Parallel genetic algorithm for creation of sort algorithms

Igor Trajkovski

Faculty of Computer Science and Engineering,
"Ss. Cyril and Methodius" University in Skopje,
Rugjer Boshkovikj 16, P.O. Box 393, 1000 Skopje, Macedonia
trajkovski@finki.ukim.mk
http://www.finki.ukim.mk/en/staff/igor-trajkovski

Abstract. In this paper we present parallel genetic algorithm that was used to the task of evolving imperative sort programs. A variety of interesting lessons were learned. With proper selection of the primitives, sorting programs were evolved that are both general and non-trivial. Unique aspect of our approach is that we represent the individual programs with simple assembler code, rather than usual tree like structure. We also report the effect of different parameters on quality of the programs and time needed for finding the solution.

Keywords: genetic algorithm, sort algorithm, parallel algorithm

1 Introduction

Genetic Programming (GP) derives from Genetic Algorithms (GA), developed initially by John Holland [1] and extended since then by many others. Genetic Algorithms often work on fixed length, linear representations of genetic material, and operate on this material with fitness proportionate selection, reproduction, crossover, and mutation [2], [3].

John Koza has developed GP, using analogues of GA genetic operators to directly modify and evolve tree structured programs (typically LISP functions) [4], [5]. While genetic programming does not imitate nature as closely as do genetic algorithms, it does offer the opportunity to evolve programs of a high complexity, without having to define the size or the structure of the program in advance. Koza has shown in [5] that GP can be effectively applied to an unusual variety of problem areas.

Sorting is a procedure of rearranging the elements of a given input sequence into ascending or descending order. Sorting has a rich history in computer science and is a procedure that requires complex control structures. There are a lot of solutions to this problem domain - some are simple and intuitive, while others are complicated but of greater efficiency. Sorting algorithms fall into two classes of time complexity, specifically, $O(n^2)$ and $O(n \times \log(n))$. The algorithms with quadratic complexity are usually expressed iteratively, whereas algorithms with $O(n \times \log(n))$ complexity leads more naturally to recursive expressions. The

sorting problem has attracted a great deal of research, due to its ubiquity, and due to the challenge of solving it efficiently.

This paper presents work on evolving general iterative sorting algorithms represented as linear list of instructions, contrary to the traditional representation of programs in GP, as trees. Sorting is clearly a general problem for GP since the evolved hypothesis will have to sort the comparable elements of any input sequence of arbitrary length into ordered sequence. This is a challenging problem, and in this paper we will show that it can be solved with the proposed method.

The remainder of the paper is organized as follows: first we provide a literature review on the evolution of sorting programs. Then the fitness function used to guide the evolutionary search is described. We then proceed to describe the experimental context: instruction set used for construction of the programs, evolutionary algorithms control parameters, parallelization strategies followed by a section that details and discusses the experimental results, and the computational effort involved. Finally, conclusions are drawn and ideas for future work are presented.

2 Related work on the evolution of sorting algorithms

A literature survey on evolution of sorting programs evolution revealed several attempts in this problem domain.

Kinnear [**?**] evolved iterative sorting algorithms, mainly of bubble sort's simplicity. He investigated the relative probability of success and the difficulty resulting from varying the primitive terminal and non-terminal elements. The primitive alphabet contained elements that could result in an exchange-oriented sorting strategy. Primitive functions were defined for swapping sequence elements, comparing elements in specified sequence index values, incrementing and decrementing arithmetic variables. Control functions contained conditionals and a bounded iteration construct. It considered the addition of a linear parsimony function in order to discourage the individuals' increasing size as well as a disorder penalty, in the case where the remaining disorder was greater than the initial sequence disorder.

O'Reilly and Oppacher [7], [8] also investigated ways of evolving iterative sorting algorithms. Their first attempt [7] failed to produce a 100% correct individual. The representational constructs and fitness function used were different than those used in Kinnear's experiments. Specifically, primitive functions and control structures included decrementing a variable, accessing indexed sequence elements, swapping adjacent sequence elements, bounded looping, and conditional. The fitness function counted the number of out-of-place items. Their second attempt [8] yielded a successful outcome. It considered different primitive constructs and two fitness functions; the first fitness function was the same as in [7] whereas the second was based on permutation order [9], the count for each sequence element of the smaller elements that follow it.

