# An Application of Genetic Programming to the 4-Op Problem using Map-Trees

Tevfik Aytekin, E. Erkan Korkmaz and H. Altay Güvenir

Bilkent University, Computer Engineering and Information Science Department, Ankara 06533 TURKEY

**Abstract.** In Genetic programming (GP) applications the programs are expressed as parse trees. A node of a parse tree is an element either from the function-set or terminal-set, and an element of a terminal set can be used in a parse tree more than once. However, when we attempt to use the elements in the terminal set at most once, we encounter problems in creating the initial random population and in crossover and mutation operations. 4-Op problem is an example for such a situation. We developed a technique called *map-trees* to overcome these anomalies. Experimental results on 4-Op using map-trees are presented.

## 1 Introduction

Genetic algorithms, by combining the survival of the fittest among string structures with a randomized genetic information exchange, try to form a search algorithm similar to the evolution process in nature. In every generation, a new set of strings is created using bits and information coming from the fittest of the previous generations. See [4] and [3] for details on GAs.

Genetic programming (GP) on the other hand employs programs instead of strings [5]. Both genetic methods differ from most of the search techniques in that they simultaneously involve a parallel search involving a large number of points. In GP this is done by the random creation of a population of individuals represented by programs which are the candidate solutions to the problem. These programs are expressed in GP as parse trees. The individuals in the population then go through a process of evolution.

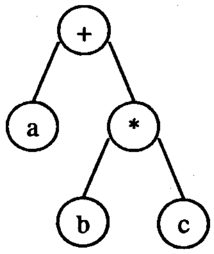

**Fig. 1.** A parse tree example.

Thus for example, a simple program that computes "$a + b * c$" would be expressed as in Fig.1, or to be precise as suitable data structures linked together to achieve this effect.

The programs in the population are composed of elements from a *function-set* and a *terminal-set*, which are typically fixed sets of symbols selected to be appropriate for the solution of problems in the domain of interest. The initial population consisting of individual programs is randomly created after determining these two sets. In GP the genetic information exchange is done by taking randomly selected subtrees in the individual programs and exchanging them. This is the recombination operation which is referred to as *crossover* because of the way that genetic material crosses over from one chromosome to another. Because of the closure property of the functions and terminals, this genetic crossover operation always produces syntactically legal parse trees as offspring regardless of the selection of parents or crossover points.

The crossover operation takes place in an environment where the selection of who gets to mate is a function of the *fitness* of the individual, i.e. how good the individual is at competing in its environment. Some GP techniques use a simple function of the fitness measure to select individuals (probabilistically) to undergo genetic operations such as crossover or *reproduction* (the propagation of genetic material unaltered). This is called *fitness proportionate selection. Mutation* also plays a role in this process, though it is not the dominant role that is popularly believed to be the process of evaluation, i.e. random mutation and survival of the fittest. It cannot be stressed too strongly that the GP is not a random search for a solution to a problem. The GP uses stochastic processes, but the result is distinctly better than random.

The GP executes the following cycle: Evaluate the fitness of all individuals in the population; Create a new population by performing operations such as crossover, fitness proportionate reproduction and mutation on the individuals based on the fitness; Discard the old population and iterate using the new population. One iteration of this loop is referred to as a *generation.*

As a last remark, we will state an important point that was pointed out by Koza [5]:

> Seemingly different problems for a variety of fields can be reformulated as problems of program induction (requiring the discovery of a computer program that produces some desired output when presented with particular inputs), GP paradigm provides a way to search the space of possible computer programs for an individual program that is highly fit to solve the problems of program induction.

The reason behind reformulating various problems as problems of program induction is because computer programs have the flexibility and complexity needed to express the solutions to a wide variety of problems and there is a way to solve the problem of program induction which is the GP paradigm.

Usually in GP applications there is no restriction on the number of function-set and terminal-set elements used. However in some applications there may be a restriction on the number of occurrences for each element of these sets. In this case standard crossover and mutation operation will lead to illegal parse trees. In this paper we present such an application called 4-Op where the function-set is $\{+, -, /, *\}$ and the terminal-set consists of six integers.

The next section gives a description of the 4-Op problem. Section three presents our formalism called *map-trees* which helps to redefine the crossover and mutation operations to guarantee that off-springs are legal parse trees. The fourth section makes an empirical study of our new technique and the last section concludes with an overall evaluation.

## 2 Description of the 4-Op Problem

4-Op is a well known TV-game where the players try to find an arithmetical expression, involving six integers, whose value is closest to a given target value. The expression may contain any number of the four arithmetical operations "$+, -, *, /$". The first four of the input number set are between one and ten and the last two are chosen from the set {25, 50, 75, 100}. The target value is between 100 and 999. An important restriction is that the players can use each element of the input set at most once.

For example, let the input number set be {2, 3, 5, 8, 25, 100} and the target value be 467. The expression $(5 * 100) - (25 + 8) = 467$ is one of the possible answers to the question. However it is not always possible to find an exact solution. A player can get points if no other player has a closer expression.

In genetic programming applications usually there is no restriction on how many times each element of the terminal set can be used. However in our problem we can use each element at most once. So this brings a restriction to parse trees formed and to the operations on the parse trees like crossover and mutation. We can not perform crossover and mutation operations at an arbitrary point in the parse trees, since this may cause repetition of a terminal-set element in the parse tree. Let us illustrate these anomalies with an example. Consider the two parse trees named P1 and P2 in Fig.2a. The tree P1 stands for the expression (* 5 (+ (* 3 25) 50)) and P2 stands for the expression (+ 3 (* (+ 5 (− 8 2)) 25)) in prefix notation. The numbers near each node of the tree represent the crossover points. Now let us perform a crossover at points 4 on P1 and 3 on P2. The crossover fragments are shown in Fig.2a inside dashed lines. After the crossover operation, we get two offsprings as shown in Fig.2b.

Also if we consider a mutation at point 7 on P1 in Fig.3a and if we generate the mutation fragment as in Fig.3b, we get the off-spring shown in Fig.3c after the mutation operation.

Now, let us examine the trees we get after mutation and crossover. In all of them at least one element of the terminal set is used more than once. In O1 the element 5, in O2 the element 3 and in new-P1 the element 3 and 5 are used twice. Hence, the new form of expressions we have are invalid and cannot be used as solutions to our initial problem (Since there is a restriction that we can use each element of the terminal-set at most once). However this is not the case

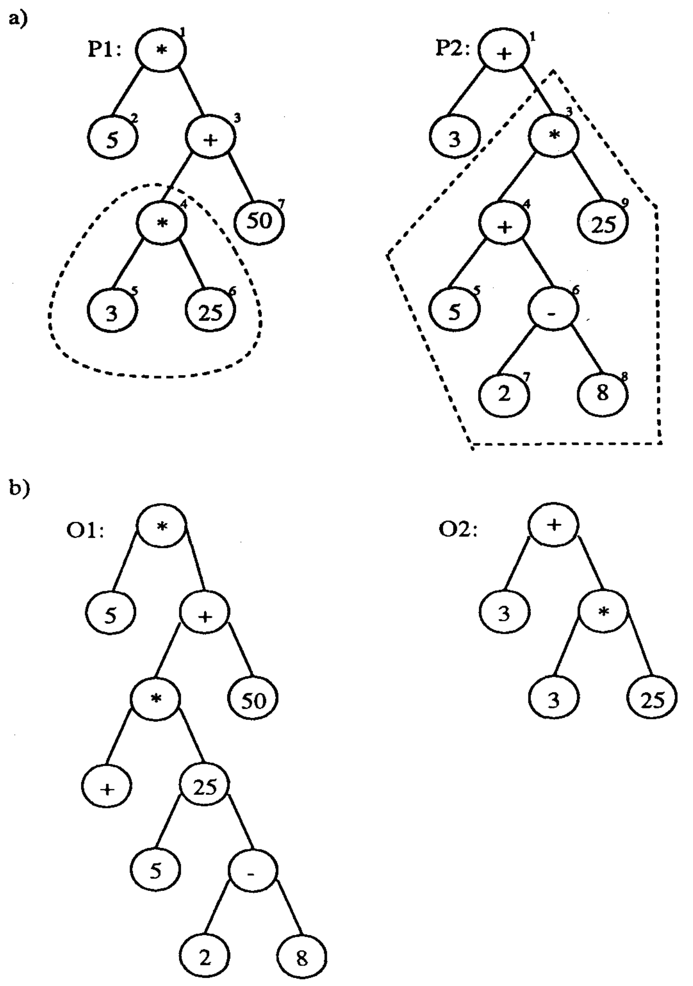

**Fig. 2.** a) Parents in crossover, where crossover fragments are enclosed in dashed lines. b) Offsprings after crossover. Note that both offsprings contain an element used twice.

for all crossover points and for all mutations. For instance, a crossover operation at points 2 on P1 and 4 on P2 will not violate our problem constraints.

The trees obtained after this crossover can be seen in Fig.4, and these are valid parse trees since each element of the terminal set appears at most once. Similarly we can find mutation points which generate valid parse trees.

The main problem here is to develop the appropriate data structures and techniques to overcome the illustrated anomalies. The data structures and techniques we used are not specific to our problem, but can be considered as a general approach to solving problems by using genetic programming where each element

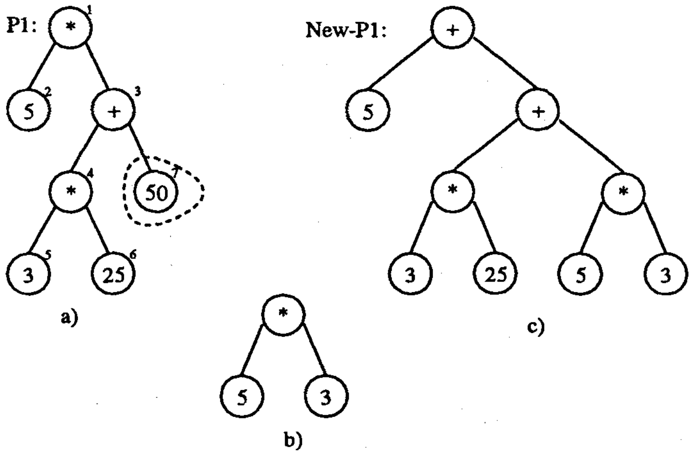

**Fig. 3.** a) An individual in mutation; mutation fragment is shown in dashed lines. b) Generated fragment for mutation. c) Offspring after mutation.

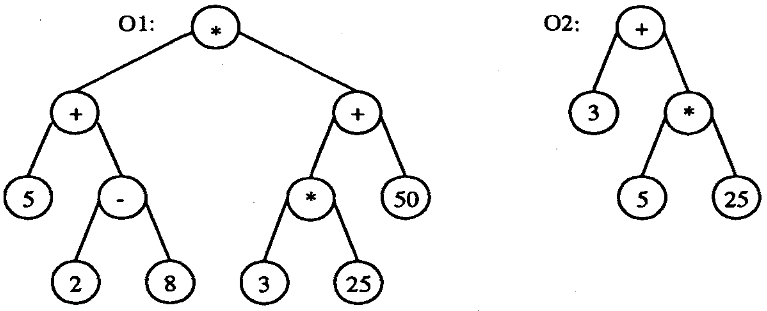

**Fig. 4.** Valid off-springs after crossover.

of the terminal set can be used at most once.

Before presenting our solution, to get an insight of our problem, let us analyze the search space. The search space for a GP, whose target language is LISP, is the space of all possible LISP S-expressions that can be recursively created by compositions of the available functions and available terminals for the problem.

In our problem the cardinality of the function-set is 4 and the cardinality of the terminal-set is 6. We define a valid tree as follows:

- each of the internal nodes should be an element of the function-set
- each of the leaf nodes should be an element of the terminal-set
- consists of at most 11 nodes
- leaf nodes should be distinct
- its depth should be at least 1
- every node except the leaves must have exactly 2 children, leaves do not have any children

Note that the first two of the conditions given above are from the definition of GP, and the last one is just a property of the function set used in 4-Op.

Any valid tree is a sample point in the search space. In order the find the number of points in the search space we should count all the possible valid trees that can be created. Given n terminal elements the number of different valid tree topologies that can be generated is equal to the different paranthesizations of a sequence of n numbers which is $K(n-1)$. This is known as Catalan numbers where:

$$K(n) = \left(\frac{1}{n+1}\right) C(2n, n) \tag{1}$$

Here, $C$ denotes the combination operation.

The valid trees we can generate will have at least 2 and at most 6 children. We will divide our computation into classes where $class(n)$ contains the set of trees with exactly $n$ leaves. After determining all different valid tree topologies in each class, we are going to compute the number of different valid trees we can create using the given function-set and terminal-set. For an illustration consider the valid tree topology shown in Fig.5. The numbers in each node represents the number of different choices we can insert into that node. Since there is no restriction on the choices of the functions as terminals at each internal node we have 4 choices. However, since we cannot use a terminal-set element more than once, at each external (leaf) node we have a decreasing sequence of choices. Therefore, in for the valid tree topology shown in Fig.5, there are $*4*4*6*5*4*3 = 23040$ different valid trees.

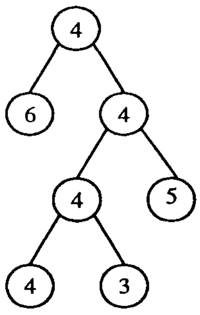

**Fig. 5.** A valid tree topology. The numbers represent the number of different choices for a node.

Let us now compute the number of all valid trees we can create given a specific terminal-set and function-set whose cardinalities are 6 and 4, respectively.

| | | |
|---|---|---|
| $class(2)$: | $K(1) = 1$ | $1 * (4 * 6 * 5) = 120$ |
| $class(3)$: | $K(2) = 2$ | $2 * (4 * 4 * 6 * 5 * 4) = 2608$ |
| $class(4)$: | $K(3) = 5$ | $5 * (4 * 4 * 4 * 6 * 5 * 4 * 3) = 115200$ |
| $class(5)$: | $K(4) = 14$ | $14 * (4 * 4 * 4 * 4 * 6 * 5 * 4 * 3 * 2) = 2580480$ |
| $class(6)$: | $K(5) = 42$ | $42 * (4 * 4 * 4 * 4 * 4 * 6 * 5 * 4 * 3 * 2 * 1) = 30965760$ |

For example, in $class(4)$ there are 5 different tree topologies, and 115200 different valid trees. Therefore, we can create a total of 33,666,168 different valid trees, i.e we have 33,666,168 sample points in the search space.

## 3 Solution

The easiest way to handle the anomalies discussed in the previous section is to generate crossover and mutation operations as usual and then discard the invalid parse trees. However when we implemented this solution, we saw that it is a very inefficient way to handle our problem, because about half of the population were formed with such invalid parse trees and we had to discard all of them.

Random keys, developed by Bean and Norman could be another solution [1, 2]. Random keys are developed to overcome the difficulty of genetic algorithms maintaining feasibility from parent to off-spring. To illustrate the use of random keys, consider a simple genetic algorithm approach to the traveling salesman problem. A candidate solution to a TSP is a tour through $n$ cities. Two such tours for a map of five cities are 2-1-3-5-4 and 4-2-3-1-5. Consider a crossover operation after the second city, then resulting off-springs are 4-2-3-5-4 and 2-1-3-1-5. Neither of these is a valid tour. As it can be seen in TSP a city cannot occur in a solution more than once, at first glance we may think that this is exactly the same problem we have in 4-Op, so that we can use random keys to overcome anomalies described in section two. However what makes our problem different is that, in GAs the strings have constant lengths but in GP the parse trees have variable sizes. This difference causes improper probabilistic distribution of terminal-set elements and we may have repetition of keys in later generations.

However we were able to develop another technique and a suitable data structure to overcome this problem. Before explaining our solution let us define some notions.

The function $S(T, node)$: returns the set of terminal-set elements appearing at the leaves of the tree rooted at node whose infix order numbering is $node$ in a tree $T$. Fig.6 gives the values of this function on an example tree.

We can state the necessary condition to guarantee having valid off-springs after crossover and mutation operations. Let $T1$ and $T2$ be two parse trees. An off-spring obtained by crossover operation applied to $x$ of $T1$ and $y$ of $T2$ is a valid tree if

$$(S(T1,1) - S(T1,x)) \cap S(T2,y) = \emptyset \qquad (2)$$

A crossover operation using map-trees is shown in Fig.7. In this example, the crossover points are 4 on T1 and 6 on T2.

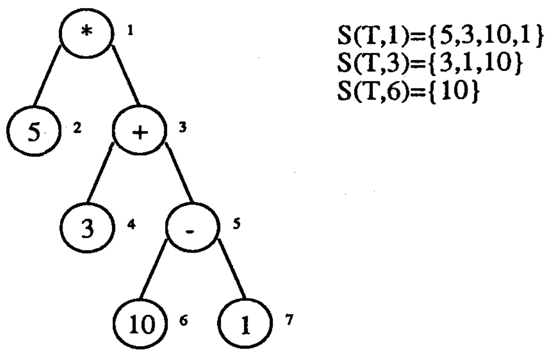

**Fig. 6.** Values of the function $S(T, node)$ on an example tree.

$$\begin{aligned}(S(\mathrm{T1},1) - S(\mathrm{T1},4)) \cap S(\mathrm{T2},6) &= \{\{5,3,25,50\} - \{3,25\}\} \cap \{2,8\}\\ &= \{5,50\} \cap \{2,8\}\\ &= \emptyset.\end{aligned}$$

Therefore, Off-T1 is a valid tree. However, since

$$\begin{aligned}(S(\mathrm{T2},1) - S(\mathrm{T2},6)) \cap S(\mathrm{T1},4) &= \{\{3,5,8,2,25\} - \{2,8\}\} \cap \{3,25\}\\ &= \{3,5,25\} \cap \{3,25\} = \{3,25\}\\ &\neq \emptyset,\end{aligned}$$

Off-T2 is not a valid tree.

Also let $S(Tm, z)$ be the set of terminal elements of the mutation subtree and $x$ be the mutation point on $T1$. The resulting off-spring is a valid tree if

$$(S(T1,1) - S(T1,x)) \cap S(Tm,z) = \emptyset \tag{3}$$

In our implementation, we first check if the crossover points are valid for parents $T1$ and $T2$. If they are not valid for both of them we generate randomly two other crossover points and continue the process until we can generate a valid offspring at least for one of the trees. If the crossover is valid for only one of the trees then we generate the valid offspring and reproduce the remaining tree.

It is not possible to implement the set operations using only the parse trees because of the time efficiency reasons, so we have used another data-structure. For each parse tree, we also store the *map-tree*. A node of a map-tree stores the set of terminal-set elements occuring in the leaves of the subtree rooted in that node. A parse tree and its corresponding map-tree are shown in Fig.8.

The set operations are carried out on this tree more efficiently. It can be easily seen that the map tree can be constructed by exchanging every node of the parse tree with the set returned by $S(T, node)$.

