modified.

It is worthwhile to note that the main loop in Step 2 of Algorithm 5 can be divided into two phases, the diversification phase in Steps 2.1-2.2 and the intensification phase in Steps 2.3-2.4. The diversification phase follows the GP strategy, where choosing a suitable selection strategy and using the crossover and mutation operations guarantee the diversity in the current population. On the other hand, the intensification phase, which uses the LSP algorithm to intensify promising programs obtained in the diversification phase, tries to catch the best solution. We note that, the MP algorithm at least behaves like the GP algorithm in case that no improvement occurs in the intensification phase. However, in this case the MP algorithm will be more costly than the GP algorithm, because of the computational effort spent in the intensification phase.

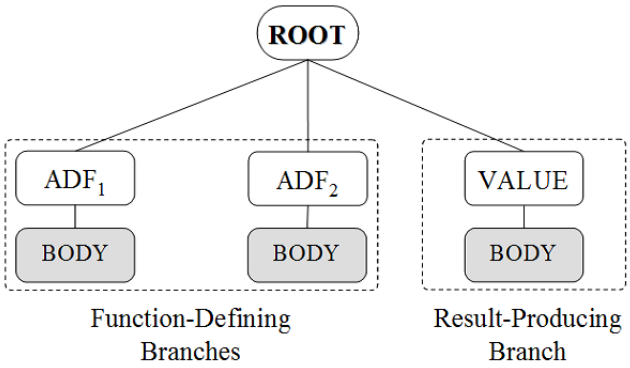

Figure 6: Example of representing a program in GP using ADF technique

4.3 Automatically Defined Functions

A program in GP is represented as a tree consisting of function and terminal nodes. The function set in GP usually contains primitive functions chosen based on the problem itself. For example, in the symbolic regression problem, one can use the set of binary functions $\{+, -, *, /\}$, while in the Boolean problems. one can use the set of Boolean functions {AND, OR, NAND, NOR}, and so on. Using these functions along with the set of terminals, the GP algorithm constructs and evolves programs through a set of steps.

One of important improvements of GP has been made through the use of automatically defined functions for reusing codes [18]. The ADFs are sub-trees that can be used as functions (called defun, subroutines, subprograms, or modules) of dummy arguments in the main tree of a program to exploit symmetries and modularities in the problem environments. In the standard GP algorithm with the ADF technique, each ADF is defined in a separate function-defining branch as a part of a program. In addition, for each ADF, the user must specify the number of dummy arguments and the function set which is allowed to contain other ADFs. The main program is defined in the result-producing branch that yields the fitness value of this program. For each program in the population, the result-producing branch is allowed to call functions from the function set that includes the original primitive functions as well as the ADFs defined for this program. In fact, the ADF technique has been successful in improving the performance of GP for a set of problems. Fig. 6 shows an example of the structure of a GP program that contains two function-defining branches (two ADFs) and a result-producing branch.

In this subsection, we focus on the effect of using the ADF technique within the proposed MP algorithm. In fact, the proposed MP algorithm uses the multigenic representation to express programs in the population, where each program is represented as a tree consisting of several genes [7]. The gene itself represents a sub-tree that consists of function and terminal nodes. Those genes in a program are linked together by means of a suitable linking function from the function set. More details about the representation of MP programs will be given in the next section.

Actually, the multigenic strategy enables MP to deal easily with ADFs for reusing codes. Since each program in MP can contain more than one gene, we can adopt one or more of these genes to work as ADF(s). In other words, each program in MP with ADFs contains two types of genes, ADF genes which represent function-defining branches, and regular genes which represent the result-producing branches. The result-producing branches are linked together by a suitable function to produce the final form of the program. Fig. 7 shows an example of the structure of a MP program using the ADF technique. In Section 5 we will show more details and explanations about implementing the ADF technique with MP.

5 Implementation of MP

In this subsection, we illustrate the details of some basic topics that are essential in the implementation of the MP algorithm, Algorithm 5, concerning representation of individuals, selection techniques, and so on.

5.1 Representation of Individuals

As described in the previous section, like GP, a solution generated by MP is called a program and it is represented as a tree consisting of one or more "gene(s)". Each gene represents a subtree consisting of

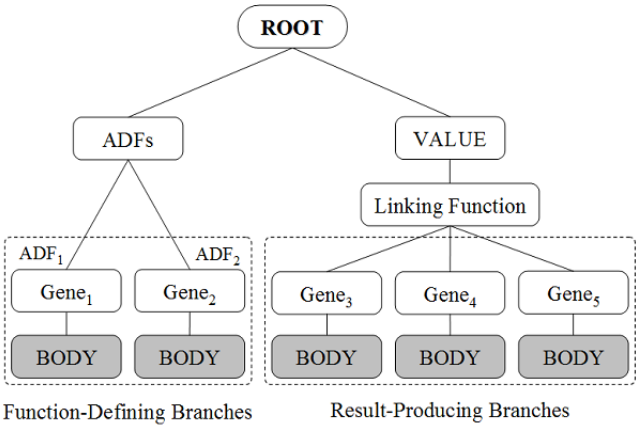

Figure 7: Example of representing a program in MP using ADF technique

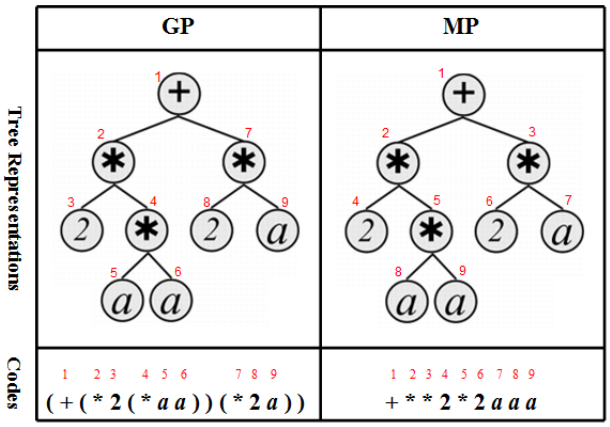

Figure 8: The tree structure and the coding representation of a solution in GP and MP

some external nodes called terminals and some internal nodes called functions. Genes inside a program are linked together by using a suitable linking function, e.g., "+" for the symbolic regression problem, to produce the final form of a solution. In the coding process, the tree structure of a program transforms into an executable code called genome. In MP, we code genes using a strategy that differs from the one used in the standard GP, where each gene is represented as a linear symbolic string composed of terminals and functions. Fig. 8 shows tree structures along with their executable codes for a gene in the GP and MP algorithms.

We use the following steps to generate each gene in the initial population: First, we generate a temporary random gene consisting of two parts, head (functions and terminals) and tail (terminals only). The length of the temporary gene is the head length `hLen` plus the tail length $t = h(n-1)+1$, where n is the maximum number of arguments of the function. Second, we adjust the final form of the gene by deleting unnecessary elements, based on the functions and terminals that are generated randomly within the gene. In this way, we guarantee the syntactically correct structure for genes in the initial population. For more details and examples, see [10].