The most recent attempt to the evolution of sorting algorithms is presented in the work of [10] with their PushGP system. They used primitives along the lines of earlier investigations: swapping and comparing indexed elements, getting the list length, accessing list elements. The Push3 programming language offers a variety of explicit iteration instructions but also allows for the evolution of novel control manipulation structures. They evolved an $O(n^2)$ general sorting algorithm and suggested an efficiency component addition to the fitness function as a precursor to the evolution of ingenious $O(n \times \log(n))$ algorithms, though they report no experiments towards that direction.

3 Sorting Fitness Function

When evolving a sorting algorithm, the following two problems arise:
How to quantitatively measure the algorithm's ability to sort?, and
How to Recognize Success?, or how can we be sure that we found the solution.

The fitness function is the driving force behind any evolutionary algorithm. The choice of fitness function can play a critical part in the success of a given run. In this work, we consider fitness functions based on different measures of change in the sequence disorder.

We first familiarize the reader with a concepts of sequence disorder, namely, *inversions*.

Let $a_1\ a_2\ \ldots\ a_n$ be a permutation of the set $1,\ 2,\ \ldots,\ n$. The pair $(a_i,\ a_j)$ is called an inversion of the permutation if and only if $i < j$ and $a_i > a_j$. Inversion pairs represent pairs of sequence elements that are out-of-order, so that a completely sorted sequence is a permutation with no inversions. Let $Inv(A)$ represent the number of inversions in sequence A. Using these notations we define the following fitness measure F of the program P on given input sequence A:

$$F(P) = Inv(A) - Inv(B)$$

where $B = P(A)$ is the output sequence of the program P executed on input sequence A. In some cases B is the modified, by program P, input sequence A.

We can see that programs which completely sort the input sequence will have maximum fitness. Off course we do not test the program on only one input sequence, but on several (A_i) test sequences. In this case the fitness measure F is calculated with the following formula:

$$F(P) = \sum_{i=1}^{N} Inv(A_i) - Inv(P(A_i))$$

where $N = |A|$.

When evolving a sorting program, a problem presents itself. Frequently, an evolved program which performs successfully all of the fitness tests used (a *completely fit individual*) will fall short of complete generality. Therefore, a particular run is not terminated with the appearance of a completely fit individual. Instead,

additional testing is performed on every completely fit individual and the run is terminated as successful only when one individual passes these more tests.

The fitness tests used to drive evolution consist of K random tests, with a maximum length of 30, with the random tests changed in each evaluation. The additional tests for generality (used only to terminate the run) consist of 256 random tests of maximum length 32, and all 256 sequences of length 8 consisting only of 0 and 1.

While many of the completely fit individuals fail to pass the additional tests (usually only 60% to 80% pass), no algorithm that has passed all of the additional tests has ever been shown to fail on any sequence subsequently presented to it. Many of these evolved and potentially fully general algorithms have been tested with thousands of random sequences of lengths of up to 600 with not a single failure. Thus, while testing can never prove generality, the sorting problem as expressed with these primitives appears to have a certain threshold of testing that, once passed, ensures at least a very high likelihood of generality.

The last fitness measure F has a small shortage. It prefers moderate partial sorting of long sequences than perfect complete sorting of short sequences because the number of fixed inversion is bigger in the first case. Therefore, the following fitness measure is used that normalize the number of fixed inversions relative to the number of inversions sequence had in the first place:

$$F(P) = \sum_{i=1}^{N} \frac{Inv(A_i) - Inv(P(A_i))}{Inv(A_i)}$$

4 Program Representation and Genetic Operators

The basic approach to genetic programming 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. The desired function of the individuals (programs) is to reorder a sequence of integers so as to leave it in "sorted" order, small to large. In our approach, the individuals of the population are pseudo assembler programs. Assembler instructions are defined to compare and swap the various values in the sequence, to increase, decrease and compare various registers, and control the flow of the execution by various conditional jump instructions.

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. If the program access out-of-memory address, or it does not end in predefined maximum number of cycles, it gets lowest possible fitness value.

