COEN 352

Fall 2018

Meta Algorithm for Optimizing a Sorting Algorithm - Application of Genetic Algorithms

Tabish Ashfaq

26752624

Problem Statement

We explore the features and constraints of Genetic Algorithms by applying this evolutionary strategy to a problem of optimizing a sorting algorithm. Traits in the sorting algorithm are represented as swap operations between elements at to distinct positions within a list. In particular, we examine the effects of Fitness Proportionate (Roulette Wheel) selection, in comparison to a naive method of selecting the N/10 fittest individuals for crossover. This examination is done by observing successive populations for the following properties: fitness, efficiency, and effectiveness.

Methods

We present two variations of the model. The differences between these models stem from the choice of data structures representing the populations. The two models shall henceforth be referred to as Model 1 and Model 2.

Representation of Individuals

Each individual is represented as a linked list whose nodes contain the indices for each swap operation and a pointer to the next node in the list. Nodes are also comparable, and two nodes are considered equal if they're indices are identical. This feature can be utilized later to test for diversity between individuals.

```
public class Node
     int i;
     int j;
     Node next;
```

In addition, each individual contains a value representing its fitness, stored as a double value. Functions within the individual facilitate operations for adding, deleting, and modifying nodes. We also implement functions to apply a given individual to a sublist of data, and to set fitness based on the result of this application. The pseudocode for the apply function is provided below.

```
Definitions:   sublist[]: array containing elements from one row of
               the dataset
               swap(int i, int j, sublist[]): exchange elements at
               indices i and j in sublist if i<j and
               sublist[i]>sublist[j]

Function: apply():
               while individual.hasNextNode
                    swap(node.i, node.j, sublist)
                    node = node.next
               End-while
```

Both Model 1 and Model 2 make use of the same representation of individuals.

Representation of Populations

Model 1

In Model 1, populations are represented as ArrayLists of individuals. This allows for trivial implementations for adding, removing, and replacing individuals in the population.

Model 2

Model 2 uses a redblack tree to represent the population, and sorts individuals by fitness. This allows for efficient retreival of the fittest individuals for selection.

Evaluation of Fitness

Both models use a similar method for computing fitness. Fitness is computed as a measure that aims to reflect the efficiency and effectiveness of a prospective solution. The definitions are provided below. We use these definitions in conjunction with the simulated annealing concept of temperature to vary the relative weight given to each of the two parameters over successive generations.

Definitions:

```
     A: array resulting from individual.apply(sublist)
     efficiency = 1/individual.size
     effectiveness = 1/number of inversions in A
     ascendingRuns = number of ascending runs in A

if(inversions == 0)
      effectiveness = 1
    else
      effectiveness = 1/inversions + 1/ascendingRuns
End-if

if (temperature == 1000)
      fitness =  effectiveness*temperature + efficiency/temperature
  else if(temperature > 750)
      fitness = effectiveness*temperature
    else if(temperature < 750 and temperature > 500)
      fitness = effectiveness + efficiency/temperature
    else
      fitness =  effectiveness + efficiency
End-if
```

These definitions are determined empirically to allow for the model to explore the solution space sufficiently before attempting to converge.

Selection

The significant differences in representation of populations between the two models allow us to explore varying methods of selection of individuals for crossover.

Model 1

Model 1 uses a simple arraylist to contain individuals within a population, and therefore lends itself to a probabilistic selection criteria. We choose to implement fitness proportionate, or roulette wheel, selection according to the definition:

$$p_i = \frac{f_i}{\Sigma_{j=1}^{N} f_j},$$

Where p_i represents the probability of selection of individual i, and f_i represents the fitness of that individual. The summation is calculated for the fitness of all individuals in the current generation.

Using this probability, scaled by a constant to ensure sufficient numbers of individuals selected, we populate a queue of selected individuals which is then passed to crossover. The procedure is illustrated below.

```
Definitions:   selected: queue of individuals
               totalFitness = sum of fitnesses of generation

For individual in population
     if 10*individual.fitness/totalFitness > randomDouble(0,1.0)
          add individual to selected
     End-if
End-for
```

Once the next generation is returned from crossover, a given child from that generation may be mutated with a small random probability of the form:

```
if(randomInteger()%8>6) mutate(child)
```

We then compare each child to the next parent from the selected queue by their fitnesses. If the child has greater fitness, it replaces the parent in the population. We also implement a simulated annealing concept that allows for the selection of a child that is less fit than its parent. The pseudocode below illustrates this further.

```
Definitions:   child = individual in new generation
               parent = individual in previous generation
               deltaFitness = child.fitness – parent.fitness

if(deltaFitness > 0)
      nextGeneration.replace(parent, child)
  else if(exp(deltaFitness/temperature) > randomInteger(5)){
      nextGeneration.replace(parent, child)
End-if
```

Model 2

Model 2 uses a naive selection method which involves traversing the redblack tree representing the population, beginning at the fittest individual and moving sequentially down the tree. Individuals are passed to crossover in pairs – the fittest with the second fit, the third fit with the fourth, and so on. Crossover will then return pairs of offspring. This is done for N/10 individuals, where N is the number of individuals in the current generation. The resulting offspring are then used to replace the least fit individual in the population. Thus, the fittest individuals are not necessarily replaced by their offspring, and may be kept to produce future generations.

```
Definitions: children: array of 2 individuals

For individual in population
     children = crossover(individual, individual.nextFit)
     population.replaceLowest(children[0])
     population.replaceLowest(children[1])
End-for
```

Mutations

Mutations are implemented as illustrated by the following code segment, identically for both models:

```
Definitions:   mutation: random integer from 0 to 2
               position: random integer from 0 to individual size
               push: add node to front of individual
               delete: remove node from individual
               modify: change indices of node at given position
               randI, randJ: random indices


 switch(mutation)
      case 0: individual.push(randI, randJ)
              break
      case 1: individual.delete(position)
              break
      case 2: individual.modify(position, randI, randJ)
              break
```

Crossover

Crossover is implemented through similar methods in both models. We generate a random crossover point ranging from 0 to the size of the given individual. Since individuals are represented as linked lists, we need only traverse the first individual up to the position given by its crossover point and modify the node preceeding the crossover point so that its next node is the first node of the second individual that follows the crossover point of the second individual.

```
Definitions:   temp1 = first node of child1
               temp2 = first node of child2
               savedTemp1Next: Node
               parent1, parent2 = next individual in selected queue
               children: queue of individuals
child1 = parent1
child2 = parent2

For all i=0 to cross1
     temp1 = temp1.next
End-for

For all i=0 to cross2
     temp2 = temp2.next
End-for

savedTemp1Next = temp1.next
temp1.next = temp2.next
temp2.next = savedTemp1Next

children.add(child1)
children.add(child2)
```

The above code segment is the implementation provided in Model 1. In Model 2, the selected queue would be replaced with two selected individuals, and the children queue would instead be an array of two elements.

