CGP Visits the Santa Fe Trail – Effects of Heuristics on GP

Cezary Z Janikow
Dept. of Math and Computer Science
University of Missouri – St. Louis
St. Louis, MO 63121 USA
janikow@umsl.edu

Christopher J Mann
Dept. of Math and Computer Science
University of Missouri – St. Louis
St. Louis, MO 63121 USA
cjm5e8@umsl.edu

ABSTRACT
GP uses trees to represent chromosomes. The user defines the representation space by defining the set of functions and terminals to label the nodes in the trees, and GP searches the space. Previous research and experimentation show that the choice of the function/terminal set, choice of the initial population, and some other explicit and implicit "design" factors have great influence on both the quality and the speed of the evolution. Such heuristics are valuable simply because they improve GP's performance, or because they enforce some desired properties on the solutions. In this paper, we evaluate the effect of heuristics on GP solving the Santa Fe trail. We concentrate on improving the solution quality, but we also look at efficiency. Various heuristics are tried and mixed by hand, while evaluated with the help of the CGP system. Results show that some heuristics result in very substantial performance improvements, that complex heuristics are usually not decomposable, and that the heuristics generalize to apply to other similar problems, but the applicability reduces with the complexity of the heuristics and the dissimilarity of the new problem to the old one. We also compare such user-mixed heuristics with those generated by the ACGP system which automatically extracts heuristics improving GP performance.

Categories and Subject Descriptors
I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

I.2.6 [**Artificial Intelligence**]: Learning

General Terms
Design, Experimentation.

Keywords
Evolutionary Computation, Genetic Programming, Heuristics.

1. INTRODUCTION
Genetic Programming (GP), proposed by Koza [9], is an *evolutionary algorithm* utilizing a population of solutions evolving under limited resources. The solutions, or *chromosomes*, are evaluated by a problem-specific, user-defined evaluation method assessing their *fitness*. They compete for survival based on this fitness, and they undergo simulated evolution by means of *crossover* and *mutation* operators. GP differs from other evolutionary methods by using different representation, usually trees, to represent potential solutions to problems. Trees provide a rich representation that is sufficient to represent computer programs, analytical functions, variable length structures, and even computer hardware [1][9]. The user defines the representation space by defining the set of functions and terminals labeling the nodes of the trees. Of course, all of the needed labels must be predefined or solution trees cannot be evolved. Due to this principle of *sufficiency*, and the related *closure* principle, which allows functions and terminals to mix in any arity-consistent fashion [9], the representation space is highly redundant and enlarged.

GP searches this enlarged/redundant representation space, which is much greater than just the solution space. Because of this, obvious questions to be explored are:

- How GP makes its choices as to what subspace the solutions should come from (the *design* question).
- What is the impact of early design decisions on the performance of GP.
- What would be the impact of imposing some design constraints on GP.

Some researchers have already looked at some of these questions. McPhee with Hopper [12], and Burke [2] analyzed the effect of the root node selection on GP. Hall and Soule [4] have performed even more extensive study of this phenomenon. They concluded that the choice of the root node has a very highly significant impact on the solutions generated, and that fixing the root node properly amounts to limiting the search space needed to be searched. Supporting this, Daida has shown that later GP generations introduce little variation into the structure of the generated trees [3], indicating that these later generations search a smaller subspace of the search space (close in the *genotype* space). Moreover, Langdon has shown that GP typically searches only a well defined region of the potential search space [11]. Hall and Soule call these phenomena the design evolved by GP, which process in fact resembles *top-down design* strategy [4] – first set your choice on the most general design issues, then continue with the more specific ones.

However, very little has been done to study the effect of imposing specific designs on GP, except for the above mentioned choice of the root nodes. Such decisions are heuristics which, as previously shown for the root node, should have a great impact on the performance of GP when applied globally, or locally across the whole tree structure. In this paper, we study the impact heuristics have on GP, using the Santa Fe trail for experimentation. To aid

GECCO'05, June 25-29, 2005, Washington, DC, USA.

with this study, we use the CGP software package which allows processing various heuristics in GP [5][6][8].

In section 2, we look at the role of heuristics in GP, and the current technologies for processing such constraints, paying special attention to CGP. In section 3, we define the experimental methodology, the Santa Fe trail problem, and the heuristics to be processed. In section 4, we show the results of our experiments, identifying the best heuristics and the improvements in GP performance. We also evaluate the generality of the best heuristics by applying them in other contexts, and we also evaluate the heuristics by comparing them with those extracted by an automated heuristics extraction system ACGP [7].

2. HEURISTICS AND CGP

GP users often have some knowledge of the domain, and thus can either suggest or may want to impose various preferences (heuristics) for the solution trees. Such heuristics can be classified along two dimensions. First, we have *global* heuristics, such as the choice of a specific root node, and we have *local* heuristics, that is preferences regardless of the position in the tree. Second, we have *weak* and *strong* heuristics. Weak heuristics are probabilistic preferences, while strong heuristics are actual constraints. For example, disallowing recursion on the *sin()* function is a strong constraint, while requiring that some terminal *x* labels *sin()*'s argument half the time is an example of weak heuristics. Because of the difficulties of enforcing these heuristics, Koza has proposed the principle of *closure* [9], which allows any arity-consistent labeling, often accomplished through elaborate semantic interpretations. In such a standard GP system, mutation is always generating subtree structures from the uniform distribution space, disregarding anything discovered by GP so far. Crossover does a little better, as it only selects subtrees still maintained in the population, and thus presumed better. However, crossover mixes these subtrees out of context, again using uniform distribution space. Thus, in the standard GP, selective pressure in reproduction is the only driving force, and many unwanted trees may show up and be converged to.

Structure-preserving crossover was introduced as the first attempt to handle some strong constraints [9] (the initial primary intention was to preserve structural constraints imposed by automatic modules ADFs). In the nineties, three independent generic methodologies were developed to allow problem-independent strong heuristics. Montana proposed STGP [13], which uses types to control the way functions and terminals can label local tree structures. For example, if the function *if* requires a Boolean as its first argument, only Boolean-producing functions and terminals would be allowed to label the root of that subtree. Janikow proposed CGP, which originally required the user to explicitly specify allowed and/or disallowed labels in different contexts [5]. These local constraints could be based on types, but also on some problem specific semantics. In v2.1, CGP also added explicit type-based constraints, along with polymorphic functions and weak heuristics [6][7]. Finally, those interested in program induction following specific syntax structure have used similar ideas in CFG-based GP [14].

CGP (Constrained GP) relies on closing the search space in the subspace satisfying the desired strong heuristics while putting more exploratory resources in the subspaces identified by the weak heuristics. The search space is bound by the strong heuristics with the help of all operators, initialization included, which all guarantee valid offspring from valid parents [5]. The weak heuristics are used to adjust the probabilities of labeling nodes in mutation and the probabilities of selecting different subtrees for exchange in crossover. This is done with minimal overhead [5]. The heuristics allowed are only *first-order* local heuristics, that is heuristics on parent-child labeling, and separately global heuristics on labeling the root node.

The heuristics play a very important role in CGP. They not only drive the initialization of the population, but in mutation they allow growing subtrees from a non-uniform space rather than randomly as in GP. In crossover, they allow selecting subtrees, for exchange, in such a way that the resulting offspring satisfy the strong constraints and are also more plausible with respect to the weak constraints.

3. EXPERIMENTAL METHODOLOGY

3.1 Experimental Methodology and Setup

Our aim is to study various types and levels of heuristics and how they affect GP's performance: learning speed and efficiency. We measure the learning speed by tracing the learning curves, that is the quality of the best solution per generation. We measure the efficiency by tracing the average tree size per generation – which determines the evaluation (and thus execution) time.

We only study strong, or non-probabilistic, constraints. We use two kinds of heuristics: reducing the function set, and specific local and global (root) structural heuristics. The experimental approach is to try such heuristics one at a time, select those individually showing good performance, and then to combine them into more complex heuristics. This process is performed by a human.

All results are averages of 10 independent runs, 50 generations, population size 1000, half-and-half initialization of depth 2-8, maximum 1000 nodes and 20 levels, tournament selection of size 7, and crossover, reproduction, and mutation operators with probabilities 0.85, 0.05, and 0.1, respectively. Other parameters use default values as in *lil-gp*. If a particular run solves the problem before generation 50, we stop the run, assume max on the learning curve, and 0 for the average tree complexity - this way the area under the complexity curve is proportional to the amount of work needed to solve the problem.

3.2 The Santa Fe Problem

We use the widely studied Santa Fe trail. As noted before, this problem has been widely studied with prior results indicating the best function choices for the root node. The problem involves a simulated ant traveling through a 32x32 grid while following a food trail. This path is 144 cells long, with 21 turns and 89 pieces of food. The ant begins on the northwest corner of the grid facing east (following Koza [9]) and it is allowed a total of 400 basic actions. The fitness is the number of food pieces consumed.

The terminals include: turn *left*, *right*, and also *move* action. The functions include: *if-food-ahead*, *progn2*, and *progn3*. The binary function *if-food-ahead* tests the position directly ahead (wherever the ant is facing) for food, and if true it performs the first action, otherwise it performs the second action. *Progn2* and *progn3* take

two and three arguments, respectively, and execute them sequentially.

3.3 Reducing Function and Terminal Sets

The first question we look at is whether the whole function and terminal sets are necessary for the best performance: we remove each terminal and function one at a time, and then try various combinations. As to notation, **!R** indicates removal of the *right* terminal, *etc*.

3.4 Constraining Tree Structure

Another question is what function is the most effective at the root, and whether there are any additional constraints or requirements on the placement of functions and terminals as arguments to other functions. This is exactly the "design" issue discussed in [2][4], however here extended to local contexts and not just the root node. We use the same approach as above: we test individual heuristics for those most promising, and then we combine them into more complex heuristics. As to notation, **ifroot** heuristic indicates that *if-food-ahead* should be placed at the root, and **if1p** indicates that the second argument of *if-food-ahead* should use any of the two available *progn* functions, etc.

3.5 Combining Both Classes of Heuristics

In this section, we study the result of combining some of the most promising heuristics from both of the above experiments. That is, we reduce the function/terminal set while also constraining the design. For example, **!Rif0m** indicates the heuristic disallowing *right* turns and forcing *move* as the first argument of every *if-food-ahead* (*move* whenever food is ahead). As another example, **P!P2!P3** indicates that every *progn* function is prohibited from using either of the *progn* as any of its arguments.

