Implementation and Evaluation of a Novel "Branch" Construct for Genetic Programming

Kevin A. Gibbs
Department of Computer Science
Stanford, California 94305
kgibbs@cs.stanford.edu
http://www-cs-students.stanford.edu/~kgibbs

Abstract: This paper describes a technique for implementing a novel type of "branch" operator within a genetic programming system. This branch construct is a new operator type that allows arbitrary branching from one location in an individual's execution tree to another. The branch can be understood as alternatively allowing arbitrary code reuse or approximating access to a potentially infinite number of automatically defined functions. This paper describes the proposed design of this branch operator. This proposed design is then implemented in a real world system, and the performance effects of the branch operator are evaluated in two well known genetic programming problems: the artificial ant problem and the lawnmower problem. [1,2] The branch is found to provide some performance benefits in both of these problems, and areas for further investigation are outlined.

Introduction and Overview

In the day-to-day programming done by humans, most all control structures in code originate from a high level. Whether programming in a low-level language like C or a higher-level language like LISP, we are accustomed to using high-level control constructs like functions, loops, if statements, and recursion to control the path of execution and maximize code reuse. The thought of using a branch, or "goto" or "jump" statement, to directly skip around in code, is something we are generally taught as a bad programming practice. And for human-based programming, it may well be—branches lack the abstractness and concise qualities of higher-level constructs like loops and functions. Yet all programmed code is ultimately reduced to assembly language machine code, which performs all control with branch statements, simply jumping from place to place in the code, frequently based on the results of a simple test like "less than equal to zero."

So, in the domain of genetic programming, is there any reason why simple branch statements should be avoided? Certainly the rules of human programming etiquette no longer apply, and cosmetic or readability reasons are not particularly valid. Higher-level operations like if statements and automatically defined functions certainly seem to provide evolutionary shortcuts to a genetic programming system, by implementing more powerful control structures. But their lower-level components would also seem to be desirable tools for an evolutionary system to have at its disposal in developing particular control structures for the system at hand.

In that light, this paper investigates the use of a "branch" operator in genetic programming. The Background section, which follows this one, talks briefly about the current types of control constructs available in genetic programming to provide motivation for the branch operator. The third section, Statement of Problem, discusses the desirables of a branch construct for GP. The Proposed Design section describes the design of the branch operator that was developed and how it interacts with the rest of the genetic programming system. The Methods section talks about how our test system was implemented, and how we implemented branching support in the given problems. Results and Discussion of Results go over what was observed from the experiments on the test problems. Finally, the Conclusion reviews the overall effectiveness of the branch operator, as demonstrated by our test problems, and the Future Work section outlines areas for further investigation of the branch operator that could provide interesting results.

Background

Currently, the main evolutionary control structure used in genetic programming problems is the automatically defined function, or ADF. ADFs are formed by evolving a separate tree, possibly with argument variables, and allowing that tree to be called as a function node from another tree. ADFs have shown great performance benefits in a wide range of genetic programming problems, primarily because of their ability to allow developed functionality to be reused throughout a larger genetic program. ADFs, however, since they are evolved separately and specifically by the genetic programming system, generally add a significant layer of complexity to the genetic programming system and involve making significant changes to that system. These ADFs are also usually created for a particular problem in some small finite number, usually around one or two, which limits the amount of arbitrary code and functionality reuse that can go on in the evolved program.

Other types of evolutionary control structures used in genetic programming include automatically defined loops, automatically defined recursive functions, and if-like statements, that execute one segment of code or another, usually based on the result of a test. All of these are useful and powerful control constructs, but they are all ultimately used to either make a decision on the path of execution for an individual (if statements), or to repeatedly reuse segments of code (things like loops and recursion). But while all of these structures can be simply defined in terms of a more fundamental branch statement, these simpler branch statements are not available to the genetic programming system. Moreover, these functions like loops, recursions, and if statements, usually only allow control changes and code to be reused only in a local way, based on the things that are near them in the program tree. Arbitrary control and reuse functions, in the sense of allowing any code to be used anytime, anywhere, do not really exist for genetic programming systems.

Statement of Problem

To build a genetic programming operator to function as a "branch" operator, some desirables seem apparent. We want an operator that:

- Can jump execution to *arbitrary* locations in the code, above, to the side, or below the current node in the tree.
- Able to reuse any segment of code found in the tree.
- Able to cope with common reproduction operators for genetic programming in a sensible way. Function of unmodified code areas with branches operators should be affected as little as possible by crossover and mutation.
- Generality in the operator, so that different types of branches can be used for different types of problems.
- Simple and elegant solution, so that any modification to the genetic programming system itself requires little change and is relatively painless.
- Will not "break" the system if an infinite loop is encountered.

Proposed Design

To solve this problem, the `branch` type for a function was devised. The `branch` function type works by obtaining a pointer to some other random node within the program tree when an instance of it is created. This random node then becomes the *branch destination* for that particular `branch` statement. Then, based upon the implementation of the particular function, either that random branch is executed, or some other segment of code is. Since `branch` is a type of function, rather than a particular function, this is left open to the particular branch function that implemented.

Functions of type `branch` are simply provided, along with their other specified arguments, one additional argument that appears to be merely another statement that can be evaluated, just like the rest of the arguments. The only thing "special" about this additional argument is that it actually points to some other subtree within the execution tree that may not be a child of the particular function.

Thus, a "branch always" function can be simply implemented by specifying a `branch` type function that takes no arguments and always execute the special branch argument. A "iflte" or "if less than equal to zero" branch can be easily be implemented by taking two arguments, one as the test expression and the other as

the expression to be run if the test is false. Then the test is simply executed and the branch or the other remaining expressions are evaluated based on the results of the test.

The random branch destination that the instance of a function of type `branch` is assigned to upon creation always lies within the same execution tree as the function itself. In other words, the branch will not jump from the main body to an ADF tree, or vice versa. The random branch destination for a function of type `branch` is intrinsic to that particular instance, and is not modified directly by crossover, mutation, or other operators. It is in essence a constant, immutable pointer to some other node in the execution tree.

However, if the node that a `branch` function instance points to is destroyed by crossover or mutation, the branch destination of that function instance is assigned to some new random value. The branch destination is like a pointer in a particular memory space: as long as the source and destination lie in the same "space," or particular execution tree of a given individual, the link exists, but as soon as one or the other lies elsewhere, a new random link is made. Thus, the desired property of minimal change of functionality by crossover and mutation is upheld for the `branch` functions.

Now that `branch` type functions can exist, which can link to arbitrary locations in the execution tree, it is possible for infinite recursion to occur. To solve this, the genetic programming system is provided with a count of the number of branch operations executed, as well as the total number of operations executed. Evaluation can be stopped after one or the other of these goes over some certain maximum, which then stops the infinite recursion. Default values are then returned, rather than the value the `branch` function would have returned.

Thus, this design allows for nearly any type of branching function to be created, simply by creating a new function of type `branch` and having it evaluate whatever is desired from its arguments and the branch argument. Moreover, the design constraints listed above are satisfied. Functions can jump to arbitrary locations and any in the tree code can be reused. The `branch` functions deal with tree-modification operators by making the minimum change possible, by "pointing" at some particular node, and they escape from an infinite loop, thanks to the branch counters provided. And finally, this design is simple and elegant, allowing for a very generalized type of `branch` function class, while requiring little change on the code of the host system, since all `branch` type functions need in additional to the usual facilities are a random branch destination and the ability to update their branch destination if the destination node disappears.

Implementation and Methods

To test the `branch` operator design out and evaluate it on actual problems, the `branch` function type was implemented in the lil-gp genetic programming system, from the Michigan State University's Genetic Algorithms Research and Applications Group [3]. Lil-gp is a genetic programming environment written in C that produces programs that follow Koza's Simple LISP model in *Genetic Programming I* [1]. The final evolved program can be output to a functional LISP program, which can be run for testing.

Implementation of the `branch` function type for lil-gp was straightforward. It followed the design given above as would be expected. Since lil-gp already encompassed a FUNC_EVAL and a FUNC_EXPR function type, which correspond to user functions which receive their arguments pre-evaluated and user functions which receive their arguments as tree pointers that can be evaluated, adding a new FUNC_BRANCH type based on FUNC_EXPR was not difficult. The only real complexity in implementing the branch functionality for lil-gp was in the system for maintaining the branch destination pointers during crossover and mutation. Since lil-gp stores its execution trees in a linear array format for speed, branch destination pointers were set as relative offsets into the array. These relative pointers had to be updated by addition and subtraction when a crossover occurred if their destination was on the opposite side of the crossover segment. In other words, if the destination pointer jumps over the whole crossovered section of the array, which will grow or shrink in size, the destination pointer must be updated properly. However, since all of these arithmetic pointer offset updates can be done in a single pass of the individual, after crossover has completed, the running time addition is negligible.

The max_branch_ops and max_ops counters were implemented to stop infinite recursion, and we found that setting max_branch_ops to a reasonable value like 200 worked well for eliminating the infinite loops without otherwise altering execution of the program.

With the `branch` type of function now implemented in lil-gp, suitable problems now had to be implemented with `branch` functions to evaluate the usefulness of the branch operator. The lil-gp system comes with a selection of well known genetic programming problems already implemented, and for testing the branch construct we decided to use the artificial ant problem, after Koza [1], and the lawnmower problem, after Koza [2]. Both of these programs are implemented exactly as described in the books and hence would seem to provide a good starting point for evaluating the effectiveness of the branching operator.

In addition, both the artificial ant and lawnmower problems are simulation type problems, where the user functions in the produced code cause a system to take some action, like “move forward” or “turn.” Thus these problems would seem to benefit from exploiting repetitive control structures so that beneficial segments of code can be run repeatedly or “linked to” from different places in the output program. Other problem types, like numerical regression and multiplexer development, do not seem to have as clear of a rational justification for control structures like branches, and preliminary testing on these example problems found little improvement.

Both the artificial ant problems and the lawnmower problems are quite well known, so a short explanation should suffice. The artificial ant problem is composed of a field that has food along a marked “trail” on the field [1]. This trail contains gaps, however, where there is no food, so following the path can be difficult. The genetic program functions as a controller for the ant, and has the functions “if-food-ahead,” “move,” “right,” and “left” available. The lawnmower problem is somewhat similar, but also quite different. For the lawnmower problem, the evolved program must guide a lawnmower around an 8x8 field, mowing all the grass, while not wasting additional time removing already mowed areas [2]. It has functions “left,” “mow,” “frog,” and “vma,” along with a random constant. “frog” jumps the lawnmower forward a certain number of squares, and “vma” adds vectors together.

Tableaus have not been included for the artificial ant and lawnmower problems as the problems were not originally derived for this paper, and they are both very well known genetic programming problems. But if desired, tableaus for the artificial ant and lawnmower problems exist verbatim in *Genetic Programming I* and *Genetic Programming II*, respectively [1,2], and we use those tables exactly in our experiments.

Implementing the `branch` type functions into the two sample problems was simple. Since the artificial ant problem already contains an if-type function, “if-food-ahead,” which takes two arguments, one to be evaluated for true and one evaluated for false, this function was simply duplicated to form a “br-food-ahead” function, which takes the branch destination and one argument, with the branch expression evaluated when food was ahead, and the other argument if not. This function can then was then compared separately and in tandem with the previous “if-food-ahead” function.

For the lawnmower problem, no function similar to a branch existed, and hence one was created from scratch. The branch function chosen was “br”, or simply branch, which did just that: it always branched to the random branch location when executed. This function was used because the lawnmower problem has the ability to use ADFs, and we wanted to compare the ability of branch functions to that of ADFs.

With all of this implemented, our method was to run a large number of runs of both problems with different combinations of branch and lack of branch statements to see how the `branch` function type affected performance.

Results

Tests were first run on the artificial ant problem. We initially tested the problem with both well-known trails, the Santa Fe trail and the harder Los Altos, with the original settings containing no branch functions and thus only the if-food-ahead function (ifonly), containing both the br-food-ahead branch and if-food-

ahead functions (both), and containing only the br-food-ahead branch function and no if-food-ahead function (bronly). We ran each test with the settings given in Koza [1], with 400 time steps for the Los Altos trail and 3000 time steps for the Santa Fe trail. Each test had 1000 individuals and ran for 50 generations. We also ran each test 30 times with different random seeds, to reduce random chance in the results. However, after running these tests, we considered the fact that we should run a test with two copies of the if-food-ahead function (twoif), to see if merely the number of control functions was having an effect on the output, since functions are selected uniformly for random creation. In other words, the "twoif" experiment better compares to the "both" experiment in that they have the same number of similar functions, rather than only one if-like function in "ifonly."

	Santa Fe Trail				Los Altos Trail			
Avg. 'Best' Hits	61.03	66.86	62.07	63.47	113.47	130.20	102.70	125.23
Std. Dev.	10.68	11.09	9.30	12.39	17.59	17.41	16.50	15.94
Max Hits	89	89	89	89	148	156	142	154
Occurrences	1	3	1	3	1	3	1	3
	ifonly	*both*	*bronly*	*twoif*	*ifonly*	*both*	*bronly*	*twoif*

Table 1. Best-of-run artificial ant performance, averaged over 30 distinct runs.

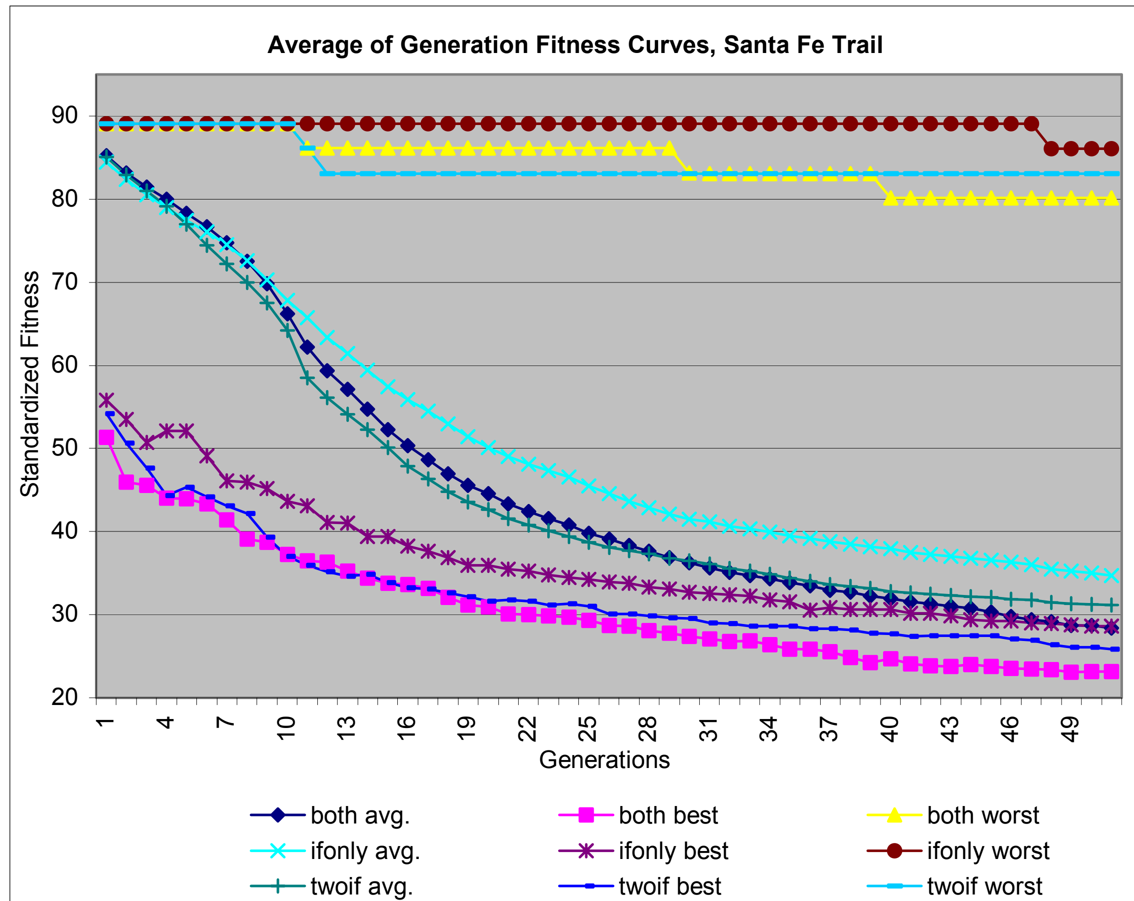

Chart 1. Fitness curves for Santa Fe artificial ant problem averaged over 30 different runs.