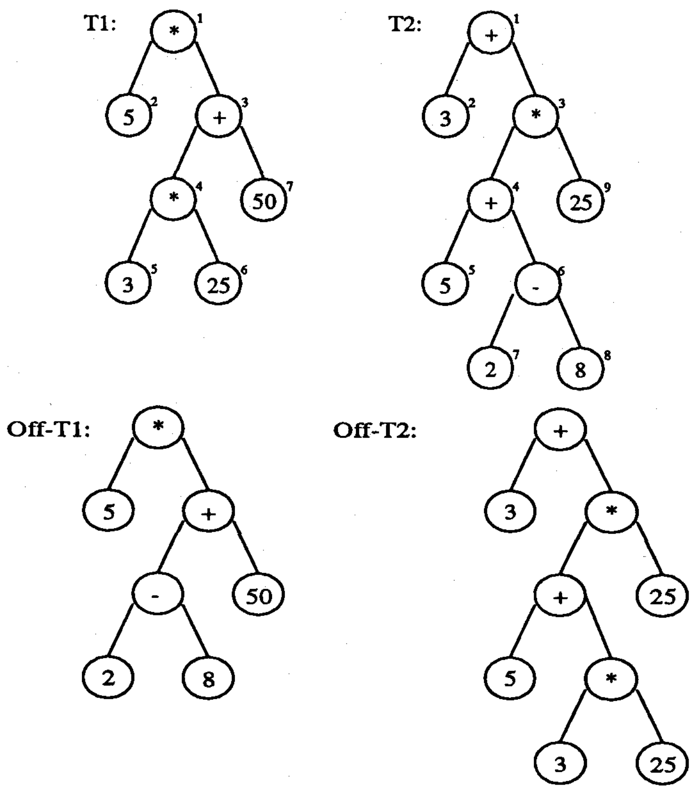

**Fig. 7.** A crossover operation using "map-tree." Crossover points are 4 on T1 and 6 on T2.

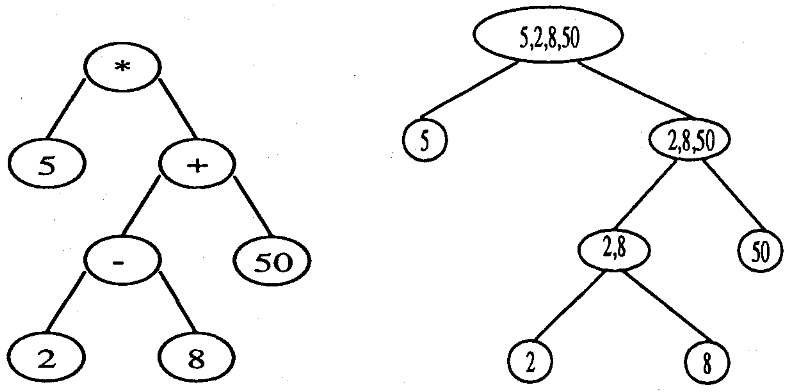

**Fig. 8.** A parse tree, and its corresponding map-tree.

Other than crossover and mutation anomalies, creating the initial random population is another problem that we have encountered. Since the individuals in the initial population are created randomly, this may easily lead to forming illicit parse trees where there is a repetition of terminal-set elements. Preventing such parse trees in the initial population is easier than preventing mutation and crossover anomalies. The idea is that, for each individual in the population, after choosing a terminal-set element randomly, this element is discarded from the set so that repetition of elements is prevented.

## 4 Empirical Evaluation

We have used 110 randomly generated input data in order to test and examine the results of our technique. In these experiments the population size used is 250, mutation rate is 10% and number of generations examined is 20. 10% mutation rate can be considered rather high since in genetic programming applications the mutation rate is usually zero. However the search space of our problem is relatively small and the loss of genetic information due to randomness of mutation can be recovered by crossover operations. On the other hand 10% mutation rate in this problem provides a means for recovering from local maximas and leads to a better examination of the search space.

In Table 1, for the following five input data sets, the values of the average fitness and best fitness versus generation numbers are given. In these five selected examples we can have an insight of how our program approaches to the target value in each generation. The data sets are:

Data set 1: Input integers are {2,4,6,7,25,75} and target value is 458
Data set 2: Input integers are {1,3,5,9,25,50} and target value is 846
Data set 3: Input integers are {1,4,8,9,25,50} and target value is 359
Data set 4: Input integers are {4,6,7,9,25,75} and target value is 793
Data set 5: Input integers are {2,3,7,9,25,100} and target value is 458

In Table 2 the test results for the 110 random input data sets are grouped according to fitness measure. As it can be seen in Table 2, in 40% of the test results we have found an exact solution. If we consider that some input data sets do not contain exact solutions, we can claim that these test results are successful.

The graphs given in Fig.9 and Fig.10 show the average of "average fitness" values versus generation number and average of "best fitness" values versus generation number. In these figures the fitness of a tree is computed as the absolute value of the difference between the target value and the value of the expression represented by the given tree. As it can be seen in Fig.9 after the dramatic fall in the first generation, although there is a fluctuation due to the high mutation rate (10%), the average of "average fitness" of the population shows a decreasing behavior throughout the generations. In Fig.10 the average of "best fitness" values decreases steadily, and after nineteen generations the value of the average of "best fitness" reaches to 1.5.

