Evolving a Collection of Strings to a Target Sequence using Genetic Programming

Ramón Reszat

December 19, 2019

1 Introduction

Genetic search sequentially evolves a randomly chosen distribution of strings until a subset of the remaining genotypes have converged to a desired goal state. As an example of the types of problems encountered in this field we adapt the infinite monkey theorem [1] to random text output produced as a population of N_p strings, getting cross combined over a number of N_g discrete steps. Introducing a fitness measure for the strings allows the algorithm to choose from the N_p fixed-size sequences and eventually terminate with an approximate solution. Thus the element of evolutionary pressure becomes a driving force to compose the desired sequence.

2 Methods

Using genetic programming the task becomes an optimization problem on the sequences described by the formal grammar $G = (N, \Sigma, P, S)$.

$$\Sigma = \{a, b, c, ..., x, y, z\}$$

$$S \to \{c_i \epsilon \Sigma\}_l$$

The corresponding criterion can be defined as the error between the words of the language generated by G and the target sequence. Their Hamming distance [2, p. 8f.], the number of positions at which they differ, provides a measure of similarity and ensures that a fitness score s can be assigned to each element of $L(G)$. Please see appendix B. Assuming the hit probability per character of Σ remains $p = 1/\Sigma$ and $l = 10$ is the sequence length, the fitness scores are binomially distributed over the initial population.

$$B(s; l, p) = p^s (1 - p)^{l-s}$$

To pick a crossover operation [3] consider the longest common sequence [4] between two arbitrary words of $L(G)$.

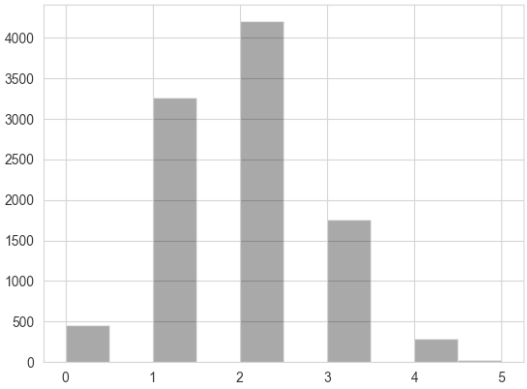

Figure 1: longest common sequence with goal state in $N_p = 10000$

We expect to find two consecutive letters in each word that match the target sequence (Figure 1). In terms of the underlying grammar this means we can add a substructure to the production rules of G such that

$$S \to \{A\}_N$$

$$A \to \{c_i\}_{l/N}$$

In this case the alignment of the resulting genes matters for matching to the target sequence. Therefore we choose a random locus for the

split und interchange at this position to produce two offsprings. This keeps the population size constant.

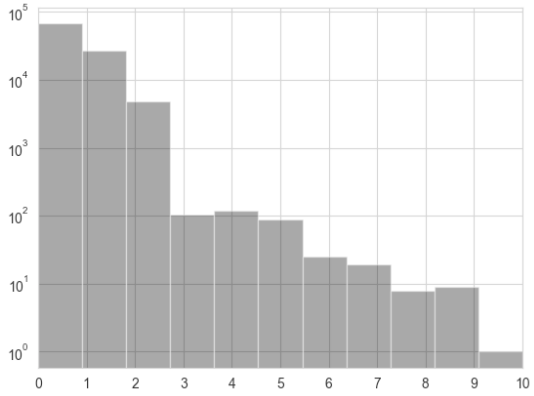

Figure 2: fitness distribution over $N_p = 100000$ at $N_g = 20$ $(k = 400, r = 0.2)$

For candidate selection we sort the population by fitness score and take the k best for cross combination.

Additionally mutation is introduced to switch a random position of the sequence to an arbitrary element of the alphabet Σ. The chance of its occurance is denoted by the mutation rate r.

3 Results

The parameters k and r introduce diversity [3] into the population at each iteration step by manipulating input and output of the elementary genetic operation. Choosing a sufficiently high k moves a bigger part of the population towards higher fitness scores (Figure 2). Please also refer to Figures 4-7 in appendix D.

