International Journal of Information Technology & Decision Making
© World Scientific Publishing Company

TABU PROGRAMMING: A NEW PROBLEM SOLVER THROUGH ADAPTIVE MEMORY PROGRAMMING OVER TREE DATA STRUCTURES

ABDEL-RAHMAN HEDAR

Department of Computer Science,
Faculty of Computers and Information,
Assiut University, EGYPT
hedar@aun.edu.eg

EMAD MABROUK* and MASAO FUKUSHIMA†

Department of Applied Mathematics and Physics,
Graduate School of Informatics,
Kyoto University, Kyoto 606-8501, JAPAN
**hamdy@amp.i.kyoto-u.ac.jp*
†fuku@i.kyoto-u.ac.jp

Received Day Month Year
Revised Day Month Year

Since the first appearance of the Genetic Programming (GP) algorithm, extensive theoretical and application studies on it have been conducted. Nowadays, the GP algorithm is considered one of the most important tools in Artificial Intelligence (AI). Nevertheless, several questions have been raised about the complexity of the GP algorithm and the disruption effect of the crossover and mutation operators. In this paper, the Tabu Programming (TP) algorithm is proposed to employ the search strategy of the classical Tabu Search algorithm with the tree data structure. Moreover, the TP algorithm exploits a set of local search procedures over a tree space in order to mitigate the drawbacks of the crossover and mutation operators. Extensive numerical experiments are performed to study the performance of the proposed algorithm for a set of benchmark problems. The results of those experiments show that the TP algorithm compares favorably to recent versions of the GP algorithm in terms of computational efforts and the rate of success. Finally, we present a comprehensive framework called Meta-Heuristics Programming (MHP) as general machine learning tools.

Keywords: Machine Learning; Meta-heuristics; Local Search; Parse Tree; Tabu Programming; Tabu Search.

1. Introduction

Many different problems from artificial intelligence, symbolic processing, and machine learning require discovery of a "computer program" as an output when presented with particular inputs. Koza[1,2] introduces a lot of problems that can be viewed as problems of discovering desirable computer programs. Many research ar-

eas contain such problems as machine learning, planning in artificial intelligence and robotics, symbolic regression, concept formation, game playing, neural network design and training, etc. The process of solving those problems can be reformulated as a search for the fittest individual computer program in the space of possible computer programs for the problem under consideration.

The purpose of this paper is to propose a new algorithm called Tabu Programming (TP) algorithm that searches for desirable computer program as an output. The proposed algorithm follows the search strategy in the well-known Tabu Search (TS) algorithm using a different data structure. Specifically, the TP algorithm deals with working computer programs represented as a tree data structure. Therefore, the main contribution of the proposed algorithm is to design more alternatives to the Genetic Programming (GP) algorithm in order to accommodate more application areas. Because of the good performance of the TS algorithm compared to the Genetic Algorithms (GAs) in some implementations,[3,4] we expect that the TP algorithm also has a chance to outperform the GP algorithm for certain problems.

Genetic Algorithms (GAs) are search algorithms inspired from the biological processes of natural selection and survival of the fittest. GAs have been widely studied, experimented and applied in many fields through a huge amount of publications.[5,6,7] In addition, GAs have been shown to outperform other traditional methods in various sets of applications.[8,9,10,11] On the other hand, the Genetic Programming (GP) algorithm inherits the basic idea of GAs and deals with working computer programs obtained from a given problem. The first proposal of "tree-based" genetic programming was given by Cramer in 1985.[12] This work was popularized by Koza[1,2,13,14], and subsequently, the feasibility of this approach in well-known application areas has been demonstrated.[15,16,17]

The main difference between GAs and GP lies in the representation of a solution and the application fields. GAs usually use an array of bits to represent a solution,[7,9] while GP deals with computer programs represented as trees, in which leaf nodes are called terminals and internal nodes are called functions.[13,14,15,16,17] Actually, the tree-based representation of a solution is a great advantage of GP and makes it possible to cover a wide area of applications. The GP algorithm starts a search with a "population" of computer programs that are usually generated randomly. Then, the algorithm selects a portion of the current population based on their fitness to breed a new generation using the crossover and mutation operators. This process is iterated with the hope of driving the population toward an optimal solution. It is worthwhile to mention that although there are many metaheuristic alternatives to GAs, GP is almost the only metaheuristics adapted for searching the space of computer programs.

The crossover and mutation operators are the main breeding operators in GP. Indeed, the crossover and mutation operators have extensively been studied. Many effective settings of these operations have been proposed to deal with a wide variety of problems. One of the early GP crossover operators is introduced in Ref. 1. However, some bad performance of the crossover operators are reported in Refs. 18,

19. Specifically, it was argued that trees tend to grow in size over the generations, causing the crossover operation to be computationally expensive and yielding the program bloat problem, see Ref. 20 and references therein. Actually, this problem produces a high computational cost in GP due to the growth of individuals in size and complexity during the evolution process.[21] Moreover, Ref. 22 shows that 75% of crossover events can result in disruption of building blocks. It has also been addressed that crossover and mutation are highly disruptive with the risk of convergence to a non-optimal solution.[18,19,23,24,25] Altering a node high in the tree may result in serious disruption of the subtree below it. This motivates some researchers to propose some hypotheses about the causes behind this phenomenon. One of these hypotheses is "Protective effect against destructive nature of modifying operators", see Refs. 26, 27 for more details. Since these operators are the essential solution generation operators in GP, there have been many attempts to edit them to make changes in small scales, for example by using natural language processing.[16,24] Using new crossover operators, such as brood crossover,[28,29] context-aware crossover,[30,31] homologous crossover and crossover-hill climbing,[22] ripple crossover,[32] and depth-fair crossover,[33] were practical remedies for these problems. Moreover, the importance of local search and improving the local structure of individuals have been addressed.[34,35]

This motivates us to try to invoke the use of the local search in new fashions in order to provide an alternative remedy for the solution generation process. Specifically, we use new local search procedures over a tree space as alternative operations to crossover and mutation. These procedures aim to generate a set of trial programs in the neighborhood of a program by making moderate changes in it. We will show through extensive numerical experiments that the procedures in the TP algorithm work effectively and, in fact, the TP algorithm outperforms the GP algorithm on some benchmark test problems. From the numerical experiments, we may expect that TP provides a promising approach for problem solving.

The rest of the paper is organized as follows. Section 2 recalls the Tabu Search algorithm and its procedures. Basic procedures for stochastic local search over a tree space are described in Section 3. After presenting the TP algorithm in Section 4, we report numerical results in Section 5 for three types of benchmark problems; the symbolic regression problem, the 6-bit multiplexer problem, and the 3-bit even-parity problem. A comprehensive framework called the Meta-Heuristics Programming framework is presented as a generalization of the TP algorithm in Section 6. Finally, concluding remarks make up Section 7.

2. Tabu Search

Tabu Search (TS) is a meta-heuristic method originally proposed by Glover[36] in 1986. Afterward, it has spread quickly to become one of the most powerful meta-heuristic methods for tackling difficult combinatorial optimization problems.[37,38,39] In particular, TS improves an ordinary local search method by accepting uphill

movements, and storing attributes of the recently visited solutions in a short-term memory called tabu list (TL). This strategy enables TS to avoid getting trapped in local minima and prevent cycling over a sequence of non-optimal solutions. In several cases, TS provides solutions very close to optimality with reasonable computing time. In addition, the high efficiency of TS has captured the attention of many researchers. Several papers, book chapters, special issues and monographs have surveyed the rich TS literature.[3,4,37,38,39,40,41]

Suppose that the goal is to minimize the objective function $f : S \rightarrow \mathbb{R}$ over all $s \in S$, where each solution s has an associated neighborhood $N(s) \subset S$. A simple TS algorithm with short-term memory begins with an initial solution s chosen, usually randomly, from the feasible set S. The best non-tabu solution in the neighborhood $N(s)$ replaces the current one s, and its attributes will be stored in the TL. However, an aspiration criterion may be applied to accept a tabu solution if it is better than the best solution found so far. These steps are repeated with new solutions until some termination conditions are satisfied (e.g., reaching the maximum number of iterations). The best solution found during the search process is designated as a solution of the problem.

The short-term memory TL is built to keep the recency only. In order to achieve better performance, a long-term memory has been proposed to keep more important search features, such as the quality and the frequency, along with the recency. Specifically, the long-term memory in a more advanced TS records attributes of special characteristics of a solution or a move.[37,38,39,40,41] In this case, the search process of TS can adapt itself using intensification and diversification strategies. The purpose of the diversification strategy is to allow the algorithm to guide the search to new areas of the search space. However, the intensification strategy enables the algorithm to return to attractive regions, and performing a thorough search around elite solutions in order to obtain much better solutions in their vicinities.

During the search process in the TS algorithm, the best solution that replaces the current one is chosen from a modified neighborhood called $\widetilde{N}(s)$, where the structure of $\widetilde{N}(s)$ mainly depends on the history of the search process.[41] In case of using TS based on a short-term memory, the modified neighborhood $\widetilde{N}(s)$ is a subset of the ordinary neighborhood $N(s)$, where TL and aspiration criteria are used to recognize solutions that belong to $N(s)$ and are excluded from $\widetilde{N}(s)$. However, in a more advanced TS with short-term and long-term memories, the modified neighborhood $\widetilde{N}(s)$ may be expanded to include some solutions that do not ordinarily exist in $N(s)$. For example, $\widetilde{N}(s)$ may contain the high quality neighbors of elite solutions in the attractive regions to be used if the intensification strategy is needed. It is worthwhile to note that the modified neighborhood $\widetilde{N}(s)$ of a solution s depends on the history of the search.

3. Local Searches over Tree Space

As mentioned in the introduction, GP deals with computer programs as solutions to a problem. Those computer programs can be represented as parse trees[a], 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, those codes are expressed to closely match the Polish notation of logic expressions.[42] Fig. 1 shows three examples of a tree representation of individuals and their executable codes below these trees. These codes are geometrically illustrated in Fig. 2 as solid lines or curves. Then, a fitness function is 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 on the given dataset.

