Stochastic search and optimization

Mathematical techniques of search and optimization are aimed at providing a formal means for making the best decisions in many real situations. Given the difficulties in many real-world problems and the inherent uncertainty in information that may be available for carrying out the task, stochastic search and optimization methods have been playing a rapidly growing role.

DEFINITION: Stochastic optimization methods are optimization algorithms which incorporate probabilistic elements, either in the problem data (the objective function, the constraints, etc.), or in the algorithm itself (through random parameter values, random choices, etc.), or in both.

The concept contrasts with the deterministic optimization methods, where the values of the objective function are assumed to be exact, and the computation is completely determined by the values sampled so far.

The two main problems of interest are:

- Find the value(s) of a vector $\mathbf{x} \in S$ that minimize a scalar-valued loss function $L(\mathbf{x})$
- Find the value(s) of a vector $\mathbf{x} \in S$ that solve the equation $g(\mathbf{x})=0$ for some vector-valued function $g(\mathbf{x})$

Formalization of the objectives

The vector $\mathbf{x}$ represents a collection of "choices" that one is aiming to pick in the best way.

The loss function $L(\mathbf{x})$ is a scalar measure that summarizes the performance of the system for a given value of these choices. Other common names for the loss function are performance measure, objective function, fitness function, or criterion.

The domain S reflects allowable values or the constraints on the elements of $\mathbf{x}$. It can be continuous, discrete or both (continuous for some coordinate of $\mathbf{x}$ and discrete for the others)

While the first problem above refers to minimizing a loss function, a maximization problem (e.g., maximizing profit) obviously can be trivially converted to a minimization problem by changing the sign of the criterion.

The root-finding function $g(\mathbf{x})$ (which is generally a vector) often arises via calculating the gradient (derivative) of the loss function (i.e., $g(\mathbf{x}) = \nabla L(\mathbf{x})$).

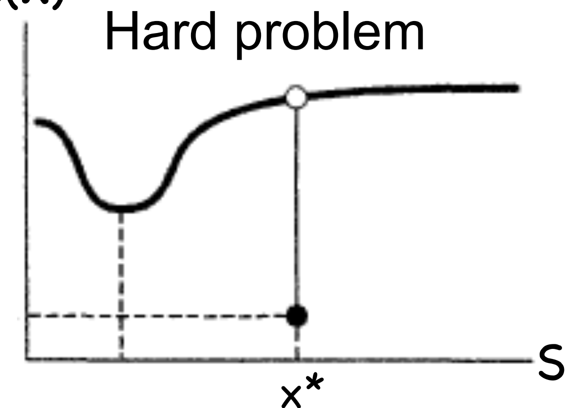
Conversely, the second problem can be converted directly into an optimization problem by noting that a $\mathbf{x}$ such that $g(\mathbf{x})=0$ is equivalent to a x such that $||g(\mathbf{x})||$ is minimized for any vector norm $||\bullet||$

Finding a global minimum?

One of the major distinctions in optimization is between global and local optimization. All other factors being equal, one would always want a globally optimal solution to the optimization problem.

In practice, however, a global solution may not be available and one must be satisfied with obtaining a local solution.

For example, $L(\mathbf{x})$ may be shaped such that there is a clearly defined minimum point over a broad region of the domain S, while there is a very narrow spike at a distant point…



