Long-term bacterial adaptation to a constant environment as a model for genetic algorithm design

Benjamin D. Lee1,2,3,✉

Evolutionary algorithms are a class of computational techniques for problem solving based on Darwinian evolution. These methods are centered on increasing the “fitness” of a population of candidate problem solutions through successive generations exposed to the forces of selection, mutation, and recombination until a solution is found. Using experimental evolutionary results from bacterial populations adapting to a constant environment, a novel genetic algorithm experimental model was tested, yielding an 18% reduction in the amount of computation required to converge on an optimal solution for a baseline problem. These results support the applicability of experimental evolution to evolutionary algorithm research and warrant further investigation.

1 In-Q-Tel Lab41  
2 Harvard Medical School  
3 School of Engineering and Applied Sciences, Harvard University

✉ Correspondence: [Benjamin D. Lee <benjamindlee@me.com>](mailto:benjamindlee@me.com)

# Introduction

Algorithm development is one of the central research areas of computer science. One frequent source of inspiration for algorithm design is biology in general and evolution in specific. Indeed, some computationally intractable problems can be approximately solved using evolutionarily inspired methods, broadly termed evolutionary algorithms.

One specific and widely used evolutionary algorithm method is genetic algorithms (Holland 1992). To imitate evolution, genetic algorithms attempt to solve problems by increasing the fitness of (*i.e.* adapting) a population of breeding individuals, subject to the effects of mutation, recombination, and directional selection, over multiple generations. A genetic algorithm is centered on the optimization (minimization or maximization) of a real-valued fitness function , by which an individual in the population is evaluated. An individual is defined by their “genome”, which a list of “genes” (*i.e.* numbers), that can be fed into to yield another number which represents the individual’s fitness. After the creation of the initial population (usually with random genomes), each individual is evaluated with respect to . Then, a selection function is applied over the population to determine which individuals to use as the basis of the next generation. One common selection method is called *tournament selection*, in which random subsetss of the population are chosen, with the fittest member of the subset being propogated. This allows for fit, albeit not the fittest, genes to persist in the population. Often, *elitism* is used, which ensures that the fittest individual in the population is persevered identically into the next generation, thereby guaranteeing that the fitness function is monotonically increasing between generations. After the selection of individuals, a recombination function can be applied to generate members of the next generation, which takes two parents and yields two children. Additionally, a mutation function can be applied to individuals to introduce further stochasticity. This process is repeated until a stopping criterion is met. The fittest member of the final population is taken as the final result of the optimization.

Even in its basic formulation, there are a variety of parameters left to the genetic algorithm’s user, such as the appropriate fitness function, population size, genetic representation, population size, selection function, mutation function and rate, recombination function and rate, and stopping criterion. Furthermore, due to the inherent randomness of the initial population, multiple populations must often be evolved independently to yield an acceptable solution. Thus, a genetic algorithm optimization experiment often consists of multiple randomly initialized populations evolving in parallel, thus increasing the chances of finding an acceptable solution.

Unfortunately, genetic algorithms are not reflective of the biology *in natura*. Biological systems generally do not evolve in a controlled, unchanging environment. Furthermore, populations’ genomes are not completely random, but rather highly interrelated and conserved. Indeed, this discrepancy has been one of the major areas of criticism for genetic algorithms. For example, Skiena (2008) described the use of genetic algorithms as “pseudobiology” and “voodoo”. Additionally, the often-large number of generations required to converge on an solution means that numerous invocations of the fitness function are required. If the fitness function is costly to compute, genetic algorithms can be prohibitively slow to compute.

