The Genetic Algorithm and Maximum Entropy Dice

Laura Suzanne Fellman
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Physics Commons

Let us know how access to this document benefits you.

Recommended Citation

10.15760/etd.7120

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.
THESIS APPROVAL

The abstract and thesis of Laura Suzanne Fellman for the Master of Science in Physics were presented January 29, 1996, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

Jack Semura, Chair

Erik Bodegom

Laird Brodie

Dorothy Williams

Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL:

Erik Bodegom, Chair
Department of Physics

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by on 28 February 1996
ABSTRACT


Title: The Genetic Algorithm and Maximum Entropy Dice.

The Brandeis dice problem, originally introduced in 1962 by Jaynes as an illustration of the principle of maximum entropy, was solved using the genetic algorithm, and the resulting solution was compared with that obtained analytically. The effect of varying the genetic algorithm parameters was observed, and the optimum values for population size, mutation rate, and mutation interval were determined for this problem. The optimum genetic algorithm program was then compared to a completely random method of search and optimization. Finally, the genetic algorithm approach was extended to several variations of the original problem for which an analytical approach would be impractical.
THE GENETIC ALGORITHM
AND
MAXIMUM ENTROPY DICE

by
LAURA SUZANNE FELLMAN

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
PHYSICS

Portland State University
1996
Acknowledgments

Special thanks are due to my advisor, Dr. Jack Semura, for providing me with endless ideas for research projects, and then allowing me the time and freedom to choose a topic that was of great interest to me. I would also like to thank my grandfather, Dr. Jack Fellman, whose advice and support I have valued through the years and especially appreciated in regard to this project.
# Table of Contents

**Chapter 1** Introduction  

**Chapter 2** Maximum Entropy  

2.1 The Principle of Maximum Entropy  

2.2 Traditional Maximum Entropy Formalism  

2.3 Maximum Entropy Dice: A Classic Problem Revisited  

**Chapter 3** The Genetic Algorithm  

3.1 What is the Genetic Algorithm?  

3.2 Application of the Genetic Algorithm to Maximum Entropy Dice  

**Chapter 4** Variation of Genetic Algorithm Parameters  

4.1 Population Size  

4.2 Mutation Rate  

4.3 Mutation Interval  

**Chapter 5** A Purely Random Comparison  

**Chapter 6** The n-sided Die  

**Chapter 7** Conclusion  

**References**  

**Appendix** Computer Programs
Table of Figures

Fig. 1  Analytical probability distribution for the next toss of the die, where the average of the previous tosses is (a) 4.5, (b) 3.5.

Fig. 2  Schematic flow diagram for the genetic algorithm.

Fig. 3  Probability distribution for the next toss of the die (where the average of the previous tosses is 4.5), as determined by the genetic algorithm after 100 generations.

Fig. 4  Best entropy as a function of generation for the maximum entropy dice problem with initial genetic algorithm parameters.

Fig. 5  Best, average, and worst entropies as a function of generation for the maximum entropy dice problem with initial genetic algorithm parameters.

Fig. 6  Best entropy vs. generation for populations of size 4, 10, and 40.

Fig. 7  Average number of generations required to reach 99.9 % of the analytical solution vs. population size.

Fig. 8  Total time required to reach 99.9 % of the analytical solution vs. population size.

Fig. 9  Best entropy vs. generation for mutation rates of 1/4, 1/2, 3/4, and 1.

Fig. 10  Average number of generations required to reach 99.9 % of the analytical solution vs. mutation rate, for mutation rates of 1/4, 1/2, 3/4, and 1.
Fig. 11  *Best entropy vs. generation for mutation intervals of ±.0005, ±.005, ±.05, and ±.5.*

Fig. 12  *Average number of generations required to reach 99.9% of the analytical solution vs. mutation interval.*

Fig. 13  *The best entropy obtained vs. generation for both the optimum genetic algorithm program, and a purely random search.*

Fig. 14  *Probability distribution for the next toss of a 10-sided die when the average of the previous tosses is 9 (as determined after 1000 generations).*

Fig. 15  *Best entropy vs. generation for the 10-sided die (previous average = 9).*

Fig. 16  *Probability distribution for the next toss of an 8-sided die when the average of the previous tosses is 3 (as determined after 300 generations).*

Fig. 17  *Probability distribution for the next toss of a 20-sided die when the average of the previous tosses is: 16, 18 (both after 2000 generations).*

Fig. 18  *Probability distribution for the next toss of a 14-sided die when the average of the previous tosses is: 10, 13 (both after 1500 generations).*
Chapter 1  Introduction

Optimization has long been a focus of study in physics as well as in mathematics. Perhaps the earliest contribution to this field was made by Fermat in the mid-Seventeenth century with his principle of least time, in which he demonstrated that Snell’s law holds for a refracted ray.\(^1\) This served as inspiration for, and was shortly followed by Bernoulli’s solution of the brachistochrone problem.\(^2\) In these and subsequent endeavors, the calculus of variations was born.

Today’s economists, engineers, and computer scientists seek optimal solutions to such widely differing problems as resource distribution, structural design, and function approximation. Classical methods based on calculus are appropriate for some of these problems, but for many, these methods are not suitable. In problems with a large number of variables, calculus methods become cumbersome and are often completely useless. Furthermore, these methods seek only local optima, often neglecting the desired global maxima.\(^3\) Finally and most importantly, calculus-based optimization requires the continuity of a function and the existence of its derivatives, making these methods unavailable for many problems.