3.6 Other Heuristics

Selection of the best complex heuristics is a tedious process of analyzing, testing, deciding, and more testing by a skilled operator. Even though CGP allows the heuristics to be entered without the need for recompilation, it is still a tedious process. Thus, the obvious question is that of evolving the heuristics automatically. Here, we used the system designed to accomplish just this, ACGP, introduced last year [7]. We executed the system in the off-line mode, and then compared the quality of the extracted heuristics with those selected by the human operator.

In all experiments, **Base** refers to GP without any heuristics.

4. EXPERIMENTAL RESULTS

4.1 Reducing Function and Terminal Sets

First, we remove one element at a time. The best learning curves are shown in figure 1, with the corresponding efficiency as in figure 2 (as indicated in section 3.1, the area under the complexity curve is proportional to the number of node evaluations needed to solve the problem). As seen, some heuristics result in improvements, while others detriment learning. Clearly, the learning figure indicates that *progn3* is the most unnecessary function - since *progn2* can do the same more efficiently.

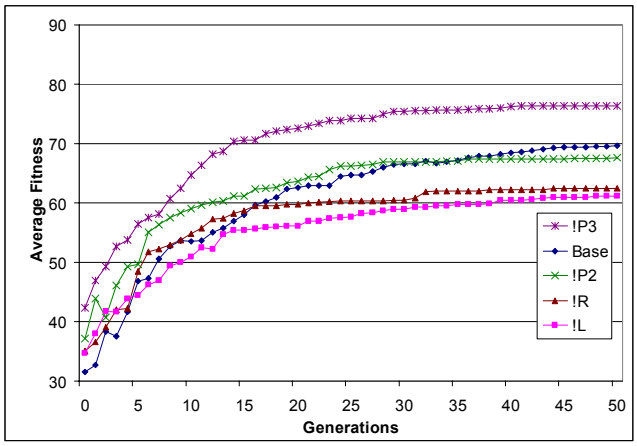

Figure 1. Learning for reduced sets (individually).

This is further supported by the efficiency figure, where solutions without *progn3* are overall of the lowest complexity. It is also interesting to note that only one turn action is sufficient (*left* or *right*); however, because of the necessary additional turns needed, such removals seem to detriment both the learning and the efficiency.

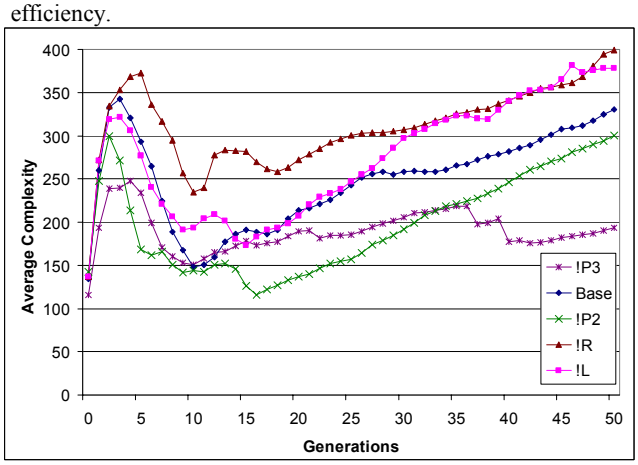

Figure 2. Efficiency for reduced sets (individually).

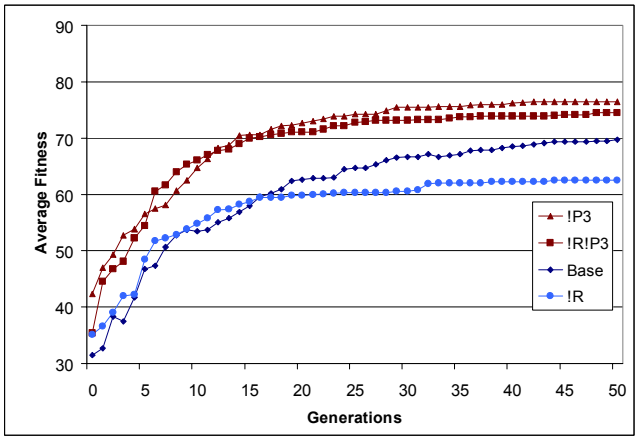

Figure 3. Learning for reduced sets (combined).

Subsequently, we combine the best individual heuristics. Not all combinations are vital. For example, **!R!L** clearly never solves the problem as it would only allow straight-ahead moves. The most interesting results are shown in figure 3 and 4. As seen in figure 3, removing both *progn3* and *right* results in learning indistinguishable from removing *progn3* alone.

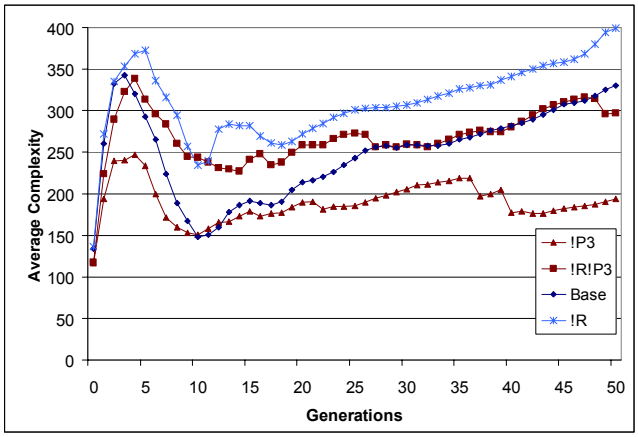

Figure 4. Efficiency for reduced sets (combined).

