POLYTECHNIC UNIVERSITY
Department of Computer and Information Science

GENETIC ALGORITHMS

K. Ming Leung

Abstract: The use of genetic algorithms in optimization problems is introduced. The differences of the approach from traditional methods are pointed out. The power of the genetic algorithms can be analyzed using the concept of schema or similarity templates.

Directory

Copyright © 2000 mleung@poly.edu
Last Revision Date: March 30, 2004

Table of Contents

1. Introduction

Genetic algorithms are computer algorithms or procedures modeled after genetics and the laws of natural selection. Genetics algorithms are often used to conduct searches and in particular to find optimal solutions for optimization problems.[2] We will consider a more common form of an optimization problem known as unconstrained optimization. In order to be able to fully appreciate the power of genetic algorithms we will first discuss briefly some of the more traditional optimization methods. We will see some of their strengths and shortcomings when compared with genetic algorithms.

2. Traditional Methods in Optimization

In an optimization problem, one has a scalar function f, which dependents on a set of parameters, $x_1, x_2, \ldots, x_N$. The goal is to find a set of values for $x_1, x_2, \ldots, x_N$ which maximizes the function. The function is often referred to as the payoff function. In genetic algorithms, the function is more commonly known as the fitness function. By reversing the sign of the function we can of course also consider

minimization.

We will briefly mention some of the traditional methods for optimization problems.

2.1. Enumerative

The basis for enumerative techniques is simplicity itself. To find the optimum value in a problem space (which is finite), look at the function values at every point in the space. The problem here is obvious. This is horribly inefficient. For very large problem spaces, the computational task is massive, perhaps intractably so.

2.2. Random Search Algorithms

Random searches simply perform random walks of the problem space, recording the best optimum values discovered so far. Efficiency is a problem here as well. For large problem spaces, they should perform no better than enumerative searches. They do not use any knowledge gained from previous results and thus are both dumb and blind.

2.3. Hill Climbing

Hill climbing optimization techniques have their roots in the classical mathematics developed in the 18th and 19th centuries. In essence, this class of search methods finds an optimum by following the local gradient of the function (they are sometimes known as gradient methods). They are deterministic in their searches. They generate successive results based solely on the previous results.

The steepest descent method is one of the more well-known methods within this class. Starting at an initial point $x_1, x_2, \ldots, x_N$ in an N-dimensional space the gradient at that point is computed and a step is made in a direction opposite to that direction. The procedure is then repeated until a maximum of the function is found. An example of the trajectory traced by these points are shown in the following figure for a case in which f has a rather simple form.

There are several drawbacks to hill climbing methods. Firstly, they assume that the problem space being searched is continuous in nature. In other words, derivatives of the function representing the problem space exist. This is not true of many real world problems

where the problem space is noisy and discontinuous.

Another major disadvantage of using hill climbing is that hill climbing algorithms only find the local optimum in the neighborhood of the current point. They have no way of looking at the global picture in general. However, parallel methods of hill-climbing can be used to search multiple points in the problem space. This still suffers from the problem that there is no guarantee of finding the optimum value, especially in very noisy spaces with a multitude of local peaks or troughs. Examples of some of these more challenging types of functions are shown in the following figures.