Once the initial population is generated, it will be evolved and improved using the MP operations; i.e., crossover, mutation, shaking, grafting and pruning. For each problem to solve, the sets of functions and terminals, the number of genes, the head length `hLen` for the initial population, and the fitness function must be determined before calling the algorithm. In fact, adapting each program to contain more than one genes increases the probability of finding suitable solutions, and enables the algorithm to deal with more complex problems [7].

5.2 Selection Techniques

The selection technique is considered a key issue in the MP algorithm as well as the GP algorithm, since it affects other steps in a direct way. Moreover, it implicitly affects the diversity of the population since

the best program(s) in the old population can propagate excessively and produce a lot of offsprings in the new population, which reduces the diversity in the population [3]. In the literature, e.g., [3, 4, 13], several selection techniques have been introduced and discussed extensively, all of which make use of the fitness values of the current population.

The proposed MP algorithm, Algorithm 5, mainly contains two selection steps: First, Step 2.1 selects a pool of programs to breed a new population using the crossover and mutation operators. Second, Step 2.3 (as well as Step 1.2) selects a set of promising programs to improve them by the LSP algorithm. In our earlier paper [25], we used the roulette wheel selection in all selection steps in the old version of the MP algorithm.

In the present paper, we use three different selection techniques that have shown promise in our extensive numerical experiments. In Step 2.1, we use the tournament selection [13] of size 4 as the default selection strategy, and the roulette wheel selection will be used for some special experiments, as shown later. In Steps 1.2 and 2.3, we always select the best `nLs` programs from the current population, where `nLs` is a positive integer less than or equal to the population size.

5.3 Crossover and Mutation

The crossover and mutation are the basic operators in EAs. Since programs are represented by trees, the crossover is applied by choosing two programs (trees) randomly, choosing a node randomly from each tree, and exchanging the two subtrees rooted at these two nodes to get offsprings. On the other hand, the mutation operator is applied for one program (tree) chosen randomly from the pool set. Then, one can get a new offspring by choosing a node randomly and exchanging the subtree rooted at this node by a new one that is generated randomly.

The crossover and mutation operators are summarized in the following two procedures.

Procedure 6 $[Y_1, Y_2] = \texttt{Crossover}(X_1, X_2)$

Step 1. *Choose a node n_1 from X_1 randomly.*

Step 2. *Choose a node n_2 from X_2 randomly.*

Step 3. *Swap the two subtrees rooted at n_1 and n_2, and call the new trees Y_1 and Y_2.*

Procedure 7 $[Y] = \texttt{Mutation}(X)$

Step 1. *Choose a node n_1 from X randomly.*

Step 2. *Generate a new subtree $\hat{X}$ randomly.*

Step 3. *Replace the subtree rooted at n_1 by $\hat{X}$ and call the new tree Y.*

In Step 2.2 of Algorithm 5, the MP algorithm starts with a new empty population and repeats the following steps until the new population becomes full: First, it picks up an operator randomly from the set of GP operators, i.e., reproduction (copy), crossover and mutation operators, based on a predetermined probability value. Second, the algorithm picks up one or two program(s), based on the selected operator, randomly from the pool set generated in Step 2.5. Third, the algorithm generates new offsprings by applying the selected operator to the selected program(s). Finally, new offsprings are added to the new population. Once the new population becomes full, it will replace the old one.

5.4 Set of Parameters

The proposed MP algorithm makes use of a set of parameters that can be classified into two types of parameters; representation parameters and search parameters. We list these parameters in the following:

- Representation Parameters
  - `hLen`: The head length for every gene generated randomly in the initial program.
  - `MaxLen`: The maximum length, i.e., the number of nodes, of a gene allowed in the search process.
  - `nGenes`: The number of genes in each program.
  - ζ: The depth of branches added to and removed from a tree X in the grafting search (Procedure 2) and the pruning search (Procedure 3), respectively. Throughout this paper, we use the branch depth $\zeta = 1$.

- `LnkFun`: The function used to link genes in each program.

- Search Parameters
  - `nPop`: The population size.
  - `nGnrs`: The maximum number of generations.
  - `nLs`: The number of programs selected to apply local search procedures at the intensification phase in Steps 2.3-2.4 in Algorithm 5.
  - `nTrs`: The number of trial programs generated in the neighborhood of the selected program using a local search procedure, i.e., shaking, grafting or pruning.
  - `nFails`: The maximum number of non-improvements for each call of the LSP algorithm in Step 2.4 of the MP algorithm.

If no improvements occurs in the intensification phase in Steps 2.3-2.4 of Algorithm 5, then the number of fitness evaluations consumed in this phase is $minFit_{IntP} = \texttt{nLs}\,(2\,\texttt{nFails} + 1)\,\texttt{nTrs}$. This is because Algorithm 5 calls the LSP algorithm `nLs` times during the intensification phase in Steps 2.3-2.4, while the number of fitness evaluations during one call of the LSP algorithm is $(2\,\texttt{nFails} + 1)\,\texttt{nTrs}$, equation (4). If the values of `nLs`, `nFails` and `nTrs` are large, the MP algorithm will need a lot of fitness evaluations in the intensification phase in Steps 2.3-2.4.

Certainly, if the LSP algorithm succeeds to improve the chosen programs in Steps 2.3 of Algorithm 5, then we do not mind the increase of the number of fitness evaluations in the intensification phase, since it increases the probability of finding an optimal solution. On the other hand, if the LSP algorithm fails to improve the chosen programs in Steps 2.3 of Algorithm 5, then the MP algorithm will not gain benefits by using a large number of fitness evaluations. For that reason, we let $minFit_{IntP}$ be the maximum number of fitness evaluations in the intensification phase. Specifically, we set $minFit_{IntP} = \alpha\,\texttt{nPop}$, where α is a positive constant determined before calling the algorithm. Then the values of `nLs`, `nTrs` and `nFails` must be chosen to satisfy the equation

$$\texttt{nLs}\,(2\,\texttt{nFails} + 1)\,\texttt{nTrs} = \alpha\,\texttt{nPop}. \tag{5}$$

In practice, we first choose the values of `nTrs` and `nFails`, and then determine the value of `nLs` (from (5)) by

$$\texttt{nLs} = \left\lceil \frac{\alpha\,\texttt{nPop}}{(2\,\texttt{nFails} + 1)\,\texttt{nTrs}} \right\rceil, \tag{6}$$

where $\lceil x \rceil$ means the smallest integer greater than or equal to x. In particular, if $\alpha = 0$, then no program will be processed by the LSP algorithm.

5.5 Building and Evolving ADFs in MP

Actually, based on the individual representation in MP, we can easily extend the MP algorithm to build and reuse ADFs. For each program in MP, we can exploit one or more genes to work as ADF(s), which will be created and evolved during the run of the algorithm. In addition, these ADFs will be added automatically to the function set of other genes, the result-producing branches, in the same program that contains these ADFs. In this case, the fitness value for a program in MP with ADFs will be computed from the result-producing branches by linking all of them using a suitable function, e.g., a primitive function or one of the ADFs themselves.

Suppose that $\{\text{ADF}_1, \text{ADF}_2, \cdots, \text{ADF}_n\}$ is the set of ADFs used for the problem at hand, where each ADF has its own set of dummy arguments. As in GP with ADFs, the function set used to build the ADFs is the original function set $\mathfrak{F}$, and it is possible to include one or more function(s) from the set $\{\text{ADF}_i | i = 1, \cdots, j-1\}$ in the body of ADF_j, where $j = 2, \cdots, n$. In addition, the function set for result-producing branches (regular genes) is $\mathfrak{F} \cup \{\text{ADF}_1, \cdots, \text{ADF}_n\}$, while the terminal set is the original terminal set $\mathfrak{T}$. Fig. 7 shows an example of the overall structure of a program in the MP algorithm using two ADFs.

In fact, ADFs can increase the size of genes dramatically, especially if there exist several ADFs in the body of genes. In other words, if one replaces each ADF in the body of a regular gene by its equivalent subtree, then the number of nodes of this regular gene can increase rapidly according to the number of ADFs in its main tree. Fig. 10 shows the actual trees of genes 3 and 4 in Fig. 9 by replacing ADF_1 and

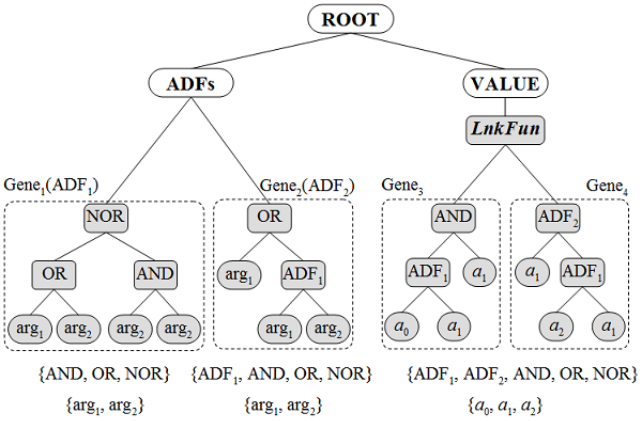

Figure 9: Example of representing a program in MP using ADFs technique. Last two lines represent the function and terminal sets for each gene.

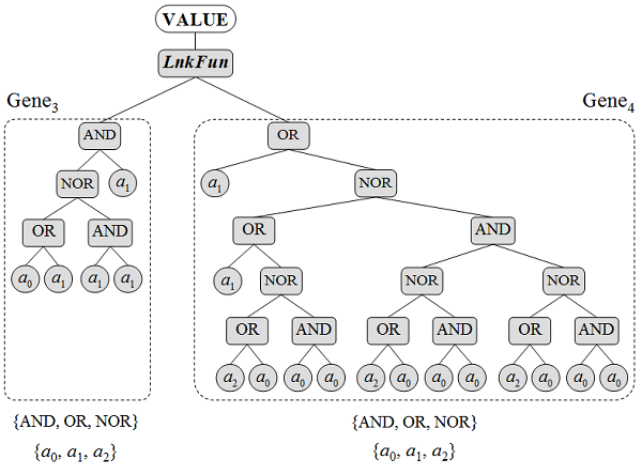

Figure 10: The actual tree representations of genes 3 and 4 in Fig. 9

ADF_2 by their equivalent subtrees. In Fig 9, one can notice that the number of nodes of gene 3 is 5, and the number of nodes of gene 4 is 5. However, by replacing each ADF function by its equivalent subtree as in Fig. 10, we can see that the actual number of nodes for genes 3 and 4 are 9 and 27, respectively. In addition, in case of using a function from the set {AND, OR, NOR} to link genes 3 and 4 in Fig. 10, the total number of nodes in the program will be 37. On the other hand, if the `LnkFun` in Fig. 9 is ADF_1 or ADF_2, then the total number of nodes in the program in Fig. 10 will be 93 or 103, respectively. Therefore, the values of `hLen` and `MaxLen` parameters must be chosen small enough in case of using the ADF technique. In this paper we set `hLen` $:= (h_1, h_2)$ and `MaxLen` $:= (m_1, m_2)$, which means `hLen` $= h_1$ and `MaxLen` $= m_1$ for ADFs genes, and `hLen` $= h_2$ and `MaxLen` $= m_2$ for regular genes.

It is important to note that special care is needed to use the crossover operator in MP with ADFs, since each program contains two different types of genes, i.e., ADF genes and regular genes. Specifically, we choose two parents randomly from the current population and choose a gene randomly from the first one. If the chosen gene is an ADF gene, then we have to select the corresponding gene in the second parent. Otherwise, we can choose any gene randomly from the set of regular genes in the second parent, and use the crossover operator normally for the selected genes.

6 Numerical Experiments

This section discusses the performance of the MP algorithm on some well-known benchmark problems. First, we introduce the benchmark problems under consideration in Subsection 6.1. Then, we examine the performance of the MP algorithm through extensive experiments. In these experiments, we focus on

the performance of the proposed MP algorithm under different environments. Specifically, we study the performance of MP under different selection strategies and different values for the `nLs`, `nTrs` and `nFails` parameters. These parameter settings are performed for the MP algorithm with and without the use of the ADF technique.

Finally, we make a set of comparisons, between the MP algorithm and various recent versions of the GP algorithm, to show the efficiency of the proposed MP algorithm. In fact, the proposed MP algorithm shows promising performance compared to recent GP algorithms as we will see later. In all the experiments, we terminate the algorithm as soon as an optimal solution with the highest fitness value is found, or the maximum number of fitness evaluations is reached.

The computational effort (CE) has been introduced by Koza [17] to measure the computational costs required for GP to solve a problem, and its value is based on some data collected from a set of independent runs. The formula of CE is given as follows:

$$\begin{aligned} \text{CE} &= \min_i I(\texttt{nPop}, i, z), \\ I(\texttt{nPop}, i, z) &= \texttt{nPop} * R(z) * i, \\ R(z) &= \left\lceil \frac{\log(1-z)}{\log(1-P(\texttt{nPop}, i))} \right\rceil, \\ P(\texttt{nPop}, i) &= \frac{N_s(i)}{N_{all}}, \end{aligned} \tag{7}$$

where z is positive number less than 1, $N_s(i)$ is the number of successful independent runs up to generation i, and N_{all} is the total number of runs. As in Koza [17], $P(\texttt{nPop}, i)$ represents the cumulative probability of success up to generation i, $R(z)$ represents the number of independent runs required to produce a solution up to generation i with probability z, and $I(\texttt{nPop}, i, z)$ is the number of programs that must be processed to get a satisfactory solution, with probability z, using population size `nPop` up to generation i. All results in this paper are computed using $z = 0.99$ as in Koza [17].