Application

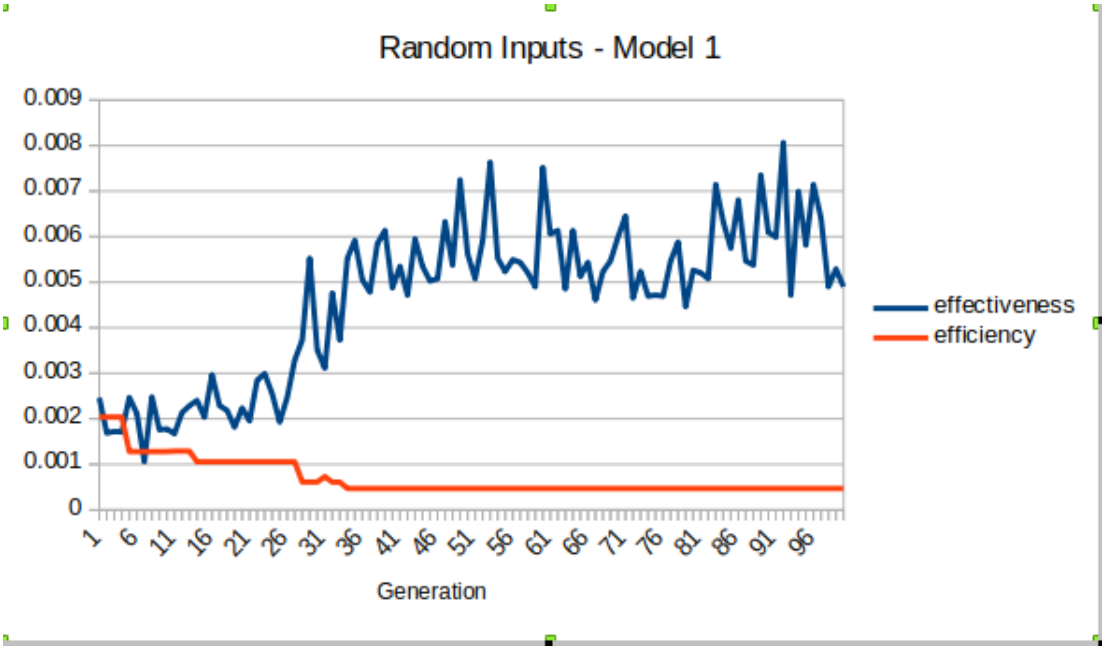
The first generation is initialized as a population of 1000 random individuals, of random size up to 500 nodes, who are subsequently trained on the 99th sublist of the dataset. This is an arbitrary choice. Subsequent generations are evolved from this generation and the fitness of these generations is evaluated based on their output of the sequential sublists of data. That is, generation i is trained on the ith sublist.

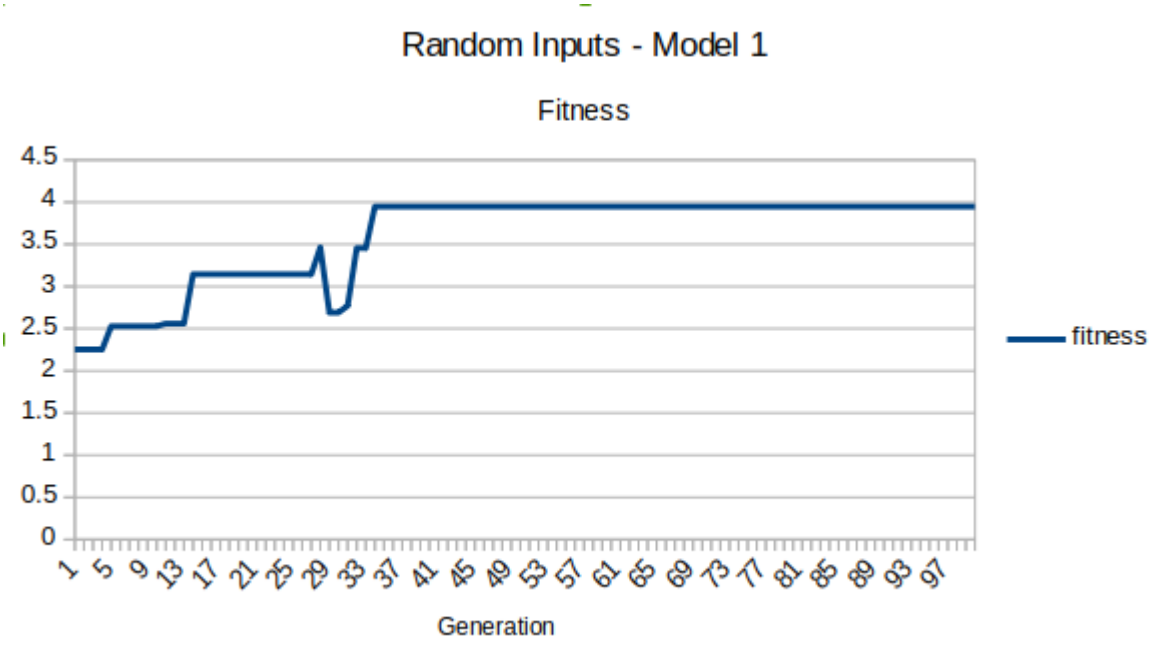
Results

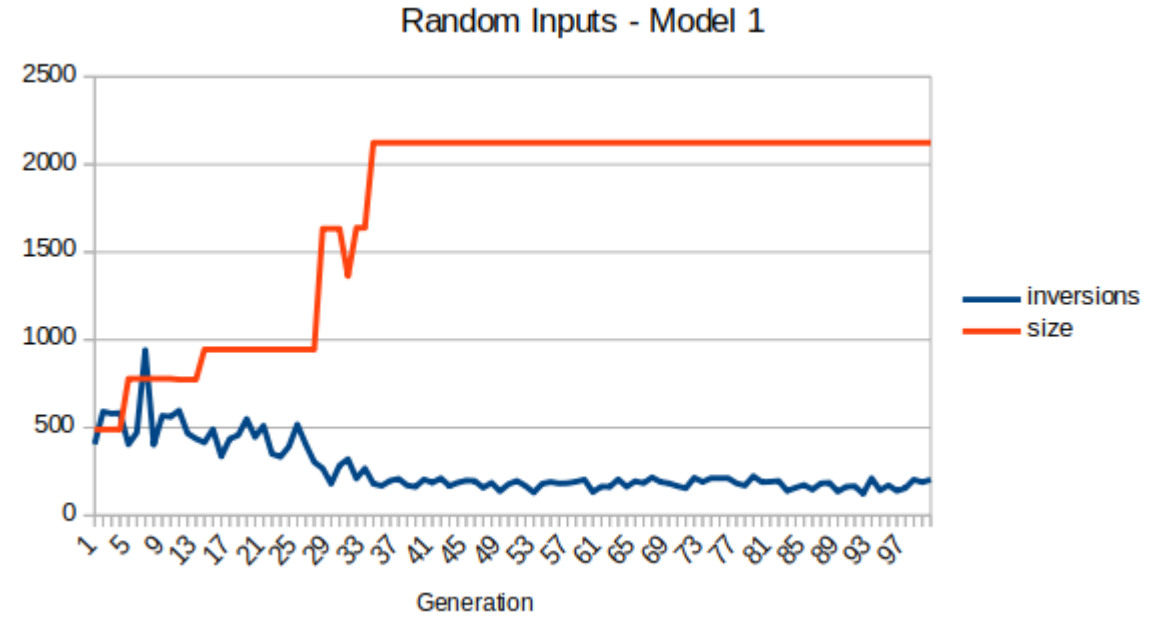
The algorithms introduced above are applied to input datasets following four different distributions: random inputs, small random inputs, nearly sorted inputs, and nearly inversley sorted inputs. We illustrate the results for each of the distributions separately. The graphs that follow plot the efficiency, effectiveness, and fitness of the fittest individual in each generation, by the definitions given in the previous section. We illustrate exhaustive results for Model 1, and introduce some minor results for Model 2 in the discussion.

The results are discussed in the next section.