In addition to calculus-based optimization, there are two other commonly recognized methods of optimization known as enumerative and random searches.
Enumerative search algorithms are used within a finite search space, and the function values at all points in the space are evaluated. This is a very straightforward method, but one with obvious drawbacks. It cannot be used on a problem with an infinite search space, and it is not efficient for a finite space of any significant size. Randomized techniques, on the other hand, exploit the search space by utilizing random choice in a highly directional search. These techniques are becoming very popular, and include such methods as simulated annealing\(^4\) and genetic algorithms. The latter will be discussed here, and applied to the problem of entropy maximization.

In Chapter 2, the principle of maximum entropy is discussed, and the traditional approach to solving maximum entropy problems is reviewed. The Brandeis dice problem is described here as well, and its analytical solution presented. In Chapter 3, the genetic algorithm is described and then applied to the same maximum entropy dice problem. In Chapter 4, the effect of varying the genetic algorithm parameters is explored, and the optimum parameter values for this particular problem are noted. Chapter 5 contains a comparison between the genetic algorithm approach to this problem and a purely random search. Chapter 6 extends the genetic algorithm solution to the problem of an n-sided die, and Chapter 7 is the conclusion.
Chapter 2  Maximum Entropy

2.1 The Principle of Maximum Entropy

When presented with incomplete information about a system, we realize that there are multiple possibilities for the state of that system which are all consistent with our limited data. It is desirable to have a criterion for selecting the most probable of these possibilities based purely on the information obtained. Suppose that there are \( n \) possible states for the system, and exactly one is true. We wish to assign probabilities \( \{ p_i \} \), \( i = 1, 2, ..., n \), to these states on the basis of our data. According to Shannon's information theory, the average missing information in this situation is

\[
S = -k \sum_{i=1}^{n} p_i \ln p_i
\]  

(2.1)

where \( k \) is a positive constant which depends on the particular situation. To ensure an unbiased assignment of probabilities \( \{ p_i \} \), it is this measure that must be maximized subject to constraints imposed by our data. Any other set of probabilities, although it may be consistent with the data, would imply the knowledge of additional information about the system.
The above expression (2.1) has an interesting history in physics. Long before Shannon identified this expression as a measure of uncertainty, Planck associated it with the thermodynamic entropy of a macroscopic system, where the probabilities represent possible microstates of the system.\(^6\) Because of the similarity in form of these two expressions, Shannon's measure of uncertainty also came to be referred to as entropy, much to the dismay of many who believed that the word 'entropy' should be reserved for thermodynamics. Although these two seemingly unrelated concepts involving the same expression developed completely independently of one another, today they are recognized to be very closely related. In fact, if we let \(k\) be the Boltzmann constant in (2.1), it is identical to the thermodynamic entropy.

2.2 Traditional Maximum Entropy Formalism

Consider a system that can be in one of \(n\) possible states. Each state is represented by a variable \(x_i\), where \(i = 1, 2, ..., n\). Assuming that we have limited information about the system, our objective is to develop a probability distribution \(\{p_i\}\) which is both consistent with our data and has maximum entropy. For the special case in which our information consists of mean values of functions of \(\{x_i\}\), there is a general solution which was formulated by Jaynes.\(^7\) In this case our data is of the form
\( \langle f_1(x_i) \rangle = F_1 \)
\( \langle f_2(x_i) \rangle = F_2 \)
\( \quad \vdots \)
\( \langle f_m(x_i) \rangle = F_m \)

where \( \{ f_1(x_i), f_2(x_i), \ldots, f_m(x_i) \} \) are all functions of \( \{x_i\} \) whose averages are respectively measured to be \( \{F_1, F_2, \ldots, F_m\} \). Equations (2.2) represent the constraints on the probabilities of our system, and they can be represented concisely as

\[
\sum_{i=1}^{n} p_i f_k(x_i) = F_k, \quad 1 \leq k \leq m.
\] (2.3)

Thus we have \( m \) constraints on the function that we wish to maximize, this function being the entropy with the \( n \) variables, \( \{p_1, p_2, \ldots, p_n\} \). Recalling that our problem is one of incomplete knowledge of the system, this corresponds to an under-constrained system in which \( m < n \). This is a standard problem which is traditionally solved using Lagrange multipliers, and the solution is presented here as outlined by Jaynes.\(^7\) The probabilities are

\[
p_i = \frac{\exp \left[ -\lambda_1 f_1(x_i) - \lambda_2 f_2(x_i) - \ldots - \lambda_m f_m(x_i) \right]}{Z(\lambda_1, \lambda_2, \ldots, \lambda_m)}
\] (2.4)
where the partition function, $Z$, is a function of the Lagrange multipliers, $\lambda_1, \lambda_2, \ldots$:

$$Z(\lambda_1, \lambda_2, \ldots, \lambda_m) = \sum_{i=1}^{n} \exp \left[ -\lambda_1 f_1(x_i) - \lambda_2 f_2(x_i) - \cdots - \lambda_m f_m(x_i) \right]. \quad (2.5)$$

The Lagrange multipliers are chosen according to the constraints, such that

$$F_k = -\frac{\partial \log Z}{\partial \lambda_k}, \quad 1 \leq k \leq m \quad (2.6)$$

which yields a set of $m$ equations with $m$ unknowns. This procedure becomes somewhat more clear in the example that follows.

