META-HEURISTICS PROGRAMMING AND ITS APPLICATIONS

EMAD H. A. MABROUK

META-HEURISTICS PROGRAMMING AND ITS APPLICATIONS

By

EMAD H. A. MABROUK

Submitted in partial fulfillment of
the requirement for the degree of
Doctor of Informatics

KYOTO UNIVERSITY
KYOTO 606-8501, JAPAN
JANUARY 2011

Preface

The importance of Artificial Intelligence (AI) increases rapidly, and its implementations in real-life applications are widely spread. Moreover, extensive novel publications continually appear to introduce theoretical studies on AI and new applications in different fields of Computer Science and others, such as machine learning, information technology, communications, cryptography and security algorithms. One of the most well-known AI algorithms is the Genetic Programming (GP) algorithm. Since the first appearance of the GP algorithm, extensive theoretical and application studies on it have been conducted. The main advantage of the GP algorithm is its ability to deal with "computer programs" by using the tree data structure to represent solutions of a given problem. This tree-based representation enables the GP algorithm to evolve solutions conveniently, and to cover a wide range of applications. Recently, several versions and implementations of the GP algorithm have been proposed to improve its results. Nevertheless, several questions have been raised too about the complexity of the GP algorithm and the disruption effect of the crossover and mutation operators used in the algorithm.

The main contribution of this thesis is to propose a new set of AI algorithms that deal with computer programs to inherit the main advantage of the GP algorithm, and to overcome some of its drawbacks. In addition, the importance of local search is highlighted here by introducing a new set of breeding operators that are applied with a small scale of change to avoid the disruption of a solution. Specifically, each proposed algorithm incorporates the search strategy of a well-known meta-heuristic algorithm with the tree data structure through the proposed local search procedures. Moreover, the proposed algorithms are considered to introduce new alternatives to the GP algorithm.

Another contribution of this thesis is to consider two topics in cryptography and security algorithms, prime numbers generator and efficient Pseudorandom Number Generators (PRNGs). Recently, the usage of prime numbers has increased in various fields of security algorithms, e.g., public key cryptography algorithms and hash tables. On the other hand, PRNGs comprise a very important component in many topics of Computer

Science, especially computer security and cryptography, where developing new efficient PRNGs is an important problem. Therefore, the algorithms proposed in this thesis are used to attack those applications. Numerous mathematical formulas that can produce many distinct prime numbers are discovered. In addition, a new set of highly nonlinear functions that can be implemented in both hardware and software are generated to produce new efficient PRNGs.

Although this thesis is only one small step that just opens the door for introducing a new generation of efficient AI algorithms, the author hopes that the thesis will be helpful for further research and implementations in various applications.

Emad H. A. Mabrouk
January 2011

Acknowledgements

I would like to express my sincere appreciation to my supervisor, Prof. Masao Fukushima of Kyoto University, for many things. I thank him for accepting me to be one of his students, for guiding my research, and for reading and correcting my draft manuscripts carefully. Although I may have often troubled him, even outside the research framework, he has been always very understanding and encouraging. Moreover, he often spared his precious time for me to answer my questions. His eager and earnest attitude to the studies was respectable and incentive. Really, I would like to bear such a spirit in my mind, and make effort to improve my skill of research in the future.

I would also like to tender my acknowledgments to Dr. Abdel-Rahman Hedar of Assiut University, EGYPT. In fact, a lot of ideas in this thesis was first suggested by Dr. Hedar, and then improved and supported by him and Prof. Fukushima. He supported me and gave me a lot of kind advices by phone and E-mail. I am really grateful to him.

I am also thankful to Prof. Hideaki Sakai of Kyoto University and Prof. Hajime Kita of Kyoto University for serving on my dissertation committee. Further, I wish to thank Prof. Honglei Xu of Central South University, CHINA, for his careful review of my dissertation to improve its quality.

I would like to express my gratitude to all members of Prof. Fukushima's research group, Prof. Nobuo Yamashita, Dr. Shunsuke Hayashi and my colleagues, for the good scientific atmosphere they offered me during my study in Kyoto University. My especial thanks to Ms. Fumie Yagura, Mr. Koichi Nabetani, Mr. Noritoshi Kurokawa, and Mr. Kenji Hamatani, for their kindness and help which made my stay at the laboratory and the university easier.

I share the success of this dissertation with my wife and my kids. For four years, they have been always encouraging me to overcome difficulties encountered in advancing my research. Without their moral support, I could not complete this dissertation. So, I am

very grateful for their continual support and encouragement. In addition, I owe great thanks to my parents, brothers and sisters for all things that they gave me or taught me. Without their support, I would never have made any success.

I am greatly indebted to The Egyptian Ministry of Higher Education for managing the scholarship program and financing my entire study in Japan for four years. Moreover, I am grateful for all research facilities that have been provided by Kyoto University to achieve this study.

Lastly and above all, I would like to give back all the glory, honor and praise to **ALLAH**, the Creator and the Ultimate Source of all gifts in life. I would thank Him Almighty, for without Him, all these would not have been possible.

Contents

Chapter 1

Introduction

Artificial Intelligence (AI) is a branch of computer science that studies and designs intelligent agents. John McCarthy, the father of AI, defines AI as "the science and engineering of making intelligent machines" [87]. The actual research of AI started at the Dartmouth conference "The Dartmouth Summer Research Conference on Artificial Intelligence" in 1956 [103]. Moreover, the scientists who attended the conference became leaders of AI research for many decades, e.g., John McCarthy, Marvin Minsky, Allen Newell and Herbert Simon. By the middle of the 1960s, the US Department of Defense had heavily funded and established many AI research laboratories around the world [56]. In the early 1980s, the commercial success of expert systems enabled the AI research to be vitalized [87, 103], where an expert system is an AI program to simulate the knowledge and analytical skills of human experts. In the 1990s and early 21st century, AI has achieved its greatest success and it has expanded through several applications [103].

Nowadays, applications of AI are too numerous to list and can be seen in the infrastructure of every industry, for example, information technology [6, 17, 21, 71, 84, 91, 97, 116], communications [29, 78, 101, 102, 104], security [8, 22, 26, 42, 44, 52, 75], etc [15, 30, 35, 45, 46, 47, 48, 63, 77]. Therefore, researchers continually try to develop new efficient AI algorithms or improve the current ones to maximize their benefits [4, 14, 27, 33, 49, 51, 58, 82, 83, 93, 96, 108, 114, 123]. On the other hand, many different problems in applications of AI require a discovery of a "computer program" as an output when presented with particular inputs [67]. Those problems include, for example, symbolic regression [110, 111, 121], machine learning [18, 80, 117], pattern recognition [67, 81, 124], and generating an intrusion detection system in network security [1, 40, 42, 94], 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 [32, 67].

The Genetic Programming (GP) algorithm is known as the most suitable algorithm used to solve such problems [5, 19, 25, 64, 67, 68, 69, 70, 71, 96, 99, 123]. The GP algorithm inherits the basic idea of the well-known meta-heuristics Genetic Algorithms (GAs) [17, 25, 38, 39, 88, 112] and deals with working computer programs obtained from a given problem [67, 68, 69, 70, 76]. The main difference between GAs and the GP algorithm lies in the representation of a solution and the application fields. GAs usually use an array of bits to represent a solution [38, 112], while the GP algorithm deals with computer programs represented as trees [66, 67, 68, 69, 70, 76, 120]. However, in spite of the popularity and great importance of the GP algorithm, a number of authors have pointed out that its main breading operators suffer from some drawbacks [11, 54, 57, 59, 82, 92, 93, 108, 123]. Altering a node high in a tree may result in serious disruption of the subtree rooted at that node. Therefore, there have been many attempts to edit GP operators to make changes in small scales [55, 60, 61, 85]. In addition, the importance of local search has been well recognized, and methods of improving the local structure of individuals have been developed [47, 54, 79, 82, 83].

In this thesis, we introduce a set of new algorithms as alternatives to the GP algorithm in order to accommodate more application areas. All of these algorithms use the tree data structure to represent a solution, and search for a desirable computer program as an output. In addition, we use new local search procedures over a tree space as alternative operators to the classical breading operators in the GP algorithm. The new procedures aim to generate a set of trial programs in the neighborhood of a program by making moderate changes in it. Finally, the proposed algorithms can be gathered under a unified framework which we call Meta-Heuristics Programming (MHP).

In the next section, we give a brief introduction of meta-heuristics, and discuss two well-known meta-heuristics, Genetic Programming and Tabu Search, that are considered the ancestors of the proposed algorithms. Finally, the aims, contributions and the organization of this thesis are stated in Section 1.2.

1.1 Meta-Heuristics

Meta-heuristics are high-level strategies that guide other heuristics in a search for feasible solutions [3, 10, 32, 49, 98, 102]. The term “meta-heuristics”, first used by Glover [30], contains all heuristics methods that show evidence of achieving good quality solutions for a problem of interest within an acceptable time. In addition, meta-heuristics are often considered to be a promising choice for solving combinatorial optimization problems,

where exploring the exact solutions for these problems becomes very hard due to some limitations like an extremely large running time. Nevertheless, meta-heuristics offer no guarantee of obtaining global best solutions [32, 98, 102].

The most commonly used data structure types in meta-heuristics are bit-strings and real-valued vectors, except for Genetic Programming. Moreover, meta-heuristics are typically applied to problems that can be modeled or transformed to optimization problems [3, 13, 32, 98, 102]. In terms of the process of updating solutions, meta-heuristics can be classified into two classes; population-based methods and point-to-point methods [10, 13, 32]. 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. Additional classifications for meta-heuristics can be seen in [10, 13], in terms of methodology, memory usage, neighborhood structure, etc. In the next subsection, the GP algorithm and its procedures will be discussed in detail to show the idea of the tree data structure, and to point the drawbacks of the current breading operators. Subsection 1.1.2 highlights the principles of the well-known Tabu Search. These two algorithms will be adapted later to produce new evolutionary algorithms that deal with the tree data structure, using a new set of breading operators.

1.1.1 Genetic Programming

The GP algorithm is an evolutionary algorithm (EA) inspired from the biological processes of natural selection and survival of the fittest [67, 68, 69, 70, 71, 76]. The GP algorithm evolves a population of computer programs represented as trees to find an acceptable solution for a given problem. The first proposal of "tree-based" genetic programming was given by Cramer [21] in 1985. This work was popularized by Koza [67, 68, 69, 70], and subsequently, the feasibility of this approach in well-known application areas has been demonstrated [5, 19, 25, 64, 71, 96, 99, 123]. The algorithm starts the search process with a "population" of computer programs that are usually generated randomly. Then, each program in the population is evaluated and some good programs are selected and recombined using the mutation and the crossover operators to breed a new population of programs. This process of selection and recombination to breed new programs is iterated with the hope of driving the population toward an optimal solution. When a termination condition is satisfied, the search process stops and the best program found is considered the output of the algorithm. Fig. 1.1 represents a flowchart of the GP algorithm.

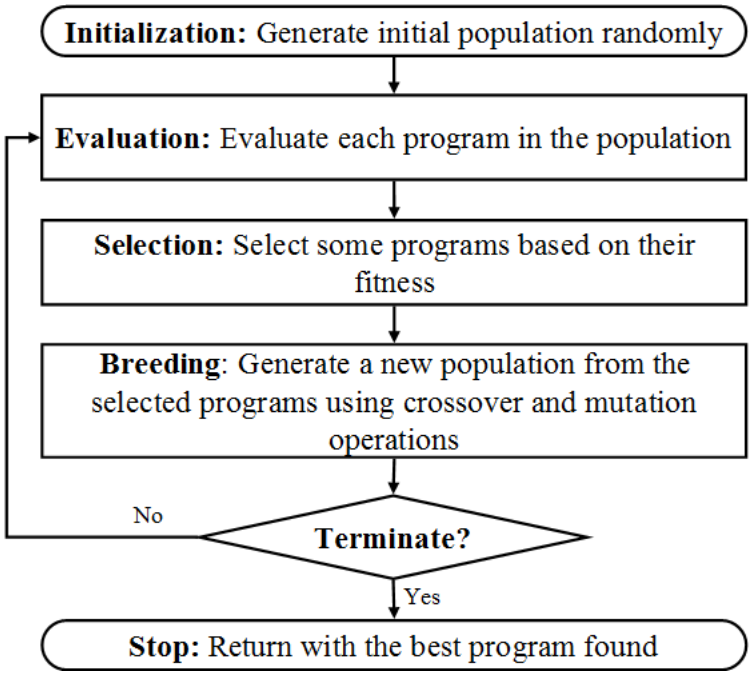

Figure 1.1: Flowchart of GP algorithm

The computer programs treated in the GP algorithm are represented as trees in which leaf nodes are called terminals and internal nodes are called functions [67, 68, 69, 76]. Depending on the problem at hand, the user defines the domains of terminals and functions. In the coding process, the tree structure of a solution should be transformed to an executable code. Usually, these codes are expressed to closely match the Polish notation of logic expressions [25]. Fig. 1.2 shows two examples of individuals represented as trees, along with their executable codes in the Lisp computer languages [16, 62, 68, 70].

One of important improvements of GP has been made through the use of Automatically Defined Functions (ADFs) for reusing codes [69]. The ADFs are sub-trees that can be used as functions (called defun, subroutines, subprograms, or modules) of dummy arguments in the main tree of a program to exploit symmetries and modularities in the problem environments. In the standard GP algorithm with the ADF technique, each ADF is defined in a separate function-defining branch as a part of a program. In addition, for each ADF, the user must specify the number of dummy arguments and the function set which is allowed to contain other ADFs. The main program is defined in the result-producing branch that yields the fitness value of this program. For each program in the population, the result-producing branch is allowed to call functions from the function set that includes the original primitive functions as well as the ADFs defined for this program. In fact, the ADF technique has been successful in improving the performance

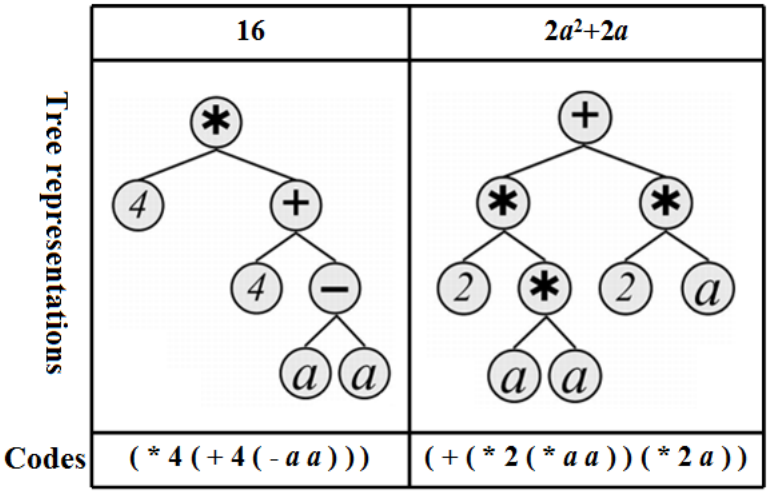

Figure 1.2: Examples of GP representation

of GP for a set of problems, see numerical experiments in [69]. Fig. 1.3 shows an example of the structure of a GP program that contains two function-defining branches (two ADFs) and a result-producing branch.

The selection technique is a key issue in the GP algorithm, since it affects other steps in a direct way [4, 12, 106, 107, 122]. Moreover, it implicitly affects the diversity of the current population. For example, in the roulette wheel selection [12, 122], the best program in the old population has the highest probability of propagating and producing offsprings in the new population, which reduce the diversity in that population. In the literature, e.g., [4, 12, 19, 106, 107, 122], several selection techniques are introduced and discussed extensively, all of which make use of the fitness values of the current population. However, the most common selection techniques used in the GP algorithm are the roulette wheel selection and the tournament selection. In the tournament selection of size k, k programs are chosen randomly from the current population, and the one with the best fitness is selected as the winner. However in the roulette wheel selection, the fitness values of all programs in the population are represented as contiguous segments of a line, where the length of each segment is equal to its corresponding fitness value. A random number, between 0 and the sum of all fitness values, is generated and the program corresponding to the part that contains the generated random number is selected as the winner. For more details and discussions about these selection strategies and others, see [12, 19, 106, 122].

The crossover and mutation operators are the main breeding operators in the GP

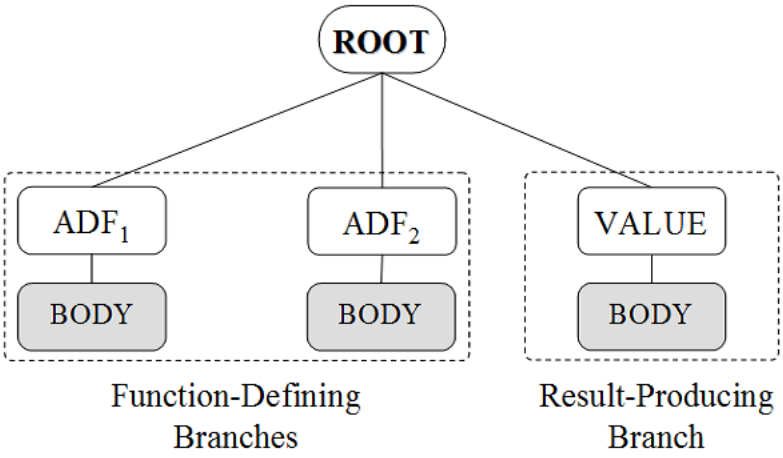

Figure 1.3: Example of representing a program in GP using ADF technique

algorithm. Since programs in the GP algorithm are represented by trees, the crossover is applied by choosing two programs (trees) randomly, choosing a node randomly from each tree, and exchanging the two subtrees rooted at these two nodes to get offsprings. On the other hand, the mutation operator is applied for one program (tree) chosen randomly from the pool set. Then, one can get a new offspring by choosing a node randomly and exchanging the subtree rooted at this node by a new one that is generated randomly [20, 67, 68, 69, 70, 71, 76, 108]. The crossover and mutation operators are summarized in the following two procedures, and Fig. 1.4 illustrates an example of applying the crossover and mutation operators in the GP algorithm.

Procedure 1.1. $[Y_1, Y_2] = \texttt{Crossover}(X_1, X_2)$

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

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

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

Procedure 1.2. $[Y] = \texttt{Mutation}(X)$

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

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

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

To generate a new population form the current one, the GP algorithm prepares a pool of programs selected from the current population using a selection technique. Then, it

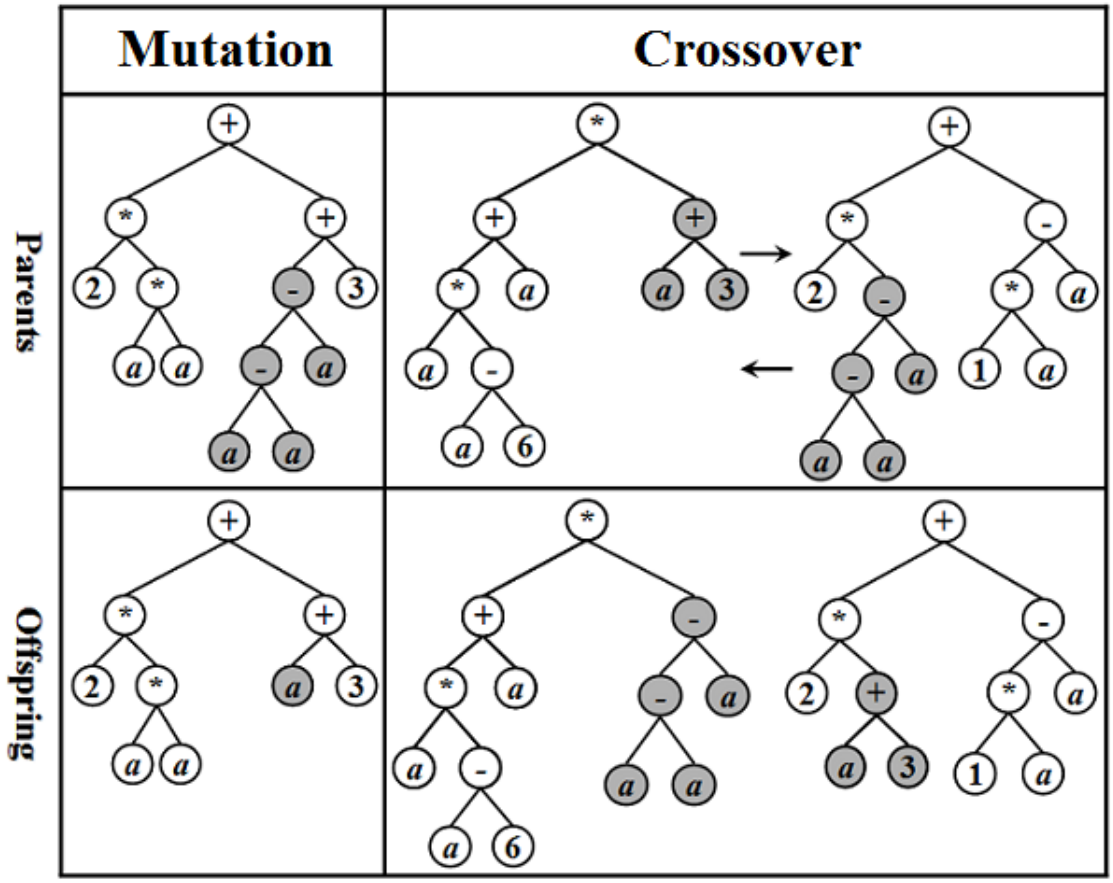

Figure 1.4: Mutation and crossover operations in GP

starts with a new empty population and repeats the following steps until the new population becomes full: First, it picks up an operator randomly from the set of breading operators, i.e., reproduction (copy), crossover and mutation operators, based on a predetermined probability value. Second, the algorithm picks up one or two program(s), based on the selected operator, randomly from the pool set. Third, the algorithm generates new offsprings by applying the selected operator to the selected program(s). Finally, new offsprings are added to the new population. Once the new population becomes full, it will replace the old one [67, 68, 69, 70].