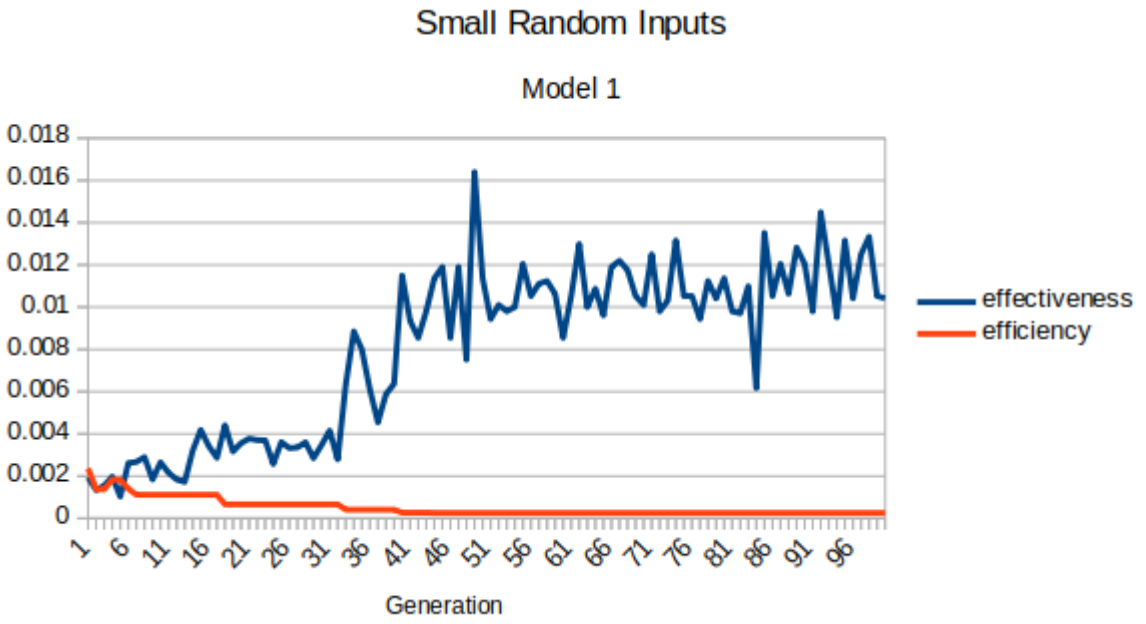
Random inputs

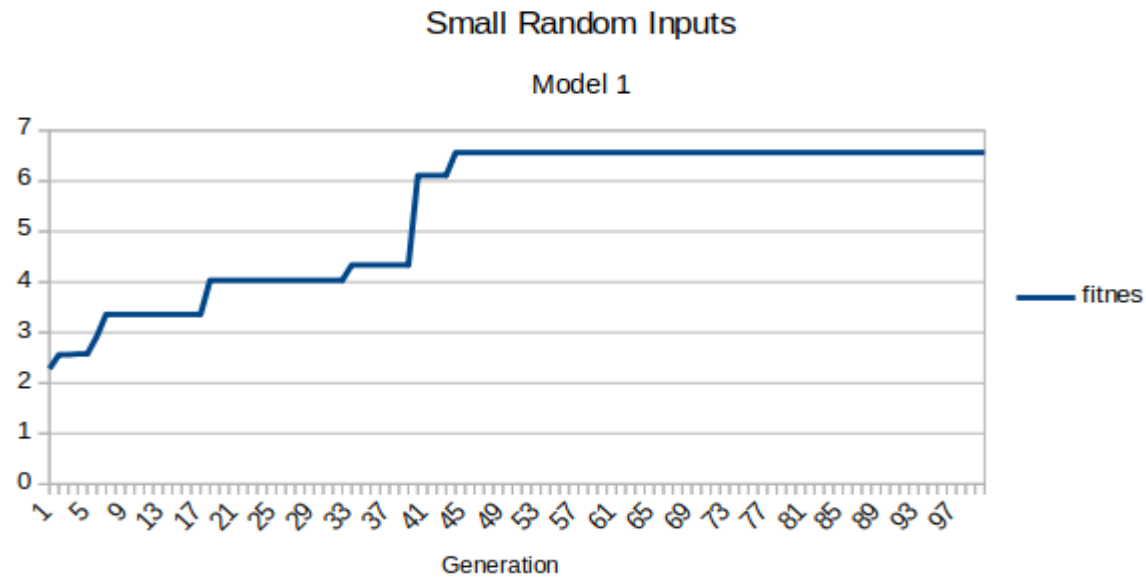


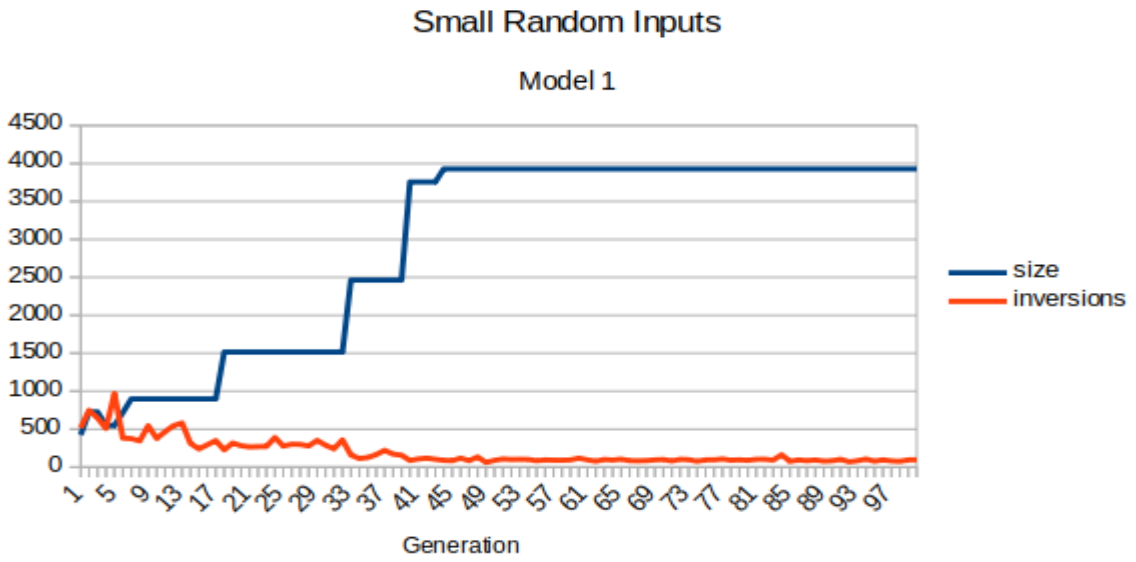

Random Inputs - Model 1
Fitness
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97
Generation
fitness


Random Inputs - Model 1
2500
2000
1500
1000
500
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97
Generation
inversions
size