In the MP algorithm, the number of fitness evaluations per each generation varies according to the performance of the LSP algorithm. So we will slightly modify the definition of CE for the MP algorithm. Specifically, we define the computational effort CE for the MP algorithm by (7), where $N_s(i)$ is the number of successful independent runs using up to $i * \texttt{nPop}$ fitness evaluations in the MP algorithm, which is equal to the number of successful independent runs up to generation i in the GP algorithm with fixed population size `nPop`.

6.1 Test Problems

We use three different benchmark problems collected from the literature. These benchmark problems are the symbolic regression problem, the Boolean N-bit even-parity problem and the Boolean 6-Bit multiplexer problem.

6.1.1 Symbolic Regression (SR) Problem

The symbolic regression problem is the problem of fitting a dataset $\{(x_{j1}, \cdots, x_{jm}, f_j)\}_{j=1}^{n}$, by a suitable mathematical formula g such that the absolute error

$$\sum_{j=1}^{n} |f_j - g(x_{j1}, \cdots, x_{jm})| \tag{8}$$

is minimized.

Here, we study the performance of the MP algorithm for the multivariate polynomial (POLY-4) $f(x_1, \cdots, x_4) = x_1x_2 + x_3x_4 + x_1x_4$, see [33]. A dataset consisting of 50 fitness cases has been generated randomly with $x_i \in [-1, +1]$, $i = 1, 2, 3, 4$. The target in the problem (which will be referred to as the SR-POLY-4 problem in the rest of the paper) is to detect a function $g(x_1, x_2, x_3, x_4)$ that approximates the original polynomial POLY-4, with the minimum error, by using the dataset. The fitness value for a program is calculated as the sum, with the sign reversed, of the absolute errors between the output produced by a program and the desired output for each of the fitness cases. Therefore, the maximum fitness value for this problem is 0.

6.1.2 The Boolean N-Bit Even-Parity Problem

The Boolean N-bit even-parity (N-BEP) function is a function of N-bit arguments, namely $a_0, a_1, \cdots, a_{N-1}$. It returns 1 (True) if the arguments contain an even number of 1's and it returns 0 (False) otherwise. All 2^N combinations of the arguments, along with the associated correct values of the N-BEP function, are considered to be the fitness cases. The fitness value for a program is the number of fitness cases where the Boolean value returned by the program for a given combination of the arguments is the correct Boolean value. Therefore, the maximum fitness value for the N-BEP problem is 2^N.

6.1.3 The Boolean 6-Bit Multiplexer Problem

An input to the Boolean N-bit multiplexer (N-BM) function consists of k "address" bits a_i and 2^k "data" bits d_i as a string of length $N = k + 2^k$ of the form a_0, a_1, $\cdots$, a_{k-1}, d_0, d_1, $\cdots$, d_{2^k-1}. The value of the N-BM function is the value (0 or 1) of the particular data bit that is singled out by the k address bits of the multiplexer. All 2^N combinations of the arguments, along with the associated correct values of the N-BM function, are considered the fitness cases. The fitness value for a program is the number of fitness cases where the Boolean value returned by the program for a given combination of the arguments is the correct Boolean value. Therefore, the maximum fitness value for the N-BM problem is 2^N.

6.1.4 Settings of the Test Problems

In the rest of this section, we will use the following settings, unless otherwise stated, for our test problems:

- For the SR-POLY-4 problem, the set of independent variables $\{x_1, x_2, x_3, x_4\}$ is regarded as the terminal set, and the set of functions $\{+, -, *, \%\}$ is regarded as the function set, where $x\%y := x$ if $y = 0$; $x\%y := x/y$ otherwise.
- For the N-BEP problem, the set of arguments $\{a_0, a_1, \cdots, a_{N-1}\}$ is used as the terminal set, and the set of Boolean functions {AND, OR, NAND, NOR} is used as the function set. In fact, the GP research community considers evolving the N-BEP function by using those Boolean functions as a good benchmark problem for testing the efficiency of new GP techniques [17, 36].
- For the 6-BM problem, the set of arguments $\{a_0, a_1, d_0, d_1, d_2, d_3\}$ is used as the terminal set, and the set of Boolean functions {AND, OR, NOT, IF} is used as the function set, where IF(x, y, z) returns y if x is true, and it returns z otherwise.

6.2 Performance Analysis

In this subsection, we study the setting of MP parameters and components, and its effect on the performance of the MP algorithm. Specifically, we mainly focus on the selection strategy and the parameter setting associated with the LSP algorithm, i.e., `nLs`, `nTrs` and `nFails`. A set of different values for each of these parameters is chosen, and for each value, 100 independent runs are performed to compute the average number of evaluations (AV), the computational effort (CE) and the rate of success (R). In this subsection, the values of the parameters under consideration will be specified for each set of experiments. The values of other parameters are set to their standard values shown in Table 1, which are determined from our pilot experiments or the common setting in the literature.

6.2.1 Performance under Different Selection Strategies

In this set of experiments, we use the MP algorithm with two different selection strategies; the tournament selection and the roulette wheel selection. For each selection strategy, we performed 100 independent runs to compute the average number of evaluations (AV), the computational effort (CE) and the rate of success (R).

The performance of the MP algorithm with these two selection strategies is shown in Table 2. From these results, we can see that the tournament selection gives better results than the roulette wheel selection. When the LSP algorithm is not employed, i.e., $\alpha = 0$, changing the selection strategy from the tournament to the roulette wheel caused a collapse in the rate of success, and increased the computational efforts unreasonably, in particular, for the 6-BM problem. On the other hand, by using the LSP algorithm, the effect of changing the selection strategy is not high compared to the previous case. Therefore, one can conclude that the LSP algorithm increases the stability of the MP algorithm with different selection

Table 1: The standard values of the MP parameters for the benchmark problems.

	MP without ADFs				MP with ADFs	
Parameter	SR-POLY-4	3-BEP	N-BEP$_{16BF}$	6-BM	SR-POLY-4	N-BEP
`hLen`	3	3	1	3	(3,1)	(3,1)
`MaxLen`	30	30	15	30	(7,3	(7,3)
`nGenes`	3	2	N	3	4	$N+1$
`nPop`	50	500	50	500	50	$250(N-1)$
`nGnrs`	100	100	$\rightarrow \infty$	100	100	$\rightarrow \infty$
`LnkFun`	+	AND	XOR	IF	+	ADF_1
Equation (6): $\alpha := 1/2$, `nTrs` := 2 and `nFails` := 1						
Crossover probability:= 0.9						
Mutation probability:= 0.05						
Reproduction (Copy) probability:= 0.05						
Selection strategy: Tournament selection of size 4						

Table 2: Comparison of selection strategies.

Problem	α	Selec.	AV	CE	R%
SR-POLY-4	0	Tour.	2,155	7,000	72
		Roul.	3,639	26,250	44
	$\frac{1}{2}$	Tour.	1,165	4,400	96
		Roul.	1,804	7,400	91
3-BEP	0	Tour.	19,225	60,000	80
		Roul.	48,030	1,550,000	14
	$\frac{1}{2}$	Tour.	15,568	54,000	90
		Roul.	37,403	245,000	62
6-BM	0	Tour.	9,075	25,000	100
		Roul.	49,965	21,343,500	1
	$\frac{1}{2}$	Tour.	9,019	21,500	100
		Roul.	41,587	350,000	50

strategies. This impressive property can save computations for learning experiments to detect desirable environments for the problem under consideration.