The computational effort (CE) has been introduced by Koza [68] to measure the computational costs required for GP to solve a problem, and its value is based on some

data collected from a set of independent runs. The formula of CE is given as follows:

$$\begin{aligned} \text{CE} &= \min_i I(\texttt{nPop}, i, z), \\ I(\texttt{nPop}, i, z) &= \texttt{nPop} * R(z) * i, \\ R(z) &= \left\lceil \frac{\log(1-z)}{\log(1-P(\texttt{nPop}, i))} \right\rceil, \\ P(\texttt{nPop}, i) &= \frac{N_s(i)}{N_{all}}, \end{aligned} \tag{1.1}$$

where z is a positive number less than 1, $N_s(i)$ is the number of successful independent runs up to generation i, and N_{all} is the total number of runs. As in Koza [68], $P(\texttt{nPop}, i)$ represents the cumulative probability of success up to generation i, $R(z)$ represents the number of independent runs required to produce a solution up to generation i with probability z, and $I(\texttt{nPop}, i, z)$ is the number of programs that must be processed to get a satisfactory solution, with probability z, using population size `nPop` up to generation i.

Indeed, the crossover and mutation operators have extensively been studied, and many effective settings of these operations have been proposed to deal with a wide variety of problems [20, 57, 60, 69, 70, 76, 85, 93, 97, 109]. These efforts have popularized the GP algorithm and expanded the range of applications. Nevertheless, 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 [65, 93] and references therein. This problem produces a high computational cost in the GP algorithm due to the growth of individuals in size and complexity during the evolution process [93, 108]. Moreover, it is reported in [9] 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 [20, 55, 59, 82, 83, 92, 93]. Altering a node high in the tree may result in serious disruption of the subtree below it.

These drawbacks of the crossover and mutation operators motivate researchers to propose some hypotheses about the causes behind these phenomena [11]. One of these hypotheses is "Protective effect against destructive nature of modifying operators", see [11] and [125] for more details. Since these operators are main breeding operators in the GP algorithm, there have been many attempts to edit them to make changes in small scales, for example by using natural language processing [55, 71]. New crossover operators, such as brood crossover [106, 107], context-aware crossover [57, 85], homologous crossover and crossover-hill climbing [9], ripple crossover [60], and depth-fair crossover

[61], are practical remedies for these problems. Moreover, the importance of local search and improving the local structure of individuals have been addressed [54, 79, 82, 83].

1.1.2 Tabu Search

Tabu Search (TS) is a heuristic method originally proposed by Glover [30] in 1986. Afterward, it has spread quickly to become one of the most powerful meta-heuristic methods for tackling difficult combinatorial optimization problems [31, 35, 36, 47]. 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 [33, 36, 47].

Suppose that the goal is to minimize the objective function $f : S \to \mathbb{R}$ 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. Algorithm 1.3 shows an outline of the TS algorithm.

Algorithm 1.3. *(Tabu Search Algorithm)*

1. ***Initialization.*** *Choose an initial solution s, and set the* `TL` *being empty.*
2. ***Main Loop.*** *Repeat the following Steps 2.1-2.3 until a termination condition is satisfied, and then proceed to Step 3.*
   - **2.1.** *Generate a neighborhood $N(s) \subset S$ of the current solution s based on the tabu restrictions.*
   - **2.2.** *Let the best solution s' in $N(s)$ replace the current one, i.e., set $s = s'$.*
   - **2.3.** *Using the* `TL`*, mark the current solution s as a tabu move.*
3. ***Termination.*** *If a termination condition is satisfied, then go to Step 5. Otherwise, go to Step 4.*

4. ***Diversification.*** *Use a diversification strategy to generate a new diverse solution s and go to Step 2.*
5. ***Intensification.*** *Improve the elite solutions obtained.*
6. ***Stop.*** *Stop and return with the best program found.*

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 [31, 36, 47]. 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 perform 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 [35]. 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.

1.2 Aims, Organization and Contributions

The tree-based representation of a solution is a great advantage of the GP algorithm. This representation enables the GP algorithm to evolve solutions conveniently, and to cover a wide range of applications. It is worthwhile to mention that although there are many meta-heuristic alternatives to GAs, the GP algorithm is almost the only meta-heuristics adapted for searching the space of computer programs. In spite of that, the crossover and mutation operations in the GP algorithm suffer from some drawbacks as we explained in the previous section. Therefore, our aim in this thesis is to propose new algorithms as alternatives to the GP algorithm. The proposed algorithms use the

tree-based representation strategy to exploit the advantages of the tree data structure. However, the search strategies and the set of breading operators of the proposed algorithms will be different than those of the GP algorithm. The performance of the new algorithms will be discussed through extensive numerical experiments on a set of test problems. In addition, some applications for the proposed algorithms will be introduced as well. The organization of the thesis is as follows.

In Chapter 2, we introduce the common aspects of the proposed algorithms. First, we describe the tree representation of a program and the strategy of the coding process. In particular, we represent a program as a tree like the GP algorithm. However, the coding process for a tree and the way of generating a random tree are different than those used in the GP algorithm. Then, a new set of local search procedures over a tree space will be introduced as the main breading operations for the proposed algorithms. Using these search procedures, various meta-heuristics can be reconsidered to deal with computer programs using the tree data structure in a unified framework which we call Meta-Heuristics Programming (MHP). Finally, the MHP framework will be introduced at the end of Chapter 2.

In Chapter 3, we introduce a new algorithm that deals with computer programs, and we call it the Tabu Programming (TP) algorithm. This algorithm inherents the search strategy of the classical TS algorithm, and uses the tree data structure to represent a solution. The local search procedures introduced in Chapter 2 are used to generate new trial solutions from the current one. The performance of the TP algorithm is discussed through extensive numerical experiments.

In Chapter 4, a new hybrid evolutionary algorithm, called the Memetic Programming (MP) algorithm, is introduced. The MP algorithm hybridizes the standard GP algorithm with a new local search algorithm over a tree space to intensify elite programs generated by the GP algorithm. The new local search algorithm is called the Local Search Programming (LSP) algorithm. Therefore, each generation of the MP algorithm is composed of two phases, a diversification phase using the GP strategy, and an intensification phase using the LSP algorithm to improve results of the diversification phase. The performance of the MP algorithm will be shown through several numerical experiments.

In Chapter 5, we use the proposed algorithms in two different types of applications in the information technology. Specifically, the MP algorithm is used to discover a new set of formulas that can produce many distinct prime numbers. On the other hand, the TP algorithm is used to generate numerous high efficient nonlinear functions that can be used as strong pseudorandom number generators. The prime numbers and the pseudorandom number generators are very important tools in several applications of

information technology, i.e., cryptography and network security algorithms, hash tables, and many fields of Computer Science.

Finally, Chapter 6 summarizes the main contributions of the thesis and discusses some possible future work.

Chapter 2

Meta-Heuristics Programming

2.1 Introduction

Our aim in this thesis is to introduce new algorithms as alternatives to the GP algorithm. Therefore, we introduce a new set of tools that helps to adapt meta-heuristics to deal with the tree data structure. These tools include the way to represent a solution as a tree, and some suggested breeding operators to produce programs from the current one(s). In this chapter, we introduce these tools associated with the main framework of the proposed algorithms. In the next section, the tree representation of a solution will be introduced as well as the coding strategy supported with some examples. In addition, a new set of breeding operations over a tree space will be introduced in Section 2.3. In fact, for some algorithms, these procedures can perform several tasks as we will see in the coming chapters. Using these breeding operators, various meta-heuristics can be adapted to deal with tree data structures. The universal framework of the proposed algorithms will be introduced Section 2.4.

2.2 Representation of Individuals

For all algorithms proposed in this thesis, a solution generated by each algorithm is called a program and it is represented as a tree consisting of one or more "gene(s)". Each gene represents a subtree consisting of some external nodes called terminals and some internal nodes called functions. Genes inside a program are linked together by using a suitable linking function, e.g., "+" for the symbolic regression problem, to produce the final form of a solution. In the coding process, the tree structure of a program is transformed into an executable code called genome. We code genes using a strategy that differs from the one used in the standard GP, where each gene is represented as a linear symbolic string

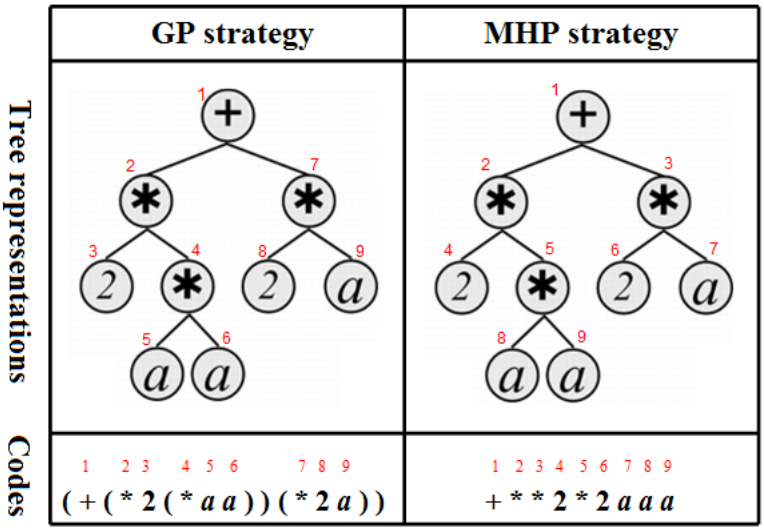

Figure 2.1: The tree structure and the coding representation of a solution

composed of terminals and functions. Fig. 2.1 shows an example of tree structures along with their executable codes for a gene using the GP strategy and our strategy.

As to the initial program(s) of the algorithm, we generate its genes one by one. In addition, 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, based on the functions and terminals that are generated randomly within the gene. 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. 2.2A. By converting this gene to its tree representation (Fig. 2.2B), one can see that the last 4 elements are unnecessary elements. Consequently, we can delete these unnecessary elements and keep the rest as in Fig. 2.2C.

Once the initial program or the initial population is generated, it will be evolved and improved using the breeding operations based on the considered algorithm. For each problem to solve, the sets of functions and terminals, and the fitness function must be

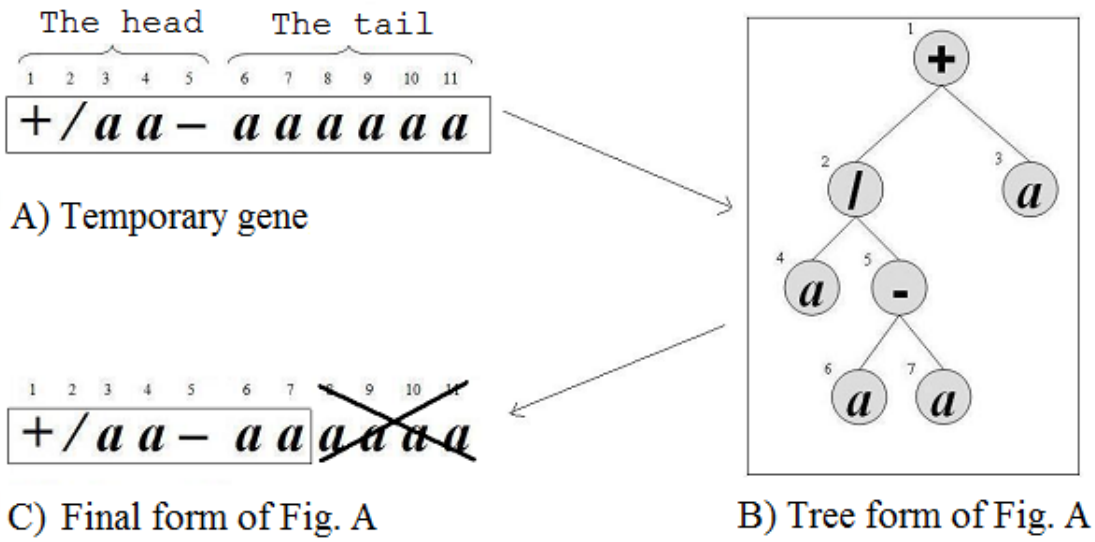

Figure 2.2: Constructing a new gene

determined before calling the algorithm. In addition, the following set of representation parameters must be determined as well.

- `hLen`: The head length for every gene generated randomly in the initial program.
- `MaxLen`: The maximum length, i.e., the number of nodes, of a gene allowed in the search process.
- `MaxDepth`: The maximum depth of a gene, where the depth of a tree is the number of links in the path from the root of this tree to its farthest node.
- `nGenes`: The number of genes in each program.
- `LnkFun`: The function used to link genes in each program.

It is worthwhile to note that the `MaxLen` and the `MaxDepth` parameters are used (one of them or together) to bound the size of trees during the search process, since the size of those trees can increase rapidly according to the breeding operations. In addition, adapting each program to contain more than one genes, i.e., `nGenes` ≥ 1, increases the probability of finding suitable solutions, and enables the algorithm to deal with more complex problems [27].

2.3 Local Searches over Tree Space

In this section, some local search operators over a tree space are introduced. These operators aim to generate a new tree in a neighborhood of the current tree. We discuss two types of local searches; *static structure search* and *dynamic structure search* [47, 51, 82, 83]. Static structure search aims to explore the neighborhood of a tree by altering its nodes without changing its structure. On the other hand, dynamic structure search changes the structure of a tree by expanding its terminal nodes or cutting its subtrees. *Shaking* operator is shown as a static structure search, while *Grafting* and *Pruning* operators are introduced as a dynamic structure search.

Before proceeding to the description of Shaking, Grafting and Pruning procedures, we introduce some basic notations. For a tree X, we define its size, leaf number and depth as follows.

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

2.3.1 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 2.1 states the formal description of shaking search, while Fig. 2.3 shows an example of shaking search that alters two nodes of X. In Procedure 2.1, $\lambda \in [1, |X|]$ is a positive integer that represents the number of nodes to be changed, and λ must be determined before calling the procedure.

Procedure 2.1. $\tilde{X} = \texttt{Shaking}(X, \lambda)$

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

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

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

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

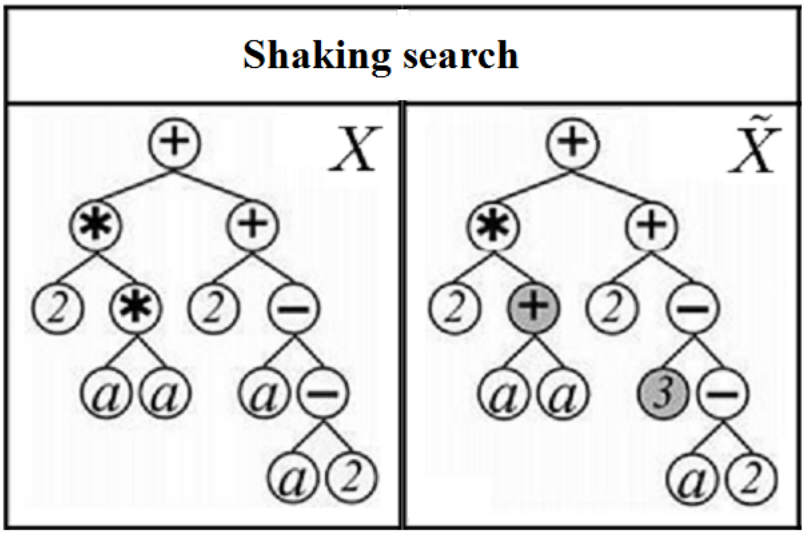

Figure 2.3: Example of shaking search ($\lambda = 2$)

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

Step 3. *Return.*

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

$$N_S(X) = \{\tilde{X} | \tilde{X} = \texttt{Shaking}(X, \lambda), \lambda = 1, \ldots, |X|\}. \tag{2.1}$$

2.3.2 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[1]. 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 2.2 states the formal description of grafting search, where μ refers to the number of branches of depth[2] ζ that will be added to X. In addition, μ and ζ must be determined before calling the procedure. Figure 2.4 shows an example of grafting search that alters two nodes in X by two branches in $\tilde{X}$.

Procedure 2.2. $\tilde{X} = \texttt{Grafting}(X, \mu, \zeta)$

[1]Throughout the thesis, the term "*branch*" is used to refer to a subtree, see [23].

[2]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 $d(B)$.

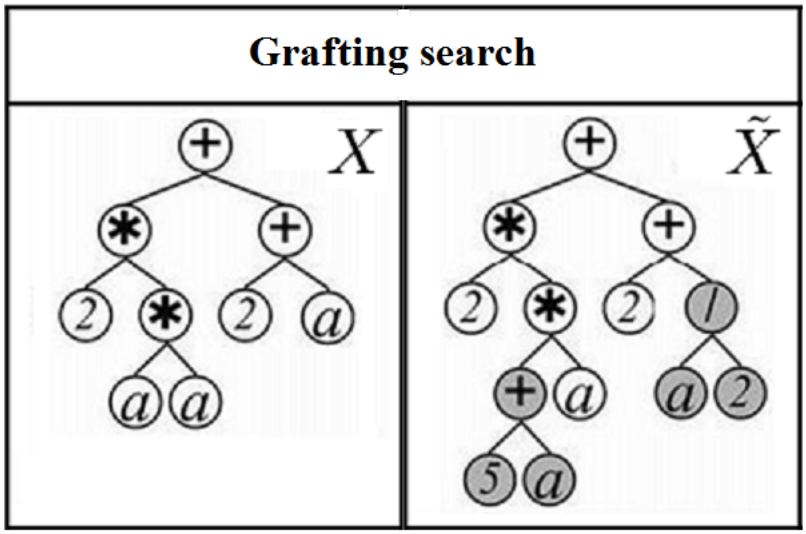

Figure 2.4: Example of grafting search ($\mu = 2$ and $\zeta = 1$).

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

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

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

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

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

Step 3. *Return.*

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

$$N_G(X) = \Big\{\tilde{X} | \tilde{X} = \texttt{Grafting}(X, \mu, \zeta), \\ \mu = 1, \ldots, l(X), \zeta = 1, \ldots, \zeta_{max}\Big\}, \tag{2.2}$$

where ζ_{max} is a predetermined positive integer.

2.3.3 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 $d(\tilde{X}) \leq d(X)$. The formal description of pruning search is given below

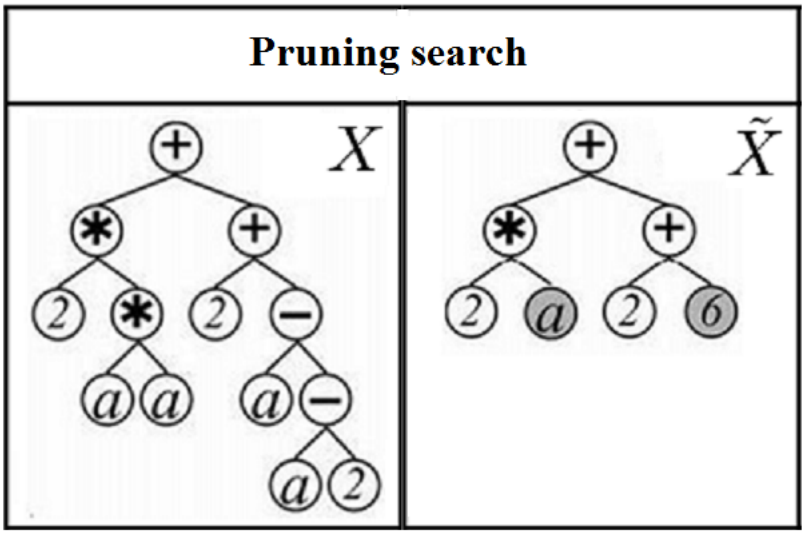

Figure 2.5: Example of pruning search ($\nu = 2, \zeta_1 = 1, \zeta_2 = 2$).

in Procedure 2.3, where ν refers to the number of branches of depth ζ that will be replaced by terminals. In addition, ν and ζ must be determined before calling the pruning procedure. Fig. 2.5 shows an example of pruning search that cuts two branches in X.