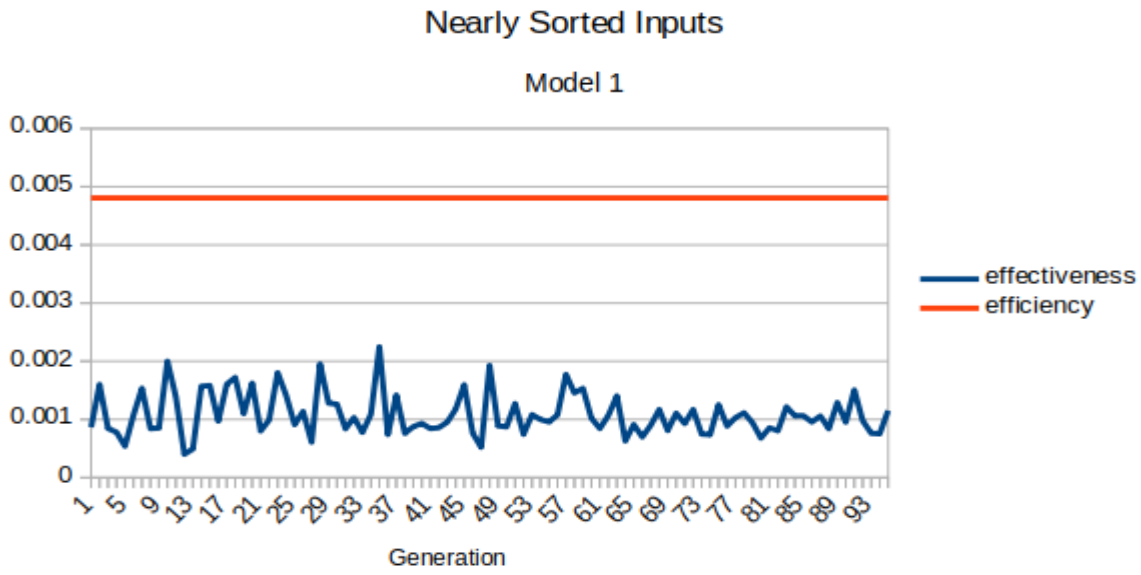
Small random inputs

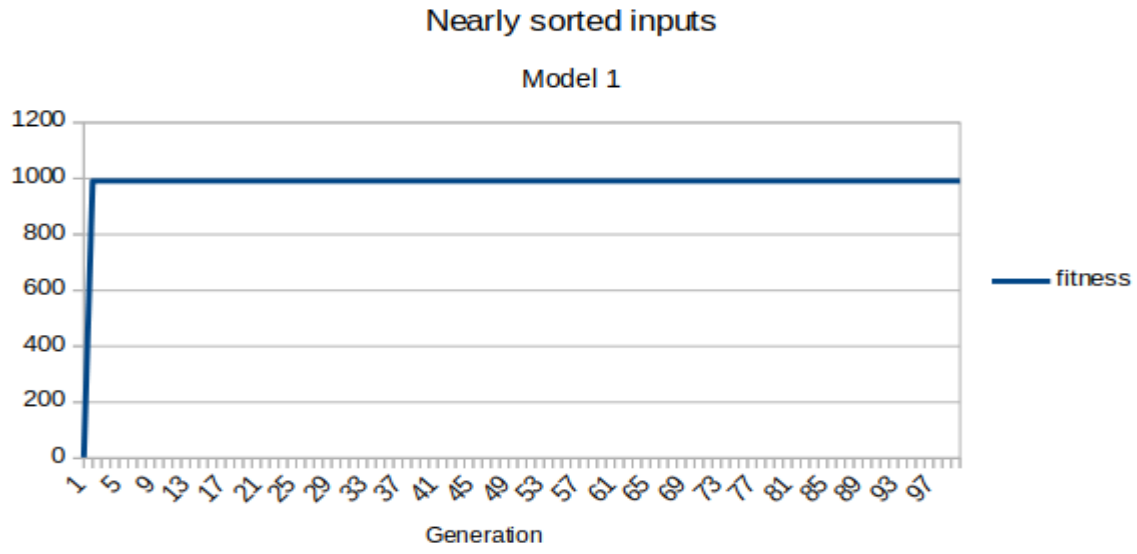


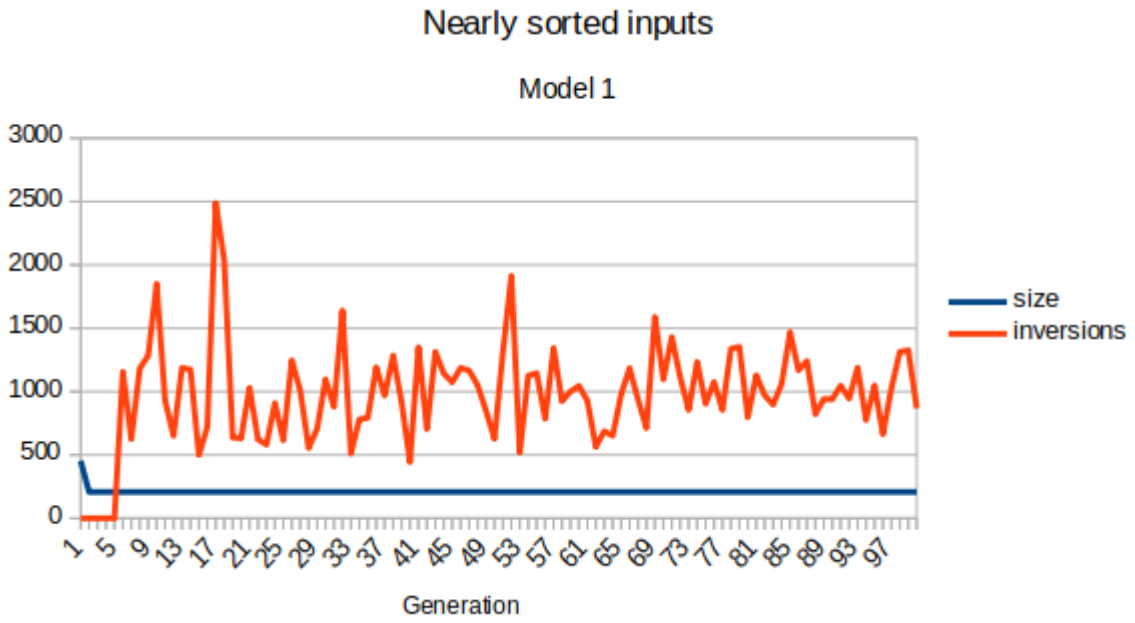




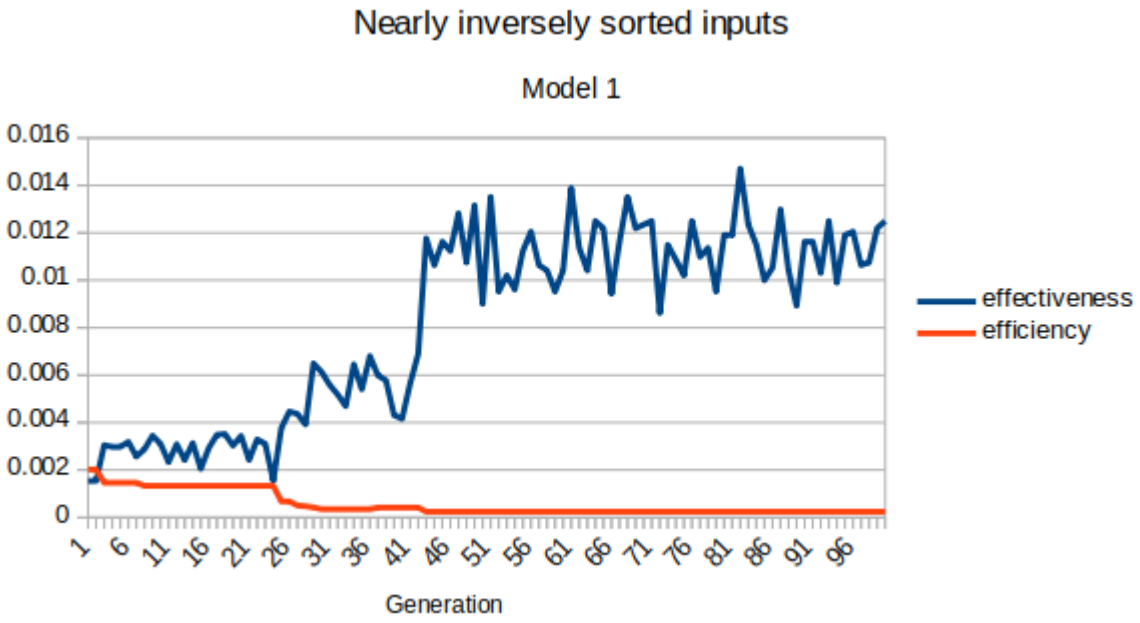
Nearly Sorted Inputs

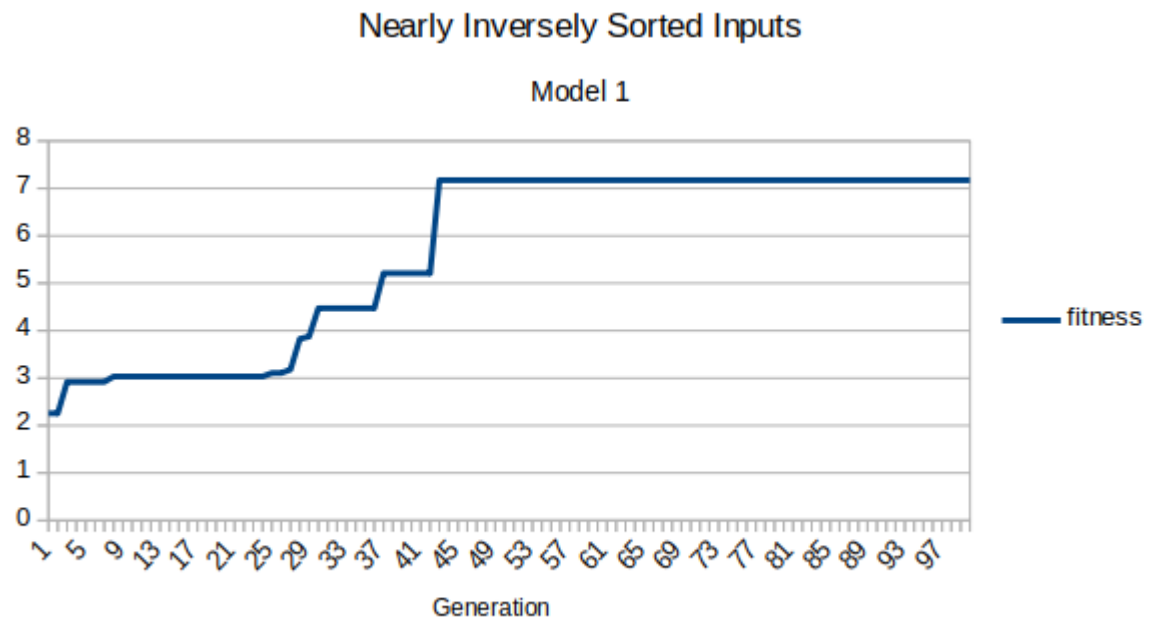


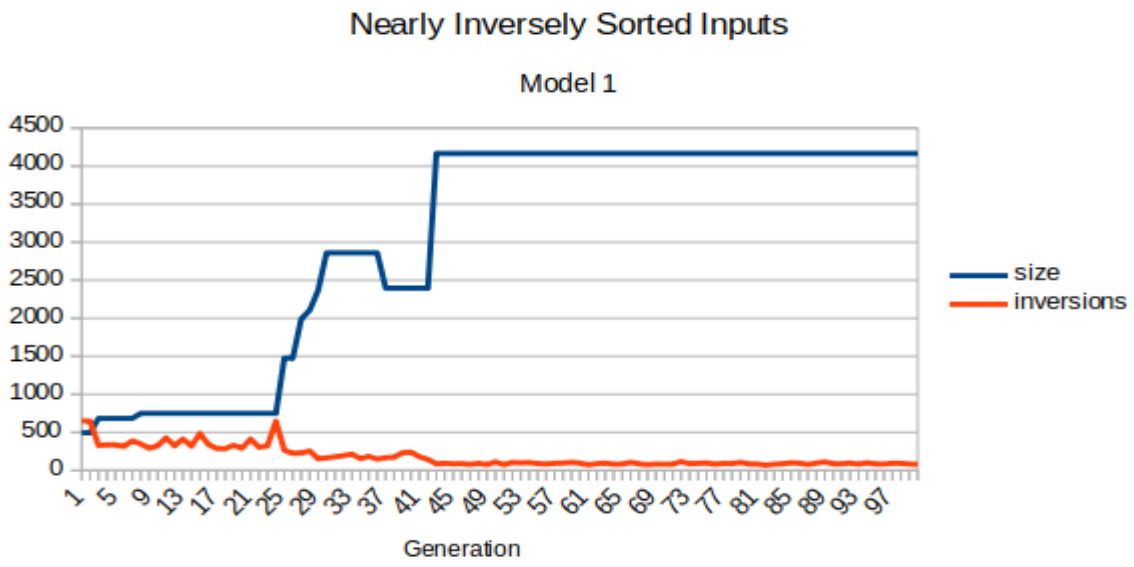




NearlyInversley Sorted Inputs




Nearly Inversely Sorted Inputs
Model 1
8
7
6
5
4
3
2
1
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97
Generation
fitness


Nearly Inversely Sorted Inputs
Model 1
4500
4000
3500
3000
2500
2000
1500
1000
500
0
1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97
Generation
size
inversions

Discussion

The results above indicate several characteristics of the genetic algorithm model that are worth noting. Here, we discuss these characteristics by addressing the various features observed above.

Random inputs

We see that for random inputs and for small random inputs, the model evolves to accumulate traits indiscriminately. This results in a general pattern of increasing effectiveness, and a steady decrease in efficiency. The result is consistent with our intuition, since, without a natural pattern to which the model can adapt, its only means of producing successively fitter offspring is to disseminate as many traits as possible. Despite our attempts to encourage more efficient individuals in later generations, the effectiveness of smaller individuals becomes negligible compared to those of larger ones.

Fitness of individuals trained on random inputs and small random inputs increases in sharp stages in early generations before plateauing in later generations. We can infer that in many cases, an especially large individual is occasionally generated, and remains the fittest individual for several generations to come, by virtue of its size.