### 2.3 Maximum Entropy Dice: A Classic Problem Revisited

This example and variations of it will form the focus of the remainder of this paper. It is known as the Brandeis Dice Problem, and was introduced in lectures by Jaynes in 1962 to illustrate the maximum entropy formalism. Supposing a die is tossed a number of times. Each toss results in a value $i$, where $1 \leq i \leq 6$. We would expect that for a fair die, the average value of $i$ for a large number of tosses would be 3.5, but suppose instead that the average is the unexpected value 4.5. Now suppose that this average value is the only information available to us regarding our system (the die), and we wish to assign probabilities to each possible
value of $i$ for the next toss. This situation is precisely that for which the above solution applies, as our information consists only of the mean value of a function of the variables which represent possible states of the system. More succinctly put, our constraint is

$$\sum_{i=1}^{6} i p_i = 4.5 \quad . \quad (2.7)$$

Referring back to the maximum entropy formalism, we see that since there is only one constraint, we have $m = 1$, and so there is only one Lagrange multiplier, $\lambda$.

The partition function (2.5) becomes

$$Z(\lambda) = \sum_{i=1}^{6} e^{-\lambda i} = \sum_{i=1}^{6} x^i = x + x^2 + x^3 + x^4 + x^5 + x^6 = x (1-x)^{-1} (1-x^6) \quad (2.8)$$

where $x = e^{-\lambda}$. It is now possible to solve for $x$ (and therefore $\lambda$) using equation (2.6):

$$4.5 = - \frac{\partial \log Z}{\partial \lambda} = - \frac{\partial \log [ x (1-x)^{-1} (1-x^6) ]}{\partial \lambda} = \frac{1 - 7x^6 + 6x^7}{(1-x) (1-x^6)} \quad (2.9)$$

which can be simplified to

$$3x^7 - 5x^6 + 9x - 7 = 0 \quad . \quad (2.10)$$

Solving this numerically yields the desired root $x = 1.44925$, and it follows that $\lambda = -0.37105$. From (2.8) we see that the partition function becomes
\[ Z(-.37105) = \sum_{i=1}^{6} e^{37105i} = 26.66365 \]  

and then from Eqn. (2.4) the probabilities are calculated to be

\[
\begin{align*}
p_1 &= .05435 \\
p_2 &= .07877 \\
p_3 &= .11416 \\
p_4 &= .16545 \\
p_5 &= .23977 \\
p_6 &= .34749
\end{align*}
\]  

These probabilities are a result of the maximum entropy formalism, and as such, they comprise the most unbiased probability distribution for the next toss of the die without violating the constraint (the average of the previous tosses). The probability distribution is plotted in Fig. 1a. The value of the entropy of this distribution (Eqn. (2.1), with \( k = 1 \)), is

\[ S = 1.61358 \]  

An unconstrained maximization of the entropy of this system would have yielded a value for the entropy of

\[ S_0 = 1.79176 \]  

which corresponds to the probabilities

\[ p_1 = p_2 = p_3 = p_4 = p_5 = p_6 = 1/6 \]  

This would also have been the result if we had used as our constraint, the average value of 3.5 (the expected value for a fair die) instead of 4.5. (See Fig. 1b.)
Fig. 1 Analytical probability distribution for the next toss of the die, where the average of the previous tosses is (a) 4.5, (b) 3.5.
This is a very simple example, and it is not difficult to see how tedious this method would become with additional constraints and/or a more complex system such as an n-sided die. It would therefore be advantageous to have an alternate method of constrained optimization which does not involve numerical solutions to high order polynomial equations or, more importantly, the subsequent selection of 'desired' roots, which clearly poses a problem in computer application of the traditional method.
3.1 What is the Genetic Algorithm?

Although in the 1850’s Charles Darwin certainly never intended his theory of natural selection to pertain to the survival and reproduction of strings of computer code, this is exactly what is taking place in genetic programming today. Darwin’s theory is an explanation of the evolution and adaptation of biological entities. It is based on the fact that variations in form and behavior within a reproducing population are responsible for differences in rates of survival and reproduction. Specifically, members of a population with a higher ‘fitness’ survive and reproduce at a greater rate than members with a lower fitness. In this way, those traits that contribute positively to the fitness of an individual are propagated into future generations.

In the 1970’s, John Holland, who is widely recognized to be the father of genetic programming, made the connection between natural adaptive systems such as plants and animals, and artificial adaptive systems such as solutions to optimal problems. For a system to be considered adaptive, three criteria must be met: It must consist of a population of individuals which are capable of reproduction, there must be some variation in form and/or behavior among these individuals, and their ability to survive and reproduce must depend to some extent on this variation.
Many problems of optimization can be formulated in this way, allowing a solution to evolve through 'survival of the fittest'. Initially, randomly chosen potential solutions make up a population in which individuals are evaluated according to their fitness, which is usually the quality that is being optimized. Reproduction then takes place in such a way that members of the population with greater fitness are more likely to have their characteristics or 'genes' survive into the next generation. Furthermore, in order to introduce new genes into the process, some portion of the reproducing solutions undergo a form of 'mutation'. This entire process is repeated until either a certain number of generations has elapsed or until some other specific criteria has been met.

This basic concept of a genetic algorithm can actually take many different forms involving various optional genetic operations. The conventional genetic algorithm, as presented by Sutton and Boyden, is among the simplest of these forms, and the one utilized here. It is summarized in the following steps and diagrammed in Fig. 2.

(I) Initial population members, which are represented as fixed-length strings, are chosen randomly from the search space and individually evaluated according to fitness. For example, if we wished to obtain a best-fit line for a set of data points, a potential solution would consist of two 'genes', (slope and y-intercept), so our population members would be strings of length two. Fitness in this case would be
measured by the chi-squared test, of which a low value would indicate a high fitness.

(II) In this and the following two steps, a new population or 'generation' of the same size is created. First, each position in the population is filled by a string randomly selected (with replacement) from the previous population. The probability of a particular string being selected is proportional to its fitness.

(III) Recombination occurs after the strings are grouped into pairs. Each pair is cut at a random, shared location, and the strings are spliced together, swapping genetic material.

(IV) Finally, a certain percentage of genes in the population is randomly selected to mutate according to previously defined rules, thereby introducing new genetic material.

(V) At this point, population members are once again evaluated for fitness, and the cycle is repeated beginning with step (II), until the criteria for ending the process is reached.
Step I
Initial population is created, and each member is evaluated for fitness.

Step II
New population is created from old population by fitness-proportionate reproduction.

Step III
Recombination

Step IV
Mutation

End

Step V
Population members are evaluated according to fitness. Is ending criteria met?

Fig. 2  Schematic flow diagram for the genetic algorithm.
Although this algorithm is quite simple, it has been shown to be an incredibly
efficient method of search and optimization. Among the reasons for this are the
uses of both random selection and fitness-proportionate reproduction. There has
been extensive analysis of the mathematical theory behind genetic programming by
Goldberg,3 Holland,9 and Koza.10

3.2 Application of the Genetic Algorithm to Maximum Entropy Dice

In order to utilize a genetic algorithm approach in the Brandeis dice problem, it
is necessary to decide on a form of representation for potential solutions. Since a
solution to the problem consists of six probabilities, \{p_1, ..., p_6\}, it would at first
seem reasonable for each population member to be a string of six numbers between
zero and one. After further consideration, however, we realize that these six
variables are not independent. Obviously, since they represent probabilities, their
sum must equal one. This constraint was already taken into account in the
maximum entropy formalism, but must be dealt with separately here. So the first
constraint in this situation is

$$\sum_{i=1}^{6} p_i = 1$$

(3.1)

Secondly, we consider the average value constraint of
\[ \sum_{i=1}^{6} \pi_i = 4.5 \]  

(3.2)

Because these two constraints act on our six probabilities, the actual number of independent variables in any given solution is effectively reduced to four. So a population member is a string of four probabilities, arbitrarily chosen here to be \( \{p_1, \ldots, p_4\} \), with \( p_5 \) and \( p_6 \) calculated according to the above constraints:

\[
p_5 = 6 - 6 \sum_{i=1}^{4} \pi_i + \sum_{i=1}^{4} \pi_i - 4.5
\]  

(3.3)

\[
p_6 = 1 - \sum_{i=1}^{4} \pi_i - p_5
\]  

(3.4)

After the form of population members has been decided upon, the next step is to create an initial population of solutions. Unfortunately, simple random selection of four numbers \( \{p_1, \ldots, p_4\} \) between zero and one does not necessarily yield values for \( p_5 \) and \( p_6 \) which are always positive. Instead, it is advantageous to draw randomly from numbers between zero and a fraction of one, such as 1/6, as this improves the chance of creating a valid solution. Furthermore, it is necessary to test solutions for eligibility before tossing them into the gene pool to compete. This random selection and testing continues until an eligible population of the
desired size is created. Each member of the population is subsequently evaluated for fitness which, in this case, is the entropy from Eqn. (2.1):

\[
S = -\sum_{i=1}^{6} p_i \ln p_i
\]  

(3.5)

where we set \( k = 1 \) because it is merely a constant, and therefore it has no effect on maximization. This completes step (I) of the genetic algorithm as previously outlined.

Step (II) is executed by randomly selecting two population members to compete for each position (with one exception) in the new population. In each case, the solution with the greater entropy is chosen to fill the position. This accomplishes fitness-proportionate reproduction while retaining an element of randomness. The remaining position is automatically filled by the best solution from the previous generation in order to guarantee its survival.

In step (III), the new population is first grouped into pairs. Each solution is then randomly cut at the same position as its partner, and then recombined as in the following example. Suppose that these two solutions have been paired:

\[
\begin{align*}
\{ &0.07971, 0.05623, 0.08357, 0.05354 \} \\
\{ &0.08507, 0.07086, 0.09083, 0.08512 \}
\end{align*}
\]

and suppose that the random location chosen to cut these solutions is between the third and fourth positions. After recombination, they would be
Step (IV) involves mutation, which must occur in order to provide variation from the probabilities present in the original population of solutions. If we consider the probabilities that make up each string to be ‘genes’, then mutation is carried out by allowing a randomly selected, fixed percentage of the genes in a population to vary by some small increment.

At this point, a new generation has been created from which population members can be evaluated for fitness and the entire cycle repeated, as mentioned in step (V).

The computer program which executes this procedure is located in the Appendix. The number of generations completed in the program is determined by the user, as the process will continue until terminated. At that point, the best solution of the current generation becomes the final result.

For a primary test of this program, the genetic algorithm parameters were set as follows: the population size was 10, mutation occurred in 1/2 of the genes in each population, and each affected probability was allowed to vary from its previous value by a random amount between \pm .005. The result after 100 generations (requiring about 30 seconds) is plotted in Fig. 3.
Fig. 3  *Probability distribution for the next toss of the die (where the average of the previous tosses is 4.5)*, as determined by the genetic algorithm after 100 generations.

The probabilities after 100 generations were {0.05349, 0.07944, 0.11408, 0.16720, 0.23815, 0.34764}. These varied from the analytical solution (2.12) by small amounts ranging from 0.04% to 1.6%.

An alternative way to evaluate the genetic algorithm solution is to compare the entropy of the resulting probability distribution with that of the analytical solution.
For example, the entropy of the distribution plotted in Fig. 3 is 1.61355, which compares very favorably with the value of 1.61358 from Eqn. (2.13).

It is also interesting to monitor the progress of the genetic algorithm through each generation by recording the entropy of the best solution for that generation. In Fig. 4, the ‘best of generation’ entropy is plotted for 100 generations of a sample run of the program.

![Graph showing best entropy as a function of generation for the maximum entropy dice problem with initial genetic algorithm parameters.]

Fig. 4 Best entropy as a function of generation for the maximum entropy dice problem with initial genetic algorithm parameters.
In addition to the solution with the best entropy, we can also consider for each generation, both the entropy of the worst solution and the average entropy of all members of that generation. Another sample run of the program for 50 generations is plotted this way in Fig. 5.

Fig. 5  *Best, average, and worst entropies as a function of generation for the maximum entropy dice problem with initial genetic algorithm parameters.*
Chapter 4  Variation of Genetic Algorithm Parameters

4.1 Population Size

In order to explore the effect of population size on the performance of the genetic algorithm in this problem, the program was carried out with populations of various sizes, in addition to the population size of 10 that has been used previously. All other parameters remained at their initial values. Examples of the results for populations of 4, 10, and 40 are shown in Fig. 6.

![Figure 6: Best entropy vs. generation for populations of size 4, 10 and 40.](image)

Fig. 6 Best entropy vs. generation for populations of size 4, 10 and 40.
In order to gain a more detailed understanding of the dependence of performance on population size, the program was run several times for each even population size between 2 and 40, and the average number of generations required to reach 99.9% of the analytical maximum entropy was recorded for each case. The results are plotted in Fig. 7.

Fig. 7 *Average number of generations required to reach 99.9% of the analytical solution vs. population size.*
From Fig. 7, we see that the number of generations necessary to obtain the desired result in this problem decreases rapidly as the population is increased from 2 to 6. Then as the population increases from 6 to 16, the relationship appears to be linear with a slope of 2 generations per population member. For populations between 16 and 40, the results are relatively unchanged, requiring roughly 50 generations for completion in each case.

The program was also run several times for populations of size 100 and 1000 in order to observe any long range differences in performance. For a population of 100, the result was reached after an average of 43 generations, and for 1000, an average of 39 generations. This indicates only a slight change in performance for populations differing by an order of magnitude.

After the performance of the genetic algorithm has been evaluated for populations of different size, efficiency should also be considered. Although a population of 12 may arrive at a solution in fewer generations than a population of 6, each generation in the former case requires twice as many calculations and therefore twice as much computer time as in the latter case. If we define 1 time unit as the computational time required per population member per generation, then using the same data as in Fig. 7, we can plot the total time required to obtain a solution vs. population size. This is shown in Fig. 8, in which we see that the most
efficient population for this particular problem, and the one that will therefore be used in the following sections, has a size of 4.

![Graph showing total time required to reach 99.9% of the analytical solution vs. population size.](image)

**Fig. 8**  Total time required to reach 99.9% of the analytical solution vs. population size. (One time unit is the computational time required per population member per generation, which is roughly .03 sec.)

### 4.2 Mutation Rate

In all previous examples, mutation has altered 1/2 of the genes (probabilities) in each generation of solutions. In order to explore the effects of varying the rate
of mutation, the same program was run with mutation occurring in $1/4$, $1/2$, $3/4$, and finally all of the genes in each generation. The results are displayed in Fig. 9.

![Graph showing best entropy vs. generation for mutation rates of $1/4$, $1/2$, $3/4$, and $1$.](image)

**Fig. 9** Best entropy vs. generation for mutation rates of $1/4$, $1/2$, $3/4$, and $1$.

The fact that a higher mutation rate leads to a faster solution is not surprising, considering that mutation is responsible for introducing new genetic material into the process. However, the results shown in Fig. 9 only reflect one trial run of the program for each different mutation rate. In order to determine more accurately the dependence of performance on mutation rate, the program was run several times at each rate, and once again, the average number of generations required to
reach 99.9 % of the analytical maximum entropy was recorded for each case. The results are plotted in Fig. 10.

![Average number of generations required to reach 99.9 % of the analytical solution vs. mutation rate, for mutation rates of 1/4, 1/2, 3/4 and 1.](image)

The additional computer time required to perform mutation on all of the genes in each generation is insignificant compared to the time saved by reaching a solution in fewer generations. For this reason, in future sections the mutation rate will be set at 1 (all genes mutate in every population).
4.3 Mutation Interval

In addition to the rate at which genes in a population undergo mutation, we should also consider the size of the interval over which the probabilities are allowed to vary. Previously, this was set at $\pm .005$, and a selected gene was altered by a random amount between these two extremes. To obtain a general idea of the effect of changing this interval on the performance of the genetic algorithm in this problem, the program was run using several different mutation intervals.

![Fig. 11](image_url)  
*Fig. 11  Best entropy vs. generation for mutation intervals of $\pm .0005$, $\pm .005$, $\pm .05$, and $\pm .5$.***
From Fig. 11, we see that increasing the size of the mutation interval over several orders of magnitude does not have either a continuously positive or negative effect on performance. When mutation occurs over a very small interval, such as ± .0005, the number of generations necessary to achieve a final result becomes very large. Increasing the interval to ± .005 dramatically improves performance. Further increases in mutation interval, although expedient in the first few generations, appear to eventually delay the achievement of a final result.

In order to confirm these original observations, several intermediate mutation intervals were used in addition to those mentioned above, and once again the average number of generations required to reach 99.9 % of the analytical solution was recorded for each case. This is shown in Fig. 12, in which it can be seen that a solution is achieved in the least number of generations for a mutation interval of ± .02. Thus, in future sections, this will be the value used for mutation interval.

Actually, performance is quite comparable for intervals between ± .005 and ± .03, but declines noticeably for intervals outside of this range. Although a smaller interval guarantees eventual success, the number of generations necessary to attain a solution in this case grows steadily as the mutation interval decreases. For larger intervals, what appears to be lacking is the ‘fine-tuning’ of the probabilities that allows the achievement of a result that is at least 99.9 % of the analytical solution. For example, it was observed in many such cases, that the ‘best entropy of generation’ would approach the analytical value and then remain at
some intermediate value for hundreds of generations without improvement, as evident in Fig. 11 for mutation intervals of both ±.05 and ±.5.

Fig. 12 Average number of generations required to reach 99.9% of the analytical solution vs. mutation interval.
Chapter 5  A Purely Random Comparison

The genetic algorithm was previously described as a method of optimization that exploits the search space by utilizing random choice in a highly directional search. We are now able to solve the maximum entropy dice problem with a genetic algorithm program in which all parameters have been adjusted for maximum efficiency. It is interesting, at this point, to consider the extent to which the genetic algorithm provides 'direction', as opposed to a purely random search. In this chapter, we wish to address the question, "How much more efficient is such a genetic algorithm approach to this problem, than a purely random search?"

In order to answer this question, a program was written in which random solutions to the same maximum entropy dice problem were generated (see Appendix). For each solution, 4 probabilities between 0 and 1/6 were randomly generated, the remaining two probabilities once again being determined by Eqs. (3.3) and (3.4). Solutions were tested for eligibility, and those qualifying were grouped into 'generations' of 4 so that a comparison could be made between the progress of this program and that of the optimum genetic algorithm program which uses a population size of 4.
In the previous chapter, it was discovered that when parameters are set appropriately, the genetic algorithm achieves a solution to this problem that is 99.9% of the analytical solution in approximately 55-60 generations. After 200,000 generations of randomly generated solutions from the other program (requiring just over 4 hours), the best entropy achieved was 1.55240, only 96.2% of the analytical solution. After one million generations (requiring over 20 hours), the situation was improved only slightly with a result of 1.56214, which is 96.8% of the analytical solution. The 99.9% value quite possibly may never have been attained, even if the program were left running until this moment!

A sample run of the purely random program is compared with a sample run of the genetic algorithm program in Fig. 13. In each case, the best entropy obtained as of each generation is plotted for the first 1000 generations. It is clear from these examples that in the purely random case, improvement in the value for best entropy occurs in much smaller increments than in the genetic algorithm case, and very infrequently after the first 50 generations.
Fig. 13  The best entropy obtained vs. generation for both the optimum genetic algorithm program, and a purely random search.
Chapter 6  The n-sided Die

Until this point, we have considered the case of a 6-sided die which has been thrown a number of times previously to yield an average value of 4.5. The problem of determining the probabilities for the next throw of the die was solved first analytically, and then with the genetic algorithm. A major advantage of the genetic algorithm approach to optimization, is the ease with which it handles problems with a large number of variables. It is for this reason that we are now able to extend a solution of the maximum entropy dice problem to a hypothetical n-sided die, using the same genetic algorithm program adjusted for a variable number of sides and previous average. This program can be found in the Appendix.

The first case explored was that of a 10-sided die with a value of 9 for the average of the previous tosses. After 1000 generations, the best entropy ceased to show significant improvement, and the resulting probability distribution is shown in Fig. 14. The progress of the genetic algorithm over those 1000 generations can be observed in Fig. 15, where the best entropy vs. generation plot follows a path similar in shape to those of the 6-sided die. The fact that a result in this case requires a greater number of generations than the 6-sided die is not surprising, considering that we have doubled the effective number of variables in the problem.
Fig. 14 Probability distribution for the next toss of a 10-sided die when the average of the previous tosses is 9 (as determined after 1000 generations).

Fig. 15 Best entropy vs. generation for the 10-sided die (previous average = 9).
The genetic algorithm program was run for some additional values of n, with various values for previous average. The program was terminated in each case when the best entropy ceased to show significant improvement. The results are displayed in Figs. 16-18. In each example, as in the preceding examples, the shape of the resulting probability distribution is exponential, as predicted by the analytical solution (Eqn. (2.4)).

**Fig. 16** Probability distribution for the next toss of an 8-sided die when the average of the previous tosses is 3 (as determined after 300 generations).
Fig. 17  Probability distribution for the next toss of a 20-sided die when the average of the previous tosses is: 16(top), 18(bottom), both after 2000 generations.
Fig. 18  Probability distribution for the next toss of a 14-sided die when the average of the previous tosses is: 10(top), 13(bottom), both after 1500 generations.
Chapter 7  Conclusion

In applying the genetic algorithm to any problem of optimization, perhaps the most important step is determining a representation for solutions which will fit within the framework of the process. After some thought on the matter, almost any problem can be structured in such a way that potential solutions are allowed to compete and strive for optimal fitness. This was the case with the problem of maximum entropy dice.

As demonstrated in Chapter 4, adjustment of the genetic algorithm parameters has a significant effect on the performance of this algorithm. Both the efficiency of the process and the ultimate accuracy of the final solution depend upon the values of these parameters (population size, mutation rate, and mutation interval).

The genetic algorithm has been shown to be an effective method of solving the Brandeis dice problem by providing a numerical solution that compares very favorably with the analytical solution. Furthermore, the genetic algorithm approach can be easily extended to the problem of an n-sided die, unlike the analytical method.
References


Appendix  Computer Programs

(All programs are written in QBASIC.)

********************PROGRAM # 1: THE 6-SIDED DIE********************

***This program utilizes the genetic algorithm to find the maximum entropy probability distribution for the next toss of a 6-sided die, when the average of the previous tosses is 4.5. ********************

CLS
SCREEN 12
WINDOW (.5, 1)-(7, -1)
RANDOMIZE TIMER

pop = 4
***population size***
avg = 4.5

DIM prob(pop, 6)
DIM bestprob(6)
DIM survivor(pop, 6)
DIM ent(pop)
DIM sum1(pop)
DIM sum2(pop)
DIM reserve(pop, 6)

********************Creation of the initial population********************

FOR i = 1 TO pop
  tryagain:
  sum1(i) = 0
  sum2(i) = 0
  FOR j = 1 TO 4
    prob(i, j) = RND * 1 / 6
    sum1(i) = sum1(i) + prob(i, j)
    sum2(i) = sum2(i) + j * prob(i, j)
  NEXT j
  prob(i, 5) = 6 - 6 * sum1(i) + sum2(i) - avg
  prob(i, 6) = 1 - sum1(i) - prob(i, 5)
  IF prob(i, 5) <= 0 OR prob(i, 6) <= 0 THEN GOTO tryagain
NEXT i

bestent = 0
FOR j = 1 TO 6
bestprob(j) = prob(1, j)
bestent = bestent - bestprob(j) * LOG(bestprob(j))
NEXT j

z = 0
DO
CLS
z = z + 1
PRINT z
PRINT "press escape to quit"

**************************************************************************Evaluation of fitness**************************************************************************

FOR i = 1 TO pop
ent(i) = 0
FOR j = 1 TO 6
ent(i) = ent(i) - prob(i, j) * LOG(prob(i, j))
NEXT j
NEXT i

FOR i = 1 TO pop
IF ent(i) > bestent THEN
bestent = ent(i)
FOR j = 1 TO 6
bestprob(j) = prob(i, j)
NEXT j
END IF
NEXT i

PRINT " "
FOR j = 1 TO 6
LINE (j, bestprob(j))-(j+1, 0), B
PRINT "p", j, ";", bestprob(j)
NEXT j
PRINT " 
PRINT "best entropy ="; bestent
*******Fitness proportionate reproduction***************

FOR i = 1 TO (pop - 1)
a = INT(RND * pop) + 1
B = INT(RND * pop) + 1
IF ent(a) > ent(B) THEN c = a ELSE c = B
FOR j = 1 TO 6
survivor(i, j) = prob(c, j)
NEXT j
NEXT i

FOR j = 1 TO 6
survivor(pop, j) = bestprob(j)
NEXT j

FOR i = 1 TO pop
FOR j = 1 TO 6
reserve(i, j) = survivor(i, j)
NEXT j
NEXT i

******************Recombination*******************************

FOR w = 1 TO (pop / 2)
FOR r = 1 TO (INT(RND * 3) + 1) 'splits each gene pair randomly
SWAP survivor(w, r), survivor(w + (pop / 2), r)
NEXT r
NEXT w

FOR i = 1 TO pop
sum1(i) = 0
sum2(i) = 0
FOR j = 1 TO 4
sum1(i) = sum1(i) + survivor(i, j)
sum2(i) = sum2(i) + j * survivor(i, j)
NEXT j
survivor(i, 5) = 6 - 6 * sum1(i) + sum2(i) - avg
survivor(i, 6) = 1 - sum1(i) - survivor(i, 5)
IF survivor(i, 5) <= 0 OR survivor(i, 6) <= 0 THEN
FOR j = 1 TO 6
survivor(i, j) = reserve(i, j)
NEXT j
ELSE
END IF
NEXT i
NEXT j
END IF
NEXT i

**************************************************************************Mutation**************************************************************************

FOR i = 1 TO pop
FOR d = 1 TO 4
p = INT(RND * 4) + 1
IF INKEY$ = CHR$(27) THEN END
here:
survivor(i, p) = survivor(i, p) - .02 + RND * .04
IF survivor(i, p) <= 0 THEN
FOR j = 1 TO 6
survivor(i, j) = reserve(i, j)
NEXT j
GOTO here
ENDIF
NEXT d
sum1(i) = 0
sum2(i) = 0
FOR j = 1 TO 4
sum1(i) = sum1(i) + survivor(i, j)
sum2(i) = sum2(i) + j * survivor(i, j)
NEXT j
survivor(i, 5) = 6 - 6 * sum1(i) + sum2(i) - avg
survivor(i, 6) = 1 - sum1(i) - survivor(i, 5)
IF survivor(i, 5) <= 0 OR survivor(i, 6) <= 0 THEN
FOR j = 1 TO 6
survivor(i, j) = reserve(i, j)
NEXT j
ENDIF
NEXT i

FOR i = 1 TO pop
FOR j = 1 TO 6
prob(i, j) = survivor(i, j)
NEXT j
NEXT i

LOOP
END
***************PROGRAM #2: RANDOM SOLUTIONS***************

***This program generates random solutions to the 6-sided die problem. These solutions are grouped into 4's to facilitate comparison with the genetic algorithm program with optimum parameters.******************

DIM prob(6)
RANDOMIZE TIMER

bestent = 0
count = 1
num = 0

DO
num = num + 1
IF num = 5 THEN
  num = 1
  count = count + 1
END IF

tryagain:
  sum1 = 0
  sum2 = 0
  FOR j = 1 TO 4
    prob(j) = RND * 1 / 6
    sum1 = sum1 + prob(j)
    sum2 = sum2 + j * prob(j)
  NEXT j
  prob(5) = 6 - 6 * sum1 + sum2 - 4.5
  prob(6) = 1 - sum1 - prob(5)
  IF prob(5) <= 0 OR prob(6) <= 0 THEN GOTO tryagain
  NEXT j

entropy = 0
  FOR j = 1 TO 6
    entropy = entropy - prob(j) * LOG(prob(j))
  NEXT j

IF entropy >= bestent THEN bestent = entropy

CLS
PRINT count
PRINT bestent
LOOP UNTIL INKEY$ = CHR$(27)
END
PROGRAM #3: THE N-SIDED DIE

This program finds the probability distribution for the next toss of an n-sided die with a specified previous average. The values shown are for a 14-sided die with a previous average of 10.