However, looking at figure 4 we can see that this is done at the expense of about 50% greater work needed (number of nodes processed). Therefore, removing the function *progn3* alone, or heuristic **!P3**, is identified as the best heuristic here.

4.2 Structural Heuristics

First, we evaluate basic structural heuristics individually, including global constraints on the root. The learning curves and the corresponding efficiency for the most interesting basic heuristics are shown in figure 5 and 6, respectively. It is very interesting to see that even though forcing *if-food-ahead* in the root and forcing *move* to be the first action of every *if-food-ahead* hasn't helped a great deal when applied individually, when combined they produce a much more powerful heuristic for improving the learning speed (figure 5).

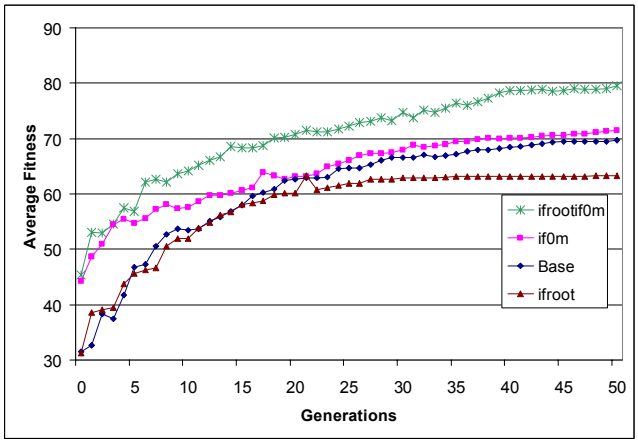

Figure 5. Learning for basic structural heuristics.

What is even more interesting, we can see from figure 6 that while forcing *move* in *if-food-ahead* produces trees double the size, when combined with the other root heuristic the tree size remains in check. This leads to the hypothesis that heuristics are unfortunately non-decomposable. We will see more to support this claim in what follows.

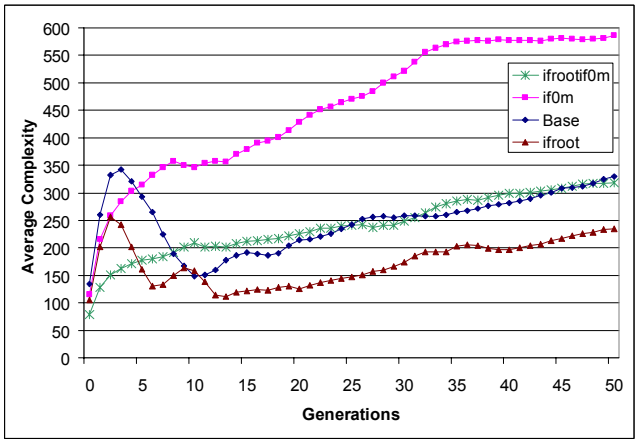

Figure 6 Efficiency for basic structural heuristics.

Subsequently, we combine these and other basic structural heuristics into more complex ones. The results are presented in figure 7 (learning) and 8 (efficiency). As seen, **ifrootif0m** still remains the heuristic producing the best learning speed while doing it relatively efficiently. It is also interesting to observe that forcing the ant to do a sequence of actions if there is no food ahead (**if1p** forces either *progn2* or *progn3* to be the second argument to *if-food-ahead*) reduces the learning speed substantially. This is counterintuitive to our initial beliefs, but upon closer examination we realize that this is beneficial in some nodes but cannot be always required – this is an example where probabilistic heuristics would be useful, a feature available in CGP [6][8] but not used here.

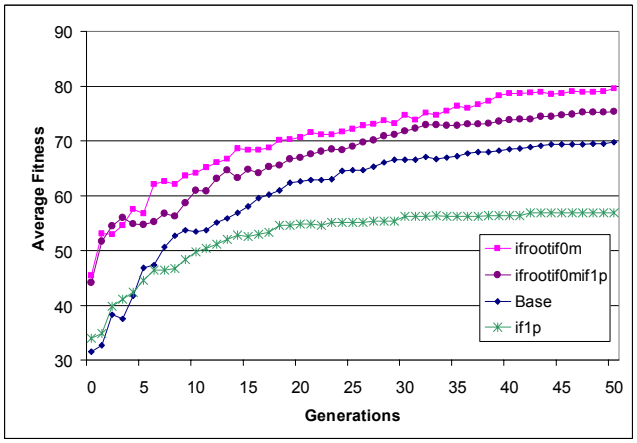

Figure 7. Learning for combined structural heuristics.

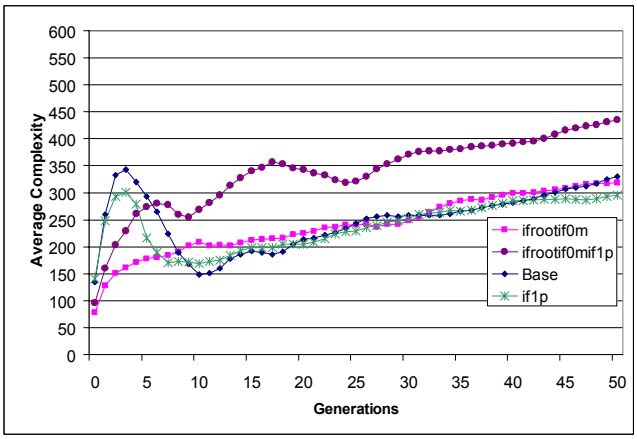

Figure 8. Efficiency for combined structural heuristics.