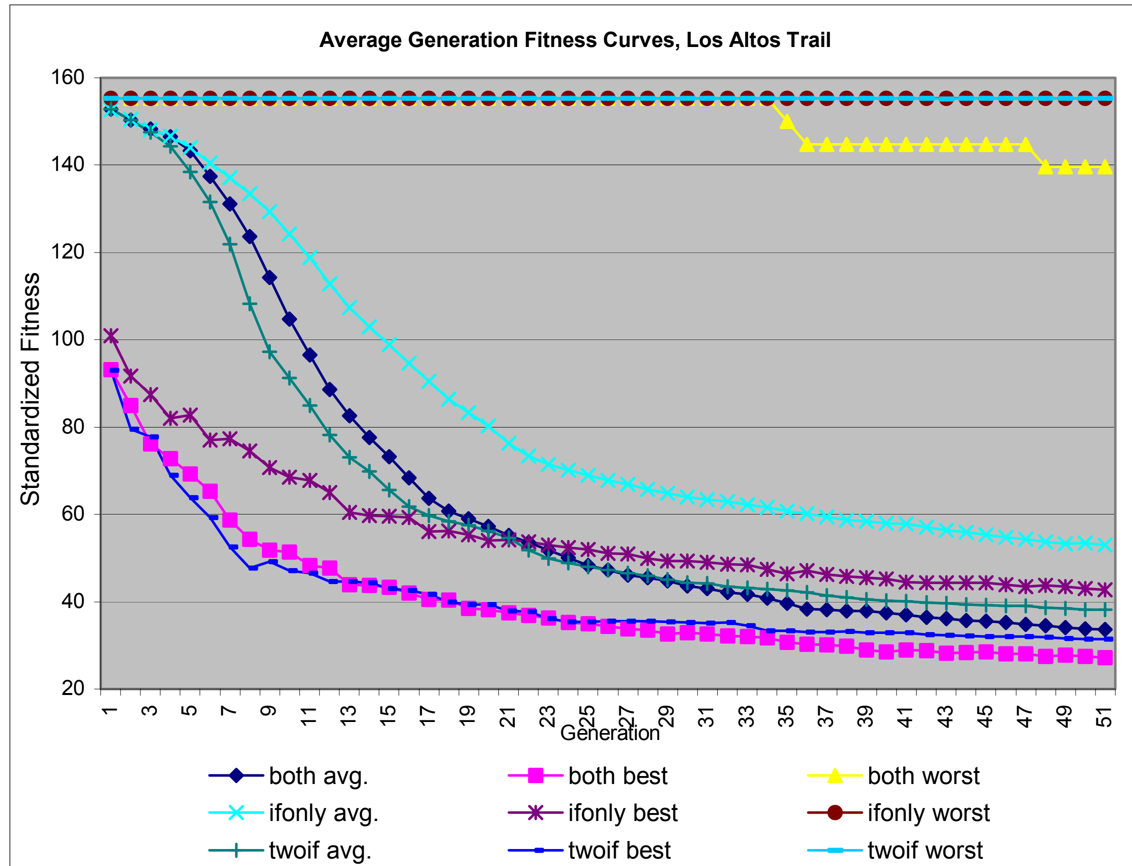

Chart 2. Fitness curves for Los Altos artificial ant problem averaged over 30 different runs.

```
(br-food-ahead 18 (if-food-ahead (progn3 (br-food-ahead 12 move) (progn2(br-
food-ahead 32 (progn3 (progn3 (progn2 left left) (br-food-ahead 61 right)
(progn2 right right)) (if-food-ahead (br-food-ahead 13 (br-food-ahead 16 (br-
food-ahead -4 (progn2 (progn2 (progn2 (progn2 move move) (progn2 left left))
(progn2 right left)) (br-food-ahead 81 (progn3 right move right)))))) (progn2
(progn3 (if-food-ahead (if-food-ahead right right) (if-food-ahead left right))
(br-food-ahead -5 (progn3 right move left)) (progn3 (br-food-ahead -3 move)
(progn3 (if-food-ahead left left)(progn3 left right left) (br-food-ahead 122
right)) (br-food-ahead 56 right))) (progn2 move right))) (if-food-ahead (br-
food-ahead -62 left) (progn2 left left)))) (progn2 move left)) left) (br-food-
ahead 102 (if-food-ahead (progn2 move left) (if-food-ahead (progn3 (progn3 (if-
food-ahead left left) (progn3 left right left) (br-food-ahead -25 right)) (if-
food-ahead (if-food-ahead left right) (progn2 right move)) (progn3 left right
right)) (progn3 (if-food-ahead (progn3 left right left) (progn2 right move))
(if-food-ahead (progn3 (progn3 (if-food-ahead left left) (progn3 left right
left) (br-food-ahead 60 right)) (if-food-ahead (if-food-ahead left right) (br-
food-ahead -13 right)) (progn3 (if-food-ahead left right) (progn3 (if-food-
ahead (progn3 left right left) (progn2 right move)) (if-food-ahead move (if-
food-ahead left right)) (progn2 (if-food-ahead (progn3 left right left) (progn2
right left)) (br-food-ahead 6 (progn3 (if-food-ahead move left) (if-food-ahead
move left) (progn2 right left))))) (br-food-ahead -43 move))) (if-food-ahead
left right)) (progn2 (if-food-ahead (progn3 left right move) (progn2 right
move)) (br-food-ahead 7 (progn3 (progn3 right move right) (if-food-ahead move
left) (progn2 right left))))))))))
```

Listing 1. Example of “both” best of run individual for Los Altos trail. 156 hits, 1754 time units.

Tests were also run on the lawnmower problem. For the lawnmower problem, we investigated runs in the original problem without adfs (nobr), the original problem without adfs and with the br branch (withbr), the original problem with ADFs and without br (adfs), and with ADFs and with brs (bradfs). Tests were run with lawnmower's original settings given in Koza [2], which are an 8x8 board, population 300, with 3 generations. We again averaged performance over 100 runs with distinct random seeds. Then, after looking at these results, we ran all tests again with 3 generations, but a population of 900.

	8x8 Lawnmower, Pop = 300				8x8 Lawnmower, Pop = 900			
Avg. 'Best' Hits	30.11	48.73	60.13	59.68	32.45	57.02	62.63	62.59
Std. Dev.	3.15	9.47	1.55	4.72	3.23	6.91	2.61	2.31
Max Hits	39	64	64	64	42	64	64	64
Occurrences	1	13	2	38	1	37	71	66
	nobr	*withbr*	*adfs*	*bradfs*	*nobr*	*withbr*	*adfs*	*bradfs*

Table 2. Best-of-run lawnmower performance, averaged over 100 distinct runs.

Since only 3 generations were run, no charts were made for the lawnmower problem, as they are uninformative.

```
(frog (prog2 left (vma (vma left (vma left (vma (frog (frog (frog (vma (prog2
(6,0) (3,6)) (frog (5,0)))))) (vma (prog2 (frog (frog (br -12 (4,7)))) (frog
(prog2 (br 7 (vma (prog2 left (0,1)) (vma mow left))) (prog2 mow left)))) (br 3
(br -19 (prog2 (frog (4,3)) (prog2 left (2,3)))))))))) (br -42 (7,6)))))
```

Listing 2. Example of "withbr" best of run individual for lawnmower Population = 300. 64 hits.

Discussion of Results

First we discuss the results of the branch function `br-food-ahead` in the artificial ant problem. Results were somewhat surprising. In our first tests, we neglected to run the "twoif" test, which made the "both" test look more successful, since the next best test, "ifonly," differs from "both" much more significantly. It was this large distance that gave us the clue to run the "twoif" test. Looking in both Table 1 and Charts 1 and 2, we can see that "both," the test with both if and branch statements, still does come out at the leader in all tests. "twoif" is not far behind, but significantly it is always behind "both."

So it seems that the branch construct offers a gain in performance to the artificial ant problem. That gain is quite significant if you compare it to the original artificial ant problem ("ifonly"), but is smaller when compared to the "twoif" version, which just has two if statements instead of one. However, even though this gain is minor, since it is averaged over 30 random runs of the problem, it is likely to be a real and not imagined gain in performance.

However, in one other significant way the branch construct benefits the performance of the artificial ant problem. In our tests, for the Los Altos trail, none of the non-branching tests ("twoif," "ifonly") were able to find a perfect solution to the problem. The "both" version, though, with both ifs and branches, was able to find 3 instances of the perfect solution. Hence, if performance is based on the likelihood of finding a perfect solution, given the constraints, then the "both" version with branching would seem to offer some real improvement to the search, as no other method found a perfect candidate.