Procedure 2.3. $\tilde{X} = \texttt{Pruning}(X, \nu, \zeta)$

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

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

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

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

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

Step 3. *Return.*

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

$$N_P(X) = \Big\{\tilde{X} | \tilde{X} = \texttt{Pruning}(X, \nu, \zeta), \\ \nu = 1, \ldots, f(X), \zeta = 1, \ldots, d(X)\Big\}, \tag{2.3}$$

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

The values of λ, μ, ν and ζ in the proposed local search procedures, Procedures 2.1, 2.2 and 2.3, 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 throughout this thesis, we choose the values of λ, μ and ν 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 $\lambda = 1$, $\lambda = 2$ and $\lambda = 3$, respectively. In other words, the first trial solution ($\lambda = 1$) is generated by choosing one node randomly and replacing it by an alternative node, while the second trial solution ($\lambda = 2$) is generated by choosing two nodes randomly and replacing them by two alternative nodes, and so on.

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 a diversity and escaping from the current position are needed. Actually, keeping diversity is one of the major issues that should be taken into account in designing efficient global search techniques [19].

To determine all neighbors around a tree X, we need a lot of information like the size of X, the number of function nodes in X, the number of terminal nodes in X, the number of functions in the function set, and the number of terminals in the terminal set. Moreover, we need to determine number of arguments for each function node in case of using the shaking procedure, and the depth of branches that will be added to or cut from X in case of using the grafting and pruning procedures. All these required information make it difficult (may be impossible) to determine all neighbors around a tree X. Therefore, the GP research community always proposes stochastic search algorithms that select a set of random candidate solutions in the neighborhood of solutions in the current population. In this chapter, Procedures 2.1-2.3 behave as stochastic searches due to the random choices in Step 2 in each procedure. In other words, by calling any of these procedures several times for the same tree X and using the same parameters, one may get a different $\tilde{X}$ for each call.

2.4 Meta-Heuristics Programming Framework

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. It has been reported that the performance of search methods, especially meta-heuristics, varies even when they are applied to the same problem [14, 24]. Moreover, the concept of No Free Lunch [118, 119] 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 has inspired us to extend meta-heuristics to deal with applications by utilizing a tree data structure, and using the local search procedures introduced in Section 2.3. We call the new framework Meta-Heuristics Programming.

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 [31, 34, 74]. 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 tree(s) can be adapted through the following four 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.

- 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 2.3 are used in the TRIALPROGRAM procedure, while the UPDATEPROGRAM procedure depends on the type of invoked meta-heuristics.

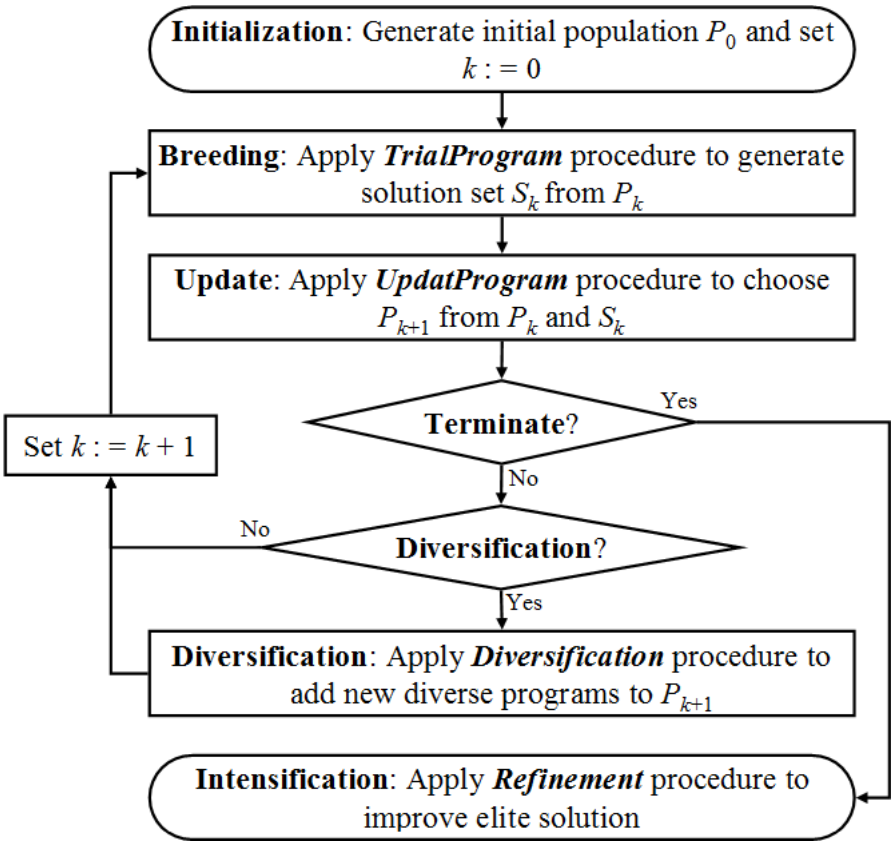

Figure 2.6: The MHP flowchart.

The main structure of the MHP framework is shown in Fig. 2.6. 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. 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. 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.

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. 2.6, 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.

Chapter 3

Tabu Programming

3.1 Introduction

The TS algorithm has shown a good performance compared to the GAs in some implementations [47, 50]. This encourages us to propose a new algorithm that follows the search strategy in the TS algorithm and uses the tree data structure. Specifically, the purpose of this chapter is to propose a new algorithm called Tabu Programming (TP) algorithm. This algorithm searches for a working computer program, represented as a tree, as an output. Moreover, we use the local search procedures over a tree space, described in Section 2.3, as the breeding operators of the proposed TP algorithm. 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. Through extensive numerical experiments, the TP algorithm is shown to work effectively and, in fact, it outperforms the GP algorithm on some benchmark test problems. From the numerical experiments, we may claim that the TP algorithm provides a promising approach for problem solving.

After presenting the TP algorithm in next section, we report numerical results in Section 3.3 for three types of benchmark problems; the symbolic regression problem, the 6-bit multiplexer problem, and the 3-bit even-parity problem. Finally, the conclusion makes up Section 3.4.

3.2 Tabu Programming Algorithm

The Tabu Programming 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 2.3. The first idea of the TP algorithm was proposed by Hedar and Fukushima [49] in a short paper presented in a 2006 workshop. Then, in 2007, Balicki [7] 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 chapter, 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 3.1 shows the main structure of the TP algorithm, and its formal description is given below.

Algorithm 3.1. *(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* `nTL`*, n_T, n'_T, n''_T, ζ_1, ζ_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 $N_S(X_k)$, Equation (2.1), 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 $N_G(X_k)$, Equation (2.2), 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 $N_P(X_k)$, Equation (2.3), 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 Test.*** *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 3.1 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. 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

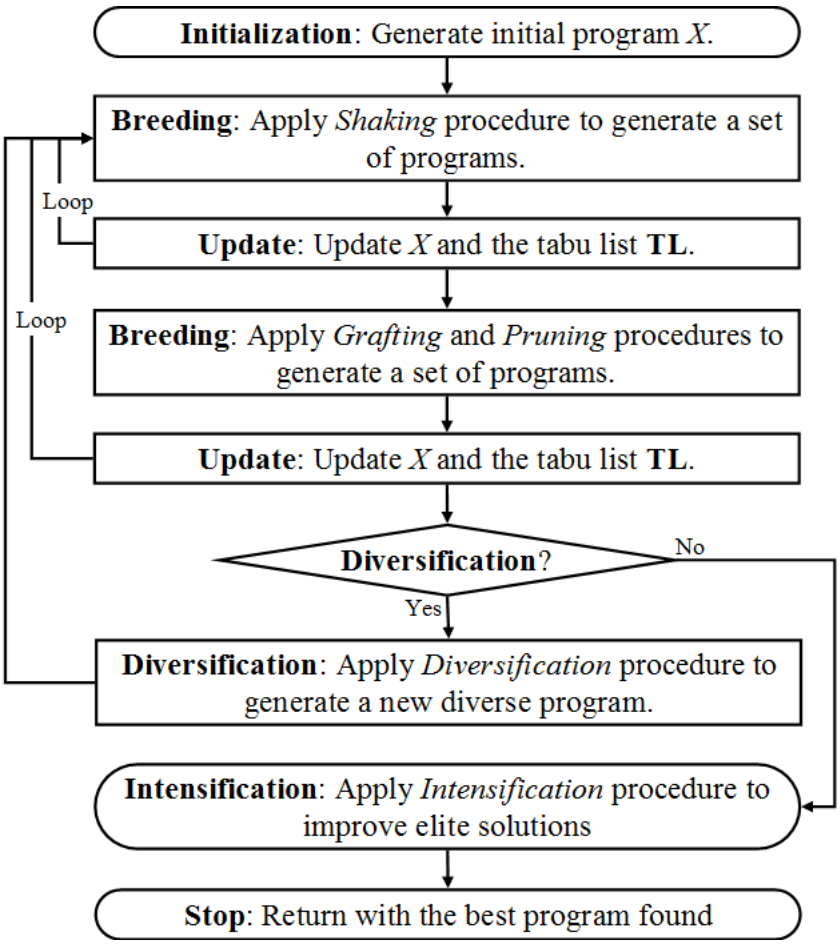

Figure 3.1: The TP flowchart.

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 2.3, where n_k is a positive integer that represents the number of trial programs gener-

ated 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 $N_S(X_k)$ in (2.1), i.e., $S_k = \{\tilde{X}_\lambda | \ \tilde{X}_\lambda = \texttt{Shaking}(X_k, \lambda),\ \lambda = 1, 2, ..., n_T\}$. 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 $N_G(X_k)$ in (2.2) and $N_P(X_k)$ in (2.3) using the grafting and pruning procedures, respectively, i.e., $S'_k = \{\tilde{X}_\mu | \ \tilde{X}_\mu = \texttt{Grafting}(X_k, \mu, \zeta_1),\ \mu = 1, 2, ..., n'_T\}$ and $S''_k = \{\tilde{X}_\nu | \ \tilde{X}_\nu = \texttt{Pruning}(X_k, \nu, \zeta_1),\ \nu = 1, 2, ..., n''_T\}$. 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 3.1, 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.

3.3 Numerical Experiments

In this section, 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.

3.3.1 Set of Parameters in TP

From Section 2.2 and Section 3.2, 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.
  - `MaxDepth`: The maximum depth of a gene.
  - `nGenes`: The number of genes in a program.

- `LnkFun`: The function used to link genes in each 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 3.1 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).

Through the numerical experiments in this chapter, we used $\zeta_1 = 1$, $\zeta_2 \geq 3$ and $n^* \geq 3$ in Algorithm 3.1, and the maximum depth for a program is set to be `MaxDepth` $= 10$.

3.3.2 Test Problems

In the rest of this section, we test the performance of the TP algorithm through the SR-QP, SR-POLY-4, 6-BM and 3-BEP problems, see Appendix A for more details. In addition, we use the following settings, unless otherwise stated, for our test problems:

- For the SR-QP problem, the terminal set is the singleton $\{x\}$, and the function set is $\{+, -, *, \%\}$, where % is called "the protected division", and $x\%y := x$ if $y = 0$; $x\%y := x/y$ otherwise.

- For the SR-POLY-4 problem, the set of independent variables $\{x_1, x_2, x_3, x_4\}$ is regarded as the terminal set, and the function set is $\{+, -, *, \%\}$.

- For the 6-BM problem, the set of arguments $\{a_0, a_1, d_0, d_1, d_2, d_3\}$ is used as the terminal set, and the set of Boolean functions {AND, OR, NOT, IF} is used as the function set, where IF(x, y, z) returns y if x is true, and it returns z otherwise.

- For the 3-BEP problem, the set of arguments $\{a_0, a_1, a_2\}$ is used as the terminal set, and the set of Boolean functions {AND, OR, NAND, NOR} is used as the

Table 3.1: Standard values of the TP parameters.

Parameter	Value		
	SR-QP	6-BM	3-BEP
`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
`LnkFun`	+	IF	AND

function set. In fact, the GP research community considers evolving the N-BEP function by using those Boolean functions as a good benchmark problem for testing the efficiency of new GP techniques [68, 114].

For the SR-QP and SR-POLY-4 problems, the algorithm starts a run by generating a dataset randomly under the conditions described in Subsections A.1.1 and A.1.2, respectively. Moreover, the generated dataset will be fixed during the running time of the algorithm for each independent run.

3.3.3 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. Second, we study the efficiency of the local search procedures introduced in Section 2.3 and its effects on the TP algorithm. Finally, 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 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.

Table 3.2: Performance of the TP algorithm with different tabu list structures.

TL	AV	ME	R%
$\texttt{TL}_1$	4,171	5,000	31
$\texttt{TL}_2$	3,476	4,239	67
$\texttt{TL}_3$	1,485	1,015	95
$\texttt{TL}_4$	795	681	100

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-QP problem. The values of the TP parameters are set as in Table 3.1, except `FunCnt` = 5000. The results of the TP algorithm with the above-mentioned four `TL` structures are compared in Table 3.2, 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 section, we adjust the `TL` to store the last `nTL` visited solutions in its

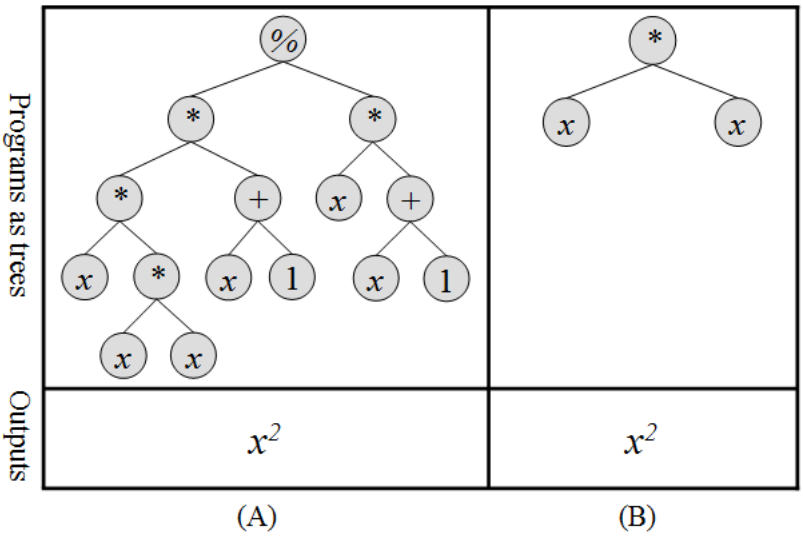

Figure 3.2: Two different trees that are represent the same formula.

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.

Suppose that the TP algorithm generates the two trees in Fig. 3.2 as two programs for the SR-QP problem. One can note that these two trees represent the same formula, i.e., x^2. However, the first tree Fig. 3.2(A) can be evolved easily to get the optimal solution of the SR-QP problem, by changing the root function node from "%" to be "+". On the other hand, extensive steps of grafting, shaking and maybe pruning procedures are needed to evolve the second tree to get the optimal formula. Therefore, a solution for a given problem can be represented by many different trees generated by the TP algorithm. Moreover, the TP algorithm can accept a new tree even if the formula represented by this tree is already represented by a different tree that already exists in the TL. In fact, this is acceptable since the TP algorithm deals with trees not formulas.

Efficiency of Local Search Procedures

In this part, we try to get some indicators about the efficiency of the local search procedures. We have applied four different versions of the TP algorithm to the SR-QP problem, and observed the influence of those differences on the performance of the TP algorithm

Table 3.3: Comparison among four versions of the TP algorithm for the SR-QP problem.

Algorithm	AV	ME	R%
TP-ShPr	1,948	2,500	28
TP-ShGr	1,608	2,021	57
TP-GrPr	1,611	1,772	67
TP	1,238	1,029	85

1. TP-ShPr: TP algorithm with the shaking and the pruning procedures.

2. TP-ShGr: TP algorithm with the shaking and the grafting procedures.

3. TP-GrPr: TP algorithm with the grafting and the pruning procedures.

4. TP: The proposed TP algorithm as described in Algorithm 3.1.

We performed 100 runs for each version of the algorithm using the parameter values shown in Table 3.1, except for hLen = 15. Comparisons with respect to the average and the median of the number of fitness evaluations as well as the rate of success are displayed in Table 3.3. In addition, Fig. 3.3 shows the relation between the number of fitness evaluations and the rate of success to show the efficiency of each algorithm.

From these comparisons, we can observe that the proposed TP algorithm has the best performance among the four versions. On the other hand, the TP-ShPr algorithm is the worst one in terms of AV, ME and R. In addition, the importance of the local search can be confirmed by comparing the results of the TP and TP-GrPr algorithms. Using the shaking procedure to refine the current solution and search its neighborhood locally, the TP algorithm can reduce the number of fitness evaluations and increase the rate of success significantly.

It is important to note that the parameter `hLen` has a great influence on the performance of the TP-ShGr and TP-ShPr algorithms. In the TP-ShGr algorithm, the size of a current tree always increased during the search, which means that starting the TP-ShGr algorithm with a small value of `hLen` may increase the probability of finding a good solution. On the other hand, the size of a current tree always decreased during the search in the TP-ShPr algorithm, which suggests that starting the TP-ShPr algorithm with a large value of `hLen` may increase the probability of finding a good solution. However, the

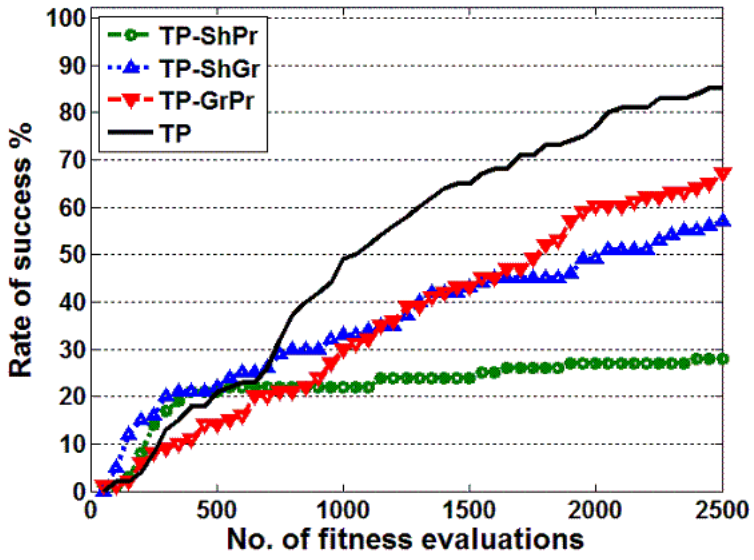

Figure 3.3: Comparison among four versions of the TP algorithm.

TP-ShPr algorithm that relies on the shaking and pruning procedures seems to be an unwise choice because the tree will eventually reduce to one node and this will prevent the algorithm from finding a good solution.

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 3.1.

The computational results are displayed in Table 3.4. 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 $2 \leq$ `nGenes` ≤ 5 for the SR-QP and 3-BEP problems, and $3 \leq$ `nGenes` ≤ 9 for the 6-BM 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 $5 \leq$ `nTrs` ≤ 9. Lastly, a proper choice for `StNonImp` is a large value, e.g., $5 \leq$ `StNonImp` ≤ 9.

Table 3.4: Performance of the TP algorithm with different values of each TP parameter.

	SR-QP			6-BM			3-BEP		
Param.	Val.	AV	R%	Val.	AV	R	Val.	AV	R%
	1	1,092	90	1	8,062	96	1	7,311	96
	3	870	97	3	7,739	96	3	7,958	92
hLen	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
	1	1,098	84	1	16,268	63	1	16,074	57
	2	981	95	3	9,432	91	2	9,383	86
nGenes	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
	2	1,597	61	2	21,451	43	2	18,905	48
	3	1,274	82	3	12,295	89	3	9,123	94
nTrs	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
	1	1,143	85	1	10,569	90	1	9,507	92
	3	906	93	3	8,419	96	3	6,825	98
StNonImp	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
	1	1,242	80	1	14,592	76	1	12,972	80
	3	1,146	89	3	9,635	96	3	7,640	94
MnNonImp	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
	1	1,073	85	1	7,488	98	1	7,211	96
	3	1,279	80	3	8,157	97	3	6,704	97
nTL	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

In fact, we may expect that increasing the size of a tree horizontally reduces the bad effect of dormant nodes, where a dormant node is a node that does not contribute to the fitness value of a program [58]. Therefore, a high value of `nGenes` parameter is preferable to a small one, since the high value of `nGenes` increases the size of a tree horizontally. On the other hand, the `nTrs` and `MnNonImp` parameters control the number of trial programs that will be generated around the current program. Therefore, it is reasonable to expect that increasing the values of those parameters improves the performance of the TP algorithm.

3.3.4 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 [105]. 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 3.5.

TP Algorithm vs GPLab Toolbox

GPLab [105] 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 [116].