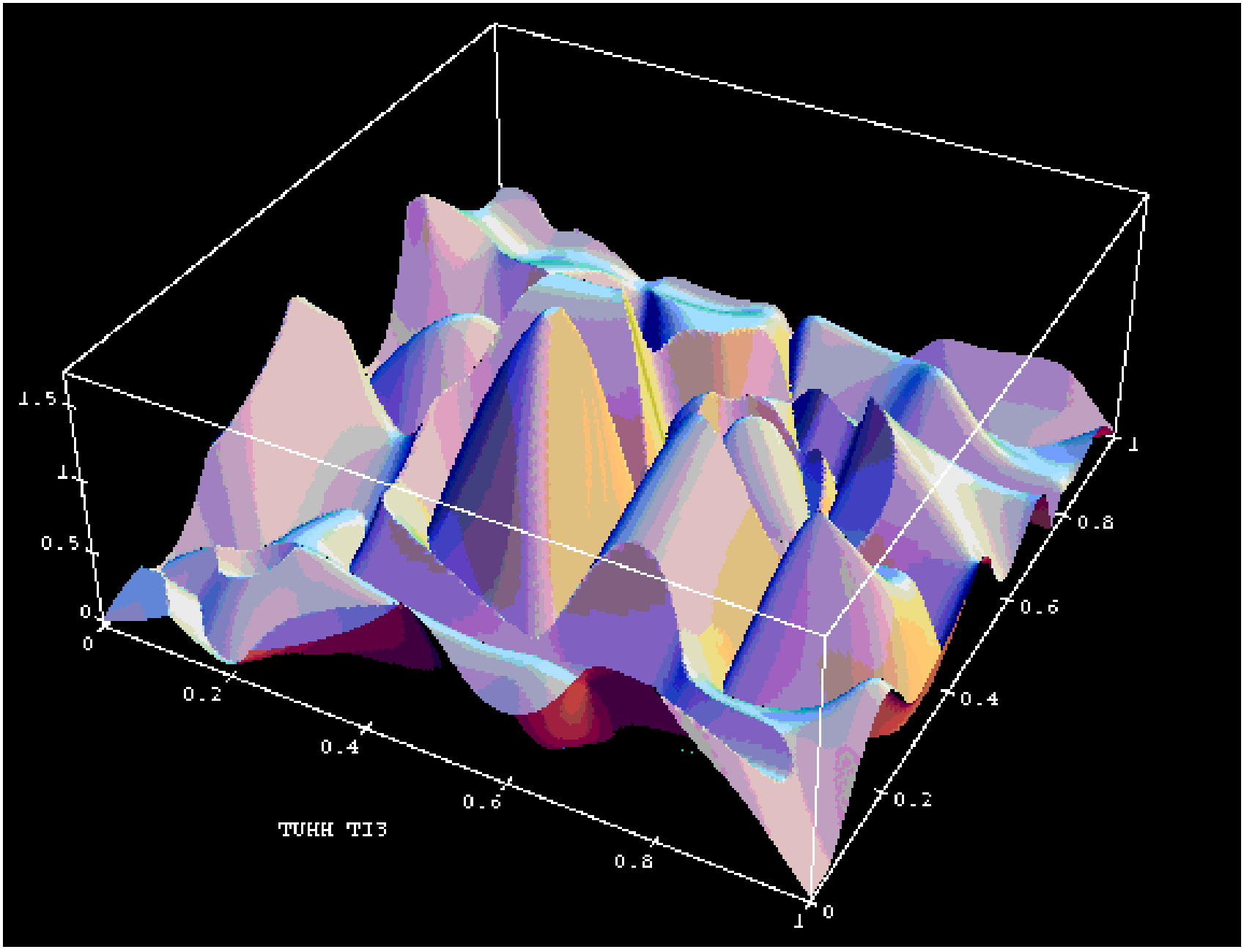
2.4. Randomized Search Techniques

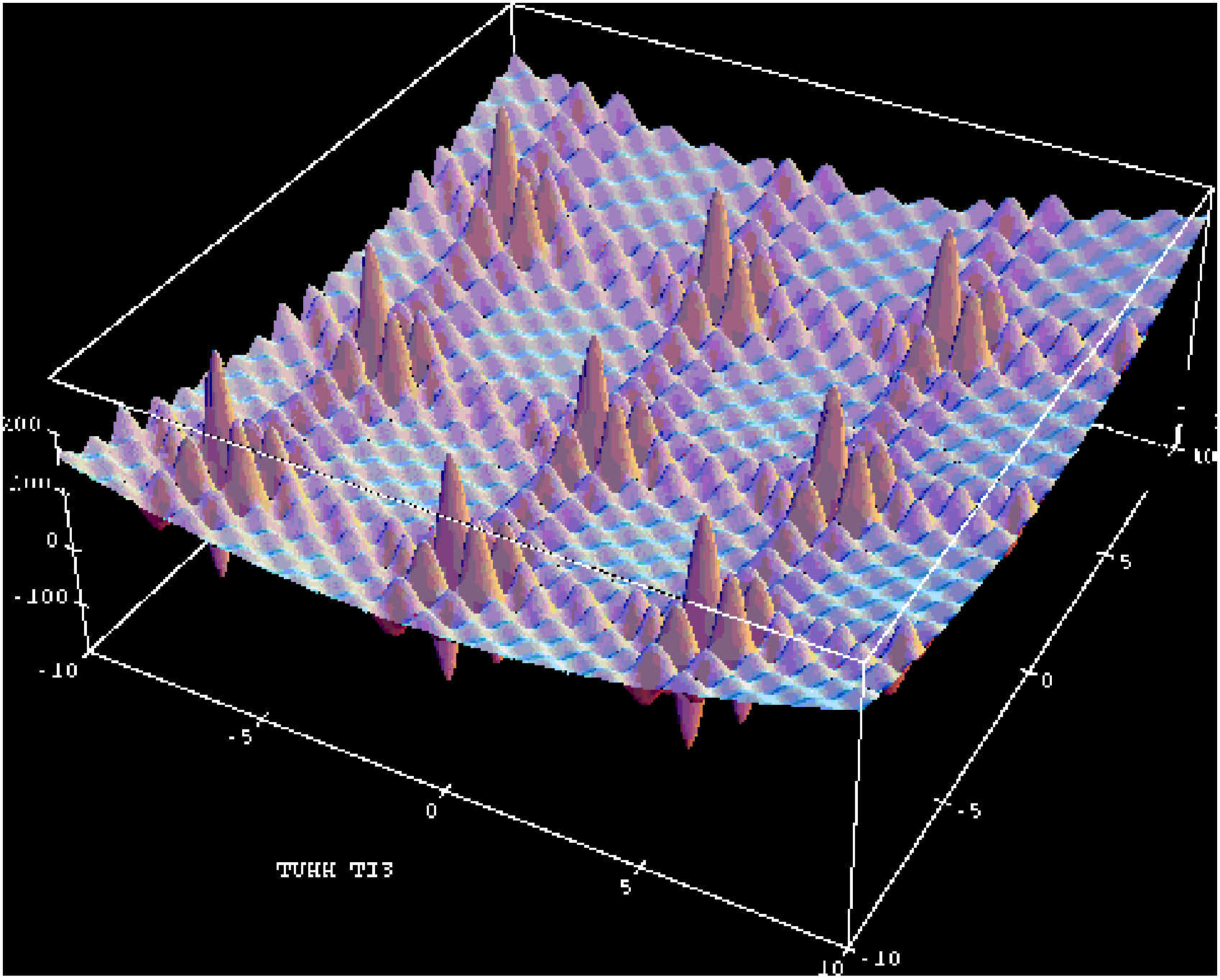
Randomized search algorithms uses random choice to guide themselves through the problem search space. But these are not just simply random walks. These techniques are not directionless like the random search algorithms. They use the knowledge gained from previous results in the search and combine them with some randomizing features. The result is a powerful search technique that can handle

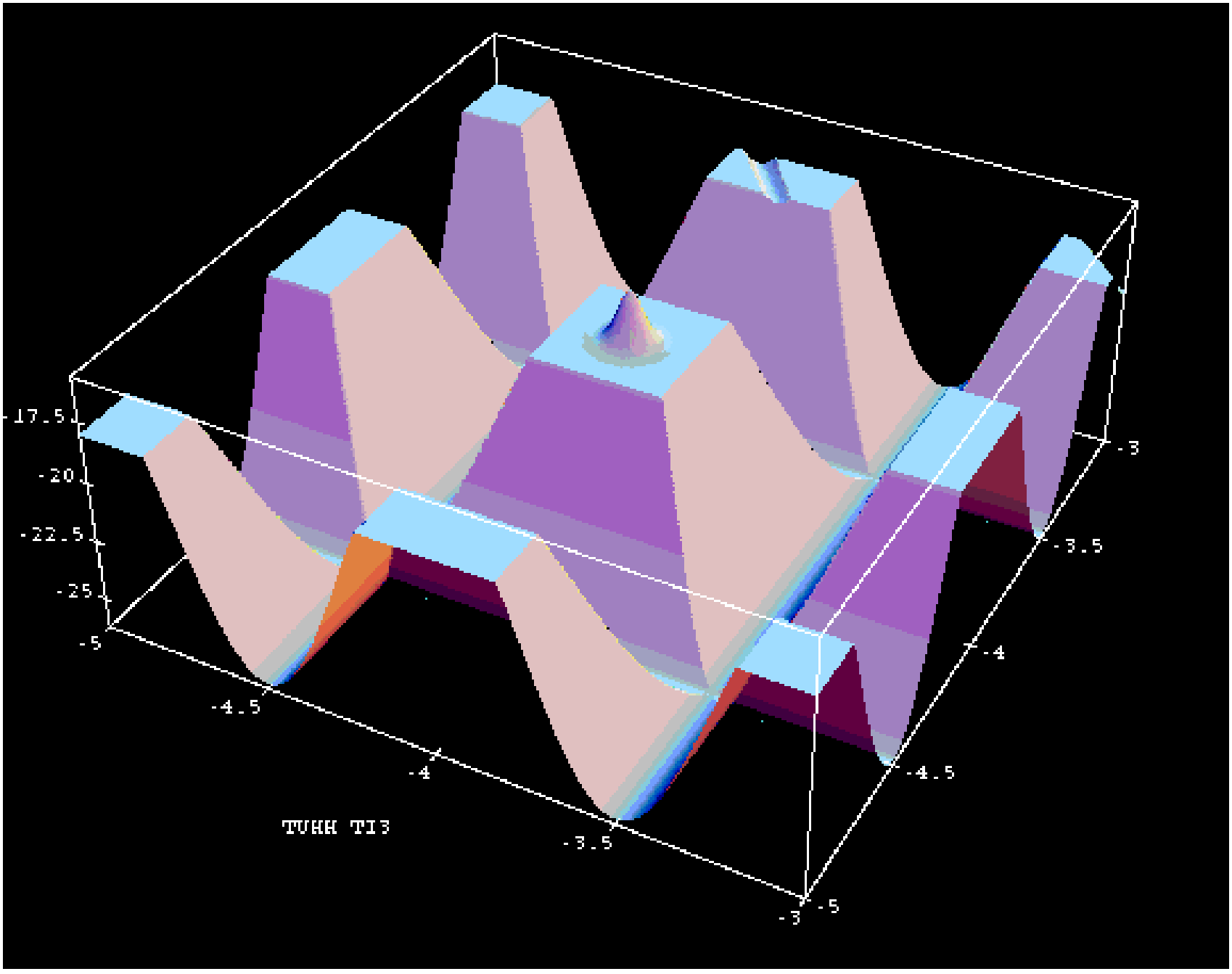
noisy, multimodal (having multiple peaks) search spaces with some relative efficiency. The two most popular forms of randomized search algorithms are simulated annealing and genetic algorithms.

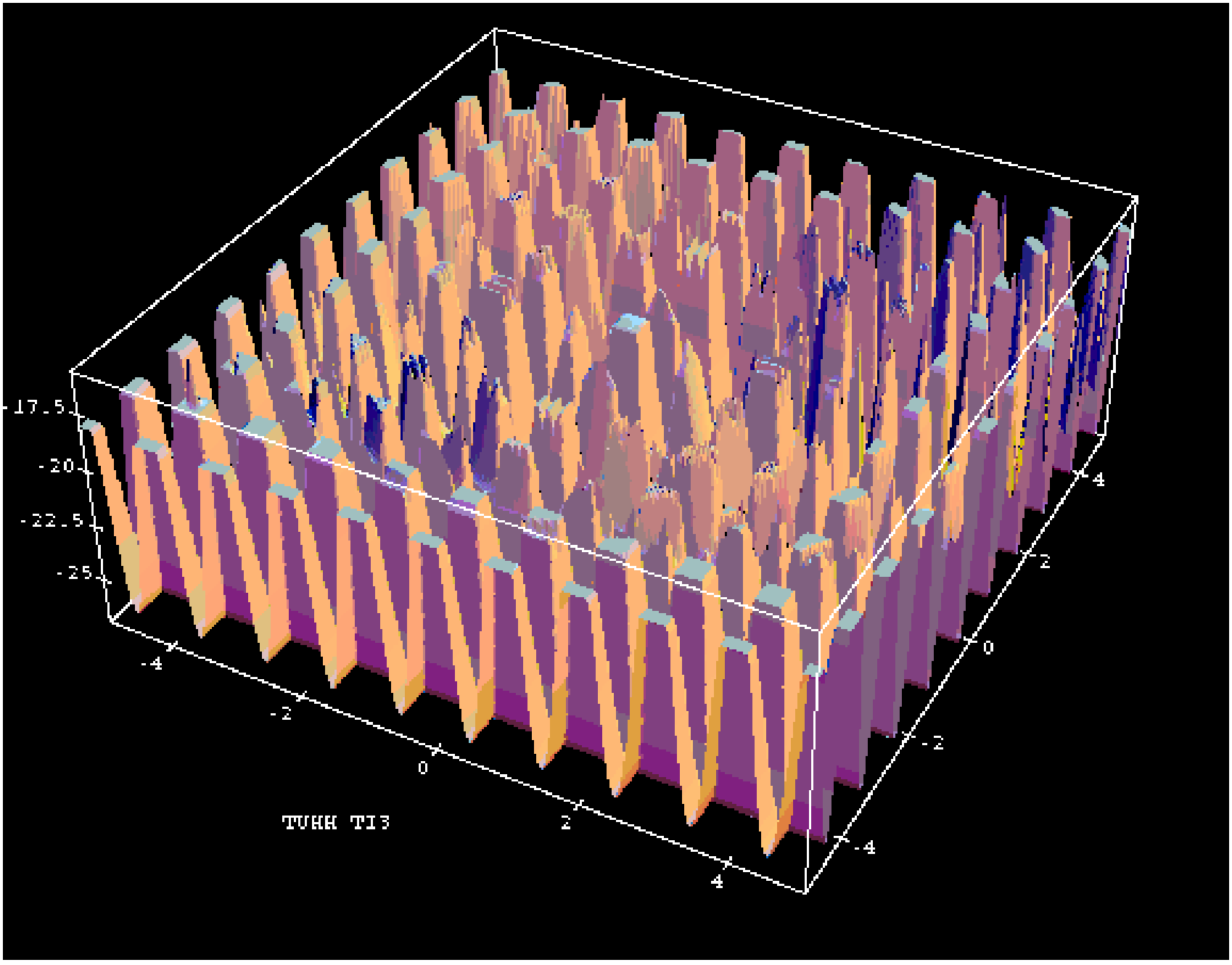
2.5. Some Challenging Problem Spaces

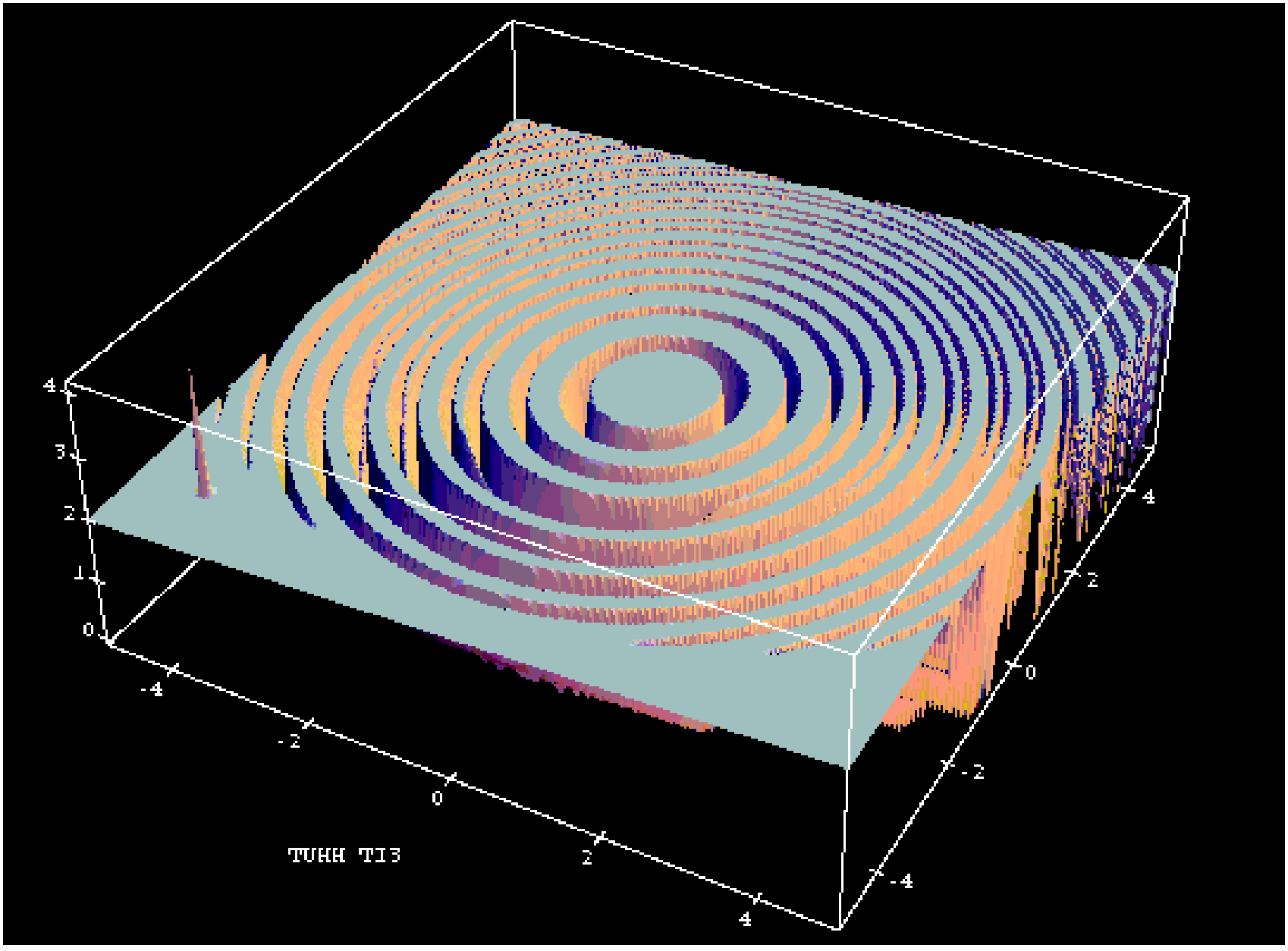
Optimization problems usually become rather challenging when the number of parameters N in the problem is large and when the function to be optimized is rather complex. Examples of some of the most complex functions in two dimensional parameter space are shown in the following figures. Traditional optimization techniques do not have a chance in attacking these problems.

