Tabu Programming Method: A New Meta-Heuristics Algorithm Using Tree Data Structures for Problem Solving

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

Abstract

The core of artificial intelligence and machine learning is to get computers to solve problems automatically. One of the great tools that attempt to achieve that goal is Genetic Programming (GP). GP is a generalization procedure of the well-known meta-heuristic of Genetic Algorithms (GAs). Meta-heuristics have shown successful performance in solving many combinatorial search problems. In this paper, we introduce a more general framework of meta-heuristics called Meta-Heuristics Programming (MHP) as general machine learning tools. As an alternative to GP, Tabu Programming (TP) is proposed as a special procedure of MHP frameworks. One of the main features of MHP is to exploit local search in order to overcome some drawbacks of GP, especially high disruption of its main operations; crossover and mutation. We show the efficiency of the proposed TP method through numerical experiments.

1 Introduction

The core of artificial intelligence and machine learning is to get computers to solve problems automatically. One of the great tools that attempt to achieve that goal is Genetic Programming (GP). GP is a generalization procedure of the well-known meta-heuristic of Genetic Algorithms (GAs). Specifically, GP can be regarded as a method of machine learning, while GAs are search paradigms that seek optimal solution candidates. GP was first introduced by Koza [20], and subsequently, the feasibility of this approach in well-known application areas has been demonstrated [21, 22, 23].

Genetic algorithms, the ancestor of GP, belong to the upper class of heuristics called meta-heuristics. Meta-heuristics have shown successful performance in solving many combinatorial search problems. Although there are many alternative heuristics to GAs, the latter are almost the only ones that have been generalized as machine learning tools. To the best of the authors' knowledge, there have been just a few attempts to generalize other meta-heuristics. Specifically, ant colony programming [2, 3] is a generalized meta-heuristic of the ant colony optimization algorithm. This work along with the future planned works attempt to develop many machine learning tools alternative to GP by generalizing other meta-heuristics.

Crossover and mutation, the main operations in GP, have been extensively studied. Many effective settings of these operations have been proposed to fit a wide variety of problems. However, it has been addressed that crossover and mutation are highly disruptive with a risk of convergence to a non-optimal structure [19, 27, 28]. Altering a node high in the tree may result in serious disruption of the subtree below. There have been many attempts to edit GP operations to make changes in small scales, for example by using natural language processing [19, 23]. This motivates us to use more local searches with gradual changes of scales within a general framework.

In this paper, new treatments are introduced to overcome the disruption of crossover and mutation by composing new alternatives for GP. GP searches in a solution space of computer programs. These programs can be represented by using tree data structures. We introduce some local search procedures over

[1]Department of Computer Science, Faculty of Computers and Information, Assiut University, EGYPT (email: hedar@aun.edu.eg).

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

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

a tree space as alternative operations to crossover and mutation. These procedures aim to generate trial moves from a current tree in its neighborhood. Using these search procedures, various meta-heuristics can be generalized to deal with tree data structures in a unified framework which we call Meta-Heuristics Programming (MHP).

The paper is organized as follows. In the next section, we give a classification of meta-heuristics along with their brief description. In Section 3, we highlight the main structure of GP to give motivation of Meta-Heuristics Programming (MHP). The basic procedures for stochastic local search over a tree space is presented in Section 4. Then, we show the main framework of MHP in Section 5, with an example of Tabu Programming (TP). In Section 6, we report numerical results for two types of benchmark problems; symbolic regression problems and the 6-Bit multiplexer problem. Finally, concluding remarks and some future work make up Section 7.

2 Meta-heuristics

The term meta-heuristics, first used by Glover [9], contains all heuristics methods that show evidence of achieving good quality solutions for a problem of interest within an acceptable time. Usually, meta-heuristics offer no guarantee of obtaining the global best solutions [10].

In terms of the process of updating solutions, meta-heuristics can be classified into two classes; population-based methods and point-to-point methods. In the latter methods, the search keeps only one solution at the end of each iteration, from which the search will start in the next iteration. On the other hand, the population-based methods keep a set of many solutions at the end of each iteration.

In terms of search methodologies and trial solutions generation, meta-heuristics can be categorized into several groups of methods, as shown in Figure 1 and described as follows.

- **Evolutionary Algorithms (EAs).**
  EAs try to mimic the evolution of a species. Specifically, EAs simulate the biological processes that allow the consecutive generations in a population to adapt to their environment. The adaptation process is mainly applied through genetic inheritance from parents to children and through survival of the fittest.
  The main EAs are Genetic Algorithms, Evolution Strategies, Evolutionary Programming, and Scatter Search. In contrast to other EAs, Scatter Search invokes more artificial elements, like using memory elements to update populations.

- **Memory-Based Heuristics**
  The main feature of memory-based heuristics is their use of an adaptive memory and responsive exploration. The role of the adaptive memory is to prevent the search from getting trapped in local optimal solutions, and to direct the search to more effective diversification and intensification processes.
  Tabu Search and Scatter Search are the most well-known memory-based heuristics.

- **Neighborhood Search Heuristics**
  A neighborhood search heuristics starts from a candidate solution and then iteratively moves to a neighbor solution. Therefore, a neighborhood relation and structure should be defined on the search space.
  Neighborhood search heuristics contains Variable Neighborhood Search and Tabu Search.

- **Swarm Intelligence**
  Swarm intelligence consists of artificial intelligence techniques that study and simulate the collective behaviors and self-organized systems of animal swarms.
  The most well-studied swarm intelligence methods are Ant Colony Optimization and Particle Swarm Optimization.

- **Probabilistic-Based Heuristics**
  Probabilistic-based heuristics determine whether or not the current solution is replaced by a new

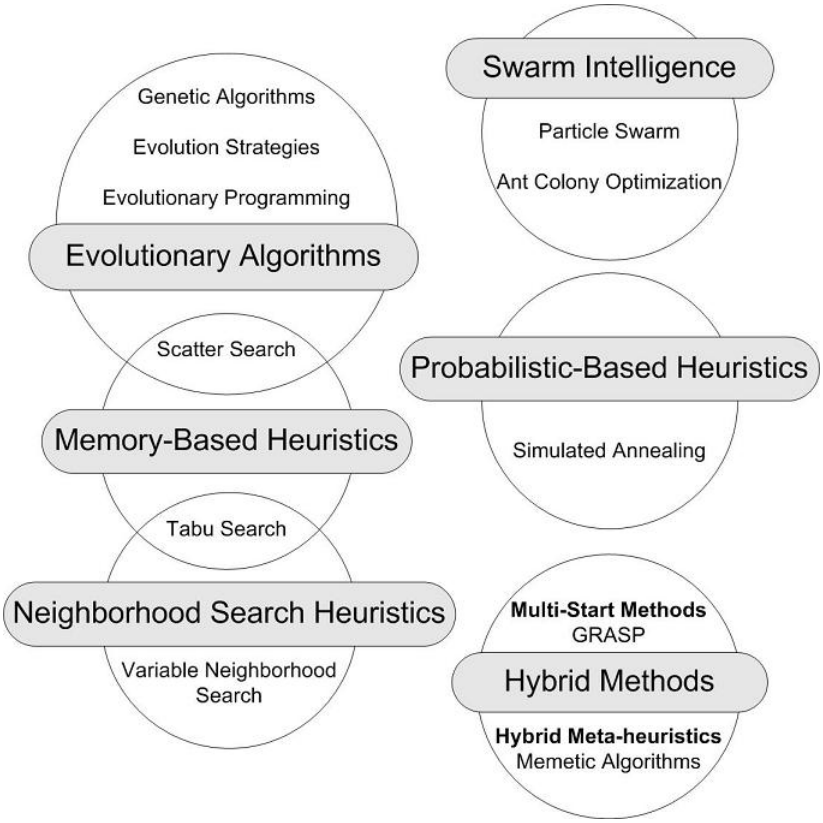

Figure 1: Classifications of Meta-heuristics

trial point based on a probability depending on the difference between their function values. Simulated Annealing is the most well-known probabilistic-based heuristics.

- **Hybrid Methods**
  There are many different scenarios to compose hybrid meta-heuristics. Using local search methods inside meta-heuristics is the most effective way to overcome the slowness of meta-heuristics. In addition, a multi-start local search method is another scheme for composing hybrid meta-heuristics.
  Memetic Algorithm is an example of hybrid meta-heuristics and Greedy Randomized Adaptive Search Procedure (GRASP) is an example of multi-start methods.

The most commonly used data structure types in the above-mentioned meta-heuristics are bit-strings and real-valued vectors. Moreover, meta-heuristics are typically applied to problems that can be modeled or transformed to optimization problems [29]. On the other hand, GP is a general machine learning tool that can deal with more general data structure. In this paper, Meta-Heuristics Programming (MHP) is introduced as a competitor to GP. Specifically, MHP is a general machine learning tool that deals with tree structure data.

3 Local Search over Tree Space

MHP searches a solution space of computer programs like GP. These programs can be represented as a parse tree[1], in which leaf nodes are called terminals and internal nodes are called functions. Depending on the problem at hand, users can define the domains of terminals and functions. In the coding process, tree structures of solutions should be transformed to executable codes. Usually, these codes are

[1]Parse tree is a data structure representation in a language, and each element in a parse tree is called a node. In addition, the start node represents the root of the structure, the interior nodes represent non-terminals (functions) symbols and the leafs nodes represent terminals symbols.

expressed to closely match the Polish notation of logic expressions [5]. Figure 2 shows three examples of tree representation of individuals and their executable codes below these trees. These codes are geometrically executed in Figure 3 as solid lines or curves. Then, a fitness function should be defined to measure the quality of the individuals represented by these codes. If the target is to obtain the dotted curve as a fitting curve of some given dataset (i.e., a symbolic regression problem), then the fitness function may be defined as an error function measured on the given dataset.