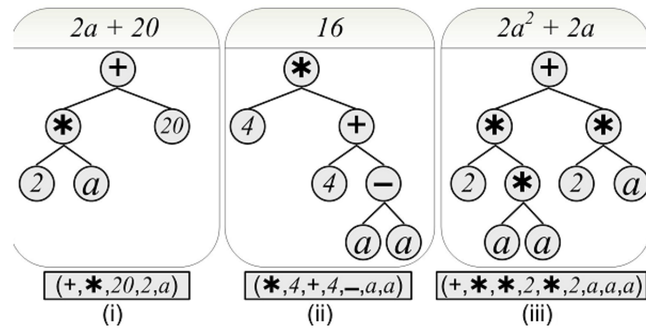

Fig. 1. Examples of GP representation.

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; *static structure search* and *dynamic structure search*. Static structure search aims to explore the neighborhood of a tree by altering its nodes without changing its structure. Dynamic structure search changes the structure of a tree by expanding its terminal nodes or cutting its subtrees[b]. We introduce *Shaking* as a static structure search procedure, and *Grafting* and *Pruning* as dynamic structure search procedures.

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

- *Tree Size* $|X|$ is the number of nodes in tree X.

[a]A 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 leaf nodes represent terminals symbols.
[b]Throughout the paper, the term "*branch*" is used to refer to a subtree, see Ref. 43.

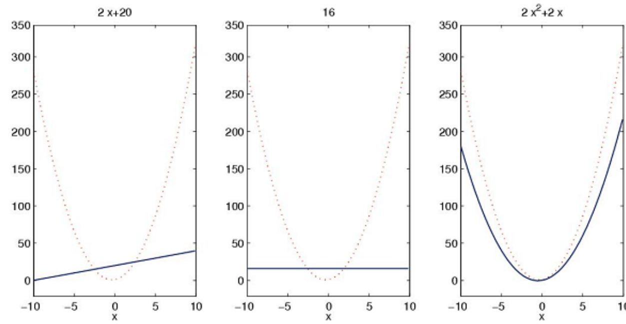

Fig. 2. Geometrical illustration of Examples of Fig. 1.

- *Tree Leaf Number* $l(X)$ 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.

3.1. *Shaking search*

Shaking search is a static structure search procedure that generates a tree $\tilde{X}$ from a tree X by replacing the terminals or functions at some of its nodes by alternative ones without changing its structure, i.e., an altered terminal node is replaced 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 Fig. 3 shows an example of shaking search that alters two nodes of X. In Procedure 3.1, $\nu \in [1, |X|]$ is a positive integer that represents the number of nodes to be changed, and ν must be determined before calling the procedure.

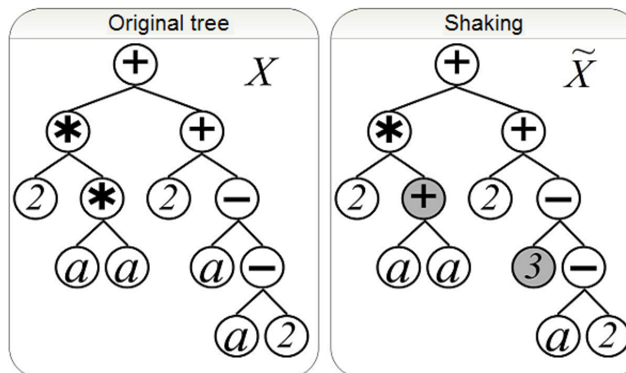

Fig. 3. Example of shaking search ($\nu = 2$).

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

1. If $\nu > |X|$, return.
2. Set $\tilde{X} := X$.
3. Repeat the following Steps 3.1 - 3.3 for $j = 1, \ldots, \nu$.
   - 3.1 Choose a node (function or terminal) from $\tilde{X}$ randomly.
   - 3.2 Generate an alternative randomly from the set of functions and terminals.
   - 3.3 Update $\tilde{X}$ by replacing the chosen node by the new alternative one.
4. Return.

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

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

Therefore, a modified neighborhood $\widetilde{N}_S(X)$ of a tree X, associated with shaking search, can be defined as

$$\widetilde{N}_S(X) = \{\tilde{X} | \tilde{X} \in N_S(X), \tilde{X} \text{ is not tabu or it satisfies an aspiration criterion}\}. \tag{3.1}$$

It is worthwhile to note that the random choices of Steps 3.1 and 3.2 of Procedure 3.1 make the shaking procedure behave as stochastic search. Therefore, for a tree X, one may get a different $\tilde{X}$ after 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 dynamic structure 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 $|\tilde{X}| > |X|$, $l(\tilde{X}) > l(X)$, and $d(\tilde{X}) \geq d(X)$. Procedure 3.2 states the formal description of grafting search, where λ refers to the number of branches of depth[c] ζ that will be added to X. In addition, λ and ζ must be determined before calling the procedure. Figure 4 shows an example of grafting search that alters two nodes in X by two branches in $\tilde{X}$.

Procedure 3.2. $\tilde{X} = \texttt{Grafting}(X, \lambda, \zeta)$

1. Set $\tilde{X} := X$.
2. Repeat the following Steps 2.1 - 2.3 for $j = 1, \ldots, \lambda$.
   - 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.

[c]The depth of a branch B has the same definition as the depth $d(X)$ of a tree X, and will also be denoted by $d(B)$.

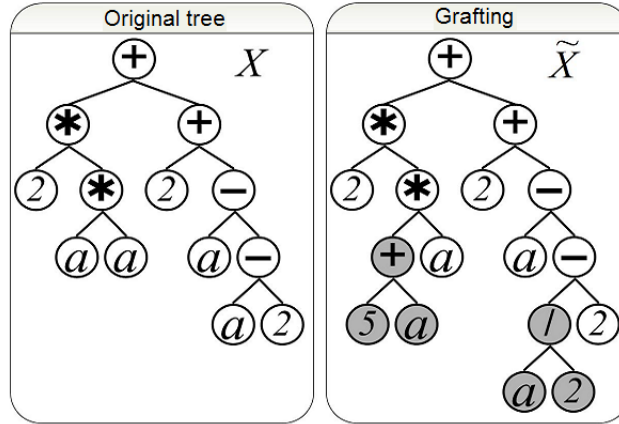

Fig. 4. Example of grafting search ($\lambda = 2$ and $\zeta = 1$).

3. Return.

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

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

where ζ_{max} is a predetermined positive integer. Therefore, a modified neighborhood $\widetilde{N}_G(X)$ of a tree X, associated with grafting search, can be defined as

$$\widetilde{N}_G(X) = \{\tilde{X} | \tilde{X} \in N_G(X), \tilde{X} \text{ is not tabu or it satisfies an aspiration criterion}\}. \tag{3.2}$$

Note that the random choices of Steps 2.1 and 2.2 of Procedure 3.2 also make the grafting procedure behave as stochastic search. Therefore, for a tree X, one may get a different $\tilde{X}$ after each run of the procedure.

3.3. *Pruning search*

Pruning search is another dynamic structure search procedure. In contrast with 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 $|\tilde{X}| < |X|$, $l(\tilde{X}) < l(X)$, 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. Those 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 $[B_1, \ldots, B_\xi] = \text{Branches}(X, \zeta)$, where $d(B_1) = \cdots = d(B_\xi) = \zeta$. In addition, as in the grafting procedure, the depth ζ of the branches is chosen depending on the use of the pruning procedure. If it works as an intensification procedure, then ζ is chosen to be small, preferably 1. But, in the case of using the pruning procedure as a diversification procedure, ζ is chosen to be large.

For instance, if pruning search is used to cut a branch of depth $\zeta = 1$ in tree X, then the branch coding procedure is called to express each branch of depth $\zeta = 1$ in X as a meta-terminal node as shown in Fig. 5. Hence, pruning search can easily choose one of these branches and replace it by a randomly generated leaf node.

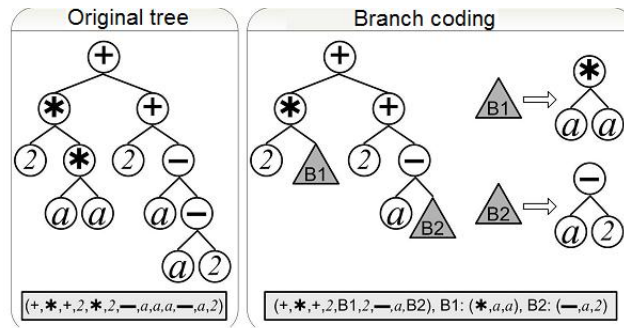

Fig. 5. Example of branch coding ($\xi = 2, \zeta = 1$).

Procedure 3.3. $[B_1, \ldots, B_\xi] = \texttt{Branches}(X, \zeta)$

1. If $\zeta \geq d(X)$, return.
2. Select all branches $B_1, \ldots, B_\xi$ in X with depth ζ.
3. Return.

The formal description of pruning search is given below in Procedure 3.4. Fig. 6 shows an example of pruning search that cuts two branches in X. In Procedure 3.4, η is a positive integer that represents the number of branches replaced by a leaf node during pruning search.

Procedure 3.4. $\tilde{X} = \texttt{Pruning}(X, \eta, \zeta)$

1. Set $\tilde{X} := X$ and set the counter $j = 1$.
2. While $\zeta \leq d(\tilde{X})$ and $j \leq \eta$, repeat Steps 2.1 - 2.3
   2.1 Use Procedure 3.3 to get $[B_1, \ldots, B_{\xi_j}] := \text{Branches}(\tilde{X}, \zeta)$.
   2.2 Choose a terminal node t_j randomly from the set of terminals.
   2.3 Update $\tilde{X}$ by replacing a randomly chosen branch from $\{B_1, \ldots, B_{\xi_j}\}$ by t_j and set $j = j + 1$.
3. Return.

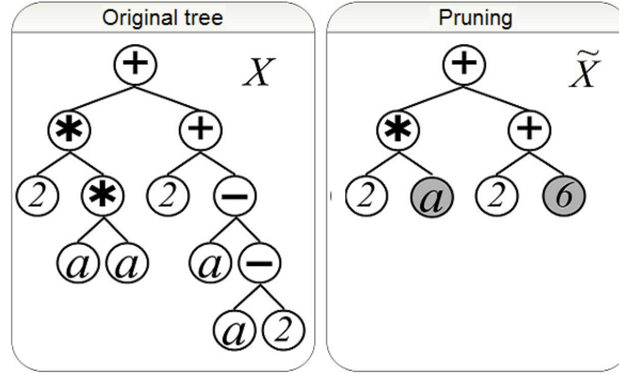

Fig. 6. Example of pruning search ($\eta = 2, \zeta_1 = 1, \zeta_2 = 2$).

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

$$N_P(X) = \{\tilde{X} | \tilde{X} = \texttt{Pruning}(X, \eta, \zeta), \eta = 1, \ldots, |X| - l(X), \zeta = 1, \ldots, d(X)\}.$$

Therefore, a modified neighborhood $\widetilde{N}_P(X)$ of a tree X, associated with pruning search, can be defined as

$$\widetilde{N}_P(X) = \{\tilde{X} | \tilde{X} \in N_P(X), \tilde{X} \text{ is not tabu or it satisfies an aspiration criterion}\}. \tag{3.3}$$

The random choices of Steps 2.1 and 2.2 of Procedure 3.4 also make the pruning procedure behave as stochastic search. Therefore, for a tree X, one may get a different $\tilde{X}$ after each run of the procedure.

The values of ν, λ, η and ζ in the proposed local search procedures, Procedures 3.1, 3.2 and 3.4, must be determined before calling these procedures. In particular, one can choose the values of ν, λ and η as random integers, based on the numbers of functions and terminals inside the tree X. For example, the value of ν can be chosen as a random integer between 1 and $|X|$. However in this paper, we choose the values of ν, λ and η based on the number of trials that we wish to generate. For example, to generate three trial solutions using the shaking procedure, we call the shaking procedure three times with $\nu = 1$, $\nu = 2$ and $\nu = 3$, respectively. In other words, the first trial solution ($\nu = 1$) is generated by choosing one node randomly and replacing it by an alternative node, while the second trial solution ($\nu = 2$) is generated by choosing two nodes randomly and replacing them by two alternative nodes, and so on. More details will be described later in Section 4.

On the other hand, the value of ζ may depend on the mission of the search process. Specifically, during the ordinary search process, the local search procedures should be applied with a small scale of change to avoid the disruption of the current solution. However, those local search procedures should be applied with a bigger scale of change if diversification is needed. Actually, keeping diversity is one of the

major issues that should be taken into account in designing efficient global search techniques.[44]

4. Tabu Programming

The TP algorithm is a modified version of the Tabu Search algorithm that uses tree-based representations and different neighborhood structures. In particular, every solution generated by the TP algorithm is a computer program represented by a tree consisting of terminals and functions. Therefore, the search space of the TP algorithm is the set of all computer programs that can be represented as trees. In addition, neighborhoods of a solution X should be generated by using the local search procedures introduced in Section 3.

In fact, to the best of the authors' knowledge, there have been just a few attempts to extend other meta-heuristics to deal with tree data structures. Specifically, ant colony programming[45,46] is a generalized meta-heuristic of the ant colony optimization algorithm. In the literature, however, there is no explanation about the performance of ant colony programming in comparison with the GP algorithm. The first idea of the TP algorithm was proposed by Hedar and Fukushima,[47] in a short paper presented in a 2006 workshop. Then, in 2007, Balicki[48] discussed TP as an extension of TS to deal with tree representations using a special set of procedures. However, Balicki's algorithm uses operations similar to the mutation in GP and does not pay much attention to the drawbacks of the mutation procedure. In this paper, we introduce a more advanced version of the TP algorithm and its implementation for problem solving.

The proposed TP algorithm invokes three basic search stages; *local search*, *diversification* and *intensification*. In the local search stage, the TP algorithm uses two types of local searches; static structure search to make good exploration around the current solution, and dynamic structure search to accelerate the search process if successive non-improvements face the static structure search. Static structure search aims to explore the neighborhood of a current solution X_k by altering its nodes without changing its structure through *shaking* search. In addition, dynamic structure search tries to change locally the tree structure of X_k through *grafting* and *pruning* searches using branches of small depth. Then, the DIVERSIFICATION procedure is applied (if needed) in order to diversify the search for new tree structures. Finally, in order to explore close tree structures around the best programs visited so far, the INTENSIFICATION procedure is applied to improve these best programs further. Figure 7 shows the main structure of the TP algorithm, and its formal description is given below.

Algorithm 4.1. (TP Algorithm)

1. **Initialization.** Choose an initial program X_0, set the tabu list `TL` and other memory elements empty, and set the counter $k := 0$. Choose the values of $\mathtt{nTL}, n_T, n'_T, n''_T, \zeta_1, \zeta_2$ and n^*.

2. **Main Loop.** Repeat the following Steps 2.1 - 2.3 until the non-improvement condition for the main loop is satisfied.
   - 2.1 **Static Structure Search.** Repeat the following Steps 2.1.1 - 2.1.3 until a non-improvement condition for the static structure search is satisfied. If the condition is satisfied, proceed to Step 2.2.
     - 2.1.1 Generate a set of n_T trial programs S_k from the neighborhood $\widetilde{N}_S(X_k) \subset N_S(X_k)$, Equation (3.1), based on the tabu restrictions and an aspiration criterion.
     - 2.1.2 Choose the best program in S_k and denote it by X_{k+1}.
     - 2.1.3 Add X_{k+1} to the `TL` and remove the oldest program in it. Update other memory elements and set $k := k + 1$.
   - 2.2 **Dynamic Structure Search.** Do the following Steps 2.2.1 - 2.2.4.
     - 2.2.1 Generate a set of n'_T trial programs S'_k from the neighborhood $\widetilde{N}_G(X_k) \subset N_G(X_k)$, Equation (3.2), based on the tabu restrictions and an aspiration criterion.
     - 2.2.2 Generate a set of n''_T trial programs S''_k from the neighborhood $\widetilde{N}_P(X_k) \subset N_P(X_k)$, Equation (3.3), based on the tabu restrictions and an aspiration criterion.
     - 2.2.3 Choose the best program in $S'_k \cup S''_k$ and denote it by X_{k+1}.
     - 2.2.4 Add X_{k+1} to the `TL` and remove the oldest program in it. Update other memory elements and set $k := k + 1$.
   - 2.3 **Check for the non-improvement.** If a non-improvement condition is satisfied, go to Step 3. Otherwise, return to Step 2.1.
3. **Termination.** If a termination condition is satisfied, then go to Step 5. Otherwise, go to Step 4.
4. **Diversification.** Choose a new diverse structure program X_{k+1}, set $k := k + 1$ and go to Step 2.
5. **Intensification.** If additional refinements are needed, then improve the n^* best obtained programs.
6. **Stop.** Stop and return with the best program found.

Algorithm 4.1 exhibits the more advanced TP algorithm. In Step 1, the algorithm starts the search process with a random program X_0 and the empty `TL` and other memory elements. During the search process, those memory elements will be updated regularly to contain information that helps in guiding the algorithm toward an optimal solution, and terminate the search process in a suitable time. Specifically, the memory elements store a set of elite solutions, the numbers of successive non-improvements faced by the static and dynamic structure searches, and the number of fitness evaluations that have been used.

The main loop in the algorithm starts at Step 2 and it contains two basic stages, Step 2.1 and Step 2.2. In Step 2.1, the algorithm uses the shaking procedure to generate n_T trial programs around the current one, based on the tabu restrictions.

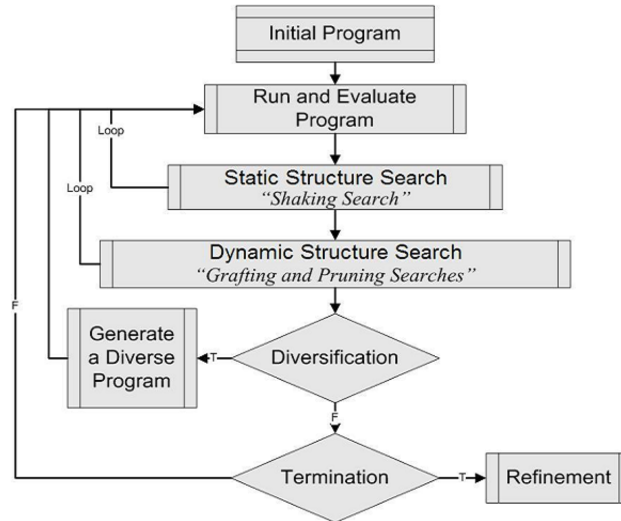

Fig. 7. The TP flowchart.

Then, the best program in the trial set replaces the current program, and the TL and other memory elements are updated. Until a non-improvement condition, e.g., reaching the maximum number of successive internal iterations without improvements, for the static structure search is satisfied, the inner loop consisting of Steps 2.1.1 - 2.1.3 is repeated. Then the algorithm moves to Step 2.2.

In Step 2.2, the algorithm explores new programs a little bit far from the current program by applying the grafting and pruning procedures using random branches of depth ζ_1 (preferably a small positive integer). Two sets containing n'_T and n''_T trial programs are generated using the grafting and pruning procedures, respectively, based on the tabu restrictions. Then, the best program among those trials replaces the current program, and the TL and other memory elements are updated.

In Step 3, the algorithm proceeds to the next iteration, but the need of diversification is checked first, unless a termination condition (e.g., reaching the maximum number of function evaluations) is satisfied. In Step 5, the algorithm refines the n^* best programs during the search process, and in Step 6, stops with the best program obtained.

During the search process, the algorithm generates new programs $\tilde{X}_i$, $i = 1, 2, ..., n_k$ from the current one X_k using one of the local search procedures defined in Section 3, where n_k is a positive integer that represents the number of trial programs generated by the chosen procedure, e.g., $n_k = n_T$ for the shaking procedure in Step 2.1.1 and $n_k = n'_T + n''_T$ for the grafting and the pruning procedures in Steps 2.2.1 and 2.2.2. Specifically, in Step 2.1.1, the algorithm uses the shaking procedure to generate a set S_k of n_T trial programs from the neighborhood $\widetilde{N}_S(X_k) \subset N_S(X_k)$, i.e., $S_k = \{\tilde{X}_\nu | \ \tilde{X}_\nu = \texttt{Shaking}(X_k, \nu),\ \nu = 1, 2, ..., n_T\}$. Similarly, in Steps 2.2.1 and

2.2.2, the algorithm generates sets S'_k and S''_k of n'_T and n''_T trial programs from the neighborhoods $\widetilde{N}_G(X_k) \subset N_G(X_k)$ and $\widetilde{N}_P(X_k) \subset N_P(X_k)$ using the grafting and pruning procedures, respectively, i.e., $S'_k = \{\tilde{X}_\lambda | \ \tilde{X}_\lambda = \texttt{Grafting}(X_k, \lambda, \zeta_1)$, $\lambda = 1, 2, ..., n'_T\}$ and $S''_k = \{\tilde{X}_\eta | \ \tilde{X}_\eta = \texttt{Pruning}(X, \eta, \zeta_1)$, $\eta = 1, 2, ..., n''_T\}$. In addition, each program in S_k, S'_k and S''_k must be verified to be accepted, based on the tabu restrictions or the aspiration criteria.

In Algorithm 4.1, the diversification and intensification procedures are optional, i.e., these procedures should be employed only when a simple TP algorithm is not effective enough. In Step 4, the algorithm randomly chooses one of the grafting or pruning procedures as a diversification procedure with a large depth ζ_2. In addition, in Step 5, the algorithm uses the shaking procedure as an intensification procedure.

Different types of long-term memories can also be invoked to enhance the search process. Memory may save visited elite solutions for further use in the intensification mechanism. Moreover, historical search information may 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 Fig. 8. Moreover, frequencies of choosing a node as a terminal or a function can be saved in order to use them in generating a new tree structure which may have been overlooked in the previous search process.

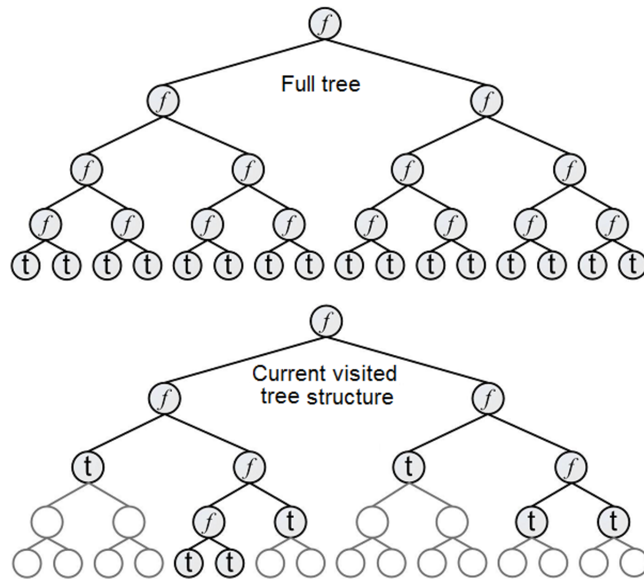

Fig. 8. Example of a visited tree structure for full tree with depth 4.

5. Numerical Experiments

In this section, we discuss the implementation of the TP algorithm. Then, we study the performance of the TP algorithm on three types of benchmark problems; the symbolic regression problem, the 6-bit multiplexer problem and the 3-bit even-parity problem. Some preliminary experiments were carried out first to study the behavior of TP parameters, 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 algorithm. Finally, some comparisons between the TP algorithm and the GP algorithm are reported.

5.1. *Implementation of the algorithm*

In this subsection, we describe the detailed representation of a program in the TP algorithm. In addition, the set of parameters used in the TP algorithm is defined. In the numerical experiments, we used $\zeta_1 = 1$, $\zeta_2 \geq 3$ and $n^* \geq 3$ in Algorithm 4.1.

5.1.1. *Representations of TP individuals*

The TP algorithm generates solutions represented as trees for a given problem. Those solutions are called "computer programs". Each program consists of one or more "gene(s)", where each gene represents a subtree consisting of some external nodes called terminals and some internal nodes called functions. All genes in a program are linked together by using a suitable linking function to produce the final form of that program. In the coding process, the tree structure of a program is transformed to an executable code called genome.[49] We used a new strategy, extracted from the individual representation of the Gene Expression Programming (GEP),[49] to code genes in TP as a linear symbolic string composed of some symbols that represents the underlying terminals and functions.

As to the initial program in the TP algorithm, we generate its genes one by one. Each gene is generated according to the following simple steps: First, a temporary gene in its genome form is generated by choosing its nodes randomly. This gene is composed of two parts, the head which contains function and terminal nodes, and the tail which contains terminal nodes only. The total length (the number of nodes) of the temporary gene is the sum of the head length `hLen` and the tail length $t = \texttt{hLen}(n-1)+1$, where n is the maximum number of arguments of a function. Second, we adjust the final form of the temporary gene by constructing its tree representation and deleting unnecessary elements. In this way, we can guarantee to generate a gene with a syntactically correct structure. For example, when $\texttt{hLen} = 5$, the set of functions is $F = \{+, -, *, /\}$ and the set of terminals is $T = \{a\}$, which implies that $t = \texttt{hLen}(n-1)+1 = 6$ since $n = 2$. Suppose that the generated temporary gene has 11 nodes as in Fig. 9A. By converting this gene to its tree representation (Fig. 9B), we can see that the last 4 elements are unnecessary elements. Consequently, we can delete these unnecessary elements and keep the rest as in Fig.

9C.

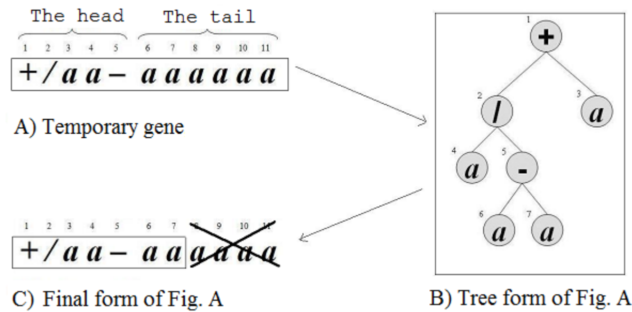

Fig. 9. Constructing a new gene.

Once the initial program is generated, it will be evolved and improved using the shaking, grafting and pruning procedures. For each problem, 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.[49] During this paper, the maximum depth for a program is set to be 10.

5.1.2. *Set of parameters in TP*

As described in 5.1.1, the TP algorithm makes use of a set of two kinds of parameters; representation parameters and search parameters. We list these parameters in the following:

- Representation Parameters
  - `hLen`: The maximum head length for every gene in the initial program.
  - `nGenes`: The number of genes in a program.
- Search Parameters
  - `nTrs`: The number of trial programs to be generated in the neighborhood of the current program. We set n_T, n'_T and n''_T in Algorithm 4.1 all equal to `nTrs`.
  - `StNonImp`: The maximum number of consecutive non-improvements for the static structure search (used as the termination condition for Step 2.1).
  - `MnNonImp`: The maximum number of consecutive non-improvements for the main loop (used as the termination condition for the main loop in Step 2).
  - `IntNonImp`: The maximum number of consecutive non-improvements in the intensification step (used in the termination condition for Step 5).
  - `nTL`: The tabu list size.

- `FunCnt`: The maximum allowed number of fitness evaluations (used to specify the upper limit of the amount of computations).

5.2. *Test problems*

The benchmark problems are described in this subsection.

5.2.1. *Symbolic regression (SR) 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 try to find a function g such that the absolute error

$$\sum_{j=1}^{N} |y_j - g(x_j)| \tag{5.1}$$

is minimized.

In our numerical experiments, we use the quartic polynomial (QP) function given by

$$f(x) = x^4 + x^3 + x^2 + x. \tag{5.2}$$

We generated 20 fitness cases of the form $(x, f(x))$ obtained by choosing x uniformly at random in the interval $[-1, +1]$. In addition, the set of terminals is the singleton $\{x\}$, the set of functions is the set of binary functions $\{+, -, *, \%\}$, where % is called "the protected division". The protected division function is protected against division by zero, and it returns 1 if the denominator argument is 0; otherwise, it returns the normal quotient.[13,14] The fitness value was calculated as the sum, with the sign reversed, of the absolute errors between the output produced by a program and the desired output on each of the fitness cases.

5.2.2. *6-Bit multiplexer (6BM) 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-bit 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) as its output. 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. Our goal is to detect the Boolean 6-bit multiplexer function.

For the 6-bit multiplexer problem, we make the following assumptions.

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

(2) The set of functions consists of the Boolean functions {AND, OR, NOT, IF}, where IF(x, y, z) returns the value y if x is true, and it returns the value z otherwise.
(3) 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 use the entire set of 64 combinations of activities as the fitness cases for evaluating the fitness.[13] The fitness value in this case is the number of fitness cases where the Boolean value returned by the TP algorithm for a given combination of arguments is the correct Boolean value. Thus, the fitness value for this problem ranges between 0 and 64.

5.2.3. *3-Bit even-parity (3BEP) problem*

The Boolean N-bit even-parity function is a function of N-bit arguments. In addition, it returns 1 (True) if the arguments contain an even number of 1's and it returns 0 (False) otherwise. In fact, the Boolean N-bit even-parity functions appear to be the most difficult Boolean functions to detect via a blind random search.[13] Here, using the TP algorithm, we try to detect the Boolean 3-bit even-parity function as a special case of N-bit even-parity functions. The Boolean 3-bit even-parity function is a function of 3 activities a_2, a_1, a_0 and we make the following assumptions.

(1) The 3 activities $\{a_2, a_1, a_0\}$ constitute the set of terminals.
(2) The Boolean functions {AND, OR, NAND, NOR} constitute the set of functions.
(3) There are $2^3 = 8$ possible combinations of the 3 activities a_2, a_1, a_0 along with the associated correct values of the 3-bit even-parity function. Therefore, we use the entire set of 8 combinations of activities as the fitness cases for evaluating the fitness.[13] The fitness value is the number of fitness cases where the Boolean value returned by the TP algorithm for a given combination of arguments is the correct Boolean value. Thus, the fitness value for this problem ranges between 0 and 8.

5.3. *Performance analysis*

In this subsection, we study the performance of the TP algorithm under different environments. First, we discuss several tabu list structures and representations to authorize one of them as the `TL` of the TP algorithm. Then, the parameter setting of the TP algorithm is introduced to choose the best parameter values that will be used through the rest of the numerical experiments. Throughout this subsection, we use the standard values of the TP parameters shown in Table 1.

Table 1. Standard values of the TP parameters.

Parameter	Value		
	SR-QP	6BM	3BEP
hLen	3	3	3
nGenes	3	7	3
nTrs	5	7	5
StNonImp	3	3	3
MnNonImp	7	7	7
IntNonImp	3	3	3
nTL	7	7	7
FunCnt	2500	25000	25000

5.3.1. *Structure of the tabu list*

In TS, the tabu list TL represents the short-term memory which is limited in terms of time and storage capacity. The TL is a list that stores certain attributes for the last nTL visited solutions, to decide if a new solution is accepted or not. Basically, the TL is used to prevent the algorithm from being trapped in a cycle or a local optimum. In addition, in most implementations of TS, TL is used to store nTL latest moves, or attributes of nTL latest solutions instead of storing the entire solutions themselves.

Here, we discuss the structure of the TL in the proposed TP algorithm. We start by introducing some of the important attributes for a program generated in the TP algorithm:

- The fitness value of a program.
- The structure of a program and its information, for example, the number of nodes, the depth, the width (the maximum number of nodes that lie in the same depth from the root node), the number of function nodes, and the number of terminal nodes.
- The latest operations performed to get a program, for example, the latest node(s) changed and the latest local search procedure used to generate this program.
- The program itself.

To use the fitness value as an attribute in the TL, the algorithm must construct each candidate solution and evaluate its fitness value before determining its tabu status. However, this process increases the number of fitness evaluations and the computational effort for the algorithm. Therefore, we omit the use of the fitness value as an attribute in the TL for the TP algorithm. To choose a more suitable TL structure, the following TL structures are used and examined

(1) $\texttt{TL}_1$: The tabu list stores some information about the structure of current tabu programs (the number of nodes, the depth, the width, and the number of function nodes). Therefore, the $\texttt{TL}_1$ is a list of `nTL` vectors, each consisting of four elements, which contain the structure information of the last `nTL` visited solutions.
(2) $\texttt{TL}_2$: The tabu list stores vectors of bits corresponding to the genome representations of current tabu programs. Specifically, the $\texttt{TL}_2$ is a list of `nTL` vectors of bits gotten from the genome representations of the last `nTL` visited solutions, where 1 corresponds to a function node, and 0 corresponds to a terminal node.
(3) $\texttt{TL}_3$: The tabu list stores the same vectors of bits in $\texttt{TL}_2$, as well as some information about the latest operations performed to get the tabu programs. Specifically, the $\texttt{TL}_3$ is a list of `nTL` vectors, where each vector stores bit values that correspond to the genome representation of a program, the latest gene changed, the latest node processed, and the latest local search procedure used to generate this program.
(4) $\texttt{TL}_4$: The tabu list stores the genome representations of the latest `nTL` visited solutions. Therefore, the $\texttt{TL}_4$ is a list that contains `nTL` vectors of strings. In addition, storing a program in the $\texttt{TL}_4$ as a vector of strings is not costly in terms of memory.

The algorithm may generate many trees that have the same number of nodes, the same depth, the same width, and the same number of function nodes, but have different structures or different results. The $\texttt{TL}_1$, $\texttt{TL}_2$ and $\texttt{TL}_3$ will therefore reject some (may be a lot) of the new unvisited programs if they share the same attributes with one of the tabu programs. On the other hand, the $\texttt{TL}_4$ will reject a new program if it exactly matches one of the tabu programs. In fact, $\texttt{TL}_i$ is considered more restrictive than $\texttt{TL}_{i+1}$ for $i = 1, 2, 3$, in terms of accepting new solutions.

For each `TL` structure discussed above, we performed 100 runs of the TP algorithm for the SR problem with the QP function (5.2), which will be referred to as the SR-QP problem in the rest of the paper. The values of the TP parameters are set as in Table 1, except `FunCnt` = 5000. The results of the TP algorithm with the above-mentioned four `TL` structures are compared in Table 2, in terms of the average (AV) and the median (ME) of the number of fitness evaluations as well as the rate of success (R). It is clear from these results that a less restrictive tabu list yields a more efficient TP algorithm. In particular, the $\texttt{TL}_4$ is the best choice for the current version of the TP algorithm.

In the rest of this paper, we adjust the `TL` to store the last `nTL` visited solutions in its genome coding representation. This means each element in the `TL` is a vector of strings that represents a program. If there is one program in the `TL` that has the same number of nodes as the new program, then the complete test is performed. If the new program matches to some program in the `TL`, it will be rejected. Otherwise, the new program will be accepted.

Table 2. Performance of the TP algorithm with different tabu list structures.

TL	AV	ME	R%
TL_1	4,171	5,000	31
TL_2	3,476	4,239	67
TL_3	1,485	1,015	95
TL_4	795	681	100

5.3.2. *Parameters setting*

In this subsection, we study the effect of the TP parameters and discuss the choice of their proper values for each problem. For each parameter, we select a set of values, and for each value, we performed 100 independent runs to compute the average of the number of fitness evaluations as well as the rate of success. Other parameters are fixed at their standard values given in Table 1.

The computational results are displayed in Table 3. It is clear from these results that the crucial parameters in the TP algorithm are `nGenes`, `nTrs` and `MnNonImp`, while other parameters affect the success rate only slightly. In addition, although the best value for `nGenes` depends on the problem itself, appropriate values are roughly $2 \leq \texttt{nGenes} \leq 5$ for the SR-QP and 3BEP problems, and $3 \leq \texttt{nGenes} \leq 9$ for the 6BM problem. As to `nTrs`, it should be large enough but should not be too large (to avoid consuming the allowed number of fitness evaluations in earlier stages of the algorithm) and suitable values are roughly $5 \leq \texttt{nTrs} \leq 9$. Lastly, a proper choice for `StNonImp` is a large value, e.g., $5 \leq \texttt{StNonImp} \leq 9$.

5.4. *TP vs GP*

To examine the performance of the TP algorithm compared to the GP algorithm, we performed several experiments for different problems. First, we compared the results of the TP algorithm with the results of the release 3.0 of the Genetic Programming Lab (GPLab) toolbox.[50] Second, the results of the TP algorithm were compared with the results of different versions of the GP algorithm that appeared in the literature. Finally, we performed some experiments for the TP and GP algorithms using different sets of operators, to show the effects of our local search procedures.

In fact, a perfect comparison between the TP algorithm and the GP algorithm cannot be made due to the differences in the search techniques, since TP is a point-to-point algorithm, while GP is a population-based algorithm. But here, we just try to show their performance in terms of the rate of success and the number of fitness evaluations needed to get a desired solution. Throughout the experiments in this subsection, the parameter values for the TP algorithm, the GPLab toolbox and the standard GP algorithm are set as in Table 4.

Table 3. Performance of the TP algorithm with different values of each TP parameter.

	SR-QP			6BM			3BEP		
Param.	Val.	AV	R%	Val.	AV	R	Val.	AV	R%
hLen	1	1,092	90	1	8,062	96	1	7,311	96
	3	870	97	3	7,739	96	3	7,958	92
	5	897	94	5	7,985	97	5	7,206	98
	7	964	92	7	8,074	94	7	6,853	98
	9	1,147	88	9	7,646	95	9	7,847	96
nGenes	1	1,098	84	1	16,268	63	1	16,074	57
	2	981	95	3	9,432	91	2	9,383	86
	3	801	99	5	7,606	95	3	6,840	99
	4	963	90	7	8,198	95	4	7,540	96
	5	1,068	89	9	9,102	95	5	7,693	96
nTrs	2	1,597	61	2	21,451	43	2	18,905	48
	3	1,274	82	3	12,295	89	3	9,123	94
	5	901	94	5	7,971	95	5	6,477	99
	7	1,012	91	7	7,825	98	7	6,973	96
	9	1,068	86	9	8,593	96	9	7,185	99
StNonImp	1	1,143	85	1	10,569	90	1	9,507	92
	3	906	93	3	8,419	96	3	6,825	98
	5	1,028	91	5	8,353	96	5	6,841	99
	7	967	96	7	7,829	98	7	6,483	98
	9	1,107	89	9	8,070	95	9	5,897	97
MnNonImp	1	1,242	80	1	14,592	76	1	12,972	80
	3	1,146	89	3	9,635	96	3	7,640	94
	5	1,097	91	5	7,093	96	5	7,524	96
	7	987	93	7	8,527	95	7	7,450	94
	9	1,005	93	9	8,037	97	9	5,612	100
nTL	1	1,073	85	1	7,488	98	1	7,211	96
	3	1,279	80	3	8,157	97	3	6,704	97
	5	914	94	5	7,928	99	5	6,755	98
	7	971	90	7	9,659	97	7	7,055	97
	9	1,035	89	9	6,988	98	9	7,429	94

Table 4. Parameter values for the TP algorithm, the GPLab toolbox and the standard GP algorithm.

Algorithm	Parameter	SR-QP	SR-Poly-4	6BM	3BEP
TP	hLen	3	3	3	3
	nGenes	3	3	7	3
	nTrs	5	9	7	5
	StNonImp	3	7	7	3
	MnNonImp	7	9	7	9
	IntNonImp	3	3	3	3
	nTL	7	7	7	7
GPLab	nPop	50	-	500	500
	nGnrs	50	-	50	50
GP	nPop	50	-	-	500
	nGnrs	50	-	-	50
	Crossover probability: 0.8				
	Mutation probability: 0.2				
	Selection: the tournament selection of size 4				

5.4.1. *TP algorithm vs GPLab toolbox*

GPLab[50] is a free Matlab toolbox that can be used under general public license (GNU) software regulations. In addition, the current release of GPLab includes most of the traditional features usually found in GP tools and it has the ability to accommodate a wide variety of usages. For more details about GPLab and its usage, see Ref. 51.

We show the performance of TP compared to GPLab in the case where both of them have a limited number of fitness evaluations. So, we performed 100 independent runs for both of TP and GPLab under the same limitation on the number of fitness evaluations. The parameter values for the TP and GPLab toolbox are shown in Table 4. For the GPLab toolbox, we set its parameter values as the standard values,[50] except for the population size `nPop` and the number of generations `nGnrs` to meet the condition for the maximum number of fitness evaluations. Note that, the maximum number of fitness evaluations for TP and GPLab is `nPop*nGnrs`. The results are shown in Table 5, where comparisons are made in terms of the average and the median of the number of fitness evaluations as well as the rate of success.

It is clear, from the results shown in Table 5, that the TP algorithm generally outperforms the GPLab toolbox. Specifically, the TP algorithm was able to obtain good and acceptable solutions in an early stage of computations, compared with

Table 5. Comparison among the TP algorithm and the GPLab toolbox for the SR-QP, 6BM and 3BEP problems.

	TP			GPLab		
Prob.	AV	ME	R%	AV	ME	R%
SR-QP	801	652	99	1,303	1,075	81
6BM	7,829	6,393	98	8,445	7,500	100
3BEP	5,612	4,272	100	11,175	6,500	77

the GPLab toolbox. At the same time, the rate of success for the TP algorithm is better than the corresponding rate of success for the GPLab toolbox, especially for the 3BEP problem.

5.4.2. *TP vs GP, BC-GP, CGP and ECGP*

Here, we compare the TP algorithm with different versions of the GP algorithm that appeared in the literature, and show that the TP algorithm performs well compared to all those versions of the GP algorithms. The parameter values for the TP algorithm are set as in Table 4.

Poli and Langdon[52] conducted extensive numerical experiments to compare the backward-chaining GP (BC-GP) algorithm and the standard GP algorithm. They considered two types of symbolic regression problems; one is the SR-QP problem with $f(x) = x^4 + x^3 + x^2 + x$, and the other is the symbolic regression with the multivariate polynomial $f(x_1, \cdots, x_4) = x_1x_2 + x_3x_4 + x_1x_4$, which we call the SR-Poly-4 problem. For the SR-QP problem, they used 20 fitness cases of the form $(x, f(x))$ obtained by choosing x uniformly at random in the interval $[-1, +1]$. In addition, 50 fitness cases of the form $(x_1, x_2, x_3, x_4, f(x_1, \cdots, x_4))$ generated by randomly setting $x_i \in [-1, +1], i = 1, 2, 3, 4$ have been used for the SR-Poly-4 problem. The function set for these problems included the functions $+, -, *$ and $\%$, where $x\%y = x$ if $y = 0$; otherwise, $x\%y = x/y$. The terminal set included the independent variables for each problem. The fitness value was calculated as the sum, with the sign reversed, of the absolute errors between the output produced by a program and the desired output on each of the fitness cases.

Poli and Langdon[52] performed two independent experiments for each of the SR-QP and the SR-Poly-4 problems, to compare between the BC-GP and GP algorithms using different population sizes. In the first two experiments, they performed 5000 independent runs using $\texttt{nPop} = 100$ and $\texttt{nPop} = 1,000$ for the SR-QP and SR-Poly-4 problems, respectively. In the other experiments, they performed 1000 independent runs using $\texttt{nPop} = 1,000$ and $\texttt{nPop} = 10,000$ for the SR-QP and SR-Poly-4 problems, respectively. For all experiments they used $\texttt{nGnrs} = 30$. The results shown in Fig. 10 for the BC-GP and GP algorithms are taken from Figs.

8-11 in the original reference, Ref. 52.

We performed the same experiments for the SR-QP and SR-Poly-4 problems using the TP algorithm to compare its results with those of Poli and Langdon[52]. The parameter values for the TP algorithm are shown in Table 4, and the results of the TP, BC-GP and GP algorithms are shown in Fig. 10. As we can see from these figures, the TP algorithm can obtain a desired solution very fast compared to both of the BC-GP and GP algorithms, especially for the more difficult problem SR-Poly-4. It is clear that the TP algorithm can save a lot of efforts and computations compared to the BC-GP and GP algorithms.

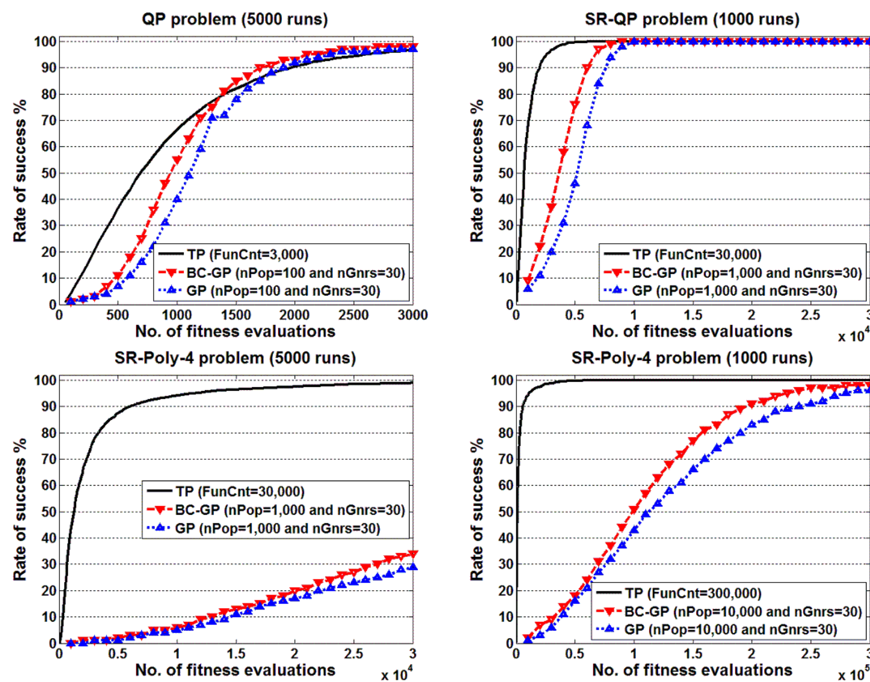

Fig. 10. Comparison among the TP, BC-GP and GP algorithms for the SR-QP and SR-Poly-4 problems.

Walker and Miller[53] conducted extensive numerical experiments to examine the performance of the Cartesian GP (CGP) algorithm and the Embedded CGP (ECGP) algorithm. They reported a lot of results for several test problems with the CGP and ECGP algorithms, and showed that those algorithms outperformed the standard GP algorithm and several contemporary algorithms. In particular, the ECGP algorithm is a generalization of the CGP algorithm that utilizes an automatic module acquisition technique to automatically build and evolve modules, and re-use these modules as functions in the main program. Here, we consider the results of the CGP and ECGP algorithms for the 3BEP problem.

Walker and Miller[53] used the Boolean functions {AND, OR, NAND, NOR} as the function set and the arguments $\{a_0, a_1, a_2\}$ as the terminal set for the 3BEP problem. In addition, they used the module technique to automatically build and evolve modules in the ECGP algorithm. They performed 50 independent runs for the 3BEP problem and, for each run, the algorithm was run until the exact solution was found. Then, they made some statistical analysis for their results to discuss the performance of the CGP and ECGP algorithms.

Here, we also performed 50 independent runs for the 3BEP problem using the TP algorithm, where the parameter values for the TP algorithm are shown in Table 4. For each run, the TP algorithm was run until the exact solution was obtained. A comparison of the performance of the CGP, ECGP and TP algorithms is shown in Table 6. From these results, we can see that the TP algorithm performs well compared to the CGP and ECGP algorithms. The results for the CGP and ECGP algorithms have been taken from Table IV in the original reference, Ref. 53.

Table 6. Comparison among the TP, CGP and ECGP algorithms for the 3BEP problem, in terms of median (ME) number of evaluations, median absolute deviation (MAD), and interquartile range (IQR) for 50 independent runs.

Algorithm	ME	MAD	IQR
CGP	5,993	2,936	6,610
ECGP (with modules)	5,931	3,804	10,372
TP	4,170	2,296	6,511

Koza[13] published a book on GP, which contains numerical results for a large number of problems. In addition, a book chapter by Azad and Ryan[54] also contains a lot of numerical results. These two references also contain results for some of the problems used in this paper, and our results seem to compare favorably to those reported there.

5.4.3. *TP with mutation vs GP with local search*

In the previous subsections, we saw that TP outperforms the standard GP and various contemporary versions of GP. In this subsection, we study the performance of the local search procedures introduced in Section 3, compared to the ordinary mutation operator in the standard GP algorithm. Therefore, several comparisons, in terms of the average and the median of the number of fitness evaluations as well as the rate of success, will be given for the following algorithms:

(1) GP: The standard GP.
(2) GP-LS: The standard GP using the local search procedures in Section 3 instead of the mutation operator.

(3) TP: The proposed TP algorithm as described in Algorithm 4.1.
(4) TP-Mut: The proposed TP algorithm using the mutation operator instead of the local search procedures in Section 3.

To make comparisons, we performed 100 independent runs of each algorithm for the SR-QP and 3BEP problems. The parameter values are set as shown in Table 4. When applying the mutation, a node is chosen randomly from a program, and the subtree rooted at this node is replaced by a new subtree generated randomly as a gene in the initial population.[13,14] The initial populations for the GP and GP-LS algorithms are generated and represented in the same way as TP generates and represents genes, where each program in GP and GP-LS is represented as one gene according to the standard GP algorithm. The results are shown in Table 7, where the maximum number of fitness evaluations for the GP, GP-LS, TP and TP-Mut algorithms is `nPop*nGnrs`.

Table 7. Comparison among the GP, GP-LS, TP and TP-Mut algorithms for the SR-QP and 3BEP problems.

	SR-QP			3BEP		
Algorithm	AV	ME	R%	AV	ME	R%
GP	1,056	700	79	13,960	10,750	76
GP-LS	783	625	96	13,600	11,500	82
TP-Mut	1,615	1,851	62	3,969	3,261	100
TP	801	652	99	5,612	4,272	100

From Table 7, we observe that, for the SR-QP problem, TP and GP with local search procedures performed better GP and TP with the mutation operator, in terms of AV, ME and R. For the 3BEP problem, the mutation operator helped to improve the results slightly compared with the local search procedures, in terms of AV and ME. Nevertheless, the local search procedures were slightly more effective than the mutation operator in terms of the rate of success especially for the GP algorithm. Although it is not a completely fair comparison, one can see that the two versions of the TP algorithm outperform the current two versions of the standard GP algorithm for the 3BEP problem. However, the two versions of the GP algorithm slightly outperform the two versions the TP algorithm for the SR-QP problem. From the previous results of the GP, GP-LS, TP and TP-Mut algorithms, one may conclude that no algorithm can be the absolute winner. Each algorithm can outperform other algorithms for some set of problems, but it cannot outperform them for all problems. This may be a good motivation for those who introduce new algorithms.

6. Discussion and Future Works

The promise of the TP results shown in the previous section has inspired the authors to model a comprehensive framework containing a class of general problem solvers. Using the local search procedures introduced in Section 3, various meta-heuristics can be extended to deal with tree data structures. We call this new framework Meta-Heuristics Programming (MHP).

The main steps of applying meta-heuristics to solve a given problem are summarized as follows:

(1) Select a meta-heuristic method that has shown efficient evidences in related problems.
(2) Compose the best configurations of the search procedures of the selected method. For example, if GA is the selected method, then define the crossover, mutation and selection procedures that fit the given problem.
(3) Set and tune the initial and controlling parameters of the selected method in order to obtain the best results.

Hence, the choice of a suitable meta-heuristic is an essential issue in problem solving. Moreover, it has been reported that the performance of search methods, especially meta-heuristics, varies even when they are applied to the same problem.[46,55] Moreover, the concept of No Free Lunch[56,57] has shown that no search algorithm is better than the others when its performance is averaged over all possible applications. Therefore, the existence of different types of meta-heuristics and their diversity are highly needed to accommodate different types of applications. This also has inspired the authors to extend meta-heuristics to deal with applications by utilizing a tree data structure through the MHP framework. More details of this framework are stated below.

Generally speaking, heuristic algorithms make use of four procedures. Two of them are essential and the other two procedures are optional. The first one is the procedure that generates a set of trial solutions around the current solution, e.g., the neighborhood structure in the Tabu Search and Scatter Search algorithms. The second one is the procedure that updates the search process to move to the next iteration, e.g., the best solution from the set of trial solutions replaces the current one in the Tabu Search algorithm or replaces the current solution in the Simulated Annealing algorithm if a certain probability is greater than a random number between 0 and 1. On the other hand, the third procedure (optional) is the procedure that drives the search process to explore new regions in the case of being trapped in local optima. Finally, the fourth one (optional) is the procedure that improves the best program(s) obtained during the search process.

In the MHP, initial computer program(s) represented as parse tree(s) can be adapted through the following five procedures to obtain acceptable target solutions of the given 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: Accelerate the search process if a promising solution is detected, or escape from the current solution 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.

The 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 algorithm. The local search procedures introduced in Section 3 are used in the TrialProgram procedure, while the UpdateProgram procedure depends on the type of invoked meta-heuristics.

The main structure of the MHP framework is shown in Fig. 11. In its initialization step, the 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 the MHP framework starts by calling the TrialProgram procedure to generate a set of trial programs from the current population. Then, the MHP framework detects characteristic states in the recent search process and applies the Enhancement procedure to generate new promising trial programs using the following tactics.

- *Intensive Enhancement.* Apply a faster local search technique 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, the UpdateProgram procedure is used to extract the next program or the next population from the current ones. Consequently, the controlling parameters are also updated to fit in 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 an empty memory and collect useful search information hierarchically.
- *Global Memory.* Start with a full memory that samples the whole search space, and remove unnecessary memory elements when new solutions are visited.

When a full assembly memory or an empty global memory is obtained, we may 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. The Diversification procedure may be applied to generate new diverse solutions by guidance of the adaptive memory.

It is worthwhile to note that the MHP framework can be implemented in different ways depending on the type of the invoked meta-heuristics; a point-to-point algorithm or a population-based algorithm. In Fig. 11, if the population size is 1, then the algorithm will work as a point-to-point algorithm. Otherwise, the algorithm will work as a population-based algorithm.

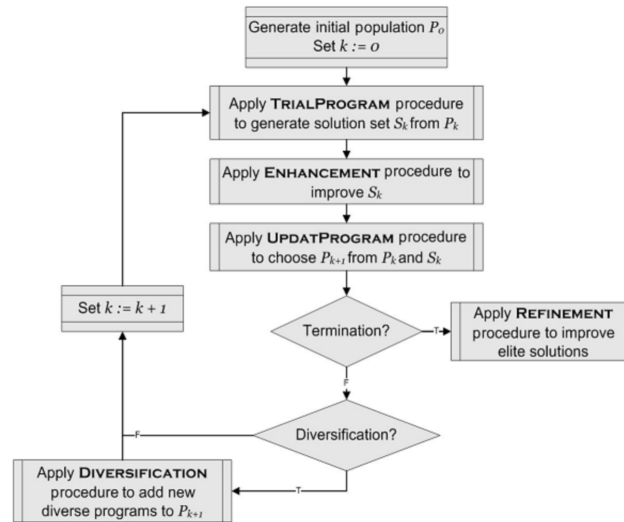

Fig. 11. The MHP flowchart.

Finally, the MHP framework can extend other meta-heuristics to develop general problem solvers. Actually, the authors and their research groups are currently trying to apply the MHP framework to extend some other meta-heuristics such as memetic algorithm, simulated annealing, scatter search, artificial immune systems, etc. Some of these works have already been presented in conferences, see Memetic Programming[58] and Scatter Programming.[59]

Although the TP algorithm outperforms the CGP and ECGP algorithms, we expect that the results of the TP algorithm can be improved further by means of the Automatically Defined Functions (ADFs). The ADF is a sub-tree that can be used as a function (called subroutine, subprogram, or module) of dummy arguments in the main tree to exploit symmetries and modularities in problem environments.[14] In GP with ADFs, each program consists of one or more function-defining branches, i.e., ADF, and one result-producing branch that produces the final value of this program. While the algorithm is running, ADFs for a program are created, evolved and included automatically into the original function set for the result-producing branch of this program. In addition, the idea of using modules in the ECGP algorithm is similar to using ADFs in the standard GP algorithm. Actually, based on the individual representation in TP, we can extend the TP algorithm to build and

reuse ADFs. Since each program in TP contains one or more gene(s), we can adapt one or more of these genes to work as ADF(s). However, this technique needs a lot of experiments to adjust the parameters and improve the results. Therefore, we consider this point to be a future work.

7. Concluding Remarks

Genetic Programming (GP) is one of the powerful tools in computational intelligence. It deals with a search space of computer programs which can be represented as parse trees. However, it has been argued that crossover and mutation in GP are highly disruptive with the risk of non-convergence to an optimal structure. To address this issue, this work has introduced some local search procedures over a tree space as alternative operations to crossover and mutation. The proposed procedures have two aspects; static structure search and dynamic structure search. Static structure search aims to explore a neighborhood of a tree by altering its nodes without changing its structure. Dynamic structure search changes the structure of a tree by expanding its terminal nodes or cutting its subtrees.

The Tabu Programming (TP) algorithm has been proposed by incorporating the basic idea in the Tabu Search (TS) algorithm, a popular point-to-point meta-heuristic method. The main difference between TP and TS lies in the representation of a solution and the neighborhood structure. More specifically, every solution in the TP algorithm is a computer program represented by a parse tree. Therefore, the search space of the TP algorithm is the set of computer programs that can be represented by parse trees. In addition, the neighborhoods of a solution X are defined and explored using the proposed local search procedures.

We have tested the performance of the TP algorithm for three types of benchmark problems and made some experiments to analyze the main components of the TP algorithm. From these numerical experiments, we have shown that the TP algorithm is a promising algorithm compared with the GP algorithm. In fact, the TP algorithm performs better than the GP algorithm in terms of the rate of success and the number of fitness evaluations at least for the considered test problems.

Finally, the procedures of TP are generalized to construct a more general framework. Using these procedures, various meta-heuristics can be employed to deal with tree data structures in a unified framework which we call Meta-Heuristics Programming.

Acknowledgment

The authors are grateful to the referees for their many helpful comments and suggestions.

References

1. J. R. Koza, Hierarchical genetic algorithms operating on populations of computer programs, in *Proc. 11th Int. Joint Conference on Artificial Intelligence* (Morgan Kauf-

mann: Los Altos, CA, 1989), pp. 768–774.