CLS
SCREEN 12

sides = 14  'must be an even number

WINDOW (.5, 1)-(sides + 1, -.1)
RANDOMIZE TIMER

pop = 4  ***population size***
avg = 10

DIM prob(pop, sides)
DIM bestprob(sides)
DIM survivor(pop, sides)
DIM ent(pop)
DIM sum1(pop)
DIM sum2(pop)
DIM reserve(pop, sides)

Creation of the initial population

count = 0
fraction = 1 / (2 * sides)
m = fraction

FOR i = 1 TO pop
  IF count = 100 THEN  'if it ends without creating a gene pool, make this larger
    m = m + fraction  'or make fraction smaller
    IF m >= 1 THEN
      PRINT "Change initial gene pool selection"
      END
  END IF
  count = count + 1
END
END IF
sum1(i) = 0
sum2(i) = 0
FOR j = 1 TO sides - 2
prob(i, j) = RND * m
sum1(i) = sum1(i) + prob(i, j)
sum2(i) = sum2(i) + j * prob(i, j)
NEXT j
prob(i, sides - 1) = sides - sides * sum1(i) + sum2(i) - avg
prob(i, sides) = 1 - sum1(i) - prob(i, sides - 1)
IF prob(i, sides - 1) <= 0 OR prob(i, sides) <= 0 THEN GOTO tryagain ELSE count = 0
NEXT i

bestent = 0
FOR j = 1 TO sides
bestprob(j) = prob(1, j)
bestent = bestent - bestprob(j) * LOG(bestprob(j))
NEXT j
z = 0
DO
CLS
z = z + 1
PRINT z
PRINT "press escape to quit"

***************************Evaluation of fitness***************************

FOR i = 1 TO pop
ent(i) = 0
FOR j = 1 TO sides
ent(i) = ent(i) - prob(i, j) * LOG(prob(i, j))
NEXT j
NEXT i

FOR i = 1 TO pop
IF ent(i) > bestent THEN
bestent = ent(i)
FOR j = 1 TO sides
bestprob(j) = prob(i, j)
NEXT j
END IF
NEXT i
FOR j = 1 TO sides  
LINE (j, bestprob(j))-(j + 1, 0), , B  
PRINT "p": j: "=": bestprob(j)  
NEXT j  
PRINT " "  
PRINT "best entropy :=": bestent  

************************Fitness proportionate reproduction************************

FOR i = 1 TO (pop - 1)  
a = INT(RND * pop) + 1  
B = INT(RND * pop) + 1  
IF ent(a) > ent(B) THEN c = a ELSE c = B  
FOR j = 1 TO sides  
survivor(i, j) = prob(c, j)  
NEXT j  
NEXT i  

FOR j = 1 TO sides  
survivor(pop, j) = bestprob(j)  
NEXT j  

FOR i = 1 TO pop  
FOR j = 1 TO sides  
reserve(i, j) = survivor(i, j)  
NEXT j  
NEXT  

*****************************Recombination*****************************

FOR w = 1 TO (pop / 2)  
FOR r = 1 TO (INT(RND *(sides - 3)) + 1)  
'Splits each gene pair randomly  
SWAP survivor(w, r), survivor(w + (pop / 2), r)  
NEXT r  
NEXT w  

FOR i = 1 TO pop  
sum1(i) = 0  
sum2(i) = 0  
FOR j = 1 TO sides - 2  
sum1(i) = sum1(i) + survivor(i, j)  
NEXT j
sum2(i) = sum2(i) + j * survivor(i, j)
NEXT j

survivor(i, sides - 1) = sides - sides * sum1(i) + sum2(i) - avg
survivor(i, sides) = 1 - sum1(i) - survivor(i, sides - 1)

IF survivor(i, sides - 1) <= 0 OR survivor(i, sides) <= 0 THEN
FOR j = 1 TO sides
survivor(i, j) = reserve(i, j)
NEXT j
END IF
NEXT i

***********Mutation***********

FOR i = 1 TO pop
FOR d = 1 TO (sides - 2)

e = INT(RND * (sides - 2)) + 1

IF INKEY$ = CHR$(27) THEN END

here:

survivor(i, e) = survivor(i, e) - .02 + RND * .04

IF survivor(i, e) <= 0 THEN
FOR j = 1 TO sides
survivor(i, j) = reserve(i, j)
NEXT j
GOTO here
END IF
NEXT d

sum1(i) = 0
sum2(i) = 0
FOR j = 1 TO sides - 2
sum1(i) = sum1(i) + survivor(i, j)
sum2(i) = sum2(i) + j * survivor(i, j)
NEXT j

survivor(i, sides - 1) = sides - sides * sum1(i) + sum2(i) - avg
survivor(i, sides) = 1 - sum1(i) - survivor(i, sides - 1)

IF survivor(i, sides - 1) <= 0 OR survivor(i, sides) <= 0 THEN
FOR j = 1 TO sides
survivor(i, j) = reserve(i, j)
NEXT j
END IF
NEXT i

FOR i = 1 TO pop
FOR j = 1 TO sides
prob(i, j) = survivor(i, j)
NEXT j
NEXT i

LOOP

END