Homework 8: Genetic Algorithms

Robert Grant

1 The Assignment

The purpose of this assignment was to implement a genetic algorithm and use it to evolve a sorting network capable of sorting eight elements.

2 Submission Information

My submission consists of a very few files: `hw8Train.m`, `hw8Test.m`, and `hw8Intermediate.m`. `hw8Train.m` and `hw8Test.m` are function files with signatures corresponding the specification, and `hw8Intermediate.mat` contains the single variable `hw8Intermediate`, which is the best sorting network produced by my genetic algorithm. `hw8Train` contains the bulk of my implementation, and makes heavy use of subfunctions. Both files are heavily commented, and more information about the functions can be seen in the files themselves or via MATLAB's online help functionality.

3 Implementation Details

My implementation is split into composable subfunctions as much as possible, and most every function has a corresponding test function. The tests should be turned off in my submitted version, but they are turned on by setting the flag RUN_TESTS to 1 at the top of `hw8Train.m`.

My first pass at implementing a genetic algorithm is a simple one. I represent a sorting network as an $N \times 2$ matrix, where each row represents the indices of a pair of elements to compare and swap (iff the first element is larger than the second). Internally, I call each row of a sorting network a `gene` and a sorting network itself a `chromosome` or just `chrom`. I represent the population of all chromosomes as a cell array since I allow the individual chromosomes to be of varying length. The actual application of a sorting network to an unsorted array is implemented in `hw8Test.m`. All other algorithm functionality is implemented in `hw8Train`.

To implement the genetic algorithm, I had to make some design decisions about how to implement mutation, crossover, and selection and breeding. Mutation was the simplest operator to implement.

3.1 Mutation

Every generation, I first count the number of genes in the population. Let N_{chrom} be the population size (number of chromosomes) and l be the length of a particular chromosome (number of swap pairs). Then,

$$N_{genes} = \sum_{i=1}^{N_{chrom}} l_i \quad . \tag{1}$$

Then, letting r be a pre-defined mutation rate, I select the number of genes in the population to mutate

equal to

$$N_{mutations} = \lceil r \times N_{genes} \rceil \quad . \tag{2}$$

My operation to mutate a gene is simply to randomly replace it with a new, randomly generated one. This can result in a gene being replaced with an exact replica with small probability. To perform the mutation operation on the whole population, I first select $N_{mutations}$ number of genes by picking $N_{mutations}$ numbers randomly from 1 to N_{genes}. Then I index the population absolutely by gene, i.e., gene 1 is the first gene of the first chromosome, and gene N_{genes} is the last gene of the last chromosome, and mutate the particular genes I've selected.

3.2 Crossover

In my program I implement a form of single point crossover. This is only complicated by the fact that I allow chromosomes to be of variable length. In my implementation, I randomly select an initial segment from one chromosome and a tailing segment from the other, and join these two into a child chromosome. If a length of 0 is selected for the initial segment, the child of the crossover is an exact duplicate of the second chromosome; if a length of 0 is selected for the tailing segment, the child is made a duplicate of the first chromosome. This is the only way in which I allow chromosomes to continue unmodified into a following generation.

3.3 Selection and Breeding

Each generation, I select a portion of the previous generation to breed to create the next generation. The number of individuals to select is set by hand and is currently

$$N_{selections} = N_{chrom} \times 0.1 \quad , \tag{3}$$

or 10% of the current population size.

After setting this number, I use a form of roulette wheel selection to choose which individuals will go into the breeding pool. Essentially, for a total of $N_{selections}$ times, I choose an individual (without replacement) to be in the breeding pool. Each round, the probability of an individual being selected is proportional to its fitness relative to the other remaining individuals. MATLAB's `randsample` function has been used for this and many other random selection tasks in my implementation.

To score a chromosome, my fitness function generates a number of random 8-element arrays (currently 50), applies the sorting network to them, and then computes the average Hamming distance between the output array and a correctly sorted array (using MATLAB's built in sort function). This is not as good as evolving the tests along with the sorting networks, but it does score the chromosomes with random arrays rather than allowing them to specialize on a few permutations.

After selection a fraction of the population in this way, I randomly pair the selected chromosomes and breed them using crossover until I have created a new population of the desired size (currently the same size as the previous population). No individuals from the previous generation are carried over to the next, except by the crossover mechanism previously explained.

4 Experiments

For all experiments I used the following initial settings:

```
MUTATION_RATE = 0.001;
SELECTION_FRAC = 0.1; % percentage of top performers to select
MIN_INITIAL_CSIZE = 8; % smallest initial chromosome size
MAX_INITIAL_CSIZE = 50; % largest initial chromosome size
```

For my first experiment, I tried

```
POP_SIZE = 10; % population size
NGENERATIONS = 20;
```

and I begin to get decent results even with this very short runtime. See Figure 1.