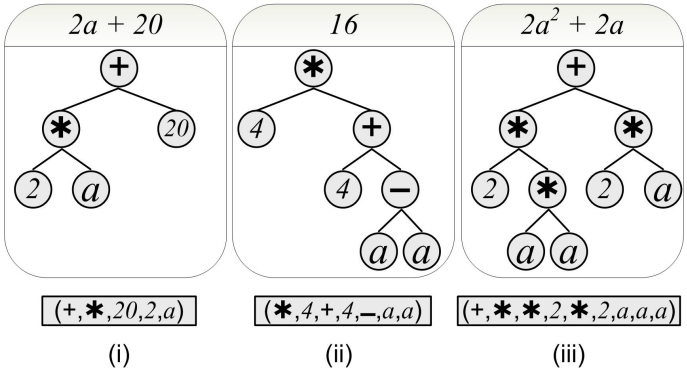

Figure 2: Example of GP Representation

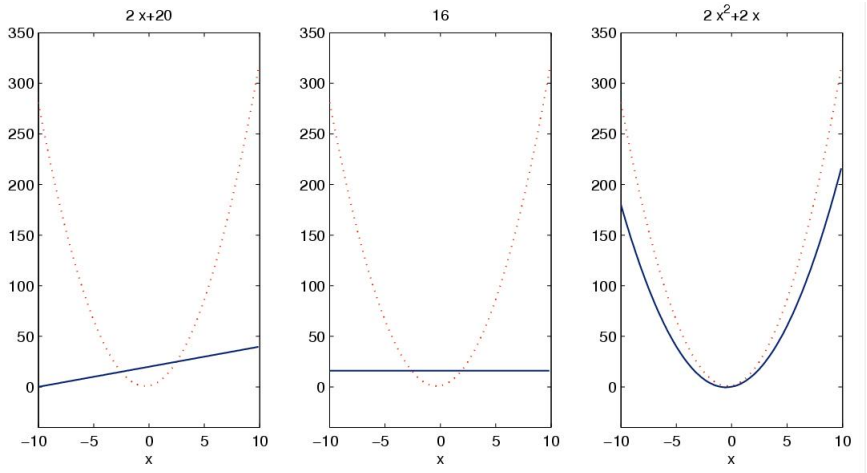

Figure 3: Example of GP Representation

In this section, some local search procedures over a tree space are introduced. These procedures aim to generate trial moves from a current tree to another tree in its neighborhood. The proposed local searches have two aspects; intensive and diverse. Intensive local search aims to explore the neighborhood of a tree by altering its nodes without changing its structure. Diverse local search changes the structure of a tree by expanding its terminal nodes or cutting its subtrees[2]. We introduce *Shaking* as an intensive local search procedure, and *Grafting* and *Pruning* as diverse local search procedures.

For a parse tree X, we define its order, size and depth as follows.

- *Tree Order* $|X|$ is the number of all nodes in tree X.
- *Tree Size* $s(X)$ is the number of leaf nodes in tree X.
- *Tree Depth* $d(X)$ is the number of links in the path from the root of tree X to its farthest node.

[2]Throughout the paper, the term "*branch*" is used to refer to a subtree, see [4, 25].

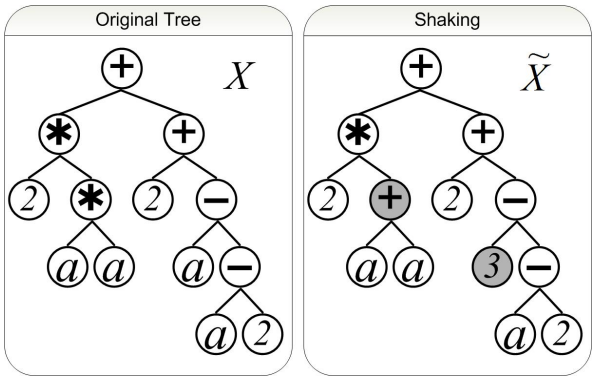

Figure 4: An Example of Shaking Search ($\nu = 2$)

3.1 Shaking Search

Shaking search is an intensification search procedure that alters a tree X to a new one $\tilde{X}$. Both X and $\tilde{X}$ have the same tree structure since the altered nodes are replaced by alternative values, i.e., an altered terminal node is updated by a new terminal value and an altered node containing a binary function is replaced by a new binary function, and so on. Procedure 3.1 states the formal description of shaking search, while Figure 4 shows an example of shaking search that alters two nodes of X. In Procedure 3.1, ν is a positive integer.

Procedure 3.1 $\tilde{X} = \text{Shaking}(X, \nu)$

Step 1. *If $\nu > |X|$, return.*

Step 2. *Set $\tilde{X} := X$ and choose ν nodes of $\tilde{X}$ randomly.*

Step 3. *Update the chosen nodes by new randomly chosen alternatives.*

Step 4. *Return.*

A neighborhood $N_S(X)$ of a tree X based on shaking search can be defined by

$$N_S(X) = \{\tilde{X} | \tilde{X} = \textit{Shaking}(X, \nu), \nu = 1, \ldots, |X|\}. \quad (1)$$

It is worthwhile to note that the random choices of Steps 2 and 3 of Procedure 3.1 make Shaking Procedure behave as stochastic search. Therefore, for a tree X, one may get a different $\tilde{X}$ in each run of the procedure.

3.2 Grafting Search

In order to increase the variability of the search process, grafting search is invoked as a diverse local search procedure. Grafting search generates an altered tree $\tilde{X}$ from a tree X by expanding some of its leaf nodes to branches. As a result, X and $\tilde{X}$ have different tree structures since $|\tilde{X}| > |X|$, $s(\tilde{X}) \geq s(X)$, and $d(\tilde{X}) \geq d(X)$. Procedure 3.2 states the formal description of grafting search where λ refers to the number of leaf nodes which are updated to be branches. Figure 5 shows an example of grafting search that alters two nodes of X by two branches in $\tilde{X}$.

Procedure 3.2 $\tilde{X} = \text{Grafting}(X, \lambda)$

Step 1. *If $\lambda > s(X)$, return.*

Step 2. *Set $\tilde{X} := X$.*

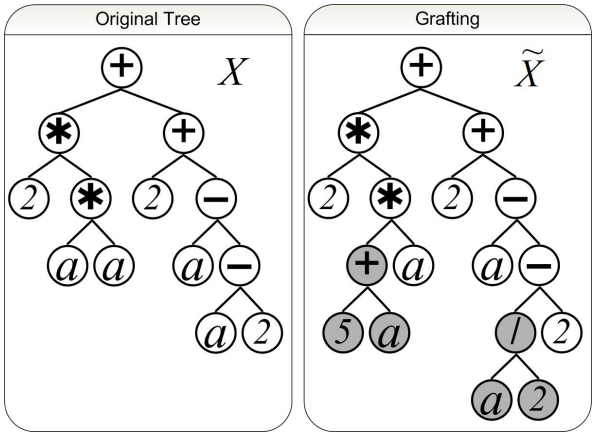

Figure 5: An Example of Grafting Search ($\lambda = 2$)

Step 3. *Generate λ branches $B_1, \ldots, B_\lambda$ randomly. Choose terminal nodes $t_1, \ldots, t_\lambda$ of $\tilde{X}$ randomly.*

Step 4. *Update $\tilde{X}$ by replacing the nodes $t_1, \ldots, t_\lambda$ by the branchs $B_1, \ldots, B_\lambda$.*

Step 5. *Return.*

A neighborhood $N_G(X)$ of a tree X based on grafting search can be defined by

$$N_G(X) = \{\tilde{X} | \tilde{X} = \textit{Grafting}(X, \lambda), \lambda = 1, \ldots, s(X)\}. \tag{2}$$

It is worthwhile to note that the random choices of Step 3 of Procedure 3.2 also make Grafting Procedure behave as stochastic search. Therefore, for a tree X, one may get a different $\tilde{X}$ in each run of the procedure.

3.3 Pruning Search

Pruning search is another diverse local search procedure. Contrary to grafting search, pruning search generates an altered tree $\tilde{X}$ from a tree X by cutting some of its branches. Therefore, X and $\tilde{X}$ have different tree structures since $|\tilde{X}| < |X|$, $s(\tilde{X}) \leq s(X)$, and $d(\tilde{X}) \leq d(X)$. In the coding process, it is more convenient to express the tree X in a special code and use it to distinguish all possible branches which may be selected for pruning. Specifically, we introduce the branch coding (Procedure 3.3) to assist pruning search, which expresses X as a parse tree containing meta-terminal-nodes. These meta-terminal-nodes are the branches of X that have the same depth ζ. If X has ξ branches $B_1, \ldots, B_\xi$ of depth ζ, then the branch coding expresses X in a form that distinguishes these branches. In other words, Procedure 3.3 extracts all branches in X with depth ζ, which can be written as $[B_1, \ldots, B_\xi] = \boldsymbol{BC}_\zeta(X)$, where $d(B_1) = \cdots = d(B_\xi) = \zeta$. For instance, if pruning search is applied to cut a branch of depth 1 in tree X, then the branch coding procedure is called to express each branch of depth 1 in X as a one meta-terminal-node as shown in Figure 6. Hence, pruning search can easily choose one of these branches and replace it by a randomly generated leaf node.

Procedure 3.3 $[B_1, \ldots, B_\xi] = \mathrm{BC}_\zeta(X)$

Step 1. *If $\zeta \geq d(X)$, return.*

Step 2. *Select all branches $B_1, \ldots, B_\xi$ in X with depth ζ.*

Step 3. *Return.*

The formal description of pruning search is given below in Procedure 3.4, while Figure 7 shows an example of pruning search that cuts two branches of X. In Procedure 3.4, η is a positive integer that represents the number of times a branch is replaced by a leaf node during pruning search.

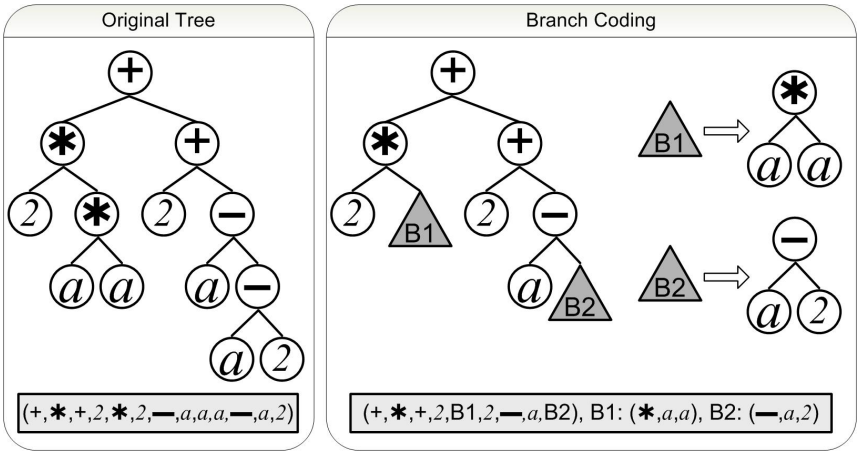

Figure 6: An Example of Branch Coding ($\xi = 2, \zeta = 1$)

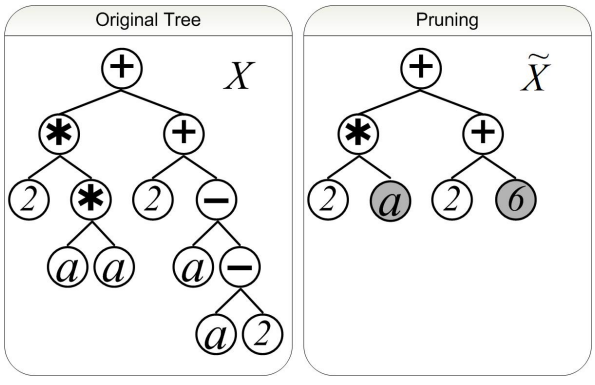

Figure 7: An Example of Pruning Search ($\eta = 2, \zeta_1 = 1, \zeta_2 = 2$)

Procedure 3.4 $\tilde{X} = \text{Pruning}(X, \eta)$

Step 1. *Set* $\tilde{X} := X$.

Step 2. *Repeat the following Steps (2.1)-(2.4) for* $j = 1, \ldots, \eta$.

2.1 *Generate a natural number* ζ_j *randomly such that* $\zeta_j < d(X)$.

2.2 *Use Procedure 3.3 to get* $[B_1, \ldots, B_{\xi_j}] := \text{BC}_{\zeta_j}(\tilde{X})$.

2.3 *Generate a random terminal node* t_j.

2.4 *Update* $\tilde{X}$ *by replacing a randomly chosen branch from* $\{B_1, \ldots, B_{\xi_j}\}$, *by* t_j.

Step 3. *Return.*

A neighborhood $N_P(X)$ of a tree X based on pruning search can be defined by

$$N_P(X) = \{\tilde{X} | \tilde{X} = \textit{Pruning}(X, \eta), \eta = 1, \ldots, |X| - s(X)\}. \tag{3}$$

It is worthwhile to note that the random choices of Steps 2.1 and 2.3 of Procedure 3.1 make Pruning Procedure behave as stochastic search. Therefore, for a tree X, one may get a different $\tilde{X}$ in each run of the procedure.

4 Meta-Heuristics Programming

Most of the search methodologies in meta-heuristics depend on local search. Therefore, by using the local search procedures defined in Section 3, various meta-heuristics can be generalized to deal with tree data structures, which we call Meta-Heuristics Programming (MHP). This section shows the main procedures of MHP and describes how they can be implemented. Then, Tabu Programming is presented in Section 5 in order to give a practical example of how a meta-heuristic of Tabu Search is modified to yield a meta-heuristic of Tabu Programming.

The MHP framework tries to cover many of the well-known meta-heuristics as special cases. In addition, the MHP framework generalizes the data structures used in most of the ordinary meta-heuristics, by introducing tree data structures instead of bit strings or vectors of numbers. In the MHP framework, initial computer programs (or an initial computer program) represented as parse trees can be adapted through the following five procedures to obtain acceptable target solutions of the problem.

- TRIALPROGRAM: Generate trial program(s) from the current ones.
- UPDATEPROGRAM: Choose one program or more from the generated ones for the next iteration.
- ENHANCEMENT: Enhance the search process to be accelerated if a promising solution is detected, or escape from local information if an improvement cannot be achieved.
- DIVERSIFICATION: Drive the search to new unexplored regions in the search space by generating new structures of programs.
- REFINEMENT: Improve the best programs obtained so far.

TRIALPROGRAM and UPDATEPROGRAM procedures are the essential ones in MHP. The other three procedures are recommended to achieve better and faster performance of MHP. Actually, these procedures make MHP behave like an intelligent hybrid framework. Table 1 summarizes the MHP procedures and shows some examples of each procedure.

The search procedures defined in Section 3 are used in TRIALPROGRAM procedure, while UPDATEPROGRAM procedure depends on the invoked type of meta-heuristics.

The main structure of the MHP framework is shown below in Algorithm 4.1. In its initialization step, MHP algorithm generates an initial set of trial programs which may be a singleton set in the case of point-to-point meta-heuristics. The main loop in MHP algorithm starts by calling TRIALPROGRAM

Table 1: MHP Procedures

Procedure	Function	Example
TRIALPROGRAM	generates trial program(s)	GP & GA: Crossover and Mutation
UPDATEPROGRAM	chooses one program or more for the next iteration	GP & GA[14]: Selection
ENHANCEMENT	refines some characteristic programs	MA[26]: Local Search SS: Improvement Method
DIVERSIFICATION	drives the search to new unexplored structures of programs	TS & SS[24]: Diversification
REFINEMENT	improves the best programs	Hybrid SA [15, 16, 18]: Final Intensification

MA = Memetic Algorithm, SS = Scatter Search, SA = Simulated Annealing

procedure to generate a set of trial programs from the current iterate program or from the current population. Then, MHP algorithm detects characteristic states in the recent search process and applies ENHANCEMENT procedure to generate new promising trial programs using the following tactics.

- *Intensive Enhancement.* Apply a faster local search whenever a promising improvement has been detected.
- *Diverse Enhancement.* Apply an accelerated escape strategy whenever a non-improvement has been detected.

To proceed to the next iteration, UPDATEPROGRAM procedure is used to invoke the next iterate program or the next population from the current ones. Consequently, the control parameters are also updated to fit the next iteration. It is noteworthy that MHP uses an adaptive memory to check the progress of the search process. Two types of memories are defined as follows.

- *Assembly Memory*. Start with empty memory and collect useful search information hierarchically.
- *Global Memory*. Start with a full memory that samples the whole search space, and remove memory elements whenever new data structures have been visited.

Having a full assembly memory or having an empty global memory can be used to terminate the MHP algorithm. If the termination criteria are met, then the REFINEMENT procedure is applied to improve the elite solutions obtained so far. Otherwise, the search proceeds to the next iteration but the need of diversity is checked first. DIVERSIFICATION procedure may be applied to generate new diverse solutions by guidance of the adaptive memory.

Algorithm 4.1 *Meta-Heuristics Programming*

- ***Step 1.*** *Initialization.*
- ***Step 2.*** *Apply* TRIALPROGRAM *procedure.*
- ***Step 3.*** *Apply* ENHANCEMENT *procedure.*
- ***Step 4.*** *Apply* UPDATEPROGRAM *procedure.*
- ***Step 5. Update_Parameters***.
- ***Step 6.*** *If* ***Termination_Conditions*** *are satisfied, go to Step 8.*

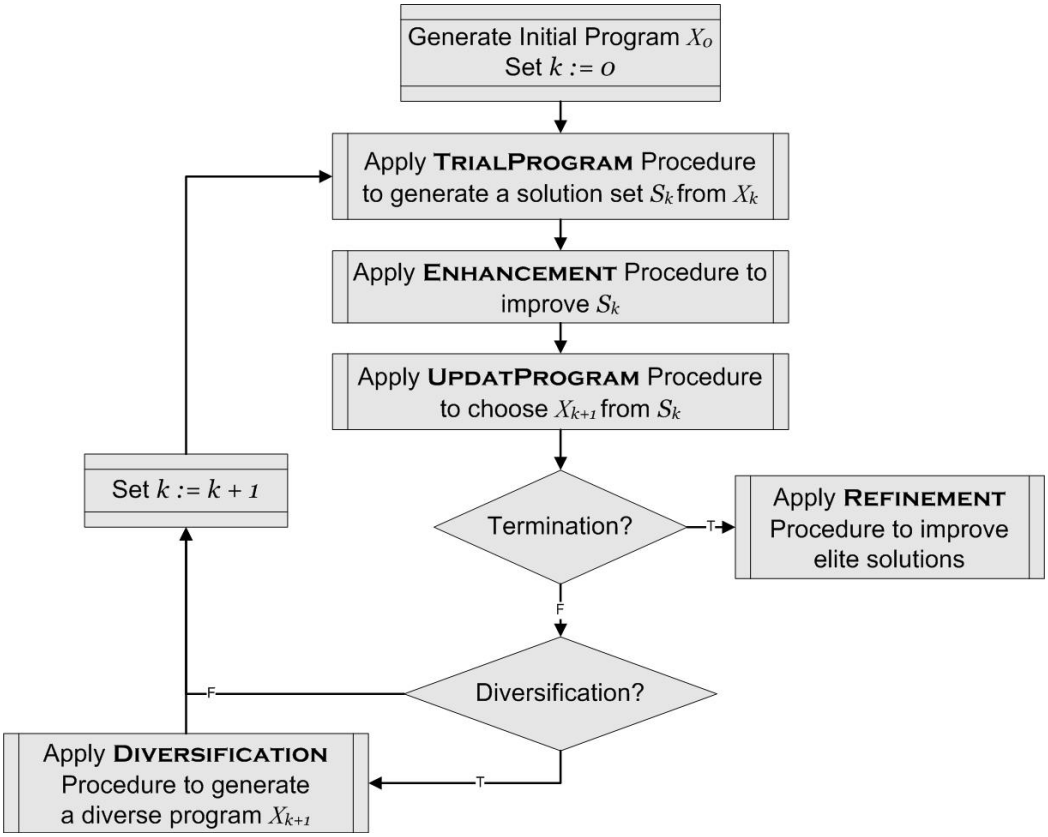

Figure 8: Point-to-Point MHP

- ***Step 7.*** *If diverse solutions are needed, apply* DIVERSIFICATION *procedure. Go to Step 2.*
- ***Step 8.*** *Apply* REFINEMENT *procedure.*

Algorithm 4.1 can be implemented in different ways depending on the type of the invoked meta-heuristics; point-to-point or population-based. Figure 8 shows a flowchart of the point-to-point MHP, while Figure 9 shows the population-based MHP.

5 Tabu Programming

The main feature of Tabu Search is to use an adaptive memory to direct the search process. The simplest form of this adaptive memory, called short-term memory, uses the recency only, i.e., it saves the recently visited solutions to avoid cycling. Using long-term memories make the search process behave more intelligently. Long-term memories may save the elite solutions, the frequency of visiting different solution structures, and so on [11, 12, 13, 17].

This section generalizes Tabu Search by introducing Tabu Programming as a special case of the MHP framework. Therefore, as in Tabu Search, TP invokes three basic search stages; *local search*, *diversification* and *intensification*. In the local search stage, TP uses local searches over the tree space, as described in Section 3, to explore the solution space around the current iterate program X_k. On the other hand, TP follows MHP by using two types of local searches; intensive local search by TRIALPROGRAM procedure, and diverse local search by ENHANCEMENT procedure. Intensive local search aims to explore the neighborhood of X_k by altering its nodes without changing its structure through *shaking* search. In addition, diverse local search tries to locally change the tree structure of X_k through *grafting* and *pruning* searches. Then, the DIVERSIFICATION procedure is applied in order to diversify the search to new tree structures. Finally, in order to explore the close tree structures around the best programs visited so far, the REFINEMENT procedure is applied to improve these best programs further. Figure 10 shows the main structure of TP, and its formal description is given below.

Algorithm 5.1 *Tabu Programming*

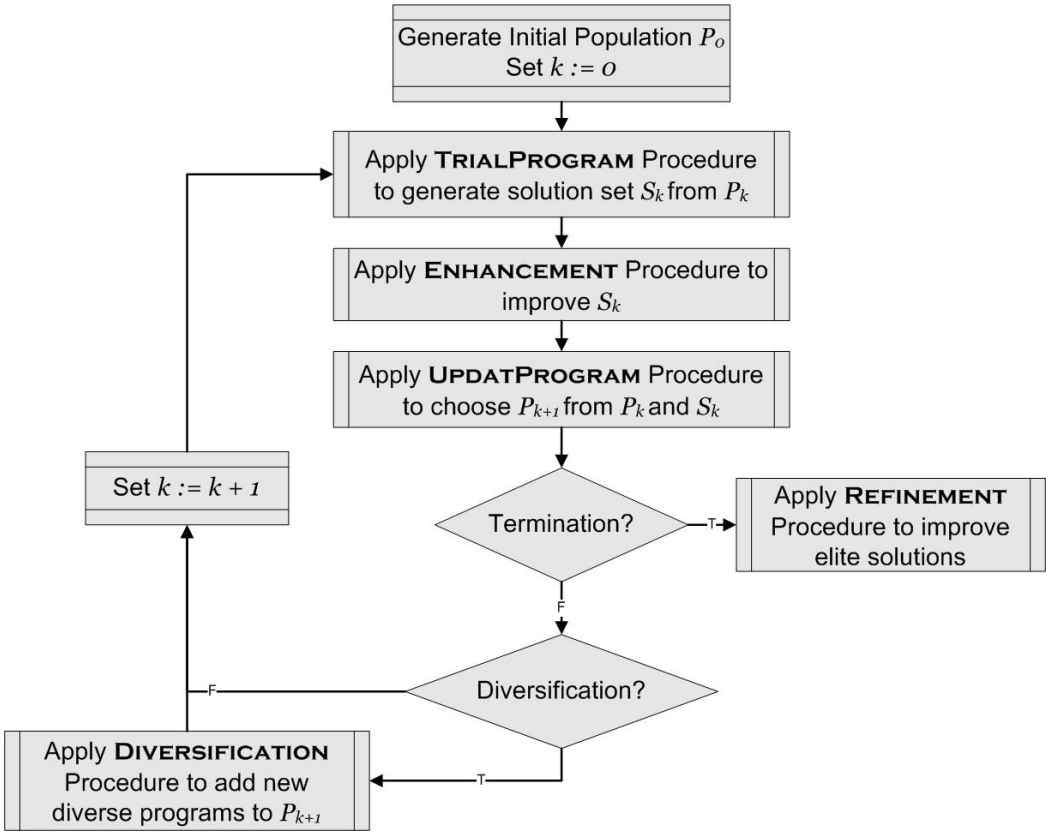

Figure 9: Population-Based MHP

1. ***Initialization.***

   *Choose an initial program X_0, set tabu list (*TL*) and other memory elements to be empty, and set a counter $k := 0$. Ask the user for the values of n_T, n'_T and n^*.*

2. ***Main Loop.***

   2.1 ***Intensive Local Search.*** *Repeat the following Steps (2.1.1)-(2.1.3) until non-improvement conditions are satisfied.*

   2.1.1 *Generate a set of n_T trial programs $S_k \subseteq N_S(X_k)$, as defined in Equation (1), based on tabu restrictions by applying Shaking procedure successively.*

   2.1.2 *Set the next iterate program X_{k+1} to be the best program in S_k.*

   2.1.3 *Update* TL *and other memory elements, and set $k =: k + 1$.*

   2.2 ***Diverse Local Search.***

   2.2.1 *If a growing tree structure is needed, then go to Step 2.2.2. Otherwise, go to Step 2.2.3*

   2.2.2 *Generate a set of n'_T trial programs $S'_k \subseteq N_G(X_k)$, as defined in Equation (2), based on tabu restrictions by applying Grafting procedure successively.*

   2.2.3 *Generate a set of n'_T trial programs $S'_k \subseteq N_P(X_k)$, as defined in Equation (3), based on tabu restrictions by applying Pruning procedure successively.*

   2.2.4 *Set the next iterate program X_{k+1} to be the best program in S'_k.*

   2.2.5 *Update* TL *and other memory elements, and set $k =: k + 1$.*

3. ***Termination.*** *If the termination conditions are satisfied, then go to Step 5. Otherwise, go to Step 4.*

4. ***Diversification.*** *If diversification is needed, choose a new diverse structure program X_{k+1}, set $k =: k + 1$, and go to Step 2.*

5. ***Refinement.*** *Try to improve the n^* best obtained programs.*

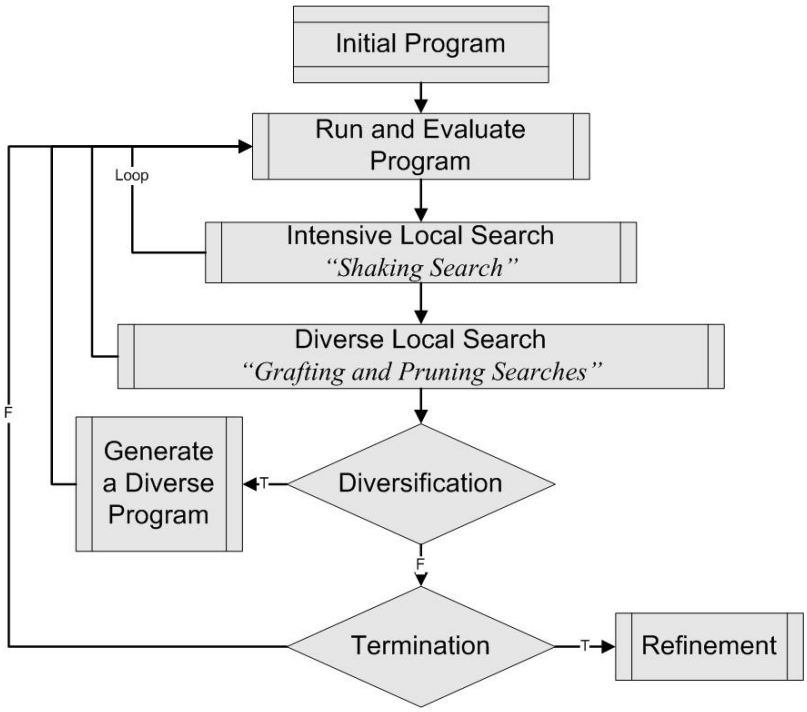

Figure 10: Tabu Programming Structure

5.1 Diversification and Intensification Searches and Long-Term Memory

Different types of long-term memories can be invoked to enhance the search process. Memory may save visited elite solutions for further use in intensification mechanism. Moreover, historical search information may also be saved to assist the diversification mechanism. For instance, visited tree structures can be saved to generate a new diverse tree structure as shown in Figure 11. Moreover, frequencies of choosing a node to be a terminal or a function can be saved in order to use it in generating a new tree structure which may have been overlooked in the previous search process.

5.1.1 Representations of TP Individuals

One of the most important objects in our method is the definition of the gene. The gene in TP is the smallest structure in the representation of the suggested solution, where every gene consists of a linear symbolic string composed of terminals and functions. In addition, every gene contains two parts, head (functions and terminals) and tail (terminals only).

The length of the gene depends on the length h of its head, and the maximum number n of arguments of the function. In addition, we compute the length of the tail by the formula $t = h(n-1)+1$. Indeed, it is clear that the length of the gene depends on the problem itself. To construct a gene in TP, we first construct a temporary random gene of temporary length $h+t$, and then depending on the shape of the gene and the functions and terminals that allowed, we reduce the length of the gene by deleting the unnecessary elements at the end of the gene. For example, when $h = 5$, $F = \{+, -, *, /\}$ and $T = \{a\}$, we have $t = h(n-1)+1 = 6$ since $n = 2$. In addition, suppose that we construct this temporary random gene of length 11 as in Figure 12a.

Now, if we try to convert this gene to a tree representation (Figure 12b), we will find that the tree representation needs only the first 7 elements of the previous gene. Consequently, we will consider these 7 elements only and delete the remaining elements as in Figure 12c to get the final form of the gene.

In TP, every individual solution has a coding representation called genome or chromosome [6], which is composed of one or more genes. In addition, for each problem, the number of genes and the length of the heads must be chosen. Then, we link these genes in every chromosome by using a suitable linking function depending on the problem itself. For example, in algebraic expressions, any mathematical function with more than one arguments (like + or *) can be used to link these genes to get a final chromosome.

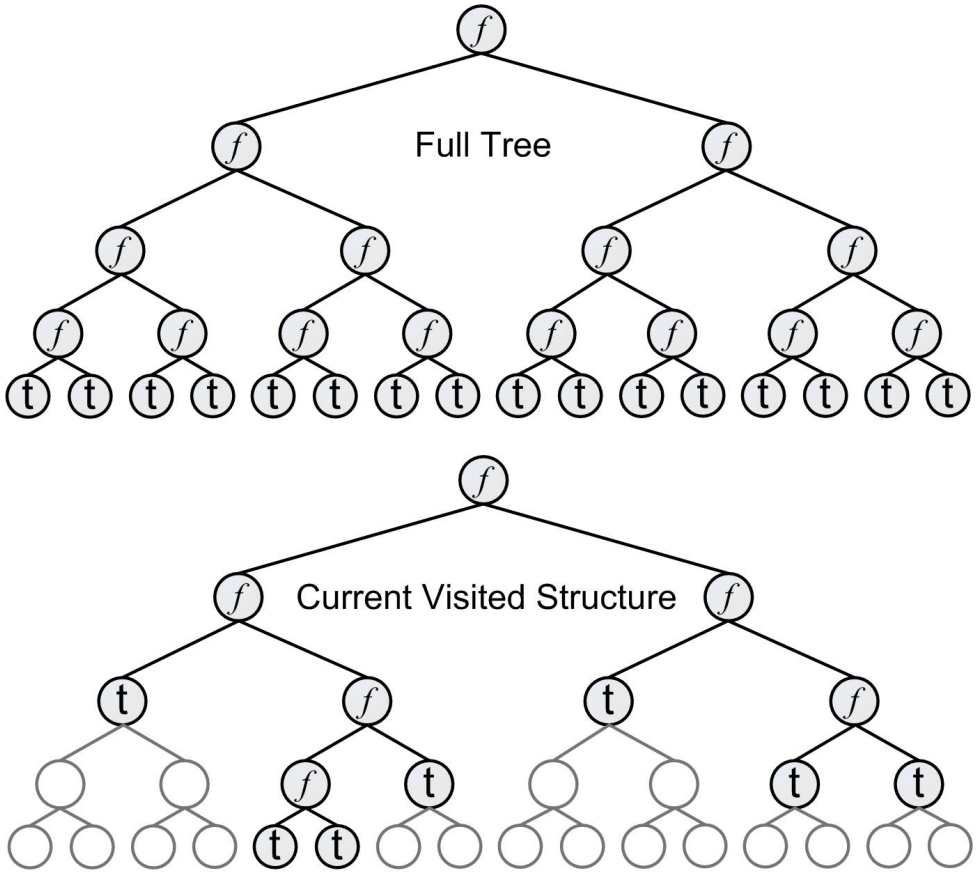

Figure 11: An Example of a Visited Tree Structure for Full Tree with Depth 4

Adapting the chromosome to contain more than one genes increases the probability of finding suitable solutions and enables the algorithm to deal with more complex problems [6].

5.1.2 Set of Parameters in TP

As described in the previous subsections, TP makes use of a set of parameters. We list these parameters in the following:

- `hLen`: The maximum length of the head for every gene in the initial solution.
- `nGene`: The number of genes in every chromosome.
- `nTL`: The number of elements in the set of tabu list, which represents the short-term memory in the program to avoid cycling in the solutions.
- `nTrs`: The number of suggested trial solutions in the neighborhood of the current solution. We set n_T and n'_T in Algorithm 5.1 equal to `nTrs`.
- `ILNonImp`: The maximum number of consecutive non-improvements in the intensive local search (termination condition of the intensive local search).
- `DLNonImp`: The maximum number of consecutive non-improvements in the diverse local search (termination condition of the diverse local search).
- `IntNonImp`: The maximum number of consecutive non-improvements in the intensification step (termination condition of the intensification step).
- `FunCnt`: The maximum number of function evaluations (fitness evaluations).

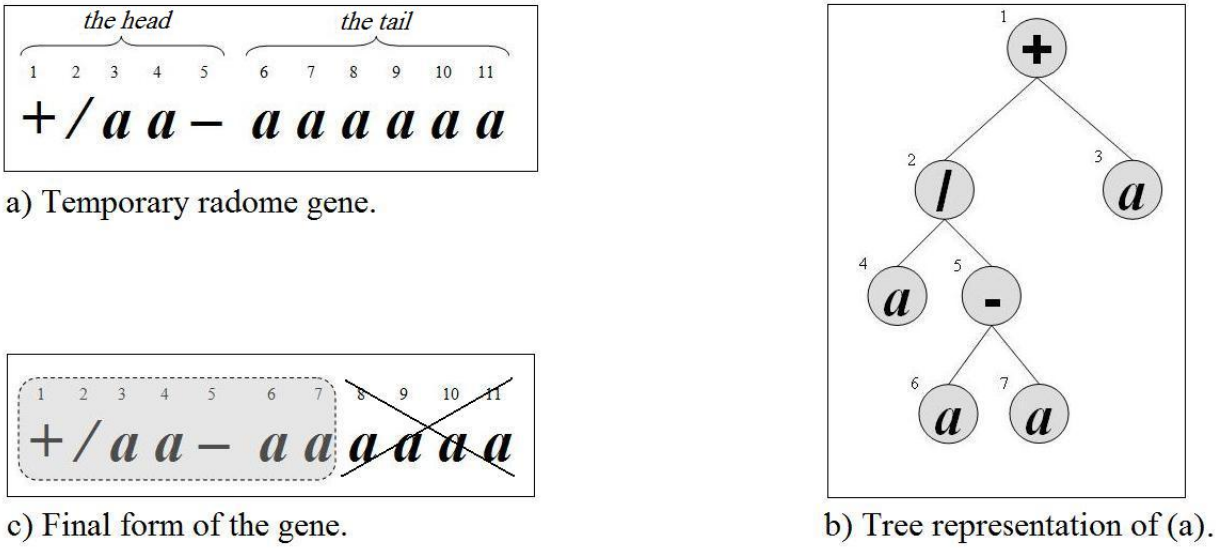

Figure 12: Constructing new gene.

6 Numerical Experiments

In this section, we discuss the performance of the TP method through two types of benchmark problems; the symbolic regression problem and the 6-Bit multiplexer problem. Several preliminary experiments were carried out first to study the effect of TP parameters behavior, and to study the efficiency of local search over the tree space. Then, we conduct extensive experiments to analyze the main components of the TP method. Finally, we make some comparisons between the TP method and the Gene Expression Programming (GEP) method [6, 7, 8].

6.1 Symbolic Regression Problem

The terminology symbolic regression represents the process of fitting a measured data set by a suitable mathematical formula. Thus, for a given dataset $\{(x_j, y_j)\}_{j=1}^{N}$, we search for a function g such that the mean-absolute error, chosen as an instance of an error function,

$$\frac{1}{N}\sum_{j=1}^{N}|y_j - g(x_j)|, \tag{4}$$

is minimized, where not only the form but also the coefficients in the expression of g are unknown.

6.1.1 Test Problems

Suppose that we are given a sample dataset over N randomly chosen points. Now, we want to find a function g to fit those values with a minimum error. In fact, one of the most important factors in evolutionary algorithms is the choice of a suitable fitness function, specially for symbolic regression problems.

Here we will use the following fitness function, which has been employed in the literature [6, 7]:

$$F = \sum_{j=1}^{N}(100 - |g(x_j) - y_j|), \tag{5}$$

where $M = 100$. We will adapt the precision of 0.01, that is, $|g(x_j) - y_j|$ is regarded as zero if $|g(x_j) - y_j| \leq 0.01$. Clearly, maximizing the fitness function (5) is equivalent to minimizing the error function (4), and the maximum value of the fitness function is $F_{max} = 100N$.

Test Problem 1. Consider the polynomial

$$f(x) = 3x^2 + 2x + 1. \tag{6}$$

We generate a dataset by computing the function values at 10 points randomly chosen from the real interval $[-10, 10]$. The dataset thus obtained is shown in Table 2.

Test Problem 2. Consider the polynomial

$$f(x) = x^4 + x^3 + x^2 + x. \tag{7}$$

We generate a dataset by computing the function values at 10 points arbitrary chosen from the real interval $[0, 20]$, and these points are also shown in Table 2.

Test Problem 1		Test Problem 2	
x	$f(x)$	x	$f(x)$
–4.2605	46.9346	2.81	95.2425
–2.0437	9.4427	6	1554
–9.8317	271.3236	7.043	2866.5486
2.7429	29.0563	8	4680
0.7326	4.0753	10	11110
–8.6491	208.1226	11.38	18386.0341
–3.6101	32.8783	12	22620
–1.8999	8.0291	14	41370
–4.8852	62.8251	15	54240
7.3998	180.0707	20	168420

Table 2: Datasets for Test Problems 1 and 2

6.1.2 Effect of Parameters

Here, we study the effect of the choice of parameters on the behavior of the TP method and discuss how we can choose their best values for each problem. For every parameter, we chose several values, and for each value, we performed 50 runs to compute the rate of success for this value, while letting the other parameters be fixed at its standard values given in Table 3.

The computational results are displayed in Figures 13–20. As we can see from these figures, for Test Problem 1, the results are not very sensitive to the changes of parameter values except nGene and nTrs and all tested values affect the success rate only slightly. On the other hand, for Test Problem 2, the results are sensitive to the changes of all parameters except nTL and IntNonImp and the success rate is significantly affected by changing their values. But, the two parameters nGene and nTrs are still most influential on the results.

Parameter	Value	
	Test Problem 1	Test Problem 2
`hLen`	3	5
`nGene`	4	4
`nTL`	3	3
`nTrs`	5	5
`ILNonImp`	5	4
`DLNonImp`	5	5
`IntNonImp`	5	5
`FunCnt`	1000	1500

Table 3: Standard values of the parameters for Test Problems 1 and 2.

As a result, the most important parameters for the TP method are the number of genes (nGene) and the number of trials (nTrs). In particular, the number of genes must be chosen carefully, because if it is very small or very large, we will get bad results. In fact, when the parameter nGene is very large, the amount of computations will increase and the maximum number of computations allowed will be reached before getting the best solution.

