RnaSec: Genetic Algorithm

Jon-Michael Deldin
Department of Computer Science
University of Montana
jon-michael.deldin@umontana.edu

2011-05-31

Contents

1 Introduction

The goal of this project was to write a genetic algorithm that permutes a single secondary structure until the original structure is achieved. This was more of a proof of concept, but the techniques employed may be useful in later stages.

2 Data

The MBE1A aptamer from M. Ellenbecker, J.M. Lanchy, and J.S. Lodmell was the primary sequence used to generate a "connected-table" (CT) file from `mfold`. The CT file was then used to generate a tree data structure.

3 Programmatic Component

This project required significant additions to the existing RnaSec library. The following items were implemented:

- CT parser for input
- support for bulges
- support for internal loops
- pruning
- grafting
- genetic algorithm

3.1 Genetic Algorithm

The genetic algorithm developed is not a true genetic algorithm because it does not use a bit vector representation or crossover. Instead, tree objects are manipulated through prune and graft operations. The only genetic operators are mutation and survival, and all of the diversity is obtained through mutating clones.

The algorithm is presented below.

1. Create initial random population
2. Loop:
   (a) Perform random point mutations on a percentage of the population. This increases the population size.
   (b) Get the fitness of the population. This is determined by taking the nucleotide positions of the source tree and comparing them with another tree. For every position that is different, a 1 is subtracted to the fitness score. A score of 0 indicates the other tree is identical to the source.
   (c) Select the most fit until the population has been reduced.
   (d) See how many are equal to the original.
   (e) Terminate if terminating condition has been met, i.e., $x\%$ of the population is equal to the source.

4 Analysis

The genetic algorithm was able to scramble the tree data structure for MBE1A and return to the original input. I experimented with varying the population size and percent of the population to verify the mutation operator had the greatest effect.

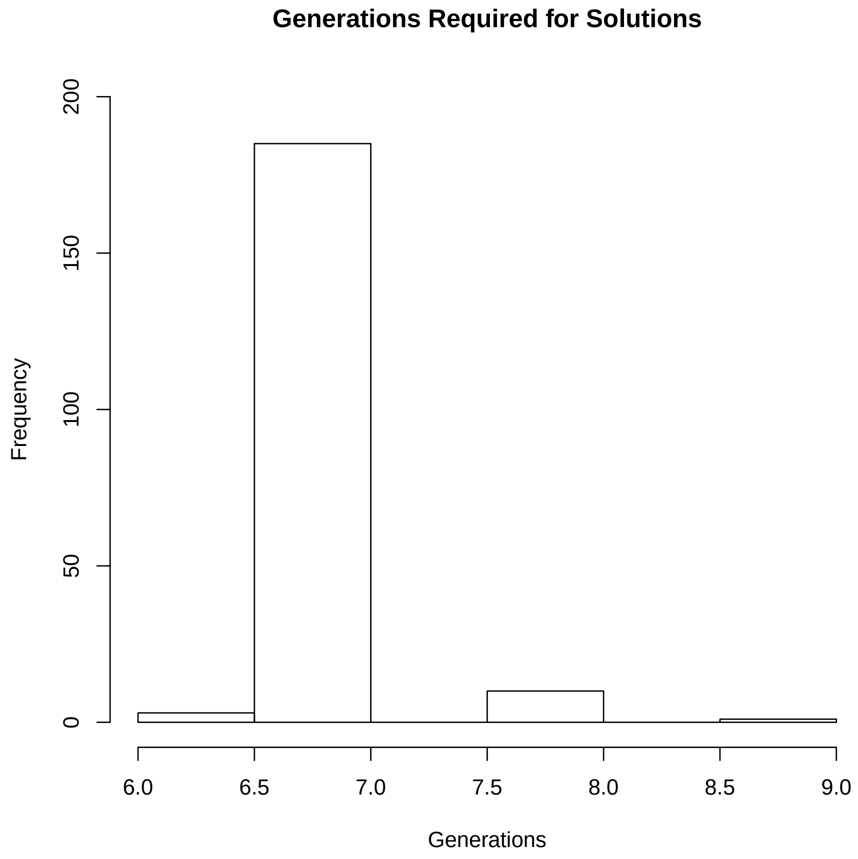

Figure 1: Frequency of generations.

4.1 Impact of Population Size

To determine the effect of varying the population size, sizes from 100–10,000 incremented by 50 were tested (199 test runs) with a mutation rate of 50% and terminating condition of 100% (i.e., every member of the population had to be identical to the original tree).

As Figure 1 shows, population size had little effect on the number of generations required for termination. On average, it took seven generations to terminate as shown in Figure 2. Thus, population size does not have any effect in the presence of a greedy selection algorithm.