The illustrations of inversions and size over successive generations can be interpretted to reaffirm the above statements regarding effectiveness and efficiency.

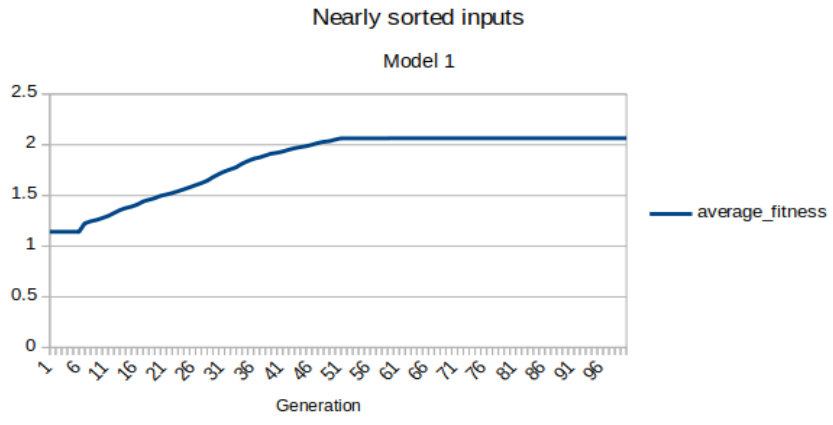
Nearly sorted inputs

The results on sorted inputs highlight the major drawbacks of this impementation. It can be inferred with reasonable certainty that the fittest individuals in the population remain the same for all generations. This can be interpretted as a result of a trial when the model faces a completely sorted input and attributes its sortedness to an individual that was trained on this input.

In fact, upon inspection of the data, we find that Generation 1 is trained on a sorted sublist, and thus individuals from that generation become attributed with a disproportionate fitness score. We also see that these individuals are largely ineffective at sorting further sublists.

This brings to question the fitness evaluation criteria used in this model. In particular, we note two areas in which the critera is lacking. One meaningful inferrence is that using inversions to evaluate sortedness of an array is largely ineffective in the case of near-sorted data. Future iterations of this model should make use of a measure that reflects the number of operations that would be required to get a sequence into sorted order (Estivill-Castro, 1992).

Since the results provided above illustrate only the characteristics of the fittest individual in each generation, they are not especially meaningful in this case. An illustration of average fitness of each generation provides some more insight into the evolution of the model.



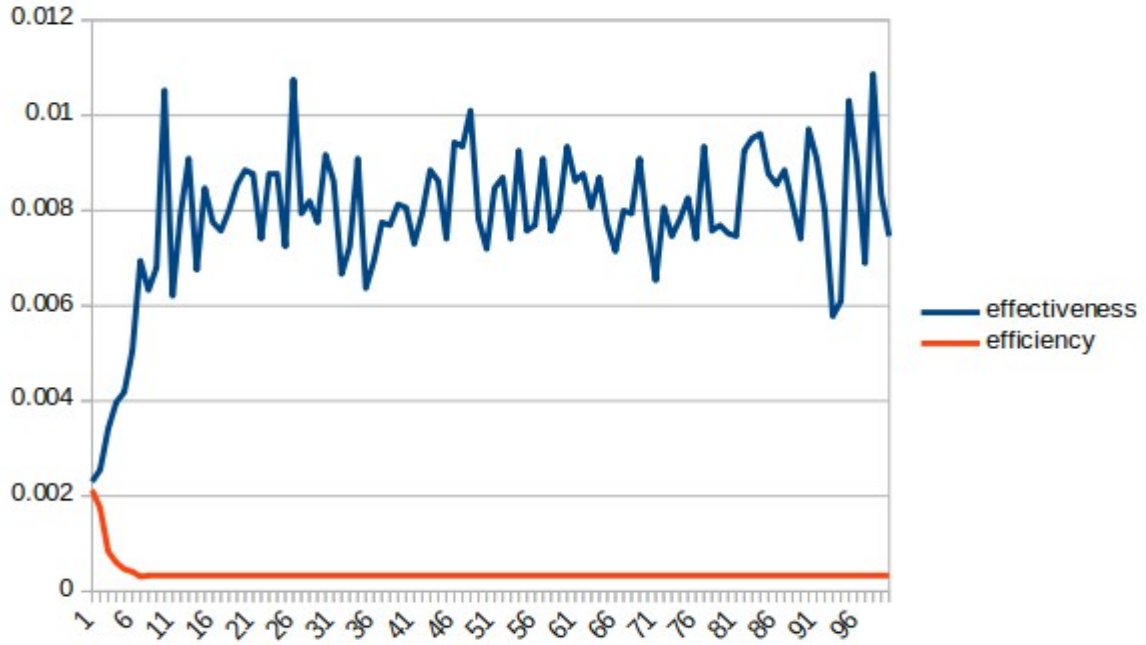
Here, we see that the behaviour of the model is not quite as stagnant as might be expected from the data provided previously. The population tends towards increasing fitness, and is inferred to converge to a local maximum. The considerations regarding convergence are discussed in the next section.

