Introducing local optimization for effective Initialization and Crossover of Genetic Decision Trees

Arindam Basak and **Sudeshna Sarkar**
Computer Science & Engineering Department
Indian Institute of Technology,
Kharagpur, WB. INDIA: 721302
{abasak,sudeshna}@cse.iitkgp.ernet.in

Abstract

We introduce a new genetic operator, Reduction, that rectifies decision trees not correct syntactically and at the same time removes the redundant sections within, while preserving its accuracy during operation. A novel approach to crossover is presented that uses the reduction operator to systematically extract building blocks spread out over the entire second parent to create a subtree that is valid and particularly useful in the context it replaces the subtree in the first parent. The crossover introduced also removes unexplored code from the offspring and hence prevents redundancy and bloating. Overall, reduction can be viewed as a local optimization step that directs the population, generated initially or over generations through crossovers, to potentially good regions in the search space so that reproduction is performed in a highly correlated landscape with a global structure. Lexical convergence is also ensured implying identical individuals always produce the same offspring.

1 Introduction

A decision tree is a hierarchical, sequential classification structure that recursively partitions the instance space into mutually disjoint sections. Non-leaf nodes of the tree are labeled with attributes, tree edges emanating from such nodes represent corresponding attribute values. The leaf nodes denote the classifications of instances reaching them on taking appropriate branches according to values of attributes being tested at the non-leaf nodes passed after starting from the root.

Genetic programming is a variant of genetic algorithms used to automatically generate computer programs by representing them as parse trees of varying sizes. Since a decision tree can be viewed to be denoting a composition of functions, the functional nodes of the parse tree to be used in genetic programming are associated with the concept attributes which specify tests to be carried out on a single attribute, with a branch for each of its possible values. The terminal nodes represent concept classes. Each path from root to a terminal leaf correspond to a concept that is a conjunction of tests in the functional nodes. In this paper we discuss an extension of the conventional genetic programming approach to learning decision trees.

2 Previous work

The crossover operator conventionally applied in genetic programming crosses two programs by exchanging subtrees at randomly selected nodes. An important constraint that has to be taken care of while generating the initial population or while implementing crossover is that of preserving the closure property which means that all variables, constants, arguments for functions and values returned from functions must be of the same data-type. This implies that any element can be a child node of any other element, which is clearly not valid in case of syntactically correct decision trees. A syntactically correct concept includes no more than one attribute from each kind. Hence, cross-points in the parent trees have to be chosen so that it results in offsprings having non-repeating attributes on any particular path from the root to a leaf.

Koza[Koza, 1992] describes a way to relax this constraint of closure somewhat with the concept of constrained syntactic structures in which argument types or return values that are not standard can occur in a tree but only at a pre-assigned position. **Strongly Typed Genetic Programming(STGP)**[Montana, 1995] allows functions that take arguments of any data-type and return values of any type by requiring the user to specify precisely the data types of the arguments of each function and its return values. The concept of strong typing has been applied in genetic programming to induce syntactically correct decision trees, as in [Martijn Bot, 2000].

Strong typing essentially introduces type-based restrictions on which an element can be chosen at each node during initialization or during crossover. During initializaton, the element chosen at a node must return the expected type. During crossover, the second parent is selected so that the chosen subtree within returns the same type as the subtree of the first parent at the crossover point. In case there are two or more valid subtrees in the second parent, a random selection is made from all satisfying this constraint.

3 Objective

To ensure syntactic correctness of the offspring using crossover used in STGP, not only one has to search for a tree which has a subtree valid for the crossover site in the first parent but also a random choice has to be made among different valid subtrees if present in the second parent The main focus is on somehow obtaining a valid subtree, rather than choosing one which might increase the accuracy of the offspring. Over generations, trees produced in this way have large parts which are never used, and which only contribute to increasing the tree size without improving its accuracy. Redundancy in the trees increases over generations, as redundant trees have a higher chance of surviving crossover. Coupled with it is the fact that increasing redundancy makes it less likely to choose a non-redundant edge as a crossover point and hence hinders the evolution of truly new individuals. Thus, the probability to escape a potential local optimum decreases with time[Tobias Blickle, 1994].

A new operation of **reduction** is introduced in this paper using which a significantly different implementation of crossover is presented which takes care of the above problems. It has been observed about the random subtree crossover in genetic programming that expressions grow because they are defending themselves against the destructive effects of crossover and attempting through redundancy to preserve their current fitness in the next generation[Singleton,]. The crossover operation introduced here eliminates redundancy. It also attempts to prevent destructive effects of crossover by using the building blocks spread throughout the entire second parent to extract sections which are particularly useful in the context the generated subtree is going to appear in the offspring. The method ensures that bloating is prevented and automatically takes care of syntactic correctness of the offspring. This also takes care of lexical convergence in the sense that two identical individuals produce the same individual as offspring, independent of the choice of the crossover point in the first parent. In this respect, it is similar to the cut and splice crossover used in the classical bit string genetic algorithms. The reduction operation introduced can also be used to remove redundant code from trees generated in the initial population.

4 The introduced operations

4.1 Reduction

The operation of reduction applied to a genetic program representing a decision tree with respect to a set of labeled instances uses the program execution path to remove unexplored sections in the code yielding a shorter code of the same accuracy on the instance set as in the original version. It effectively removes the redundant tests which were being carried out by the original program without changing the relative ordering among the tests retained.Trees incorrect originally get automatically rectified in the process.

The subroutine implementing reduction takes as input (i) a decision tree complete with all annotations except for the labeling of the terminal leaves(labeling of the leaves, if present, is ignored). and (ii) a set of labeled instances. It produces a syntactically correct decision tree with terminal leaves labelled appropriately with concept classes. The recursive algorithm for reduction is outlined below :

```
REDUCE ( tree T, non-zero instance set I )
 OUTPUT : T reduced with respect to I
 THE ALGORITHM :
  - IF ( T is leaf ) return T.
  - IF ( each instance of I is of same class C )
    - make T a leaf representing C.
  - FOR ( each child subtree t of T )
    - let S be the subset of I branching to t.
    - IF ( S is non-empty ) REDUCE ( t, s ).
       ELSE
       - make t a leaf representing the class
         of majority of I's instances.
  - IF ( same child subtree r is
          traversed by each instance in I )
    - make T equal to r.
  - IF ( all explored child subtrees are
          leaves representing same class C )
    - make T a leaf representing C.
  - return T.
```

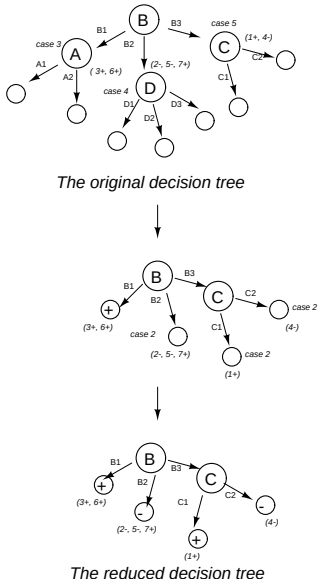

Figure 1: Illustration of the Reduction operation (the cases correspond to those in the algorithm for reduction; labelled instances reaching the nodes are shown in parentheses).

The procedure is illustrated in Figure 1 for the dataset in Table 1.

4.2 The Crossover Operation

The crossover operation generates a single offspring from two parent programs.

instances	Attributes				Class
	A	B	C	D	
1	a2	b3	c1	d1	+
2	a1	b2	c2	d1	-
3	a1	b1	c1	d2	+
4	a2	b3	c2	d2	-
5	a2	b2	c2	d1	-
6	a1	b1	c2	d2	+
7	a1	b2	c1	d1	+

Table 1: Decision table corresponding to the illustrations reduction and crossover

Two equivalent views of crossover are presented as follows:

```
CROSSOVER // first view
 INPUT: tree T1, tree T2 //the parent individuals
 OUTPUT: tree R // the offspring
 THE ALGORITHM :
  - select a subtree t in T1.
  - transform T1 by replacing t with T2.
  - reduce the new T1 with respect to the
    training data set to obtain R.
  - return R.

CROSSOVER // second view
 INPUT: tree T1, tree T2 // parent individuals
 OUTPUT: tree R // the offspring
 THE ALGORITHM :
  - select a subtree t in T1.
  - reduce T2 with respect to the subset of
    training data which would explore t if T1 is
    run on the whole dataset.
  - replace t in T1 with reduced T2 to obtain R.
  - return R.
```

The crossover procedure is illustrated in Figure 2.

The intermediate tree generated by the first approach is likely to be an incorrect one but gets corrected by the subsequent reduction, so that crossover operation preserves the syntactic correctness of decision trees. The crossover operation generates an entirely new tree by collecting useful building blocks from throughout the second tree. The redundant sections get eliminated, which ensures that bloating and redundancy are checked.

The second view of crossover evinces the fact that the tree replacing the selected subtree in the first parent effectively utilizes the classification power of the second parent as a whole as far as the subset of training instances to be classified by it is concerned. This makes it more likely that the offspring would be of better accuracy besides being limited in size. Lexical convergence is taken care of by the fact that the original tree reduced with respect to the set of instances exploring any of its subtrees yields the subtree itself. The subtree to be replaced in the first parent is chosen to avoid wasting crossovers by changing code that is never executed or disrupting code that is working. To ensure this the subtree rooted at a node randomly selected is considered to be eligible for replacement only if it is explored by instances of more than one concept class. Hence, leaves never visited or visited by instances of the same class are not considered for replacement.

5 Experiments

Performance is measured by a fitness function which assigns fitness score to each individual. For experiments described in this paper, fitness is simply the accuracy of the individual with no inherent steps taken to prevent bloating. Only among individuals of same accuracy, trees with lesser number of nodes are considered fitter. A rudimentary fitness function as the one above has been deliberately used to let us more clearly examine the strength of the introduced operations. Tournament selection is used as the selection method. It means that a number of individuals are randomly selected from the population and the fittest two among them are crossed over to generate a new offspring which overwrites the worst individual of the tournament in the next generation if the latter is worse than the one produced.

5.1 Results

Parameter settings

Population size : 250
Tournament size : 7
Terminating condition : standard deviation of training accuracy among the individuals in the population is down to less than 0.001.

Databases

Source : Machine Learning Repository on the website : http://www.ics.uci.edu/˜ mlearn/MLRepository.html
Title : The Monk's Problems
Attributes : discrete, 7 in number(including the class attribute and excluding the instance Id).
Number of classes : 2

3 datasets were used :
monks-1.train(124 instances),
monks-2.train(169 instances),
monks-3.train(122 instances).

Observations and Analysis

Experiments were conducted on learning decision trees from training data using the proposed approach and the results were compared with those obtained using the C4.5 algorithm(Quinlan,1993), an extension of the original ID3 algorithm(Quinlan,1979), a classical approach for building decision trees from discrete training instances.
For C4.5 the source code was obtained from website : http://www.cse.unsw.edu.au/˜ quinlan
and the learning was based on training data with subsequent pruning.

The observations were recorded as in Table 2.

In Table 2, corresponding to the tree produced by C4.5 for a given dataset, GP results enlist the individuals, among those produced during 10 runs of the proposed algorithm, which dominate the tree C4.5 generates(A tree is considered to dominate another if the former has

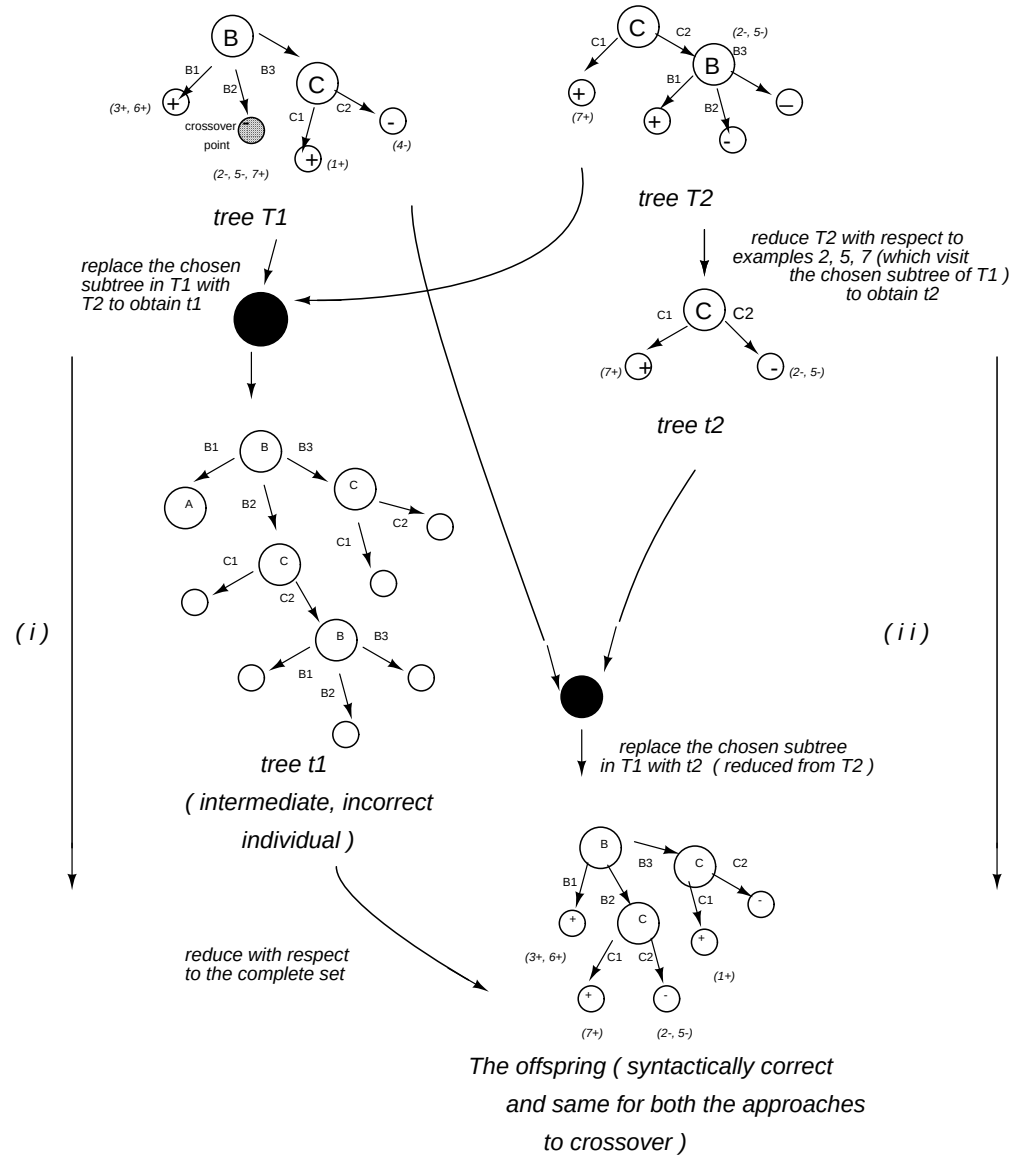

Figure 2: Illustration of the two views of crossover in (i) and (ii)(T1 and T2 are the trees being crossed over to obtain a single offspring).

Dataset	Algoritm	Training Accuracy	Number of Nodes
monks-1	C4.5(unprunned)	90.32	43
	GP	100.00	37
		99.19	35
		98.38	33
		97.58	31
		96.77	29
		95.96	29
		95.16	27
		94.35	25
		93.54	25
		92.74	23
		91.93	23
		91.12	21
		90.32	19
	C4.5(prunned)	83.87	18
	GP	87.90	18
		87.09	17
		86.29	17
		85.48	14
		84.67	16
		83.87	14
monks-2	C4.5(unprunned)	85.79	73
	GP	85.79	61
	C4.5(prunned)	76.33	31
	GP	78.69	28
		78.10	29
		77.51	26
		76.92	24
		76.33	23
monks-3	C4.5(unprunned)	96.72	25
	GP	96.72	25
	C4.5(prunned)	93.44	12
	GP	93.44	12

Table 2: Genetic decision trees compared with those learned using C4.5.

at least as much accuracy and at most as many number of nodes as compared to the latter). C4.5 trees both before and after prunning are compared.

As is evident from the observations, the genetic decision trees produced are in general of much smaller size compared to those of similar accuracy learned by C4.5, wheather before or after prunning.

This shows that the heuristic based deterministic search done by C4.5 was unable to generate optimal trees and the genetic trees produced were better in comparison. Experiments are yet to be conducted comparing the proposed approach with the state of the art ones in genetic programming and it is likely that the former will generate better individuals in shorter time.

6 Conclusion

Reduction has been introduced as a local improvement operator used during initial population choice and crossover to guide candidates which are unfit or which violate problem constraints to nearby locally optimal regions in the search space. The context-preserving non-uniform crossover proposed in this paper collects useful sections from the parents while rejecting the redundant ones so that the generated offspring is syntactically correct, limited in size and likely to be of higher accuracy. Prolification of building blocks and lexical convergence are also taken care of. In this way, exploration of search space is limited to reasonably good points so that global or nearly-global solutions are obtained reliably and quickly.

References

[Koza, 1992] Koza, J. R. 1992. *Genetic Programming: On Programming Computers by Means of Natural Selection and Genetics*. The MIT Press, Cambridge, MA.

[Martijn Bot, 2000] Martijn Bot, W. 2000. Application of genetic programming to induction of linear classification trees. In *European Conference on Genetic Programming EuroGP2000, Lecture Notes in Computer Science 1802*, 247–258. Springer.

[Montana, 1995] Montana, D. J. 1995. Strongly typed genetic programming. *Evolutionary Computation* 3(2):199–230.

[Singleton,] Singleton, A. Greediness. Posting of 8-21-1993 on genetic-programming@cs.stanford.edu mailing list.

[Tobias Blickle, 1994] Tobias Blickle, L. T. 1994. *Genetic Programming and Redundancy.*