How to reduce the memory requirement in Genetic Programming

Yodthong Rodkaew and Prabhas Chongstitvatana
Department of Computer Engineering, Faculty of Engineering,
Chulalongkorn University,
Phayathai road, Bangkok 10330, Thailand
yod@orange.cp.eng.chula.ac.th, prabhas@chula.ac.th

Abstract

Genetic Programming (GP) is a search and optimization algorithm inspired by natural evolution. GP performs search in a population, a set of individuals which represents points in a search space. An individual has a tree structure. A large problem demands memory for population with a conventional implementation. As opposed to implementing a tree as a collection of linked nodes, this work proposes an implementation of a tree as a vector. The experiment which is performed on a realistic size problem shows that the running time for genetic programming with vectors is similar to a conventional implementation but vectors use 25 times less memory. This enable us to perform the experiment on a much larger size of problem.

1. Introduction

Genetic Programming (GP) [4] is a variation of Genetic Algorithm (GA) [1] which uses a different representation of a solution. GP is a search and optimization algorithm inspired by natural evolution. GP performs search in a population, a set of individuals which represents points in the search space. At each generation, a sequence of genetic operations called reproduction, crossover and mutation transforms the existing individuals into a new set of solutions. The solution quality is evaluated in terms of fitness in which fitness function are defined for a particular problem. The individuals are probabilistically selected to the next generation proportional to their fitness. GP has been applied successfully to a wide range of problems [4,6,7].

We are motivated by the fact that our GP experiments recently have strained our computing resource such that we are unable to try a larger size of problem. Our work in [9] used as much as 30 Mbytes of memory during the run. As we are interested in investigating GP with a larger size of problem, we must find some way to reduce the requirement on memory.

In GP, an individual has a tree structure. In a typical implementation, a variable size structure is stored as a collection of fixed size objects linked together. The size of each object is determined by the type of data to be stored at each node in a tree. Each node usually contains the type of the node and links to other nodes. The object must be able to contain pointers to other nodes. As opposed to implementing a tree as a collection of linked nodes, this work proposes to implement a tree as a vector. When the arity of each node is known, traversing a tree in either depth-first or breath-first order will produce a unique ordering of all nodes. The links can be dispensed with, hence saving the storage.

2. Data structure

We are going to show an example to illustrate our scheme. Given a tree as in Figure 1 which represents a robot program from the work in [9]. Let assume that each fixed size object has the size that can contain a pointer, 4 bytes in a typical machine. Each non-terminal noted stores the node type and links to its children. A typical implementation will store this tree in 23 blocks, each block is 4 bytes, therefore it requires 92 bytes (Figure 2).

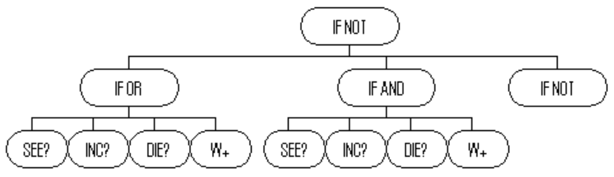

Figure 1 An example of an individual

Memory Array				
IF NOT	<addr>	<addr>	<addr>	IF OR
<addr>	<addr>	<addr>	<addr>	SEE?
INC?	DIE?	W+	IF AND	<addr>
<addr>	<addr>	<addr>	S+	E+
SEE?	W-	OUT?		

23 blocks use, 4 bytes per block, total 92 bytes

Figure 2 Nodes and links implementation of the tree

IF NOT	IF OR	SEE?	INC?	DIE?	W+	IF AND	S+	E+	SEE?	W-	OUT ?

1 byte per node, total 12 bytes

Figure 3 Vector implementation of the tree

Storing this tree as a vector required 12 nodes (Figure 3). No links are necessary. Each node needs only to store the node type. Let assume one byte per node which is sufficient to represent 256 types of terminal symbols for the terminal set in most GP problems. This vector requires only 12 bytes. Comparing this implementation to the previous one, the vector reduces the storage more than 7.5 times.

3. Operations on the vector

The main operations to be performed on an individual in GP are genetic operations and fitness evaluation. Genetic operations are : reproduction, crossover and mutation which manipulate the structure. Fitness evaluation usually requires traversing the tree.

For the implementation with nodes and links, genetic operations can be achieved by updating the pointers. Traversing nodes and links is also straightforward. However, the copying operation requires allocation and deallocation of fixed size objects from the pool of storage hence requires some form of memory management.

For the vector implementation, the operation on the vector will be similar to string operations as nodes are stored in a linear structure. Because the arity of each node type is known, any subtree can be located in a vector and can be manipulated based on substring operations : copy, concatenation, deletion and insertion.

The pseudo code for genetic operators are described as follows :

```
Crossover ( parent1, parent2)  produces child1, child2
1. randomly select point1, point2 from parent1, parent2
2. separate each parent into 3 parts : head, mid, tail
     where head is the beginning to the crosspoint,   mid is from the
     crosspoint and contains the whole  subtree, tail  is the rest
3. child1 = head of parent1 + mid of parent2 + tail of parent1
4. child2 = head of parent2 + mid of parent1 + tail of parent2
     where  + is concatenate operator

Mutate ( parent ) produces a child
1. copy parent to child (which will be mutate)
2. select a point in child to be a mutate point
3. delete the whole subtree at the mutate point
4. generate a new tree insert the new tree at the mutate point
```

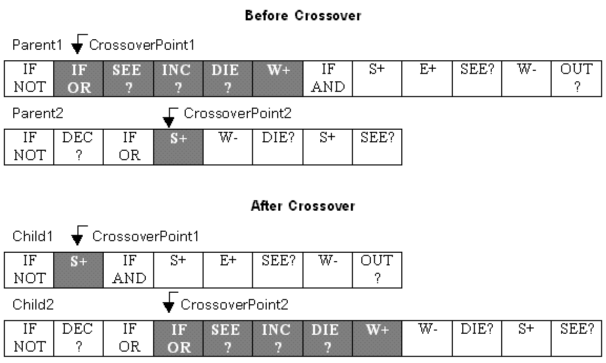

Figure 4 an example of crossover operation.

For tree traversal in a vector, skipping a subtree requires traversing the whole subtree, for example the "if-then-else" function. This can take considerable time, as in some GP application a tree can be evaluated repeatedly many times, such as in the case of robot simulation. An auxiliary array of "offset" can be constructed such that skipping any subtree can be done by looking up the "offset". Constructing this auxiliary array will take additional time but the array itself needs not to be stored. Each individual is usually evaluated only once in a generation hence the auxiliary array can be constructed prior to the fitness evaluation and afterward discarded.

4. Experiment

A study has been made based on the work [8] in which the original data structure is nodes and links. We implement the population using vectors and compare with a conventional implementation in term of running time and memory requirement. The work in [8] studied the problem of controlling a manipulator to reach a target. GP is used to generate robot programs to control a robot manipulator to reach a target while avoiding obstacles (Figure 5).

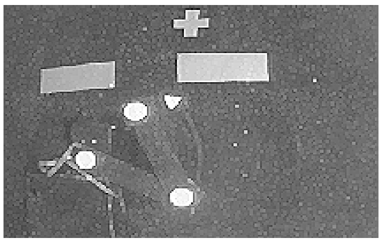

Figure 5 The problem of controlling a manipulator to reach a target.

The experiment is performed with these parameters : individual 400, reproduction 40, crossover 160, mutation (2 types) 200, maximum generation 10. The initial size of each individual is average 80 nodes. We run the experiment 100 times with a fixed random seed and analyse the running time. The result is shown in Table 1.

Function Call	Call	Vector (ms/Call)	Linked (ms/Call)
Evaluate individual fitness	364,000	0.75	0.64
Crossover	900	2.09	21.31
Addition	900	1.23	12.05
Extension	900	1.31	12.83
Initialization	100	46.20	54.36
Copy	360,000	0.00	0.06

	Vector (s)	Linked (s)
Total program execution time	285.51	285.73

Table1 The running time of both implementations

From the table 1 the genetic operation on the vector implementation is faster but the fitness evaluation is slower than the conventional implementation. The genetic operation is faster because in a vector any node can be directly access as opposed to following the links. For fitness evaluation, a vector requires computing the "offset" table, which explains why it is slower. However, total running time for both implementations are not significantly different.

How much does the vector saved memory compared to the nodes and links? For one population, assume 400 individuals, each individual contains in average 200 nodes. The conventional implementation uses more than 2Mbytes (including the copy space for offsprings), but the vector implementation requires only 80K. This is the saving of 25 times for this experiment.

5. Related work

Koza [4] used the LISP program as an individual. Therefore each individual has the data structure of LISP. If the arity of each node is known there are many encoding scheme that can mapped node location without storing the link, for example, [2,3] show a scheme for a binary tree. [5] suggests many implementation of data structure for GP. In the recent book, [10] mentions using vectors in their implementation however, our implementation of "offset" table is unique.

6. Conclusion

This work suggests using vectors to implement individuals in GP to reduce the memory requirement. The operation on the vector can be faster than the operation on a conventional structure. The experiment which is performed on a realistic size problem shows that the vector form running time performance is similar to the nodes and links form but it can save up to 25 times the memory. This enable us to perform the experiment on a much larger size of problem.

7. Acknowlegdement

This work is supported by the Thailand Research Fund contracted no. RSA4080023.

References

[1] Holland, J., "Adaptation in Natural and Artificial Systems", The University of Michigan Press, Ann Arbor, Michigan, 1975.

[2] Sohi, G., Davidson, E. and Patel, J., "An efficient LISP-execution architecture with a new representation for list structures", Proc. 12th Sym. on computer architecture, Boston, 1985, pp. 91-99.

[3] Kogge, P., "The architecture of symbolic computers", McGraw -Hill, 1991, p. 208.

[4] Koza, J., "Genetic Programming: On the Programming of Computers by mean of Natural Selection", Cambridge, MIT Press, 1992.

[5] Keith, M. and Martin, C., "Genetic Programming in C++ : Implementation issues", in Kinnear, K., ed., Advances in genetic programming, MIT Press, 1994, pp. 285-310.

[6] Koza, J., "Genetic Programming II: Automatic Discovery of Reuseable Programs", MIT Press, 1996.

[7] Banzhaf, W., Nordin, P., Keller, R. and Francone, F., "Genetic Programming: An Introduction On the Automatic Evolution of Computer Programs and Its Applications", Morgan Kaufmann Publishers, 1998.

[8] Jassadapakorn, C. and Chongstitvatana, P., "Reduction of Computational Effort in Genetic Programming by Subroutines", Proc. of National Computer Science and Engineering Conference, Bangkok, Thailand, 1998.

[9] Nopsuwanchai, R. and Chongstitvatana, P., "Analysis of Robustness of Robot Programs Generated by Genetic Programming", Proc. of 1st Asian Symposium on Industrial Automation and Robotics, Bangkok, 1999, pp. 211-215.

[10] Koza, J., Bennett, F., Andre, D. and Keane, M., "Genetic Programming III: Dawinian Invention and Problem Solving", MIT Press, 1999, pp. 1041- 1045.