6.2.2 Parameter `nLs`

Here, we focus on the parameter `nLs`, the number of programs in the population to which local search is applied, and its effect on the performance of the MP algorithm. Different values for the constant α in (6) are chosen, and 100 independent runs of the MP algorithm are performed for each value. Comparisons, in terms of the average number of evaluations, the computational efforts and the rate of success, are made for these different settings of the MP algorithm. Mainly, we focus here on four cases:

1. $\alpha = 0$, which means there is no use for the LSP algorithm.
2. $\alpha = 1/2$, which means the intensification phase will cost at least `nPop`/2 fitness evaluations for each generation of Algorithm 5.
3. $\alpha = 1$, which means the intensification phase will cost at least `nPop` fitness evaluations for each generation of Algorithm 5.
4. Applying the LSP algorithm for all programs in the current population, i.e., `nLs` = `nPop` which implies $\alpha = 5$ by (5), since `nFail` = 2 and `nTrs` = 1 in Table 1.

From Table 3, we see that the LSP algorithm has great influence on the MP algorithm. It improves performance of the MP algorithm in both the number of evaluations and the rate of success. In addition, applying the LSP algorithm for all programs in the current population is costly in terms of AV and CE, but it improved the rate of success. Throughout this section, we use $\alpha = \frac{1}{2}$ as recommended from the results in Table 3.

Table 3: Comparisons in terms of parameter `nLs`

Problem	α	`nLs`	AV	CE	R%
SR-POLY-4	0	0	2,155	7,000	72
	$\frac{1}{2}$	$\lceil$`nPop`$/10\rceil$	1,165	4,400	96
	1	$\lceil$`nPop`$/5\rceil$	1,229	4,500	99
	5	`nPop`	2,290	5,000	99
3-BEP	0	0	19,225	60,000	80
	$\frac{1}{2}$	$\lceil$`nPop`$/10\rceil$	15,568	54,000	90
	1	$\lceil$`nPop`$/5\rceil$	16,894	62,000	90
	5	`nPop`	31,417	97,000	92
6-BM	0	0	9,075	25,000	100
	$\frac{1}{2}$	$\lceil$`nPop`$/10\rceil$	9,019	21,500	100
	1	$\lceil$`nPop`$/5\rceil$	10,118	29,000	100
	5	`nPop`	26,653	73,000	100

Table 4: Results of the MP algorithm in terms of parameters `nTrs` and `nFails`

Parameters		SR-POLY-4		3-BEP		6-BM	
`nTrs`	`nFails`	CE	R%	CE	R%	CE	R%
`Without LSP`		7,000	72	60,000	80	25,000	100
1	1	3,750	99	55,500	92	18,500	100
1	2	5,200	92	54,000	89	25,000	100
1	3	5,400	92	70,000	83	29,000	99
2	1	4,400	96	54,000	90	21,500	100
2	2	4,800	96	48,000	88	28,500	100
2	3	5,250	93	52,000	89	22,500	99
3	1	5,600	90	60,000	86	19,500	100
3	2	4,600	96	60,000	83	31,000	98
3	3	5,400	92	52,500	92	28,000	99
4	1	5,250	97	52,000	87	31,000	99
4	2	6,000	93	51,000	92	33,000	98
4	3	4,800	94	64,000	88	32,000	100
5	1	5,400	89	54,000	86	30,000	100
5	2	5,750	91	58,000	90	29,000	99
5	3	6,900	94	62,500	87	26,500	99

6.2.3 Parameters `nTrs` and `nFails`

We conducted experiments in which SR-POLY-4, 3-BEP and 6-BM problems were solved using the MP algorithm without ADFs. The main parameters we focus here are the parameters `nTrs` and `nFails`. The chosen values for these parameters are `nTrs` $= 1, 2, 3, 4, 5$ and `nFails` $= 1, 2, 3$. For each combination of these parameter values, we performed 100 independent runs of the MP algorithm for each problem. The results of these experiments are shown in Table 4. It is worthwhile to note that for most values of `nTrs` and `nFails`, the use of local search helps to improve the performance of the MP algorithm; see the row labeled '`Without LSP`' in Table 4.

6.2.4 Parameter Setting for MP with ADFs

The previous experiments, which mainly focus on the effects of the parameters `nTrs` and `nFails`, indicate that our results are promising compared to recent algorithm as we will see later. Nevertheless, we still have a chance to improve further these results especially for the N-BEP problem by the use of the ADF technique. The GP research community usually uses the ADF technique to exploit the modularity in a problem, especially the N-BEP problem [18, 36]. Using GP with the ADF technique, Koza [18] has succeeded to get the exact solutions for the N-BEP problem with $N = 3, \cdots, 11$, with less CE compared to the standard GP algorithm without the ADF technique.

What we want to show here is that the MP algorithm not only can improve the results of the standard

Table 5: Results of MP with the ADF technique in terms of parameters `nTrs` and `nFails`.

Parameters		SR-POLY-4		3-BEP	
`nTrs`	`nFails`	CE	R%	CE	R%
`Without LSP`		2,700	82	32,500	85
1	1	2,500	87	17,000	98
1	2	2,400	97	15,500	99
1	3	2,600	95	22,000	99
2	1	2,600	89	19,000	99
2	2	2,400	98	21,000	99
2	3	2,400	100	24,000	100
3	1	2,600	99	20,000	99
3	2	2,400	100	21,000	100
3	3	2,400	99	20,000	100
4	1	3,000	99	22,500	99
4	2	3,000	100	18,000	98
4	3	2,800	99	20,000	100
5	1	2,250	98	20,000	99
5	2	3,150	96	21,000	98
5	3	2,850	99	18,000	99

GP algorithm by using the new set of operations introduced in Section 3, but also can deal with the ADF technique to represent programs more professionally to exploit the modularity in a problem. In this set of experiments specifically, we use the MP algorithm with the ADF technique to solve the SR-POLY-4 and N-BEP problems.

For the SR-POLY-4 problem, we use the same terminal and function sets as in the previous experiments. In addition, for each program in the population, an additional ADF function (gene) of two dummy arguments is defined, evolved and included automatically in the function set for that program. The terminal and function sets for that ADF gene (called ADF_1) are $\{\text{arg}_0, \text{arg}_1\}$ and {+, -, *, %}, respectively, while the terminal and function sets for regular genes are $\{x_1, x_2, x_3, x_4\}$ and $\{\text{ADF}_1$, +, -, *, %}, respectively.