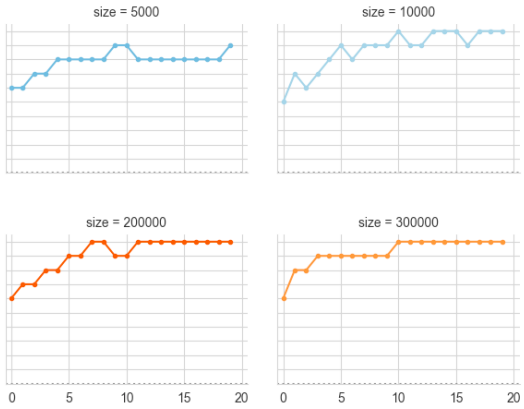

Figure 3: fitness score s after N_g steps with varying population size N_P $(k = 400, r = 0.2)$

Larger N_p results in the target sequence being more stable over a bigger number of steps (Figure 3).

Changing the mutation rate from $r = 0.1$ to $r = 0.3$ increases the probability to reach the target sequence faster within a smaller population N_p. At the same time a higher mutation rate results into an instable best solution after further generations even for large N_p. Compare Figures 8-11 in appendix E.

References

[1] Émilie Borel. Mécanique statistique et irréversibilité. *Journal de Physique*, 3:189–196, 1913.

[2] Richard Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29:147–160, 1950.

[3] Dana Ballard. *An Introduction to Natural Computation*, chapter 12, pages 277–289. MIT Press, 1997.

[4] David Sankoff Vacláv Chvátal. Longest common subsequences of two random sequences. *Journal of Applied Probability*, 12:306–315, 1975.

A Genetic Algorithm

```
def evolve(genotypes, select, mut_rate):
   selected_genotypes, suppressed_genotypes = select_k_best(genotypes, k=select)
   print(selected_genotypes[0])
   evolved_genotypes = crossover(selected_genotypes, locus=random.randrange(1,5))
   genotypes = evolved_genotypes + suppressed_genotypes
   genotypes = snp_mutation(genotypes, rate=mut_rate)
   return genotypes
```

B Fitness Function

```
def fitness_score(individual):
   # scale fitness between 0 and 10
   return len(individual) - hamming_distance(individual, target)
```

```
def hamming_distance(ind1, ind2):
   if len(ind1) == len(ind2):
      count = 0
      for i in range(len(ind1)):
         if ind1[i]!=ind2[i]:
            count+=1
      return count
```

C Crossover Function

```
def crossover(genotypes, locus=3, n_loci=5):
   children = []
   for i in range(int(len(genotypes)/2)):
      x = genotypes.pop()[1]
      y = genotypes.pop()[1]
      pos = locus * len(x)//n_loci
      children.append(x[:pos]+y[pos:])
      children.append(y[:pos]+x[pos:])
   return list(map(lambda ind: (fitness_score(ind), ind), children))
```

D K-best Selection

```
def select_k_best(genotypes, k):
    genotypes.sort(reverse=True, key=lambda ind: ind[0])
    selected = genotypes[:k]
    suppressed = genotypes[k:]
    return selected, suppressed
```