4.3 Combining Both Kinds of Heuristics

Next, we combine some of the best heuristics identified separately in the two processes above: function/terminal set reduction and structural heuristics. The resulting learning and efficiency curves are presented in figure 9 and 10, respectively.

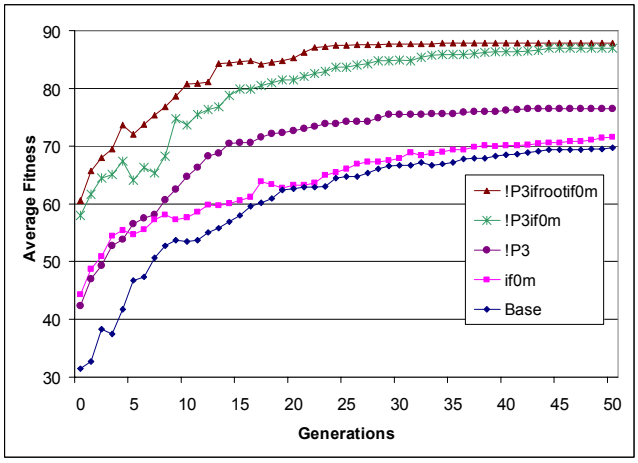

Figure 9. Learning for combinations of heuristics.

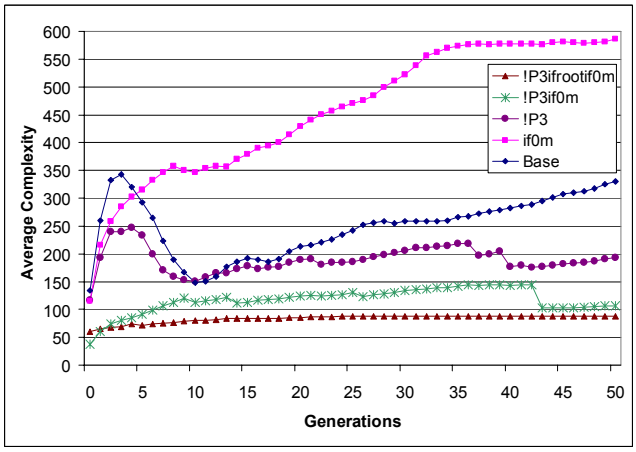

Figure 10. Efficiency for combinations of heuristics.

Figure 9 indicates that combining the best two heuristics from the previous experiments produce the best results **(!P3ifrootif0m)**, with **!P3if0m** coming close behind. Looking at the efficiency chart we observe that these two also reduce the overall time: first, the trees are smaller from the beginning; second, these heuristics allow solving the problem quite often before generation 50.

Then, we test the idea of also removing the *right* turn from the set. As seen previously, this was detrimental when combined with removal of *progn3* alone, but surprisingly when added to the combined heuristic it improves the learning speed (figure 11). Because problems are being solved more rapidly, the efficiency also improves (figure 12). This heuristic, denoted **!R!P3ifrootif0m** prohibits GP from using *right* turn and *progn3* function, while forcing *if-food-ahead* in the root and requiring that the first argument of this function be the *move* action. This further supports the idea that heuristics are not decomposable.

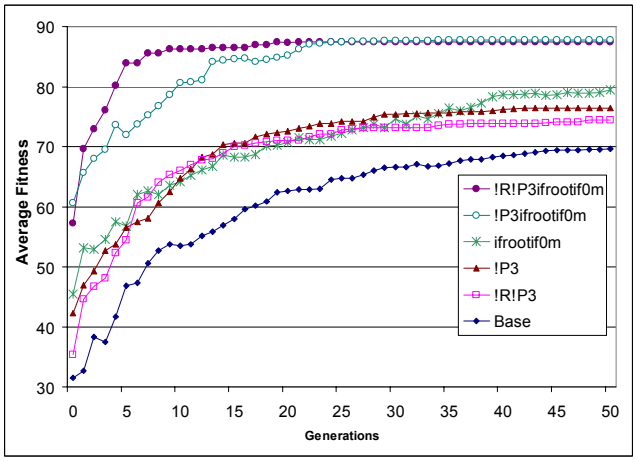

Figure 11. Learning for more combined heuristics.

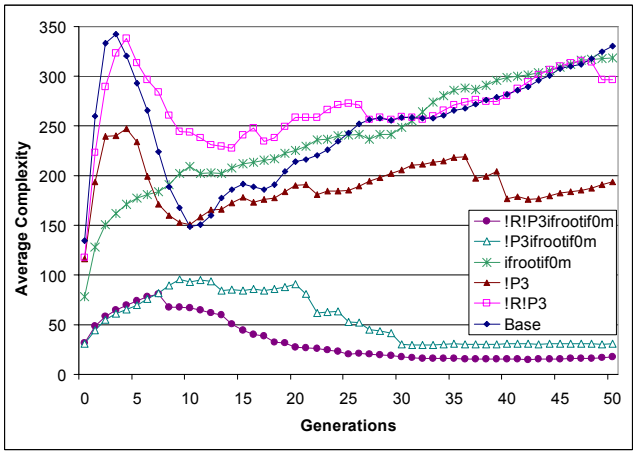

Figure 12. Efficiency for more combined heuristics.