4.1 Instruction set

In order to apply genetic programming to a problem, it must be cast into a form that can be evolved. A set of assembler instructions must be created that is

sufficient to solve the problem. Arguments to these instructions will be registers and addresses (or values) of sequence elements.

A variety of instruction sets were tried in order to successfully evolve a general and yet non-trivial sorting algorithm. The instructions described here are the least complex, and therefore most realistic that would reliably allow evolution of general sorting algorithms within a reasonable time.

Our instruction set is composed of the following 12 classes of instructions:

```
 1. MOV  Ri, Rj(const)
 2. XCHG mem[Ri], mem[Rj]
 3. INC  Ri
 4. DEC  Ri
 5. JE   Ri, Rj(const), line
 6. JA   Ri, Rj(const), line
 7. JB   Ri, Rj(const), line
 8. JMP  line
 9. JA   mem[Ri], mem[Rj], line
10. JB   mem[Ri], mem[Rj], line
11. JE   mem[Ri], mem[Rj], line
12. END
```

$R_j(const)$ means R_j or *constant value*, which will get us to 16 different types of instructions.

Instruction 1 moves the value of register R_j to register R_i, or moves a constant value in register R_i. Instruction 2 swap the values of sequence elements at positions R_i and R_j. Instructions 3-4 increase/decrease the value of register R_i. Instructions 5-7 are conditional jumps, comparing register R_i with another register R_j or constant value, to instruction *line*. Instruction 8 is a unconditional jump to instruction *line*. Instructions 9-11 are conditional jumps, comparing the values of sequence elements at positions R_i and R_j, to instruction *line*. If instruction 12 is executed the program terminates.

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

4.2 Initial population

Initial population was constructed by generating 1000 random programs with fixed length of L instructions. The effect of L was investigated in the experiments. Each instruction was randomly chosen from the 16 different types of instructions.

4.3 Mutation

Mutation works by selecting an individual by uniform random selection, selecting an instruction within the program and then one of the three types of mutations is done: mutating (changing) the instruction arguments, changing the type of the instruction or generating completely new instruction. When instruction argument is changed, if it is a constant, random value with power low distribution

is added/subtracted, if it is a register it is substituted by another random register. When type of the instruction is changed, another type is chosen that has the same type of arguments. For example, if the type of INC instruction is mutated, it can become DEC, but not XCHG.

4.4 Crossover

Crossover is done in the following way: two individuals are selected with fitness proportional selection, than two continuous blocks of instructions, equal size, one from each individual were selected, and than they were exchanged. In this way we get two new individuals. The size of the blocks is a random number from 4 to 8 instructions. The position of the block was also random number from 0 to L - $block\_size$.

5 Experimental Setup and Parallelization

A population of 10.000 programs were randomly generated with a fixed length L, generated by uniform selection over the list of instructions. Evaluation of the programs was done by their execution and analyzing the disorder of the input sequence before and after the execution of the program in order to compute its fitness. The number of registers used by the programs was fixed to NR. Before the execution of the programs, it is assumed that the length of the input sequence N is stored in register R_0 and that numbers of the input sequence are located on the memory locations 0 to $N-1$. The memory working area was set to be from $-3*N$ to $3*N$. If a program access a memory location outside this interval, the fitness of the program was set to minimum possible value. Also, if the program did not finish its execution after 5000 executed instructions it is asumed that it does not halt, and it is given the lowest possible fitness value. The initial value of all registers, except R_0, was set to 0. The initial value of all memory locations, except from 0 to $N-1$, was also set to 0.

5.1 Parallelization strategies

The basic idea behind most parallel programs is to divide a task into chunks and to solve the chunks simultaneously using several processors. This divide-and-conquer approach can be applied to GA in many different ways, and the literature contains many examples of successful parallel implementations. Some parallelization methods use a single population, while others divide the population into several relatively isolated subpopulations. Some methods can exploit massively parallel computer architectures, while others are better suited to multicomputers with fewer and more powerful processing elements.