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 3.5. For the GPLab toolbox, we set its parameter values as the standard values [105], except for the

Table 3.5: Parameter values for the TP algorithm, the GPLab toolbox and the standard GP algorithm.

Algorithm	Parameter	SR-QP	SR-Poly-4	6-BM	3-BEP
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
	`LnkFun`	+	+	IF	AND
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				

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 3.6, 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 3.6, 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 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 3-BEP problem.

Table 3.6: Comparison among the TP algorithm and the GPLab toolbox for the SR-QP, 6-BM and 3-BEP problems.

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

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 3.5.

Poli and Langdon [96] 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; the SR-QP problem and the SR-Poly-4 problem with the same settings in Subsection 3.3.2. In addition, they 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 `nPop` $= 100$ and `nPop` $= 1,000$ for the SR-QP and SR-Poly-4 problems, respectively. In the other experiments, they performed 1000 independent runs using `nPop` $= 1,000$ and `nPop` $= 10,000$ for the SR-QP and SR-Poly-4 problems, respectively. For all experiments they used `nGnrs` $= 30$. The results shown in Fig. 3.4 for the BC-GP and GP algorithms are taken from Figs. 8-11 in the original reference [96].

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 [96]. The parameter values for the TP algorithm are shown in Table 3.5, and the results of the TP, BC-GP and GP algorithms are shown in Fig. 3.4. 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.

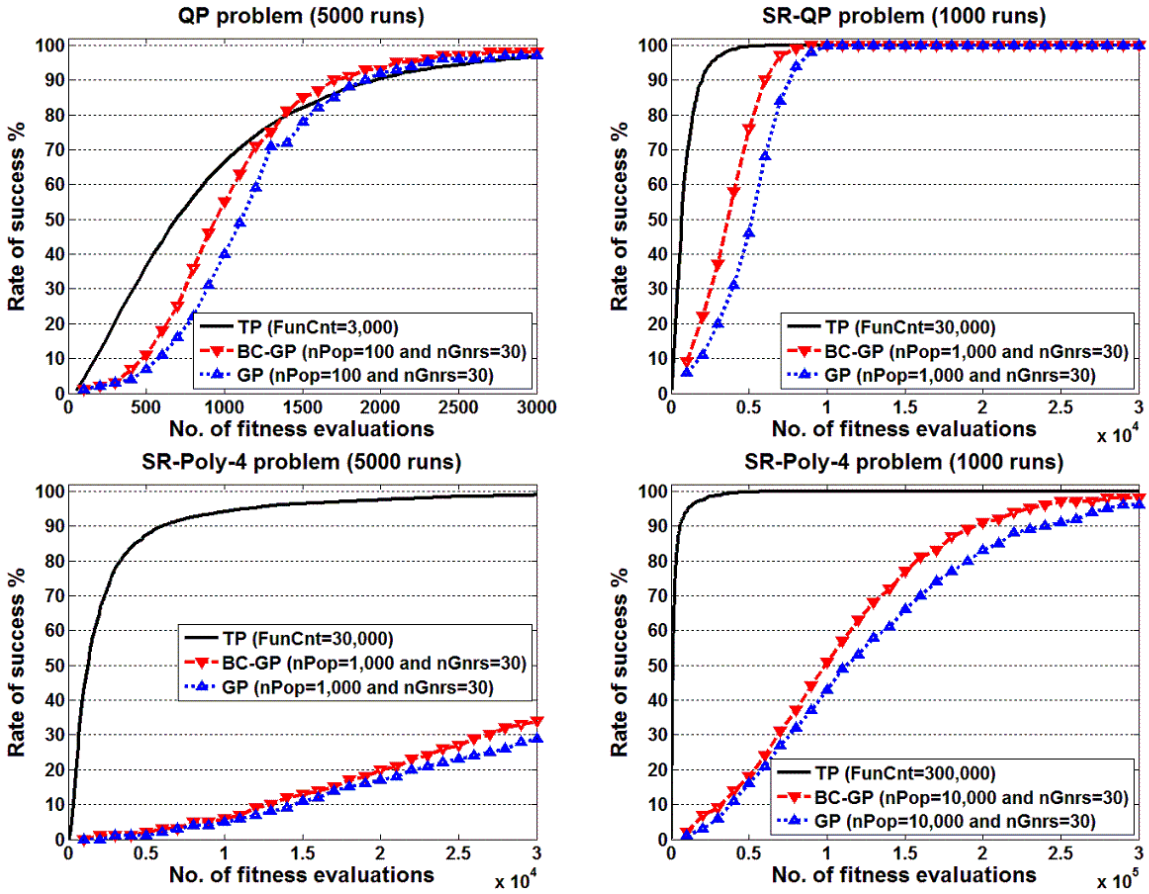

Figure 3.4: Comparison among the TP, BC-GP and GP algorithms for the SR-QP and SR-Poly-4 problems.

Walker and Miller [114] 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 3-BEP problem.

Walker and Miller [114] 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 3-BEP problem. In addition, they used the module technique to automatically build and evolve modules

Table 3.7: Comparison among the TP, CGP and ECGP algorithms for the 3-BEP 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

in the ECGP algorithm. They performed 50 independent runs for the 3-BEP 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 3-BEP problem using the TP algorithm, where the parameter values for the TP algorithm are shown in Table 3.5. 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 3.7. 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 [114].

Koza [68] published a book on GP, which contains numerical results for a large number of problems. In addition, a book chapter by Azad and Ryan [5] also contains a lot of numerical results. These two references also contain results for some of the problems used in this chapter, and our results seem to compare favorably to those reported there.

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 2.3, 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.

Table 3.8: Comparison among the GP, GP-LS, TP and TP-Mut algorithms for the SR-QP and 3-BEP problems.

	SR-QP			3-BEP		
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

2. GP-LS: The standard GP using the local search procedures in Section 2.3 instead of the mutation operator.

3. TP: The proposed TP algorithm as described in Algorithm 3.1.

4. TP-Mut: The proposed TP algorithm using the mutation operator instead of the local search procedures in Section 2.3.

To make comparisons, we performed 100 independent runs of each algorithm for the SR-QP and 3-BEP problems. The parameter values are set as shown in Table 3.5. 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 [68, 69]. 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 3.8, where the maximum number of fitness evaluations for the GP, GP-LS, TP and TP-Mut algorithms is `nPop*nGnrs`.

From Table 3.8, 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 3-BEP 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

3-BEP 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.

3.4 Conclusions

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.

Chapter 4

Memetic Programming

4.1 Introduction

The aim of this chapter is to introduce a hybrid evolutionary algorithm, called Memetic Programming (MP) algorithm, as an improvement of the GP algorithm. The term "memetic" comes from memetic algorithms since the MP algorithm inherits the basic idea from memetic algorithms [43, 72, 73, 88, 89, 90], while the term "programming" comes from GP since MP deals with computer programs represented by trees. Specifically, the proposed algorithm hybridizes GP with new local search procedures over a tree space to intensify promising programs generated by the GP algorithm. These local searches are used to generate trial programs in the neighborhood of the current one by changing it in small scales. In addition, the proposed algorithm can easily incorporate the *Automatically Defined Function* (ADF) technique to exploit the modularities in problem environments [69]. We will show through numerical experiments that the proposed MP algorithm is more efficient in finding an optimal solution than the GP algorithm especially when using the ADF technique.

The rest of the chapter is organized as follows. In the next section, we introduce the proposed MP algorithm. In Section 4.3, more explanations about the practical implementation of the MP algorithm are given. In Section 4.4, we report numerical results for some benchmark problems. Finally, conclusions make up Section 4.5.

4.2 Memetic Programming

In this section a new hybrid evolutionary algorithm is presented as an improvement to the GP algorithm. The proposed algorithm hybridizes GP with the local search procedures introduced in Section 2.3 to improve individuals generated by using GP operators.

In the next subsection, a new local search algorithm over a tree space called Local Search Programming algorithm is discussed. In Subsection 4.2.2, the proposed Memetic Programming (MP) algorithm is described. Finally, the MP algorithm will be extended to deal with the ADF technique in Subsection 4.2.3.

4.2.1 Local Search Programming

A local search algorithm starts with an initial solution, and subsequently applies some operators to generate new solutions in a neighborhood of the current one. This process iterates until no better solution can be found in the neighborhood, and then the algorithm is terminated. In this subsection, a new local search algorithm over a tree space, called Local Search Programming (LSP) algorithm, is proposed to find the best program in the neighborhood of the current program X. The proposed LSP algorithm mainly uses the local search procedures described in Section 2.3. The details of the proposed algorithm are shown below.

Algorithm 4.1. *Local Search Programming*

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

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

   2.1 ***Static Structure Search***

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

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

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

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

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

   2.4 ***Dynamic Structure Search***

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

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

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

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

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

In the initialization step, Step 1, the algorithm starts with a program X that is generated randomly or received from another algorithm. In addition, X_{best} and a counter k take their initial values. In fact, the counter k is used to count the number of non-improvements during the search process. In Step 1, the user must choose two positive integers `nFails` and `nTrs`. Specifically, `nFails` is the maximum number of non-improvements and `nTrs` represents the number of trial programs generated in the neighborhood of the current program using static and dynamic structure searches. For the shaking procedure, we put `nTrs` $:= \min(\texttt{nTrs}, |X|)$, and for the pruning procedure, we put `nTrs` $:= \min(\texttt{nTrs}, d(X))$.

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

In Step 2.3, when the algorithm reaches the maximum number of non-improvements `nFails`, it will stop and return with X_{best}. Otherwise, it proceeds to Step 2.4 to diversify the search process using a new program with different structure by applying either the grafting or the pruning procedure, which is chosen randomly. In Step 2.4.2, a set $\mathbb{Z}$ of trial programs is generated by the selected procedure. In Step 2.5, the algorithm replaces X by the best program in $\mathbb{Z}$ and goes back to Step 2.1. Finally, when the termination condition is satisfied, the algorithm stops at Step 3 and returns with the best program found. Fig. 4.1 shows the flowchart of the proposed LSP algorithm.

In Algorithm 4.1, the main loop starts in Step 2.1 by generating `nTrs` programs using the shaking procedure. If there is no improvement occurred in the current program, then the counter k is increased by 1, and the algorithm keeps working and generating new programs according to the following two processes. First, the algorithm proceeds to generate two sets of `nTrs` programs; the first `nTrs` programs are generated using the grafting or pruning procedures in Step 2.4, and the remaining `nTrs` programs are generated using the shaking procedure in Step 2.1. Second, the algorithm increases the counter k by one if no improvement occurred in the current program. These two processes are repeated as long as the value of the counter k does not exceed the maximum value `nFails`. Therefore, the number of fitness evaluations needed during a single run of the LSP algorithm varies depending on the improvement of the current program. If the algorithm completely fails to improve the current program, then the number of fitness

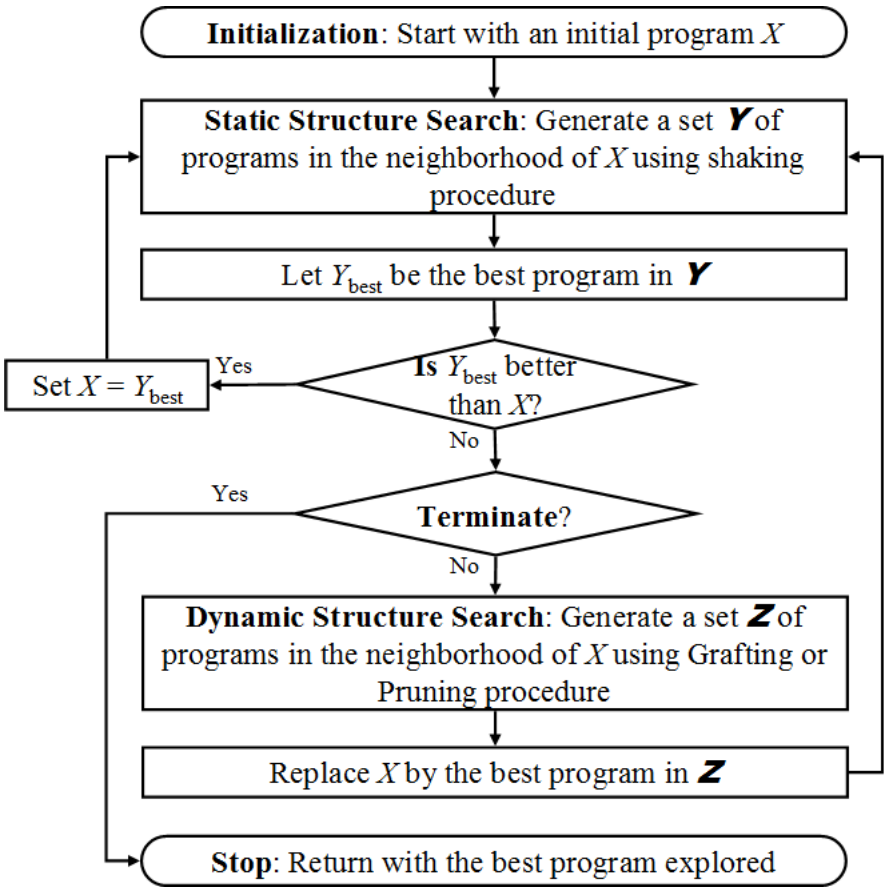

Figure 4.1: The flowchart of LSP

evaluations consumed during the run of the algorithm is

$$minFit_{LSP} = (2\,\texttt{nFails} + 1)\,\texttt{nTrs}. \tag{4.1}$$

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

4.2.2 Proposed Algorithm

The main target of the MP algorithm is to improve the behavior of the GP algorithm by reducing the disruption effect of the crossover and mutation operators. Performing a local search for some promising programs during the search process can improve these programs. Moreover, if the search process succeeds to reach the area near an optimal

solution, then a simple local search algorithm can capture that optimal solution easily. On the contrary, if the GP algorithm continues to be applied without the help of local search, there is a high probability of losing such promising solutions due to the disruption effect of crossover and mutation operators. In the MP algorithm, we use the LSP algorithm described in the previous subsection to improve some programs chosen from the current population based on their fitness values. Figure 4.2 shows the main structure of the proposed MP algorithm, and its formal is stated as follows:

Algorithm 4.2. *(MP Algorithm)*

1. ***Initialization.***

   1.1. *Generate a random population of programs and evaluate the fitness value for each program. Set the initial values of controlling parameters needed in the search process.*

   1.2. *Select some promising programs according to their fitness values.*

   1.3. *Apply the LSP algorithm, Algorithm 4.1, to the selected programs.*

   1.4. *Update the controlling parameters.*

2. ***Main Loop.*** *Repeat the following Steps 2.1-2.6 until a termination condition is satisfied. If a termination condition is satisfied, proceed to Step 3.*

   2.1. *Select a set of parents from the current population, according to their fitness values.*

   2.2. *Generate a new population using crossover and mutation operators, and evaluate the fitness value for each program in the new population.*

   2.3. *Select a set of promising programs according to their fitness values.*

   2.4. *Apply the LSP algorithm, Algorithm 4.1, to the selected programs.*

   2.5. *Update the controlling parameters.*

   2.6. *Return to Step 2.1 to breed a new population.*

3. *Stop with the best program found.*

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

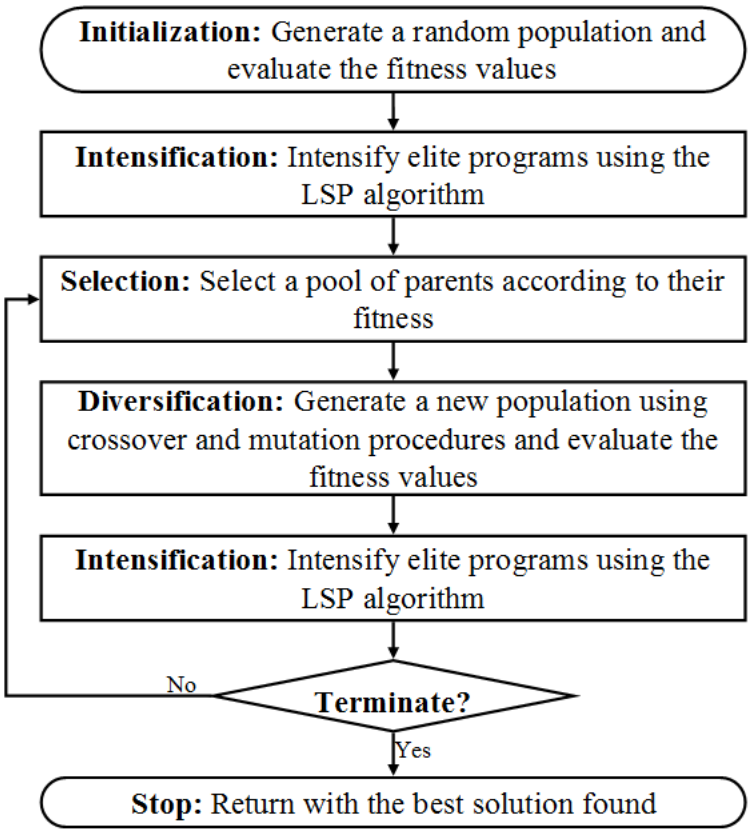

Figure 4.2: The MP flowchart.

number of fitness evaluations, or reaching the maximum number of consecutive non-improvements for the best program during the course of generating populations.

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

It is worthwhile to note that the main loop in Step 2 of Algorithm 4.2 can be divided into two phases, the diversification phase in Steps 2.1-2.2 and the intensification phase in Steps 2.3-2.4. The diversification phase follows the GP strategy, where choosing a suitable selection strategy and using the crossover and mutation operations guarantee the diversity in the current population. On the other hand, the intensification phase, which uses the LSP algorithm to intensify promising programs obtained in the diversification

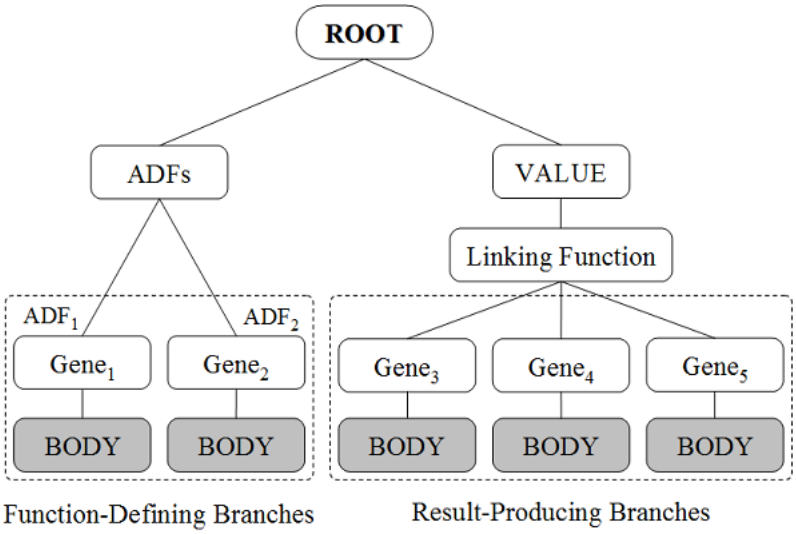

Figure 4.3: Example of representing a program in MP using ADF technique

phase, tries to catch the best solution. We note that, the MP algorithm at least behaves like the GP algorithm in case that no improvement occurs in the intensification phase. However, in this case the MP algorithm will be more costly than the GP algorithm, because of the computational effort spent in the intensification phase.

4.2.3 Memetic Programming with ADF Technique

In this subsection, we focus on the effect of using the ADF technique within the proposed MP algorithm. The proposed MP algorithm uses the multigenic representation to express programs in the population, where each program is represented as a tree consisting of several genes, see Section 2.2. Actually, the multigenic strategy enables MP to deal easily with ADFs for reusing codes. Since each program in MP can contain more than one gene, we can adopt one or more of these genes to work as ADF(s). In other words, each program in MP with ADFs contains two types of genes, ADF genes which represent function-defining branches, and regular genes which represent the result-producing branches. The result-producing branches are linked together by a suitable function to produce the final form of the program. Fig. 4.3 shows an example of the structure of a MP program using the ADF technique. In the next section, we will show more details and explanations about implementing the ADF technique with MP.

4.3 Implementation of MP

In this subsection, we illustrate the details of some basic topics that are essential in the implementation of the MP algorithm, Algorithm 4.2, concerning representation of individuals, breeding operators, selection techniques, and so on.

4.3.1 Individuals Representation and Breeding Operators

The MP algorithm uses the tree data structure to represent a solution like the GP algorithm. However, the MP algorithm represents a solution according to the representation strategy described in Section 2.2. Therefore, each solution in the MP algorithm is called a program, and it consisting of one or more genes. These genes are linked together using a linking function to produce the final form of a program. For the initial population, we generate its genes randomly using the strategy described in Section 2.2.

As we described in the previous section, the main loop of Algorithm 4.2 can be divided into two phases, the diversification phase and the intensification phase. In the diversification phase, the algorithm uses the classical crossover and mutation operators, Procedures 1.1 and 1.2, respectively, as breeding operators. When applying the mutation operator, the algorithm needs to generate a new subtree to exchange it with an alternative one chosen randomly from the current program. This subtree is generated using the same strategy of generating a new gene in the initial population, see [67, 68, 69]. On the other hand, the algorithm uses the shaking grafting and pruning procedures, Procedures 2.1, 2.2 and 2.3, respectively, as breeding operators in the intensification phase. Throughout this chapter, we use $\zeta = 1$, where ζ is the depth of branches added to and removed from a tree X in the grafting search (Procedure 2.2) and the pruning search (Procedure 2.3), respectively.

4.3.2 Selection Techniques

The proposed MP algorithm, Algorithm 4.2, mainly contains two selection steps: First, Step 2.1 selects a pool of programs to breed a new population using the crossover and mutation operators. Second, Step 2.3 (as well as Step 1.2) selects a set of promising programs to improve them by the LSP algorithm. In the present chapter, we use three different selection techniques that have shown promise in our extensive numerical experiments. In Step 2.1, we use the tournament selection [122] of size 4 as the default selection strategy, and the roulette wheel selection will be used for some special experiments, as shown later. In Steps 1.2 and 2.3, we always select the best `nLs` programs from the

current population, where `nLs` is a positive integer less than or equal to the population size.

For the tournament selection of size 4, the MP algorithm chooses 4 programs randomly from the current population, and the one with the best fitness is selected as the winner. However in the roulette wheel selection, the algorithm specifies a probability value for each program in the population based on its fitness value compared to the fitness values of other programs in the current population. Specifically, the probability value p_i of the i-th program in the current population is computed as:

$$p_i = \begin{cases} \frac{f_i}{\sum_{j=1}^{\texttt{nPop}} f_j}, & \text{for positive fitness values,} \\ \\ \frac{(af)_i}{\sum_{j=1}^{\texttt{nPop}} (af)_j}, & \text{for non-positive fitness values,} \end{cases} \tag{4.2}$$

where f_i and $(af)_i = \frac{1}{1-f_i}$ are the fitness value (the raw fitness value) and the adjusted fitness value, respectively, of the i-th program in the current population of size `nPop`. See [68] for more details about the raw fitness and the adjusted fitness.

4.3.3 Set of Parameters in MP

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

- Representation Parameters
  - `hLen`: The head length for every gene generated randomly in the initial program.
  - `MaxLen`: The maximum length, i.e., the number of nodes, of a gene allowed in the search process.
  - `nGenes`: The number of genes in each program.
  - `LnkFun`: The function used to link genes in each program.
- Search Parameters
  - `nPop`: The population size.
  - `nGnrs`: The maximum number of generations.
  - `nLs`: The number of programs selected to apply local search procedures at the intensification phase in Steps 2.3-2.4 in Algorithm 4.2.

- `nTrs`: The number of trial programs generated in the neighborhood of the selected program using a local search procedure, i.e., shaking, grafting or pruning.
- `nFails`: The maximum number of non-improvements for each call of the LSP algorithm in Step 2.4 of the MP algorithm.

If no improvements occurs in the intensification phase in Steps 2.3-2.4 of Algorithm 4.2, then the number of fitness evaluations consumed in this phase is $minFit_{IntP} = \texttt{nLs}\,(2\,\texttt{nFails}+1)\,\texttt{nTrs}$. This is because Algorithm 4.2 calls the LSP algorithm `nLs` times during the intensification phase in Steps 2.3-2.4, while the number of fitness evaluations during one call of the LSP algorithm is $(2\,\texttt{nFails}+1)\,\texttt{nTrs}$, equation (4.1). If the values of `nLs`, `nFails` and `nTrs` are large, the MP algorithm will need a lot of fitness evaluations in the intensification phase in Steps 2.3-2.4.

Certainly, if the LSP algorithm succeeds to improve the chosen programs in Steps 2.3 of Algorithm 4.2, then we do not mind the increase of the number of fitness evaluations in the intensification phase, since it increases the probability of finding an optimal solution. On the other hand, if the LSP algorithm fails to improve the chosen programs in Steps 2.3 of Algorithm 4.2, then the MP algorithm will not gain benefits by using a large number of fitness evaluations. For that reason, we let $minFit_{IntP}$ be the maximum number of fitness evaluations in the intensification phase. Specifically, we set $minFit_{IntP} = \alpha\,\texttt{nPop}$, where α is a positive constant determined before calling the algorithm. Then the values of `nLs`, `nTrs` and `nFails` must be chosen to satisfy the equation

$$\texttt{nLs}\,(2\,\texttt{nFails}+1)\,\texttt{nTrs} = \alpha\,\texttt{nPop}. \tag{4.3}$$

In practice, we first choose the values of `nTrs` and `nFails`, and then determine the value of `nLs` (from (4.3)) by

$$\texttt{nLs} = \left\lceil \frac{\alpha\,\texttt{nPop}}{(2\,\texttt{nFails}+1)\,\texttt{nTrs}} \right\rceil, \tag{4.4}$$

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

4.3.4 Building and Evolving ADFs in MP

Actually, based on the individual representation in MP, we can easily extend the MP algorithm to build and reuse ADFs. For each program in MP, we can exploit one or more genes to work as ADF(s), which will be created and evolved during the run of

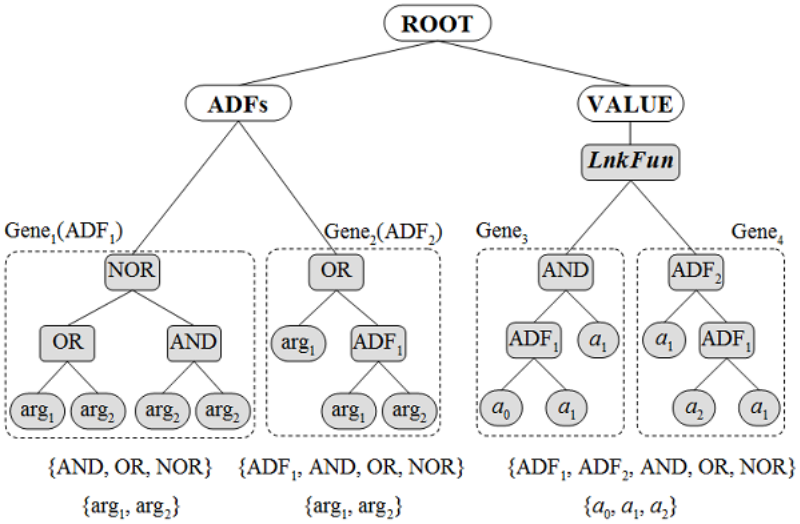

Figure 4.4: Example of representing a program in MP using ADFs technique. Last two lines represent the function and terminal sets for each gene.

the algorithm. In addition, these ADFs will be added automatically to the function set of other genes, the result-producing branches, in the same program that contains these ADFs. In this case, the fitness value for a program in MP with ADFs will be computed from the result-producing branches by linking all of them using a suitable function, e.g., a primitive function or one of the ADFs themselves.

Suppose that $\{\text{ADF}_1, \text{ADF}_2, \cdots, \text{ADF}_n\}$ is the set of ADFs used for the problem at hand, where each ADF has its own set of dummy arguments. As in GP with ADFs, the function set used to build the ADFs is the original function set $\mathfrak{F}$, and it is possible to include one or more function(s) from the set $\{\text{ADF}_i | i = 1, \cdots, j-1\}$ in the body of ADF_j, where $j = 2, \cdots, n$. In addition, the function set for result-producing branches (regular genes) is $\mathfrak{F} \cup \{\text{ADF}_1, \cdots, \text{ADF}_n\}$, while the terminal set is the original terminal set $\mathfrak{T}$. Fig. 4.4 shows an example of the overall structure of a program in the MP algorithm using two ADFs and two result-producing branches.

In fact, ADFs can increase the size of genes dramatically, especially if there exist several ADFs in the body of genes. In other words, if one replaces each ADF in the body of a regular gene by its equivalent subtree, then the number of nodes of this regular gene can increase rapidly according to the number of ADFs in its main tree. Fig. 4.5 shows the actual trees of genes 3 and 4 in Fig. 4.4 by replacing ADF_1 and ADF_2 by their equivalent subtrees. In Fig 4.4, one can notice that the number of nodes of gene 3 is 5, and the number of nodes of gene 4 is 5. However, by replacing each ADF function by its equivalent subtree as in Fig. 4.5, we can see that the actual number of nodes for

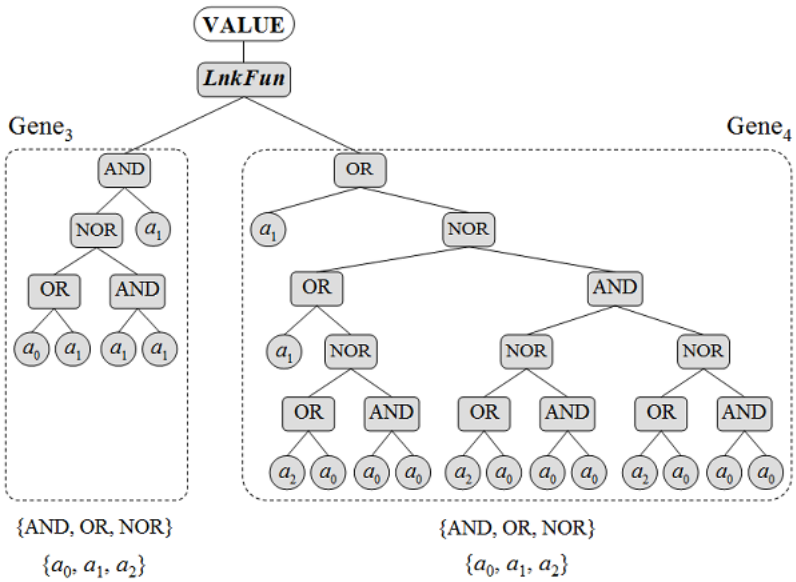

Figure 4.5: The actual tree representations of genes 3 and 4 in Fig. 4.4

genes 3 and 4 are 9 and 27, respectively. In addition, in case of using a function from the set {AND, OR, NOR} to link genes 3 and 4 in Fig. 4.5, the total number of nodes in the program will be 37. On the other hand, if the `LnkFun` in Fig. 4.4 is ADF_1 or ADF_2, then the total number of nodes in the program in Fig. 4.5 will be 93 or 103, respectively. Therefore, the values of `hLen` and `MaxLen` parameters must be chosen small enough in case of using the ADF technique. In this chapter we set `hLen` $:= (h_1, h_2)$ and `MaxLen` $:= (m_1, m_2)$, which means `hLen` $= h_1$ and `MaxLen` $= m_1$ for ADFs genes, and `hLen` $= h_2$ and `MaxLen` $= m_2$ for regular genes.

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

4.4 Numerical Experiments

This section discusses the performance of the MP algorithm on some well-known benchmark problems. First, we introduce the benchmark problems under consideration in

Subsection 4.4.1. Then, we examine the performance of the MP algorithm through extensive experiments. In these experiments, we focus on the performance of the proposed MP algorithm under different environments. Specifically, we study the performance of MP under different selection strategies and different values for the `nLs`, `nTrs` and `nFails` parameters. These parameter settings are performed for the MP algorithm with and without the use of the ADF technique. Finally, we make a set of comparisons, between the MP algorithm and various recent versions of the GP algorithm, to show the efficiency of the proposed MP algorithm. In fact, the proposed MP algorithm shows promising performance compared to recent GP algorithms as we will see later. In all the experiments, we terminate the algorithm as soon as an optimal solution with the highest fitness value is found, or the maximum number of fitness evaluations, i.e., `nPop*nGnrs`, is reached.

Through the subsequent numerical experiments, we will perform a lot of comparisons in terms of the computational effort CE defined in Equation (1.1) using $z = 0.99$ as in Koza [68]. However, in the MP algorithm, the number of fitness evaluations per each generation varies according to the performance of the LSP algorithm. Therefore, we will slightly modify the definition of CE for the MP algorithm. Specifically, we define the computational effort CE for the MP algorithm by (1.1), where $N_s(i)$ is the number of successful independent runs using up to $i*$`nPop` fitness evaluations in the MP algorithm, which is equal to the number of successful independent runs up to generation i in the GP algorithm with fixed population size `nPop`.

4.4.1 Test Problems

In the rest of this section, we test the performance of the TP algorithm through the SR-POLY-4, N-BEP and 6-BM problems, see Appendix A for more details. In addition, we use the following settings, unless otherwise stated, for our test problems:

- For the SR-POLY-4 problem, the set of independent variables $\{x_1, x_2, x_3, x_4\}$ is regarded as the terminal set, and the set of functions $\{+, -, *, \%\}$ is regarded as the function set, where $x\%y := x$ if $y = 0$; $x\%y := x/y$ otherwise.

- For the N-BEP problem, the set of arguments $\{a_0, a_1, \cdots, a_{N-1}\}$ is used as the terminal set, and the set of Boolean functions {AND, OR, NAND, NOR} is used as the function set. In fact, the GP research community considers evolving the N-BEP function by using those Boolean functions as a good benchmark problem for testing the efficiency of new GP techniques [68, 114].

- For the 6-BM problem, the set of arguments $\{a_0, a_1, d_0, d_1, d_2, d_3\}$ is used as the terminal set, and the set of Boolean functions {AND, OR, NOT, IF} is used as the function set, where IF(x, y, z) returns y if x is true, and it returns z otherwise.

For the SR-POLY-4 problem, the algorithm starts a run by generating a dataset randomly under the conditions described in Subsections A.1.2. Moreover, the generated dataset will be fixed during the running time of the algorithm for each independent run.

4.4.2 Performance Analysis

In this subsection, we study the setting of MP parameters and components, and its effect on the performance of the MP algorithm. Specifically, we mainly focus on the selection strategy and the parameter setting associated with the LSP algorithm, i.e., `nLs`, `nTrs` and `nFails`. A set of different values for each of these parameters is chosen, and for each value, 100 independent runs are performed to compute the average number of evaluations (AV), the computational effort (CE) and the rate of success (R). In this subsection, the values of the parameters under consideration will be specified for each set of experiments. The values of other parameters are set to their standard values shown in Table 4.1, which are determined from our pilot experiments or the common setting in the literature.

Performance under Different Selection Strategies

In this set of experiments, we use the MP algorithm with two different selection strategies; the tournament selection and the roulette wheel selection. Since the fitness values of the SR-POLY-4 are always non-positive, the algorithm computes the corresponding probability values for the roulette wheel selection using the adjusted fitness values. However, the probability values corresponding to the fitness values of the 3-BEP and 6-BM problems are computed using the row fitness values.

For each selection strategy, we performed 100 independent runs to compute the average number of evaluations (AV), the computational effort (CE) and the rate of success (R). The performance of the MP algorithm with these two selection strategies is shown in Table 4.2. From these results, we can see that the tournament selection gives better results than the roulette wheel selection. When the LSP algorithm is not employed, i.e., $\alpha = 0$, changing the selection strategy from the tournament to the roulette wheel caused a collapse in the rate of success, and increased the computational efforts unreasonably, in particular, for the 6-BM problem. On the other hand, by using the LSP algorithm, the effect of changing the selection strategy is not high compared to the previous case. Therefore, one can conclude that the LSP algorithm increases the stability of the MP

Table 4.1: The standard values of the MP parameters for the benchmark problems.

	MP without ADFs				MP with ADFs	
Parameter	SR-POLY-4	3-BEP	N-BEP$_{16BF}$	6-BM	SR-POLY-4	N-BEP
`hLen`	3	3	1	3	(3,1)	(3,1)
`MaxLen`	30	30	15	30	(7,3	(7,3)
`nGenes`	3	2	N	3	4	$N+1$
`nPop`	50	500	50	500	50	$250(N-1)$
`nGnrs`	100	100	$\rightarrow \infty$	100	100	$\rightarrow \infty$
`LnkFun`	+	AND	XOR	IF	+	ADF_1

Equation (4.4): $\alpha := 1/2$, `nTrs` := 2 and `nFails` := 1
Crossover probability:= 0.9
Mutation probability:= 0.05
Reproduction (Copy) probability:= 0.05
Selection strategy: Tournament selection of size 4

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

Parameter `nLs`

Here, we focus on the parameter `nLs`, the number of programs in the population to which local search is applied, and its effect on the performance of the MP algorithm. Different values for the constant α in (4.4) are chosen, and 100 independent runs of the MP algorithm are performed for each value. Comparisons, in terms of the average number of evaluations, the computational efforts and the rate of success, are made for these different settings of the MP algorithm. Mainly, we focus here on four cases:

1. $\alpha = 0$, which means there is no use for the LSP algorithm.
2. $\alpha = 1/2$, which means the intensification phase will cost at least `nPop`$/2$ fitness evaluations for each generation of Algorithm 4.2.
3. $\alpha = 1$, which means the intensification phase will cost at least `nPop` fitness evaluations for each generation of Algorithm 4.2.

Table 4.2: Comparison of selection strategies.

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

4. Applying the LSP algorithm for all programs in the current population, i.e., `nLs` = `nPop` which implies $\alpha = 6$ by (4.3), since `nFail` = 1 and `nTrs` = 2 in Table 4.1.

From Table 4.3, we see that the LSP algorithm has great influence on the MP algorithm. It improves performance of the MP algorithm in both the number of evaluations and the rate of success. In addition, applying the LSP algorithm for all programs in the current population is costly in terms of AV and CE, but it improved the rate of success. Throughout this section, we use $\alpha = \frac{1}{2}$ as recommended from the results in Table 4.3.

Parameters `nTrs` and `nFails`

We conducted experiments in which SR-POLY-4, 3-BEP and 6-BM problems were solved using the MP algorithm without ADFs. The main parameters we focus here are the parameters `nTrs` and `nFails`. The chosen values for these parameters are `nTrs` $= 1, 2, 3, 4, 5$ and `nFails` $= 1, 2, 3$. For each combination of these parameter values, we performed 100 independent runs of the MP algorithm for each problem. The results of these experiments are shown in Table 4.4. It is worthwhile to note that for most values of `nTrs` and `nFails`, the use of local search helps to improve the performance of the MP algorithm;

Table 4.3: Comparisons in terms of parameter `nLs`

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

see the row labeled '`Without LSP`' in Table 4.4.

It is clear from the results in Table 4.4 that the `nTrs` and `nFails` parameters affect the success rate only slightly. This is because the total number of programs treated by the LSP algorithm is almost similar for all combinations of the `nTrs` and `nFails` values, Equation (4.4).

Parameter Setting for MP with ADFs

The previous experiments, which mainly focus on the effects of the parameters `nTrs` and `nFails`, indicate that our results are promising compared to recent algorithm as we will see later. Nevertheless, we still have a chance to improve further these results especially for the N-BEP problem by the use of the ADF technique. The GP research community usually uses the ADF technique to exploit the modularity in a problem, especially the N-BEP problem [69, 114]. Using GP with the ADF technique, Koza [69] has succeeded to get the exact solutions for the N-BEP problem with $N = 3, \cdots, 11$, with less CE compared to the standard GP algorithm without the ADF technique.

What we want to show here is that the MP algorithm not only can improve the results of the standard GP algorithm by using the new set of operations introduced in Section

Table 4.4: Results of the MP algorithm in terms of parameters `nTrs` and `nFails`

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

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

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

For the N-BEP problem, we use two additional ADFs, ADF_1 and ADF_2, each of which has two dummy arguments, arg_0 and arg_1. For each program in the population, ADF_1 and ADF_2 are defined, evolved and included automatically in the function set for that program. The terminal sets for ADF_1, ADF_2 and regular genes are $\{\text{arg}_0, \text{arg}_1\}$, $\{\text{arg}_0, \text{arg}_1\}$ and $\{a_0, \cdots, a_{N-1}\}$, respectively. The function sets for them are {AND,

OR, NAND, NOR} {ADF_1, AND, OR, NAND, NOR} and {ADF_1, ADF_2, AND, OR, NAND, NOR}, respectively.

In the current set of experiments, we still focus on detecting the best values of `nTrs` and `nFails` parameters. Several values are chosen such as `nTrs` $= 1, 2, 3, 4, 5$ and `nFails` $= 1, 2, 3$, and for each combination of these values, we performed 100 independent runs for each problem using the MP algorithm with ADFs. Other MP parameters are shown in Table 4.1, while the results of this experiment is shown in Table 4.5. It is clear from these results that the MP algorithm significantly reduces the computational effort and improves the rate of success by means of ADFs.

Table 4.5: Results of MP with the ADF technique in terms of parameters `nTrs` and `nFails`.

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

Moreover, we performed a set of experiments to solve the 6-BM problem using the MP algorithm with the ADF technique. Unfortunately, we could not improve the performance of the MP algorithm by using one ADF or two ADFs. This show that the ADF technique is useful for problems which have special characteristics, e.g., similar-

ity and modularity. Nevertheless, our results for the 6-BM problem are still good and competitive, without the use of the ADF technique, as we will see in the next subsection.

It is worthwhile to mention that the Boolean even parity functions are compactly represented using XOR and XNOR Boolean functions [114]. Therefore, by evolving ADF_1 to produce the XOR or XNOR Boolean function, an optimal solution can be found easily using the MP algorithm, especially by using the ADF_1 to link genes of a program. For example, the ADF_1 and the ADF_2 of the 100 final output programs in Table 4.5 for `nTrs` = 2 and `nFails` = 1 produce the XOR Boolean function 47 times and 10 times, respectively, and produce the XNOR Boolean function 53 times and 6 times, respectively. This example can give us a clear vision about the effect of the ADF technique and its exploitations for the similarity and modularity of a given problem.

4.4.3 MP vs GP

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

The SR-POLY-4 Problem

Poli and Langdon [96] conducted a lot of numerical experiments for the SR-POLY-4 problem using the backward-chaining GP (BC-GP) algorithm, and made a comparison between the BC-GP algorithm and the standard GP algorithm. They performed 5000 independent runs to solve the SR-POLY-4 problem using the BC-GP and standard GP algorithms without the use of ADFs, where `nPop` = 1,000 and `nGnrs` = 30. The results of their experiments show a good performance of the BC-GP algorithm compared to the standard GP algorithm.

Here, we compare the results of the MP algorithm with those of the BC-GP algorithm in terms of fitness evaluations and the rate of success. We implement the MP algorithm (without ADFs) for the SR-POLY-4 problem three times using different population sizes. For each implementation, we performed 5000 independent runs for the MP algorithm using `nTrs` = 1 and `nFails` = 1 (as recommended from the results in Table 4.4), while the other parameters are set as shown in Table 4.1. The results of this experiment are summarized in Fig. 4.6, where the results of the BC-GP algorithm have been taken from Fig. 10 in the original reference [96].

Fig. 4.6 shows that the MP algorithm significantly outperforms the BC-GP algorithm. The MP algorithm gets the 100% success for all 5000 runs using 10,000 fitness evaluations at most, even by using a very small population size `nPop` = 50. On the other hand,

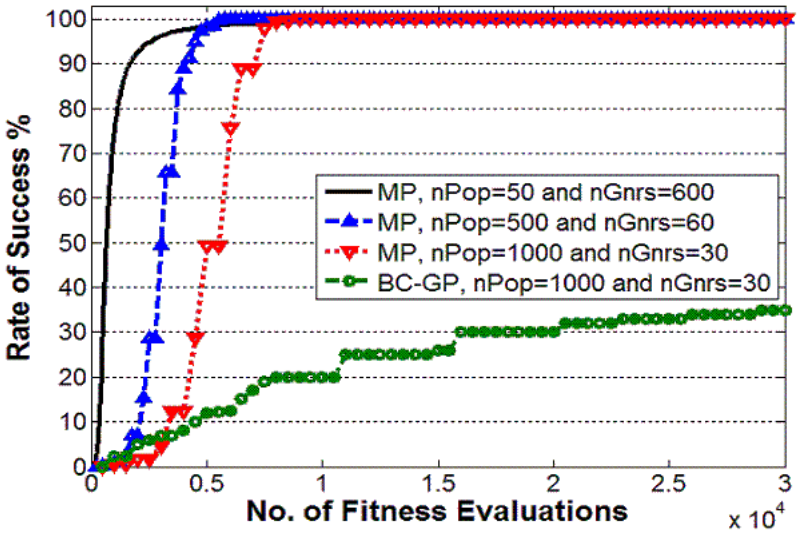

Figure 4.6: Comparison between the MP algorithm and the backward-chaining GP algorithm for the SR-POLY-4 problem in terms of the rate of success

the BC-GP hardly succeeds to find the exact solution for 35% of total runs by using 30,000 fitness evaluations. Poli and Langdon in the same paper [96] repeated the same experiment using a large population size, `nPop` $= 10,000$ and `nGnrs` $= 30$. Then, they were able to improve their results and the rate of success reached approximately 98% after 300,000 fitness evaluations. Nevertheless, one can see that the MP algorithm is faster than the BC-GP algorithm by more than 30 times for the SR-POLY-4 problem.

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

The N-BEP Problem

Walker and Miller [114] conducted a lot of numerical experiments to show the performance of the Cartesian GP (CGP) algorithm and the Embedded CGP (ECGP) algorithm. The ECGP algorithm generalizes the CGP algorithm by utilizing the automatic module acquisition technique to automatically build and evolve modules. Walker and Miller [114] reported good results for several test problems, compared to the standard GP algorithm and several contemporary algorithms. Here, we are interested in comparing their results for the Boolean even parity problems with the results of the MP algorithm

for the same problems.

Walker and Miller [114] used the Embedded Cartesian GP (ECGP) algorithm to solve the N-BEP problem using the set of Boolean functions {AND, OR, NAND, NOR} as the function set, and the set of arguments $\{a_0, \cdots, a_{N-1}\}$ as the terminal set. For each of the N-BEP problems with $N = 3, \cdots, 8$, they performed 50 independent runs with $\texttt{nGnrs} \rightarrow \infty$, i.e., for each run the algorithm works until the exact solution was found.

Walker and Miller [114] measure the performance of the ECGP algorithm in terms of CE and other statistical measures. Undoubtedly, the results of Walker and Miller [114] for the N-BEP problem are significant, as they showed through several comparisons with other extensions of the GP algorithms. Our target here is to make a comparison between the ECGP algorithm (with modules) and the MP algorithm with ADFs for the N-BEP problem with $N = 3, \cdots, 8$.

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

We performed 50 independent runs for the N-BEP problem with $N = 3, \cdots, 8$, using $\texttt{nTrs} = 1$ and $\texttt{nFails} = 2$, and the values of the remaining parameters are shown in Table 4.1. A comparison, in terms of CE, between the ECGP algorithm (with modules) and the MP algorithm with ADFs are presented in Table 4.6. In addition, Table 4.7 shows additional comparisons between ECGP and MP, in terms of median (ME) number of evaluations, median absolute deviation (MAD), and interquartile range (IQR). All the results for the standard GP algorithm and the ECGP algorithm shown in Tables 4.6 and 4.7 have been taken from [114]. It is clear from Tables 4.6 and 4.7 that the MP algorithm with ADFs outperforms the standard GP algorithm and the ECGP algorithm.

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

Table 4.6: The CE for the standard GP, ECGP and MP algorithms for the N-BEP problem

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

Table 4.7: The ME, MAD and IQR for the ECGP and MP algorithms for the N-BEP problem

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

The N-BEP Problem with 16 Boolean Functions

Poli and Page [97] introduced various extensions of the GP algorithm by using new search operators and a new node representation together with a tree evaluation method known as sub-machine-code GP. The sub-machine-code GP technique allows the parallel evaluation of 32 or 64 fitness cases per program execution, which gives their algorithms the ability to evaluate the fitness values faster than the usual way. Poli and Page [97] succeeded to solve the N-BEP problems, up to $N = 22$, with the function set consisting of all 16 Boolean functions of two arguments [97].

Table 4.8: The CE for the standard GP, GP-UX, GP-SUX and MP algorithms using the 16 boolean functions of 2 arguments for the N-BEP problems

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

Poli and Page [97] conducted experiments to show the performance of their algorithms on the N-BEP problem with $N = 3, \cdots, 6$, using a small population size, `nPop` $= 50$. They performed 50 independent runs for each problem to compute the CE of their algorithms using the set of all 16 Boolean functions of two arguments as the function set.

In our experiments, we compare results of the MP algorithm with those appeared in [97] for the N-BEP problem with $N = 3, \cdots, 6$. We performed 50 independent runs for each N-BEP problem with $N = 3, \cdots, 8$, using the same function set and the population size as those used by Poli and Page [97]. The parameter values for the MP algorithm are shown in Table 4.1, while `nTrs` $= 2$ and `nFails` $= 1$. The results are shown in Table 4.8, where the results of the standard GP, the GP-UX and the GP-SUX algorithms are taken from Poli and Page [97].

From the results in Table 4.8, it is clear that the MP algorithm outperforms other algorithms. As a matter of fact, the high performance of the MP algorithm and other algorithms comes from the modularity of the problem itself. The Boolean even parity functions are compactly represented using XOR and XNOR Boolean functions [114]. Therefore, by adjusting the function set to contain XOR and XNOR functions, all versions of the GP algorithm can find the exact solution for the N-BEP problem easily. In particular, since the MP algorithm expresses a solution with more than one genes and uses the Boolean function XOR to link these genes, it is possible to find the exact solution of the N-BEP problem easily and fast.

The 6-BM Problem

Poli and Page [97] also conducted a set of experiments on the 6-BM problem to study the performance of the standard GP, GP-UX and GP-SUX algorithms in terms of the CE. They used the set of all 256 Boolean functions of three arguments [97]. Indeed, our results for the 6-BM problem in Table 4.4 seem to outperform their results shown in Table 2 in their paper [97]. However, we do not consider that this is a fair comparison, since the function set used for the MP algorithm is not the same as the one used for their algorithms.

Jackson [58] introduced a new technique to detect dormant nodes in GP programs and to prevent the neutral crossover process. A dormant node is a node in a program that does not contribute to the fitness value of the program. When the crossover operator switches a subtree rooted at a dormant node in a program, the resulting child will have the same fitness value as the parent, and this process is called a fitness-preserving crossover (FPC) [58]. Jackson [58] states that preventing the FPC improves the performance of GP in at least three ways; improving the execution efficiency, increasing the rate of success, and simplifying evolved programs.

Jackson [58] carried out a set of experiments on the 6-BM problem using the standard GP algorithm with and without preventing FPCs. For each algorithm, 100 independent runs were made using `nPop` = 500 and `nGnrs` = 50. A comparison between these two versions of the GP algorithm was made in terms of the CE and the rate of success. In fact, as reported in [58], GP with preventing FPCs improved the CE and the rate of success for all test problems, except the 6-BM problem. For the 6-BM problem, the GP algorithm with preventing FPCs succeeded to reduce the CE, but it failed to improve the rate of success.

Table 4.9 shows a comparison between the MP algorithm and standard GP algorithm with and without preventing FPCs. We implemented the MP algorithm using different values for the parameters `nPop` and `nGnrs`. For each implementation, we performed 100 independent runs to compute the CE and the rate of success. The parameter values for the MP algorithm are `nTrs` = 1, `nFails` = 1 and the other parameters are set as shown in Table 4.1. The values of the parameters `nPop` and `nGnrs` are shown in Table 4.9 with the corresponding results of the MP algorithm. The results of GP with and without preventing FPCs are taken from the original paper [58]. As observed in Table 4.9, the MP algorithm outperforms the standard GP algorithm with and without preventing FPCs in terms of both the CE and the rate of success.

Table 4.9: The CE and the rate of success for the standard GP algorithm, with and without preventing FPCs, and the MP algorithm for the 6-BM problem

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

Table 4.10: Comparison among the TP algorithm and the MP algorithm for the 6-BM and 3-BEP problems.

	TP			MP		
Prob.	AV	ME	R%	AV	ME	R%
6-BM	7,829	6,393	98	9,019	8,158	100
3-BEP	5,612	4,272	100	12,934	10,641	84

4.4.4 MP vs TP

In this subsection we aim to compare between the TP algorithm, described in Chapter 3, and the MP algorithm for the SR-POLY-4, 6-BM and 3-BEP problems. During these comparisons, the results of the TP algorithm are taken from Fig. 3.4 for the SR-POLY-4 problem, and from Table 3.6 for the 6-BM and 3-BEP problems. On the other hand, the results of the MP algorithm for the SR-POLY-4 problem are taken form Fig. 4.6. In addition, 100 runs are performed using the MP algorithm for each of the 6-BM and 3-BEP problems. The parameter values of the MP algorithm are shown in Table 4.1, except for `nGnrs` $= 50$ to meet the maximum number of fitness evaluations used for the TP algorithm. Fig. 4.7 and Table 4.10 show the comparisons between the performance of the TP and the MP algorithms for the SR-POLY-4, 6-BM and 3-BEP problems.

From Fig. 4.7, one can note that the MP algorithm performs better than the TP

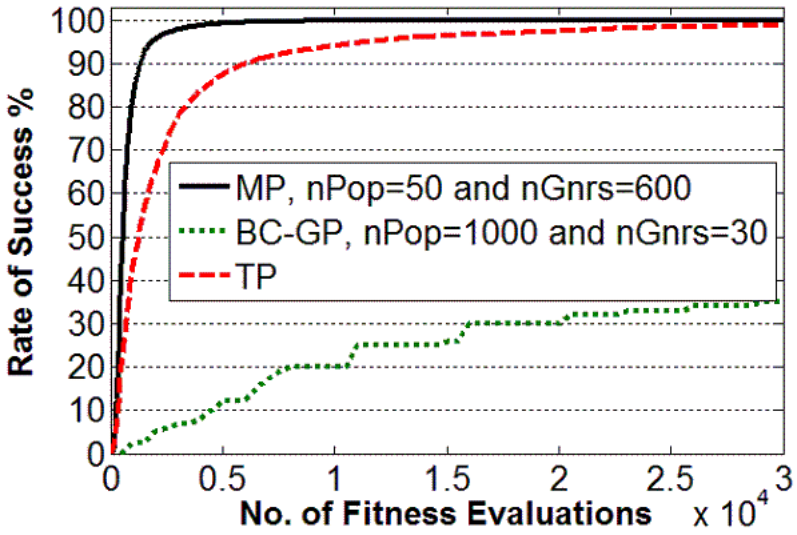

Figure 4.7: Comparison between the MP, TP and BC-GP algorithms for the SR-POLY-4

algorithm for the SR-POLY-4. Specifically, the MP algorithm can find an optimal solution for the SR-POLY-4 faster than the TP algorithm. Nevertheless, the TP algorithm outperforms the MP algorithm for the 6-BM and 3-BEP problems as shown in Table 4.10. Therefore, no one can argue that the TP algorithm is absolutely better than the MP algorithm or vise versa. Each algorithm can be more efficient for a specified set of problems.

4.5 Conclusions

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

Chapter 5

Applications

5.1 Introduction

A natural number greater than 1 is called a prime if it is only divisible by 1 and itself. For centuries, the study of prime numbers has been regarded as a subject of number theory in pure mathematics. Recently, this vision has changed and the importance of prime numbers increased rapidly especially in information technology, e.g., public key cryptography algorithms, hash tables, and pseudorandom number generators. One of the most popular topics that attract attention is to find a formula that maps the set of integers into the set of prime numbers. However, up to now there is no known formula that produces all primes. On the other hand, Pseudorandom Number Generators (PRNGs) play a key role in numerous algorithms of Computer Science and security. Because of the high cost of hardware implementations of True Random Number Generators (TRNGs), it is important to develop powerful and efficient PRNGs that can be implemented in hardware and software.

In this chapter, we aim to give real meaning for the proposed TP and MP algorithms by exploiting their high performances in useful applications of information technology. In the next section, we use the MP algorithm to discover a new set of mathematical formulas that can produce distinct prime numbers. In Section 5.3, the TP algorithm will be used to generate a set of highly nonlinear functions used as cores for high efficient PRNGs. Finally, conclusions make up Section 5.4.

5.2 Prime Number Generation

The study of prime numbers and their properties have attracted mathematicians for several centuries. Questions related to prime numbers have puzzled mathematicians for

many years, e.g., "is there a formula that maps the set of integers into the set of primes?" Recently, several applications in the field of information technology have increased the importance of prime numbers, and changed the vision that classifies the study of primes as pure mathematics.

The Memetic Programming (MP) algorithm is a new evolutionary algorithm that hybridizes the well-known Genetic Programming (GP) algorithm [68] with some local search procedures over a tree space to intensify promising programs generated by the GP algorithm. In this section, we use the MP algorithm to generate some mathematical formulas which produce distinct primes for a set of consecutive integers. In the next subsection, we give a brief reminder about the MP algorithm, which was presented in Chapter 4. Then, we end this section by some numerical experiments in which a new set of mathematical formulas are generated by the MP algorithm and these formulas can produce some distinct primes for consecutive integers.

5.2.1 MP Algorithm

The MP algorithm is a hybrid evolutionary algorithm that searches for desirable computer programs as outputs. Computer programs treated in the MP algorithm are represented as trees in which leaf nodes are called terminals and internal nodes are called functions. Depending on the problem at hand, the user defines the domains of terminals and functions. In the coding process, the tree structure of a solution should be transformed to an executable code.

The main loop of the MP algorithm consists of two phases; diversification phase and intensification phase. In the diversification phase, the MP algorithm guarantees the diversity in the current population by using the GP strategy. Specifically, the MP algorithm selects some programs using a suitable selection strategy, and generates a new population from the current one by using crossover and mutation operators. In the intensification phase, the MP algorithm uses a set of local search procedures to intensify elite programs of the current population through the LSP algorithm. The LSP is used to discover the best program in the neighborhood of the current program X, where X is generated using the crossover and mutation operations through the diversification phase. If the search process succeeds to reach the area near an optimal solution, then the LSP algorithm can capture that optimal solution easily. Moreover, the MP algorithm at least behaves like the GP algorithm if the LSP algorithm fails to improve the selected programs.

Procedures of the main loop in the MP algorithm, i.e., diversification and intensifica-

tion phases, are repeated until a termination condition is satisfied. Then, the algorithm stops with the best program obtained.

5.2.2 Numerical Experiments

In this subsection we report the results of three different experiments for the MP algorithm to generate formulas that produce primes. The parameter values for the MP algorithm during all experiments in this section are `hLen` = 3, `MaxLen` = 40, `nGenes` = 3, `nTrs` = 4, `nFails` = 1, `nPop` = 100 and `nGnrs` = 100. Other parameters are set as those in Table 4.4. The fitness value for each program is computed as the maximum number of consecutive integers in the interval $[-100, 100]$ for which the program produced distinct primes.

Polynomials

In this experiment, we used the set of binary functions $\{+, -, *\}$ as the function set, i.e., each program generated by the MP algorithm represents a polynomial. In addition, we used $\{x, 2, 3, 5, 7, 9\}$ as the terminal set, where x is an integer. We performed 1000 independent runs for the MP algorithm, and we got a number of polynomials with the fitness values up to 40.

Table 5.1 shows some of polynomials which generated by the MP algorithm. The first three polynomials in Table 5.1 have already been found in the literature. Specifically, the first two polynomials are the Euler and Legendre polynomials, and the third one is the polynomial generated by the CGP algorithm [113]. During our experiments, these three polynomials were found frequently. To the best of the authors' knowledge, the other polynomials seem to be new polynomials.

In the literature, researchers consider the first three polynomials in Table 5.1 to be different polynomials. However, all of these polynomials produce the same set of primes for different values of the independent variable x. Specifically, one can generate those entire polynomials one after another by using $x := x - \lambda$ for some integers λ. On the other hand, the last three polynomials in Table 5.1 produce different sets of distinct primes for different sets of consecutive integers. Therefore, we consider these three polynomials to be the best results for the current experiment since all of them are different and independent.

Table 5.1: Polynomials generated by the MP algorithm to produce distinct primes

	Polynomial	Fitness	x
1	$x^2 - x + 41$	40	{1,...,40}
2	$x^2 + x + 41$	40	{0,...,39}
3	$x^2 - 3x + 43$	40	{2,...,41}
4	$9x^2 + 33x + 71$	40	{-28,...,11}
5	$4x^2 - 50x + 197$	40	{-13,...,26}
6	$8x^2 - 22x - 647$	40	{-19,...,20}

Rational Functions

We performed another experiment to find formulas that produce primes with fitness values greater than 40. In this experiment, we modified the function set in the previous experiment to include the protected division operator %, where $x\%y = 1$ if $y = 0$, and $x\%y = x/y$ otherwise. In this case, programs of the MP algorithm will produce real values. Therefore, we let the nearest integer less than or equal to the produced real value be the output of the program. Using the new function set, we got several new formulas that produce up to 42 distinct primes for a set of consecutive integers. For example, $f(x) = \left|\left\lfloor \frac{-8x^3+69x^2-461x-176}{8x+3} \right\rfloor\right|$, with fitness value 42.

Composition Functions

Since we have already got new independent polynomials that can generate different sets of distinct primes, we can use these polynomials to composite new formulas. In this experiment, the output of a program evolved by the MP algorithm is expressed as a linear composition of its genes with some independent polynomials that produce distinct primes. Suppose that G_1, G_2 and G_3 are the genes of a program evolved by the MP algorithm. Then, the output formula of this program is composed as $f(x) = \lfloor G_1 \rfloor * P_1 + \lfloor G_2 \rfloor * P_2 + \lfloor G_3 \rfloor * P_3$, where P_1, P_2 and P_3 are independent polynomials. Using this strategy, we got new formulas that produce distinct primes up to 59, for example, $f(x) = \left|\left\lfloor \frac{7}{81x+27} \right\rfloor (x^2 + x + 41) + \left\lfloor \frac{9}{5-45x} \right\rfloor (8 * x^2 - 22 * x - 647)\right|$, with fitness value 59.

