Genetic Programming with Simple Loops

QI Yuesheng (齐越胜), WANG Baozhong (王保中), KANG Lishan (康立山)

State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, P.R. China

E-mail: qiys@rjgc.whu.edu.cn

Received November 9, 1997; revised March 10, 1999.

Abstract A kind of loop function Loop$N$ in Genetic Programming (GP) is proposed. Different from other forms of loop function, such as While-Do and Repeat-Until, Loop$N$ takes only one argument as its loop body and makes its loop body simply run N times, so infinite loops will never happen. The problem of how to avoid too many layers of loops in Genetic Programming is also solved. The advantage of LoopN in GP is shown by the computational results in solving the mower problem.

Keywords loop function, genetic programming, mower problem

0 Introduction

Genetic Programming is first introduced by Koza[1] in 1992 and becomes an important branch of Evolutionary Computation. This paper proposes a new kind of loop function Loop$N$ in GP to make iterations easy to use and still effective. By introducing Loop$N$ in GP, we have obtained quite good results in solving the mower problem which appears first in [2]. Compared with those that don't use any loop function, the results are quite satisfactory. The evolved programs are competitive to human written ones.

This article includes four Sections. Section 1 is a brief introduction to GP. Section 2 discusses the loop functions fully, focusing on their representations and the problems they are likely to cause and the methods other researchers used to deal with. Loop$N$ is introduced here and comparisons with other loop functions are also given. Section 3 presents the computational results of the mower problem and Section 4 draws the conclusions and makes a plan for future work.

1 Genetic Programming

There is a famous formula in computer science: program = data structure + algorithm. When we try to solve a problem with a computer, we need not only the data and its structure but also a definite algorithm. Sometimes it is not easy for people to prepare an efficient algorithm for some problems. Can we manage to make the computer find a good algorithm all by itself? GP suggests a way to do this.

GP represents an algorithm as a tree. Using evolutionary method to make the computer find a satisfactory tree, e.g. a good algorithm, is just what Genetic Programming does. GP always takes the following steps:

1. Determine the function set Fset. Fset is also the set of operators. All the nodes in a tree, except those leaf nodes, are elements of Fset.

2. Determine the terminal set Tset. All the leaf nodes are elements of Tset. To guarantee the tree's validity, suppose that any terminal element and any return value of any function can serve as argument of any function of the Fset. This is called the closure requirement of operations in GP.

The following steps are similar to Genetic Algorithm[3].

This paper is supported by the National Natural Science Foundation (No.69635030) and National '863' Hi-Tech Programme of China

3. Determine the evolution parameters, including maximum generation, population size of every generation, copy rate and crossover rate (mutation operation is seldom used here). In GP maximum length of a tree is needed to control the complexity of the tree. The fitness function must be defined before going ahead.

4. Generate the initial generation randomly (Generation:=0).

5. Evaluate every tree by the fitness function. If one tree satisfies the need of the problem, or Generation equals to the maximum generation, stop evolution; otherwise go to next.

6. Generate a new generation by a selection strategy and genetic operations. The selection strategy should make the better individual have more chance to be selected as parent. Copy operation adds the selected individual (parent) from the current generation into the new generation. Crossover operation works a little more complicated. Select two individuals from the current generation as parents. Randomly select one node in one parent and get subtree1, randomly select one node in another and get subtree2, exchange subtree1 and subtree2 and get two new individuals.

Some special rules are always used to make all offspring legal. For example, when function division takes 0 as its second argument, the divisor, just define 1 as its return value. Koza[1] calls this protected division. However, Montana[4] takes a different but natural way. He marks such individuals illegal and discards them and produces new ones to replace them. Anyhow, before running and being evaluated, both offsprings are regarded legal. Add these two offsprings into the new generation. Repeat copy and crossover operations until the new generation has sufficient individuals.

7. Replace the current generation with the new generation. Increase Generation by 1 and go to 5.

2 Loop Function

It has been proved that all programs can be constructed by three simple structures: Sequence, Branch and Loop structures. Generally speaking, a tree in GP often executes every node sequentially with depth preference. ADF (Automatically Defined Function) in Koza[2] makes use of the technique of subroutine in programming. IF_ELSE brings branch structure to GP. Although loop is so important that it should be used to develop good algorithm, loop will cause some problems in GP:

1. How to represent a loop? A program with a loop in GP is likely to be represented as a graph with cycles.

2. How to control layers of loop? Too many layers of loops will surely cost much time and the efficiency must be very low.

3. What about infinite loops?

Although these problems restricted the use of loop in GP, there are a number of papers where programs evolved contain loops. They often use DO_UNTIL or DO_TIMES or something like that. These functions have two arguments, one is to control loop, and the other is the loop body. A number of techniques to address the problems of infinite loops are used. Kinnear[5] applies both a limit on the total number of iterations and a limit for each loop function. Cramer[6] aborts any program that fails to stop within a specified time. Teller[7] allows fitness testing to continue whilst it continues to do something interesting (which may increase its fitness) but imposes a maximum waiting time between interesting events. Nordin[8] limits the total number of executing times of every loop body. The techniques above solve problem 1 (representation) completely, solve problem 3 (infinite loops) unnaturally, and leave problem 2 (layer control) almost open.

We designed a simple loop function Loop$N$. Though it is not as powerful as DO_UNTIL or DO_TIMES, Loop$N$ is efficient to particular problems. Loop$N$ takes one argument as its loop body. N is a positive integer given beforehand by experience and intuition and is the number of executing times of the loop body. Thus infinite loops will never take place. To simplify the control of loop layers, nested loop is forbidden in our following experiments. Thus we get around all the three problems above. We take such actions to avoid nested loop.

In the initial generation, if a node is a loop function Loop$N$, we don't select Loop$N$ again as part of its loop body. Later, when crossover operation works, if one subtree contains Loop$N$, we mark it as type-A; if one subtree is part of a loop body, we mark it as type-B. Before exchanging type-A subtree and type-B subtree, we delete the Loop$N$ node in type-A subtree. See Fig.1. The meaning of every node is explained in the following experiment. More subtle classification allows two or more layers of loop.

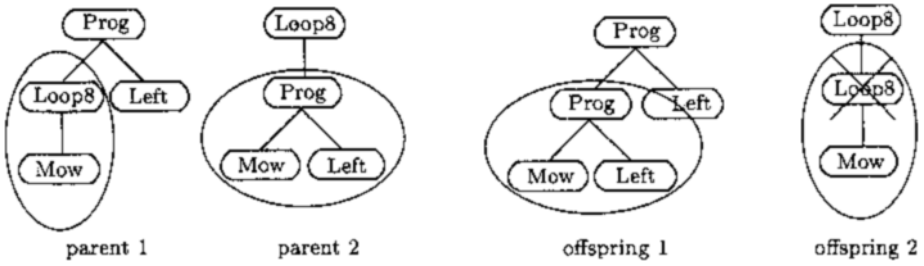

Fig.1. Nested loop control.

3 Example of Application

Mower problem appears in [2]. There is a mower on an 8×8 lawn. The mower is at (0,0) at beginning, facing north. The mower has two basic actions: Left and Mow. Left makes the mower turn left 90°, without changing its position. Mow makes the mower move forward one square and mow the grass in the square where it just arrived if there is still any grass in the square, but the mower's direction remains unchanged. We are asked to control the mower so that it can mow all the grass in the whole lawn. Passing one square two or more times is permitted. Suppose the lawn is toroid. As shown in Fig.2, when the mower moves north from square $A(3,7)$, it arrives at square $B(3,0)$. In our experiments we define Fset and Tset as follows:

Function set Fset={Loop8, Prog};

Terminal set Tset={Left, Mow, Right}.

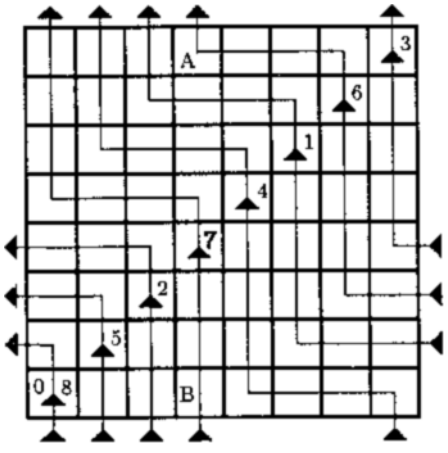

Fig.2. The mower route of one evolved program.

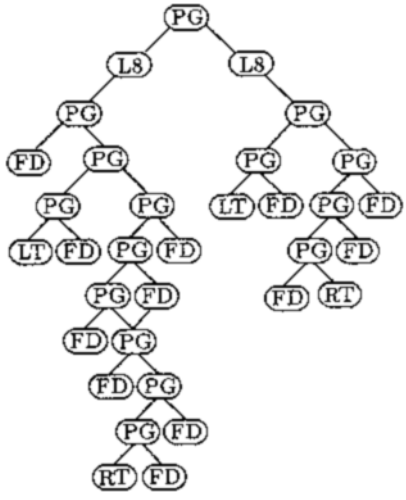

Fig.3. One of the best programs.

Loop8 is a function with one argument and executes the argument 8 times; Prog is a function with two arguments and executes the two arguments sequentially; Left and Mow have been explained above, Right is added in our experiments and makes the mower turn

right 90°. Other parameters are:

MaxGeneration=40;

PopulationSize=20;

MaxLength=250;

copy rate Crate=0.1;

crossover rate Xrate=0.9;

fitness = number of squares the mower passes.

We made six experiments, among which five got the desired programs whose fitness are 64, and one failed. The reason of failure was that all trees lost Left and Right nodes from generation 6.

Fig.3 is the structure of the best solution of the fifth experiment. PG, L8, LT, RT and FD are Prog, Loop8, Left, Right and Mow respectively. This program has two main branches. It is interesting that the first branch can complete the mowing task. Fig.2 shows the route of the mower. The digits show the mower positions when the first loop body completes executing one time.

The route shows that the evolved program is almost as good as human written ones. To demonstrate the efficiency of Loop$N$ more convincingly, we did other experiments. They were divided into two groups. Group 1 used Loop$N$, group 2 didn't use any kind of loop functions. The scale of the lawn varied from 4×4 to 10×10. For any scale of lawn, the experiment repeated 20 times in each group. The genetic parameters are different from those of the above six experiments. They are:

MaxGeneration=100;

PopulationSize=50;

MaxLength=500;

In group 1, for $M \times M$ lawn, Loop8 became Loop$M$. The results are shown in Table 1 and Table 2.

Table 1. Results of Group 1 Which Used Loop$N$

lawn scale	success rate	valid generation	length	time (s)
4×4	20/20	3.35	22.10	1
5×5	20/20	4.75	22.35	1
6×6	20/20	5.70	27.75	1
7×7	20/20	6.75	35.70	1
8×8	20/20	7.35	39.80	1
9×9	20/20	8.35	34.95	1
10×10	20/20	11.15	49.35	1
20×20	18/20	16.33	123.38	1

Table 2. Results of Group 2 Which didn't Use Loop

lawn scale	success times	valid generation	length	time (s)
4×4	20/20	11.40	105.70	1
5×5	20/20	16.30	178.15	1
6×6	20/20	19.90	317.70	2
7×7	20/20	23.75	381.70	3
8×8	19/20	42.58	451.74	6
9×9	6/20	47.33	467.67	14
10×10	0/20	–	–	15

In the two tables, column 3 is the average of valid generation for all the successful evolutions; column 4 is the average length of the best individuals for all the successful evolutions; column 5 is the total time for all the 20 evolutions.

The data of success rate, CPU time and complexity of trees in group 1 are all much better than those in group 2. Even 20×20 scale can be solved quite easily in group 1 while 10×10 scale is unsolvable in group 2.

4 Conclusion

Loop plays a very important role in programming. GP should try to make good use of loop functions in evolving. DO_UNTIL is likely to result in infinite loops while DO_TIMES is likely to bring more than one kind of data types and makes GP become Strongly Typed Genetic Programming (STGP)[4]. STGP is much more complicated than standard GP which is used in this paper. Loop$N$ introduced in this paper is the simplest kind of loop function, yet it performs quite well in solving some problems.

In general, we forbid nested loop but the number of Loop$N$ is unlimited (limited only by the maximum length of the tree). We think that loop does not do much harm if it is not nested. As for the value of N in Loop$N$ in other applications, it should be set according to the problem scale. More work needs to be done on the loop subject.

1. How to use the loop control variable in the loop body without causing possible infinite loops? Sometimes the loop control variable must be available in the loop body.

2. How about putting loop function, ADF and IF_ELSE all in the function set of GP? We believe this is powerful for some complicated problems but the search space will swell dramatically. High-efficiency evolutionary strategy and genetic operations must be designed to deal with the challenging problems.

References

[1] Koza J R. Genetic Programming I, On the Programming of Computers by means of Natural Selection. Cambridge, MA: MIT Press, 1992.

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

[3] Liu Yong, Kang Lishan, Chen Yuping. Nonnumeric Parallel Algorithm (Book II). Genetic Algorithm, Science Publishing House, 1995.

[4] Montana D J. Strongly Typed Genetic Programming. *Evolutionary Computation,* 1995, 3(2).

[5] Kenneth E. Kinnear Jr. Evolving a sort: Lessons in genetic programming. In *Proceedings of the 1993 International Conference on Neural Networks,* Volume 2, San Francisco, USA, 1993, IEEE Press.

[6] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of an International Conference on Genetic Algorithms and the Applications,* Grefenstette John J, (ed.), Carnegie-Mellon University, Pittsburgh, PA, USA, July 24-26 1985, pp.183–187.

[7] Astro Teller, Genetic programming, indexed memory, the halting problem, and other curiosities. In *Proceedings of the 7th Annual Florida Artificial Intelligence Research Symposium,* pp.270–274, Pensacola, Florida, USA, May 1994, IEEE Press.

[8] Peter Nordin, Wolfgang Banzhaf. Evolving turing-complete programs for a register machine with self-modifying code. In *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95),* Eshelman E (ed.), pp.318–325, Pittsburgh, PA, USA, July 15-19, 1995.

QI Yuesheng was born in 1966. He is now a Ph.D. candidate of computer science. His research interests are evolutionary computation.