However, there have been experiments experiments which very closely mirror that of genetic algorithms, namely the long-term evolution experiment (LTEE) and its offshoots (Lenski and Travisano 1994). In the LTEE, twelve identical populations of asexual *Escherichia coli* have been allowed to evolve in parallel for tens of thousands of generations in a medium containing limited nutrients to study bacterial adaptation to a constant environment. Each day, 1% of the population are moved to a flask containing replenished albeit equally limited nutrients. Throughout the course of the experiment, viable samples have been frozen at 500 generation intervals, allowing biological “time travel” (Good et al. 2017). One population of *E. coli* evolved the ability to metabolize citrate, a nutrient whose nonmetabolizability is a hallmark of the species, thereby dramatically increasing its fitness. While a full overview of the experiment is beyond the scope of this paper, one key insight from the LTEE is that identical starting populations can, without recombination, yield populations with significantly diverging fitnesses. In addition, the fitnesses of the populations have continued improving without bound for 50,000 generations (Wiser, Ribeck, and Lenski 2013), seeming to follow a power-law relation.

The implications of this insight are twofold with respect to genetic algorithms. First, the ability to yield different results from the same starting population means that repeating the evolution process on the same population should be investigated, in contrast to the standard genetic algorithm approach of using numerous completely randomized initial populations. Second, by caching the fitness of each individual evaluated, the optimization of costly fitness functions can potentially be improved by reducing the amount duplicated computation.

# Methods

As an experimental test of the applicability of the LTEE results to evolutionary computation, we will use the counting ones problem to examine the behavior of genetic algorithms. The counting ones problem is a model optimization problem in which the goal is to create a genome consisting entirely of 1s. The fitness function is formally defined as

where is a genome of size . The mutation function is defined as flipping a random bit in and the stopping criterion is when . Genomes are initialized at random. This optimization problem is widely used in genetic algorithm research because the optimal solution is known and the fitness function is trivially implementable.

Three experimental groups were tested, each of which contained twenty populations of one hundred individuals with one hundred genes (). That multiple populations were used was intended to be reflective of a real-world genetic algorithm use case. The first experimental group was a baseline of traditional genetic algorithms in which twenty independent randomly initialized population of size 100 were optimized (“control”). The second was a set of twenty initially identical populations of size 100, each of which evolved independently (“LTEE”). The third group was an intermediate of the first two, and consisted of four different subgroups of five identical populations, each of size 100. Each group was allowed to evolve until the optimal solution was found (“LTEE-subpopulations”). Four mutation rates (0.01, 0.02, 0.03, and 0.04) were tested, as these have been shown to have the fastest convergence to the global optimum when (Thierens 2002). To characterize the effect of caching costly fitness functions on the different groups, the number of unique individuals in each group was counted. The experiment was repeated twenty times for each mutation rate and experimental group.

To analyze the results, a one way ANOVA was carried out to determine whether the groups had the same mean fitness function invocation count and unique fitness function invocation count. Then, a Tukey HSD test (Tukey 1949) was carried out with a threshold of  to identify whether mean unique fitness function invocation count differed between pairs of experimental groups.

The experiments were carried out using the pyeasyga genetic algorithm software library version 0.3.0 (https://github.com/remiomosowon/pyeasyga), and Python 3.7.3. The data were analyzed using the pandas data analysis library version 0.24.2 (https://github.com/pandas-dev/pandas), the scipy scientific computing library version 1.2.1 (https://www.scipy.org), the numpy scientific computing library version 1.16.2 (https://www.numpy.org), and the statsmodels statistical analysis library version 0.9.0. The full experimental and analytical code is included in the appendix.

# Results

The LTEE-based models both performed comparably to the baseline in terms of mean fitness function invocation count required for convergence to the optimum while requiring significantly fewer unique fitness function invocations on the counting ones problem. Overall, the control required a mean total fitness function invocations to converge, in comparison to invocations for the LTEE group, and invocations for the LTEE-subpopulations group, which was not statistically significantly different. However, the mean unique function invocation count differed significantly between the groups, with the control requiring  invocations to converge, in comparison to  and ) invocations for the LTEE and LTEE-subpopulation groups, respectively. This corresponds to a decrease of 18.5% and 13.4% unique function invocations for the LTEE and LTEE-subpopulation groups, respectively. A more detailed breakdown of the mean fitness function invocation count and the mean unique fitness function invocation count by mutation rate can be seen in the table below.

