in *Proceedings of the 1993 International Conference on Neural Networks*, New York, NY: IEEE Press

Evolving a Sort: Lessons in Genetic Programming

Kenneth E. Kinnear, Jr.

Adaptive Computing Technology
62 Picnic Rd.
Boxboro, MA 01719 USA
kim.kinnear@adapt.com

Abstract—In applying the Genetic Programming paradigm to the task of evolving iterative sorting algorithms, a variety of interesting lessons were learned. With proper selection of the primitives, sorting algorithms were evolved that are both general and non-trivial. The sorting problem was then used as a testbed to evaluate the value of several alternative parameters, with some small gains shown. The value of applying Steady State Genetic Algorithm techniques to Genetic Programming, called Steady State Genetic Programming is demonstrated. One unusual genetic operator was created, non-fitness single crossover, which shows promise in at least this environment.

I. Introduction

Trying to evolve even as relatively simple an algorithm as a sorting routine can teach a number of useful lessons concerning the power as well as the pitfalls of using genetic techniques to directly evolve computer programs.

A. Genetic Programming

Programming paradigms that exploit evolutionary techniques are becoming more common. The techniques used in the work described here derive from Genetic Algorithms (GA) developed by John Holland [7]. Genetic Algorithms typically work on fixed length, linear representations of genetic material, and operate on this material with fitness proportionate selection, reproduction, crossover, and mutation [3, 4]. John Koza has extended the work done by Holland and many others in the GA field by using analogies of GA genetic operators directly on tree structured programs (typically LISP functions) and calls this approach Genetic Programming (GP) [8-11]. GP differs from traditional GA in that it uses hierarchical genetic material, and this genetic material is also not of fixed size. While GP doesn't mimic nature as closely as does GA, it does offer the opportunity to directly evolve programs of unusual complexity, without having to define the structure or the size of the program or genetic material in advance.

Others have described GA approaches that operate on variable length or program structured genomes [5,14]. These approaches typically require more constraints on the form of the final solution than does GP.

The work described here uses GP as defined by Koza in [9, 11] as well as a further elaboration of GP developed by Craig Reynolds called Steady State GP (SSGP) [13]. Reynolds developed SSGP through reference to the work of Gilbert Syswerda on Steady State Genetic Algorithms [15]. Steady State GA is also discussed in some depth by Lawrence Davis in [2], and an early description by Darrell Whitley appears in [16]. SSGP is described in more detail along with comparative results for the sorting problem in Section V: Genetic Operations.

B. Why Sorting?

I was quite impressed by the startling generality of the GP approach as reported by Koza in [9], and wanted to apply it to a different problem area to gain experience with GP. Sorting was interesting for a variety of reasons. Sorting appeared to be a problem that was at least as "difficult" as many of the problems already tackled by Koza. In addition, sorting requires fairly complex control structures and operates on an infinite data domain. Many reasonable and quite different solutions exist for sorting, some of them quite concise. Sorting algorithms generally have a strongly algorithmic flavor, and many of them can be expressed either iteratively or recursively. Fundamentally, sorting seemed to contain in a microcosm many of the elements and pitfalls that would present themselves when trying to apply GP to problems of even greater complexity and practical utility.

Interesting work applying genetic techniques to sorting networks has been done by W. Daniel Hillis [6]. In contrast to Hillis, who evolved sorting networks for both correctness and efficiency of sorting, the work described here is centered around evolving correct sorting algorithms as often and as quickly as possible.

C. Results

The results described fall into two rough categories. The first consists of observations which result from applying GP to a new problem area. After reading Koza's Stanford report [9]. I rather expected to be able to write the Common LISP code and then immediately evolve a sort. I learned many things in simply trying to get my GP system to evolve even one sorting algorithm. The observations I will make about what I learned during this phase of the work are anecdotal in nature.

After simplifying the problem sufficiently, general sorting algorithms of non-trivial complexity were evolved.