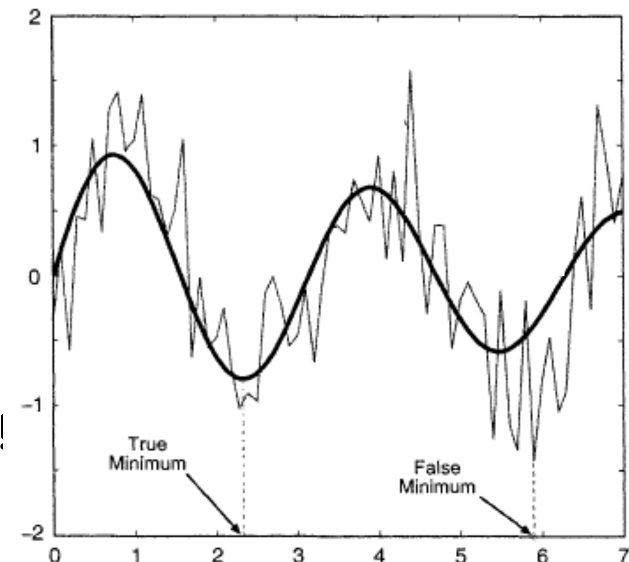
This example illustrates that, in general, one cannot be guaranteed of ever obtaining a global solution. Without prior knowledge about the Possible existence of such a point, no typical algorithm would be able to find this solution because there is nothing near $\mathbf{x}^*$ to indicate the presence of a minimum. Because of the inherent limitations of the vast majority of optimization algorithms, it is usually only possible to ensure that an algorithm will approach a local minimum with a finite amount of resources being put into the optimization process.

Stochastic data or procedures

However, since the local minimum may still yield a significantly improved solution (relative to no formal optimization process at all), the local minimum may be a fully acceptable solution for the resources available (human time, money, computer time, etc.) to be spent on the optimization. However, we will also consider some algorithms (random search, simulated annealing, genetic algorithms, etc.) that are sometimes able to find global solutions among multiple local solutions.

As quoted in the first slide, stochastic search and optimization applies where there is random noise in the measurement of L(**x**) or g(**x**) and/or there is a random choice made in the search direction as the algorithm iterates toward a solution.

Partly-random input data arise in situation like simulation-based optimization where simulations are run as estimates of an actual system, and problems where there is experimental error in the measurements of the criterion. In the presence of noise the search for a global minimum could have no meaning at all!

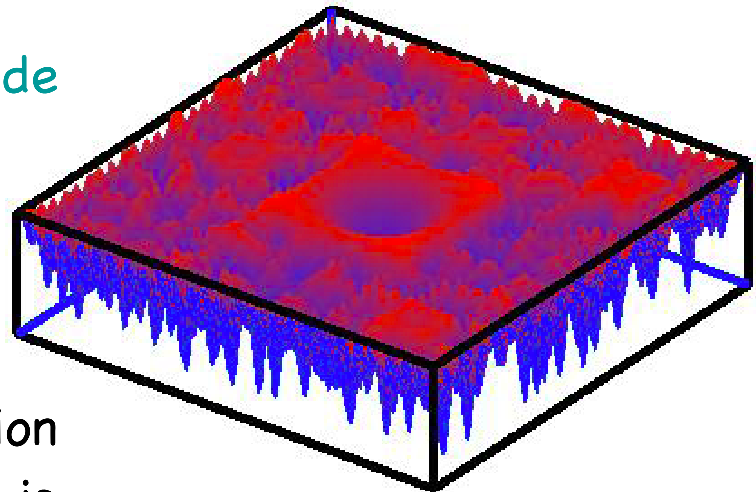


Stochastic data or procedures

In such cases, knowledge that the function values are contaminated by random "noise" leads naturally to algorithms that use statistical inference tools to estimate the "true" values of the function and/or make statistically optimal decisions about the next steps.

On the other hand, even when the data is exact, it is sometimes beneficial to deliberately introduce randomness into the search process as a means of speeding convergence and making the algorithm less sensitive to modelling errors.

Further, the injected randomness may provide the necessary impetus to move away from a local solution when searching for a global optimum. Think about a multidimensional space S full of local minima; any kind of deterministic optimization procedure would be unable to explore all this complex “landscape”.



Indeed, this randomization principle is known to be a simple and effective way to obtain algorithms with almost certain good performance uniformly across all data sets, for all sorts of problems.

The traveling salesman problem

Before presenting a first example, let us provide some general background on the traveling salesman problem: A traveling salesman must visit every city in some territory once and only once. The problem is to find the minimum path.

