An Introduction to Genetic Algorithms

Jenna Carr

May 16, 2014

Abstract

Genetic algorithms are a type of optimization algorithm, meaning they are used to find the maximum or minimum of a function. In this paper we introduce, illustrate, and discuss genetic algorithms for beginning users. We show what components make up genetic algorithms and how to write them. Using MATLAB, we program several examples, including a genetic algorithm that solves the classic Traveling Salesman Problem. We also discuss the history of genetic algorithms, current applications, and future developments.

Genetic algorithms are a type of optimization algorithm, meaning they are used to find the optimal solution(s) to a given computational problem that maximizes or minimizes a particular function. Genetic algorithms represent one branch of the field of study called evolutionary computation [4], in that they imitate the biological processes of reproduction and natural selection to solve for the 'fittest' solutions [1]. Like in evolution, many of a genetic algorithm's processes are random, however this optimization technique allows one to set the level of randomization and the level of control [1]. These algorithms are far more powerful and efficient than random search and exhaustive search algorithms [4], yet require no extra information about the given problem. This feature allows them to find solutions to problems that other optimization methods cannot handle due to a lack of continuity, derivatives, linearity, or other features.

Section 1 explains what makes up a genetic algorithm and how they operate. Section 2 walks through three simple examples. Section 3 gives the history of how genetic algorithms developed. Section 4 presents two classic optimization problems that were almost impossible to solve before the advent of genetic algorithms. Section 5 discusses how these algorithms are used today.

1 Components, Structure, & Terminology

Since genetic algorithms are designed to simulate a biological process, much of the relevant terminology is borrowed from biology. However, the entities that this terminology refers to in genetic algorithms are much simpler than their biological counterparts [8]. The basic components common to almost all genetic algorithms are:

- a fitness function for optimization
- a population of chromosomes
- selection of which chromosomes will reproduce
- crossover to produce next generation of chromosomes
- random mutation of chromosomes in new generation

The *fitness function* is the function that the algorithm is trying to optimize [8]. The word "fitness" is taken from evolutionary theory. It is used here because the fitness function tests and quantifies how 'fit' each potential solution is. The fitness function is one of the most pivotal parts of the algorithm, so it is discussed in more detail at the end of this section. The term *chromosome* refers to a numerical value or values that represent a candidate solution to the problem that the genetic algorithm is trying to solve [8]. Each candidate solution is encoded as an array of parameter values, a process that is also found in other optimization algorithms [2]. If a problem has N_{par} dimensions, then typically each chromosome is encoded

as an N_{par}-element array

$$\text{chromosome} = [p_1, p_2, ..., p_{N_{par}}]$$

where each p_i is a particular value of the i^{th} parameter [2]. It is up to the creator of the genetic algorithm to devise how to translate the sample space of candidate solutions into chromosomes. One approach is to convert each parameter value into a bit string (sequence of 1's and 0's), then concatenate the parameters end-to-end like genes in a DNA strand to create the chromosomes [8]. Historically, chromosomes were typically encoded this way, and it remains a suitable method for discrete solution spaces. Modern computers allow chromosomes to include permutations, real numbers, and many other objects; but for now we will focus on binary chromosomes.

A genetic algorithm begins with a randomly chosen assortment of chromosomes, which serves as the first generation (initial population). Then each chromosome in the population is evaluated by the fitness function to test how well it solves the problem at hand.

Now the *selection operator* chooses some of the chromosomes for reproduction based on a probability distribution defined by the user. The fitter a chromosome is, the more likely it is to be selected. For example, if f is a non-negative fitness function, then the probability that chromosome C_{53} is chosen to reproduce might be

$$P(C_{53}) = \left| \frac{f(C_{53})}{\sum_{i=1}^{N_{pop}} f(C_i)} \right| .$$

Note that the selection operator chooses chromosomes with replacement, so the same chromosome can be chosen more than once. The *crossover operator* resembles the biological crossing over and recombination of chromosomes in cell meiosis. This operator swaps a subsequence of two of the chosen chromosomes to create two offspring. For example, if the parent chromosomes

$$[11010111001000] \text{ and } [01011101010010]$$

are crossed over after the fourth bit, then

$$[01010111001000] \text{ and } [11011101010010]$$

will be their offspring. The *mutation operator* randomly flips individual bits in the new chromosomes (turning a 0 into a 1 and vice versa). Typically mutation happens with a very low probability, such as 0.001. Some algorithms implement the mutation operator before the selection and crossover operators; this is a matter of preference. At first glance, the mutation operator may seem unnecessary. In fact, it plays an important role, even if it is secondary to those of selection and crossover [1]. Selection and crossover maintain the genetic information of fitter chromosomes, but these chromosomes are only fitter relative to the current generation. This can cause the algorithm to converge too quickly and lose "potentially useful genetic material (1's or 0's at particular locations)" [1]. In other words, the algorithm can get stuck at a local optimum before finding the global optimum [3]. The mutation operator helps protect against this problem by maintaining diversity in the population, but it can also make the algorithm converge more slowly.

Typically the selection, crossover, and mutation process continues until the number of offspring is the same as the initial population, so that the second generation is composed entirely of new offspring and the first generation is completely replaced. We will see this method in Examples 2.1 and 2.2. However, some algorithms let highly-fit members of the first generation survive into the second generation. We will see this method in Example 2.3 and Section 4.

Now the second generation is tested by the fitness function, and the cycle repeats. It is a common practice to record the chromosome with the highest fitness (along with its fitness value) from each generation, or the "best-so-far" chromosome [5].

Genetic algorithms are iterated until the fitness value of the "best-so-far" chromosome stabilizes and does not change for many generations. This means the algorithm has converged

to a solution(s). The whole process of iterations is called a *run*. At the end of each run there is usually at least one chromosome that is a highly fit solution to the original problem. Depending on how the algorithm is written, this could be the most fit of all the "best-so-far" chromosomes, or the most fit of the final generation.

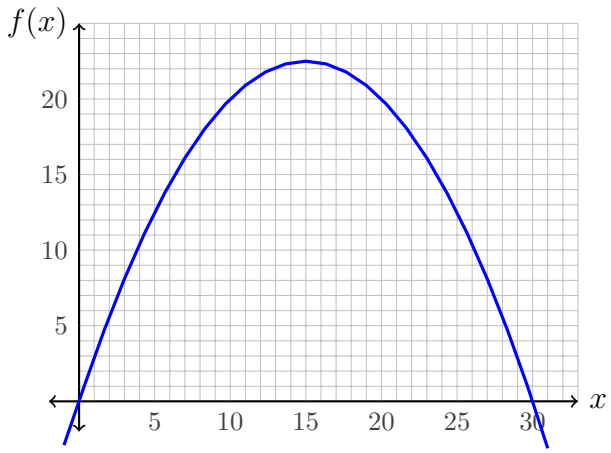
The "performance" of a genetic algorithm depends highly on the method used to encode candidate solutions into chromosomes and "the particular criterion for success," or what the fitness function is actually measuring [7]. Other important details are the probability of crossover, the probability of mutation, the size of the population, and the number of iterations. These values can be adjusted after assessing the algorithm's performance on a few trial runs.

Genetic algorithms are used in a variety of applications. Some prominent examples are automatic programming and machine learning. They are also well suited to modeling phenomena in economics, ecology, the human immune system, population genetics, and social systems.

1.1 A Note About Fitness Functions

Continuing the analogy of natural selection in biological evolution, the fitness function is like the habitat to which organisms (the candidate solutions) adapt. It is the only step in the algorithm that determines how the chromosomes will change over time, and can mean the difference between finding the optimal solution and finding no solutions at all. Kinnear, the editor of *Advances in Genetic Programming*, explains that the "fitness function is the only chance that you have to communicate your intentions to the powerful process that genetic programming represents. Make sure that it communicates precisely what you desire" [4]. Simply put, "you simply cannot take too much care in crafting" it [4]. Kinnear stresses that the population's evolution will "ruthlessly exploit" all "boundary conditions" and subtle defects in the fitness function [4], and that the only way to detect this is to just run the algorithm and examine the chromosomes that result.

Figure 1: Graph of $f(x) = \frac{-x^2}{10} + 3x$



The fitness function must be more sensitive than just detecting what is a 'good' chromosome versus a 'bad' chromosome: it needs to accurately score the chromosomes based on a *range* of fitness values, so that a somewhat complete solution can be distinguished from a more complete solution [4]. Kinnear calls this awarding "partial credit" [4]. It is important to consider which partial solutions should be favored over other partial solutions because that will determine the direction in which the whole population moves [4].

2 Preliminary Examples

This section will walk through a few simple examples of genetic algorithms in action. They are presented in order of increasing complexity and thus decreasing generality.

2.1 Example: Maximizing a Function of One Variable

This example adapts the method of an example presented in Goldberg's book [1].

Consider the problem of maximizing the function

$$f(x) = \frac{-x^2}{10} + 3x$$

where x is allowed to vary between 0 and 31. This function is displayed in Figure 1. To solve this using a genetic algorithm, we must encode the possible values of x as chromosomes. For this example, we will encode x as a binary integer of length 5. Thus the chromosomes for our genetic algorithm will be sequences of 0's and 1's with a length of 5 bits, and have a range from 0 (00000) to 31 (11111).