|  |  |  |
| --- | --- | --- |
| Group | Mean fitness function invocation count (std) | Mean unique fitness function invocation count (std) |
| Control, mutation rate = 0.01 | 1024180 (50925) | 9850 (244) |
| mutation rate = 0.02 | 577885 (32931) | 10959 (228) |
| mutation rate = 0.03 | 413755 (14768) | 11826 (211) |
| mutation rate = 0.04 | 336090 (13158) | 12799 (272) |
| LTEE, mutation rate = 0.01 | 994680 (49232) | 7680 (314) |
| mutation rate = 0.02 | 565600 (26572) | 8855 (330) |
| mutation rate = 0.03 | 407780 (23970) | 9763 (367) |
| mutation rate = 0.04 | 337775 (13086) | 10722 (363) |
| LTEE-subpopulations, mutation rate = 0.01 | 991435 (52354) | 8210 (269) |
| mutation rate = 0.02 | 576615 (29386) | 9385 (228) |
| mutation rate = 0.03 | 419345 (15192) | 10426 (215) |
| mutation rate = 0.04 | 337185 (12625) | 11319 (299) |

The one-way ANOVA analysis yielded an  static of 0.039 and a  value of 0.961 for comparison between the total function invocations between the groups, failing to reject the null hypothesis that the means of the groups differ. However, a one-way ANOVA analysis of the unique function invocation count yielded an static of 69.525 and a value < .0005, rejecting the null hypothesis that the means of the groups differ. The Tukey HSD pairwise comparison of the unique function invocation count yielded significant () differences between all pairs.

# Discussion

The results show that the LTEE-based models are able to match the baseline performance in terms of mean number of generations required to converge while requiring significantly fewer unique fitness function invocations. This is an important consideration because, as the cost of calculating the fitness function increases, the importance of reducing the number of unique fitness function invocations grows faster than the importance of the number of generations required for convergence. This is due to the fact that fetching cached genome fitnesses requires significantly fewer computational resources ( computational complexity) than calculating them.

Furthermore, this experiment demonstrates the applicability of long-term bacterial evolution experimental results to that of genetic algorithm research. One area for future investigation is dynamic population sizes such that, as the fitness of the population increases, so does its population. One potential way to increase the population is to include clones of individuals in proportion to the individuals’ fitnesses. In addition, the effects of random selection as compared to tournament selection would also be subject to reinvestigation, as random selection would select fit individuals at a greater rate by virtue of their overabundance in the population.

One limitation of this study is that it eliminated recombination, which is often used to speed up convergence. In the long-term bacterial evolution experiments, recombination has been shown to be able overwhelm selection (Maddamsetti and Lenski, n.d.). This does not generally appear to be the case in genetic algorithm populations, although further investigation into the effect of recombination is warranted, particularly to what types of fitness function, if any, recombination damages the performance of genetic algorithms on.

The mutation rate is of particular biological interest since, unlike the fitness function, there is a direct parallel to actual an actual, measurable biological metric. There are a variety of open research questions about genetic algorithm mutation rates, such as what the optimal mutation rate for a genetic algorithm optimization problem is (Thierens 2002). Furthermore, should this rate vary throughout the course of the experiment? If so, how? The answers to these questions could potentially be elucidated by examining the LTEE. In it, some populations became hypermutators (Good et al. 2017). One area for experimentation could be to use multiple (and adaptive) mutation rates in different populations when conducting a genetic algorithm optimization experiment.

As the LTEE progresses and more results are generated, more inspiration for genetic algorithm development may be created, allowing for further refinement to this valuable model of computation.

# Conclusions

This paper introduces two key findings for evolutionary algorithm research. First, it connects the computational model of evolution used in genetic algorithms to long-term bacterial evolution experiments, which it mirrors more closely than evolution in natural environments. This development provides a clear body of experimental data from which to identify sources of inspiration that may be more readily translated into computational advances. Second, using the long-term bacterial evolution experiment model as a basis for computational experimentation, a novel method for reduction in the required number of unique fitness function invocations was identified and validated using the counting ones problem.

The project’s software repository and raw data may be viewed at https://github.com/Benjamin-Lee/oeb278-paper.

# Appendix

Experimental code:

#!/usr/bin/env python  
# coding: utf-8  
  
# In[6]:  
  
  
from pyeasyga.pyeasyga import GeneticAlgorithm  
import random  
  
genome\_size = 100  
population\_size = 100  
  
def create\_individual(data):  
 return [random.randint(0, 1) for \_ in range(genome\_size)]  
  
  
# In[66]:  
  
  
def basic\_population(mutation\_probability):  
 results = []  
 cached = set()  
   
 def fitness (individual, data):  
 if tuple(individual) not in cached:  
 cached.add(tuple(individual))  
  
 return sum(individual)  
  
 for \_ in range(20):  
 ga = GeneticAlgorithm(create\_individual(None),   
 crossover\_probability=0,   
 population\_size=population\_size,  
 mutation\_probability=mutation\_probability)  
 ga.fitness\_function = fitness  
 ga.create\_individual = create\_individual  
  
 generations = 1  
 ga.create\_first\_generation()  
  
 while ga.best\_individual()[0] < genome\_size:  
 generations += 1  
 ga.create\_next\_generation()  
   
 results.append(generations)  
 return results, len(cached), sum(results)\*population\_size  
  
  
# In[68]:  
  
  
def ltee\_population(mutation\_probability):  
 results = []  
 cached = set()  
   
 def fitness (individual, data):  
 if tuple(individual) not in cached:  
 cached.add(tuple(individual))  
  
 return sum(individual)  
   
 ga = GeneticAlgorithm(create\_individual(None),   
 crossover\_probability=0,   
 population\_size=population\_size,  
 mutation\_probability=mutation\_probability)  
 ga.fitness\_function = fitness  
 ga.create\_individual = create\_individual  
  
  
 ga.create\_first\_generation()  
   
 starting\_population = ga.current\_generation  
 for \_ in range(20):  
 ga = GeneticAlgorithm(create\_individual(None),   
 crossover\_probability=0,   
 population\_size=population\_size,  
 mutation\_probability=mutation\_probability)  
 ga.fitness\_function = fitness  
 ga.create\_individual = create\_individual  
  
 generations = 1  
 ga.current\_generation = starting\_population  
  
 while ga.best\_individual()[0] < genome\_size:  
 generations += 1  
 ga.create\_next\_generation()  
   
 results.append(generations)  
 return results, len(cached), sum(results)\*population\_size  
  
  
# In[71]:  
  
  
def ltee\_subpopulations(mutation\_probability):  
 results = []  
 cached = set()  
   
 def fitness (individual, data):  
 if tuple(individual) not in cached:  
 cached.add(tuple(individual))  
  
 return sum(individual)  
   
 for i in range(5):  
   
 ga = GeneticAlgorithm(create\_individual(None),   
 crossover\_probability=0,   
 population\_size=population\_size,  
 mutation\_probability=mutation\_probability)  
 ga.fitness\_function = fitness  
 ga.create\_individual = create\_individual  
  
  
 ga.create\_first\_generation()  
  
 starting\_population = ga.current\_generation  
 for \_ in range(4):  
 ga = GeneticAlgorithm(create\_individual(None),   
 crossover\_probability=0,   
 population\_size=population\_size,  
 mutation\_probability=mutation\_probability)  
 ga.fitness\_function = fitness  
 ga.create\_individual = create\_individual  
  
 generations = 1  
 ga.current\_generation = starting\_population  
  
 while ga.best\_individual()[0] < genome\_size:  
 generations += 1  
 ga.create\_next\_generation()  
  
 results.append(generations)  
 return results, len(cached), sum(results)\*population\_size  
  