There are three main types of parallel GA: (1) global single-population master-slave GA, (2) single-population fine-grained, and (3) multiple-population coarse-grained GA. In a master-slave GA there is a single population (just as in

a simple GA), but the evaluation of fitness is distributed among several processors. Since in this type of parallel GA selection and crossover consider the entire population, it is also known as global parallel GA. Fine-grained parallel GA are suited for massively parallel computers and consist of one spatially structured population. Selection and mating are restricted to a small neighbourhood, but neighbourhoods overlap permitting some interaction among all the individuals. The ideal case is to have only one individual for every processing element available. Multiple-population (or multiple-deme) GA are more sophisticated, as they consist of several subpopulations which exchange individuals occasionally.

This exchange of individuals is called migration and it is controlled by several parameters. Multiple-deme GA are very popular, but they are also the class of parallel GA which is most difficult to understand, because the effects of migration are not fully understood. Multiple-deme parallel GA introduce fundamental changes in the operation of the GA and have a different behaviour than simple GA. Multiple-deme parallel GA are known with different names. Sometimes they are known as "distributed" GA, because they are usually implemented on distributed memory MIMD computers. Since the computation-to-communication ratio is usually high, they are occasionally called coarse-grained GA. Finally, multiple-deme GA resemble the "island model" in population genetics which considers relatively isolated demes, so the parallel GA are also known as "island" parallel GA.

Another method to parallelize GA combines multiple population with master-slave or fine-grained GA. We call this class of algorithms hierarchical parallel GA, because at a higher level they are multiple-deme algorithms with single-population parallel GA (either master-slave or fine-grained) at the lower level. A hierarchical parallel genetic algorithm (HPGA) combines the benefits of its components, and it promises better performance than any of them alone.

We implemented a HPGA. In our test computations, we ran the same sequential version on several processors, and after every 100 generations 3 randomly selected individuals from 10 best individuals of every population were sent to all other populations. The best known solutions are always found for all instances. According to the results, we observe that the algorithm is well fitted to a large number of instances, and that speedup is proportional to the number of the processors. We can mention that there is a case with superlinear speedup where the speedup is more than 8 when we run our GA on 8-processor machine. This is due to the fact that GA are non-deterministic algorithms.

6 Evolved Sort Programs

Let's look at some of the sort programs that have evolved using the presented approach, $L = 20$, $NR = 5$.

```
Example 1:
0  XCHG mem[R4], mem[R3]
1  INC R4
2  MOV R1, R4
3  DEC R0
4  JA mem[R0], mem[R1], 18
5  JA mem[R4], mem[R0], 17
6  XCHG mem[R0], mem[R1]
7  MOV R4, R3
8  JB mem[R1], mem[R4], 17
9  JNE R0, R2, 2
10 MOV R2, 2
11 END
12 INC R1
13 JB R0, -1, 1
14 DEC R0
15 DEC R0
16 DEC R4
17 XCHG mem[R0], mem[R4]
18 INC R0
19 JNE R4, R0, 0
```

```
Example 2:
0  DEC R0
1  MOV R4, R3
2  JB mem[R1], mem[R0], 5
3  JMP 16
4  XCHG mem[R4], mem[R3]
5  MOV R3, R2
6  JB R1, R1, 0
7  XCHG mem[R0], mem[R3]
8  JB mem[R0], mem[R1], 16
9  JE R0, R4, 0
10 XCHG mem[R4], mem[R3]
11 JNE R4, R2, 2
12 DEC R0
13 MOV R1, R1
14 MOV R3, -1
15 MOV R4, R4
16 INC R4
17 XCHG mem[R0], mem[R3]
18 JB R1, R0, 4
19 END
```

The first impression is that they look very messy, and it is hard to follow the flow of the algorithm. The second thing is that they have a lot of unnecessary instructions that do not have any effect on the functionality. But on the other hand it was expected in some way, because also the DNA of almost any species has these features. If we remove instructions that do not have impact on the semantic of the programs and rewrite them in more readable PASCAL-like language, the second program for example, will look like this:

```
Example 2 (PASCAL-like):
    r0 := n; r1 = r2 = r3 = r4 := 0;
0:  dec(r0);                          11: if r4 <>0 then goto l2;
1:  r4 := 0;                          12:
2:  if smaller(0, r0) then goto l5;   13:
3:  goto l16;                         14:
4:  swap(r4, 0);                      15:
5:                                    16: inc(r4);
6:                                    17: swap(r0, 0);
7:  swap(r0, 0);                      18: if 0 <r0 then goto l4;
8:  if smaller(r0, 0) then goto l16;  19: goto end;
9:  if r0 = r4 then goto l0;              end:
10: swap(r4, 0);
```

swap(X, Y) is a procedure for swapping the values on memory locations X and Y, and *smaller(X, Y)* is a boolean function that compares values on memory locations X and Y. Empty instructions represent instructions that do not have

any effect on the semantic of the program, something like a NOP (no operation assembler instruction).

If we look closely we can see that this is INSERTION SORT algorithm. It is written in a very messy way, but if we follow the execution of the program it is doing exactly the same thing as insertion sort algorithm. Also the other program is strange implementation of insertion sort algorithm. Most of the discovered sorting programs were implementation of insertion sort algorithm, but there were also some versions of bubble sort. In any case all of them had at least $O(n^2)$ complexity to sort n elements of an array. If we want to discover $O(n \times log(n))$ sort algorithm we gonna need procedure calls necessary for implementing recursion.

7 Results

Results of the experiments are presented in Table 1. 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 1000 generations. The significant conditions of the experiments 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 column under # SUCC 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 passed all of the postrun generality tests.

AVG INDS 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% SUCC 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 indicates how quickly the 100% SUCC RUNS reached 100%.

As we can see, when having more registers it was harder (slower) to find the solutions, because the search space was much bigger. In experiments where programs were 10 instructions long, it was harder finding the solution because the programs were to short and there is no space for redundancy. Also the general conclusion is that number of tests should be bigger, even if that takes longer to evaluate the individuals, but on aggregate the search time was shorter, because with clever evaluation the search procedure earlier removes partial solutions.

Table 1. Experimental Results

Set	**# registers** *NR*	**# instructions** *L*	**# tests**	**75%**	**90%**	**# SUCC RUNS**	**# GEN RUNS**	**AVG INDS**
A	5	10	15	18	15	0	0	/
B	5	10	30	20	16	6	6	2943893
C	5	20	15	20	20	19	10	4149037
D	5	20	30	20	18	17	16	1157621
E	10	10	15	5	3	0	0	/
F	10	10	30	7	3	0	0	/
G	10	20	15	11	8	2	0	6551290
H	10	20	30	12	10	5	0	2922396

8 Conclusion & Future Work

The process of evolving general sorting algorithms provided an interesting testbed for assessing the power of genetic programming on solving a challenging problem. In this paper we showed that evolving imperative programs, instead of tree programs, is possible and effective. This investigation has been another step towards allowing the evolution of more complex algorithms. In the future we plan to apply this methodology for evolving programs that even when they are executed probabilistically their functionality is unaffected.

References

1. Holland, J. H., Adaptation in Natural and Artificial Systems, Ann Arbor, MI: The University of Michigan Press., 1975.
2. De Jong, K. A., 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.
3. Goldberg, D. E., Genetic Algorithms in Search, Optimization, and Machine Learning, MA: Addison-Wesley, 1989.
4. Koza, J. R., 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.
5. Koza, J. R.,Genetic Programming, Cambridge, MA: MIT Press, 1992.
6. Kinnear K. E., Evolving a sort: Lessons in genetic programming, in Proceedings of the 1993 International Conference on Neural Networks, vol. 2. San Francisco, USA: IEEE Press, 1993.
7. O'Reilly U.-M. and Oppacher F., An experimental perspective on genetic programming, in Parallel Problem Solving from Nature 2, 1992.
8. O'Reilly U.-M., A comparative analysis of Genetic Programming, in Advances in Genetic Programming 2, Cambridge, MA, USA: MIT Press, 1996.
9. Knuth D. E.,The art of computer programming, volume 3, sorting and searching, Redwood City, CA, USA: Addison Wesley Longman Publishing Co., 1998.
10. Spector L., Klein J. and Keijzer M., The push3 execution stack and the evolution of control, in GECCO 05: Proceedings of the 2005 conference on Genetic and evolutionary computation. New York, USA: ACM Press, 2005.