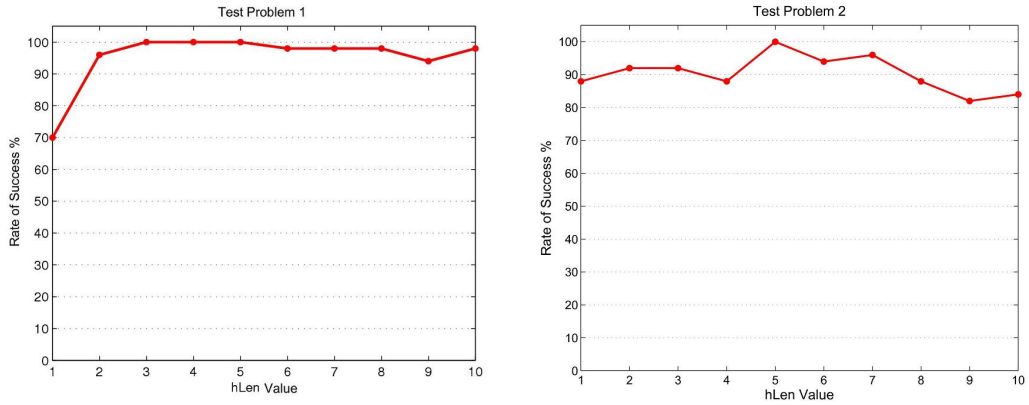

Figure 13: Rate of Success for 50 runs with different values of hLen for Test Problems 1 and 2.

6.1.3 Efficiency of Local Search in Tree Space

In this subsection we study the efficiency of the local search Procedures 3.1, 3.2 and 3.4. Here, we consider four cases to illustrate the importance and the effect of each procedure. First we consider the TP method with the intensive local search only. Second, we consider the TP method with the shaking procedure as an intensive local search and the grafting procedure as a diverse local search. Third, we consider the TP method with the shaking procedure as an intensive local search and the pruning procedure as a diverse local search. Lastly, we consider the standard method as described in Algorithm 5.1. We performed 50 runs for every case to compute the rate of success, and the results are displayed in Figure 21. Finally, one of these runs is shown in Figure 22 to illustrate the influence of the procedures on the performance of the method.

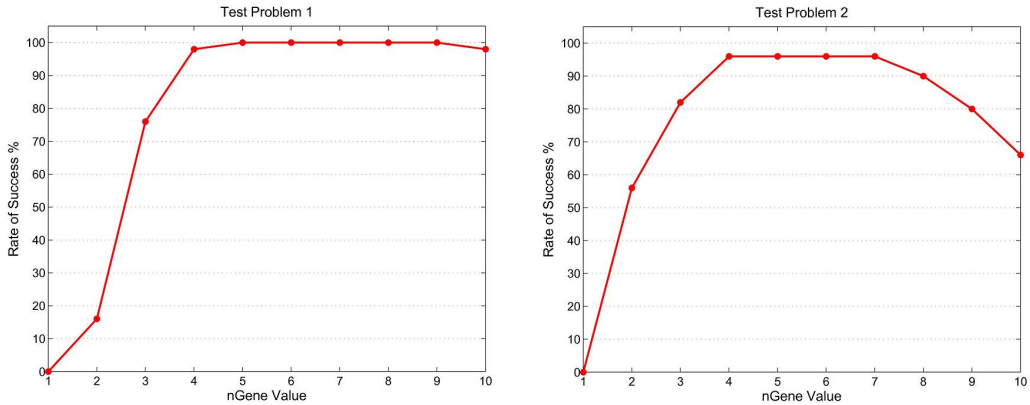

Figure 14: Rate of Success for 50 runs with different values of nGene for Test Problems 1 and 2.

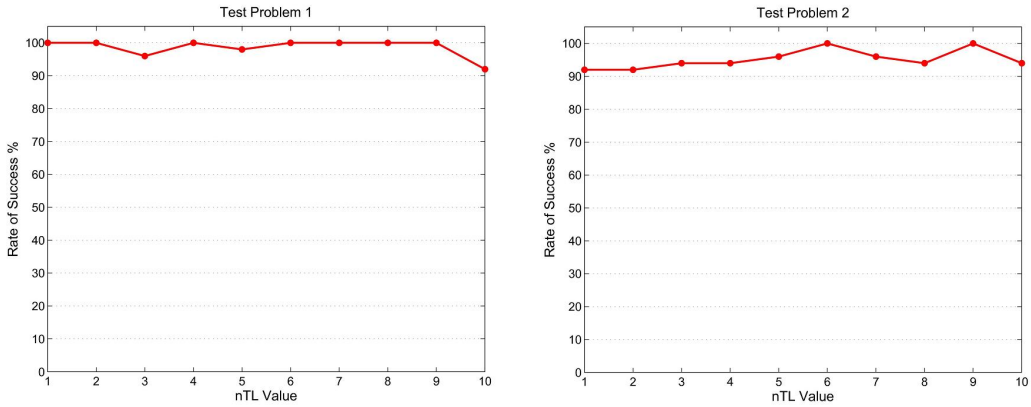

Figure 15: Rate of Success for 50 runs with different values of nTL for Test Problems 1 and 2.

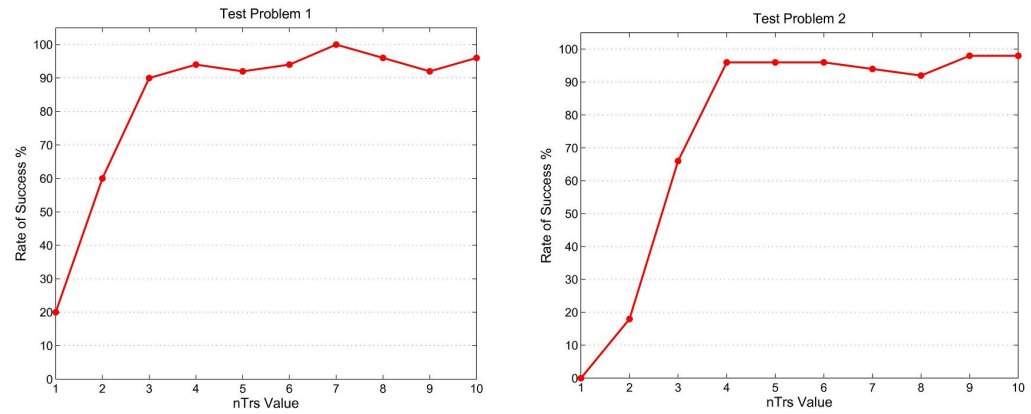

Figure 16: Rate of Success for 50 runs with different values of nTrs for Test Problems 1 and 2.

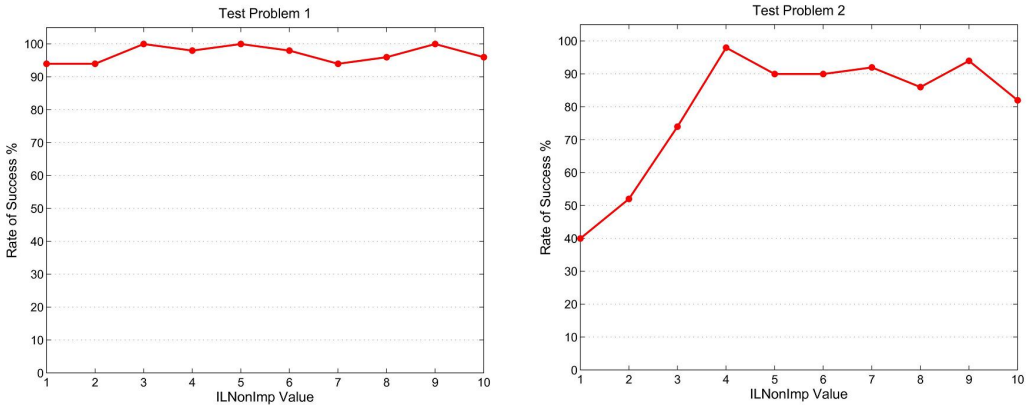

Figure 17: Rate of Success for 50 runs with different values of ILNonImp for Test Problems 1 and 2.

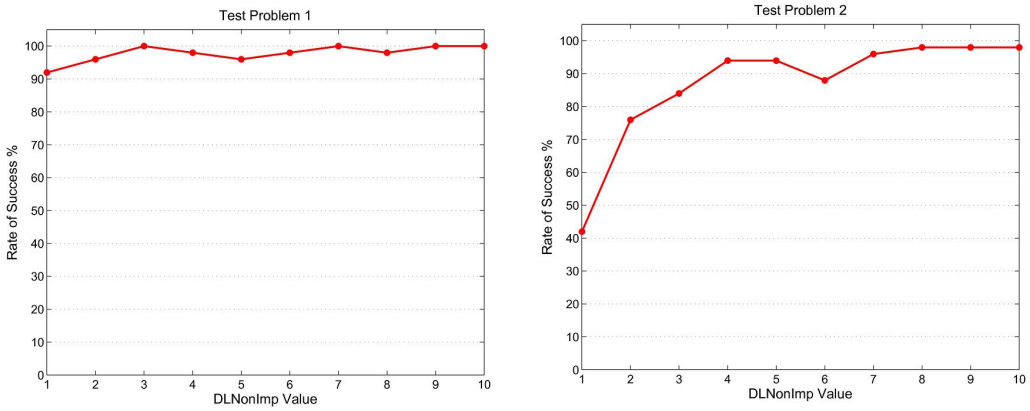

Figure 18: Rate of Success for 50 runs with different values of DLNonImp for Test Problems 1 and 2.

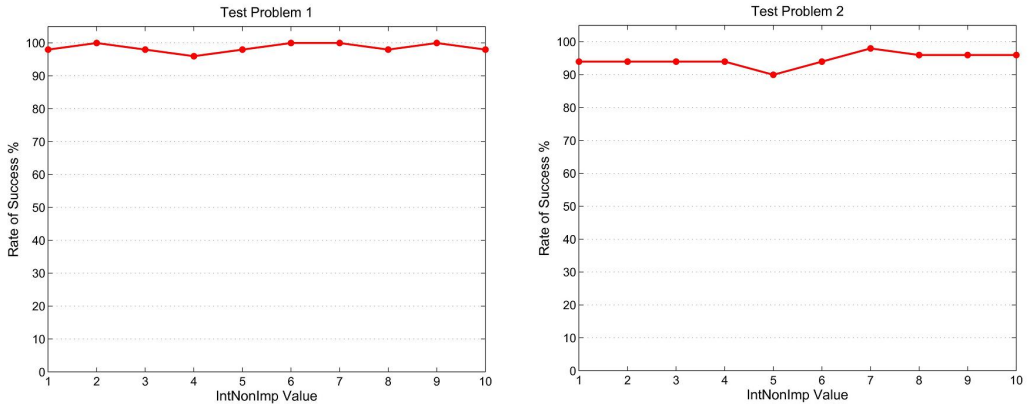

Figure 19: Rate of Success for 50 runs with different values of IntNonImp for Test Problems 1 and 2.