Using a higher population and more generations, I get more obvious steady improvement:

```
POP_SIZE = 50; % population size
NGENERATIONS = 100;
```

See Figure 2.

5 Conclusions

In conclusion, my algorithm seemed to work quite well, despite the fact that it was simplistic. In a relatively short number of iterations and with a relatively small population it was able to generate sorting networks that sorted correctly with a high probability.

Many improvements or additions to my algorithm could be made. For one, I didn't penalize for long solutions, only incorrect ones. Despite this, my algorithm tended to produce relatively short solutions. I believe this is because of the fact that short, successful sections are more likely to survive and be propagated. I did neither pruning of solutions for duplicated genes nor did I include solution length in my fitness function; either of these could have resulted in shorter solutions.

Another improvement which could improve my solutions is to allow a higher percentage of successful parents to continue on to future generations. I suspect a model like mine (where a very small number of parents survive) is better suited to dynamic environments. Since in this simulation we have static requirements, my algorithm likely would have converged faster if I allowed successful individuals to continue on.

Finally, I suspect larger populations and longer generational runtimes would also result in better solutions

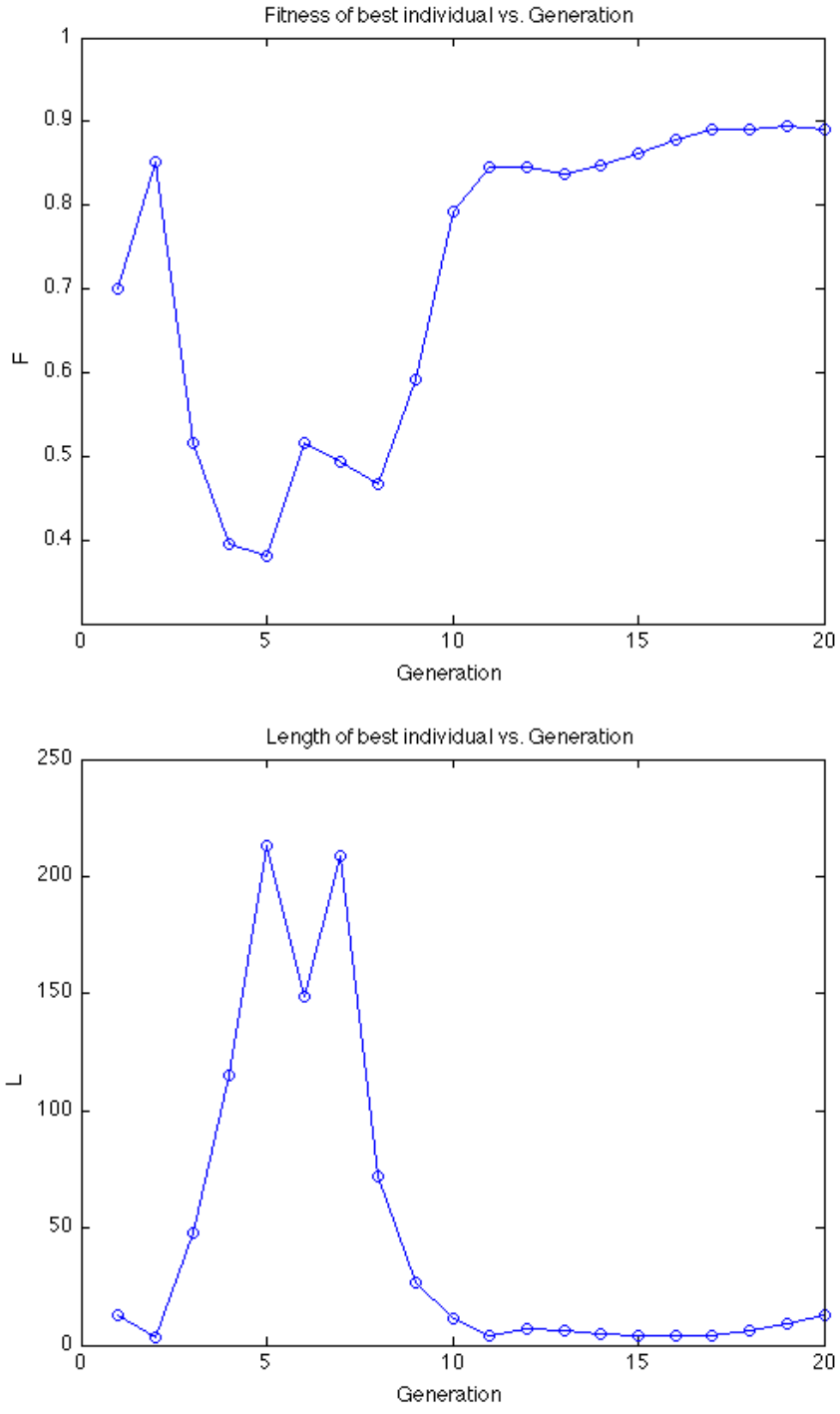
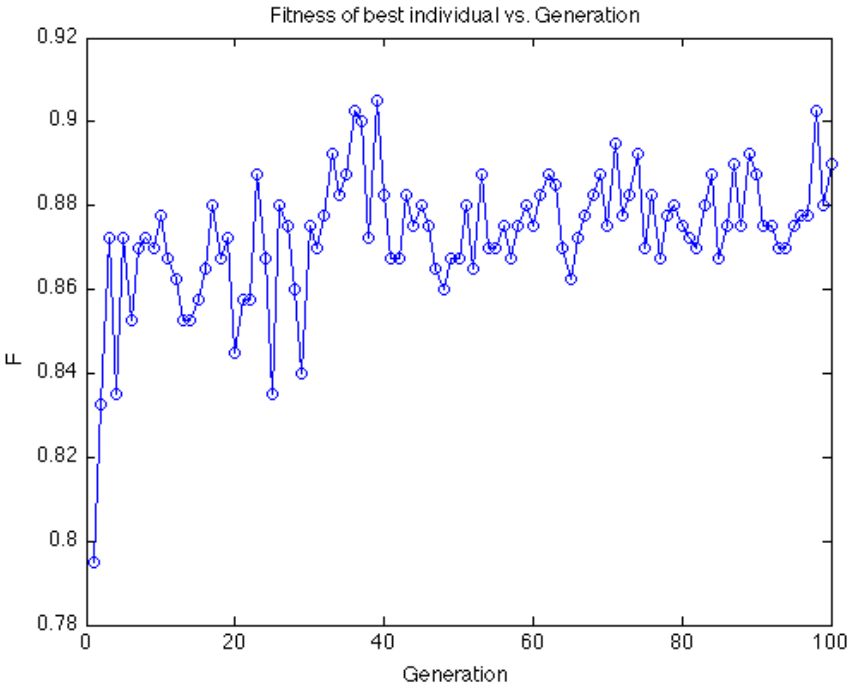