  
# In[109]:  
  
  
results = []  
trials = 20  
  
methods = [basic\_population, ltee\_population, ltee\_subpopulations]  
mutation\_rates = [0.01, 0.02, 0.03, 0.04]  
  
for mutation\_rate in mutation\_rates:  
 print(f"Mutation rate {mutation\_rates.index(mutation\_rate) + 1}/{len(mutation\_rates)}")  
 for i in range(trials):  
 print(f"\tTrial {i+1}/{trials}")  
 for method in methods:  
 print(f"\t\tMethod {methods.index(method) + 1}/{len(methods)}")  
 result = method(mutation\_rate)  
 results.append(dict(method=method.\_\_name\_\_,  
 mutation\_rate=mutation\_rate,  
 generations=result[0],  
 unique\_individuals=result[1],  
 total\_fitness\_invocations=result[2]))  
  
  
# In[110]:  
  
  
import pandas as pd  
  
  
# In[111]:  
  
  
df = pd.DataFrame.from\_dict(results)  
  
  
# In[112]:  
  
  
df.to\_csv("results.csv")

Analysis code:

# In[140]:  
  
  
df = pd.DataFrame.from\_csv("results.csv")  
  
  
# In[141]:  
  
  
import ast  
  
  
# In[142]:  
  
  
df.generations = df.generations.apply(ast.literal\_eval)  
  
  
# In[114]:  
  
  
from scipy.stats import f\_oneway  
  
  
# In[268]:  
  
  
print(f\_oneway(df.loc[df.method == "basic\_population"].total\_fitness\_invocations,  
 df.loc[df.method == "ltee\_population"].total\_fitness\_invocations,  
 df.loc[df.method == "ltee\_subpopulations"].total\_fitness\_invocations))  
  
  
# In[267]:  
  
  
print(f\_oneway(df.loc[df.method == "basic\_population"].unique\_individuals,  
 df.loc[df.method == "ltee\_population"].unique\_individuals,  
 df.loc[df.method == "ltee\_subpopulations"].unique\_individuals))  
  
  
# In[269]:  
  
  
from statsmodels.stats.multicomp import MultiComparison  
print(MultiComparison(df.unique\_individuals.to\_numpy(), df.method.to\_numpy()).tukeyhsd())

# References

Good, Benjamin H., Michael J. McDonald, Jeffrey E. Barrick, Richard E. Lenski, and Michael M. Desai. 2017. “The Dynamics of Molecular Evolution over 60,000 Generations.” *Nature* 551 (7678): 45–50. <https://doi.org/10.1038/nature24287>.

Holland, John H. 1992. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. 1st MIT Press ed. Complex Adaptive Systems. Cambridge, Mass: MIT Press.

Lenski, R. E., and M. Travisano. 1994. “Dynamics of Adaptation and Diversification: A 10,000-Generation Experiment with Bacterial Populations.” *Proceedings of the National Academy of Sciences* 91 (15): 6808–14. <https://doi.org/10.1073/pnas.91.15.6808>.

Maddamsetti, Rohan, and Richard E Lenski. n.d. “Analysis of Bacterial Genomes from an Evolution Experiment with Horizontal Gene Transfer Shows That Recombination Can Sometimes Overwhelm Selection,” 30.

Skiena, Steven S. 2008. *The Algorithm Design Manual*. 2nd ed. London: Springer.

Thierens, D. 2002. “Adaptive Mutation Rate Control Schemes in Genetic Algorithms.” In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC’02 (Cat. No.02TH8600)*, 1:980–85. Honolulu, HI, USA: IEEE. <https://doi.org/10.1109/CEC.2002.1007058>.

Tukey, John W. 1949. “Comparing Individual Means in the Analysis of Variance.” *Biometrics* 5 (2): 99. <https://doi.org/10.2307/3001913>.

Wiser, M. J., N. Ribeck, and R. E. Lenski. 2013. “Long-Term Dynamics of Adaptation in Asexual Populations.” *Science* 342 (6164): 1364–7. <https://doi.org/10.1126/science.1243357>.