Several variations in the GP environment were then examined in an attempt to improve its ability to more reliably evolve correct sorting algorithms (with the goal of applying GP to more difficult sorting problems as well as other difficult problems in the future). These results are quantitative comparisons of various approaches used to evolve sorting algorithms more or less frequently, or more or less quickly.

II. Context for Sorting

The basic approach to GP is to generate a random population of individuals, evaluate their fitness, perform various genetic operations on them based in some way on their fitness, and then go back and evaluate the results again. In this situation, the individuals of the population are LISP functions whose only interesting results are their side effects. The side effect desired is to reorder a sequence of integers so as to leave it in "sorted" order, small to large. Primitives are defined to compare and swap the various values in the sequences.

To evaluate the fitness of an individual, it is presented with an "unsorted" sequence and then executed. The "disorder" of the sequence is measured before and after execution, and the fitness is based on the decrease in disorder. Typically, an individual is presented with 15 or more such sequences to evaluate its fitness for each generation.

A. *Primitives for Sorting*

In order to apply GP to a problem, it must be cast into a form that can be evolved. A set of primitives consisting of functions and terminals must be created that is sufficient to solve the problem. Terminals are the constants or variables.

A variety of primitives were tried in order to successfully evolve a general and yet non-trivial sorting algorithm. The primitives described here are the least complex, and therefore most realistic that would reliably allow evolution of general sorting algorithms within a reasonable time. O'Reilly discusses primitive selection and the impact on difficulty in [12].

In this primitive set, 7 functions and 2 terminals are available. The two terminals are `index`, an iterator variable which takes on various sequence index values when included in the `work` of a `dobl`, or 0 otherwise, and `*len*`, which contains the length of the sequence under test. The functions are:

```
(dobl start end work)      (wismaller x y)
(swap x y)                 (wibigger x y)
(e1+ x)                    (e- x y)
(e1- x)
```

Briefly, `dobl` is an iterative operator which takes a starting index `start` into the sequence under test and increments the index variable `index` by 1 until either `end-1` or `*len*-1` is reached, executing the `work` once for each iteration. The number of iterations that `dobl` will perform is restricted. Each occurrence of `dobl` will not iterate more than 200 times, and the sum of all `dobl` iterations within a single test will not exceed 2000. `dobl` returns `(min end *len*)`. `swap` exchanges the values at indices `x` and `y` in the sequence and returns `x`. `wismaller` and `wibigger` compare the values at the indices `x` and `y` and return the index of the smaller or larger, respectively. `swap`, `wismaller` and `wibigger` return `0` if either argument is non–numeric, `< 0`, or `>= *len*`. The three functions `e1+`, `e1-` and `e-` are protected versions of the standard Common LISP functions. They will return `0` and not cause an error if the arguments are non–numeric.

One thing worth noting is that this function and terminal set will not allow the individuals in the population to modify the data in any way — they can only change the order.

I learned that the return value of every function is important, because in the initial random population as well as in "successful" sorts, the functions are not combined in a way that a human would ever expect. Thus, defined and "reasonable" return values (in both data type and value) are important. These return values are important even in error cases, because the error cases may well outnumber the non-error cases even in the results of successful runs. For example, `(swap x y)` originally returned `nil` when either `x` or `y` was out of range of the sequence under test, usually causing the evaluation to terminate soon after with a fatal error. Now it returns `0`. The result of these changes is to allow the existence of effectively dormant but non-fatal genetic material.

B. *An Evolved Sort*

The example shown below is the most concise of the sorts evolved in the runs described here. In its unsimplified version, it has a total of 42 functions and terminals. The next smallest has 78. Typically they are in the 150 to 300 range.

```
(dobl (wismaller (wismaller (e1- *len*) *len*)
                 index)
      (dobl (wismaller index
                       (wismaller
                        (e1- index)
                        (e1+ (e1- index))))
            (e1- *len*)
            (swap (swap (e1- *len*) index)
                  index))
      (dobl (swap (wibigger index (e- index *len*))
                  (e- index *len*))
            (e1- *len*)
            (swap (wismaller (e1+ index) index)
                  index))))
```

This can be hand simplified to the following completely correct and minimal expression:

```
(dobl 0
      (e1- *len*)
      (dobl 0
            (e1- *len*)
            (swap (wismaller (e1+ index) index)
                  index))))
```

C. Generation of Initial Population

The initial population for each run is a random function tree of depth less than or equal to 6, generated by uniform selection over the list of functions (and terminals for the terminal nodes of the tree). A population size of 1000 was used for all runs described here. In contrast to standard practice in [11], uniqueness was not required in the initial population.

The likelihood of success seems dependent on the content of the initial population, which in turn is linked to the population size. Population size affects continuing diversity during the run as well.

III. Comparison Methodology

In an attempt to evolve *any* general sorting algorithm, I spent an unfortunate amount of time trying different strategies for one or two runs, and then comparing them with previous trials of one or two runs. This led me down several false paths, as it takes many runs with the same strategy to be able to evaluate it in comparison to some other strategy. For the work described here, 20 runs of a maximum of 49 generations (or generation equivalents for SSGP) were performed. For several of the more interesting cases, 40 runs were performed. These sets of an additional 20 runs essentially duplicated the results of the previous 20.

A. Success and Generality

The primary goal for each run is production of at least one individual that has at least once correctly sorted all of the test sequences presented to it during a fitness evaluation (typically 15). The run is terminated 5 generations (for GP) and 2 generation equivalents (for SSGP) after this event (termed success) occurs.

At that time, the best individual is heavily tested with 300 sequences of maximum length 40, and all 256 sequences of length 8 consisting of 0 and 1. These tests require much greater generality than the fitness tests used during the runs, but of course are not tests of full generality.

For any algorithm (such as sorting) that operates on an infinite domain of data, no amount of testing can ever establish generality. Testing can only increase confidence.

While many of the evolved sorting algorithms that successfully sort all of the fitness tests fail to pass the more stringent post–run tests, no algorithm that has passed all of the post–run tests has ever been shown to fail on any sequence presented. Many of these evolved and potentially general sorting algorithms have subsequently been tested with thousands of random sequences with not a single failure.

Therefore, while many of these algorithms are too complicated to hand analyze and therefore prove fully general, the post–run generality tests can be presumed to be reasonable indicators of generality.

B. Description of the Tables

Results of the experiments are presented in several tables. Experiments consist of sets of runs compared to each other. Each set of runs consists of 20 runs, where a run uses 1000 individuals and processes them for up to 49 generations (or generation equivalents). The significant conditions of the experiment are to the left of the vertical double line, and the results are to the right.

The principal metric for each set of 20 runs is the number of runs that were completely successful in evolving at least one individual that correctly sorted the fitness tests presented to it (removing 100% of the disorder in the sequences), and this is recorded in the 100% column under # SUCCESSFUL RUNS. Additional information is presented as to how many runs produced at least one individual which removed 90% of the disorder in the fitness tests and 75% of the disorder in the fitness tests. The 90% and 75% categories are cumulative, in that each includes the count of runs that did better as well as the count of runs that made it past the 90% or 75% boundary but didn't reach the next highest.

The column # GEN RUNS indicates for how many of the 100% successful runs the single individual tested for generality passed all of the post–run generality tests. This understates the run's true generality, since a sample of only one 100% successful individual is tested from each 100% successful run.

AVG INDS 100% records the average number of individuals that had to be processed for the 100% successful runs (averaged only over those runs that reached 100%).

Thus, the number of 100% SUCCESSFUL RUNS indicates how effectively the parameters used by a particular set of runs evolved a sort. The # GEN RUNS give some indication of the generality of the resulting sorts. The AVG INDS 100% indicates how quickly the 100% SUCCESSFUL RUNS reached 100%.

IV. Fitness Test Selection

There are special problems associated with using genetic techniques on problems where all of the potential test data cannot be used for evaluating fitness. These problems stem from the very power of genetic techniques, in that they will exploit every loophole available. GP will evolve a function to solve the problem presented to it by the various fitness tests, and if these fitness tests are not a good representation of the problem, then the resulting individuals will not do well on the problem either. An extreme case for sorting would be if each individual were presented with one sequence of randomly ordered numbers for the entire run. Very quickly an individual will evolve which will correctly order this single sequence, but there is little chance that it will be in fact a fully general sorting algorithm.

In addition to the various test strategies used below, many other possibilities exist. Hillis did some fascinating work in co-evolving the test sequences to use for his sorting networks while evolving the sorting networks themselves [6].

A. *Random Length Sequences of Random Integers*

Fitness test sequences with a length determined by a uniform random distribution from 1 up to a maximum length of 30 were used for the majority of the runs. The numbers in these sequences were selected through a uniform random distribution from 0 through twice the allowed maximum length (not the actual length of this particular sequence).

Two approaches were used for these random tests -- the first, where the test set was regenerated at each generation or generation equivalent, and the second where the test set was not changed for the entire run.

Different numbers and lengths of tests were used. In the baseline sets of runs, 15 random tests of a random length not exceeding 30 were used (Sets A and B) and are compared in Table 1, below, with sets of runs with 25 tests with a random length not exceeding 30 (Set C), and sets with 15 tests with a random length not exceeding 50 (Set D). Little difference is noted although the two sets with fewer total numbers to sort (Sets A and B) produced more successful runs with slightly lower generality than the sets of runs with more total numbers to sort (Sets C and D). The runs with the same tests for an entire run (Set E) showed moderately lower success and generality. It is interesting that this set produced any general runs at all with only 15 test sequences.

V. Genetic Operations

In the drive to increase the likelihood that a given run would evolve a general sorting algorithm, several strategic changes in the form of the genetic operations were investigated, and are described below. Steady state GP, as described by Reynolds [13], was implemented and showed improvement. Several modifications to batch GP were necessary to implement SSGP, and the incremental improvement of each was investigated. Then both of the batch approaches were compared to SSGP. Finally, a different form of crossover was created which yielded good results in the sorting problem described here.

In both batch and steady state GP, an initial population is created totally at random from the primitive set. Then, in batch oriented GP, all the individuals are evaluated for fitness, and based on these fitness calculations a new generation of individuals is created by a variety of genetic operations. This process is then repeated.

In steady state GP, after the initial population is randomly created and evaluated for fitness, a single new individual is created by various genetic operations and, if it is not currently present in the population, evaluated for fitness. It is then merged into the population. A single individual is then selected for removal, based here on the inverse fitness, i.e. the fitness (which is a probability) subtracted from 1. Then another new individual is created, evaluated, and merged into the population. 1000 of these single individual operations are considered a "generation equivalent".

In the discussions below, the term "fitness proportional" selection is used frequently. Fitness proportional (FP) selection is simply the selection of an individual from the population with a probability based on its fitness. In this work, like that described by Koza in [9], the calculated fitness is used as the probability of selection.

In all of the descriptions below whenever "a point is selected inside of a function," a function point is selected 90% of the time and a terminal point 10% of the time. Both function and terminal points are chosen with uniform random selection, following [9].

Table 1: Fitness Test Selection

Set	Significant Feature(s) (all are non-fit single crossover)	# tsts	max tst len	# GEN RUNS	# SUCCESSFUL RUNS (out of 20) 100%	>90%	>75%	AVG INDS 100%
A	SSGP, baseline	15	30	4	**12**	12	12	25050
B	SSGP, baseline	15	30	3	**10**	11	13	23095
C	SSGP, more tests	25	30	5	**8**	8	9	20451
D	SSGP, longer tests	15	50	5	**8**	9	10	14830
E	SSGP, no new tests each generation	15	30	3	**5**	8	8	7173

A. Batch Oriented Genetic Operators

The initial runs were performed with a batch oriented GP environment as described above, as a baseline of comparison. The operators used in these runs to create a new generation are as follows:

- ***Reproduction*** uses FP selection to choose an individual to be copied into the next generation.
- ***Mutual crossover*** uses FP selection to choose two individuals, and a crossover point is selected within a copy of each. The subtrees from each crossover point are swapped, creating two new individuals. The crossover points are chosen (and re-chosen when necessary) so that neither of the new individuals is more than 180% of the maximum allowable initial depth.
- ***Uniform Mutation*** operates by selecting an individual by uniform random selection, selecting a point within the function, and then generating a random tree to replace that of the selected point. Mutation allows a particular individual to grow by no more than 15% in depth.

In batch GP as described above, the number of individuals created by each type of genetic operation was predetermined. Exactly 10% of the new population was created by reproduction, and exactly 90% was created by mutual crossover. Exactly 10% of the resulting new population were altered by uniform mutation. This is in contrast to the probabilistic reproduction strategies described in the next section.

B. Alternative Genetic Operators

In an SSGP system only one individual is created at a time. The batch approach to generating an entire population for each generation needs to be altered in order to generate only one individual at a time. The predetermined reproductive mix of the batch system needs to be changed to a probabilistic reproductive mix. In this approach, at each opportunity to produce a new individual, the actual genetic operator is chosen based on a probabilistic strategy. The alternative operators are:

- ***Single crossover*** produces one new individual, in contrast to mutual crossover, described above. As in mutual crossover two individuals are chosen for crossover, and crossover points are chosen inside of them. However, the crossover is performed by placing a copy of the subtree selected in one into the place of the subtree selected in a copy of the other. This becomes the only new individual. In addition to producing only one new individual, another difference is that while reselection of the crossover points within the individuals is still required to meet depth limitations on the resulting individual, it is possible for the result of single crossover to be significantly smaller than is likely for both individuals resulting from mutual crossover. In mutual crossover, if one of the resulting individuals were to be very much smaller the other would have to be very much larger, and this would be prevented by the overall depth limitations imposed on the results of all crossovers.
- ***Hoist*** was created in order to increase the probability even further of the results of genetic operations creating smaller individuals. Hoist uses FP selection to choose an individual, selects a function point inside of the individual, and returns a copy of this sub-tree as the new individual. While single crossover can have the same effect, as the individuals grow the probability of this happening is reduced, while the probability of a hoist operation is set.
- ***Mutation*** creates a new individual by FP selecting an individual, selecting a point within a copy of that individual, and then generating a random tree to replace that of the selected point. Mutation will allow the individual to grow by no more than 15% in depth.
- ***Create*** returns a new, randomly created individual of depth not to exceed 6. This is the same operator used to create the initial population.

The sets of runs shown in Table 2 compare the traditional genetic operators (Set H) and the alternative genetic operators (Set J) in a batch context. For the runs using alternate operators, exactly 10% of the individuals of each generation were created using FP reproduction as in traditional GP, and the remaining 90% of the individuals of each generation were created by probabilistic selection among the alternative operators described above using the following probabilities for each operator: 70% for single crossover, and 10% each for hoist, mutate, and create. The quantitative results in Table 2 show that there was little difference between these approaches.

Table 2: Batch Oriented Genetic Operations

Set	Significant Feature(s)	# tsts	max tst len	# GEN RUNS	# SUCCESSFUL RUNS (out of 20) 100%	>90%	>75%	AVG INDS 100%
H	Batch, Traditional GP	15	50	3	**5**	5	5	25914
J	Batch, Alternate Operators	15	50	3	**7**	8	8	23598

Table 3: Steady State GP

Set	Significant Feature(s)	# tsts	max tst len	# GEN RUNS	# SUCCESSFUL RUNS (out of 20) 100%	>90%	>75%	AVG INDS 100%
F	Batch, Traditional GP	15	30	2	**4**	4	7	25513
G	Batch, Alt. Operators, Unique Repl.	15	30	4	**5**	5	6	30381
K	SSGP	15	30	3	**8**	8	9	17510
L	SSGP	15	30	3	**7**	9	10	17939

C. Steady State GP

The step to steady state GP is now a small one. Probabilistic selection of genetic operators (using the alternate operators described above) is used to generate a single new individual which, if unique, is merged into the population. Then, based on the criteria discussed above, an individual is removed from the population. In this case, there is no direct reproduction (copying) of individuals from one generation to another.

New individuals are created using the alternative operators with the following probabilities: 70% for single crossover, and 10% each for hoist, mutate, and create.

The runs in Table 3 compare traditional batch GP (Set F), and SSGP (Sets K and L). The number of 100% successful runs shows real improvement for SSGP. This is a good example of why many runs are important. In the first trials of SSGP, it seemed to provide little to no value. Only after many runs (which take significantly longer than batch runs) was the value of SSGP apparent

In an effort to determine if the improvement shown in Table 3 for SSGP over traditional batch GP was due to the unique replacement strategy used, a set of runs using the alternate genetic operators in a batch context using unique replacement were made (Set G). They showed little gain over traditional batch GP (Set F). Davis [2: p. 36] contains an interesting discussion on unique replacement in batch and steady state GA.

D. Non-Fitness Single Crossover

All of the approaches discussed above contain a strong requirement for individuals to do well at every test in order to be allowed to participate significantly in production of new individuals. This limits the diversity of the populations and in turn limits the complexity of solutions that are likely to evolve. Of course, there is some likelihood that low fitness individuals will participate in production of new individuals for several generations, but clearly not much. It seemed that there might be merit to allowing not only survival of low fitness individuals (which SSGP does) but also some way to allow some of them to participate in production of new individuals for at least a few generations, free from the high pressures of fitness.

I found a simple scheme to allow this, which I call non-fitness single crossover. In the single crossover scheme described above, each individual is selected based on fitness. Non-fitness single crossover is identical to single crossover, but it uses two individuals selected based on uniform random selection across the entire population. Non-fitness single crossover is not designed to replace single crossover, rather it is designed to augment it.

Much work has been done in GA around types of crossover and selecting the points for crossover [4]. Considerably less has been done on selecting the individuals for crossover. The usual approach seems to reduce to some form of fitness proportional selection, although sometimes it may be rather indirect.

In order to evaluate the value of non-fitness single crossover, runs using only single crossover were made and compared to runs combining both non-fitness single crossover and single crossover. In the single crossover runs, there was a 70% chance of the new individual of being produced by single crossover. In the joint runs, there was a 35% chance of the new individual being produced by non-fitness single crossover and 35% by single crossover.

In the comparisons shown in Table 4, there is clearly an improvement in the sets of runs that include non-fitness single crossover (Sets A and B) over those that do not (Set K and L). The fitness calculations used in this work are from [9], but the raw fitness is developed in such a way as to strongly bias the resulting normalized fitness toward the best individuals in the population. It may be that non-fitness single crossover shows promise in this system because it produces a balance to the very strong fitness pressures present.

A set of runs was made with non-fitness single crossover in batch GP with alternate operators, and it showed no gains over any other batch set with traditional or alternate operators. Apparently it is useful only in SSGP.

E. Permutation and GP

Koza, in [9], introduced a genetic operator called permutation which is an extension and generalization of Holland's linear inversion operator [7] into the domain of tree structured

Table 4: Non-Fitness Single Crossover

Set	Significant Feature(s)	# GEN RUNS	# SUCCESSFUL RUNS (out of 20) 100%	>90%	>75%	AVG INDS 100%
K	SSGP, only single crossover	3	**8**	8	9	17510
L	SSGP, only single crossover	3	**7**	9	10	17939
A	SSGP, both non-fit single and single crossover	4	**12**	12	12	25050
B	SSGP, both non-fit single and single crossover	3	**10**	11	13	23095

genetic material. Permutation, like inversion, reorganizes an individual's genetic material and increases the likelihood that some combinations of traits will survive subsequent applications of the crossover operator, and thus appear in that individuals's offspring. Unlike inversion, however, permutation not only affects the form of the genetic material but also the current function of the individual, since in GP the individual and the genetic material are one and the same. Thus, the fitness following permutation will usually be different, while the fitness following inversion will not. The very traits that caused this individual to be selected for permutation may well be lost as a result of the permutation! Perhaps this is why the permutation operator failed to yield significant gains as reported in [9] and [11]. An interesting problem remains in designing a fitness preserving analogue to inversion for the tree structured genetic material of GP.

VI. Conclusions/Summary

Here is a brief list of the key results reported here:

- This is additional confirmation of GP as a powerful technique, capable of creating solutions of surprising complexity and power, in far fewer generations than intuition would suggest.
- Comparative evaluation of GP techniques requires many sets of runs. (Over 6 million individuals were processed for the results reported in this paper). Lots of what I thought I learned doing one run at a time simply wasn't true. In fact, it is likely that I had the entire ensemble of GP framework and sorting as a problem working long before I knew it, because I simply hadn't run enough trials to realize that it was already capable of evolving a sort. Several runs of 50 generations is simply not enough data on which to base any conclusions. My first successful run came after 7000 individuals, which I now know is rather unusual for the parameters that I was then using.
- GP will search out any loopholes in the data set presented or the scoring and evaluation criteria used in fitness.
- A very important special case of the previous observation occurs for problems where the fitness evaluation cannot be performed on the entire data set that constitutes the "problem", and the tests for fitness must be selected from a much larger (and possibly infinite) set. The process of selection is very important. The results here indicate that changing the tests periodically (e.g. each generation) can increase the likelihood of evolving a more general algorithm.
- The primitives will be used in bizarre ways, ways that people would never use -- and the relative difficulty of the problem is affected in good measure by the primitives selected. Return values for all functions for all possible inputs (i.e. the set of the return values of all other functions) should be defined.
- Steady State GP as conceived by Reynolds and based on Genetic Algorithm work by Syswerda and Whitley has been shown to be an effective step beyond batch GP for the sorting problem described here.
- Non-fitness single crossover, defined above, also shows promise as an easy to integrate genetic operator which helps the convergence in this sorting problem. Presumably it allows potentially valuable fragments of genetic material held within otherwise undistinguished individuals to come together over several crossover operations, thus allowing larger genetic steps to be made than would otherwise be the case.
- The improvement in success from the traditional batch GP to SSGP which includes non-fitness single crossover is significant for the sorting problem described here. Compare Set F (Table 3) with Sets A and B (Table 4).

VII. Future Directions

Within the sorting problem discussed here, many opportunities for future work present themselves. They include:

- Quality of results: Can GP go beyond evolving a more or less general sorting algorithm to evolving ones that are parsimonious, fast, or have other desirable attributes beyond correctness?
- Primitive selection: Much time was spent during the "getting it to work" phase modifying the primitives in an attempt to make the sorting problem "easier" but still non-

trivial. The actual effect of various primitive selections remains an open question, as enough data sets haven't been run to be sure. There is a lot of interesting work to be done surrounding the relative difficulty of various primitive selections as well as creating techniques which will allow evolution of solutions to more difficult problems.

- Recursion: Many elegant and efficient sorting algorithms are best expressed as recursive functions. Using GP to evolve recursive functions is possible [11] if a bit unwieldy. Evolving recursive sorting algorithms is an interesting challenge.
- Subroutines: Both John Koza [10, 11] and Peter Angeline [1] have developed techniques to evolve subroutines during a GP run. Applying these techniques to the sorting problem should be interesting.

Acknowledgments

Thanks to John Koza for work he has done on defining and demonstrating the wide applicability of GP, as well as for his encouragement at Alife III. Craig Reynolds generously shared his SSGP ideas with me over email. Peter Angeline as well as Craig and John engaged in interesting discussions with me over email. Thanks also to the reviewers for very helpful comments and particular thanks to my wife, Karen, for many penetrating and insightful reviews and strong support.

All the code for this system was developed by the author from scratch on a 386 laptop using XLISP-PLUS version 2.1d. The experiments were run under Lucid Common LISP.

References

[1] P. J. Angeline, and J. B. Pollack, "Coevolving High-Level Representations." LAIR Technical Report 92-PA-COEVOLVE, The Ohio State University, Columbus, OH, 1992. Submitted to *Artificial Life III*, C. G. Langton, Ed.

[2] L. Davis, *Handbook of Genetic Algorithms*. New York, NY: Van Nostrand Reinhold, 1991.

[3] K. A. De Jong, "On Using Genetic Algorithms to Search Program Spaces," in *Proceedings of the 2nd International Conference on Genetic Algorithms,* J. Grefenstette, Ed. Hillsdale, NJ: Lawrence Erlbaum Associates, 1987.

[4] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[5] J. J. Grefenstette, "A System for Learning Control Strategies with Genetic Algorithms," in *Proceedings of the 3rd International Conference on Genetic Algorithms*, J. D. Schaffer, Ed. San Mateo, CA: Morgan Kaufmann, 1989.

[6] W. D. Hillis, "Co-Evolving Parasites Improve Simulated Evolution as an Optimization Procedure," in *Artificial Life II*, C. G. Langton, C. Taylor, J. D. Farmer and S. Rasmussen, Eds. Reading, MA: Addison-Wesley, 1991.

[7] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: The University of Michigan Press, 1975.

[8] J. R. Koza, "Hierarchical Genetic Algorithms Operating on Populations of Computer Programs," in *Proceedings of the 11th International Joint Conference on Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann, 1989.

[9] J. R. Koza, "Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems." *Technical Report No. STAN-CS-90-1314*, Computer Science Department, Stanford University, 1990.

[10] J. R. Koza, "Hierarchical Automatic Function Definition in Genetic Programming," in *Foundations of Genetic Algorithms 2*, D. Whitley, Ed. San Mateo, CA: Morgan Kaufmann, 1992.

[11] J. R. Koza, *Genetic Programming*. Cambridge, MA: MIT Press, 1992.

[12] U. M. O'Reilly and F. Oppacher, "An Experimental Perspective on Genetic Programming," in *Parallel Problem Solving from Nature, 2*, R. Manner and B. Manderick, Eds. Amsterdam, The Netherlands: Elsevier, 1992.

[13] C. W. Reynolds, "An Evolved, Vision-Based Behavioral Model of Coordinated Group Motion," in *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, J. A. Meyer, H. L. Roitblat, and S. W. Wilson, Eds. Cambridge, MA: MIT Press, 1993.

[14] J. D. Schaffer, *Some Experiments in Machine Learning Using Vector Evaluated Genetic Algorithms*. Doctoral Dissertation, Department of Electrical and Biomedical Engineering, Vanderbilt University, Nashville TN, 1984.

[15] G. Syswerda, "A Study of Reproduction in Generational and Steady-State Genetic Algorithms.", in *Foundations of Genetic Algorithms*, G. J. E. Rawlins, Ed. San Mateo, CA: Morgan Kaufmann, 1991.

[16] D. Whitley, "The GENITOR Algorithm and Selection Pressure: Why Rank-Based allocation of Reproductive Trials is Best." in *Proceedings of the Third International Conference on Genetic Algorithms*, J. D. Schaffer, Ed. San Mateo, CA: Morgan Kaufmann, 1989.