Genetic Algorithm Application For Sorting (HW8 for CS 391L)

Jay Liu (jayliu@cs)

November 28, 2008

1 Program and parameters

1.1 Problem statement

The program implements a GA model to solve the sorting problem given an fixed length l numeric input vector. The generated solution is a sorting algorithm that contains an ordered list of instruction pairs, each pair consisting of two indices (i, j). Given an input, the algorithm will go down the list, and for each pair performs swap of $input(i)$ with $input(j)$ if $input(i) > input(j)$. Since the indices are 1-indexed, pairs that contain 0 are no-ops and are skipped during the running of the algorithm.

It has the following variables:

- **Population size** Denoted by n, this is the number of different sorting algorithms the program will generate, all of which attempt to solve the same problem.
- **Algorithm size** Denoted by s, this is how many comparison steps an algorithm uses to sort the input. We know that there exists optimal sorting algorithms that use $O(llogl)$ steps, and that we should definitely do better than $O(l^2)$.
- **Evolution step** Each step is a generation of the population.

1.2 Fitness function

The fitness function has two components. The major component is the correctness of the answer. A much smaller component is to reward shorter algorithms that achieve the same correctness.

For each element in the generated answer, we compare it against the element at the corresponding location in the known correct answer.[1] If they match, it is awarded 10 points. If it is incorrect, but it does appear in the correct order relative to the previous element, we award a smaller 2 points, which encourages solutions that have partial correctness. The final score is divided by l, so that highest possible fitness score is 10 after this step. For multiple inputs, the fitness score of the algorithm is the sum of the scores for each input.

We further reward algorithms that are shorter than the median size, by a small amount. In our algorithms, we denote no-op steps as those swap pairs that contain entries with 0 value. For each algorithm in our population, we calculate the number of actual steps discounting the no-ops. We can calculate the median, and for each algorithm that is shorter than the median size, we reward it; in this specific program it is worth 0.5. The amount must be small so that algorithms that are slightly larger but calculate the solution more correctly should not be penalized, since correctness is our objective.

[1] Verification of correctness can be accomplished in linear time. In this program, however, we take advantage of existing built-in sort functions to derive this slightly better fitness function.

1.3 Survival and crossover rules

We first decide on a survival rate of 70%, which is the portion of the population that are eligible for crossover. This only declares their candidacy, but some may not be used, because we randomly pick candidates from this pool for further processing.

We first randomly pick the first parent. With a small probability (20%), this parent survives as is to the next generation. With 20%, the second parent is also random. Otherwise, we deliberately pick the second parent. We would like the parents to share some similarities. We order all the parents by their fitness scores. With 30% probability, the second parent is a bit fitter. With 30%, the choice is only slightly better. Otherwise the second parent is slightly worse.

With the two parents, the crossover rule is to take the first half of the first parent, and concatenate with the second half of the second parent. Since sorting algorithm is sequential, keeping the order largely in tact allows us to keep any existing working sequences.

1.4 Mutation rule

With 10% probability, the newly created algorithm will have one of its pairs mutated. We allow mutation to have any valid input index. Since we also allow 0 to indicate no-op, we may also change the size of the algorithm here.

1.5 Convergence

The convergence criterion is arbitrarily defined to be when 80% of the population is able to achieve maximum fitness.

2 Exploration of impact of variables on convergence

Initially, we evolved the algorithm on a fixed input of size $l = 8$, with a $n = 200$, $s = 6$. The input was $\begin{pmatrix} 5 & 1 & 2 & 3 & 4 & 6 & 0 & 0 \end{pmatrix}$. As expected, the output is $\begin{pmatrix} 0 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$. The algorithm is $((1,7)(5,4)(4,3)(6,8)(2,6)(3,6))$. However, with a modified input $\begin{pmatrix} 1 & 1 & 2 & 3 & 4 & 6 & 0 & 0 \end{pmatrix}$, it fails straight away: $\begin{pmatrix} 0 & 0 & 1 & 2 & 3 & 4 & 1 & 6 \end{pmatrix}$. (The second "1" entry is incorrect.) Figure 1 shows the population fitness.

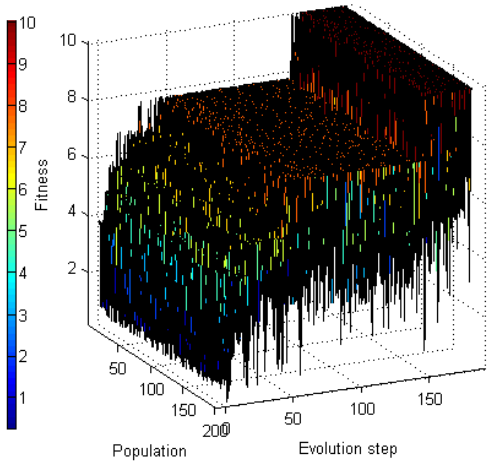

Figure 1: Evolution for a fixed input

By varying the size s of the algorithm, the average steps till convergence after 10 trials are shown in Table 1. With small s, convergence is difficult because the solution needs to be very precise. As larger sized algorithms are allowed, convergence becomes quicker. However, when the number of steps is too large, it once again becomes difficult to converge.

With algorithm size fixed at 10, we varied population size to obtain Table 2. Increasing population size generally required fewer generations to converge to optimum. However, a large population also required longer processing during each evolution step.

Algorithm size	**Average convergence steps**	**Minimum steps**	**Maximum steps**	> 1000
6	315	61	804	2
10	261	30	705	1
50	69	17	154	1
100	524	259	955	4

Table 1: Algorithm size and convergence. The last column lists runs requiring more than 1000 steps, and are not included in the average.

Population size	**Average convergence steps**	**Minimum steps**	**Maximum steps**	> 1000
50	438	124	966	3
100	156	67	537	4
200	261	30	705	1
300	100	19	344	0

Table 2: Population and convergence

3 Parasidic input evolution

3.1 Input evolution parameters

Given a fixed input, or multiple fixed input vectors, the program generates algorithms that specifically solve for those inputs. As we have shown, the algorithm will fail to output correct solution given even minute perturbation to the input. To create algorithms that can solve more general solutions, we evolve our inputs as we did our algorithms.

In Section 1.2, we discussed fitness function for the algorithm. For the input, the fitness function is the opposite of the one for the algorithm. It is the difference between the maximum possible algorithm fitness and what the algorithm achieved. Hence, the input is fit if the algorithm is not, and vice-versa.

Other parameters for input evolution are similar to those for the sorting algorithm. A smaller population is used. We therefore have two programs that compete against each other in terms of fitness. This has significant negative impact on running time until convergence.

3.2 Results

The input size $l = 10$, so we would like an algorithm faster than $O(100)$. The first run with a limit of $s \leq 40$ on algorithm size failed to produce a converging result after 15000 generations.

The second run increased the limit to $s \leq 60$, and a population $n = 500$. The population of input parasites is 30, so the cumulative maximum fitness score for an algorithm is 300.5.[2] The results converged - i.e. 80% of the algorithms solved all the inputs - after 4998 runs. The results are shown in Figure 2. In addition, Figure 3 shows the history of the median of the algorithm size. It settled on a $s = 55$, which would explain why it was difficult to converge during the first run when being limited to 40.

[2] 10 points per correct answer. 300 points for 30 correct answers. 0.5 if the algorithm size is less than median.

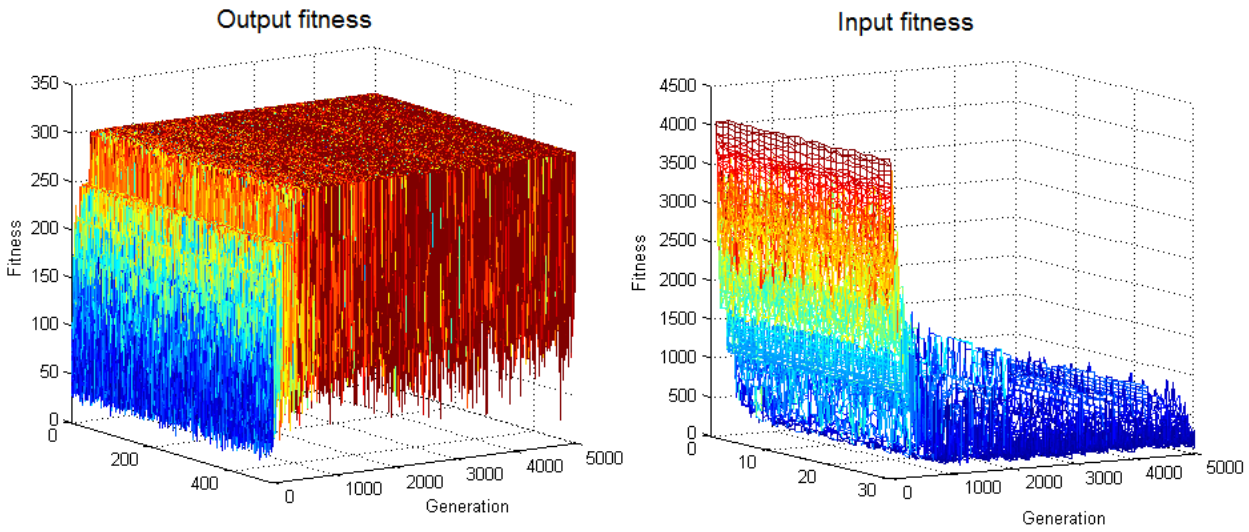

Figure 2: Final result showing progress of algorithm (left) and input (right) fitness.

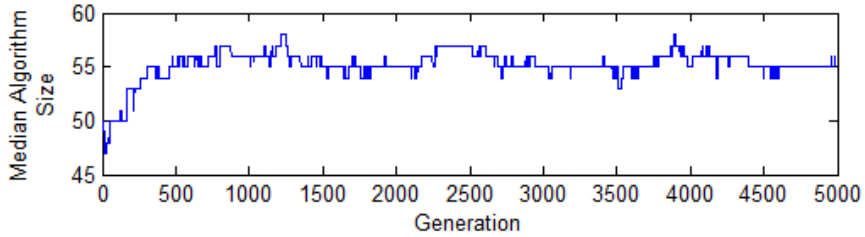

Figure 3: History of median of algorithm size.

3.3 Verification

For verification of the result, an algorithm that was yielding full (300.5) fitness score was given random inputs. Out of 120 tests, it succeeded 115 times, failed 5 times, or 4.2%. For each failure, its fitness score was 8.2 out of a possible 10. This typically means two of the values are out of place, implying it is missing one additional swap.

4 Submission comments

Use gasortnoparam.m to start the algorithm with random parameters. It calls gasort.m, which contains the main program. gafitness.m calculates fitness value, and gaevaluate.m runs an algorithm on an input. gaparasite.m is responsible for input evolution. gaalgosize.m calculates algorithm's effective size.