Because there is a finite number of possible paths, this is a discrete optimization problem, although one with a potentially very large number of elements in S. A tour with $n \geq 3$ cities has $(n-1)!/2$ possible unique tours, corresponding to the number of elements in S (Unique here refers to fundamentally different ordering. For example, if $n = 3$, the tour 1-2-3-1 is considered equivalent to 1-3-2-1 by the symmetry of the cost between cities, where the indicated numbers 1, 2, or 3 represent the labels for the three cities).

For example, finding an optimal tour of the largest cities in all 50 states of the United States entails 3×10^{62} possible routes, far more than the estimated number of atoms in the Earth ($\approx 10^{50}$).

In the language of combinatorial optimization, the problem is NP-complete ("NP" variously stands for non-polynomial)

NP-complete problems

NP-complete problems are computational problems that are all equally difficult because we do not know how to build algorithms that can solve them in *polynomial-time*; they are said to be *intractable problems*. The only algorithms we know for them need an *exponential-time*.

This means that this problem belongs to a class of problems, whose computation time for an exact solution increases with N as

exp(const.×N), becoming rapidly prohibitive in cost as N increases.

The traveling salesman problem also belongs to a class of minimization problems for which the objective function L has many local minima.

In practical cases, it is often enough to be able to choose from these a minimum which, even if not absolute, cannot be significantly improved upon. If one absolutely wants to get *the* global optimum for a large NP-complete problem, the only way is to let the computer run for several hundred years …

The stochastic way is therefore a pragmatic one. For example, for practical purposes, simulated annealing has effectively "solved" the traveling salesman problem. This method has also been used successfully for designing complex integrated circuits, another combinatorial optimization problem.

A first algorithm: Random search

Random search is what it says it is. In essence, it simply consists in picking up random potential solutions and evaluating them. The best solution over a number of samples is the result of random search.

Many people do not realize that a stochastic algorithm is nothing else than a random search, with hints by a chosen heuristics to guide the next potential solution to evaluate.

People who realize this feel uneasy about stochastic algorithms, because there is not guarantee that such an algorithm (based on random choices) will *always* find the global optimum.

The only answer to this problem is a probabilistic one: For example, if, for a particular problem, one already knows the best solution, and if, over a significative number of runs, the proposed stochastic algorithm finds a solution that in average is 99% as good as the known optimum for the tested instances of the problem, then, one can hope that on a new instance of the problem for which the solution is not known, the solution found by the stochastic algorithm will be 99% as good as the unknown optimum over a significative number of runs.

This claim is not very strong, but, as we have seen, there are not many other options available…

No free lunch theorem

In general there is a competition between algorithm efficiency and algorithm robustness (reliability and stability in a broad range of problems). In essence, algorithms that are designed to be very efficient on one type of problem tend to be not reliably transferred to problems of a different type. Hence, there can never be a universally best search algorithm. One must consider the characteristics of the problem together with the goals of the search and the resources available (computing power, human analysis time, etc.) in choosing an approach.

A very important theorem is that of the *No Free Lunch* (Wolpert & Macready, 1997). This theorem states that no search algorithm is better than a random search on the space of all possible problems. In other words, if a particular algorithm does better than a random search on a particular type of problem, it will not perform as well on another type of problem, so that all in all, its global performance on the space of all possible problems is equivalent to a random search.

While the NFL theorem is established for discrete optimization with a finite (but arbitrarily large) number of options, their applicability includes most practical continuous problems because virtually all optimization is carried out on 32- or 64-bit digital computers.

…we have to pay!

The theorem apply to the cases of both noise-free and noisy loss measurements. Because the NFL theorems seem to paint a discouraging picture that "all algorithms work the same," one might wonder about the value of studying various stochastic algorithms. A lecture devoted to blind search alone would be a short lecture indeed!

If a problem has some known structure (and all conceivable practical problems do) and the algorithm uses that structure, it is certainly possible that one algorithm will work better than another on the given problem.

The overall implication is very interesting, as it means that an off the shelf stochastic optimizer cannot be expected to give good results on any kind of problem (no free lunch); a stochastic optimizer is not a black box: to perform well, such algorithms must be expertly adapted for each specific problem.