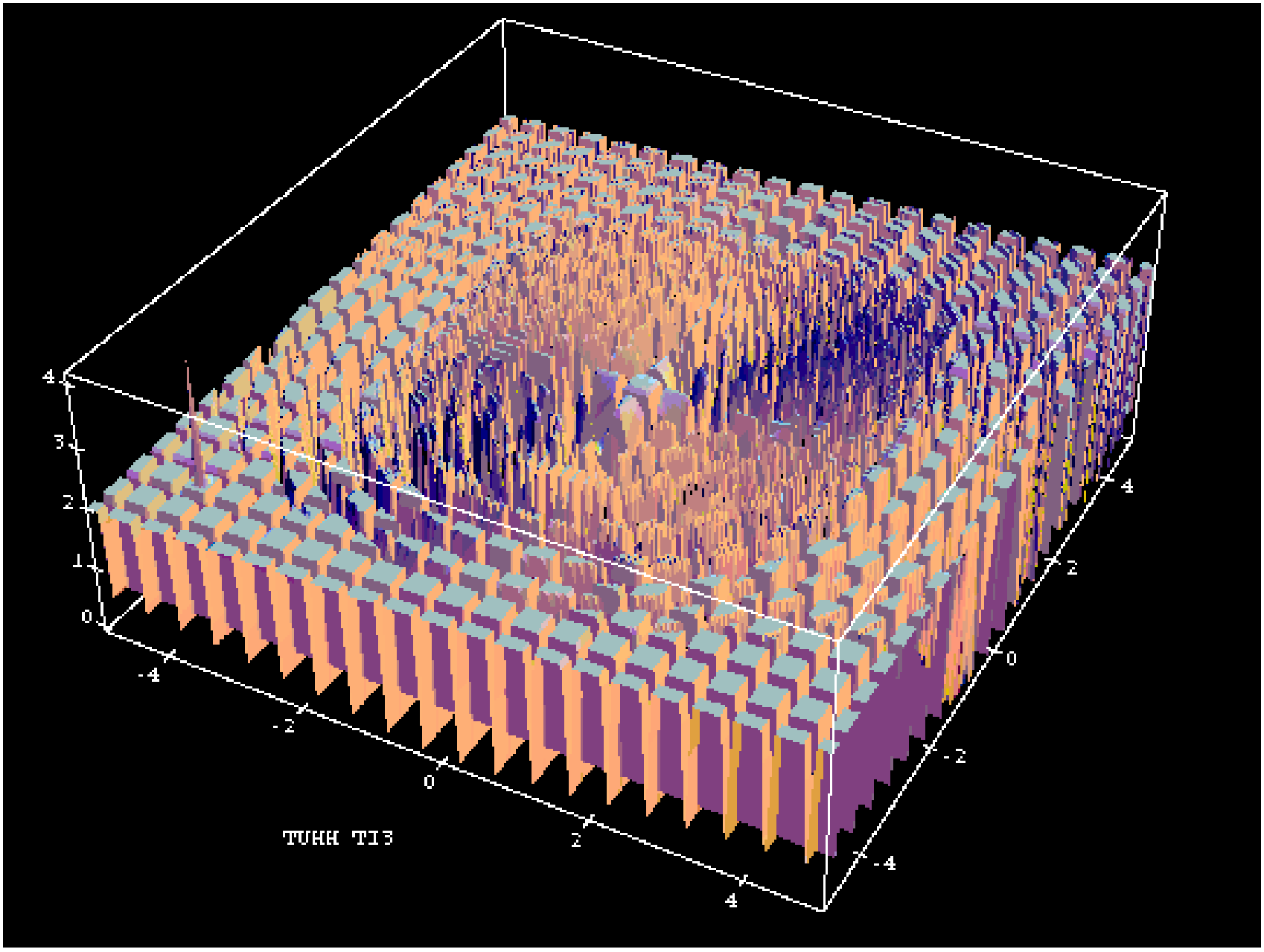

1.5
1
0.5
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1


200
100
0
-100
-10
-5
0
5
10
-10
-5
0
5
10


-17.5
-20
-22.5
-25
-5
-4.5
-4
-3.5
-3
-5
-4.5
-4
-3.5
-3
TUHH TI3


-17.5
-20
-22.5
-25
-4
-2
0
2
4
TUHH TI3
4
2
0
-2
-4


4
3
2
1
0
-4
-2
0
2
4
TUHH TI3
4
2
0
-2
-4



It is clear that these functions are artificially made up to illustrate

some of the subtle problems that can be encountered in optimizing a function. In real-world problems, these subtleties arise most often when the number of parameters are large and visualization of the function is then impossible.