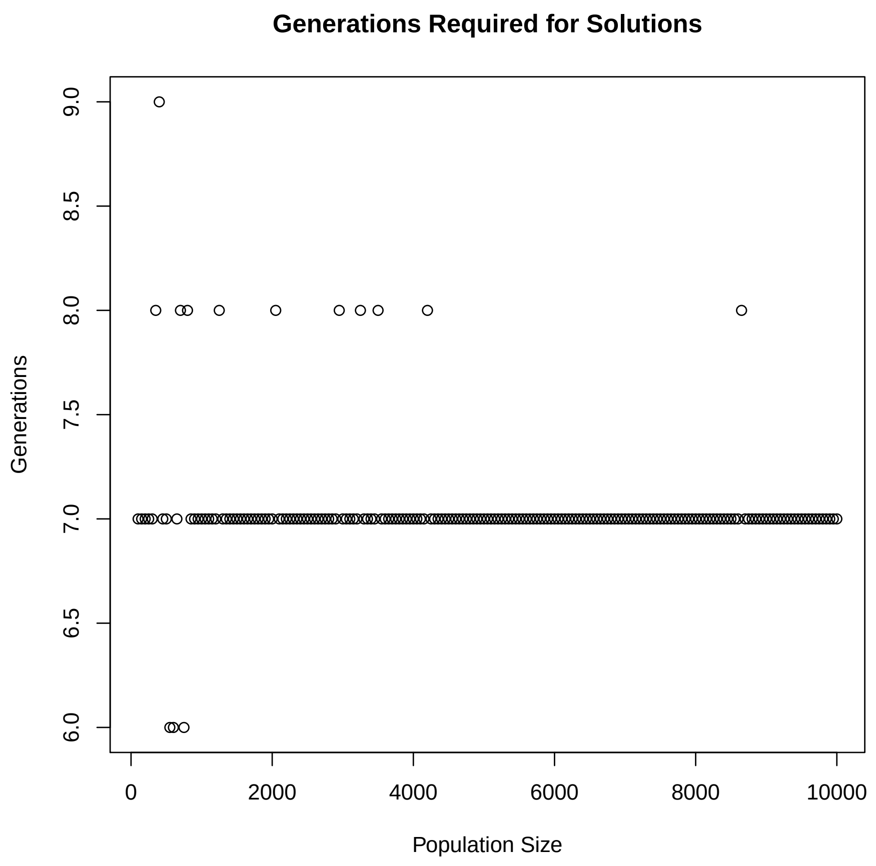

Figure 2: Effect of population size on number of generations required for solutions.

4.2 Impact of Mutation Rate

To determine the effect of varying the mutation rate on the number of generations required to terminate, mutation rates of 0.01–1.0 were tested. In this algorithm, "mutation rate" refers to the percentage of the population that has point mutations performed on it, so a value of 0.1 means 10% of the population will be mutated.

As Figure 3 shows, a higher mutation rate requires fewer generations to terminate. This is logical due to the greedy selection algorithm that selects the members with the top fitness.

5 Conclusion

In conclusion, the genetic algorithm was able to successfully permute MBE1A and return to the original tree data structure using prune and graft operations. This algorithm was primarily a proof of concept for Machine Learning (SP2011), but it was also a good integration test of the CT parser and tree operations. Overall, this was a major stepping stone towards tree-edit distance.

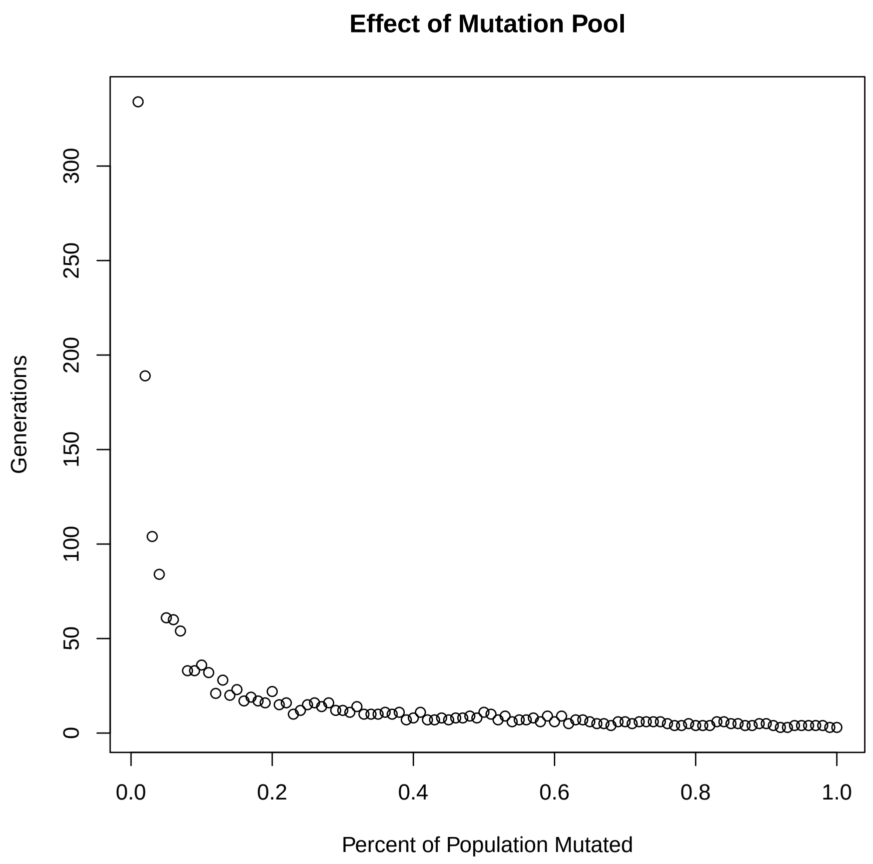

Figure 3: Effect of varying mutation rate on the number of generations required for solutions.