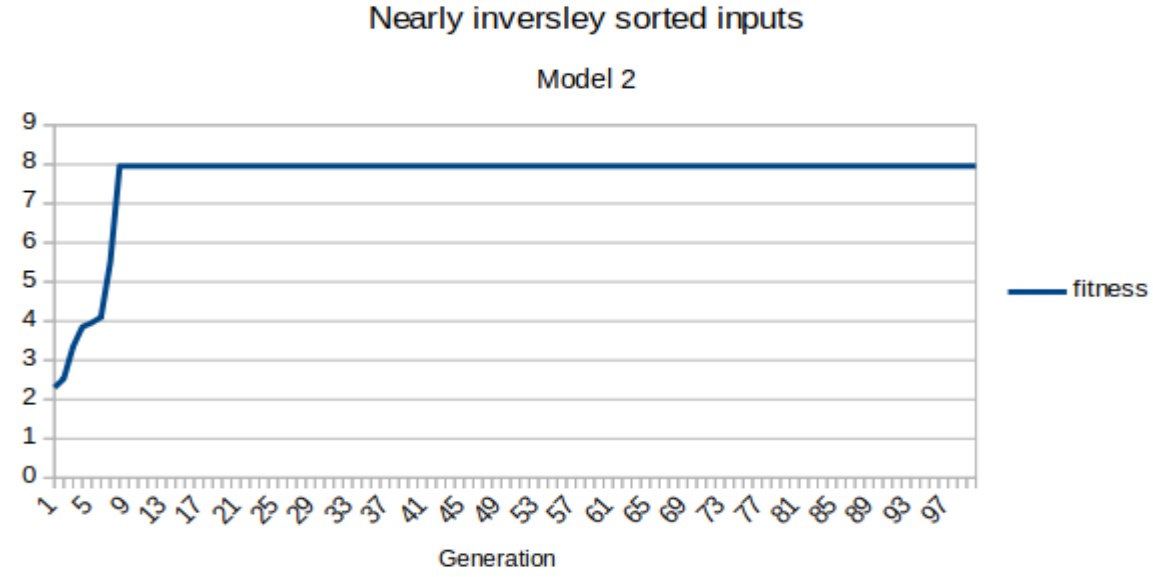
Nearly inversely sorted inputs

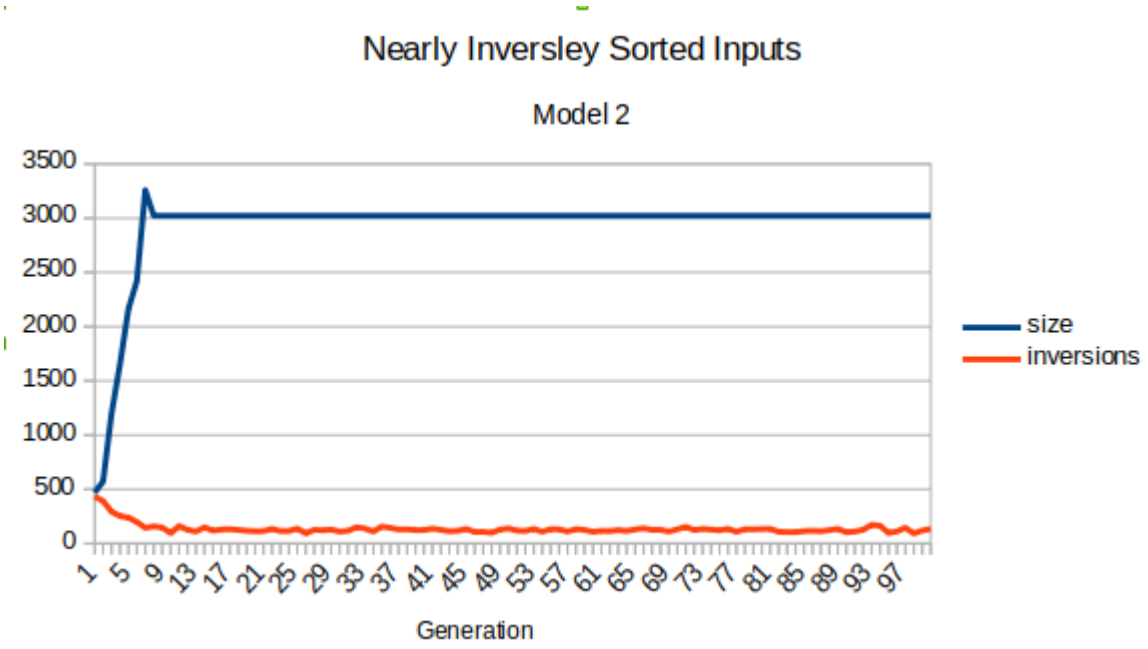
The results for nearly inversely sorted inputs indicate that the model again tends towards a strategy of occupying the solution space undescerningly. We find that the resulting output is nearly sorted, but the final solution generated is notably inefficient.

Comparisons to Model 2

Comparing the primary model to Model 2 provides some insight into the benefits of roulette wheel selection over the selection of the fittest individuals in a population. We do this comparison using the results reflecting the performance of Model 2 on the nearly inversely sorted dataset.







Comparing these results to those of Model 1, we see wider variability in effectiveness of Model 2. More notably, we see indications that the model exhibits a significant lack of diversity, since individuals in this model are not necessarily replaced by their offspring. In fact, in the case of nearly sorted datasets, we find that this feature of the model can be debilitating, resulting in generations of individuals containing minimal numbers of traits.

Conclusion and recommendations

The preceding experiment provides a great deal of insight into the design and function of genetic algorithms. By separating the analysis to scrutinize the various components of the models, we see the importance of including a probabilistic element in the selection criteria. However, even with the implementation of a traditional selection method, we see that a Genetic Algorithm may still be predisposed to converging to local maxima. This can be seen as a product of the selection criteria, but we are also strongly inclined to suspect the method of choice for fitness evaluation.

One significant recommendation for future iterations of the project would be to modify the way in which we encourage the population to favour diversity. Methods investigated thus far in this project for maintaining diversity have proven to be computationally inefficient. A number of options may be considered in the future. For example, Fitness Uniform Selection has been shown to be effective in maintaining genetic diversity (Hutter, 2001). In addition, one of several niching techniques (DeJong, 1975) may be investigated to allow for the pursuit of several peaks simultaneously.

Lastly, as stated previously, the results strongly suggest that the choice of the fitness function was ill-conceived. At present the model uses inversions and ascending runs to represent the pre-sortedness of the data. While this method may be useful in some forms for certain distributions, future work should allow for the algorithm to make inferences about the distribution of the input data before applying a suitable fitness function.

References

Estivill-Castro, V., & Wood, D. (1992). A survey of adaptive sorting algorithms. *ACM Computing Surveys,24*(4), 441-476. doi:10.1145/146370.146381

M. Hutter, (2002) “Fitness uniform selection to preserve genetic Diversity”, In Proceedings of the 2002 Congress on Evolutionary Computation, pp, 783-788.

K. A. DeJong, “An analysis of the behavior of a class of genetic adaptative systems,” Ph.D. dissertation, University of Michigan, Ann Arbor, 1975.