To begin the algorithm, we select an initial population of 10 chromosomes at random. We can achieve this by tossing a fair coin 5 times for each chromosome, letting heads signify 1 and tails signify 0. The resulting initial population of chromosomes is shown in Table 1. Next we take the x-value that each chromosome represents and test its fitness with the fitness function. The resulting fitness values are recorded in the third column of Table 1.

Table 1: Initial Population

Chromosome Number	Initial Population	x Value	Fitness Value $f(x)$	Selection Probability
1	0 1 0 1 1	11	20.9	0.1416
2	1 1 0 1 0	26	10.4	0.0705
3	0 0 0 1 0	2	5.6	0.0379
4	0 1 1 1 0	14	22.4	0.1518
5	0 1 1 0 0	12	21.6	0.1463
6	1 1 1 1 0	30	0	0
7	1 0 1 1 0	22	17.6	0.1192
8	0 1 0 0 1	9	18.9	0.1280
9	0 0 0 1 1	3	8.1	0.0549
10	1 0 0 0 1	17	22.1	0.1497
		Sum	147.6	
		Average	14.76	
		Max	22.4	

We select the chromosomes that will reproduce based on their fitness values, using the following probability:

$$P(\text{chromosome } i \text{ reproduces}) = \frac{f(x_i)}{\sum_{k=1}^{10} f(x_k)}$$

Goldberg likens this process to spinning a weighted roulette wheel [1]. Since our population has 10 chromosomes and each 'mating' produces 2 offspring, we need 5 matings to produce a new generation of 10 chromosomes. The selected chromosomes are displayed in Table 2. To create their offspring, a crossover point is chosen at random, which is shown in the table as a vertical line. Note that it is possible that crossover does not occur, in which case the offspring are exact copies of their parents.

Table 2: Reproduction & Second Generation

Chromosome Number	Mating Pairs	New Population	x Value	Fitness Value $f(x)$
5	0 1|1 0 0	0 1 0 1 0	10	20
2	1 1|0 1 0	1 1 1 0 0	28	5.6
4	0 1 1 1|0	0 1 1 1 1	15	22.5
8	0 1 0 0|1	0 1 0 0 0	8	17.6
9	0 0 0 1|1	0 **1** 0 1 0	10	20
2	1 1 0 1|0	1 1 0 1 1	27	8.1
7	1 0 1 1 0	1 0 1 1 0	22	17.6
4	0 1 1 1 0	0 1 1 1 0	14	22.4
10	1 0 0|0 1	1 0 0 0 1	17	22.1
8	0 1 0|0 1	0 1 0 0 1	9	18.9
			Sum	174.8
			Average	17.48
			Max	22.5

Lastly, each bit of the new chromosomes mutates with a low probability. For this example, we let the probability of mutation be 0.001. With 50 total transferred bit positions, we expect $50 \cdot 0.001 = 0.05$ bits to mutate. Thus it is likely that no bits mutate in the second generation. For the sake of illustration, we have mutated one bit in the new population, which is shown in bold in Table 2.

After selection, crossover, and mutation are complete, the new population is tested with the fitness function. The fitness values of this second generation are listed in Table 2. Although drawing conclusions from a single trial of an algorithm based on probability is, "at best, a risky business," this example illustrates how genetic algorithms 'evolve' toward fitter candidate solutions [1]. Comparing Table 2 to Table 1, we see that both the maximum

fitness and average fitness of the population have increased after only one generation.

2.2 Example: Number of 1's in a String

This example adapts Haupt's code for a binary genetic algorithm [3] to the first computer exercise from chapter 1 of Mitchell's textbook [7].

In this example we will program a complete genetic algorithm using MATLAB to maximize the number of 1's in a bit string of length 20. Thus our chromosomes will be binary strings of length 20, and the optimal chromosome that we are searching for is

$$[11111111111111111111] .$$

The complete MATLAB code for this algorithm is listed in Appendix A. Since the only possibilities for each bit of a chromosome are 1 or 0, the number of 1's in a chromosome is equivalent to the sum of all the bits. Therefore we define the fitness function as the sum of the bits in each chromosome. Next we define the size of the population to be 100 chromosomes, the mutation rate to be 0.001, and the number of iterations (generations) to be 200. For the sake of simplicity, we let the probability of crossover be 1 and the location of crossover be always the same.

Figures 2, 3, and 4 show the results of three runs. The best solution found by both runs 1 and 3 was $[11111111111111111111]$. The best solution found by run 2 was $[11101111110111111111]$. We can see from run 2 (3) why it is important to run the entire algorithm more than once because this particular run did not find the optimal solution after 200 generations. Run 3 (4) might not have found the optimal solution if we had decided to iterate the algorithm less than 200 times. These varying results are due to the probabilistic nature of several steps in the genetic algorithm [5]. If we encountered runs like 2 and 3 (3, 4) in a real-world situation, it would be wise for us to increase the number of iterations to make sure that the algorithm was indeed converging to the optimal solution. In practical applications, one does not know

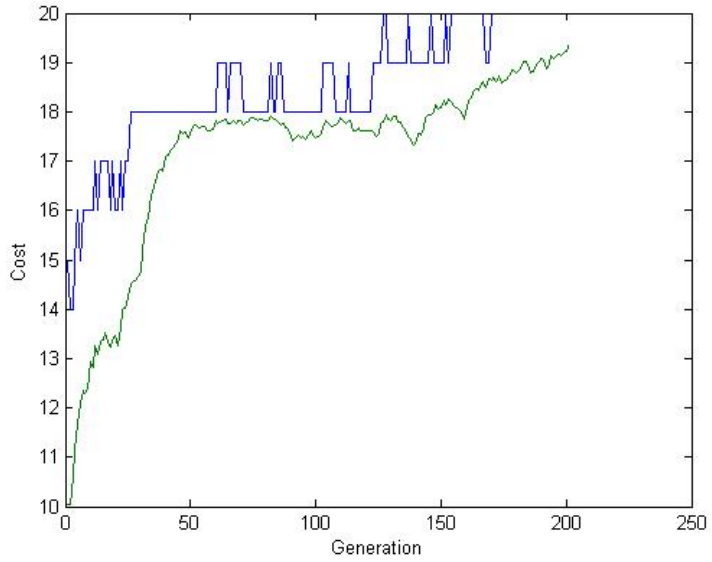


Figure 2: The first run of a genetic algorithm maximizing the number of 1's in string of 20 bits. The blue curve is highest fitness, and the green curve is average fitness. Best solution: [11111111111111111111] .

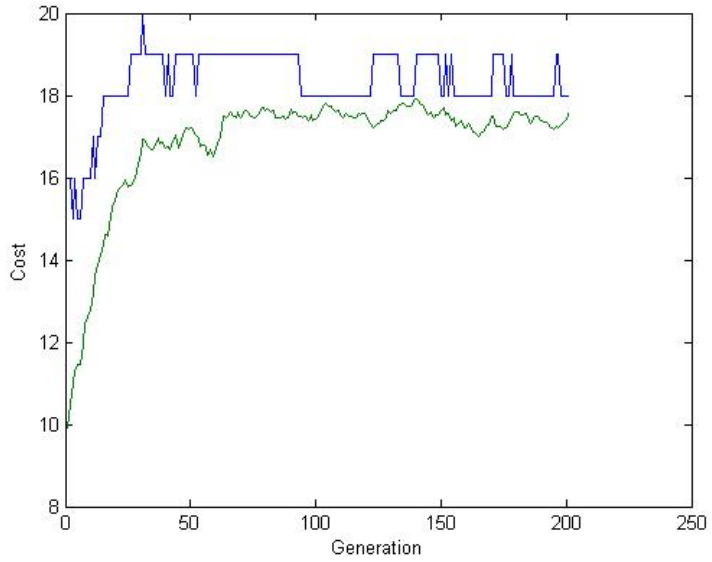


Figure 3: The second run of a genetic algorithm maximizing the number of 1's in string of 20 bits. The blue curve is highest fitness, and the green curve is average fitness. Best solution: [11101111110111111111] .

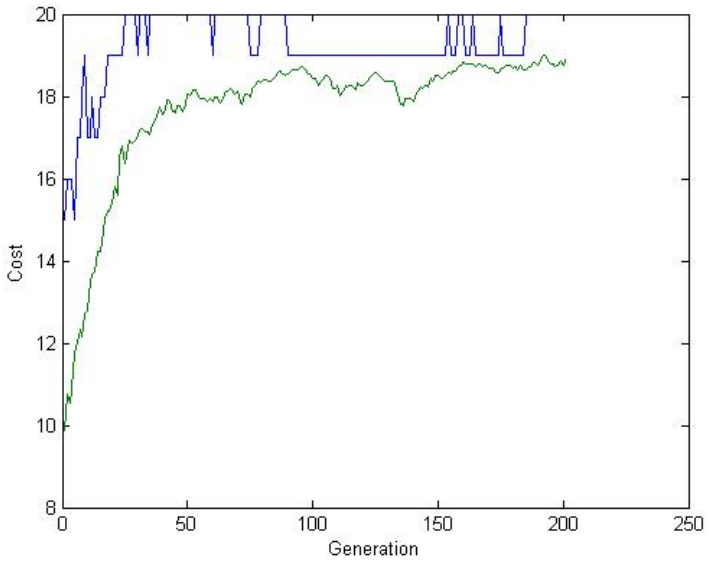


Figure 4: The third run of a genetic algorithm maximizing the number of 1's in string of 20 bits. The blue curve is highest fitness, and the green curve is average fitness. Best solution: [11111111111111111111] .

what an optimal solution will look like, so it is common practice to perform several runs.

2.3 Example: Continuous Genetic Algorithm

Here we present a new style of genetic algorithm, and hint at its applications in topography. This example adapts the code and example for a continuous genetic algorithm from Haupt's book [3].

You may have noticed that Example 2.1 only allows for integer solutions, which is acceptable in that case because the maximum of the function in question is an integer. But what if we need our algorithm to search through a continuum of values? In this case it makes more sense to make each chromosome an array of real numbers (floating-point numbers) as opposed to an array of just 0's and 1's [3]. This way, the precision of the solutions is only limited by the computer, instead of by the binary representation of the numbers. In fact, this continuous genetic algorithm is faster than the binary genetic algorithm because the chromosomes do not need to be decoded before testing their fitness with the fitness func-

tion [3]. Recall that if our problem has N_{par} dimensions, then we are going to encode each chromosome as an N_{par} element array

$$\text{chromosome} = [p_1, p_2, ..., p_{N_{par}}]$$

where each p_i is a particular value of the i^{th} parameter, only this time each p_i is a real number instead of a bit [2].

Suppose we are searching for the point of lowest elevation on a (hypothetical) topographical map where longitude ranges from 0 to 10, latitude ranges from 0 to 10, and the elevation of the landscape is given by the function

$$f(x, y) = x \sin(4x) + 1.1y \sin(2y)\,.$$

To solve this problem, we will program a continuous genetic algorithm using MATLAB. The complete code for this algorithm is listed in Appendix B.

We can see that $f(x, y)$ will be our fitness function because we are searching for a solution that minimizes elevation. Since f is a function of two variables, $N_{par} = 2$ and our chromosomes should be of the form

$$\text{chromosome} = [x, y]\,, \quad \text{for } 0 \leq x \leq 10 \text{ and } 0 \leq y \leq 10.$$

We define our population size to be 12 chromosomes, the mutation rate to be 0.2, and the number of iterations to be 100. Like in the binary genetic algorithm, the initial population is generated randomly, except this time the parameters can be any real number between 0 and 10 instead of just 1 or 0.

With this algorithm we demonstrate a slightly different approach for constructing subsequent generations than we used in the previous examples. Instead of replacing the entire population with new offspring, we keep the fitter half of the current population, and generate

the other half of the new generation through selection and crossover. Tables 3 and 4 show a possible initial population and the half of it that survives into the next generation. The selection operator only chooses parents from this fraction of the population that is kept.

Table 3: Example Initial Population

x	y	Fitness $f(x, y)$
6.7874	6.9483	13.5468
7.5774	3.1710	-6.5696
7.4313	9.5022	-5.7656
3.9223	0.3445	0.3149
6.5548	4.3874	8.7209
1.7119	3.8156	5.0089
7.0605	7.6552	3.4901
0.3183	7.9520	-1.3994
2.7692	1.8687	-3.9137
0.4617	4.8976	-1.5065
0.9713	4.4559	1.7482
8.2346	6.4631	10.7287

Table 4: Surviving Population after 50% Selection Rate

Rank	x	y	Fitness $f(x, y)$
1	7.5774	3.1710	-6.5696
2	7.4313	9.5022	-5.7656
3	2.7692	1.8687	-3.9137
4	0.4617	4.8976	-1.5065
5	0.3183	7.9520	-1.3994
6	3.9223	0.3445	0.3149

Note that the fitness function takes on both positive and negative values, so we cannot directly use the sum of the chromosomes' fitness values to determine the probabilities of who will be selected to reproduce (like we did in Example 2.1). Instead we use rank weighting, based on the rankings shown in Table 4. The probability that the chromosome in n^{th} place will be a parent is given by the equation

$$P_n = \frac{N_{\text{keep}} - n + 1}{\sum_{i=1}^{N_{\text{keep}}} i} = \frac{6 - n + 1}{1 + 2 + 3 + 4 + 5 + 6} = \frac{7 - n}{21} \ ,$$

so the probability that the chromosome in first place will be selected is

$$P(C_1) = \frac{6-1+1}{1+2+3+4+5+6} = \frac{6}{21} \ ,$$

and the probability that the chromosome in sixth place will be selected is

$$P(C_6) = \frac{6-6+1}{1+2+3+4+5+6} = \frac{1}{21} \ .$$

Recall that each mating produces 2 offspring, so we need 3 pairs of parent chromosomes to produce the right number of offspring to fill the rest of the next generation.

Since our chromosomes are no longer bit strings, we need to revise how the crossover and mutation operators work. There are several methods to achieve this; here we will use Haupt's method [3]. Once we have two parent chromosomes $m = [x_m, y_m]$ and $d = [x_d, y_d]$, we first randomly select one of the parameters to be the point of crossover. To illustrate, suppose x is the crossover point for a particular m and d. Then we introduce a random value between 0 and 1 represented by β, and the x-values in the offspring are

$$\begin{aligned} x_{\text{new1}} &= (1-\beta)x_m + \beta x_d \\ x_{\text{new2}} &= (1-\beta)x_d + \beta x_m \, . \end{aligned}$$

The remaining parameter (y in this case) is inherited directly from each parent, so the completed offspring are

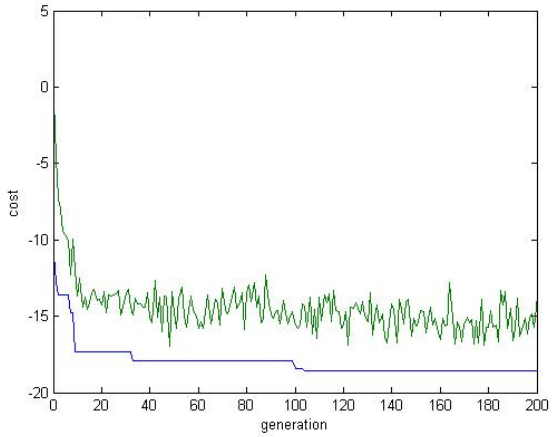
$$\begin{aligned} \text{offspring}_1 &= [x_{new1}, y_m] \\ \text{offspring}_2 &= [x_{new2}, y_d] \, . \end{aligned}$$

If we suppose that $\text{chromosome}_1 = [7.5774\,,\, 3.1710]$ and $\text{chromosome}_3 = [2.7692\,,\, 1.8687]$ are selected as a pair of parents. Then with crossover at x and a β-value of 0.3463, their offspring would be

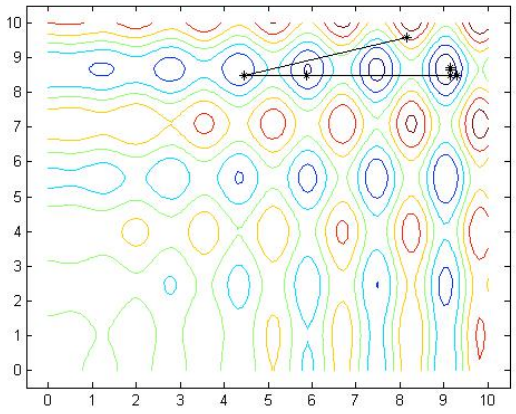
$$\text{offspring}_1 = [(1-0.3463)\cdot 7.5774 + 0.3463 \cdot 2.7692, 3.1710] = [5.9123, 3.1710]$$

$$\text{offspring}_2 = [(1 - 0.3463) \cdot 2.7692 + 0.3463 \cdot 7.5774, 1.8687] = [4.4343, 1.8687]\,.$$

Like the crossover operator, there are many different methods for adapting the mutation operator to a continuous genetic algorithm. For this example, when a parameter in a chromosome mutates, its value is replaced with a random number between 0 and 10.



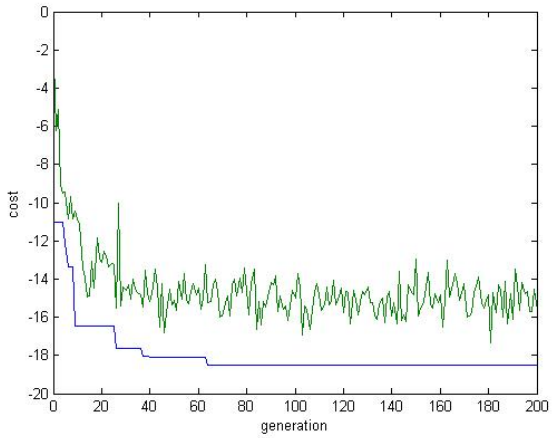
(a) The blue curve is lowest elevation, the green curve is average elevation.



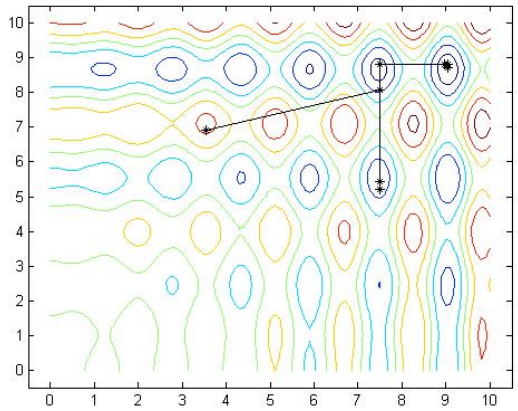
(b) Contour map showing the 'path' of the most fit chromosome in each generation to the next.

Figure 5: The first run of a continuous genetic algorithm finding the point of lowest elevation of the function $f(x, y) = x\sin(4x) + 1.1y\sin(2y)$ in the region $0 \leq x \leq 10$ and $0 \leq y \leq 10$. With elevation -18.5519, the best solution was $[9.0449, 8.6643]$.

Figures 5, 6, and 7 show the results of three runs. We can see in all three runs that the algorithm found the minimum long before reaching the 200^{th} generation. In this particular case, the excessive number of generations is not a problem because the algorithm is simple and its efficiency is not noticeably affected. Note that because the best members of a given generation survive into the subsequent generation, the best value of f always improves or stays the same. We also see that each run's final solution differs slightly from the others. This is not surprising because our parameters are continuous variables, and thus have an infinite number of possible values arbitrarily close to the actual minimum.

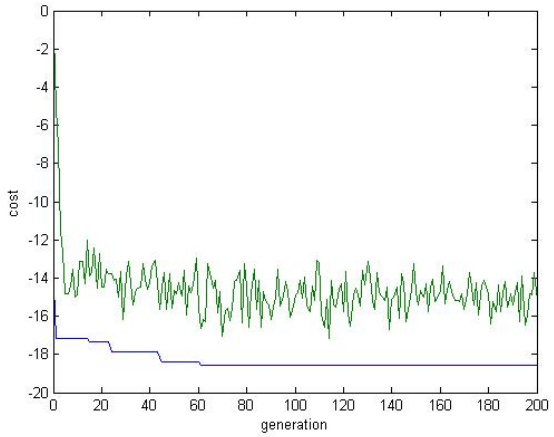


(a) The blue curve is lowest elevation, the green curve is average elevation.

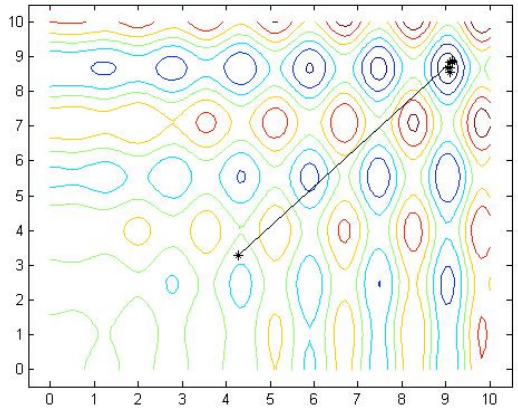


(b) Contour map showing the 'path' of the most fit chromosome in each generation to the next.

Figure 6: The second run of a continuous genetic algorithm finding the point of lowest elevation of the function $f(x, y) = x\sin(4x) + 1.1y\sin(2y)$ in the region $0 \leq x \leq 10$ and $0 \leq y \leq 10$. With elevation -18.5227, the best solution was $[9.0386, 8.709]$.



(a) The blue curve is lowest elevation, the green curve is average elevation.



(b) Contour map showing the 'path' of the most fit chromosome in each generation to the next.

Figure 7: The third run of a continuous genetic algorithm finding the point of lowest elevation of the function $f(x, y) = x\sin(4x) + 1.1y\sin(2y)$ in the region $0 \leq x \leq 10$ and $0 \leq y \leq 10$. With elevation -18.5455, the best solution was $[9.0327, 8.6865]$.

3 Historical Context

As far back as the 1950s, scientists have studied artificial intelligence and evolutionary computation, trying to write computer programs that could simulate processes that occur in the natural world. In 1953, Nils Barricelli was invited to Princeton to study artificial intelligence [9]. He used the recently invented digital computer to write software that mimicked natural reproduction and mutation. Barricelli's goal was not to solve optimization problems nor simulate biological evolution, but rather to create artificial life. He created the first genetic algorithm software, and his work was published in Italian in 1954 [9]. Barricelli's work was followed in 1957 by Alexander Fraser, a biologist from London. He had the idea of creating a computer model of evolution, since observing it directly would require millions of years. He was the first to use computer programming solely to study evolution [9]. Many biologists followed in his footsteps in the late 1950s and 1960s.

In her book, Mitchell states that John Holland invented genetic algorithms in the 1960s, or at least the particular version that is known today by that specific title [7]. Holland's version involved a simulation of Darwinian 'survival of the fittest,' as well as the processes of crossover, recombination, mutation, and inversion that occur in genetics [7]. This population-based method was a huge innovation. Previous genetic algorithms only used mutation as the driver of evolution [7]. Holland was a professor of psychology, computer science, and electrical engineering at the University of Michigan. He was driven by the pursuit to understand how "systems adapt to their surroundings" [9]. After years of collaboration with students and colleagues, he published his famous book *Adaptation in Natural and Artificial Systems* in 1975. In this book, Holland presented genetic algorithms as an "abstraction of biological evolution and gave a theoretical framework for adaptation" under genetic algorithms [7]. This book was the first to propose a theoretical foundation for computational evolution, and it remained the basis of all theoretical work on genetic algorithms until recently [7]. It became a classic because of its demonstration of the mathematics behind evolution [9]. In the same year one of Holland's doctoral students, Kenneth De Jong, presented the first

comprehensive study of the use of genetic algorithms to solve optimization problems as his doctoral dissertation. His work was so thorough that for many years, any papers on genetic algorithms that did not include his benchmark examples were considered "inadequate" [9].

Research on genetic algorithms rapidly increased in the 1970s and 1980s. This was partly due to advances in technology. Computer scientists also began to realize the limitations of conventional programming and traditional optimization methods for solving complex problems. Researchers found that genetic algorithms were a way to find solutions to problems that other methods could not solve. Genetic algorithms can simultaneously test many points from all over the solution space, optimize with either discrete or continuous parameters, provide several optimum parameters instead of a single solution, and work with many different kinds of data [2]. These advantages allow genetic algorithms to "produce stunning results when traditional optimization methods fail miserably" [2].

When we say "traditional" optimization methods, we are referring to three main types: calculus-based, exhaustive search, and random [1]. Calculus-based optimization methods come in two categories: direct and indirect. The direct method 'jumps onto' the objective function and follows the direction of the gradient towards a local maximum or minimum value. This is also known as the "hill-climbing" or "gradient ascent" method [1]. The indirect method takes the gradient of the objective function, sets it equal to zero, then solves the set of equations that results [1]. Although these calculus-based techniques have been studied extensively and improved, they still have insurmountable problems. First, they only search for local optima, which renders them useless if we do not know the neighborhood of the global optimum or if there are other local optima nearby (and we usually do not know). Second, these methods require the existence of derivatives, and this is virtually never the case in practical applications.

Exhaustive search algorithms perform exactly that—an exhaustive search. These algorithms require a "finite search space, or a discretized infinite search space" of possible values for the objective function [1]. Then they test every single value, one at a time, to find the

maximum or minimum. While this method is simple and thus "attractive," it is the least efficient of all optimization algorithms [1]. In practical problems, the search spaces are too vast to test every possibility "one at a time and still have a chance of using the [resulting] information to some practical end" [1].

Random search algorithms became increasingly popular as people realized the shortcomings of calculus-based and exhaustive search algorithms. This style of algorithm randomly chooses some representative sampling from the search space, and finds the optimal value in that sampling. While faster than an exhaustive search, this method "can be expected to do no better" than an exhaustive search [1]. Using this type of algorithm means that we leave it up to chance whether we will be somewhat near the optimal solution or miles away from it.

Genetic algorithms have many advantages over these traditional methods. Unlike calculus-based methods like hill-climbing, genetic algorithms progress from a population of candidate solutions instead of a single value [1]. This greatly reduces the likelihood of finding a local optimum instead of the global optimum. Genetic algorithms do not require extra information (like derivatives) that is unrelated to the values of the possible solutions themselves. The only mechanism that guides their search is the numerical fitness value of the candidate solutions, based on the creator's definition of fitness [5]. This allows them to function when the search space is noisy, nonlinear, and derivatives do not even exist. This also makes them applicable in many more situations than traditional algorithms, and they can be adjusted in each situation based on whether accuracy or efficiency is more important.

4 Practical Problems

In this section we present each of two classic optimization problems with a genetic algorithm. We will program these algorithms using MATLAB.

4.1 The Traveling Salesman Problem

This is perhaps the most famous, practical, and widely-studied combinatorial optimization problem (by combinatorial we mean it involves permutations—arranging a set number of items in different orders) [9]. It has applications in "robotics, circuit board drilling, welding, manufacturing, transportation, and many other areas" [9]. For instance, "a circuit board could have tens of thousands of holes, and a drill needs to be programmed to visit each of those holes while minimizing some cost function (time or energy, for example)" [9].

In this problem, we assume that there is a traveling salesman who needs to visit n cities via the shortest route possible. He visits every city exactly once, then returns to the city where he started. Therefore a candidate solution would be a list of all the cities in the order that he visits them [3]. We denote the cities as city 1, city 2, city 3, ..., city n; where city 1 is his starting point. Thus the chromosomes for the genetic algorithm will be different permutations of the integers 1 through n.

There are many variations of the traveling salesman problem, but for our purposes we make the following assumptions. We assume that the distance d from city c_i to city c_j is $d(c_i, c_j) = d(c_j, c_i)$ for all $i \in [1, n]$ and $j \in [1, n]$. We can see how in some real-world cases this would not be true: the highway in one direction may be slightly longer than the highway traveling in the opposite direction, or it might cost more to drive uphill than downhill (it is common for the cost of travel to be included in what we are calling "distance"). Those cases are called "asymmetric" traveling salesman problems [9]. Since the salesman ends in the city where he started, this is a "closed" traveling salesman problem. In the "open" variation, the salesman visits all cities exactly once, so he does not end the journey where he started [9].

4.1.1 Modifying Crossover and Mutation for Permutation Problems

Based on how we have encoded the candidate solutions into chromosomes, the operators of crossover and mutation as we defined them for the binary genetic algorithm at the beginning of this paper will not work because each city needs to be represented once and only once [3].

To see why this is the case, consider two parent chromosomes of length 5 and their offspring after normal crossover:

Parents	Normal Crossover	Offspring (faulty)
3 5 1 2 4	3 5 | 1 2 4	3 5 5 3 2
1 4 5 3 2	1 4 | 5 3 2	1 4 1 2 4

These offspring are not valid permutations of the integers from 1 to 5 since they are missing some numbers and repeat others. To overcome this obstacle, we will be using a technique called "cycle" crossover in our genetic algorithm [3]. There are several other ways to modify crossover to accommodate permutations, which are discussed in Goldberg [1] and Haupt [3]. Cycle crossover is illustrated with chromosomes of length 6 in Table 5. First we

Table 5: Example of Cycle Crossover

Parents	Offspring (step 1)	Offspring (step 2)	Offspring (step 3)	Offspring (step 4)
4 1 5 | 3 | 2 6	4 1 5 **2** | 2 | 6	4 | 1 | 5 **2 1** 6	| 4 | **4** 5 **2 1** 6	**3 4** 5 **2 1** 6
3 4 6 | 2 | 1 5	3 4 6 **3** | 1 | 5	3 | 4 | 6 **3 2** 5	| 3 | **1** 6 **3 2** 5	**4 1** 6 **3 2** 5

choose a random location in the length of the chromosome. The two parent chromosomes exchange integers at this location (marked with bars in Table 5) to create the offspring. Unless the swapped integers are the same value, each offspring has a duplicate integer. Then we switch the duplicate integer in the first offspring with whatever is in the same location in the second offspring. This means that now there is a duplicate of a different integer, so the process repeats until there are no duplicates in the first offspring (or second offspring) [3]. Now each offspring has exactly one of every integer, thus both are valid permutations.

The modified mutation operator randomly chooses two integers in a chromosome from the new generation and swaps them. This operator still typically happens with a low probability.

4.1.2 Using a Genetic Algorithm

In this section we create the genetic algorithm to solve the traveling salesman problem. To do so, we adapt and improve upon Haupt's code for a permutation genetic algorithm and for the fitness function of a traveling salesman problem [3]. The complete MATLAB code for this algorithm is listed in Appendix C.

Let there be 20 cities. To represent them, we randomly place 20 points on the xy-plane in the 1×1 square between $(0, 0)$ and $(1, 1)$. Earlier, we defined our chromosomes to be permutations of the integers 1 through 20.

Suppose that the n cities are listed in the order $c_1 \to c_2 \to \cdots \to c_n$ in a given chromosome. Since we are trying to minimize the total distance that the salesman travels, the fitness function will be the total distance D:

$$D = \sum_{k=1}^{n} d(c_k, c_{k+1})$$

where city $n + 1 =$ city 1 (the starting city) [3]. If we let (x_i, y_i) denote the coordinates of city c_i, then the fitness function becomes

$$D = \sum_{k=1}^{n} \sqrt{(x_{k+1} - x_k)^2 + (y_{k+1} - y_k)^2} \ .$$

Next we define the population size to be 20. With this algorithm we use the same approach for constructing subsequent generations that we used in Example 2.3. Rather than replace the entire population with new offspring, we keep the fitter half of the current population, and generate the other half of the new generation through selection and crossover. Recall that the selection operator only chooses from the fraction of the population that is kept. We modified the crossover and mutation operators above (4.1.1). In this algorithm, the mutation operator acts on each member of the new generation with probability 0.05.

The point where this algorithm deviates the most from Haupt's original version is in

calculating the probability distribution for the selection operator, shown in Listing 4.1.

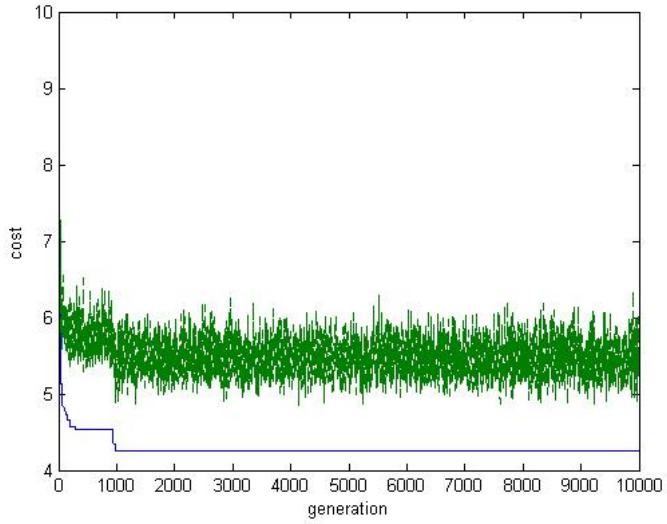
```
for ii=2:keep
    odds=[odds ii*ones(1,ii)];
end
Nodds=length(odds);
odds=keep-odds+1;
```

Listing 4.1: Determining probabilities for the selection operator based on the fitness rankings of the top 10 chromosomes.

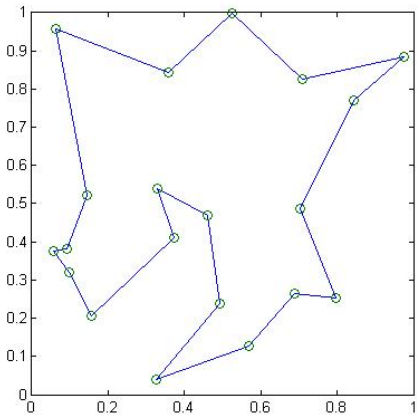
What we have done here to modify Haupt's version is add the last line, which effectively reverses the probability distribution of the pool of chromosomes that are chosen from for reproduction. Before this point in the code, the current generation of chromosomes is sorted from highest fitness to lowest fitness (from 1 to 20), and only the fitter half (10 chromosomes) is kept. The variable `odds` in Listing 4.1 is a vector containing the integers from 1 to 10, with ten instances of the integer 1, nine instances of the integer 2, down to one instance of the integer 10. A chromosome is then chosen to be a parent by picking an integer at random from the vector `odds`. Hence we see that the probability of the fittest chromosome being chosen is $\frac{10}{55} = \frac{2}{11}$, and the probability of the least fit chromosome being chosen is $\frac{1}{55}$. Without the new line of code that we added, `odds` would have ten 10's, nine 9's, etc. This would have given the least fit chromosome the highest probability of becoming a parent, and vice versa.

Lastly, we determine the number of iterations to be 10,000.

Figures 8, 9, 10, 11, and 12 show five of several runs. Since we do not know a priori what the optimal solution is, it is important to run the entire algorithm multiple times until the best result reappears a satisfactory number of times. We can see from Figure 9 and Figure 11 that we could have been fooled into thinking that 4.1816 is the shortest distance possible if we had not run the algorithm enough times for the result 4.1211 to appear. Luckily, 4.1211 appeared after just three runs (Figure 10). Several more runs confirmed that there is no

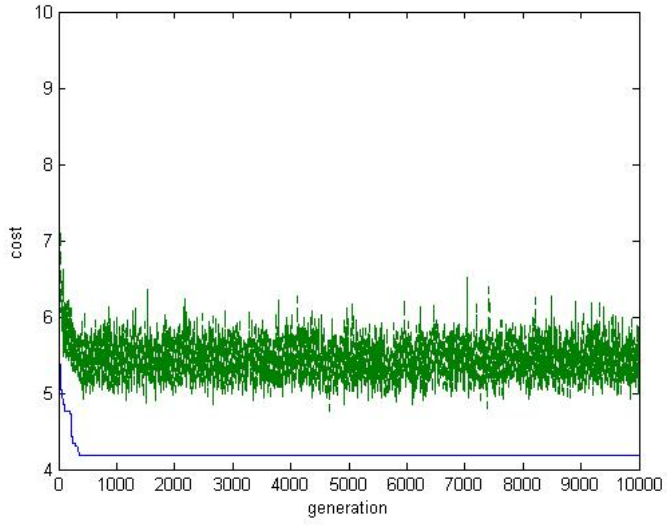


(a) The blue curve is shortest distance, the green curve is average distance.

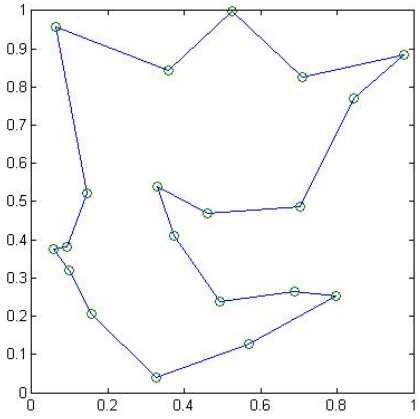


(b) Graph of the route.