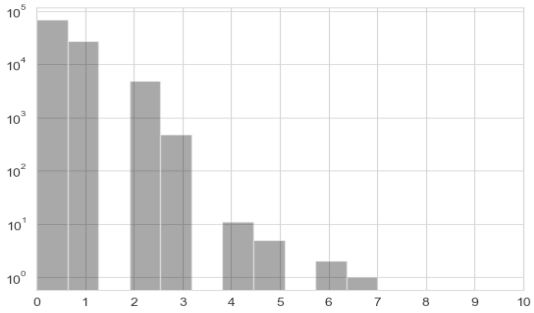

Figure 4: $N_g = 20, k = 25, r = 0.2$

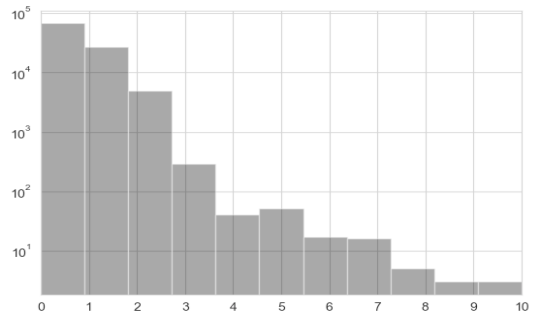

Figure 5: $N_g = 20, k = 200, r = 0.2$

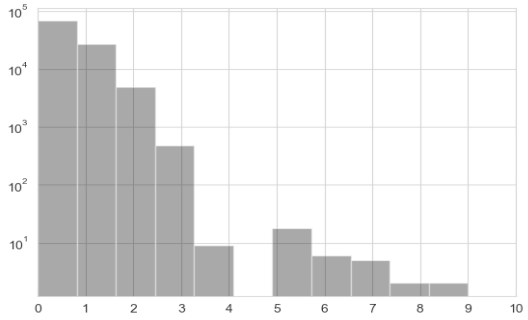

Figure 6: $N_g = 20, k = 50, r = 0.2$

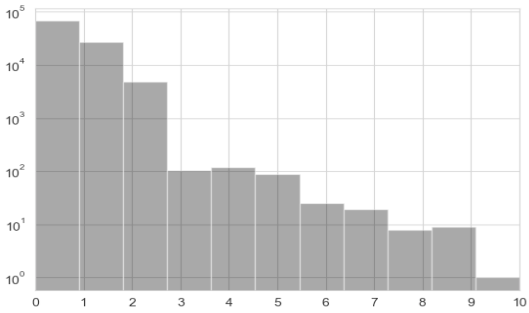

Figure 7: $N_g = 20, k = 400, r = 0.2$

E SNP Mutation

```
def snp_mutation(genotypes, rate=0.2):
   for genotype in genotypes:
      if(random.random() < rate):
         genotype[1][random.randint(0,len(genotype[1])-1)] =
              random.choice(alphabet)
   return list(map(lambda ind: (fitness_score(ind[1]), ind[1]), genotypes))
```

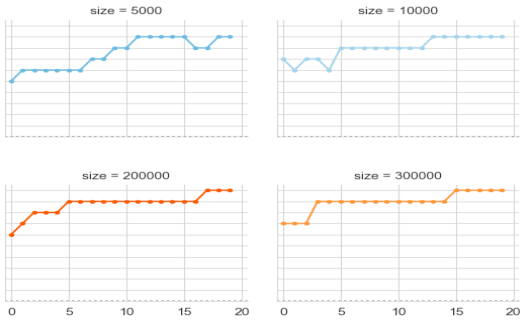

Figure 8: $k = 200, r = 0.1$

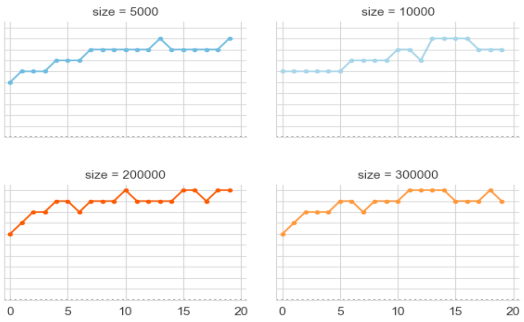

Figure 9: $k = 200, r = 0.3$

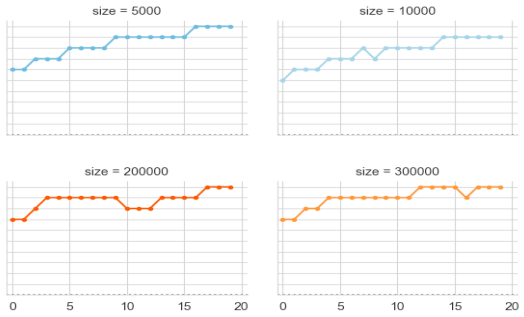

Figure 10: $k = 200, r = 0.2$

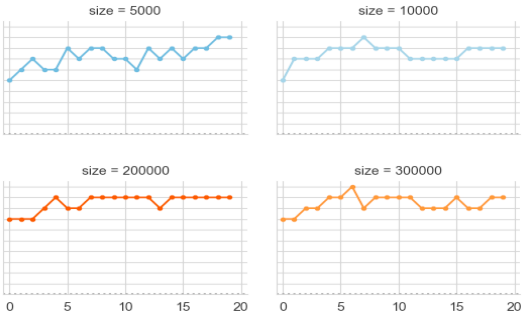

Figure 11: $k = 200, r = 0.5$