5.3 Efficient Pseudorandom Number Generators

One of the most important problems in computer security and cryptography is the designing of efficient Pseudorandom Number Generators (PRNGs). In fact, a function with a high degree of nonlinearity is essential for designing efficient PRNGs, since it has an excellent cryptographic property [75, 95]. In this section, we use the TP algorithm to generate a set of highly nonlinear functions, each of which can be used as the core of an efficient PRNG. In the next subsection, we show a mathematical property, called the Avalanche Effect, that can be used as a measure of the nonlinearity of a given function. Then, we speak about the TP briefly and this section is ended by some numerical experiments.

5.3.1 Avalanche Effect

The Avalanche Effect (AE) is an important property that reflects the intuitive idea of high nonlinearity of functions for cryptographic algorithms [75]. The AE property is evident if a very small change in the input of a function F produces an avalanche of changes in the output of F. Specifically, if $H(a, b)$ is the Hamming distance [41] between a and b, then the function $F : \{0, 1\}^m \rightarrow \{0, 1\}^n$ has the AE property if it satisfies the following condition:

$$\forall a, b | H(a, b) = 1 \quad \Rightarrow \quad Average\Big(H(F(a), F(b))\Big) = \frac{n}{2}. \tag{5.1}$$

That is, flipping randomly one single bit (a minimum change) of the input changes a half of the bits (a maximum change) of the output, on average. Actually, a perfect random function has a perfect AE property [75]. In practical implementations, researchers used to seek a function F that satisfies a more strong property called the Strict Avalanche Criterion [28] as in the following:

$$\forall a, b | H(a, b) = 1 \quad \Rightarrow \quad H(F(a), F(b)) \sim B(n, 1/2), \tag{5.2}$$

where $B(n, 1/2)$ is the Binomial distribution with parameters n and $1/2$. Consequently, a function that satisfies the Strict Avalanche Criterion (5.2) satisfies the AE property (5.1), since the mean value of the Binomial distribution $B(n, 1/2)$ is $n/2$. Now, our aim is to optimize the amount of the AE for programs generated by the TP algorithm. Therefore, in the next subsection, we will adapt the TP algorithm to discover functions that satisfy (5.2) to guarantee that the function has the AE property in (5.1).

5.3.2 TP algorithm

The proposed TP algorithm invokes three basic search stages; *local search*, *diversification* and *intensification*. In the local search stage, the TP algorithm uses the shaking, grafting and pruning procedures using branches of small depth. However, 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 3.1 shows the main structure of the TP algorithm.

The main loop in the algorithm starts with the static structure search, where the shaking procedure is used to generate a set of trial programs around the current one, based on the tabu restrictions. Then, the best program in the trial set replaces the current program, and the `TL` and other memory elements are updated. This process is repeated until a non-improvements condition is satisfied. Then, the algorithm proceeds to perform the dynamic structure search using the grafting and pruning procedures to explore new programs a little bit far from the current one. The best program generated using the grafting and pruning procedures replaces the current one, and the `TL` and other memory elements are updated. Then, the algorithm goes back to perform a new course of the static structure search using the new program generated by the dynamic structure search.

The procedures of the main loop in the TP algorithm are repeated until a termination condition is satisfied. If a diversification search is needed, the algorithm uses a diversification procedure to explore a new diverse program and goes back to the static structure search again. Otherwise, the algorithm refines the best programs found during the search process, and stops with the best program obtained.

5.3.3 Numerical Experiments

In this subsection we prepare the tools needed to apply the TP algorithm for generating highly nonlinear functions, e.g., the set of terminals, the set of functions, and the fitness function. Finally, our results for this experiment and the best program found are reported at the end of this subsection.

Terminal and Function Sets

In this experiment, we used eight unsigned integers, a_0, a_1, a_2, a_3, a_4, a_5, a_6, and a_7, as the terminal set. Moreover, each one of these terminals is represented as a binary number of 32-bit to produce a final input of a 256-bit. For more security, a constant c of a 32-bit

binary number generated randomly is used independently of the inputs, [75]. On the other hand, we included the most common efficient functions (i.e., easy to implement both in hardware and software) appeared in other implementations of PRNGs as the function set. Specifically, we included the following functions:

- ROTR(a): A one-bit right rotation for the input a.
- ROTL(a): A one-bit left rotation for the input a.
- SHFR(a): A one-bit right shift for the input a.
- SHFL(a): A one-bit left shift for the input a.
- NOT(a): The bitwise NOT for the input a.
- ROTRk(a, b): A k-bit right rotation for the input a, where k is the Hamming weight [115] of the input b.
- ROTLk(a, b): A k-bit left rotation for the input a, where k is the Hamming weight of the input b.
- SHFRk(a, b): A k-bit right shift for the input a, where k is the Hamming weight of the input b.
- SHFLk(a, b): A k-bit left shift for the input a, where k is the Hamming weight of the input b.
- SUM(a, b): The sum mod 2^{32} for the two inputs a and b.
- MULT(a, b): The multiplication mod 2^{32} for the two inputs a and b.
- AND(a, b): The bitwise AND for the two inputs a and b.
- OR(a, b): The bitwise OR for the two inputs a and b.
- XOR(a, b): The bitwise XOR for the two inputs a and b.

In fact, some researchers claim about the cost of the MULT function, e.g., the multiplication of two 32-bit values can be up to fifty times costlier than the AND operation [53]. Nevertheless, there are some widely used cryptographic primitives that use multiplication operator, like RC6 [100]. In addition, after extensive experimentation, we noticed that including the MULT function improve the results. For more details about the previous functions, see [26, 52, 75].

Fitness Function

We employ the following fitness function that is used frequently in such implementations [26, 52, 75]

$$Fit(X) = 10^9/\chi^2, \tag{5.3}$$

where χ^2 is the χ^2 goodness-of-fit test [37] that describes how well the Hamming distances, computed from the program X, follows the Binomial distribution $B(n, 1/2)$. Therefore, the aim here is to maximize the fitness function Fit, i.e., minimizing χ^2, for a program X. Because of the extremely high value of the χ^2 in the beginning of the search process, we amplified the fitness function by multiplying it by 10^9. Formally, we use the following procedure to compute the fitness value of a program X.

Procedure 5.1. *Fit(X)*

1 Start with an empty vector `RES`*. Specify positive integers M and N.*

2 Repeat the following Steps 2.1 - 2.5 N times.

2.1 ***Original Input****. Generate the tuple (a_0, a_1, a_2, a_3, a_4, a_5, a_6, and a_7) of 256-bit randomly.*

2.2 ***Original Output****. Compute the value of X using the original input in Step 2.1, and label it as* `ORG`*.*

2.3 ***New Input****. Flips M bits chosen randomly from the 256-bit input in Step 2.1.*

2.4 ***New Output****. Compute the value of X using the new input in Step 2.3, and label this value as* `NEW`*.*

2.5 ***Hamming Distance****. Compute the Hamming distance $H($*`ORG`*,* `NEW`*$)$, and then add it to the vector* `RES`*.*

3 Compute χ^2 for the observed values in `RES`*, using the χ^2 goodness-of-fit test.*

4 Return the fitness value Fit(X) computed from Equation (5.3).

In Step 2.1, the Mersenne Twister generator [86] is used to generate the tuple (a0, a1, a2, a3, a4, a5, a6, a7) randomly. In Step 2.3, the algorithm chooses M bits randomly from the original 256-bit input. The number M is determined from $M = \max(1, poissrnd)$, where *poissrnd* is a random value from the Poisson distribution [2] with parameter $\lambda = 0.5$. In fact, $M = 1$ for more than 90% of the total generated numbers using the previous strategy. In Step 3, the χ^2 goodness-of-fit test is used to detect how well the observed distribution of those computed Hamming distances follow their theoretical

Table 5.2: Results of the TP algorithm for the PRNGs experiment

Run	Best Fitness	AE value	Nodes No.
1	150,652,713.977	15.996	77
2	67,884,951.914	15.972	89
3	93,479,161.160	15.987	97
4	70,892,813.815	15.974	87
5	40,806,006.562	16.003	96
6	90,378,284.297	15.973	103
7	130,947,491.044	15.961	63
9	125,490,171.392	16.028	73
10	175,107,851.825	15.977	96

distribution $B(32, 1/2)$ under the perfect Strict Avalanche Criterion (5.2). Finally, the loop in Step 2 of Procedure 5.1 is repeated $N = 2^{14}$ times, which is shown experimentally to be enough and its results are trusted, see [75].

Results

For this problem, we performed 10 runs using the TP algorithm. The representation parameter values in this implementation of the TP algorithm are `hLen` = 3, `MaxLen` = 33, `nGenes` = 3 and `LnkFun` = SUM, while the search parameter values are `nTrs` = 5, `StNonImp` = 3, `MnNonImp` = 7, `IntNonImp` = 3, `nTL` = 7 and `FunCnt` = 3000.

In this experiment, we obtained numerous highly nonlinear functions that seem to be very nice compared to those results appeared in the literature. The fitness value, the amount of AE (the average of computed Hamming distances), and the number of nodes in the best program found in each run are shown in Table 5.2. It is clear from these results that all functions generated by the TP algorithm are highly nonlinear functions, where flipping one single bit of 256-bit input changes, approximately, 16 bits (the optimal value) on average for the 32-bit output.

Lamenca-Martinez et al. [75] have used the `lil-gp` library [126] of the GP algorithm to generate a highly nonlinear function as an efficient PRNG. Using the settings shown in the previous subsections, they have performed 20 runs for the GP algorithm with a population size of 150 individuals and 250 generations. The fitness value of the best program found by their experiment is 6,237,837.6345 and its AE value is 15.9631. In

addition, that program was discovered after performing 153 generations (22,950 fitness evaluations). On the other hand, we performed 10 runs of the TP algorithm with a maximum number of 3000 fitness evaluations for each run. The fitness value of the best program of our experiment is 175,107,851.825 and its AE value is 15.977. This means the fitness value of our best program is 28 times better than the fitness value of the best program found in Lamenca-Martinez et al. [75]. Moreover, the TP algorithm discovered the best program 7 times faster than the one obtained by the GP algorithm [75] for the same problem.

The genome representation of the best program discovered through our experiment is shown below, and its tree representation is shown in Fig. 5.1, where the hexadecimal format of the constant c is 1D416164. In addition, the SUM function is used to link genes of the TP programs.

Gene1. ROTLk XOR MULT a_2 ROTLk XOR ROTRk a_5 a_3 SUM SHFLk ROTL ROTLk SUM ROTRk c a_5 AND NOT a_2 a_5 c SHFLk a_1 a_1 ROTLk MULT a_6 a_2 a_6 a_0 a_1 a_4.

Gene2. ROTLk ROTRk XOR SHFR ROTRk SHFR ROTRk SHFL XOR SHFRk ROTR SHFL a_2 a_0 NOT ROTLk SHFRk a_0 a_0 a_1 a_4 XOR a_7 a_2 a_5 XOR ROTLk a_6 a_2 a_7 a_5.

Gene3. ROTRk SHFR XOR MULT SHFL NOT AND SHFL XOR ROTLk NOT a_7 a_0 SHFRk NOT SHFL XOR a_1 a_6 SHFLk MULT a_3 a_6 c a_7 a_1 SHFRk a_5 a_7 c.

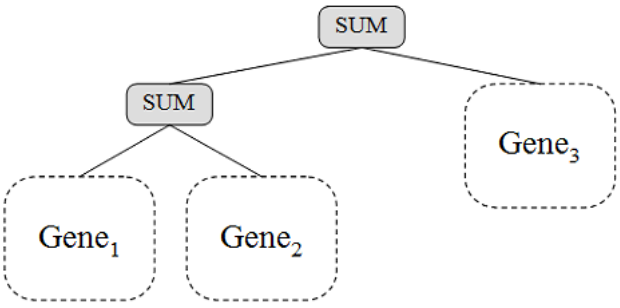

Figure 5.1: The overall structure of the best program found. Gene_1, Gene_2 and Gene_3 are shown in Figs. 5.2-5.4.

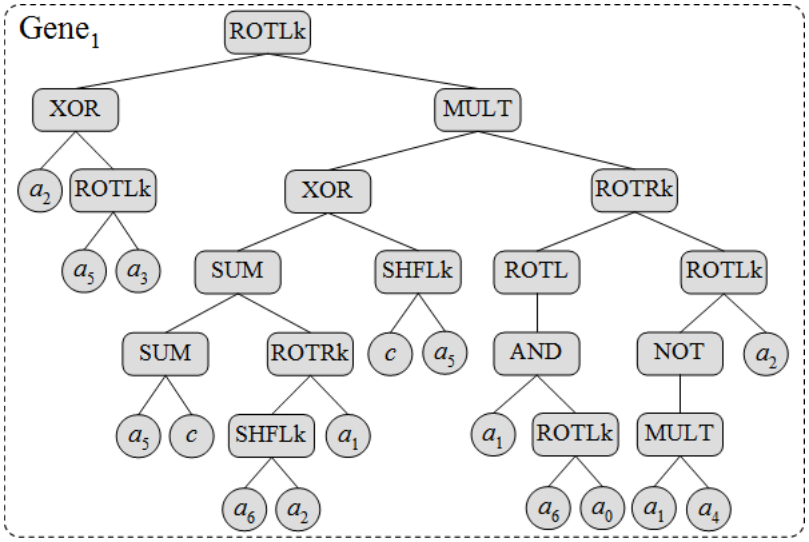

Figure 5.2: Gene_1 of the program shown in Fig. 5.1.

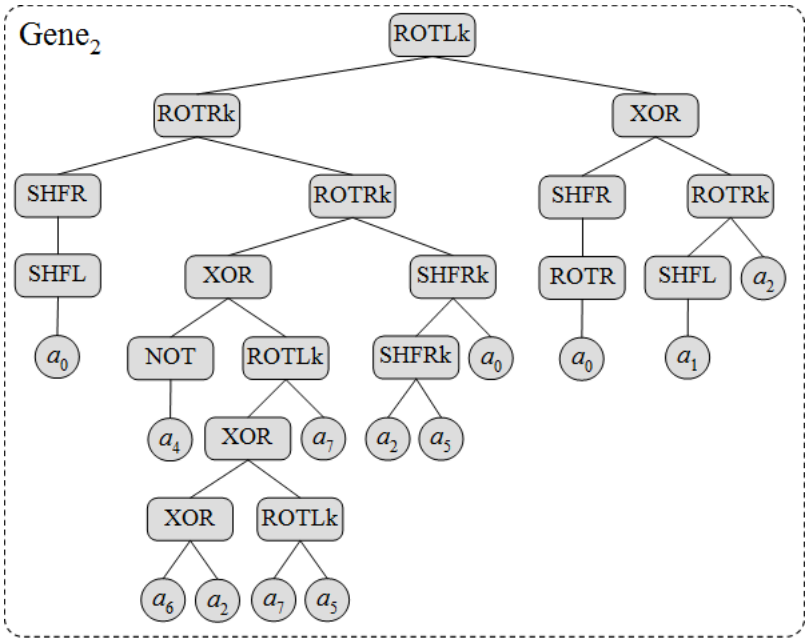

Figure 5.3: Gene_2 of the program shown in Fig. 5.1.

5.4 Conclusions

The MP algorithm has been used to generate new formulas that can produce sets of distinct primes. In fact, several mathematical formulas have been discovered by the MP algorithm. Some of the new formulas are polynomials that are able to produce up to 40 distinct primes for a set of consecutive integers. Other rational functions are also

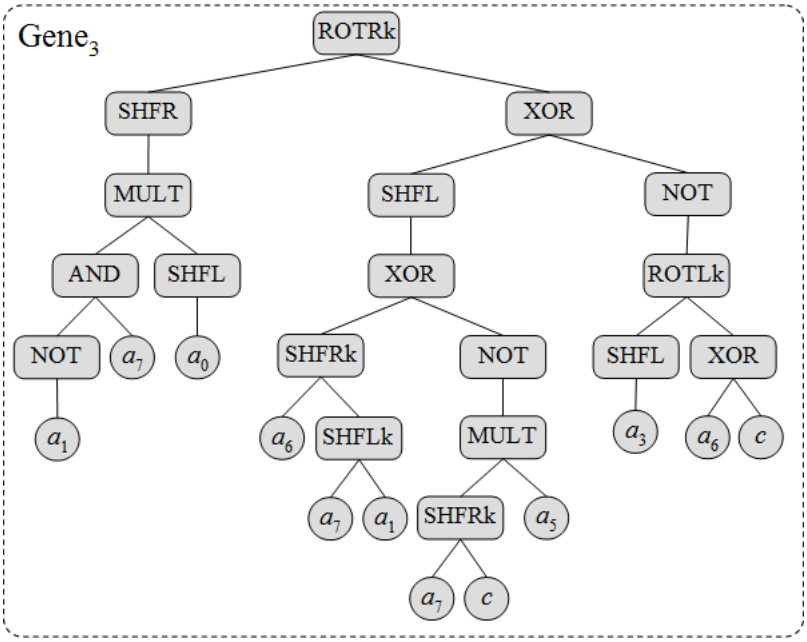

Figure 5.4: Gene_3 of the program shown in Fig. 5.1.

generated and they are able to produce up to 59 distinct primes for a set of consecutive integers.

On the other hand, the TP algorithm has been used to generate highly nonlinear functions as the cores for efficient Pseudorandom Number Generators (PRNGs). We have succeeded to discover a set of very efficient highly nonlinear functions. In the experiment, the TP algorithm not only succeeded to find very efficient nonlinear functions, but also discovered these efficient functions very fast compared to the implementations of the GP algorithm in the literature for the same problem.

Chapter 6

Summary and Conclusions

In this thesis, a set of local search procedures over a tree space has been used to propose a new comprehensive framework called the Meta-Heuristic Programming as a generator for machine learning tools. Specifically, the MHP framework uses the tree data structure to represent a solution, and uses the proposed local search procedures as breeding operators. Within the layout of the MHP framework, two new algorithms, the Tabu Programming (TP) and the Memetic Programming (MP) algorithms, have been proposed as alternatives to the well-known Genetic Programming (GP) algorithm. Moreover, the proposed algorithms have been used efficiently for two important applications in information technology, cryptography and security algorithms, the prime number generation and the designing of efficient Pseudorandom Number Generators (PRNGs).

The TP algorithm has been proposed in Chapter 3 to incorporate the search strategy of the well-known Tabu Search (TS) with the proposed local searches over a tree space. On the other hand, the MP algorithm in Chapter 4 hybridizes the standard GP algorithm with the proposed set of local search procedures to refine programs generated by the GP algorithm. Moreover, the proposed MP algorithm can deal easily with the Automatically Defined Functions (ADFs) technique based on the strategy of individual representation used in this thesis. Through extensive numerical experiments introduced in Chapters 3 and 4, we can conclude with the following remarks:

- The proposed TP and MP algorithms have shown promising behavior and both of them have outperformed the standard GP algorithm and several recent versions of it at least for the considered test problems.

- The MP algorithm has succeeded to use the ADF technique efficiently for the Boolean N-bit even-parity (N-BEP) problem of higher order.

- In Chapter 3, the GP algorithm using our local search procedures, instead of the mutation operator, has shown a promising performance for symbolic regression problems in comparison with two versions of the TP algorithm. However, the TP algorithm has outperformed the GP algorithm for the 3-BEP problem, even by using the local search procedures for the GP algorithm.
- In Chapter 4, the MP algorithm has shown to be more efficient than the TP algorithm for symbolic regression problems. On the other hand, the TP algorithm has shown to perform better than the MP algorithm for the 6-BM and 3-BEP problems.
- From the previous two points, one may conclude that the idea of finding an algorithm that always wins is impractical. Each algorithm has its own set of problems that it fits.

In Chapter 5, The MP algorithm has been used to detect new prime number generators. It has succeeded to discover a new set of polynomials that are able to produce up to 40 distinct primes for a set of consecutive integers. Moreover, rational functions that are able to produce up to 59 distinct primes for a set of consecutive integers have been discovered also. In a similar way, the TP algorithm has succeeded to generate a set of highly nonlinear functions. In fact, each function generated by the TP algorithm has the perfect Avalanche Effect property that enables those functions to work as an efficient PRNGs in cryptography and security algorithms. Moreover, the TP algorithm has shown high performance compared to the corresponding implementations of the GP algorithm for such applications.

Appendix A

Test Problems

A.1 Symbolic Regression Problem

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

A.1.1 Quartic Polynomial Problem

For the quartic polynomial function $f(x) = x^4 + x^3 + x^2 + x$, a dataset consisting of 20 fitness cases of the form $(x, f(x))$ has been obtained by choosing x uniformly at random in the interval $[-1, +1]$ [96]. The target in the problem (which will be referred to as the SR-QP problem) is to detect a function $g(x)$ that approximates the original polynomial QP, with the minimum error, by using the dataset. The fitness value for a program is calculated as the sum, with the sign reversed, of the absolute errors between the output produced by a program and the desired output for each of the fitness cases. Therefore, the maximum fitness value for this problem is 0.