Finally, we analyze the solution trees by hand looking for patterns. We identify four such basic patterns, and we test them individually and in combination. The results are presented in figure 13 and 14. The individual heuristics were: constrain *progn2* and *progn3* so that neither can call neither (**P!P2!P3**), constrain root to always test for food (**ifroot**), constrain *if-food-ahead* to always *move* first if there is food ahead (**if0m**), and disallowing testing for food again if there is no food ahead

(**if1!if**). As seen in these figures (results with previously shown constraints **if0m** and **ifroot** are not shown on this chart), the individual heuristics do not help a great deal, but when used in combination (heuristic **CJM**) they produce the fastest and most effective learning. In fact, the initial population when initialized under the CJM heuristics, contains solutions better than those evolved under no heuristics after 50 generations.

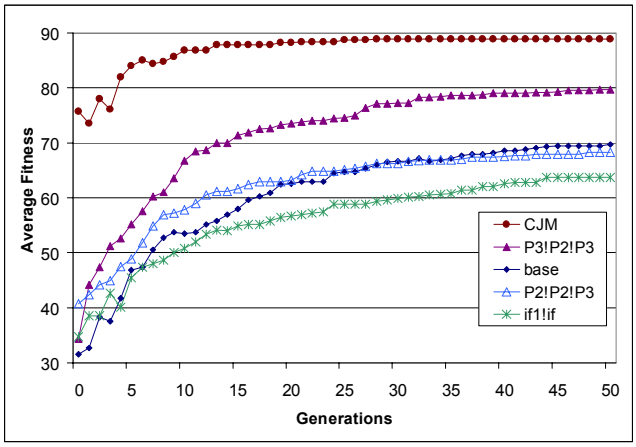

Figure 13. Learning with CJM heuristic.

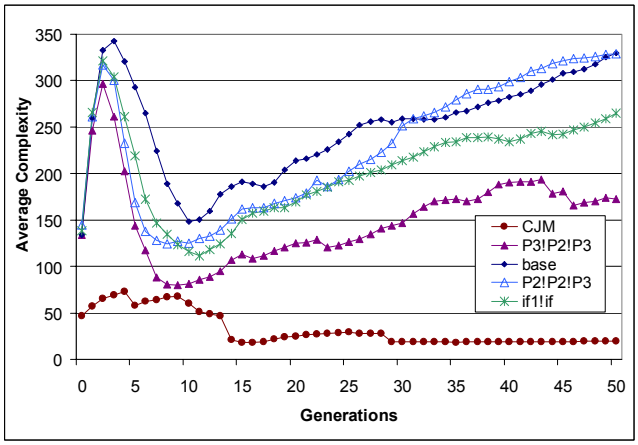

Figure 14. Efficiency with CJM heuristic.

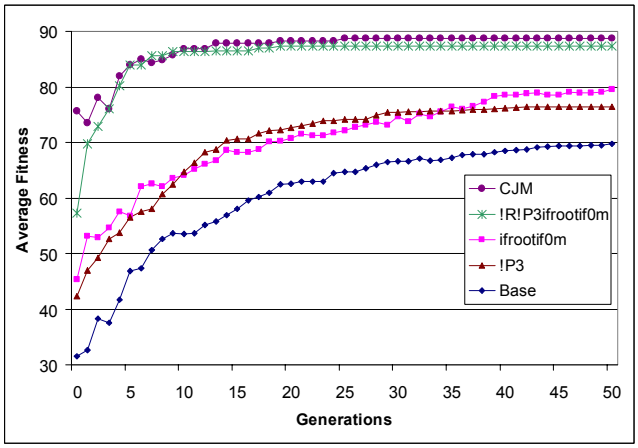

Figure 15. Learning summaries.

Figure 15 summarizes the learning curves of the best heuristics identified in the various experiments so far, while figure 16 summarizes their efficiencies while solving the problem. As seen, the **CJM** complex heuristic is the best, with the less complex **!R!P3ifrootif0m** heuristic nearly as good, both in quality and in efficiency.

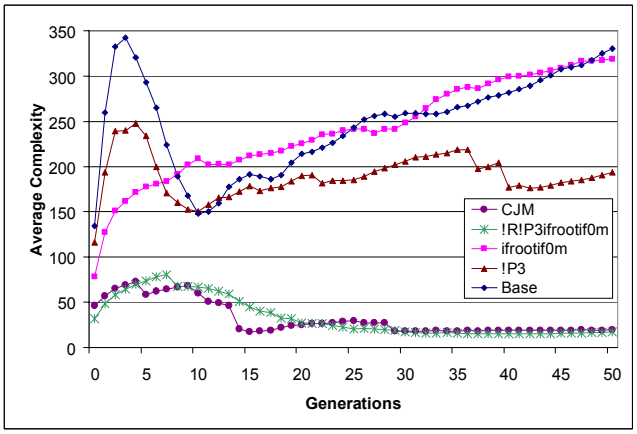

Figure 16. Efficiency summaries.

The best solution discovered had only 13 nodes:

```
(if-food-ahead move (progn3 right (if-food-ahead move (progn3 left left (if-food-ahead move right))) move))
```

4.4 Assessing Generality of the Heuristics

The next question is that of generality of the heuristics – are they specific to this particular trail, or do they extend to other trails as well. This is similar question to that explored by Kuscu [10], yet different. In [10], the generality of the solutions themselves are assessed, while we are assessing the generality of the heuristics - which in turn help shape the solutions. A good analogy for the difference between the two cases would be a screwdriver vs. a tool factory. The question in [10] is how good a screwdriver, designed for one screw, would be to drive a new screw. Our question is how good a factory, designed to produce screwdrivers, is in producing a screwdriver for the new screw.