Fitness of best individual vs. Generation
F
Generation
1
0.9
0.8
0.7
0.6
0.5
0.4
0
5
10
15
20
Length of best individual vs. Generation
L
Generation
250
200
150
100
50
0
0
5
10
15
20

Figure 1:


Fitness of best individual vs. Generation
F
Generation
0.92
0.9
0.88
0.86
0.84
0.82
0.8
0.78
0
20
40
60
80
100

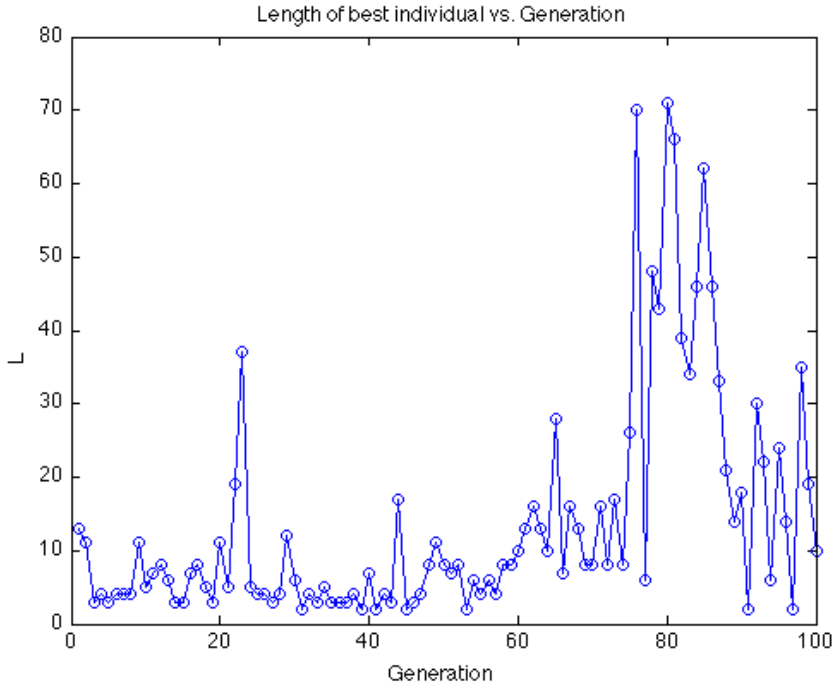

Length of best individual vs. Generation
L
Generation
80
70
60
50
40
30
20
10
0
0
20
40
60
80
100

Figure 2: