Using Genetic Programming to Evolve a General Purpose Sorting Network for Comparable Data Sets

Peter B. Lubell-Doughtie
Stanford Symbolic Systems Program
Stanford University
P.O. Box 16044
Stanford, California 94309
650-740-2784
pld@stanford.edu

ABSTRACT

This paper demonstrates how non-typed genetic programming may be used to evolve sorting networks; specifically a sorting network for a seven-element integer data sequence. This problem has shown importance over time because programs generated to solve it consume a fixed amount of time and operate in a way that is oblivious to the contents of the data sequence being sorted. Both of these factors lead to sorting networks being more desirable than recursive sorts for data sequences of limited length.

1. Introduction

The problem to be addressed in this paper is that of using genetic programming to evolve a sorting network which properly orders a data sequence, of length $n = 7$, under the relation $\leq$ for the elements of the index set $J = \{0, \ldots, 6\}$. The solution to a set of this specific size can be extended to apply to any data sequence of length n, however the computational effort required will vastly increase, as $n \times n!$ comparisons are needed to rigorously check the validity of a program against all possible permutations of the list to be sorted. The sorting problem can be conceived of as consisting of reordering an arbitrary data sequence $a_0, \ldots, a_{n-1} \in A$ to a data sequence $a_{\varphi(0)}, \ldots, a_{\varphi(n-1)}$ such that: $a_{\varphi(i)} \leq a_{\varphi(j)}$ for $i < j$ where φ is a permutation of the index set J.

The motivation behind developing such a sorting network is that it can serve as a parallelizable and computationally efficient method of sorting data sequences that are relatively short. In addition sorting networks have an advantage over other sorting methods because they operate in a fixed amount of time; that is with a fixed computational complexity. Additional motivation behind this problem comes from the relative simplicity with which field programmable gate array (FPGA) integrated circuits can be programmed and reprogrammed in the field to evolve solutions to this problem.

2. Comparator Networks

A definition of comparator networks will now be given towards obtaining a definition of sorting networks.

Given a set A denoting an order relation $\leq$ and a data sequence of length n and given a set of all orderings of that data sequence, that is all of it's permutations, A^n, a comparator is defined as a mapping from $[i{:}j] : A^n \rightarrow A^n$, $i, j \in \{0, \ldots, n-1\}$ with:

$[i{:}j](a)_i = \min(a_i, a_j)$
$[i{:}j](a)_i = \max(a_i, a_j)$
$[i{:}j](a)_k = a_k$ for all k with $k \neq i$, $k \neq j$

for all $a \in A^n$ Knuth (1973). A comparator stage, which is one step in a sorting network, can now be defined as a composition of comparators

$S = [i_1{:}j_1]\cdot\ldots\cdot[i_k{:}j_k],\ k \in \mathrm{N}$

such that all i_r and j_s are distinct, even among each other.

Now we are ready to define a sorting network as a comparator network that sorts all the input sequences. A program to accomplish this will be evolved.

3. Methods

The genetic programming tableau used to evolve a sorting network is presented in table 1.

Objective:	Find a sorting network that correctly orders integers in a data sequence of length seven.
Terminal set:	*Data Terminals*: `D0`, …, `D6`
Function set:	`COMPARE-EXCHANGE`, `PROG2`, `PROG3`, `PROG4`
Fitness cases:	Each element of the sorted array compared to the result of the application of the generated program to each permutation of the sorted array. This results in $n \times n!$ ($7 \times 7! = 35280$) fitness cases.
Raw fitness:	The sum of number of elements correctly sorted in the array sorted by the generated program for each permutation of the sorted array.
Standardized fitness:	The maximum number correctly sorted elements $n \times n!$ ($7 \times 7! = 35280$) minus the raw fitness plus one one-hundredth of the number of `COMPARE-EXCHANGE` functions executed.
Hits:	The number of `COMPARE-EXCHANGE` functions executed.
Wrapper:	None.
Paramaters:	$M = 2{,}048$. $G = 151$. $N_{\text{rpb}} = 1$.
Result-Designation:	The best so far individual.
Success predicate:	The standardized fitness is equal to or below 0.16 (for $n = 7$)