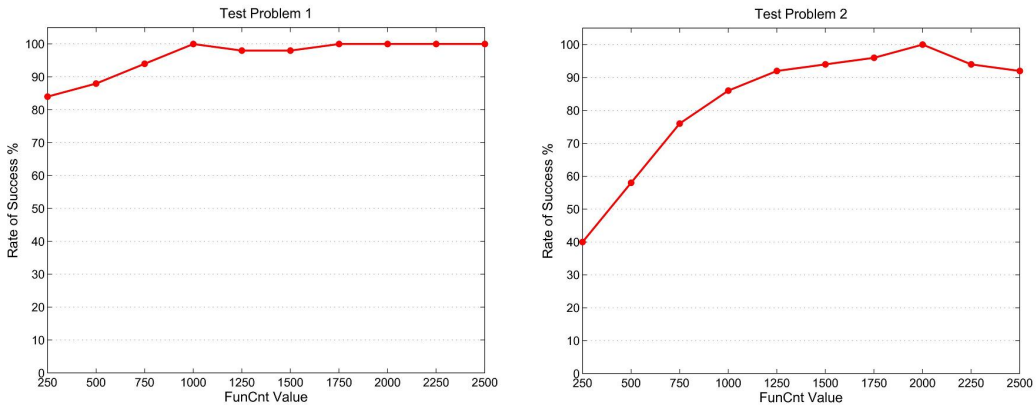

Figure 20: Rate of Success for 50 runs with different values of FunCnt for Test Problems 1 and 2.

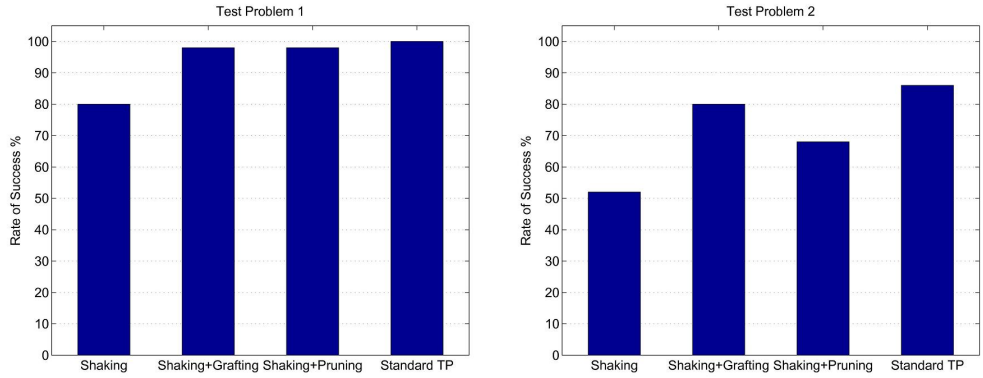

Figure 21: Rate of Success for 50 runs with different versions of TP method.

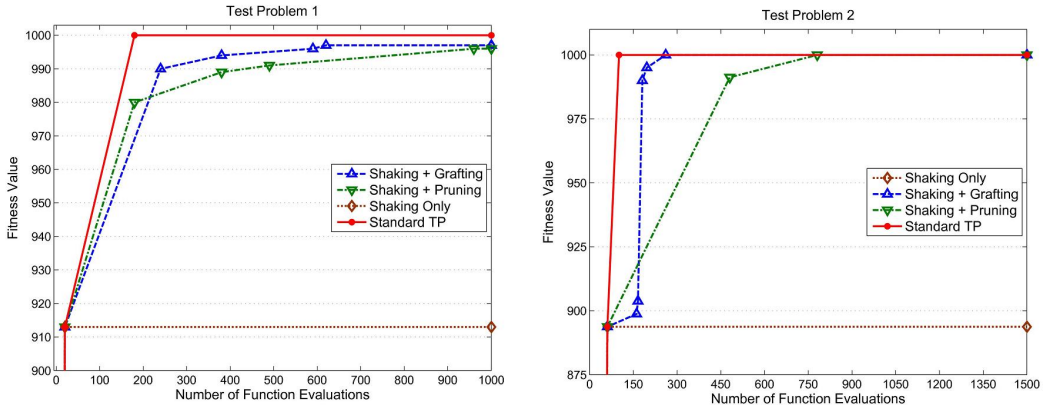

Figure 22: Comparison of four versions of TP method.

6.1.4 TP Method with Perturbed Data

To illustrate the stability of the TP method, we apply it to test problems with some perturbation in the function data. In fact, we simply modify the formula (5) for computing the fitness function as

$$F = \sum_{j=1}^{N} (100 - |g(x_j) - (1 + \varepsilon \ \mathbf{rand}\ ) y_j|), \tag{8}$$

where **rand** $\in [0, 1]$ is a random number and ε is a small positive number.

For each $\varepsilon = 0, 10^{-5}, 10^{-4}, 10^{-3}$ and 10^{-2}, we performed 50 runs and we chose the best value from these runs. The results are displayed in Figure 23.

For Test Problem 2, when we apply the perturbations with $\epsilon = 10^{-3}$ and 10^{-2}, we got new solutions, which means that the data change due to perturbation yielded another problems. In spite of that, the TP method is still successful to get correct formulas for the new data.

6.1.5 TP Method vs GEP Method

Here, we compare the proposed TP method with Gene Expression Programming (GEP) method, which is known as one of the most efficient modifications of the genetic programming method. For more details about GEP method, see [6, 7, 8]. In the comparison, the values of parameters are set as in Table 4 and the results are shown in Figure 24. It is clear from the figures that the TP method outperforms the GEP method, especially for the more complicated Test Problem 2.

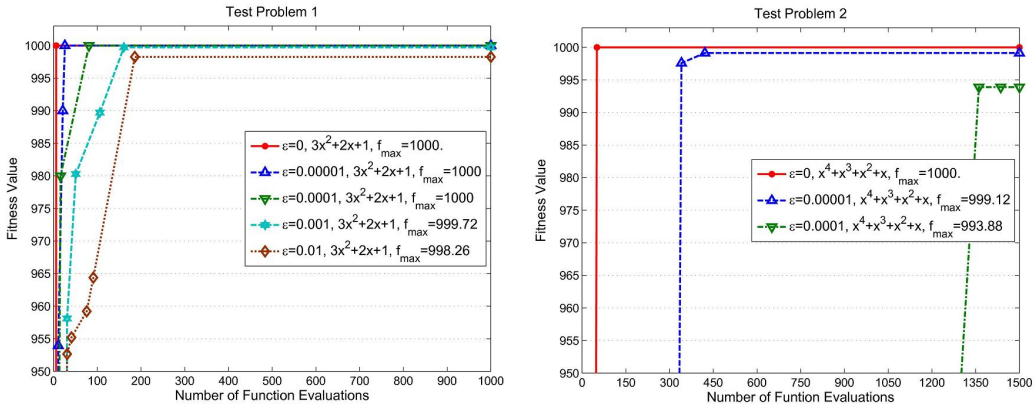

Figure 23: Results for some perturbed data.

Parameter	Value	
	Test Problem 1	Test Problem 2
hLen	3	7
nGene	3	4
nTL	3	3
nTrs	5	5
ILNonImp	5	5
DLNonImp	5	5
IntNonImp	5	5
FunCnt	1000	1500

Table 4: Values of the TP parameters used in the comparison with GEP method.

6.2 6-Bit Multiplexer Problem

The input to the Boolean N-bit multiplexer function consists of k "address" bits a_i and 2^k "data" bits d_i, and is a string of length $N = k + 2^k$ of the form $a_{k-1}, ..., a_1, a_0, d_{2^k-1}, ..., d_1, d_0$. In addition, the value of the N-multiplexer function is the value (0 or 1) of the particular data bit that is singled out by the k address bits of the multiplexer. For example, for the 6-bit multiplexer problem (where k = 2), if the two address bits a_1 and a_0 are 1 and 0, respectively then the multiplexer singles out the data bit 2 (i.e., d_2) to be its output.

In this subsection we will study the performance of the TP method applied to the 6-bit multiplexer problem with $k = 2$. Therefore, the Boolean 6-bit multiplexer is a function of 6 activities; two activities a_1, a_0 determine the address, and four activities d_3, d_2, d_1, d_0 determine the answer.

To apply the TP method to this problem, we use the following arguments.

1. The 6 activities $\{a_1, a_0, d_3, d_2, d_1, d_0\}$ as the set of terminals.

2. The Boolean function $\{\text{IF}\}$ as the set of functions. IF(x, y, z) will return the value y if x is true, and it will return the value z otherwise.

3. The fitness measure for the problem. There are $2^6 = 64$ possible combinations of the 6 activities $a_1, a_0, d_3, d_2, d_1, d_0$ along with the associated correct values of the 6-bit multiplexer function. Therefore, we will use the entire set of 64 combinations of activities as the fitness cases for evaluating the fitness [20]. The fitness value in this case will be the number of fitness cases where the Boolean value returned by the TP solution for a given combination of arguments is the correct Boolean value. Thus, the fitness value for this problem ranges between 0 and 64, where the fitness value of 64 means a 100%-correct solution.

4. The set of parameters have the values given in Table 5.

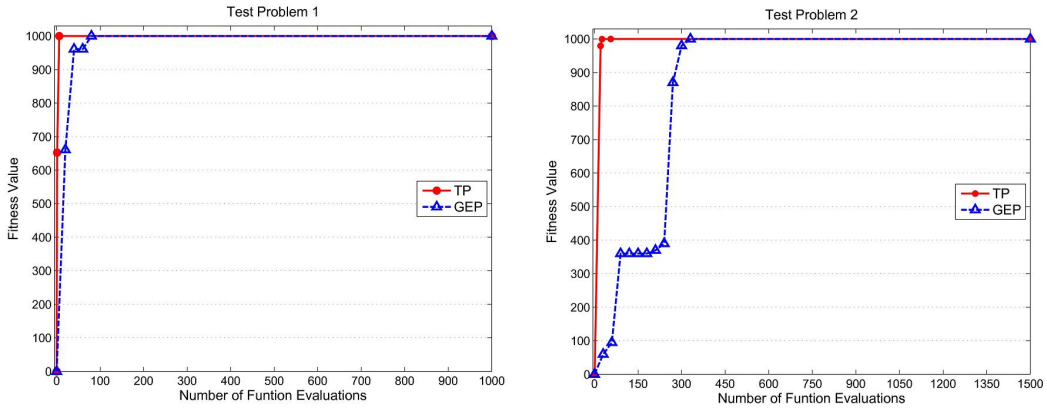

Figure 24: Comparison between TP method and GEP method for Test Problems 1 and 2.