2. J. R. Koza, Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical Report: CS-TR-90-1314, Stanford University, Stanford, USA (1990).
3. A. Hedar, and M. Fukushima, Tabu search directed by direct search methods for nonlinear global optimization, *Eur. J. Oper. Res.* **170**(2006) 329–349.
4. A. Hedar, J. Wang, and M. Fukushima, Tabu search for attribute reduction in rough set theory, *Soft Comput.* **12**(2008) 909–918.
5. M. S. Arumugam, M. V. C. Rao, On a class of hybrid systems via a novel approach for real-coded genetic algorithm with hybrid selection, *Int. J. Inf. Technol. Decis. Mak.* **6**(2) (2007) 315–332.
6. D. E. Goldberg, *The Design of Innovation: Lessons from and for Competent Genetic Algorithms* (New York: Addison-Wesley, 2002).
7. M. D. Vose, *The Simple Genetic Algorithm: Foundations and Theory* (Cambridge, MA: MIT Press, 1999).
8. E. C. Brown, C. T. Ragsdale, A. E. Carter, A grouping genetic algorithm for the multiple traveling salesperson problem, *Int. J. Inf. Technol. Decis. Mak.* **6**(2) (2007) 333–347.
9. D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning* (New York: Addison-Wesley, 1989).
10. Y. C. Hu, F. M. Tseng, Mining simplified Fuzzy If-Then rules for pattern classification, *Int. J. Inf. Technol. Decis. Mak.* **8**(3) (2009) 473–489.
11. L. Nie, X. Xu, D. Zhan, Collaborative planning in supply chains by Lagrangian relaxation and genetic algorithms, *Int. J. Inf. Technol. Decis. Mak.* **7**(1) (2008) 183–197.
12. N. L. Cramer, A representation for the adaptive generation of simple sequential programs, in *Proc. Int. Conference on Genetic Algorithms and their Applications* (Pittsburgh, USA, 1985), pp. 183–187.
13. J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Cambridge, MA: MIT Press, 1992).
14. J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs* (Cambridge, MA: MIT Press, 1994).
15. 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).
16. 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).
17. W. B. Langdon and R. Polií, *Foundations of Genetic Programming* (Springer-Verlag 2002).
18. P. Nordin and W. Banzhaf, Complexity compression and evolution, in *Proc. 6th Int. Conference on Genetic Algorithms* (Morgan Kaufmann, Pittsburgh, PA, USA, 1995), pp. 310–317.
19. P. Nordin, F. Francone and W. Banzhaf, Explicitly defined introns and destructive crossover in genetic programming, in *Proc. Workshop on Genetic Programming: From Theory to Real-World Applications* (California, USA, 1995), pp. 6–22.
20. P. Kouchakpour, A. Zaknich and T. Bräunl, A survey and taxonomy of performance improvement of canonical genetic programming, *Knowl. Inf. Syst.* **21**(2009) 1–39.
21. M. D. Terrio and M. I. Heywood (2002) Directing crossover for reduction of bloat in GP, in *Proc. Canadian Conference on Electrical and Computer Engineering* (IEEE Press, 2002), pp. 1111–1115.

22. W. Banzhaf, P. Nordin, R. E. Keller and F. D. Francone, *Genetic Programming An Introduction; On the Automatic Evolution of Computer Programs and its Applications* (Morgan Kaufmann, San Francisco, CA, 1998).
23. T. Castle and C. G. Johnson, Positional effect of crossover and mutation in grammatical evolution, in *Proc. 13th European Conference on Genetic Programming* (Istanbul, Turkey, 2010), pp. 26–37.
24. T. H. Hoang, N. X. Hoai, R. I. McKay and D. Essam, The importance of local search: A grammar based approach to environmental time series modelling, in *Genetic Programming: Theory and Practice III*, Vol 9, eds. T. Yu, R. L. Riolo and B. Worzel (Springer-Verlag, 2006), pp. 159–175.
25. C. G. Johnson, Genetic programming crossover: Does it cross over?, in *Proc. 12th European Conference on Genetic Programming* (Springer, Berlin, 2009), pp. 97–108.
26. T. Blickle, and L. Thiele, Genetic programming and redundancy, in *Proc. Genetic Algorithms within the Framework of Evolutionary Computation* (Saarbrücken, Germany, 1994), pp. 33–38.
27. L. Zhang and A. K. Nandi, Diversity-preserving non-destructive operators in genetic programming and their application to breast cancer diagnosis, *Trans. Inst. Meas. Control*, **31**(6) (2009) 533–550.
28. W. A. Tackett, Recombination, selection and the genetic construction of computer programs, PhD thesis, University of Southern California, (1994).
29. W. A. Tackett and A. Carmi, The unique implications of brood selection for genetic programming, in *Proc. 1st IEEE Conference on Evolutionary Computation* (New York, NY, 1994), pp. 160–165.
30. T. Ito, H. Iba and S. Sato, Non-destructive depth-dependent crossover for genetic programming, in *Proc. 1st European Workshop on Genetic Programming* (Springer, Heidelberg, 1998), pp. 71–82.
31. H. Majeed and C. Ryan, On the constructiveness of context-aware crossover, in *Proc. 9th Annual Conference on Genetic and Evolutionary Computation* (London, England, 2007), pp. 1659–1666.
32. M. Keijzer, C. Ryan, M. ONeill, M. Cattolico and V. Babovic, Ripple crossover in genetic programming, in *Proc. of Genetic Programming* (Springer-Verlag, Lake Como, Italy, 2001), pp. 74–86.
33. M. Kessler, T. Haynes, Depth-fair crossover in genetic programming, in *Proc. 1999 ACM Symposium on Applied Computing* (San Antonio, Texas, US, 1999), pp. 319–323.
34. N. X. Hoai, R. I. McKay, and D. Essam, Representation and structural difficulty in genetic programming, *IEEE Trans. Evol. Comput.* **10**(2) (2006) 157–166.
35. L. Lin, M. Gen, Auto-tuning strategy for evolutionary algorithms: Balancing between exploration and exploitation, *Soft Comput.* **13**(2009) 157–168.
36. F. Glover, Future paths for integer programming and links to artificial intelligence, *Comput. Oper. Res.* **13**(1986) 533–549.
37. F. Glover, E. Taillard and D. Werra, A user's guide to tabu search, *Ann. Oper. Res.* **41**(1993) 3–28.
38. F. Glover and M. Laguna, *Tabu Search* (Kluwer Academic Publishers, Boston, MA, 1997).
39. F. Glover and G. Kochenberger (eds.), *Handbook of MetaHeuristics* (Kluwer Academic Publishers, Boston, MA, 2002).
40. F. Glover and G. Kochenberger, New optimization models for data mining, *Int. J. Inf. Technol. Decis. Mak.* **5**(4) (2006) 605–609.
41. F. Glover and R. Marti, Tabu search, in *Metaheuristic Procedures for Training Neutral Networks*, eds. E. Alba and R. Marti (Springer-Verlag, Berlin, Germany, 2006), pp.

53–69.
42. A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing* (Springer-Verlag, Berlin, 2003).
43. R. Diestel, *Graph Theory* (Springer-Verlag, Berlin, 2005).
44. E. K. Burke, S. Gustafson, and G. Kendall, Diversity in genetic programming: An analysis of measures and correlation with fitness, *IEEE Trans. Evol. Comput.* **8**(1) (2004) 47–62.
45. M. Boryczka, Z. J. Czech, Solving approximation problems by ant colony programming, in *Late Breaking Papers at Genetic and Evolutionary Computation Conference* (New York, NY, 2002), pp. 39–46.
46. M. Boryczka, Eliminating introns in ant colony programming, *Fundamenta Informaticae* **68**(2005) 1–19.
47. A. Hedar and M. Fukushima, Meta-heuristics programming, in *Proc. 2nd Int. Workshop on Computational Intelligence & Applications* (Okayama, Japan, 2006).
48. J. Balicki, Tabu programming for multiobjective optimization problems, *Int. J. Comput. Sci. Network Security* **7**(10) (2007) 44–51.
49. C. Ferreira, Gene expression programming: A new adaptive algorithm for solving problems, *Complex Systems* **13**(2001) 87–129.
50. S. Silva, GPLAB: A genetic programming toolbox for MATLAB, http://gplab.sourceforge.net/
51. E. William, J. Northern, Genetic programming lab (GPLab) tool set version 3.0, in *Proc. IEEE Region 5 Technical, Professional, and Student Conference* (Kansas City, Kansas, 2008), pp. 1–6.
52. R. Poli, W. B. Langdon, Backward-chaining evolutionary algorithms, *Artif. Intell.* **170**(2006) 953-982.
53. J. A. Walker and J. F. Miller, The automatic acquisition, evolution and reuse of modules in cartesian genetic programming, *IEEE Trans. Evol. Comput.* **12**(4) (2008) 397–417.
54. R. M. A. Azad and C. Ryan, An examiation of simultaneous evolution of grammars and solutions, in *Genetic Programming: Theory and Practice III*, Vol. 9, eds. T. Yu, R. L. Riolo and B. Worzel (Springer-Verlag, 2006), pp. 141–158.
55. M. Dorigo, T. Stützle, *Ant Colony Optimization* (The MIT Press, 2004).
56. D. H. Wolpert and W. G. Macready, No free lunch theorems for search, Technical Report: SFI-TR-05-010, Santa Fe Institute, Sante Fe, NM (1995).
57. D. H. Wolpert, W. G. Macready, No free lunch theorems for optimization, *IEEE Trans. Evol. Comput.* **1**(1) (1997) 67–82.
58. E. Mabrouk, A. Hedar and M. Fukushima, Memetic programming with adaptive local search using tree data structures, in *Proc. 5th Int. Conference on Soft Computing as Transdisciplinary Science and Technology* (Paris, France, 2008), pp. 258–264.
59. A. Hedar and M. Kamel, Scatter programming, in *Proc. 7th Int. Conference on Informatics and Systems* (Cairo, Egypt, 2010).