Figure 8: The first run of a genetic algorithm minimizing the distance to travel in a closed loop to 20 cities. With distance 4.2547, the best solution was $[13, 15, 4, 18, 11, 17, 10, 14, 1, 3, 19, 12, 5, 20, 8, 2, 9, 7, 16, 6]$.

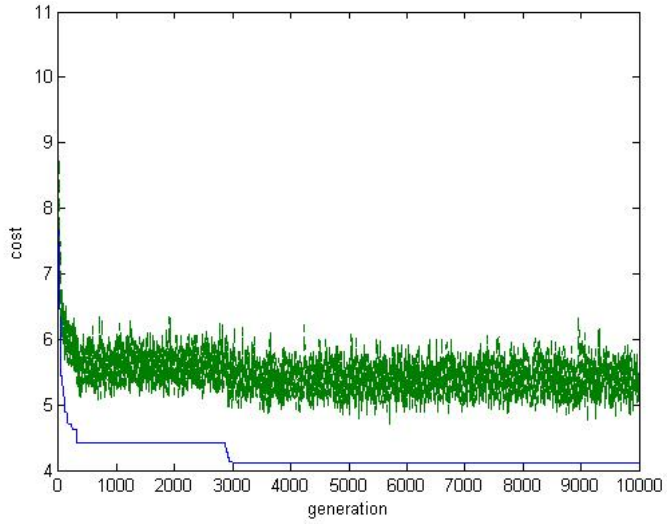


(a) The blue curve is shortest distance, the green curve is average distance.

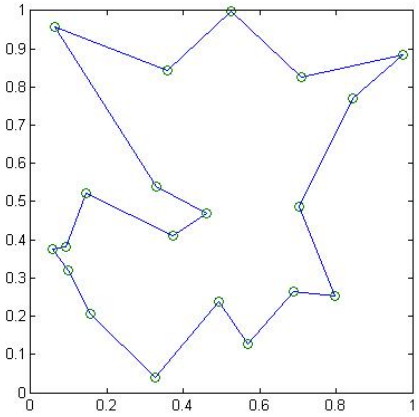


(b) Graph of the route.

Figure 9: The second run of a genetic algorithm minimizing the distance to travel in a closed loop to 20 cities. With distance 4.1816, the best solution was $[4, 18, 11, 17, 10, 14, 1, 3, 19, 2, 9, 16, 7, 8, 12, 5, 20, 6, 13, 15]$.

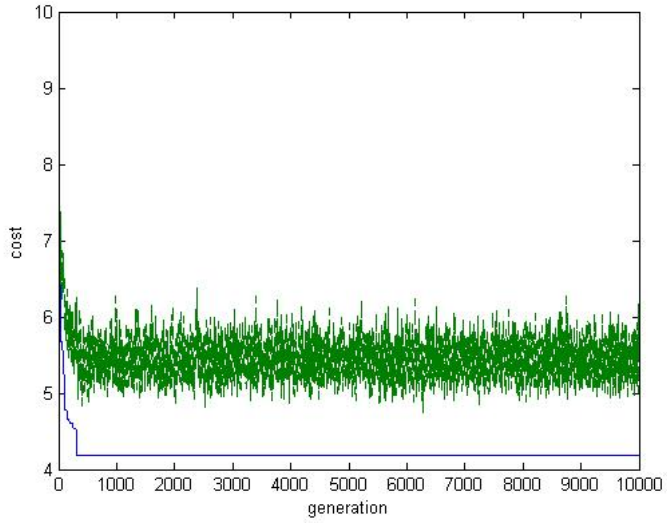


(a) The blue curve is shortest distance, the green curve is average distance.

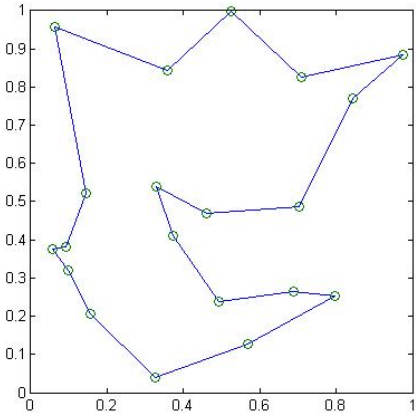


(b) Graph of the route.

Figure 10: The third run of a genetic algorithm minimizing the distance to travel in a closed loop to 20 cities. With distance 4.1211, the best solution was $[7, 9, 8, 2, 19, 3, 1, 14, 10, 12, 20, 5, 17, 11, 18, 4, 15, 13, 6, 16]$.

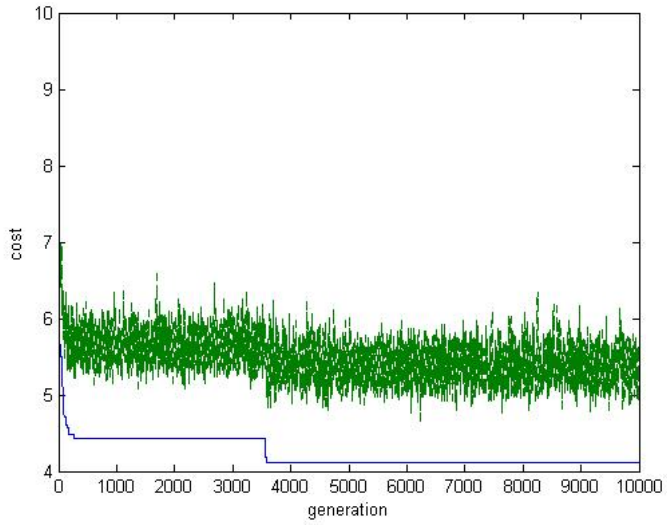


(a) The blue curve is shortest distance, the green curve is average distance.

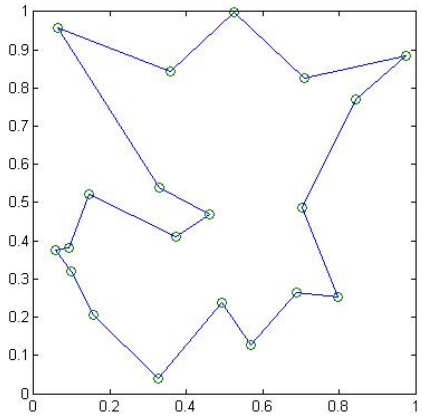


(b) Graph of the route.

Figure 11: The fourth run of a genetic algorithm minimizing the distance to travel in a closed loop to 20 cities. With distance 4.1816, the best solution was $[10, 14, 1, 3, 19, 2, 9, 16, 7, 8, 12, 5, 20, 6, 13, 15, 4, 18, 11, 17]$.



(a) The blue curve is shortest distance, the green curve is average distance.



(b) Graph of the route.

Figure 12: The eleventh run of a genetic algorithm minimizing the distance to travel in a closed loop to 20 cities. With distance 4.1211, the best solution was $[20, 12, 10, 14, 1, 3, 19, 2, 8, 9, 7, 16, 6, 13, 15, 4, 18, 11, 17, 5]$.

distance shorter than 4.1211, so the optimal solution is

$$[7, 9, 8, 2, 19, 3, 1, 14, 10, 12, 20, 5, 17, 11, 18, 4, 15, 13, 6, 16]\,.$$

Run 11 (Figure 12) was one of the runs that confirmed this. Note that the optimal solution that run 11 found is equivalent to that of run 3 even though the chromosome is a different permutation. This chromosome starts in a different city, but this does not affect the distance since the path of the salesman is a closed loop. This chromosome also lists the cities in the opposite order from that of run 3, but recall that

$$d(c_i, c_j) = d(c_j, c_i) \quad \text{for cities } c_i, c_j$$

so this does not affect the distance either.

4.1.3 Using a Traditional Algorithm

Why was the traveling salesman problem so difficult before genetic algorithms were invented? Let us consider using an exhaustive search algorithm to solve it. Note that since the salesman travels in a closed loop and the direction of travel between cities does not matter (free circular permutation), the total number of possible solutions is

$$S = \frac{(n-1)!}{2} = \frac{19!}{2} = 60,822,550,204,416,000\,.$$

This number is far too large for testing every single candidate even with smaller values of n than 20. The amount of time it would take for an exhaustive search algorithm to find the best solution completely nullifies the usefulness of whatever answer it finds.

4.2 The Knapsack Problem

Here we present another classic problem, which is used in cryptography, determining the best way to harvest raw materials, and many other applications.

This time we provide only a possible method for translating the problem into a fitness function and chromosomes, and a new way to distribute the probabilities for the selection operator. We invite the reader to fill in the rest and create a complete algorithm to solve it. This example is adapted from Steeb's workbook [10].

Suppose you are a hiker planning a backpacking trip, and you can comfortably carry no more than 20kg. Once you have picked out all of the supplies you wish to take along, you find that together they exceed 20kg. Then you assign a value to each of the 12 items, shown in Table 6, in order to help you choose which to take. Which items should you take with you to maximize the value of what you carry, without exceeding 20kg?

One way to translate the items into chromosomes is to create bit strings of length 12, where each bit position represents a particular item. Let 1 indicate that you include the particular item, and 0 indicate that you exclude it.