3. Genetic Algorithms

The inspiration and motivation of genetic algorithms comes from looking at the world around us and seeing a staggering diversity of life. Millions of species, each with its own unique behavioral patterns and characteristics, abound. Yet, all of these plants and creatures have evolved, and continue evolving, over millions of years. Each species has developed physical features and normal habits that are in a sense optimal in a constantly shifting and changing environment in order to survive. Those weaker members of a species tend to die away, leaving the stronger and fitter to mate, create offspring and ensure the continuing survival of the species. Their lives are dictated by the laws of natural selection and Darwinian evolution. And it is upon these ideas that genetic algorithms are based.

We have to understand to see how each species has evolved throughout the ages to such an almost perfect status. Within most cells in the human body (and in most other living organisms) are rodlike structures called chromosomes. These chromosomes dictate various hereditary aspects of the individual. Within the chromosomes are individual genes. A gene encodes a specific feature of the individual. For example, a person's eye color is dictated by a specific gene. The actual value of the gene is called an allele.

This is a grossly oversimplified look at genetics, but will suffice to show its correlation with genetic algorithms. A hierarchical picture is built up, with alleles being encoded as genes, with sequences of genes being chained together in chromosomes, which makes up the DNA of an individual.

When two individuals mate, both parents pass their chromosomes onto their offspring. In humans, who have 46 paired chromosomes in total, both parents pass on 23 chromosomes each to their child. Each chromosome passed to the child is an amalgamation of two chromosomes from a parent. The two chromosomes come together and swap genetic material, and only one of the new chromosome strands

is passed to the child. So the chromosome strands undergo a crossover of genetic material, which leads to a unique new individual.

As if this were not enough, genetic material can undergo mutations, resulting from imperfect crossovers or other external stimuli. Although mutation is rare, it does lead to an even greater diversification of the overall gene pool of the population. It must be noted however that too much of mutation is in fact harmful and can destroy good genetic code, so the rate of mutation must be low in order to prevent severe degradation of the genetic pool.

Genetic algorithms exploit the idea of the survival of the fittest and an interbreeding population to create a novel and innovative search strategy. The first step in a GA is to find a suitable encoding of the parameters of the fitness function. This is usually done using a population of strings, each representing a possible solution to the problem. Instead of using strings other schemes such as trees and matrices can also be used to encode the parameters. Notice that an entire population of strings rather than a single string is used. The genetic algorithm then iteratively creates new populations from the old by ranking the strings and interbreeding the fittest to create new

strings, which are (hopefully) closer to the optimum solution to the problem at hand. So in each generation, the GA creates a set of strings from the bits and pieces of the previous strings, occasionally adding random new data to keep the population from stagnating. The end result is a search strategy that is tailored for vast, complex, multimodal search spaces.

Genetic algorithms are a form of randomized search, in that the way in which strings are chosen and combined is a stochastic process. This is a radically different approach to the problem solving methods used by more traditional algorithms, which tend to be more deterministic in nature, such as the gradient methods used to find minima in graph theory.

The idea of survival of the fittest is of great importance to genetic algorithms. Genetic algorithms use a fitness function in order to select the fittest string that will be used to create new, and conceivably better, populations of strings. The fitness function takes a string and assigns a relative fitness value to the string. The method by which it does this and the nature of the fitness value does not matter. The only thing that the fitness function must do is to rank the strings in

some way by producing the fitness value. These values are then used to select the fittest strings. The concept of a fitness function is, in fact, a particular instance of a more general AI concept, the objective function

The population can be simply viewed as a collection of interacting creatures. As each generation of creatures comes and goes, the weaker ones tend to die away without producing children, while the stronger mate, combining attributes of both parents, to produce new, and perhaps unique children to continue the cycle. Occasionally, a mutation creeps into one of the creatures, diversifying the population even more. Remember that in nature, a diverse population within a species tends to allow the species to adapt to it's environment with more ease. The same holds true for genetic algorithms.

3.1. Differences Between GAs and Traditional Methods

We will now discuss the basics differences between genetic algorithms and traditional methods.

Genetic algorithms deal with a coded form of the function values (parameter set), rather than with the actual values themselves. So, for example, if we want to find the maximum of a function $f(x_1, x_2)$ of two variables, the GA would not deal directly with x_1 or x_2 values, but with strings that encode these values. For example strings representing the binary values of the variables can be used.

Genetic algorithms use a set, or population, of points to conduct a search, not just a single point on the problem space. This gives GAs the power to search noisy spaces littered with local optimum points. Instead of relying on a single point to search through the space, the GAs looks at many different areas of the problem space at once, and uses all of this information to guide it.

Genetic algorithms use only payoff information to guide themselves through the problem space. Many search techniques need a variety of information to guide themselves. Hill climbing methods require derivatives, for example. The only information a GA needs is some measure of fitness about a point in the space (sometimes known as an objective function value). Once the GA knows the current measure of "goodness" about a point, it can use this to continue searching for

the optimum.

GAs are probabilistic in nature, not deterministic. This is a direct result of the randomization techniques used by GAs.

GAs are inherently parallel. Here lies one of the most powerful features of genetic algorithms. GAs, by their nature, are very parallel, dealing with a large number of points (strings) simultaneously. Holland has estimated that a GA processing n strings at each generation, the GA in reality processes n^3 useful substrings. This becomes clearer later on when schemata are discussed.

	Genetic Algorithms	Traditional Optimization Methods
Work with:	coding of parameter set	parameters directly
Search:	a population of points	a single point
Use info:	payoff (objective function)	payoff + derivatives, etc.
Rules:	probabilistic - random population with random mating, cross-over and mutation	fully deterministic

3.2. Basic Genetic Algorithm Operations

We will now discuss the inner workings of a GA and consider operations that are basic to practically all GAs. With GAs having such a solid basis in genetics and evolutionary biological systems, one might think that the inner workings of a GA would be very complex. In fact, the opposite is true. Simple GAs are based on simple string copying and substring concatenation, nothing more, nothing less. Even more complex versions of GAs still use these two ideas as the core of their search engine. All this will become clear when we walk through a simple GA optimization problem.