**Table 1.** Average and best fitness values versus generation.

| | Dataset 1 | | Dataset 2 | | Dataset 3 | | Dataset 4 | | Dataset 5 | |
|---|---|---|---|---|---|---|---|---|---|---|
| Gen | Avg. | Best | Avg. | Best | Avg. | Best | Avg. | Best | Avg. | Best |
| 0 | 2250.8 | 11 | 2746.5 | 46 | 4027.4 | 9 | 4446.7 | 14 | 5210.4 | 8 |
| 1 | 479.11 | 4 | 1266.8 | 21 | 339.2 | 9 | 1833.4 | 7 | 996.5 | 8 |
| 2 | 350.4 | 4 | 867.8 | 21 | 996.1 | 9 | 514.1 | 7 | 2849.0 | 8 |
| 3 | 260.0 | 2 | 808.8 | 4 | 271.2 | 9 | 373.8 | 7 | 495.3 | 8 |
| 4 | 206.3 | 2 | 473.6 | 4 | 282.7 | 9 | 426.8 | 7 | 5554.6 | 4 |
| 5 | 263.6 | 2 | 455.5 | 4 | 418.3 | 9 | 309.5 | 7 | 2326.3 | 4 |
| 6 | 672.7 | 2 | 501.2 | 4 | 477.0 | 1 | 292.1 | 0 | 685.5 | 4 |
| 7 | 1043.9 | 1 | 839.3 | 4 | 320.2 | 1 | - | - | 1196.8 | 3 |
| 8 | 667.4 | 1 | 398.7 | 4 | 2094.9 | 1 | - | - | 1426.0 | 3 |
| 9 | 174.7 | 1 | 365.1 | 4 | 276.6 | 1 | - | - | 3311.4 | 3 |
| 10 | 123.2 | 1 | 233.3 | 4 | 206.1 | 1 | - | - | 2225.6 | 3 |
| 11 | 802.2 | 1 | 655.7 | 4 | 160.9 | 1 | - | - | 1275.7 | 3 |
| 12 | 255.6 | 1 | 510.1 | 4 | 199.1 | 1 | - | - | 842.6 | 3 |
| 13 | 126.0 | 0 | 1004.9 | 4 | 199.1 | 1 | - | - | 905.1 | 3 |
| 14 | - | - | 537.0 | 4 | 190.9 | 1 | - | - | 1065.7 | 3 |
| 15 | - | - | 431.9 | 4 | 157.5 | 1 | - | - | 1536.2 | 3 |
| 16 | - | - | 425.3 | 4 | 244.1 | 1 | - | - | 152.3 | 3 |
| 17 | - | - | 451.1 | 4 | 185.3 | 1 | - | - | 191.4 | 3 |
| 18 | - | - | 537.6 | 4 | 135.0 | 1 | - | - | 157.6 | 3 |
| 19 | - | - | 1241.1 | 4 | 283.7 | 1 | - | - | 255.5 | 3 |

**Table 2.** Fitness measures by grouping.

| Fitness Measure | Number of Times |
|---|---|
| 0 | 44 |
| 1 | 31 |
| 2 | 12 |
| 3 | 7 |
| 4 | 4 |
| 5 | 6 |
| 6 | 0 |
| 7 | 0 |
| 8 | 0 |
| 9 | 1 |
| 10 | 1 |

