Evolving choice structures for genetic programming

Shuaiqiang Wang\textsuperscript{a,b}, Jun Ma\textsuperscript{a,\ast}, Jiming Liu\textsuperscript{b}, Xiaofei Niu\textsuperscript{c}

\textsuperscript{a} School of Computer Science and Technology, Shandong University, 250101, Jinan, China
\textsuperscript{b} Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, Hong Kong
\textsuperscript{c} School of Computer Science and Technology, Shandong Jianzhu University, 250101, Jinan, China

\textbf{Abstract}

It is quite difficult but essential for Genetic Programming (GP) to evolve the choice structures. Traditional approaches usually ignore this issue. They define some “if-structures” functions according to their problems by combining “if-else” statement, conditional criterions and elemental functions together. Obviously, these if-structure functions depend on the specific problems and thus have much low reusability. Based on this limitation of GP, in this paper we propose a kind of termination criterion in the GP process named “Combination Termination Criterion” (CTC). By testing CTC, the choice structures composed of some basic functions independent to the problems can be evolved successfully. Theoretical analysis and experiment results show that our method can evolve the programs with choice structures effectively within an acceptable additional time.

\textcopyright 2010 Elsevier B.V. All rights reserved.

1. Introduction

Genetic Programming (GP) \cite{3} is a technique pioneered by John Koza which enables computers to solve problems without being explicitly programmed. Since then, many GP variants were proposed in order to improve the evolutionary ability of GP.

In \cite{4} Miller et al. provided a graph-based GP technique named Cartesian Genetic Programming (CGP). CGP employed a neutral search strategy which was proved to be much more effective. Walker et al. \cite{9} extended CGP by utilizing automatic module acquisition, evolution, and reuse. Zhang et al. \cite{10} introduced a paradigm called Immune Genetic Programming (IGP), which was inspired from the principles of the vertebrate immune system. In the Stack-based GP system \cite{7} approach the target programs run on a stack-based virtual machine. The system was shown to have certain advantages in terms of efficiency and simplicity of implementation. The Grammar-Guided Genetic Programming (GGGP) \cite{5} replaced common GP genetic operator with a context-free grammar. The authors declared that it could significantly outperform conventional GP, learning faster and more reliably, and could be easily configured to solve different problems without significant modifications in their kernel to be adapted to other domains \cite{8}. Brameier et al. proposed the Linear Genetic Programming (LGP), whose main character in comparison to tree-based GP was that the expressions of a functional programming language (like LISP) were substituted by programs of an imperative language (like C) \cite{1}.

Although these approaches can evolve some linear programs perfectly, it seems to fail to resolve the problems with choice structures efficiently. As we know that there are mainly three structures in the programs: sequence, choice and loop. For example, both the \texttt{if-else} and the \texttt{switch-case} statements in the C programming language are two implementation forms of the choice structures. Thus we believe that if the choice structures cannot be evolved properly, the evolutionary capability is far from complete.

However, it is really difficult to evolve the choice structures. A choice structure can be expressed formally as $A_1 \Rightarrow a_1 \mid A_2 \Rightarrow a_2 \mid \cdots \mid A_n \Rightarrow a_n$ where $A_1, A_2, \ldots, A_n$ are...
optional criterions and $a_1, a_2, \ldots, a_t$ are optional actions. According to this formal expression we can conclude that there are mainly four issues which should be taken into account: (i) How to evolve the optional actions composed of the elemental functions; (ii) How to evolve the optional conditional criterions based on the values assigned to the variables; (iii) How to match the actions and the conditional criterions one by one appropriately; (iv) How to make sure to evolve the choice structures in an acceptable additional time.

Owing to these difficulties, the traditional GP approaches often ignore this problem. Instead, they commonly define some if-structures manually according to their problem. These complex structures usually integrate the conditional criterions and elemental functions into the if-else statements. Then these if-structures can be employed as the basic functions in their approaches. For example, in order to evolve a sort program, Kinnear defined an intelligent_swap function [2]. This function could swap the adjacent elements only if they were out of order. With the help of the function a sort program could be generated easily. On the contrary, O’Reilly et al. [6] employed swap, a much essential and reusable function, to take the place of intelligent_swap. However, they reported a failure attempt. However, because of the close relationship between the if-structures and their problems, these complex actions have quite low reusability and almost cannot be adopted by other problems.