There are three basic operators found in every genetic algorithm: reproduction, crossover and mutation. There are some optimization algorithms that do not employ the crossover operator. These algorithms will be referred to as evolutionary algorithms rather than genetic algorithms.

- **Reproduction**

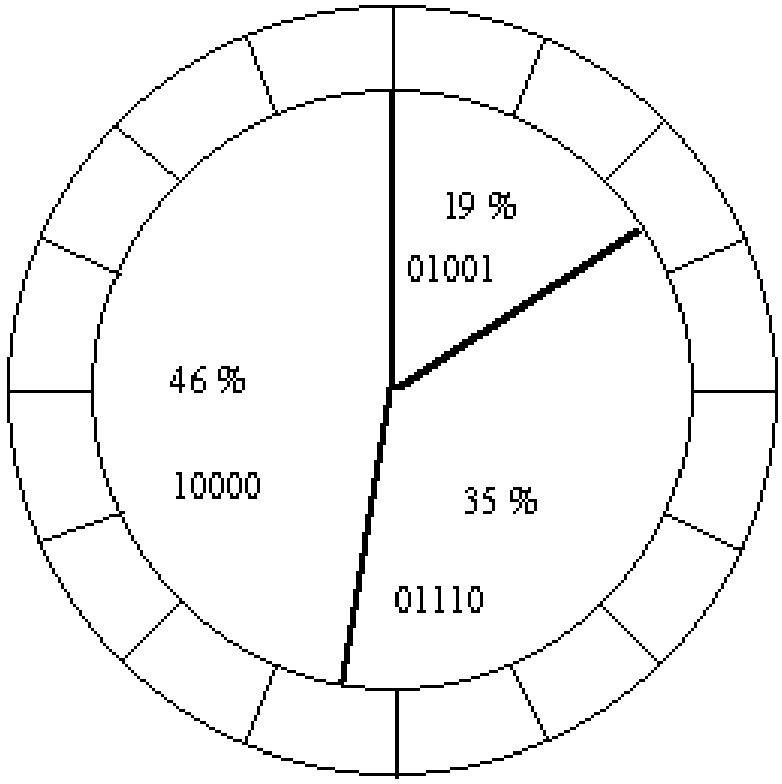
The reproduction operator allows individual strings to be copied for possible inclusion in the next generation. The chance that a string will be copied is based on the string's fitness value, calculated from a fitness function. For each generation, the reproduction operator chooses strings that are placed into a mating pool, which is used as the basis for creating the next generation.

String	Fitness Value	Fitness Percentage
01001	5	$5/26 \approx 19\%$
10000	12	$12/26 \approx 46\%$
01110	9	$9/26 \approx 35\%$
	total = 26	

There are many different types of reproduction operators. One always selects the fittest and discards the worst, statistically selecting the rest of the mating pool from the remainder of the population. There are hundreds of variants of this scheme. None are right or wrong. In fact, some will perform better than others depending on the problem domain being explored. For a detailed, mathematical com-

parison of reproduction/selection strategies for genetic algorithms, see the work of Blickle.

For the moment, we shall look at the most commonly used reproduction method in GAs. The Roulette Wheel Method simply chooses the strings in a statistical fashion based solely upon their relative (ie. percentage) fitness values. This method can be implemented for example by using a roulette wheel. The roulette wheel is nothing but an implementation of the inverse function method for generating discrete random deviates.



Roulette Wheel Selection

When selecting the three strings that will be placed in the mating pool, the roulette wheel is spun three times, with the results indicating the string to be placed in the pool. It is obvious from the above wheel that there's a good chance that string 10000 will be selected more than once. This is fine. Multiple copies of the same string can exist in

the mating pool. This is even desirable, since the stronger strings will begin to dominate, eradicating the weaker ones from the population. There are difficulties with this, as it can lead to premature convergence on a local optimum.

- **Crossover**

Once the mating pool is created, the next operator in the GA's arsenal comes into play. Remember that crossover in biological terms refers to the blending of chromosomes from the parents to produce new chromosomes for the offspring. The analogy carries over to crossover in GAs.

The GA selects two strings at random from the mating pool. The strings selected may be different or identical, it does not matter. The GA then calculates whether crossover should take place using a parameter called the crossover probability. This is simply a probability value p and is calculated by flipping a weighted coin. The value of p is set by the user, and the suggested value is p=0.6, although this value can be domain dependant.

If the GA decides not to perform crossover, the two selected strings

are simply copied to the new population (they are not deleted from the mating pool. They may be used multiple times during crossover). If crossover does take place, then a random splicing point is chosen in a string, the two strings are spliced and the spliced regions are mixed to create two (potentially) new strings. These child strings are then placed in the new population.

As an example, suppose that the strings 10000 and 01110 are selected for crossover and the GA decides to mate them. The GA then randomly selects a splicing point, say 3. Then the following crossover will then occur:

$$\begin{array}{ccc} 100|00 & & 10010 \\ & \Rightarrow & \\ 011|10 & & 01100 \end{array}$$

The newly created strings are 10010 and 01100.

Crossover is performed until the new population is created. Then the cycle starts again with selection. This iterative process continues

until any user specified criteria are met (for example, fifty generations, or a string is found to have a fitness exceeding a certain threshold).

- **Mutation**

Selection and crossover alone can obviously generate a staggering amount of differing strings. However, depending on the initial population chosen, there may not be enough variety of strings to ensure the GA sees the entire problem space. Or the GA may find itself converging on strings that are not quite close to the optimum it seeks due to a bad initial population.

Some of these problems are overcome by introducing a mutation operator into the GA. The GA has a mutation probability, m, which dictates the frequency at which mutation occurs. Mutation can be performed either during selection or crossover (though crossover is more usual). For each string element in each string in the mating pool, the GA checks to see if it should perform a mutation. If it should, it randomly changes the element value to a new one. In our binary strings, 1s are changed to 0s and 0s to 1s. For example, the GA decides to mutate bit position 4 in the string 10000:

$$10000 \Rightarrow 10010$$

The resulting string is 10010 as the fourth bit in the string is flipped. The mutation probability should be kept very low (usually about 0.001%) as a high mutation rate will destroy fit strings and degenerate the GA algorithm into a random walk, with all the associated problems.

But mutation will help prevent the population from stagnating, adding "fresh blood", as it were, to a population. Remember that much of the power of a GA comes from the fact that it contains a rich set of strings of great diversity. Mutation helps to maintain that diversity throughout the GA's iterations.

3.3. GA Example: Optimization Problem

We illustrate how GA can be applied to solve an optimization problem. Specifically we want to find the integer x in the interval $[0, 31]$ that maximizes the function $f(x) = x^2$. Clearly the answer is given by $x = 31$. Our goal here is to see how GA can be used to find this

answer.

The first thing we need to do is to convert the problem in such a way that we can apply GA ideas. In the context of GA, it is obvious that the fitness function is $f(x) = x^2$, and the parameter of interest is x.

The least obvious decision is how parameter x should be coded. There are clearly many ways to accomplish that. In this example we will code x by its binary representation. Since the maximum value of x is 31, we code x using a finite string of 5 bits.

The next choice is the size of the population. We choose a very small population of only 4 so that we can go through by hand one generation to see how GA actually works. We randomly choose 4 strings, each consisting of a random sequence of 5 bits, to represent the genes of the 4 individuals in that population.

For example, we obtain the strings as shown in the first column of the following table. Their corresponding value of x (their phenotypes) are given by the decimal values of the strings, as shown in column 2. Their fitnesses as determined by the fitness function are shown in column 3. The total fitness of the population is 1170, with a

maximum fitness of 576 and an average fitness of 293. The fitness of each individual as a fraction of the total fitness is displayed in column 4. Notice that individual 2 is the fittest and individual 3 is the least fit.

Individual	String	x	Fitness $f(x)$	Relative Fitness
1	01101	13	169	0.144
2	11000	24	576	0.492
3	01000	8	64	0.055
4	10011	19	361	0.309
			total = 1170	
			avg. = 293	
			max. = 576	

To mimic the process of reproduction, each string is copied with a probability given by its relative fitness to the next generation. Suppose we get 1 copy of individual 1, 2 copies of individual 2, 0 copy of individual 3, and 1 copy of individual 4. These strings then go into

the mating pool.

01101

11000

11000

10011

Next these strings are paired up randomly and their genetic material is switched at a randomly chosen site. For example, we pair up string 1 and 2 and pick a cross-over point at 4. String 3 and 4 are paired up and their strings are swapped at site 2. The genes of the off-springs are then given by the following strings.

$$
\begin{array}{ll}
011|01 & 01100 \\
\Rightarrow & \\
110|00 & 11001 \\
11|000 & 11011 \\
\Rightarrow & \\
10|011 & 10000
\end{array}
$$

Suppose we use a mutation rate of 0.001 and perform bit changes on a bit-by-bit fashion. We have a total of 20 bits here, and therefore the average number of bit changes per generation is $20 \times 0.001 = 0.02$. We assume there is no mutation for this generation.

Therefore the new generation has the following genetic makeup.

Individual	String	x	Fitness $f(x)$
1	01100	12	144
2	11001	25	625
3	11011	27	729
4	10000	16	256
			total = 1754
			avg. = 439
			max. = 729

Notice that the fitness of the fittest individual increases from 576 to 729. Furthermore the average fitness of the entire population increases from 293 to 429. Also notice that the best string of the first generation got 2 copies due to its fitness. The first copy $1100|0- > 1100|1$ increases its fitness, and the second copy $11|000- > 11|011$ does even better.

There are many different ways in which one can carry out the cross-over process. However there is one problem with the cross-over process that we have just considered. With an initial population of the following strings, we will never be able to find the maximum of

the function. Why?

01001

11000

11000

10011

The most optimal solution is given by $x = 31$, which has a binary representation of 11111. However if we use the above cross-over procedure then we can never obtain a 1 at the third position of the string. In this case we will have to rely on mutation to create a 1 at that position.

References

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

[2] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery *Numerical Recipes*, Second Edition, Ch. 10. (Cambridge University Press, 1992). 3

[3] Richard J. Bauer, *Genetic Algorithms and Investment Strategies*, 1993.

[4] Lance Chambers, *Practical Handbook of Genetic Algorithms*, 1995.

[5] D.B. McGarrah and R. S. Judson, *Analysis of the Genetic Algorithm Method of Molecular Conformation Determination*, 1993.

[6] C. Calwell and V. S. Johnson, *Tracking a Criminal Suspect Through 'Faec Space' with a Genetic Algorithm*, 1991.

[7] A. Horner and David E. Goldberg, *Genetic Algorithms and Computer-Assisted Music Composition*, 1991.

[8] David, B. Fogel, *Evolutionary Computation - Toward a New Philosophy of Machine Intelligence*, 1995.

[9] T. Blickle and L. Thiele *A Comparison of Selection Schemes used in Genetic algorithms*, TIK Report Nr. 11, December, Version 2. 1995.