A.1.2 Multivariate Polynomial Problem

For the multivariate polynomial function $f(x_1, \cdots, x_4) = x_1x_2 + x_3x_4 + x_1x_4$, a dataset consisting of 50 fitness cases of the form $(x_1, x_2, x_3, x_4, f(x_1, x_2, x_3, x_4))$ has been generated randomly with $x_i \in [-1, +1]$, $i = 1, 2, 3, 4$ [96]. The target in the problem (which will be referred to as the SR-POLY-4 problem) is to detect a function $g(x_1, x_2, x_3, x_4)$ that approximates the original polynomial POLY-4, with the minimum error, by using the dataset. The fitness value for a program is calculated as the sum, with the sign reversed, of the absolute errors between the output produced by a program and the

desired output for each of the fitness cases. Therefore, the maximum fitness value for this problem is 0.

A.2 Boolean N-Bit Even-Parity Problem

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

A.3 Boolean N-Bit Multiplexer Problem

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

Bibliography

[1] A. Abraham, C. Grosan and C. Martin-Vide, Evolutionary design of intrusion detection programs, *International Journal of Network Security* 4(3) (2007) 328–339.

[2] J.H. Ahrens and U. Dieter, Computer methods for sampling from Gamma, Beta, Poisson and Binomial distributions, *Computing* 12(3) (1974) 223–246.

[3] T. Andersson, Solving the flight perturbation problem with meta heuristics, *Journal of Heuristics* 12 (2006) 37–53.

[4] M.S. Arumugam and M.V.C. Rao, On a class of hybrid systems via a novel approach for real-coded genetic algorithm with hybrid selection, *International Journal of Information Technology & Decision Making* 6(2) (2007) 315–332.

[5] R.M.A. Azad and C. Ryan, An examiation of simultaneous evolution of grammars and solutions, in *Genetic Programming: Theory and Practice III*, eds. T. Yu, R.L. Riolo and B. Worzel (Springer-Verlag, 2006), pp. 141–158.

[6] M.B. Bader-El-Den, R. Poli and S. Fatima, Evolving timetabling heuristics using a grammar-based genetic programming hyper-heuristic framework, *Memetic Computing* 1 (2009) 205–219.

[7] J. Balicki, Tabu programming for multiobjective optimization problems, *International Journal of Computer Science and Network Security* 7(10) (2007) 44–51.

[8] Z. Bankovic, J.M. Moya, A. Araujo, S. Bojanic and O. Nieto-Taladriz, Improving network security using genetic algorithm approach, *Computers & Electrical Engineering* 33(5-6) 2007 438–451.

[9] 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).

[10] M. Birattari, L. Paquete, T. Sttzle and K. Varrentrapp, Classification of metaheuristics and design of experiments for the analysis of components, Technical Report AIDA-01-05, Darmstadt University of Technology (2001).

[11] T. Blickle and L. Thiele, Genetic programming and redundancy, in *Proc. Genetic Algorithms within the Framework of Evolutionary Computation* (Workshop at KI-94, Saarbrucken, 1994), pp. 33–38.

[12] T. Blickle and L. Thiele, Comparison of selection schemes used in evolutionary algorithms, *Evolutionary Computation* 4 (1997) 361–394

[13] C. Blum and A. Roli, Metaheuristics in combinatorial optimization: Overview and conceptual comparison, *ACM Computing Surveys* 35(3) (2003) 268–308.

[14] M. Boryczka, Eliminating introns in ant colony programming, *Fundamenta Informaticae* 68 (2005) 1–19.

[15] M. Boryczka and Z.J. Czech, Solving approximation problems by ant colony programming, in *Proc. Genetic and Evolutionary Computation Conference GECCO 2002* (New York, NY, 2002), pp. 39–46.

[16] M. Brameier and W. Banzhaf, A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1) (2001) 17–26.

[17] E.C Brown, C.T. Ragsdale and A.E. Carter, A grouping genetic algorithm for the multiple traveling salesperson problem, *International Journal of Information Technology & Decision Making* 6(2) (2007) 333–347.

[18] L. Bull, E. Bernad-Mansilla and J. Holmes, Learning classifier systems in data mining: An introduction, in *Learning Classifier Systems in Data Mining, Studies in Computational Intelligence*, eds. L. Bull, E. Bernadó-Mansilla and J.H. Holmes (2008), pp. 1–16.

[19] E.K. Burke, S. Gustafson and G. Kendall, Diversity in genetic programming: An analysis of measures and correlation with fitness, *IEEE Transactions on Evolutionary Computation* 8(1) (2004) 47–62.

[20] T. Castle and C.G. Johnson, Positional effect of crossover and mutation in grammatical evolution, *Lecture Notes in Computer Science* 6021 (2010) 26–37.

[21] N.L. Cramer, A representation for the adaptive generation of simple sequential programs, in *Proc. International Conference on Genetic Algorithms and their Applications* (Carnegia Mellon University, Pittsburgh, USA, 1985), pp. 183–187.

[22] I.G. Damousis, A.G. Bakirtzis and P.S. Dokopoulos, A solution to the unit-commitment problem using integer-coded genetic algorithm, *IEEE Transactions on Power Systems* 19(2) (2004) 1165–1172.

[23] R. Diestel, *Graph Theory* (Springer-Verlag, Berlin, 2005).

[24] M. Dorigo and T. Stützle, *Ant Colony Optimization* (The MIT Press, 2004).

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

[26] C. Estébanez, J.C. Hernández-Castro, A. Ribagorda and P.I. Vióuela, Finding state-of-the-art non-cryptographic hashes with genetic programming, *Lecture Notes in Computer Science* 4193 (2006) 818–827.

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

[28] R. Forré, The strict avalanche criterion: Spectral properties of Boolean functions and an extended definition, *Lecture Notes in Computer Science* 403 (1990) 450468.

[29] J. Garcia-Nieto, J. Toutouh and E. Alba, Automatic tuning of communication protocols for vehicular ad hoc networks using metaheuristics, *Engineering Applications of Artificial Intelligence* 23(5) (2010) 795-805.

[30] F. Glover, Future paths for integer programming and links to artificial intelligence, *Computers & Operations Research* 13 (1986) 533–549.

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

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

[33] F. Glover and G. Kochenberger, New optimization models for data mining, *International Journal of Information Technology & Decision Making* 5(4) (2006) 605–609.

[34] F. Glover, M. Laguna and R. Marti, Fundamentals of scatter search and path relinking, *Control and Cybernetics* 39(3) (2000) 653–684 .

[35] 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.

[36] F. Glover, E. Taillard and D. Werra, A user's guide to tabu search, *Annals of Operations Research* 41(1993) 3–28.

[37] D.M. Glover, W.J. Jenkins and S.C. Doney, Least Squares and regression techniques, goodness of fit and tests, non-linear least squares techniques, in *Modeling Methods for Marine Science* (Woods Hole Oceanographic Institution, 2008), pp. 45–66.

[38] D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning* (New York: Addison-Wesley, 1989).

[39] D.E. Goldberg, *The Design of Innovation: Lessons from and for Competent Genetic Algorithms* (New York: Addison-Wesley, 2002).

[40] C. Grosan, A. Abraham and S.Y. Han, MEPIDS: Multi-expression programming for intrusion detection system, *Lecture Notes in Computer Science* 3562 (2005) 163–172.

[41] R.W. Hamming, Error detecting and error correcting codes, *Bell System Technical Journal* 29(2) (1950) 147–160.

[42] J.V. Hansen, P.B. Lowry, R.D. Meservy and D.M. McDonald, Genetic programming for prevention of cyberterrorism through dynamic and evolving intrusion detection, *Decision Support Systems* 43(4) (2007) 1362–1374.

[43] W. Hart, N. Krasnogor and J. Smith (eds.), *Recent Advances in Memetic Algorithms* (Springer, Berlin, Heidelberg, New York, 2004).

[44] E. Hart and J. Timmis, Application areas of AIS: The past, the present and the future, *Applied Soft Computing* 8 (2008) 191–201.

[45] A. Hedar and M. Fukushima, Hybrid simulated annealing and direct search method for nonlinear unconstrained global optimization, *Optimization Methods and Software* 17 (2002) 891–912.

[46] A. Hedar and M. Fukushima, Heuristic pattern search and its hybridization with simulated annealing for nonlinear global optimization, *Optimization Methods and Software* 19 (2004) 291-308.

[47] A. Hedar and M. Fukushima, Tabu search directed by direct search methods for nonlinear global optimization, *European Journal of Operational Research* 170 (2006) 329–349.

[48] A. Hedar and M. Fukushima. Derivative-free filter simulated annealing method for constrained continuous global optimization, *Journal of Global Optimization* 35 (2006) 521–549.

[49] A. Hedar and M. Fukushima, Meta-heuristics programming, in *Proc. 2nd International Workshop on Computational Intelligence & Applications* (Okayama, Japan, 2006).

[50] A. Hedar and M. Kamel, Scatter programming, in *Proc. 7th International Conference on Informatics and Systems (INFOS 2010)* (Cairo, Egypt, 2010).

[51] A. Hedar, E. Mabrouk and M. Fukushima, Tabu programming: A new problem solver through adaptive memory programming over tree data structures, *The International Journal of Information Technology & Decision Making*, to appear (2011).

[52] J.C. Hernández-Castro, P.I. Viñuela and C.L. del Arco-Calderón, Finding efficient nonlinear functions by means of genetic programming, *Lecture Notes in Computer Science* 2773 (2003) 1192–1198.

[53] G. Hinton et al., The microarchitecture of the pentium 4 processor, *Intel Technology Journal* Q1 2001.

[54] N.X. Hoai, R.I. McKay and D. Essam, Representation and structural difficulty in genetic programming, *IEEE Transactions on Evolutionary Computation* 10(2) (2006) 157–166.

[55] 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 (Springer-Verlag, 2006), pp. 159–175.

[56] J. Howe, Artificial intelligence at Edinburgh University: A perspective, School of Informatics, Edinburgh University (2008).

[57] T. Ito, H. Iba and S. Sato, Non-destructive depth-dependent crossover for genetic programming, *Lecture Notes in Computer Science* 1391 (1998) 71–82.

[58] D. Jackson, The identification and exploitation of dormancy in genetic programming, *Genetic Programming and Evolvable Machines* 11 (2010) 89–121.

[59] C.G. Johnson, Genetic Programming Crossover: Does It Cross over?, *Lecture Notes in Computer Science* 5481 (2009) 97–108.

[60] M. Keijzer, C. Ryan, M. ONeill, M. Cattolico and V. Babovic, Ripple crossover in genetic programming, *Lecture Notes in Computer Science* 2038 (2001) 74–86.

[61] M. Kessler and T. Haynes, Depth-fair crossover in genetic programming, in *Proc. 1999 ACM Symposium on Applied Computing* (San Antonio, Texas, US, 1999), pp. 319–323.

[62] K.E. Kinnear Jr. (ed.), *Advances in Genetic Programming* (MIT Press, Cambridge, MA, 1994).

[63] S. Kirkpatrick, C.D. Gelatt Jr. and M.P. Vecchi, Optimisation by simulated annealing, *Science* 220 (1983) 671–680.

[64] J.K. Kishore, L.M. Patnaik, V. Mani and V.K. Agrawal, Application of genetic programming for multicategory pattern classification, *IEEE Transactions on Evolutionary Computation* 4 (2000) 242–258.

[65] P. Kouchakpour, A. Zaknich and T. Bräunl, A survey and taxonomy of performance improvement of canonical genetic programming, *Knowledge and Information Systems* 21 (2009) 1–39.

[66] J.R. Koza, Hierarchical genetic algorithms operating on populations of computer programs, in *Proc. 11th Int. Joint Conference on Artificial Intelligence* (Morgan Kaufmann: Los Altos, CA, 1989), pp. 768–774.

[67] 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 (1990).

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

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

[70] 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).

[71] 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).

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

[73] N. Krasnogor and J.E. Smith, A tutorial for competent memetic algorithms: model, taxonomy and design issues, *IEEE Transactions on Evolutionary Computation* 9 (2005) 474–488.

[74] M. Laguna and R. Marti, *Scatter Search: Methodology and Implementations in C* (Kluwer Academic Publishers, Boston, 2003).

[75] C. Lamenca-Martinez, J.C. Hernández-Castro, J.M. Estévez-Tapiador and A. Ribagorda, Lamar: A new pseudorandom number generator evolved by means of genetic programming, *Lecture Notes in Computer Science* 4193 (2006) 850–859.

[76] W.B. Langdon and R. Polií, *Foundations of Genetic Programming* (Springer-Verlag 2002).

[77] X.Y. Li, X.Y. Shao and L. Gao, Optimization of flexible process planning by genetic programming. *International Journal of Advanced Manufacturing Technology* 38 (2008) 143–153.

[78] M.A.C. Lima, A.F.R. Araujo and A.C. Cesar, Adaptive genetic algorithms for dynamic channel assignment in mobile cellular communication systems, *IEEE Transactions On Vehicular Technology* 56(5) (2007) 2685–2696.

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

[80] T.-Y. Liu, Learning to rank for information retrieval, *Foundations and Trends in Information Retrieval* 3(3) (2009) 225–331.

[81] H.S. Lopes, Genetic programming for epileptic pattern recognition in electroencephalographic signals, *Applied Soft Computing* 7(1)(2007) 343–352.

[82] E. Mabrouk, A. Hedar and M. Fukushima, Memetic programming with adaptive local search using tree data structures, in *Proc. 5th International Conference on Soft Computing as Transdisciplinary Science and Technology* (Cergy-Pontoise, Paris, France, 2008), pp. 258–264.

[83] E. Mabrouk, A. Hedar and M. Fukushima, Tabu programming: a machine learning tool using adaptive memory programming, in *Proc. 6th International Conference on Modeling Decisions for Artificial Intelligence* (Awaji Island, Japan, 2009), pp. 187–198.

[84] E. Mabrouk, J.C. Hernández-Castro and M. Fukushima, Prime number generation using memetic programming, in *Proc 16th International Symposium on Artificial Life and Robotics* (B-Con Plaza, Beppu, Oita, Japan, 2011).

[85] 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.

[86] M. Matsumoto and T. Nishimura, Mersenne twister: A 623-dimensionally equidistributed uniform PRNG. *ACM Transactions on Modeling and Computer Simulation* 8(1) (1998) 3–30.

[87] P. McCorduck, *Machines Who Think* (2nd ed.) (Natick, MA: A K Peters, Ltd., 2004).

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

[89] P. Moscato and C. Cotta, A gentle introduction to memetic algorithms, in *Handbook of Metaheuristics*, eds. F. Glover and G. Kochenberger (Kluwer Academic Publishers, Boston MA, 2003), pp. 105–144.

[90] P. Moscato, C. Cotta and A. Mendes, Memetic algorithms, in *New Optimization Techniques in Engineering*, eds. G.C. Onwubolu and B.V. Babu (Springer-Verlag, Berlin Heidelberg, 2004), pp. 53–85.

[91] L. Nie, X. Xu and D. Zhan, Collaborative planning in supply chains by Lagrangian relaxation and genetic algorithms, *International Journal of Information Technology & Decision Making* 7(1) (2008) 183–197.

[92] P. Nordin and W. Banzhaf, Complexity compression and evolution, in *Proc. 6th International Conference on Genetic Algorithms* (Morgan Kaufmann, Pittsburgh, PA, USA, 1995), pp. 310–317.

[93] 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* (Tahoe City, California, USA, 1995), pp. 6–22.

[94] A. Orfila, J.M. Estvez-Tapiador and A. Ribagorda, Evolving high-speed, easy-to-understand network intrusion detection rules with genetic programming, *Lecture Notes in Computer Science* 5484 (2009) 93–98.

[95] P. Peris-Lopez, Lightweight cryptography in radio frequency identification (RFID) systems, PhD Thesis, Computer Science Department, Universidad Carlos III de Madrid (2008).

[96] R. Poli and W.B. Langdon, Backward-chaining evolutionary algorithms, *Artificial Intelligence* 170 (2006) 953-982.

[97] R. Poli and J. Page, Solving high-order Boolean parity problems with smooth uniform crossover, sub-machine code GP and demes, *Genetic Programming and Evolvable Machines* 1 (2000) 37–56.

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

[99] R. Riolo, T. Soule and B. Worzel (eds.), *Genetic Programming Theory and Practice V* (Springer-Verlag, Berlin, 2008).

[100] R.L. Rivest, M.J.B. Robshaw, R. Sidney and Y.L. Yin, The RC6 block cipher, v1.1 (1998).

[101] F. Rothlauf, On optimal solutions for the optimal communication spanning tree problem, *Operations Research* 57(2) (2009) 413–425.

[102] F. Rothlauf, An encoding in metaheuristics for the minimum communication spanning tree problem, *INFORMS Journal on Computing* 21(4) (2009) 575–584.

[103] S.J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach* (2nd ed.) (Upper Saddle River, New Jersey: Prentice Hall, 2003).

[104] S. Salcedo-Sanz et al., Optimal switch location in mobile communication networks using hybrid genetic algorithms, *Applied Soft Computing* 8 (2008) 1486-1497.

[105] S. Silva, GPLAB: A genetic programming toolbox for MATLAB, http://gplab.sourceforge.net/

[106] W.A. Tackett, Recombination, selection and the genetic construction of computer programs, PhD Thesis, Department of Electrical Engineering Systems, University of Southern California (1994).

[107] W.A. Tackett and A. Carmi, The unique implications of brood selection for genetic programming, in *Proc. first IEEE Conference on Evolutionary Computation* (IEEE Press, Volume 1, New York, NY, 1994), pp. 160–165.

[108] M.D. Terrio and M.I. Heywood, Directing crossover for reduction of bloat in GP, in *Proc. Canadian Conference on Electrical and Computer Engineering* (Piscataway, NJ: IEEE Press, 2002), pp. 1111–1115.

[109] K.P. Vekaria, Selective crossover as an adaptive strategy for genetic algorithms, PhD Thesis, University College (1999).

[110] E. Vladislavleva, Model-based problem solving through symbolic regression via Pareto genetic programming, PhD Thesis, Tilburg University (2008).

[111] E. Vladislavleva, G. Smits and D. den Hertog, Order of nonlinearity as a complexity measure for models generated by symbolic regression via Pareto genetic programming, *IEEE Transactions on Evolutionary Computation* 13(2) (2009) 333–349.

[112] M.D. Vose, *The Simple Genetic Algorithm: Foundations and Theory* (Cambridge, MA: MIT Press, 1999).

[113] J.A. Walker and J.F. Miller, Predicting prime numbers using Cartesian genetic programming, in *Proc. 10th European Conference on Genetic Programming* (Valencia, Spain, 2007), pp. 205–216.

[114] J.A. Walker and J.F. Miller, The automatic acquisition, evolution and reuse of modules in cartesian genetic programming, *IEEE Transactions on Evolutionary Computation* 12(4) (2008) 397–417.

[115] P. Wegner, A technique for counting ones in a binary computer, *Communications of the ACM* 3(5) (1960) 322.

[116] E. William and 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.

[117] S. Winkler, M. Affenzeller and S. Wagner, Advanced genetic programming based machine learning, *Journal of Mathematical Modelling and Algorithms* 6(3) (2007) 455–480.

[118] 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).

[119] D.H. Wolpert and W.G. Macready, No free lunch theorems for optimization, *IEEE Transactions on Evolutionary Computation* 1(1) (1997) 67–82.

[120] A.S. Wu and W. Banzhaf, Introduction to the special issue: Variable-length representation and noncoding segments for evolutionary algorithms, *Evolutionary Computation* 6(4) (1998) iii–vi.

[121] C. Wu, H. Chou and W. Su, Direct transformation of coordinates for GPS positioning using the techniques of genetic programming and symbolic regression, *Engineering Applications of Artificial Intelligence* 21(8) (2008) 1347–1359.

[122] H. Xie, An analysis of selection in genetic programming, PhD Thesis, Victoria University of Wellington (2009).

[123] M. Zhang, X. Gao and W. Lou, A new crossover operator in genetic programming for object classification, *IEEE Transactions on Systems Man and Cybernetics Part B* 37 (2007) 1332–1343.

[124] L. Zhang and A.K. Nandi, Fault classification using genetic programming, *Mechanical Systems and Signal Processing* 21(3) (2007) 1273–1284.

[125] L. Zhang and A.K. Nandi, Diversity-preserving non-destructive operators in genetic programming and their application to breast cancer diagnosis, *Transactions of the Institute of Measurement and Control*, 31(6) (2009) 533–550.

[126] The lil-gp GP system. http://http://garage.cse.msu.edu/software/lil-gp/.