In the GGGP approach [5] O’Neill et al. evolved the choice structures for the Santa Fe ant problem via defining the if-structure grammars. In their experiments the if-statement grammar was expressed as follows.

$$<\text{if-statement}> \equiv \text{if (food\_ahead())}$$

$$\{ <\text{expr}> \}$$

$$\quad\text{else}$$

$$\{ <\text{expr}> \}$$

It should be pointed out that this method can solve the first issue among the four critical points mentioned above. But it fails to consider the left three aspects. What’s more, there is still a sound correlation between the grama and the Santa Fe ant problem, which lies in that: (i) The conditional criterion, food\_ahead, is designated manually based on the Santa Fe ant problem; (ii) The optional action number has been predesigned as two; (iii) The GP process can only take into account the matching between the two optional actions and food\_ahead, and thus the cost of the increasing time can be controlled easily. Therefore we can conclude that this method still seems to be lack of a sufficient ability to evolve the choice structures completely.

In this paper, we propose a feasible approach to evolve programs with choice structures completely, thus the evolutionary ability of GP will be improved essentially. It means that our method can consider the four aspects mentioned above simultaneously, and make an appropriate balance among them. To be specific, the contributions of our work are expressed as follows: (i) Our method can evolve the optional actions and conditional criterions separately; (ii) Our method can choose a correct matching from all of the available actions and criterions; (iii) Our method can make a practical decision to keep the additional time for the evolution of the choice structures within a reasonable limit.

This paper is organized as follows. Section 2 denotes our approach, a GP variant which can evolve the choice structures automatically. As an example, the evolution process of a baseline function intelligent_swap is shown in Section 3. Finally, the conclusions are offered in Section 4.

## 2. Evolving the choice structures

In this section, we will introduce the mechanisms to demonstrate how to evolve the choice structures via the genetic programming technologies.

### 2.1. The algorithm

It is necessary to specify the terminals, functions, fitness function, control parameters and termination criterion to solve a problem using GP Koza [3] states. The first three issues in our approach are the same as those of the traditional GP approaches.

Here we intend to specify the fitness function in detail. In the beginning, $N_c$ fitness cases are generated firstly. Each fitness case includes a series of input and output values, which satisfy the problem definitions exactly. Then we can test the individuals in the current population whether these fitness cases hold or not. For example, if an individual is $In \equiv f(x) = x + 1$ and a fitness case is $c = [x = 1, f(x) = 2]$ which means the input value of the variable $x$ is 1 and the output value is 2, obviously, $In$ can satisfy $c$. Let $C(In)$ be the set composed of the fitness cases which can be satisfied by the individual $In$. Then we can make a definition that the fitness value of the given individual $In$ is the total correct amount of its fitness tests, formally, $\mathcal{F}(In) = |C(In)|$.

The control parameters include the population size $n$, the maximum generations $G$, the maximum individual length $N_i$, the probabilities of crossover $P_c$, and mutation $P_m$, the fitness case number $N_c$, and the maximum choices number $N_b$.

In order to evolve the choice structures, we propose a novel termination criterion named Combination Termination Criterion (CTC). For comparison purposes, we call the traditional termination criterion Individual Termination Criterion (ITC).

ITC holds if $\mathcal{F}(In) = N_c$. On the contrary, the combination termination occurs only if there exist a set of individuals $\{In_i\}_{i \in J}$ whose correct fitness case set $C(\{In_i\}_{i \in J})$ satisfy the following formulae where $J = \{J_1, J_2, \ldots, J_{|J|}\}$.

$$\sum_{i=1}^{J} \mathcal{F}(In_i) = N_c$$

$$\forall j_m, j_n \in J \bullet (j_m \neq j_n \Rightarrow C(\{In_{j_m}\}) \cap C(\{In_{j_n}\}) = \emptyset) \quad (1)$$

Especially if $J = \{J_1, J_2\}$, Eq. (1) is simplified as the following formulae.

$$\mathcal{F}(In_{J_1}) + \mathcal{F}(In_{J_2}) = N_c$$

$$C(\{In_{j_1}\}) \cap C(\{In_{j_2}\}) = \emptyset \quad (2)$$
There are two if-structure statements in this algorithm: choice structures, e.g., (5).

Actually, they all omit a statement “else doNothing”. Obviously, the probabilities of both ITC and CTC are all much lower than those of ¬ITC and ¬CTC respectively.

According to the assumption, if we intend to increase the probability of satisfying CTC, we must polarize the individuals. In other words, the individuals should evolve into two extremes, either high fit or low fit. Thus the individuals whose fitness values are close to NC/2 should trend to be removed.

In order to implement our algorithm, some platform information should be considered, e.g., the bit number of CPU. Provided that the processor is PB bits. For a certain individual In, a truth table \( T(In)(PB, NC/PB) \) recording the logic results of testing fitness is maintained.\(^1\) Obviously the number of “1” in the truth table of In is its fitness value \( \mathcal{F}(In) \). If a series of individuals \( \{In_j\}_{j\in J} \) satisfy CTC, two criterions should be satisfied: (i) the sum of their fitness values is NC, called Fitness Criterion; (ii) for each fitness case, only one individual among the individual set can make it hold, called Element Criterion. Formally,

\[
\sum_{l=1}^{\lvert J \rvert} \mathcal{F}(In_{j_l}) = NC
\]

\[
\forall m \in (1..PB), \; n \in \left(1, \frac{NC}{PB}\right) \bullet \left(\sum_{l=1}^{\lvert J \rvert} T(In_{j_l})(m, n) = 1\right)
\]

Especially if \( \lvert J \rvert = 2 \), that is, \( J = \{j_1, j_2\} \), the operator “+” in the formula (3) will be substituted by the exclusive-OR operator “⊕”, which is expressed as follows.

\[
\mathcal{F}(In_{j_1}) + \mathcal{F}(In_{j_2}) = NC
\]

\[
\forall m \in (1..PB), \; n \in \left(1, \frac{NC}{PB}\right) \bullet \left(T(In_{j_1})(i, j) \oplus T(In_{j_2})(i, j)\right)
\]

2.2. Implementation

When the next generation is evolving according to the current population, the substitution strategy in our approach is a little different from that in the traditional GP approaches. The traditional GP try to make sure that some generated high fit individuals can take the place of those individuals with lowest fitness values. This strategy may lead to a fact that most of the individuals make more than 50% test cases true, and thus the probability of satisfying the first formula of (4) declines sharply. Therefore we should conceive a novel strategy instead.

In our work we make an assumption that in the most choice structures, e.g., \( A \Rightarrow a[B] \Rightarrow b \), the probabilities of the conditions satisfying A and B trend to be divergent, no matter which is higher than the other. Actually, this assumption is reasonable and correspond with the practical conditions. We can take the Algorithm 1 for example. There are two if-structure statements in this algorithm:

- if ITC\( (E_{\mathcal{L}}^{\mathcal{P}}, In) \) then return In.
- if CTC\( (E_{\mathcal{L}}^{\mathcal{P}}, \{In_j\}_{j\in\{j_1, j_2, \ldots, j_k\}}) \) then return \( In_{j_1} | In_{j_2} | \ldots | In_{j_k} \).}

The Genetic Programming Algorithm with CTC is shown as Algorithm 1.

2.3. Complexity evaluation of testing CTC

The additional space for testing CTC is so small that it can almost be ignored. Since each truth table takes \( NC \) bits and maximum individual number is \( n \), the additional space will be \( CS = n \times NC \) bits. For example, if \( n = 10000 \) and \( NC = 256 = 2^8 \), \( CS \approx 10 \times 2^10 \times 2^8 \) bits = 320 KB.