For the N-BEP problem, we use two additional ADFs, ADF_1 and ADF_2, each of which has two dummy arguments, arg_0 and arg_1. For each program in the population, ADF_1 and ADF_2 are defined, evolved and included automatically in the function set for that program. The terminal sets for ADF_1, ADF_2 and regular genes are $\{\text{arg}_0, \text{arg}_1\}$, $\{\text{arg}_0, \text{arg}_1\}$ and $\{a_0, \cdots, a_{N-1}\}$, respectively. The function sets for them are {AND, OR, NAND, NOR} $\{\text{ADF}_1$, AND, OR, NAND, NOR} and $\{\text{ADF}_1$, ADF_2, AND, OR, NAND, NOR}, respectively.

In the current set of experiments, we still focus on detecting the best values of `nTrs` and `nFails` parameters. Several values are chosen such as `nTrs` $= 1, 2, 3, 4, 5$ and `nFails` $= 1, 2, 3$, and for each combination of these values, we performed 100 independent runs for each problem using the MP algorithm with ADFs. Other MP parameters are shown in Table 1, while the results of this experiment is shown in Table 5. It is clear from these results that the MP algorithm significantly reduces the computational effort and improves the rate of success by means of ADFs.

Moreover, we performed a set of experiments to solve the 6-BM problem using the MP algorithm with the ADF technique. Unfortunately, we could not improve the performance of the MP algorithm by using one ADF or two ADFs. This show that the ADF technique is useful for problems which have special characteristics, e.g., similarity and modularity. Nevertheless, our results for the 6-BM problem are still good and competitive, without the use of the ADF technique, as we will see in the next subsection.

6.3 MP vs GP

In this subsection, we study the performance of the MP algorithm, with and without ADFs, compared to contemporary versions of the GP algorithm.

6.3.1 The SR-POLY-4 Problem

Poli and Langdon [33] conducted a lot of numerical experiments for the SR-POLY-4 problem using the backward-chaining GP (BC-GP) algorithm, and made a comparison between the BC-GP algorithm and

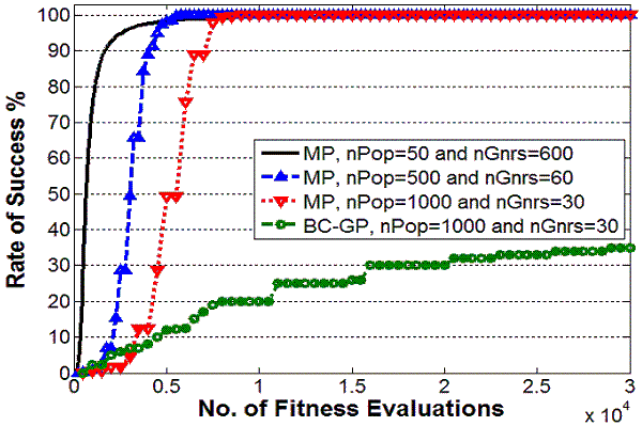

Figure 11: Comparison between the MP algorithm and the backward-chaining GP algorithm for the SR-POLY-4 problem in terms of the rate of success

the standard GP algorithm. They performed 5000 independent runs to solve the SR-POLY-4 problem using the BC-GP and standard GP algorithms without the use of ADFs, where `nPop` $= 1,000$ and `nGnrs` $= 30$. The results of their experiments show a good performance of the BC-GP algorithm compared to the standard GP algorithm.

Here, we compare the results of the MP algorithm with those of the BC-GP algorithm in terms of fitness evaluations and the rate of success. We implement the MP algorithm (without ADFs) for the SR-POLY-4 problem three times using different population sizes. For each implementation, we performed 5000 independent runs for the MP algorithm using `nTrs` $= 1$ and `nFails` $= 1$ (as recommended from the results in Table 4), while the other parameters are set as shown in Table 1. The results of this experiment are summarized in Fig. 11, where the results of the BC-GP algorithm have been taken from Fig. 10 in the original reference [33].

Fig. 11 shows that the MP algorithm significantly outperforms the BC-GP algorithm. The MP algorithm gets the 100% success for all 5000 runs using 10,000 fitness evaluations at most, even by using a very small population size `nPop` $= 50$. On the other hand, the BC-GP hardly succeeds to find the exact solution for 35% of total runs by using 30,000 fitness evaluations. Poli and Langdon in the same paper [33] repeated the same experiment using a large population size, `nPop` $= 10,000$ and `nGnrs` $= 30$. Then, they were able to improve their results and the rate of success reached approximately 98% after 300,000 fitness evaluations. Nevertheless, one can see that the MP algorithm is faster than the BC-GP algorithm by more than 30 times for the SR-POLY-4 problem.

Although, for the SR-POLY-4 problem, the results of the MP algorithm with ADFs are better than those of the MP algorithm without ADFs, we did not try to compare the MP algorithm with ADFs and the BC-GP algorithm. This is because the BC-GP algorithm did not use the idea of ADFs, so it would not be a fair comparison if we used ADFs in the MP algorithm.

6.3.2 The N-BEP Problem

Walker and Miller [36] conducted a lot of numerical experiments to show the performance of the Cartesian GP (CGP) algorithm and the Embedded CGP (ECGP) algorithm. The ECGP algorithm generalizes the CGP algorithm by utilizing the automatic module acquisition technique to automatically build and evolve modules. Walker and Miller [36] reported good results for several test problems, compared to the standard GP algorithm and several contemporary algorithms. Here, we are interested in comparing their results for the Boolean even parity problems with the results of the MP algorithm for the same problems.

Walker and Miller [36] used the Embedded Cartesian GP (ECGP) algorithm to solve the N-BEP problem using the set of Boolean functions {AND, OR, NAND, NOR} as the function set, and the set of arguments $\{a_0, \cdots, a_{N-1}\}$ as the terminal set. For each of the N-BEP problems with $N = 3, \cdots, 8$, they performed 50 independent runs with `nGnrs` $\rightarrow \infty$, i.e., for each run the algorithm works until the exact solution was found.

Walker and Miller [36] measure the performance of the ECGP algorithm in terms of CE and other statistical measures. Undoubtedly, the results of Walker and Miller [36] for the N-BEP problem are

Table 6: The CE for the standard GP, ECGP and MP algorithms for the N-BEP problem

N	GP with ADFs	ECGP with modules	MP with ADFs
3	64,000	37,446	17,000
4	176,000	201,602	22,500
5	464,000	512,002	42,000
6	1,344,000	978,882	77,500
7	-	1,923,842	108,000
8	-	4,032,002	178,500

Table 7: The ME, MAD and IQR for the ECGP and MP algorithms for the N-BEP problem

	ECGP with modules			MP with ADFs		
N	ME	MAD	IQR	ME	MAD	IQR
3	5,931	3,804	10,372	4,795	1,268	3,030
4	37,961	21,124	49,552	6,611	2,460	4,919
5	108,797	45,402	98,940	13,597	3,034	6,041
6	227,891	85,794	190,456	23,574	6,631	13,261
7	472,227	312,716	603,643	37,012	11,341	20,493
8	745,549	500,924	1,108,934	57,603	18,095	34,437

significant, as they showed through several comparisons with other extensions of the GP algorithms. Our target here is to make a comparison between the ECGP algorithm (with modules) and the MP algorithm with ADFs for the N-BEP problem with $N = 3, \cdots, 8$.