Table 1: Genetic programming tableau for 7 input sorting network

3.1. Terminals

Because this genetic program was implemented without strong typing the only necessary terminals are those data terminals which index values of the array to sort to exchange.[1]

3.2. Functions

The `COMPARE-EXCHANGE` function receives two scalars, which are interpreted as indexes into the array to be sorted, it then compares the values of the elements in the array to be sorted at these indexes and exchanges these values if either index one is less than index two and index value one is greater than index value two or index one is greater than index two and index value one is less than index value two. It essentially performs a swap function constrained to when swapping will result in the array becoming more sorted than it previously was. A value of zero is returned by this function.

The `PROGN` functions string together N sequences of program evaluations. They are necessary to link the comparator stages together in order to form a sorting network. A value of zero is returned by each of these functions.

3.3. Evaluation of Fitness

At a high level of abstraction the fitness of an individual is evaluated by comparing the ordering of the elements of a stored sorted data sequence with the ordering of those in the test data sequence after the program has been run on it. This comparison is done for each permutation of the original data sequence leading to an evaluation of the performance of the program under all possible initial circumstances. By performing the evaluation in this way it is made apparent when the sorting network will perform well for some data sequence orderings but not all. This is an essential feature of the fitness evaluation because it allows programs that perform well in the entire search space to receive a higher probability of reuse in subsequent generations of a genetic programming run. The algorithm used to generate all possible permutations was presented in Rosen (1999).

The standardized fitness range has an upper bound of $n \times n!$ + one one-hundredth the number of comparison swaps executed, which is greater than n (for $n > 3$). The lower bound of the standardized fitness is equal to the last term of the upper bound; one

[1] It may be noted that were strong typing used it would become necessary to include a `NOP` terminal that could be bound to the `PROGN` functions. This is necessary in the current implementation of Java-based Evolutionary Computation and Genetic Programming Research System (ECJ) by Sean Luke (Version 10 2002). However, it will be unnecessary in feature versions.

one-hundredth the number of comparison swaps greater than *n* (for *n* > 3). For *n* = 7 the upper bound is 35,280 plus one one-hundredth the number of comparison swaps, which is indeterminate because the `PROGN` functions allow for an arbitrary stringing of `COMPARE-EXCHANGE` functions. The lower bound is precisely the success predicate of 0.16 (16 is one one-hundredth the minimum number of `COMPARE-EXCHANGE`s necessary to solve the problem).

3.4. Sample Trees

Figure 1 pictures a sample sorting network for four inputs. The values to be swapped at each comparator stage are indicated by the black dots. Each dot is to be swapped with the dot to which it is connected by a black line.

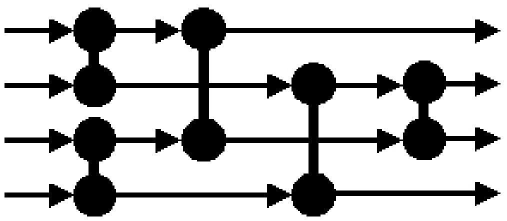

Figure 1: A 4 input sorting network

This sorting network can be represented by the following 100% correct lisp S-expression:

```
(PROG4 (PROG2
            (COMPARE-EXCHANGE 1 2)
            (COMPARE-EXCHANGE 3 4))
       (COMPARE-EXCHANGE 1 3)
       (COMPARE-EXCHANGE 2 4)
       (COMPARE-EXCHANGE 2 3))
```

This expression has a standardized fitness of the number of permutations multiplied by the length of the data sequence minus the result of comparing the run of the program on these unsorted permutations with the sorted data sequence plus one one-hundredth of the number of `COMPARE-EXCHANGE` operations performed. When calculated the fitness of the above expression is 0.05 ($96 - 96 + .01 \times 5$). This value would satisfy the success predicate for a data sequence of length four.