For this test problem, we examine the parameter setting. Fortunately, we find that the TP method gets a 100%-correct solution for all values of the parameters given in Table 5.

Parameter	Value	
	Set of Parameters	100%-correct Solution
hLen	3	≥ 3
nGene	3	3, 5 Or 7
nTL	3	≥ 3
nTrs	5	≥ 5
ILNonImp	15	≥ 15
DLNonImp	15	≥ 15
IntNonImp	15	≥ 15
FunCnt	15000	≥ 15000

Table 5: Set of parameters for our results and the 100%-correct solution parameters.

We apply the TP method to the 6-bit multiplexer problem and make 50 runs. In Figure 25, we show the results of two runs as sample results. In addition, the final solutions for these two samples are shown in Figures 26. Note that, these are the final formulas, and we get them by applying some editing operations to the original ones, so that the original formulas and the previous formulas are logically equivalent.

By comparing our results with those of GP [1] and GEP [6] for this problem, we may conclude that the TP method is promising since it converges to a 100% correct solution rapidly and saves a lot of computational time.

7 Concluding Remarks and Future Work

Genetic Programming is one of the powerful tools of the artificial intelligence methods. It searches a solution space of computer programs which can be represented as a parse tree. However, it has been addressed that crossover and mutation in GP are highly disruptive with the risk of non-convergence to an optimal structure. Therefore, this work has introduced some local search procedures over a tree space as alternative operations to crossover and mutation.

The proposed local searches have two aspects; intensive and diverse. Intensive local search aims to explore the neighborhood of a tree by altering its nodes without changing its structure. Diverse local search changes the structure of a tree by expanding its terminal nodes or cutting its subtrees. Using these search procedures, various meta-heuristics can be generalized to deal with tree data structures in a unified framework which we call Meta-Heuristics Programming (MHP).

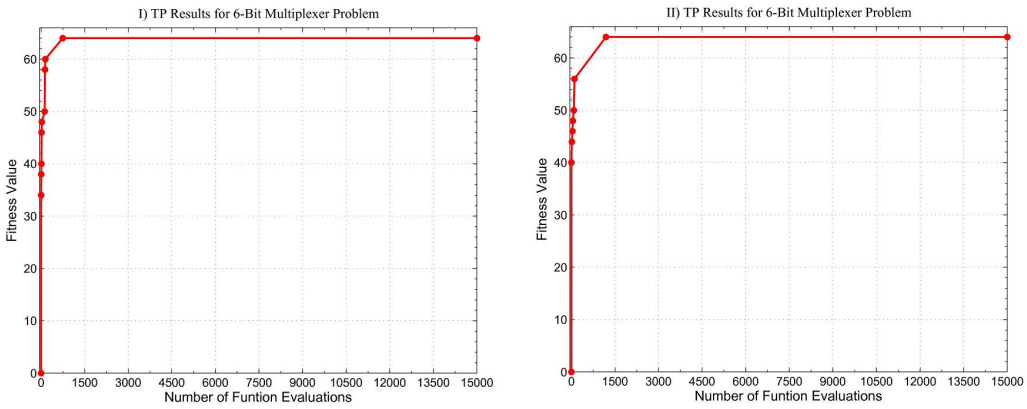

Figure 25: Sample results for 6-bit multiplexer problem.

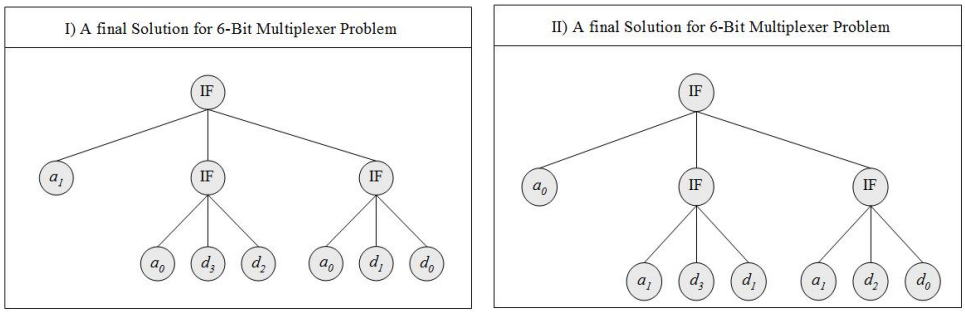

Figure 26: Final solution for TP method for 6-bit multiplexer problem.

As a special case of MHP framework, we have introduced the Tabu Programming (TP) method as a modification to the Tabu Search method which belongs to point-to-point methods. Finally, we have tested the performance of the TP method for two types of benchmark problems and made some experiments to analyze the main components of TP method. From these numerical experiments, we have shown that TP method is effective and stable, compare with the GEP method.

As a future work, we may introduce more specific classes of the MHP framework. For example, we may modify the GP method to get a new method called the Memetic Programming method that belongs to population-based methods. Also, we may develop the Annealing Programming method by applying our local search procedures over a tree space to the classical Simulated Annealing method which belongs to point-to-point methods.

References

[1] R.M.A. Azad and C. Ryan, An Examination of Simultaneous Evolution of Grammars and Solutions, in: *Genetic Programming: Theory and Practice III*, T. Yu, R.L. Riolo and B. Worzel (Eds.), Springer-Verlag, 9, 141–158, 2006.

[2] M. Boryczka, Eliminating Introns in Ant Colony Programming, *Fundamenta Informaticae*, 68, 1–19, 2005.

[3] M. Boryczka, Z.J. Czech, and W. Wieczorek, Ant Colony Programming for Approximation Problems, in: *GECCO 2003*, E. Cantú et al. (Eds.), Springer, 142-143, 2003.

[4] R. Diestel, *Graph Theory*, Springer-Verlag, Berlin, 2005.

[5] A.E. Eiben and J.E. Smith, *Introduction to Evolutionary Computing*, Springer-Verlag, Berlin, 2003.

[6] C. Ferreira, Gene Expression Programming: A New Adaptive Algorithm for Solving Problems, *Complex Systems*, 13, 87–129, 2001.

[7] C. Ferreira, Gene Expression Programming in Problem Solving, in: *Soft Computing and Industry Recent Applications*, R. Roy et al. (Eds.), Springer-Verlag, 635–654, 2002.

[8] C. Ferreira. *Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence*, Springer-Verlag, Berlin, 2006.

[9] F. Glover, Future Paths for Integer Programming and Links to Artificial Intelligence, *Computers and Operations Research*, 13, 533–549, 1986.

[10] F. Glover and G. Kochenberger (Eds.), *Handbook of MetaHeuristics*, Kluwer Academic Publishers, Boston, MA, 2002.

[11] F. Glover and M. Laguna, *Tabu Search*, Kluwer Academic Publishers, Boston, MA, 1997.

[12] F. Glover and M. Laguna, Tabu Search, in: *Handbook of Applied Optimization*, P.M. Pardalos and M.G.C. Resende (Eds.), Oxford University Press, 194–208, 2002.

[13] F. Glover, E. Taillard and D. Werra, A User's Guide to Tabu Search, *Annals of Operations Research*, 41, 3–28, 1993.

[14] D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, 1989.

[15] A. Hedar and M. Fukushima, Hybrid Simulated Annealing and Direct Search Method for Nonlinear Unconstrained Global Optimization, *Optimization Methods and Software*, 17, 891–912, 2002.

[16] A. Hedar and M. Fukushima, Heuristic Pattern Search and its Hybridization with Simulated Annealing for Nonlinear Global Optimization, *Optimization Methods and Software*, 19, 291-308, 2004.

[17] A. Hedar, and M. Fukushima, Tabu Search Directed by Direct Search Methods for Nonlinear Global Optimization. *European Journal of Operational Research*, 170, 329–349, 2006.

[18] A. Hedar, and M. Fukushima. Derivative-Free Filter Simulated Annealing Method for Constrained Continuous Global Optimization. *Journal of Global Optimization*, to appear.

[19] T.H. Hoang, X. Nguyen, R.B. McKay and D. Essam, The Importance of Local Search: A Grammar Based Approach to Environmental Time Series Modelling, *Genetic Programming: Theory and Practice III*, T. Yu, R.L. Riolo and B. Worzel (Eds.), Springer-Verlag, 9, 159–175, 2006.

[20] J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, MA, 1992.

[21] J.R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press, Cambridge, MA, 1994.

[22] J.R. Koza, F.H. Bennett III, D. Andre, and M.A. Keane, *Genetic Programming III: Darwinian Invention and Problem Solving*, Morgan Kaufmann, San Francisco, CA, 1999.

[23] J.R. Koza, M.A. Keane, M.J. Streeter, W. Mydlowec, J. Yu, and G. Lanza, *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*, Kluwer Academic Publishers, Boston, 2003.

[24] M. Laguna and R. Martí, *Scatter Search: Methodology and Implementations in C*, Kluwer Academic Publishers, Boston, 2003.

[25] E. Morsy and H. Nagamochi, Approximating Capacitated Tree-Routings in Networks, in: *TAMC 2007*, 342–353, 2007.

[26] P. Moscato, Memetic Algorithms: An Introduction, in: *New Ideas in Optimization*, D. Corne, M.Dorigo and F. Glover (Eds.), McGraw-Hill, London, UK, 1999.

[27] P. Nordin and W. Banzhaf, Complexity Compression and Evolution, in: *ICGA 1995*, L. Eshelman (Ed.), Morgan Kaufmann, Pittsburgh, PA, USA, 310–317, 1995.

[28] P. Nordin, F. Francone and W. Banzhaf, Explicitly Defined Introns and Destructive Crossover in Genetic Programming, in: *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, J.P. Rosca (Ed.), Tahoe City, California, USA, 6–22, 1995.

[29] C.C. Ribeiro and P. Hansen (Eds.), *Essays and Surveys in Metaheuristics*, Kluwer Academic Publishers, Boston, MA, 2002.