However, the time complexity of testing CTC is enormous. If we intend to cover all of the possible combination forms, the maximum total comparison times increase from \( O(n \times NC) \) into \( C_T^n \) where

\[
C_T^n = O\left(\left(\frac{n}{1} + \frac{n}{2} + \cdots + \frac{n}{n}\right) \times NC\right)
\]

\[
= O\left(2^n \times NC\right)
\]

\[
= O\left(\frac{n^{NC}}{NB}\right)
\]

\[
\Rightarrow \text{if } n \gg NB
\]

Obviously this result is absolutely unacceptable. Thus we must redefine the maximum choice number \( NB \). Such CTC can be called \( NB \) CTC, and its maximum comparison times can be expressed as \( C_T^{NC} \) where

\[
C_T^{NC} = O\left(\left(\frac{n}{1} + \frac{n}{2} + \cdots + \frac{n}{NB}\right) \times NC\right)
\]

\[
= O\left(n^{NC} \times NC\right), \; \text{if } n \gg NB
\]

Especially, \( C_T^2 = O\left(n^2 \times NC\right) \), if \( n \gg 2 \).

Some simulation experiments are designed to show the time taken by testing 2-CTC. The experiment environments include a PC with Intel Core 2 CPU 2.33 GHz

\(^1\) If \( PB \gg NC\), the truth table becomes a list; however, we still call it truth table for compatibility.
and 2G RAM on the C++ programming platform. The individual number \( n \) is assigned to 1000, 5000, 10000 respectively; the fitness case numbers \( N_C \) is assigned to 16, 32, 64, 96, 128, 192, 256 respectively.

We contrive three situations, success, failure and bad. In our experiments the simulated individuals are generated according to the rules of these three situations.

**Success**  In the first situation, success. No. \( n/2 \) and \( n/2 + 1 \) individuals meet 2-CTC while any other pairs are failures.

**Failure**  In the failure situation, the last element of every individual’s truth table is “0”, thus each pair of individuals ready to test 2-CTC is a failure definitely.

**Bad**  In the bad situation, there are no pairs of individuals satisfying 2-CTC, but it must takes plenty of comparisons, much more than those in the “failure” situation. The bad situation is constructed as follows:

- Generate two groups of individuals, and both of the two groups have \( n/2 \) individuals.
- Individuals from the same group are the same; Individuals from the different groups fail to meet CTC but the mismatch should be judged as late as possible.

Let \( I_{i}\) and \( I_{j}\) be the individuals from the different groups in the bad situation. They must satisfy the fitness criterion, formally, \( F(I_{i}) + F(I_{j}) = N_C \). Moreover, as many tests as possible for the element criterion should be carried out. According to these principles, these two individuals can be constructed as shown in Fig. 1. When the CTC test for \( I_{i}\) and \( I_{j}\) occurs, since the fitness value of \( I_{i}\) is \( N_C - 1 \) while that of \( I_{j}\) is 1, the fitness criterion holds and the element criterion test starts. According the construction in Fig. 1, the test times of the element criterion is \( N_C - 1 \). 2-CTC tests for the individuals from the same type would be carried out the test for the fitness criterion only and report a failure. So the total test step number is

\[
N_C \times \frac{n}{2} \times \frac{n}{2} + 2 \times \left( \frac{n}{2} \right)^2 = \frac{(N_C + 1)n^2}{4} - \frac{n}{2}
\]

The experiment results are shown in Table 1. According to the experiment results we can conclude that:

(i) Generally, it would only take about 2 seconds or much less to carry out 2-CTC testing in the success and failure situations. It is an acceptable cost.

(ii) Even if these experiments are performed in the extremely hard situations, the results illustrate that the 2-CTC testing is feasible: the experiments with 5000

---

**Table 1**  Times spent by testing 2-CTC for three situations: Success, Failure and Bad. The experiment environments include a PC with Intel Core 2 CPU 2.33 GHz and 2G RAM in the C++ programming platform. The individual number \( n \) is assigned to 1000, 5000, 10000 respectively, and fitness case numbers \( N_C \) is assigned to 16, 32, 64, 96, 128, 192, 256 respectively.

<table>
<thead>
<tr>
<th>(a) Times spent by testing 2-CTC for the Success situation</th>
<th>Population number</th>
<th>Fitness case number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population number (Success)</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>1000</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>5000</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>10000</td>
<td>1.70</td>
<td>1.66</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(b) Times spent by testing 2-CTC for the Failure situation</th>
<th>Population number</th>
<th>Fitness case number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population number (Failure)</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>1000</td>
<td>0.02</td>
<td>0.03</td>
</tr>
<tr>
<td>5000</td>
<td>0.56</td>
<td>0.55</td>
</tr>
<tr>
<td>10000</td>
<td>2.28</td>
<td>2.20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(c) Times spent by testing 2-CTC for the Bad situation</th>
<th>Population number</th>
<th>Fitness case number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population number (Bad)</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>1000</td>
<td>0.05</td>
<td>0.06</td>
</tr>
<tr>
<td>5000</td>
<td>1.02</td>
<td>1.58</td>
</tr>
<tr>
<td>10000</td>
<td>4.06</td>
<td>6.31</td>
</tr>
</tbody>
</table>

---
individuals could be finished in 10 seconds; the experiments with 64 fitness cases can be terminated almost in 10 seconds, too.

We should denote that the bad situation is impossible in the real world, and actually the time spent in 2-CTC testing would be quite closer to the former two situations.

2.4. Optimizing method

After our approach executes, a set of individuals can be generated. Let these individuals be $In_{j_1}, In_{j_2}, \ldots, In_{j_{|J|}}$. The last issue is to optimize the individual(s) evolved by our approach.

Each individual is composed of a series of basic functions. Since each basic function can induce a state transformation from a state to another, each individual can be regarded as a series of state sequence from the initial state $r$ to the final state $s$, named a trace. Unavoidably, there may be some cycles in this trace. E,g., an individual is comprised of two functions $increase(x)$ and $decrease(x)$, which means to add 1 to $x$ and subtract 1 from $x$ respectively. Initially $x = 0$, and after the individual acts on $x$ step by step the trace may be $[r(x = 0), s_1(x = 1), s_2(x = 0)]$. Actually $r = s_2$, thus we can merge these two same states as a single one $r(x = 0)$, and a cycle $r \rightarrow s_1 \rightarrow r$ is formed. Obviously this individual whose trace is a cycle path actually do nothing.

Based on this observation we can decide that our optimizing strategy is to remove the acyclic paths in the traces of the generated individuals. The method is illustrated in Fig. 2. Given an individual, the state diagram expressing its trace is easy to obtain, called a primitive diagram. As the Fig. 2(a) shown, some states exist more than one times, such as $s_1$ and $s_4$. According to our optimizing method we should merge these same states in the primitive diagram, and the diagram then becomes a directed cyclic graph, as shown in Fig. 2(b). Obviously, the cycles have no contributions, thus they should be removed, and the optimized path is shown in Fig. 2(c).

After the optimizing steps, the individuals $In_{j_1}, In_{j_2}, \ldots, In_{j_{|J|}}$ are optimized as $In'_{j_1}, In'_{j_2}, \ldots, In'_{j_{|J|}}$ respectively. If these individuals share a common start $P_S$ and a common end $P_E$, and their middle parts are expressed as $P_1, P_2, \ldots, P_{|J|}$ respectively, the programs can be optimized further as $P_S ; P_1[|P_2|] \cdots [|P_{|J|}|] ; P_E$.

2.5. Generating the condition formulae

Now we must generate the conditional criterions for the individuals satisfying CTC to decide which candidate action should execute at the given state. This task is actually a classification problem: the features $F$ are the terminals, and the training set are the test cases labeled $true$ or $false$. Once these discriminant functions are generated, the conditional criterions are constructed. E,g., when 2-CTC holds, two individuals satisfying Eq. (2) are generated. Let $In_{j_1}$ and $In_{j_2}$ be the individuals. In this choice structure, the discriminant function $f(F)$ must satisfy the following formulae.

$$\forall t \bullet (t \in C(In_{j_1}) \Rightarrow f(t) > 0)$$

$$\forall t \bullet (t \in C(In_{j_2}) \Rightarrow f(t) < 0)$$

(8)

Which classifier is best? This is dependent on the specified problems. Generally speaking, the linear classifiers are powerful enough for most of the problems.

3. Evolving intelligent_swap for the sort problem

The evolution of the intelligent_swap function is a classical evolution problem. Since it is simple enough but composed of all of the necessary issues for the evolution of the choice structures, it can be actually regarded as a baseline problem.