For the N-BEP problem, We have used the MP algorithm with two ADFs, ADF_1 and ADF_2, each of which has two dummy arguments, arg_0 and arg_1. For each program in the population, ADF_1 and ADF_2 are defined, evolved and included automatically to the function set for that program. The terminal sets for ADF_1, ADF_2 and regular genes are $\{\text{arg}_0, \text{arg}_1\}$, $\{\text{arg}_0, \text{arg}_1\}$ and $\{a_0, a_1, \cdots, a_{N-1}\}$, respectively, while the function sets for them are {AND, OR, NAND, NOR} $\{\text{ADF}_1$, AND, OR, NAND, NOR$\}$ and $\{\text{ADF}_1$, ADF_2, AND, OR, NAND, NOR$\}$, respectively.

We performed 50 independent runs for the N-BEP problem with $N = 3, \cdots, 8$, using `nTrs` $= 1$ and `nFails` $= 2$, and the values of the remaining parameters are shown in Table 1. A comparison, in terms of CE, between the ECGP algorithm (with modules) and the MP algorithm with ADFs are presented in Table 6. In addition, Table 7 shows additional comparisons between ECGP and MP, in terms of median (ME) number of evaluations, median absolute deviation (MAD), and interquartile range (IQR). All the results for the standard GP algorithm and the ECGP algorithm shown in Tables 6 and 7 have been taken from [36]. It is clear from Tables 6 and 7 that the MP algorithm with ADFs outperforms the standard GP algorithm and the ECGP algorithm.

The results for the N-BEP problem encouraged us to tackle higher order even parity problems. In fact, we have succeeded to find the exact solution for the N-BEP problem with $N = 3, \cdots, 15$ using a reasonable number of fitness evaluations. We believe that we still have a chance to solve the N-BEP problem with $N > 15$ using the MP algorithm with ADFs. Because of the exponential increase of fitness cases, 2^N, it become very difficult to compute the fitness function values for all fitness cases when N increases. In the future work, we wish to find another way to compute the fitness function values faster for large N.

6.3.3 The N-BEP Problem with 16 Boolean Functions

Poli and Page [32] introduced various extensions of the GP algorithm by using new search operators and a new node representation together with a tree evaluation method known as sub-machine-code GP. The sub-machine-code GP technique allows the parallel evaluation of 32 or 64 fitness cases per program execution, which gives their algorithms the ability to evaluate the fitness values faster than the usual way. Poli and Page [32] succeeded to solve the N-BEP problems, up to $N = 22$, with the function set consisting of all 16 Boolean functions of two arguments [32].

Poli and Page [32] conducted experiments to show the performance of their algorithms on the N-BEP problem with $N = 3, \cdots, 6$, using a small population size, `nPop` $= 50$. They performed 50 independent runs for each problem to compute the CE of their algorithms using the set of all 16 Boolean functions

Table 8: The CE for the standard GP, GP-UX, GP-SUX and MP algorithms using the 16 boolean functions of 2 arguments for the N-BEP problems

N	GP	GP-UX	GP-SUX	MP
3	5,550	850	900	300
4	11,250	4,200	2,250	550
5	-	-	-	1,800
6	-	34,850	17,000	3,200
7	-	-	-	6,800
8	-	-	-	18,000

of two arguments as the function set.

In our experiments, we compare results of the MP algorithm with those appeared in [32] for the N-BEP problem with $N = 3, \cdots, 6$. We performed 50 independent runs for each N-BEP problem with $N = 3, \cdots, 8$, using the same function set and the population size as those used by Poli and Page [32]. The parameter values for the MP algorithm are shown in Table 1, while `nTrs` $= 2$ and `nFails` $= 1$. The results are shown in Table 8, where the results of the standard GP, the GP-UX and the GP-SUX algorithms are taken from Poli and Page [32].

From the results in Table 8, it is clear that the MP algorithm outperforms other algorithms. As a matter of fact, the high performance of the MP algorithm and other algorithms comes from the modularity of the problem itself. The Boolean even parity functions are compactly represented using XOR and XNOR Boolean functions [36]. Therefore, by adjusting the function set to contain XOR and XNOR functions, all versions of the GP algorithm can find the exact solution for the N-BEP problem easily. In particular, since the MP algorithm expresses a solution with more than one genes and uses the Boolean function XOR to link these genes, it is possible to find the exact solution of the N-BEP problem easily and fast. On the other hand, with the ADF technique, it is quite likely that the Boolean functions XOR and XNOR are constructed as ADFs by using {AND, OR, NAND, NOR}, which facilitates the mission of the algorithm.

6.3.4 The 6-BM Problem

Poli and Page [32] also conducted a set of experiments on the 6-BM problem to study the performance of the standard GP, GP-UX and GP-SUX algorithms in terms of the CE. They used the set of all 256 Boolean functions of three arguments [32]. Indeed, our results for the 6-BM problem in Table 4 seem to outperform their results shown in Table 2 in their paper [32]. However, we do not consider that this is a fair comparison, since the function set used for the MP algorithm is not the same as the one used for their algorithms.

Jackson [14] introduced a new technique to detect dormant nodes in GP programs and to prevent the neutral crossover process. A dormant node is a node in a program that does not contribute to the fitness value of the program. When the crossover operator switches a subtree rooted at a dormant node in a program, the resulting child will have the same fitness value as the parent, and this process is called a fitness-preserving crossover (FPC) [14]. Jackson [14] states that preventing the FPC improves the performance of GP in at least three ways; improving the execution efficiency, increasing the rate of success, and simplifying evolved programs.

Jackson [14] carried out a set of experiments on the 6-BM problem using the standard GP algorithm with and without preventing FPCs. For each algorithm, 100 independent runs were made using `nPop` $=$ 500 and `nGnrs` $= 50$. A comparison between these two versions of the GP algorithm was made in terms of the CE and the rate of success. In fact, as reported in [14], GP with preventing FPCs improved the CE and the rate of success for all test problems, except the 6-BM problem. For the 6-BM problem, the GP algorithm with preventing FPCs succeeded to reduce the CE, but it failed to improve the rate of success.

Table 9 shows a comparison between the MP algorithm and standard GP algorithm with and without preventing FPCs. We implemented the MP algorithm using different values for the parameters `nPop` and `nGnrs`. For each implementation, we performed 100 independent runs to compute the CE and the rate of success. The parameter values for the MP algorithm are `nTrs` $= 1$, `nFails` $= 1$ and the other parameters are set as shown in Table 1. The values of the parameters `nPop` and `nGnrs` are shown in Table 9 with the corresponding results of the MP algorithm. The results of GP with and without preventing FPCs are taken from the original paper [14]. As observed in Table 9, the MP algorithm outperforms the standard

Table 9: The CE and the rate of success for the standard GP algorithm, with and without preventing FPCs, and the MP algorithm for the 6-BM problem

Algorithm	nPop	nGnrs	CE	R%
GP	500	50	44,000	68
GP, preventing FPCs	500	50	38,500	65
MP	500	50	18,000	100
MP	250	100	23,250	98
MP	100	250	29,400	87
MP	50	500	37,800	74

GP algorithm with and without preventing FPCs in terms of both the CE and the rate of success.

7 Conclusions

We have proposed the MP algorithm that hybridizes the GP algorithm with a new set of local search procedures over a tree space to intensify promising programs generated by the GP algorithm. Its performance has been tested through extensive numerical experiments for some benchmark problems. The results of these experiments have shown that the MP algorithm outperforms the standard GP algorithm and recent versions of GP algorithm at least for the considered benchmark problems. In addition, we have shown that the MP algorithm can deal easily with the ADF technique to exploit the modularities in problem environments.

References

[1] Azad RMA, Ryan C (2006) An examination of simultaneous evolution of grammars and solutions. In: Yu T, Riolo RL, Worzel B (eds) Genetic programming: theory and practice III. Springer-Verlag, pp 141–158

[2] Bader-El-Den MB, Poli R, Fatima S (2009) Evolving timetabling heuristics using a grammar-based genetic programming hyper-heuristic framework. Memetic Comp 1:205–219

[3] Blickle T, Thiele LA (1997) Comparison of selection schemes used in evolutionary algorithms. Evol Comput 4:361–394

[4] Burke EK, Gustafson S, Kendall G (2004) Diversity in genetic programming: an analysis of measures and correlation with fitness. IEEE Trans Evol Comput 8:47–62.

[5] Cramer NL (1985) A representation for the adaptive generation of simple sequential programs. In: International conference on genetic algorithms and their applications. Carnegie Mellon University, Pittsburgh, USA, pp 183–187

[6] Eiben AE, Smith JE (2003) Introduction to Evolutionary Computing. Springer-Verlag, Berlin

[7] Ferreira C (2001) Gene expression programming: a new adaptive algorithm for solving problems. Complex Systems 13:87–129

[8] Hart W, Krasnogor N, Smith J (eds) (2004) Recent advances in memetic algorithms. Springer, Berlin, Heidelberg, New York

[9] Hedar A, Fukushima M (2006) Meta-heuristics programming. In: 2nd International workshop on computational intelligence & applications. Okayama, Japan

[10] Hedar A, Mabrouk E, Fukushima M (2008) Tabu programming method: a new meta-heuristics algorithm using tree data structures for problem solving. Technical report 2008-004, Kyoto University, Japan

[11] Hoai NX, McKay RI, Essam D (2006) Representation and structural difficulty in genetic programming. IEEE Trans Evol Comput 10:157–166

[12] Hoang TH, Hoai NX, McKay RI, Essam D (2006) The importance of local search: a grammar based approach to environmental time series modelling. In: Genetic programming: theory and practice III. Springer-Verlag, pp 159–175

[13] Xie H (2009) An analysis of selection in genetic programming. PhD thesis, Victoria University of Wellington, New Zealand

[14] Jackson D (2010) The identification and exploitation of dormancy in genetic programming. Genet Program Evolvable Mach 11:89–121

[15] Kishore JK, Patnaik LM, Mani V, Agrawal VK (2000) Application of genetic programming for multicategory pattern classification. IEEE Trans Evol Comput 4:242–258

[16] Koza JR (1990) Genetic programming: a paradigm for genetically breeding populations of computer programs to solve problems. Technical report CS-TR-90-1314. Stanford University, Stanford, USA

[17] Koza JR (1992) Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge

[18] Koza JR (1994) Genetic programming II: automatic discovery of reusable programs. MIT Press, Cambridge

[19] Koza JR, Bennett III FH, Andre D, Keane MA (1999) Genetic programming III: Darwinian invention and problem solving. Morgan Kaufmann, San Francisco

[20] Koza JR, Keane MA, Streeter MJ, Mydlowec W, Yu J, Lanza G (2003) Genetic programming IV: routine human-competitive machine intelligence. Kluwer Academic Publishers, Boston

[21] Krasnogor N, Smith JE (2005) A tutorial for competent memetic algorithms: model, taxonomy and design issues. IEEE Trans Evol Comput 9:474–488

[22] KRAMER O (2010) Iterated local search with Powell's method: a memetic algorithm for continuous global optimization. Memetic Comp, 2:69–83

[23] Langdon WB, Polií R (2002) Foundations of genetic programming. Springer-Verlag, Berlin

[24] Lin L, Gen M (2009) Auto-tuning strategy for evolutionary algorithms: balancing between exploration and exploitation. Soft Comput 13:157–168

[25] Mabrouk E, Hedar A, Fukushima M (2008) Memetic programming with adaptive local search using tree data structures. In: 5th international conference on soft computing as transdisciplinary science and technology. Cergy-Pontoise, Paris, France, pp 258–264

[26] Mabrouk E, Hedar A, Fukushima M (2009) Tabu programming: a machine learning tool using adaptive memory programming. In: 6th international conference on modeling decisions for artificial intelligence. Awaji Island, Japan, pp 187–198

[27] Moscato P (1989) On evolution, search, optimization, genetic algorithms and martial arts: towards memetic algorithms. Technical report 826, California Institute of Technology, Pasadena, CA

[28] Moscato P, Cotta C (2003) A gentle introduction to memetic algorithms. In: Glover F, Kochenberger G (eds) Handbook of metaheuristics. Kluwer Academic Publishers, Boston MA, pp 105–144

[29] Moscato P, Cotta C, Mendes A (2004) Memetic algorithms. In: Onwubolu GC, Babu BV (eds) New optimization techniques in engineering. Springer-Verlag, Berlin Heidelberg, pp 53–85

[30] Nordin P, Banzhaf W (1995) Complexity compression and evolution. In: 6th international conference on genetic algorithms. Morgan Kaufmann, Pittsburgh, PA, USA, pp 310–317

[31] Nordin P, Francone F, Banzhaf W (1995) Explicitly defined introns and destructive crossover in genetic programming. In: Workshop on genetic programming: from theory to real-world applications. Tahoe City, California, USA, pp 6–22

[32] Poli R, Page J (2000) Solving high-order boolean parity problems with smooth uniform crossover, sub-machine code GP and demes. Genet Program Evolvable Mach 1:37–56

[33] Poli R, Langdon WB (2006) Backward-chaining evolutionary algorithms. Artif Intell 170:953–982

[34] Riolo R, Soule T, Worzel B (eds) (2008) Genetic programming theory and practice V. Springer-Verlag, Berlin

[35] VEKARIA KP (1999) Selective crossover as an adaptive strategy for genetic algorithms. PhD thesis, University College, London

[36] Walker JA, Miller JF (2008) The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. IEEE Trans Evol Comput 12:397–417

[37] Zhang M, Gao X, Lou W (2007) A new crossover operator in genetic programming for object classification. IEEE T Syst Man Cy B 37:1332–1343