Table 6: Value & Weight of Camping Supplies

Item	Value	Weight
bug repellent	12	2
camp stove	5	4
canteen (full)	10	7
clothes	11	5
dried food	50	3
first-aid kit	15	3
flashlight	6	2
novel	4	2
rain gear	5	2
sleeping bag	25	3
tent	20	11
water filter	30	1

Based on this, your fitness function must know the fixed weights and values associated with each bit position, since the chromosomes do not carry this information. Your fitness function must also be able to add up two sums simultaneously: a sum of the items' values to determine the fitness, and a sum of the items' weights to satisfy the weight limit. [10]. While the fitness function is summing up the values and weights, one way to make sure that the chromosomes stay inside the weight limit is to immediately stop summing and assign a fitness of 0 or -1 if a chromosome's weight exceeds 20kg.

Thus far we have discussed two methods for creating the probability distribution of the selection operator: using fractions of the total fitness (in Section 1 and Example 2.1) and rank weighting (in Example 2.3 and Section 4.1.2). Another strategy utilizes the softmax function to create a distribution similar to our first method, except it can be used when some (or all) of the fitness values are negative. Instead of dividing a chromosome's fitness value by the sum of the whole population's fitness values, the exponential of a chromosome's fitness value is divided by the sum of the exponentials of the population's fitness values. To illustrate, the probability that a chromosome C_{53} is chosen to reproduce would be

$$P(C_{53}) = \left| \frac{e^{f(C_{53})}}{\sum_{i=1}^{N_{pop}} e^{f(C_i)}} \right| .$$

5 Discussion & Conclusion

In the present day, genetic algorithms are used by large companies to optimize schedules and design products that range from large aircraft to tiny computer chips to medicines [6]. The Los Alamos National Laboratory has used them to analyze the data from satellites [6]. They were used to create realistic special effects for films like *Troy* and *The Lord of the Rings* [6]. There are also many applications for genetic algorithms in the financial sector, such as identifying fraud and predicting market fluctuations. They are even being used in the development new forms of art and music [3].

Parallel computing has made genetic algorithms even more appealing because it can alleviate their main drawback: the lack of speed. This allows for more iterations and more total runs, which increases both the chances of finding better solutions and the certainty that the solutions found are the optimal ones.

Computer scientists are working on devising new ways to combine genetic algorithms with other optimization algorithms as well as other branches of evolutionary computation, such as neural networks [3]. One basic example combines a genetic algorithm with the calculus-based hill-climbing method. Each of the fittest chromosomes that the genetic algorithm finds is then used as a starting point for the hill-climbing algorithm, which follows the gradient of the fitness function from there to a local optimum [10]. A prominent and relatively new field is genetic programming, which was invented in 1989 by John Koza [4]. This field expands on the framework of genetic algorithms to allow candidate solutions to be nonlinear, of varying size, and even executable computer programs [4].

Remarkably, what remains to be discovered is the theory behind how genetic algorithms work. In his seminal book *Adaptation in Natural and Artificial Systems*, John Holland presented the "Building Block Hypothesis," which states that genetic algorithms work by "discovering, emphasizing, and recombining good building blocks" [8]. A building block was defined as a string of bit values (since this hypothesis was proposed when chromosomes were always defined as bit strings) that was shorter than the length of a chromosome, that

bestows higher fitness to all the chromosomes in which it is present [8]. This remained the prominent theory until the 1990s, when published papers began to emerge that questioned its validity. What is strange is that while authors have now thoroughly explained that the Building Block Hypothesis assumes too much and is false, no one has presented a feasible alternative hypothesis. Perhaps it is because people have seen that genetic algorithms work, and that is enough explanation for them. Near the end of his book, Goldberg declares, "Nature is concerned with that which works. Nature propagates that which survives. She has little time for erudite contemplation, and we have joined her in her expedient pursuit of betterment" [1].

References

[1] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading: Addison-Wesley.

[2] Haupt, R. L., & Haupt, S. E. (1998). *Practical Genetic Algorithms*. New York: Wiley-Interscience.

[3] Haupt, R. L., & Haupt, S. E. (2004). *Practical Genetic Algorithms* (2nd ed.). Hoboken: Wiley.

[4] Kinnear, K. E. (1994). A Perspective on the Work in this Book. In K. E. Kinnear (Ed.), *Advances in Genetic Programming* (pp. 3-17). Cambridge: MIT Press.

[5] Koza, J. R. (1994). Introduction to Genetic Programming. In K. E. Kinnear (Ed.), *Advances in Genetic Programming* (pp. 21-41). Cambridge: MIT Press.

[6] Mitchell, M. (2009). *Complexity: A Guided Tour*. New York: Oxford University Press.

[7] Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. Cambridge: MIT Press.

[8] Mitchell, M. (1995). Genetic Algorithms: An Overview. *Complexity*, 1(1), 31-39.

[9] Simon, D. (2013). *Evolutionary Optimization Algorithms: Biologically-Inspired and Population-Based Approaches to Computer Intelligence.* Hoboken: Wiley.

[10] Steeb, W. (2001). *The Nonlinear Workbook : Chaos, Fractals, Cellular Automata, Neural Networks, Genetic Algorithms, Fuzzy Logic: with C++, Java, SymbolicC++ and Reduce Programs.* Singapore: World Scientific.

Appendices

A MATLAB Code for Example 2.2

```
%  Problem:  Define the population as binary strings of length 20, with the
%  fitness function:  Sum of 1's.  Use a population of 100 with certain
%  probabilities defined below to see how long it takes to reach a string
%  of all 1's.
%
% Haupt and Haupt, 2003
% Edited and modified by Hundley and Carr, 2014

%%  I. Setup the GA parameters
ff=inline('sum(x,2)'); % fitness function

maxit=200;        % maximum number of iterations (for stopping criteria)
maxcost=9999999; % maximum allowable cost (for stopping criteria)
popsize=100; % set population size
mutrate=0.001; % set mutation rate

nbits=20; % number of bits in each parameter
npar=1;   % number of parameters in each chromosome
Nt=nbits*npar; % total number of bits in a chromosome

%% II. Create the initial population
iga=0; % generation counter initialized
pop=round(rand(popsize,Nt)); % random population of 1's and 0's (100 x 20 matrix)

% Initialize cost and other items to set up the main loop
cost=feval(ff,pop); % calculates population cost using ff
[cost,ind]=sort(cost,'descend'); % max element in first entry
pop=pop(ind,:); % sorts population with max cost first
```

```
maxc(1)=max(cost); % minc contains min of population
meanc(1)=mean(cost); % meanc contains mean of population

probs=cost/sum(cost);  % simple normalization for probabilities

%% III. MAIN LOOP

while iga<maxit

    iga=iga+1; % increments generation counter

% Choose mates
    M=ceil(popsize/2); % number of matings
    ma=RandChooseN(probs,M); % mate #1
    pa=RandChooseN(probs,M); % mate #2
    % ma and pa contain the indices of the chromosomes that will mate

    xp=ceil(rand(1,M)*(Nt-1)); % crossover point
    pop(1:2:popsize,:)=[pop(ma,1:xp(1)) pop(pa,xp(1)+1:Nt)]; % first offspring
    pop(2:2:popsize,:)=[pop(pa,1:xp(1)) pop(ma,xp(1)+1:Nt)]; % second offspring

    % Mutate the population
    nmut=ceil((popsize-1)*Nt*mutrate); % total number of mutations
    mrow=ceil(rand(1,nmut)*(popsize-1))+1; % row to mutate
    mcol=ceil(rand(1,nmut)*Nt); % column to mutate
    for ii=1:nmut
        pop(mrow(ii),mcol(ii))=abs(pop(mrow(ii),mcol(ii))-1); % toggles bits
    end % ii

%% IV. The population is re-evaluated for cost
    cost=feval(ff,pop); % calculates population cost using fitness function
    [cost,ind]=sort(cost,'descend'); % maximum element in first entry
    pop=pop(ind,:); % sorts population with maximum cost first
    maxc(iga+1)=max(cost);
    meanc(iga+1)=mean(cost);
    probs=cost/sum(cost);

%% V. Stopping criteria
    if iga>maxit || cost(1)>maxcost
        break
    end
%[iga cost(1)] Uncomment this if you want to track progress for larger
%probs
end %iga
```

```
%% VI. Displays the output as shown in Figure
day=clock;
disp(datestr(datenum(day(1),day(2),day(3),day(4),day(5),day(6)),0))
format short g
disp(['popsize=' num2str(popsize) 'mutrate=' num2str(mutrate) '# par=' num2str(npar)]);
disp(['#generations = ' num2str(iga) ' best cost = ' num2str(cost(1))]);
fprintf('best solution\n%s\n',mat2str(int8(pop(1,:))));
figure(1)
iters=0:length(maxc)-1;
plot(1:(iga+1),maxc,1:(iga+1),meanc);
xlabel('Generation');ylabel('Cost');
```

B MATLAB Code for Example 2.3

B.1 Definition of the Fitness Function

```
function elev=testfunction(loc)
% Cost function for the longitude-latitude example
%
% Carr 2014

[m,n]=size(loc);
for i=1:m
    x=loc(i,1);
    if x>10
        x=10;
    elseif x<0
        x=0;
    end
    y=loc(i,2);
    if y>10
        y=10;
    elseif y<0
        y=0;
    end

    elev(i)=x*sin(4*x)+1.1*y*sin(2*y);
end
```