For this reason I like very much Genetic Algorithms, because they consist of abstract procedures, mimicking natural evolution, which can be easily modelled to a particular optimization problem.

Simulated annealing

The computational challenge in stochastic optimization methods depends strongly on the number of degrees of freedom and the complexity of the "landscape" $L(\mathbf{x})$ on S.

The latter depends on the total number of low-lying "metastable states" (local minima), the ability to efficiently explore the configuration space S and the average height of transition states that separate low-lying metastable states.

The fundamental challenge in stochastic optimization is to balance the number of downhill moves of the dynamical process against the number of uphill moves. In high-dimensional problems the number of metastable states often grows exponentially with the system size. The simplest stochastic optimization method, repeated local optimization (which moved only downhill) starting from random initial conditions, will therefore also require an exponentially large number of steps.

To significantly reduce the computational effort, stochastic optimization methods must therefore also move uphill. In simulated annealing this challenge is met by simulating the "finite temperature dynamics of the system".

Starting from a configuration **x** with "energy" L(**x**) one generates a new configuration **x'** with energy L(**x'**) which replaces the original configuration with probability:

$$P = \begin{cases} e^{-\beta(L(\vec{x}')-L(\vec{x}))} & if\ L(\vec{x}') > L(\vec{x}) \\ 1 & otherwise \end{cases}$$

where β is the fictitious inverse temperature.

At any given temperature such an (ergodic) Monte-Carlo process samples the configurations according to their thermodynamic probability. Thus, at high temperature moves with or against the gradient are accepted with almost equal probability. At low temperature only downhill moves are accepted.

In simulated annealing one thus starts with high temperature simulation and gradually cools the system to zero temperature. If ergodicity is not lost during the cooling schedule, the simulation will stop in the global minimum of the energy landscape.

For locally smooth energy landscapes the search is greatly improved by locally minimizing the new configuration after its generation (basin hopping technique).

In many rugged energy landscapes simulated annealing suffers from the so-called freezing problem. The ruggedness depends on the ratio of the energy difference of adjacent local minima to the height of the intervening transition state.

Parallel tempering

The parallel (or simulated) tempering technique was introduced to overcome difficulties in the evaluation of thermodynamic observables for models with very rugged potential energy surfaces (PES) and applied previously in several protein folding studies.

As an optimization procedure, it resembles the simulated annealing technique.

Low temperature simulations on rugged potential energy surfaces are trapped for long times in similar metastable configurations because the energy barriers to structurally potentially competing different conformations are very high.

In parallel tempering we consider n systems. In each of these systems we perform a simulation in the canonical (NVT) ensemble, but each system is in a different thermodynamic state. Usually, but not necessarily, these states differ in temperature.

Systems with a sufficiently high temperature pass over the potential. The low-temperature systems, on the other hand, mainly probe the local free energy minima. The idea of parallel tempering is to include MC trial moves that attempt to "swap" systems that belong to different thermodynamic states, e.g., to swap a high-temperature system with a low temperature system.

If the temperature difference between the two systems is very large, such a swap has a very low probability of being accepted (see below).

The solution to this problem is to use many small steps: in parallel tempering we use intermediate temperatures, i.e. instead of making attempts to swap between a low and a high temperature, we swap between ensembles with a small temperature difference.

Let us call the temperature of system i, $T_i = 1/\beta_i k_B$, and the systems are numbered according to an increasing temperature scale, $T_1 < T_2 < \ldots < T_n$, . We define an extended ensemble that is the combination of all n subsystems.

The partition function of this extended ensemble is the product of all individual NVT_i ensembles:

$$Z_{PT} = \prod_{i=1}^{n} Z_{NVT_i} = \prod_{i=1}^{n} \frac{1}{\Lambda_i^{3N} N!} \int d\vec{x}_i \, e^{-\beta_i L(\vec{x}_i)}$$

To sample this extended ensemble it is in principle sufficient to perform NVT_i simulations of all individual ensembles. But we can also introduce a Monte Carlo move, which consists of a swap between two ensembles. The acceptance rule of a swap between ensembles i and j follows from the condition of detailed balance.

It turns out to be (it should be accepted both for i and j):

$$P = \min\left(1, e^{-\{(\beta_j - \beta_i)[L(\vec{x}_i) - L(\vec{x}_j)]\}}\right)$$

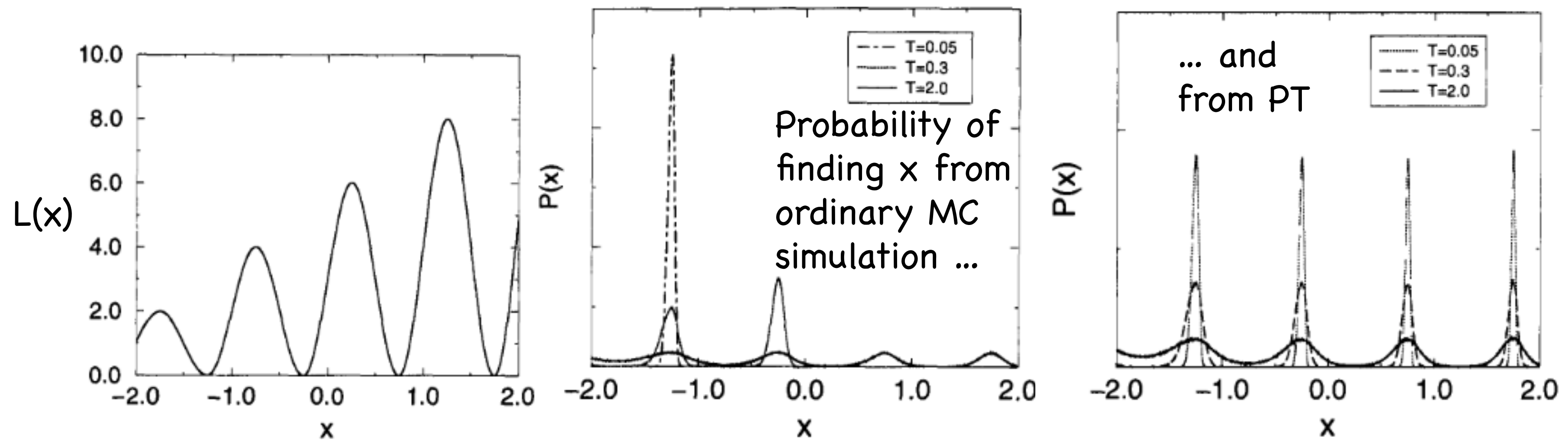
where β_i and $L(\mathbf{x}_i)$ are the inverse temperatures and energy of the ith replica

It is important to note that, as we know the total "energy" of a configuration anyway, these swap moves are very inexpensive since they do not involve additional calculations.

It should be stressed that the swap moves do not disturb the Boltzmann distribution corresponding to a particular ensemble. Therefore one can determine ensemble averages from every individual ensemble just as we do for an ordinary Monte Carlo simulations. This is an important improvement over simulating annealing, since in simulating annealing ensemble averages are not defined. Parallel tempering is a true equilibrium Monte Carlo scheme: the choice of the exchange probability ensures that all simulations remain in thermodynamic equilibrium at their respective temperatures.

Without loss of generality one may confine the exchange mechanism to simulations which are adjacent in temperature.

The exchange mechanism improves the configurational averaging of the low-temperature simulations, because the exchange with high-temperature simulations provides a mechanism to overcome the high energy barriers between low-lying metastable configurations.



The temperature scale for the highest and lowest temperatures is determined by the requirement to efficiently explore the configurational space and to accurately resolve local minima, respectively.

Applied as an optimization technique, however the simulation associated with the lowest temperature will typically yield the estimate for the global optimum, while all others are required to generate different configurations.

The computational effort of the method rises linearly with the number of temperatures.

Introduction to Genetic Algorithms

Genetic algorithms (GAs) were invented by John Holland in the 1960s and were developed by Holland and his students and colleagues at the University of Michigan in the 1960s and the 1970s.

Holland's original goal was not to design algorithms to solve specific problems, but rather to formally study the phenomenon of adaptation as it occurs in nature and to develop ways in which the mechanisms of natural adaptation might be imported into computer systems.

Holland's 1975 book Adaptation in Natural and Artificial Systems presented the genetic algorithm as an abstraction of biological evolution and gave a theoretical framework for adaptation under the GA.

Holland’s GA is a method for moving from one population of "chromosomes" (e.g., strings of ones and zeros, or "bits”) to a new population by using a kind of "natural selection" together with the genetics-inspired operators of crossover, mutation, and maybe others. Each chromosome consists of "genes" (e.g., bits), each gene being an instance of a particular "allele" (e.g., 0 or 1). The selection operator chooses those chromosomes in the population that will be allowed to reproduce, and on average the fitter chromosomes produce more descendants than the less fit ones.

The appeal of evolution

Crossover exchanges subparts of two chromosomes, roughly mimicking biological recombination between two single-chromosome organisms; mutation randomly changes the allele values of some locations in the chromosome.
Holland's introduction of a population-based algorithm with selection crossover and mutation was a major innovation. Moreover, Holland was the first to attempt to put computational evolution on a firm theoretical footing (see Holland 1975). This theoretical foundation, based on the notion of "schemas" was the basis of almost all subsequent theoretical work on genetic algorithms. Why use evolution as an inspiration for solving computational problems? As we have seen, many computational problems require searching through a huge number of possibilities for solutions. Such search problems can often benefit from an effective use of parallelism, in which many different possibilities are explored simultaneously in an efficient way.
What is needed is both computational parallelism and an intelligent strategy for choosing the next set of sequences to evaluate.
Biological evolution is an appealing source of inspiration for addressing these problems. Evolution is, in effect, a method of searching among an enormous number of possibilities for "solutions".

In biology the enormous set of possibilities is the set of possible genetic sequences, and the desired "solutions" are highly fit organisms that are well able to survive and reproduce in their environments.

Evolution can also be seen as a method for designing innovative solutions to complex problems. Seen in this light, the mechanisms of evolution can inspire computational search methods.

Of course the fitness of a biological organism depends on many factors, for example, how well it can adapt to the environment and how well it can compete with or cooperate with the other organisms around it. The fitness criteria continually change as creatures evolve, so evolution is searching a constantly changing set of possibilities.

Furthermore, evolution is a massively parallel search method: rather than work on one species at a time, evolution tests and changes millions of species in parallel. Finally, viewed from a high level the "rules" of evolution are remarkably simple: species evolve by means of random variation (via mutation, recombination, and other operators), followed by natural selection in which the fittest tend to survive and reproduce, thus propagating their genetic material to future generations. Yet these simple rules are thought to be responsible, in large part, for the extraordinary variety and complexity we see in the biosphere.

Elements of Genetic Algorithms

Genetic algorithms (GA) are search algorithms for optimal solutions based on the mechanics of natural selection and natural genetics.

It turns out that there is no rigorous definition of "genetic algorithm" accepted by all in the evolutionary-computation community.

However, it can be said that most methods called "GAs" have at least the following elements in common: populations of chromosomes, selection according to fitness, crossover to produce new offspring, and random mutation of new offspring.

Thus they combine survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm.

In every generation a set of artificial creatures (strings) is created using “information” of the fittest of the old and an occasional new part can be tried.

While randomized, genetic algorithms are no simple random walk. They efficiently exploit historical information to speculate on new search points with expected improved performance. Genetic Algorithms are good at taking large, potentially huge search spaces and navigating them, looking for optimal combinations of things, solutions you might not otherwise find in a lifetime.