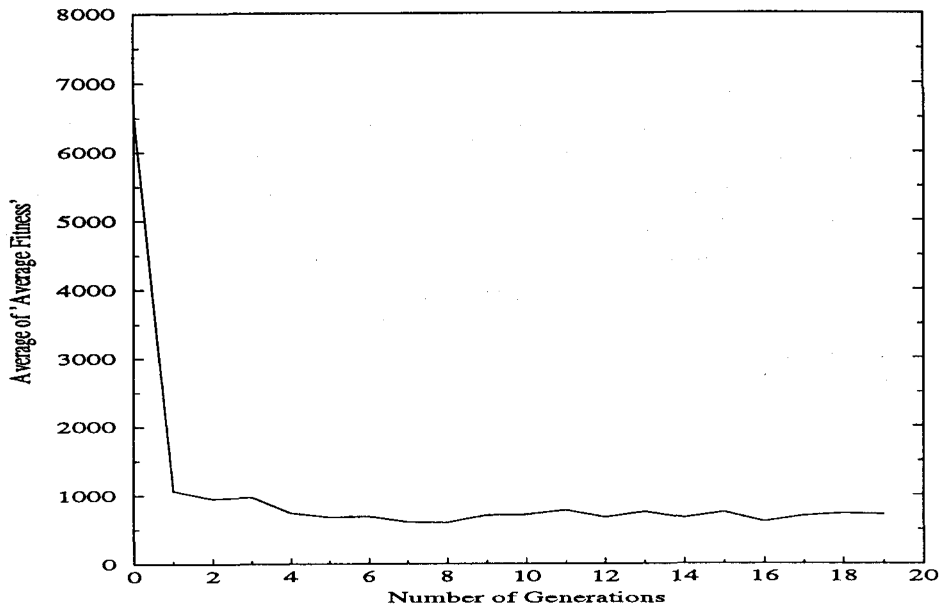

**Fig. 9.** Average of "average fitness" versus number of generations.

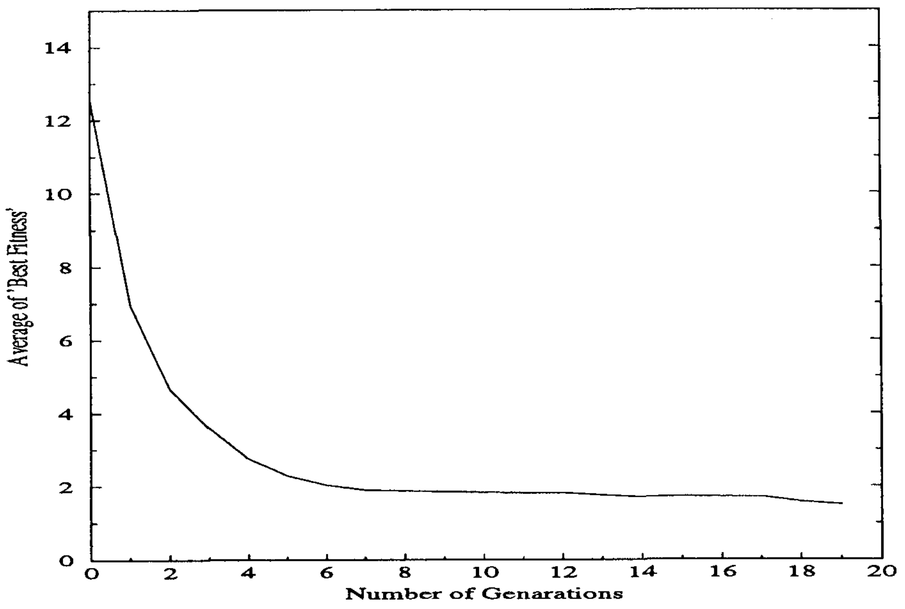

**Fig. 10.** Average of "best fitness" versus number of generations.

## 5 Conclusion and Future Directions

The genetic programming paradigm provides a way to solve a wide variety of different problems from many different fields. These problems can be reformulated as the problems of program induction. When we have a problem whose search space is well characterized and if we have also a good heuristic to solve the problem possibly genetic programming would not give a better result. However it is very convenient to use genetic programming when we do not know how to approach to the problem. Various applications of GP on many different subjects provide considerable evidence of the generality of the GP paradigm.

In standard GP the user determines the elements in the function set and the terminal set. But he/she can not put a restriction on the number of times of their usage, i.e. on the number of times of the occurrences in the parse trees. Restrictions on some problems make standard GP inapplicable. 4-Op problem is one of them and it puts a restriction on the number of times of using the terminal-set elements. More specifically a terminal-set element can be used at most once. Our technique makes GP applicable to 4-Op problem. In the experimental results we have observed that our program has given 40% exact solutions and after about eight generations the average of "best fitness" is below two. These results indicate that our technique is effective in the solution of this kind of problems.

Our technique is not specific to 4-Op problem. It can be extended, without changing the idea behind it, to problems that limit the use of not only terminal-set elements but also function-set elements. It can be used for all problems where the elements of the terminal or the function-set are to be used for a specific number of times.

## References

1. Bean J.C.: Genetic and Random Keys for Sequencing and Optimization. Dept. of Industrial & Operations Engineering, Univ. of Michigan, Technical Report (June 1992) 92-43
2. Bean J.C. and Norman B.: Random Keys for Job Scheduling. Tech. Report, Dept. of Industrial and Operations Engineering, Univ. of Michigan, Ann Arbor (January 1993)
3. Goldberg D.E.: Genetic algorithms in search, optimization, and machine learning. Addison-Wesley (1989)
4. Holland J.H.: Adaptation in natural and artificial systems. University of Michigan Press (1975)
5. Koza J.R.: Genetic programming on the programming by means of natural selection. Cambridge, MA: The MIT Press (1992)