The problem definitions of the intelligent_swap evolution problem are illustrated in Table 2. The terminals is a integer sequence $s : \mathbb{N} \rightarrow \mathbb{Z}$, that is, a series of numbers with a unique integer index, e.g., $s(1) = 5, s(2) = 7, s(3) = 2$. The functions are a set of basic operations based on sets, e.g., $length(s)$ can return the length of $s$, $sum(s)$ can evaluate the sum of the integers in $s$, etc. The termination

---

2 If ITC holds, $|J| = 1$ and the result is only one individual $In_{j_1}$.
Table 2
Sort problem definitions for our GP approach. The terminal is an integer set; The functions are a set of basic operations based on sets; The termination criterions are composed of ITC and 2-CTC, which means only \( N_B = 2 \) will be considered; The fitness case method is adopted as the fitness function.

<table>
<thead>
<tr>
<th>GP requirements</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminals</td>
<td>( s : \mathbb{N} \rightarrow \mathbb{Z} )</td>
</tr>
<tr>
<td>Functions</td>
<td>( l \leftarrow \text{length}(s) )</td>
</tr>
<tr>
<td></td>
<td>( e \leftarrow \text{get}(s, \text{index}) )</td>
</tr>
<tr>
<td></td>
<td>( s \leftarrow \text{sum}(s) )</td>
</tr>
<tr>
<td></td>
<td>( a \leftarrow \text{average}(s) )</td>
</tr>
<tr>
<td></td>
<td>( \text{delete}(s, i) )</td>
</tr>
<tr>
<td></td>
<td>( \text{append}(s, a) )</td>
</tr>
<tr>
<td></td>
<td>( \text{insert}(s, i, a) )</td>
</tr>
<tr>
<td></td>
<td>( \text{swap}(s, i, j) )</td>
</tr>
<tr>
<td></td>
<td>( \text{update}(s, i, a) )</td>
</tr>
</tbody>
</table>

Termination criterion

<table>
<thead>
<tr>
<th>Main parameters</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITC &amp; 2-CTC</td>
<td>( n = 200 ), ( G = 10 )</td>
</tr>
<tr>
<td></td>
<td>( N_C = 16 ), ( N_B = 2 )</td>
</tr>
<tr>
<td>Fitness function</td>
<td>( \mathcal{F}^*(\ln) =</td>
</tr>
</tbody>
</table>

criterions are composed of ITC and 2-CTC. The fitness case method is adopted as the fitness function.

After our algorithm executes, 2-CTC satisfies and a pair of individuals return, they are \( I_{nj_1} \equiv e \leftarrow \text{get}(s, \text{index}); s \leftarrow \text{sum}(s); \text{swap}(s, i, j); a \leftarrow \text{average}(s) \) and \( I_{nj_2} \equiv l \leftarrow \text{length}(s); e \leftarrow \text{get}(s, \text{index}) \).

After the optimizing phase, these two individuals can be optimized as \( \text{swap}(s, i, j) \) and \( \text{doNothing} \) respectively.

According to the linear classification algorithm, the discriminant function is generated as \( f = s(i) - s(j) \). Thus the intelligent_swap can be evolved as follows.

\[
s(i) - s(j) < 0 \Rightarrow \text{swap}(s, i, j) \tag{1}
\]

\[
\neg(s(i) - s(j) < 0) \Rightarrow \text{doNothing} \tag{9}
\]

Actually, it can be translated into a program shown as follows.

```plaintext
if s(i) - s(j) < 0 then
    swap(s, i, j)
else
    doNothing
end
```

4. Conclusion

In this paper we propose a variant of GP to evolve the choice structures automatically by the conception of the Combination Termination Criterion (CTC). Hence, the evolutionary ability of GP is much improved.

Our method firstly evolve several individuals satisfying CTC, which are exactly the candidate actions. The generation of the conditional criterions can be considered as a classification problem. Via this strategy the evolution process of the conditional criterions and the matching issue between the optional actions and the conditional criterions can be resolved simultaneously. According to the theoretical analysis and the experiment results, the additional time for the evolution of the choice structures can be restricted within an acceptable range by predesigning a proper value of the parameter \( N_B \) before the GP is carried out.

Acknowledgements

The authors would like to thank the editors and reviewers for their helpful comments. This research is supported by the Natural Science Fund of China No. 60970047 and the Key Science-Technology Project of Shandong Province of China No. 2008GG10001026.

References