Population, Genes and Alleles

Genetic algorithms are different from more normal optimization and search procedures in four ways:

- Gas in general work with a coding of the parameter set, not the parameters themselves
- Gas search from a population of "points", not a single point
- Gas use direct information, not derivatives or other auxiliary knowledge
- Gas use probabilistic transition rules, not deterministic rules

Genetic algorithms require the natural parameter set of the optimization problem to be coded as a finite-length string over some specific (in general finite) alphabet

The population is the ensemble of the chromosomes existing at a given time. Chromosomes (a possible solution) could be:

- Bit string [0101…1100] - Real numbers [43.2 -33.1 … 0.0 89.2]
- Program elements (genetic/evolutionary programming) - …any data structure

Genes: elements of a chromosome

Bit string: the chromosome [1001101] has 7 genes

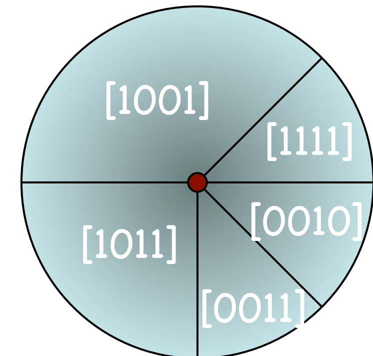
Alleles: possible values of a gene

Bit string: 2 values (0,1)

Operators and definitions

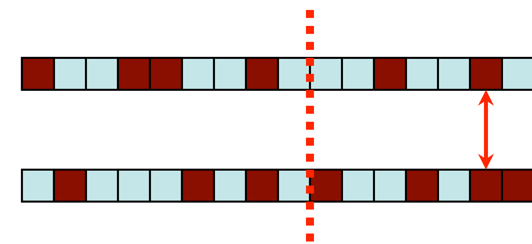
The simplest form of genetic algorithm involves three types of operators: selection, crossover (single point), and mutation.

Selection: this operator selects chromosomes in the population for reproduction. The fitter the chromosome, the more times it is likely to be selected to reproduce (Fitness: the measure of goodness of a solution in an optimization problem).



Fake roulette wheel selection

Crossover: when two individuals have been selected, both parents pass their chromosomes onto their offspring. The two chromosomes come together and swap genetic material. In binary Gas crossover is performed by swapping a part of binary strings between two solutions at a randomly chosen cross-site with some probability.



Mutation: conversion of genes from one to another. In Binary GAs mutation is performed by converting some random bit of a binary string into its complementary bit (i.e. a 1 to a 0 or vice versa) with some probability. Mutation will help prevent the population from stagnating. It adds diversity.

A simple genetic algorithm

Given a clearly defined problem to be solved and a bit string representation for candidate solutions, a simple GA works as follows:

Start with a randomly generated population of n l-bit chromosomes (candidate solutions to a problem).

Calculate the fitness f(x) of each chromosome x in the population.

Repeat the following steps until n offspring have been created:
A. Select a pair of parent chromosomes from the current population, the probability of selection being an increasing function of fitness. Selection is done "with replacement," meaning that the same chromosome can be selected more than once to become a parent.

B. With probability P_c crossover the pair at a randomly chosen point (chosen with uniform probability) to form two offspring. If no

crossover takes place, form two offspring that are exact copies of their respective parents.

C. Mutate the two offspring at each locus with probability P_m, and place the resulting chromosomes in the new population.