Second we look to the lawnmower problem with branch-always or br branching. The results here are quite interesting. We first note that the "withbr" version outperforms the standard no branching, no ADF version

“nobr” by a significant amount, 48 to 30 in the first test, and 57 to 32 in the second test. So blind branching or “jumping” here is offering a definite improvement in performance to the lawnmower problem.

However, we notice that in both cases, the ADF version “adfs” outperforms the “withbr” version. So ADFs outperform blind branching. What is very interesting, however, and is the point behind the Population = 900 test, is that as we increase the population, the blind branching version seems to approach the performance of the ADF version. Notice that when Population = 300, blind branching vs. ADFs is 48 to 60. But when Population triples to 900, blind branching vs. ADFs becomes 57 to 62, much closer together, and both approaching the maximum possible value of 64 hits. So it seems that blind branching, as the population grows, can approximate the ADF in certain situations. If so, in situations where that hold true, it would seem to suggest that ADFs are just being used to allow repetitive actions, and not being used as “functions” at all. Another important fact to keep in mind is that in the ADF version of this problem, the search space has in some ways grown to three times the normal amount, because the three distinct ADF trees (ADF0, ADF1, RFB) are being created and crossed over, rather than the single non-ADF tree. This is not the same as tripling the population, but the larger number of trees inside the ADF version would further support the idea that the ADFs are successful for this problem by simply allowing more repetitive actions, since an ADF tree for this problem has, on the average, more unique nodes to “jump around” to.

The other important thing to notice from the lawnmower results is that all the branching versions seem to have high incidents of the perfect solution, like we noticed in the artificial ant problem. It is hard to say from the limited data available, but it seems that the higher degree of randomness in the branching functions (each of which can point to anywhere) lends itself to better distribution in the search space, thus increasing the probability of finding perfect solutions, even if the average value is lower. The previous statement is mostly conjecture, but it is important to note that there does seem to be a weakly increased correlation between number of perfect solutions found and use of the branching construct.

Certainly, for the lawnmower problem, one clear result seems to be that branching offers a significant performance benefit, and could likely be made to approach the ADF solution.

Conclusion

This paper has demonstrated that a branch operation can be successfully and simply implemented in a genetic programming system. Through testing on two well-known genetic programming problems, the branch construct has offered some minor to mild performance benefits in both problems. The increase is not drastic in either problem, but it does seem to be incremental and hold some promise for further application. In particular, the branching construct seems to be good at approximating ADFs in a simpler, more abstract way, though it obviously lacks some expressive power of ADFs to pass arguments. However, in certain situations, those arguments may not be important or used, and hence branching may be a viable and more reasonable tool to use than ADFs for inducing code reuse.

Future Work

Since the branching operator is a novel concept, there are many interesting experiments that remain to be done to test its performance. The problems selected for this paper were selected mostly for reasons of convenience, and there may be other, more intricate problems that are much more appropriate for the branching operator. One definite area to be investigated is the application of the branching construct to non-simulation genetic programming problems. Also, since the randomness in the branching operator may take a long time to converge, it would be interesting to investigate in a genetic programming problem that takes a large population with many, many iterations to solve. Also, this paper touched on the potential problems of tree depth for the branching operator; experiments with a greater tree depth or possible a greater number of nodes might provide interesting results. Certainly it would also be interesting to further investigate the hypothesis that the ADFs in many problems just provide a level of repetition of random segments, something the branching operator seems more appropriately poised to do. These problems could be converted to using the branch construct and have performance comparisons made. And finally the branching operator might have very interesting effects on problems where there are compact perfect solutions, but they are so rare that a very large population must be used to lead to that proper combination. The branching construct might provide the extra variance needed to boost the speed of these searches.

Acknowledgements

The author would like to gratefully acknowledge the assistance of John Koza, in both Stanford's CS426 class and in meetings about this project. The author would also like to sincerely thank the developers of lil-gp, Bill Punch and Douglas Zongker, for providing the easy-to-use and easy-to-extend genetic programming environment that was used in this paper.

References

[1] Koza, John R. 1992. *Genetic Programming*. Bradford: MIT Press, 1992.
[2] Koza, John R. 1994. *Genetic Programming II*. Bradford: MIT Press, 1994.
[3] *Lil-gp Website*. http://garage.cps.msu.edu/software/lil-gp/lilgp-index.html