B.2 Main Algorithm

```
%% Continuous Genetic Algorithm
%
```

```
% minimizes the objective function designated in ff
%
% Before beginning, set all the parameters in parts I, II, and III
% Haupt & Haupt 2003
% Edited by Hundley and Carr, 2014

%% I Setup the GA
ff='testfunction'; % objective function
npar=2; % number of optimization variables
varhi=10; varlo=0; % variable limits

%% II Stopping criteria
maxit=200; % max number of iterations
mincost=-9999999; % minimum cost

%% III GA parameters
popsize=12; % set population size
mutrate=.2; % set mutation rate
selection=0.5; % fraction of population kept
Nt=npar; % continuous parameter GA Nt=#variables
keep=floor(selection*popsize); % #population members that survive
nmut=ceil((popsize-1)*Nt*mutrate); % total number of mutations
M=ceil((popsize-keep)/2); % number of matings

%% Create the initial population
iga=0; % generation counter initialized
par=(varhi-varlo)*rand(popsize,npar)+varlo; % random

Coords{1}=par;

cost=feval(ff,par); % calculates population cost using ff
[cost,ind]=sort(cost); % min cost in element 1
par=par(ind,:); % sort continuous
minc(1)=min(cost); % minc contains min of
meanc(1)=mean(cost); % meanc contains mean of population

%% Iterate through generations (Main Loop)
while iga<maxit
    iga=iga+1; % increments generation counter

    %------------------------------------------------------
    % Pair and mate
    M=ceil((popsize-keep)/2); % number of matings
    prob=flipud([1:keep]'/sum([1:keep])); % weights chromosomes
    odds=[0 cumsum(prob(1:keep))']; % probability distribution function
```

```
pick1=rand(1,M); % mate #1 (vector of length M with random #s between 0 and 1)
pick2=rand(1,M); % mate #2

% ma and pa contain the indices of the chromosomes that will mate
%  Choosing integer k with probability p(k)
%
ic=1;
while ic<=M
    for id=2:keep+1
        if pick1(ic)<=odds(id) && pick1(ic)>odds(id-1)
            ma(ic)=id-1;
        end
        if pick2(ic)<=odds(id) && pick2(ic)>odds(id-1)
            pa(ic)=id-1;
        end
    end
    ic=ic+1;
end


%-------------------------------------------------------
% Performs mating using single point crossover

ix=1:2:keep; % index of mate #1
xp=ceil(rand(1,M)*Nt); % crossover point
r=rand(1,M); % mixing parameter

for ic=1:M

    xy=par(ma(ic),xp(ic))-par(pa(ic),xp(ic)); % ma and pa mate

    par(keep+ix(ic),:)=par(ma(ic),:); % 1st offspring
    par(keep+ix(ic)+1,:)=par(pa(ic),:); % 2nd offspring

    par(keep+ix(ic),xp(ic))=par(ma(ic),xp(ic))-r(ic).*xy;   % 1st
    par(keep+ix(ic)+1,xp(ic))=par(pa(ic),xp(ic))+r(ic).*xy; % 2nd

    if xp(ic)<npar % crossover when last variable not selected
        par(keep+ix(ic),:)=[par(keep+ix(ic),1:xp(ic))
        par(keep+ix(ic)+1,xp(ic)+1:npar)];
        par(keep+ix(ic)+1,:)=[par(keep+ix(ic)+1,1:xp(ic))
        par(keep+ix(ic),xp(ic)+1:npar)];
    end % if
end
```

```
    %---------------------------------------------------------
    % Mutate the population
    mrow=sort(ceil(rand(1,nmut)*(popsize-1))+1);
    mcol=ceil(rand(1,nmut)*Nt);
    for ii=1:nmut
        par(mrow(ii),mcol(ii))=(varhi-varlo)*rand+varlo;
    % mutation
    end % ii
    %---------------------------------------------------------
    % The new offspring and mutated chromosomes are
    % evaluated

    cost=feval(ff,par);
    %---------------------------------------------------------
    % Sort the costs and associated parameters
    [cost,ind]=sort(cost);
    par=par(ind,:);
    Coords{iga+1}=par;

    %---------------------------------------------------------
    % Do statistics for a single nonaveraging run
    minc(iga+1)=min(cost);
    meanc(iga+1)=mean(cost);
    %---------------------------------------------------------
    % Stopping criteria
    if iga>maxit || cost(1)<mincost
        break
    end
    [iga cost(1)];
end %iga


%% Displays the output
day=clock;
disp(datestr(datenum(day(1),day(2),day(3),day(4),day(5),day(6)),0))
disp(['optimized function is ' ff])
format short g
disp(['popsize=' num2str(popsize) ' mutrate=' num2str(mutrate) ' # par=' num2str(npar)])
disp(['#generations=' num2str(iga) ' best cost=' num2str(cost(1))])
disp('best solution')
disp(num2str(par(1,:)))
disp('continuous genetic algorithm')

figure(1)
iters=0:length(minc)-1;
```

```
plot(iters,minc,iters,meanc,'-');
xlabel('generation');ylabel('cost');
```

C MATLAB Code for Traveling Salesman Problem Genetic Algorithm, Section 4.1.2

C.1 Definition of the Fitness Function

```
function dist=tspfun(pop)
% Cost function for the traveling salesman problem.
% Uses global variables "x" and "y"
%
% Haupt and Haupt, 2003
% Edited and modified by Hundley and Carr, 2014

global x y

[Npop,Ncity]=size(pop);
tour=[pop pop(:,1)];

%distance between cities
for ic=1:Ncity
    for id=1:Ncity
        dcity(ic,id)=sqrt((x(ic)-x(id))^2+(y(ic)-y(id))^2);
    end % id
end % ic

% cost of each chromosome
for ic=1:Npop
    dist(ic,1)=0;
    for id=1:Ncity
        dist(ic,1)=dist(ic)+dcity(tour(ic,id),tour(ic,id+1));
    end % id
end % ic
```

C.2 Main Algorithm

```
%%   Genetic Algorithm for permutation problems
% minimizes the objective function designated in ff
%
% Haupt and Haupt, 2003
% Edited and modified by Hundley and Carr, 2014

clear
```

```
global iga x y

%% Setup the GA
ff='tspfun';   % filename objective function
npar=20;       % # of optimization variables
Nt=npar;       % # of columns in population matrix


x=rand(1,npar);
y=rand(1,npar); % cities are at (xcity,ycity)

% Uncomment next line to use the same set of cities in multiple runs
% load cities0

% Stopping criteria
maxit=10000; % max number of iterations

% GA parameters
popsize=20; % set population size
mutrate=.05; % set mutation rate
selection=0.5; % fraction of population kept

keep=floor(selection*popsize); % #population members that survive

    M=ceil((popsize-keep)/2); % number of matings
    odds=1;
    for ii=2:keep
        odds=[odds ii*ones(1,ii)];
    end
    Nodds=length(odds);
    odds=keep-odds+1; % odds determines probabilities for being parents

% Create the initial population
iga=0; % generation counter initialized
for iz=1:popsize
    pop(iz,:)=randperm(npar); % random population
end

cost=feval(ff,pop);    % calculates population cost using ff

[cost,ind]=sort(cost); % min cost in element 1
pop=pop(ind,:);        % sort population with lowest cost first
minc(1)=min(cost);     % minc contains min of population
meanc(1)=mean(cost);   % meanc contains mean of population
```

```
%% Iterate through generations (MAIN LOOP)
while iga<maxit
iga=iga+1; % increments generation counter

% Pair and mate
pick1=ceil(Nodds*rand(1,M)); % mate #1
pick2=ceil(Nodds*rand(1,M)); % mate #2
% ma and pa contain the indices of the parents
ma=odds(pick1);
pa=odds(pick2);

% Performs mating
for ic=1:M
mate1=pop(ma(ic),:);
mate2=pop(pa(ic),:);
indx=2*(ic-1)+1; % odd numbers starting at 1
xp=ceil(rand*npar); % random value between 1 and N
temp=mate1;
x0=xp;
while mate1(xp)~=temp(x0)
mate1(xp)=mate2(xp);
mate2(xp)=temp(xp);
xs=find(temp==mate1(xp));
xp=xs;
end
pop(keep+indx,:)=mate1;
pop(keep+indx+1,:)=mate2;
end

% Mutate the population
nmut=ceil(popsize*npar*mutrate);
for ic = 1:nmut
row1=ceil(rand*(popsize-1))+1;
col1=ceil(rand*npar);
col2=ceil(rand*npar);
temp=pop(row1,col1);
pop(row1,col1)=pop(row1,col2);
pop(row1,col2)=temp;
im(ic)=row1;
end
cost=feval(ff,pop);
%------------------------------------------------------
% Sort the costs and associated parameters
part=pop;
```

```
costt=cost;
[cost,ind]=sort(cost);
pop=pop(ind,:);
%_____________________________________________________
% Do statistics
minc(iga)=min(cost);
meanc(iga)=mean(cost);
end %iga

%_____________________________________________________
% Displays the output
day=clock;
disp(datestr(datenum(day(1),day(2),day(3),day(4),day(5),day(6)),0))
disp(['optimized function is ' ff])
format short g
disp(['popsize=' num2str(popsize) 'mutrate=' num2str(mutrate) '# par=' num2str(npar)])
disp([' best cost=' num2str(cost(1))])
disp(['best solution']); disp([num2str(pop(1,:))])

figure(2)
  iters=1:maxit;
  plot(iters,minc,iters,meanc,'--');
  xlabel('generation');
  ylabel('cost');

figure(1);
  plot([x(pop(1,:)) x(pop(1,1))],[y(pop(1,:)) y(pop(1,1))],x,y,'o');
  axis square
```