During a genetic programming run a less parsimonious individual designed to solve this same sort problem might be evolved such as below:

```
(PROG4 (PROG4
            (COMPARE-EXCHANGE 1 2)
            (COMPARE-EXCHANGE 3 4)
            (COMPARE-EXCHANGE 2 4)
            (COMPARE-EXCHANGE 2 3))
       (COMPARE-EXCHANGE 3 2)
       (COMPARE-EXCHANGE 1 3)
       (COMPARE-EXCHANGE 2 3))
```

This 100% correct individual would receive a fitness value of 0.07. Because of its lack of parsimoniousness this individual would not satisfy the success predicate for a data sequence of length four. As demonstrated above a 100% correct individual that is not parsimonious could be evolved but still not satisfy the success predicate.

3.5. Parameters

The population size used was 2,048 individuals with a maximum number of generations set at 151. Runs were initially performed with the default ECJ population size of 1,024 individuals. The initial population was randomly generated. No primed individuals were included. Subsequently runs were made with the population size doubled and it was decided that the increased diversity of the initial population resulted in a desirable outcome. The breeding phase consisted of the crossover function, which was performed with a probability of 0.9, and the reproduction function, which was performed with a probability of 0.1. Tournament selection was performed with a maximum depth of 17.

Given the initial population of individuals contains a sufficient amount of genetic diversity the crossover and reproduction operators used allow all the structures in the space of all possible structures to be created. This is true because the space of all sorting networks is finite in size and the operations allow for the creation of all possible structures by creating the actual structure or creating an equivalent structure which may be either more parsimonious or less parsimonious, but will none the less be equivalent.

3.6. Success Predicate

The success predicate for *n* = 7 is set to 0.16 because this value is one one-hundredth of the known minimum number of comparison exchanges necessary to create a valid sorting network. For an arbitrary data sequence length *n* this value should be set to one one-hundredth the known or lower bound of the number of comparison exchanges necessary to create a valid sorting network for a data sequence of the length *n*. For data sequences where a proof does not exist determining a lower bound on the number of comparison exchanges necessary an arbitrary low bound may be set such that it is still reasonable, greater than the lower bound for a data sequence of length *n* – 1. An individual which satisfies the success predicate will be designated best of run and terminate the run.

4. Results and Discussion

The experiment was performed on a 996MHz Pentium III processor with 512 MB of RAM running Windows XP Professional. The base genetic programming environment was ECJ 10 compiled under Eclipse 2.1.1. For a data sequence of length 7 the runtime of a single genetic programming run varied between 15 minutes and 45 minutes depending probabilistically on the individuals generated.

4.1 Results

The first experiment was done with a data sequence of length four (using the data terminals `D0`, …, `D3`) for the purpose of testing the efficacy of the fitness measure and the validating the function of members of the function set. The individual generated scored the success predicate of .05 (one one-hundredth the known required number of steps to complete a sort of this length).

Next an experiment was done with a data sequence of length five to confirm the scalability of the problem as formed. This test was successful and the initial problem of a data sequence of length seven was next addressed.

The typical behavior observed during one run for a data sequence of length seven was a best individual of generation zero scoring a standardized fitness of roughly half that of the generation average standardized fitness, this percentage would decrease through proceeding generations. The best of generation individual's fitness was then approximately halved each proceeding generation until an intermediate generation is reached. It was found that this intermediate generation would be near generation four. Thereafter the standardized fitness rapidly decreased by approximately 75%. Subsequently the standardized fitness would drop to just above the fitness required by the success predicate. This most likely happens because the best of generation individual is a 100% correct sorting network however, it lacks parsimoniousness so much that the additional factor of one one-hundredth the number of `COMPARE-EXCHANGE`s performed pushes its standardized fitness above the success predicate.

4.2 Discussion

The results above indicate that genetic programming was able to successfully evolve a sorting network for data sequences of length four, five, and seven. Due to the adaptable and general way in which the genetic program was implemented, adjusting of the data sequence size field in the ECJ parameters file[2] will allow for solutions to sorting network problems of any input data sequence size to be generated. This is of course within the constraints of time and computing power.

4.3 Problem Scope

Sorting networks, due to the rapid increase in comparators necessary to solve the problem, are best suited for small data sequences. This problem was initially formulated to generate a solution to the more general problem of efficiently sorting a data sequence of any size. For longer data sequences a heap sort or quick sort would be evolved and for shorter sequences, short probably to be defined as 16 and under, a sorting network would be evolved. These two functions would then be linked together and the appropriate one chosen based on the length of the data sequence. Due to computing resource limitations the scope of the problem was reduced to the evolution of sorting networks. This will be discussed further in Further Work.

5. Conclusions

This paper demonstrated how genetic programming can be used to evolve a program which implements a sorting network for data sequences of length 7. These genetic programs were represented as S-expressions and reproduced and recombined according to a fitness measure evaluating the function of the sorting network on the entire search space of the problem; that is all permutations of the sorted data sequence.

Standard non-typed genetic programming methods were used in program generation. Because non-typed genetic programming was used many (all in solutions to data sequences of non trivial length) functions, such as the follow program segment:

```
(PROG4 D1 D0 D3 D0)
```

had no effect on the order of array elements and could be eliminated. After these ineffectual program segments were removed generated programs contained a number of `COMPARE-EXCHANGE`s equal to the known minimum necessary to solve the problem.

A generated program which satisfies the success predicate is guaranteed to be a valid sorting network because the fitness measure ensures correct program performance for all possible inputs to the program. Because the fitness cases were so numerous, and

[2] This file is used to specify the terminal set, function set and additional parameters for the program.

reevaluation of the program was required for each fitness case, individual evaluation was rather expensive and generations were often time consuming. However, because of the efficiency of the fitness measure, a large number of generations was not necessary to find an individual satisfying the success predicate.

6. Further Work

As previously mentioned the problem addressed in this paper is a reduced version of the problem originally intended to be addressed. Future work extending only the sorting network portion of this problem might involve allowing genetic programming do the work of evolving a `COMPARE-EXCHANGE` function from more primitive functions such as `IF`, `OR`, `AND`, `GT` (greater than), `LT` (less than), and `WRITEM`*n* and `READM`*n* functions.

Extending the program to become a link between two separately evolved genetic programs for short and long data sequences would involve significantly more time and resources. A terminal set and function set designed to lead to the creation of a quick sort function (if automatically defined loops, functions, and recursion are not incorporated) would include those presented in table 2.

Terminal set:	`L`, `R`: statics to hold array start index position (0) and end index position (length – 1). `M1`, `M2`, `M3`: memory variables, two to hold indexing values and one to hold a partition position.
Function set:	`SORT`: the function itself, included to allow for recursion (this could also be implemented using automatically defined recursion), `ADD`: addition, `INCF` / `DECF`: increment / decrement, useful because the sort involves counting, `READM1`, `READM2`, `READM3`: read memory variables 1, 2, or 3, `READ` (usage `READ K`): retrieve element at position *k* in array based memory, `SUB`: subtraction, `WRITEM1`, `WRITEM2`, `WRITEM3`: write memory variables 1, 2 or 3, `SWAPARRAY` (usage `SWAPARRAY X1 X2`): swap values a positions `X1` and `X2` in array, `WRITE` (usage `WRITE X K`): write value `X` to position *k* in array based memory, `DU`: do until iteration function, `WHEN`: allow for conditional execution, `GT / GTE`: returns a positive value if argument 1 is greater than / greater than or equal to argument 2, to be used by conditional functions, (in addition possible `DIV` to set the partition value, if it is not specified by the programmer).

Table 2: Proposed genetic programming functions and terminals for a quicksort program

The fitness measure could be defined similarly to that for a sorting network. The evolved program would be run against all permutations of the sorted list and the number of correctly sorted elements counted.

Of course genetic programming may not evolve a program that conforms to our typical notions of a quick sort. However, this is a positive feature of genetic programming. If the fitness measure also takes into account the time required to sort the list it is possible that a new sorting technique could be developed.

Acknowledgements

I would like to thank Sean Luke and Liviu Panait for the information they provided concerning the current implementation of ECJ.

Bibliography

Knuth, Donald E. 1973. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison Wesley.

Koza, John R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: The MIT Press.

Rosen, Kenneth H. 1999. *Discrete Mathematics and Its Applications*. Fourth Edition. WCB / McGraw-Hill.