To answer the question, we use two sets of five new trails. The first set contains five trails with similar characteristics to Santa Fe, that is constructed of the same *basic primitives* [10] just rearranged. The second set contains five trails constructed with the same plus slightly different set of primitive features. Both trails used the same grid and the same number of food pieces. We then repeat the same experiments, with 10 independent runs, using the previously identified best heuristics for the Santa Fe trail. The learning results for the two sets of trails are presented in figure 17 and 18, respectively.

As seen in figure 17, the heuristics still substantially outperform GP without any heuristics, and their relative performance is retained. However, while the magnitude of the resulting improvements is about the same for the two worse heuristics, it is somehow reduced for the more complex heuristics. This illustrates the obvious hypothesis that the more complex the heuristics, the more specific they are to a given problem.

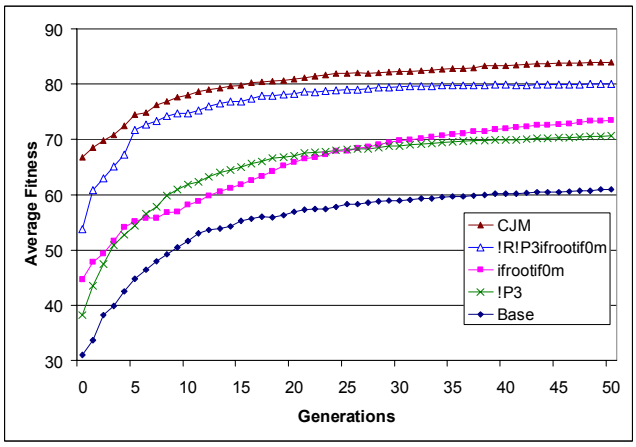

Figure 17. Learning for slightly different trails.

When applied to completely different trails, the same happens to an even greater extent. The overall relative performance of individual heuristics is the same, but they all result in even smaller improvements. Again, the improvements for the more detailed heuristics are reduced more (figure 18).

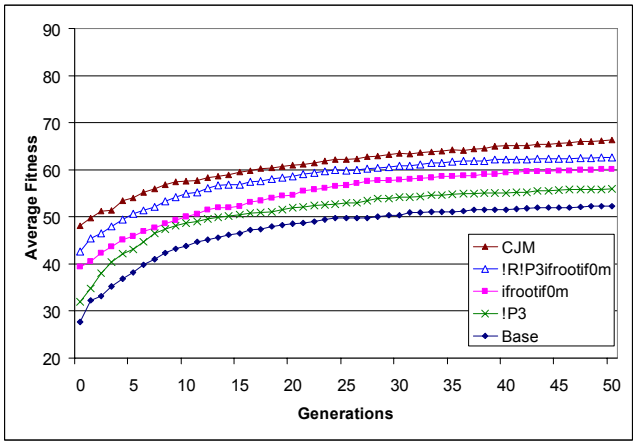

Figure 18. Learning on substantially different trails.

4.5 Comparing with Automatically Extracted Heuristics

In the final experiment, we assess the overall quality of the heuristics used for the Santa Fe trail. To do so, we evolve the best heuristics using the recently introduced system ACGP [7]. The system was executed in the off-line mode, for 500 generations, in nineteen 25-generation long iterations starting after generation 50. At the end of each iteration, the population was reinitialized while using the newly discovered heuristics. For more details, see [7].

The ACGP performance is illustrated in figure 19. As seen, the first 50 generations are indistinguishable from those of the base run, as no heuristics are adjusted until the first iteration terminates. At that time, the population is reinitialized using the newly discovered heuristics, and then the process is repeated every 25 generations. At every reinitialization we can observe a sudden drop in the performance, resulting from the randomly reinitialized population. However, the overall performance improves over time.

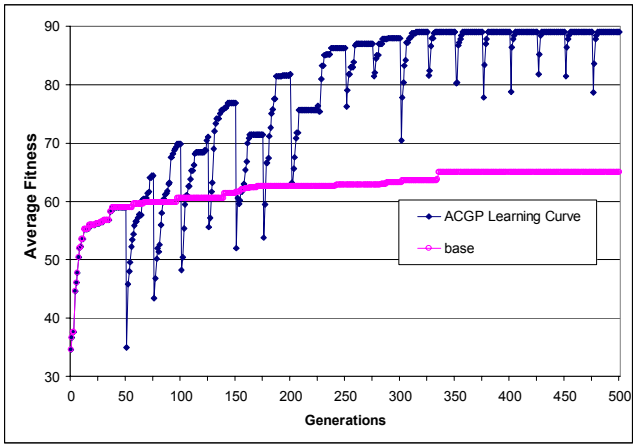

Figure 19. Learning curve in ACGP (off-line mode).

Subsequently, we take the resulting heuristics after 500 generations, and we compare them against the other previously identified heuristics. As we can see in figure 20, the ACGP discovered heuristics clearly outperform our best heuristics, even **CJM**. However, because ACGP discovers probabilistic heuristics, they are not easily comprehensible nor directly comparable (ACGP can be forced to discover strong heuristics).

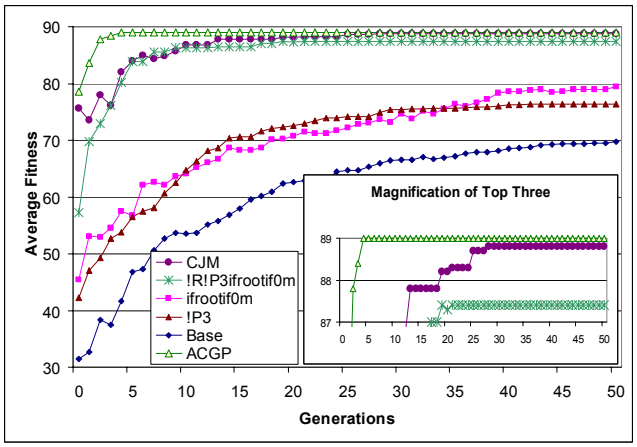

Figure 20. Comparing our heuristics against ACGP's.

5. CONCLUSIONS

In this paper, we used an experimental process to evaluate various strong non-probabilistic heuristics for GP running the Santa Fe trail problem. The heuristics included global constraints on labeling the root node and local first-order heuristics on parent-child constraints. We measured the improvements in terms of the quality of the generated solutions, but we also evaluated the learning efficiency by looking at the average number of nodes being processed. As seen, some heuristics improved GP's performance substantially. However, the more complex heuristics are not necessarily decomposable, that is a given heuristic can be beneficial even if their basic components are not, and vice-versa.

We then evaluated the resulting best heuristics on different trails, assessing their generality. We concluded that the heuristics are still beneficial, but the level of improvements diminishes with the

specialization of the heuristics and the dissimilarity of the new problem to the original one.

Finally, we used the ACGP system to generate weak heuristics and we compared those against our human-discovered strong heuristics. As seen, ACGP-generated weak heuristics outperformed our best human-generated strong heuristics.

REFERENCES

[1] Banzhaf, Wolfgang, Nordin, Peter, Keller, Robert E., and Francone, Frank D. *Genetic Programming*. Morgan Kaufmann 1998.

[2] Burke, Edmund, Gustafson, Steven, and Kendall, Graham. A survey and analysis of diversity measures in genetic programming. In Langdon, W., Cantu-Paz, E. Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M., Schultz, A., Miller, J., Burke, E. and Jonoska, N., editors. *GECCO2002: Proceedings of the Genetic and Evolutionary Computation Conference*, 716-723, New York. Morgan Kaufmann.

[3] Daida, Jason, Hills, Adam, Ward, David, and Long, Stephen. Visualizing tree structures in genetic programming. In Cantu-Paz, E., Foster, J., Deb, K., Davis, D., Roy, R., O'Reilly, U., Beyer, H., Standish, R., Kendall, G., Wilson, S., Harman, M., Wegener, J., Dasgupta, D., Potter, M., Schultz, A., Dowsland, K., Jonoska, N., and Miller, J., editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of LNCS, 1652-1664, Chicago. Springer Verlag.

[4] Hall, John M. and Soule, Terence. Does Genetic Programming Inherently Adopt Structured Design Techniques? In O'Reilly, Una-May, Yu, Tina, and Riolo, Rick L., editors. *Genetic Programming Theory and Practice (II)*. Springer, New York, NY, 2005, 159-174.

[5] Janikow, Cezary Z. A Methodology for Processing Problem Constraints in Genetic Programming. *Computers and Mathematics with Applications*, 32(8):97-113, 1996.

[6] Janikow, Cezary Z. and DeWeese, Scott. Processing Constraints in GP with CGP2.1. In *Proceedings of the GP 1998 International Conference*, 173-180.

[7] Janikow, Cezary Z. ACGP: Adaptable Constrained Genetic Programming. In O'Reilly, Una-May, Yu, Tina, and Riolo, Rick L., editors. *Genetic Programming Theory and Practice (II)*. Springer, New York, NY, 2005, 191-206.

[8] Janikow, Cezary Z. *CGP*. http://www.cs.umsl.edu/~janikow/CGP.

[9] Koza, John R. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[10] Kuscu, Ibrahim. Evolving a Generalized Behavior: Artificial Ant Problem Revisited. In Porto, V.W., Saravanan, N., Waagen, D., and Eiben, A.E., editors, *Evolutionary Programming VII 1998*, pages 799-808, San Diego, California, Mar. 1998.

[11] Langon, William. Quadratic bloat in genetic programming. In Whitley, D., Goldberg, D., Cantu-Paz, E., Spector, L., Parmee, I., and Beyer, H-G., editors, *Proceedings of the Genetic and Evolutionary Conference GECCO 2000*, 451-458, Las Vegas. Morgan Kaufmann.

[12] McPhee, Nicholas F. and Hopper, Nicholas J. Analysis of genetic diversity through population history. In Banzhaf, W., Daida, J., Eiben, A. Garzon, M. Honavar, V., Jakiela, M. and Smith, R., editors *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1112-1120, Orlando, Florida, USA. Morgan Kaufmann.

[13] Montana, David J. Strongly Typed Genetic Programming. *Evolutionary Computation*, 3(2):199-230, 1995.

[14] Whigham, P. A. Grammatically-based Genetic Programming. In Rosca, Justinian P., editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33-41, Tahoe City, California, 9 July 1995.