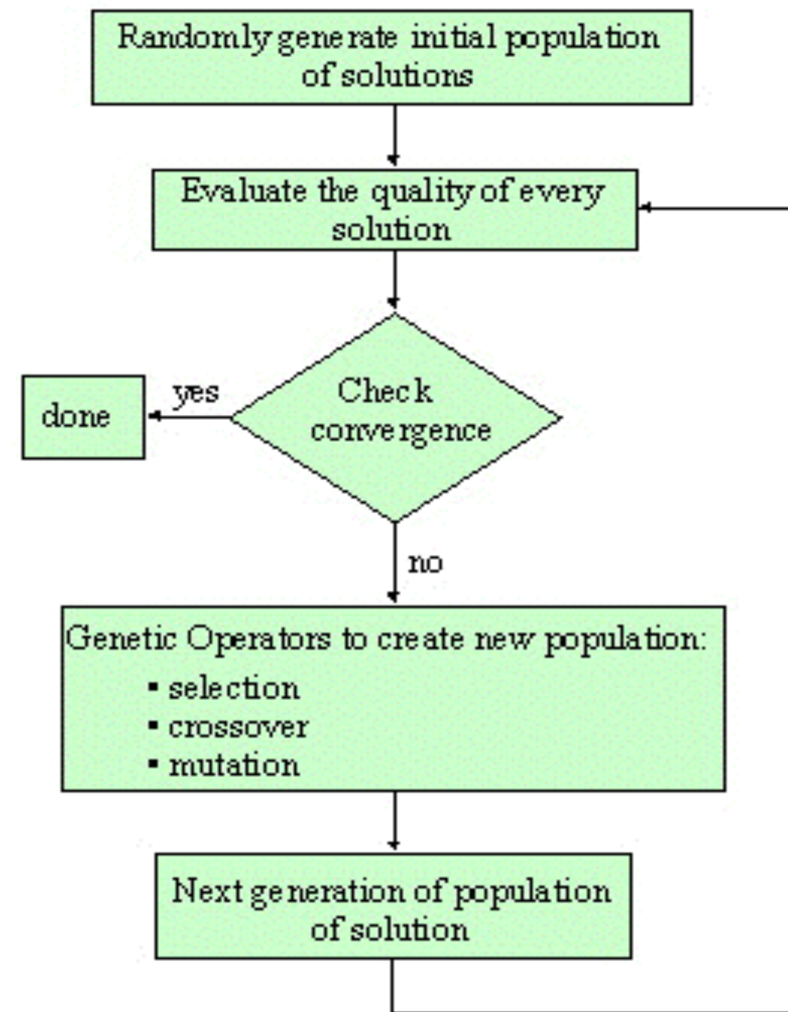
1. Replace the current population with the new population.
2. Repeat: Go to step 2.

Each iteration of this process is called a generation. The entire set of generations is called a run. At the end of a run there are often one or more highly fit chromosomes in the population.

- Since randomness plays a large role in each run, two runs with different random-number seeds will generally produce different detailed behaviours.

GA researchers often report statistics (such as the best fitness found in a run and the generation at which the individual with that best fitness was discovered) averaged over many different runs of the GA on the same problem.

Flow chart



The simple procedure just described is the basis for most applications of GAs. There are a number of details to fill in, such as the size of the population and the probabilities of crossover and mutation, and the success of the algorithm often depends greatly on these details.

For example one can have:

Solutions Generational GA: entire populations replaced with each iteration

Steady-state GA: a few members replaced each generation

Elitism: Some elite (good) solutions are carried onto the next generation without being destroyed.

GAs are especially useful when

The search space is large, complex or the knowledge about it is scarce or it is difficult to encode to narrow the search space.

Traditional search methods fail.

Moreover GAs:

support multi-objective optimization

give always an answer; and this answer gets better with time

are inherently parallel

Question:"If GAs are so smart, why ain't they rich?"

Answer:"Genetic algorithms are rich - rich in application across a large and growing number of disciplines."

– David E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*

The “building block”: schema

The traditional theory of GAs (first formulated in Holland 1975) assumes that, at a very general level of description, GAs work by discovering, emphasizing, and recombining good "building blocks" of solutions in a highly parallel fashion. The idea here is that good solutions tend to be made up of good building blocks—combinations of bit values that confer higher fitness on the strings in which they are present.

Holland (‘75) introduced the notion of schema to formalize the informal notion of "building blocks.”

DEFINITION: a schema is an equivalence class of chromosomes

A schema is a set of bit strings that can be described by a template made up of ones, zeros, and asterisks, the asterisks (*) representing wild cards (or "don't cares").

For example, the schema H=[1****1] represents the set of all 6-bit strings that begin and end with 1. (here I use Goldberg's notation in which H stands for "hyperplane." H is used to denote schemas because schemas define hyperplanes-"planes" of various dimensions in the l-dimensional space of length-l bit strings.)

Instances, order and defining length

The strings that fit this template (e.g., [100111] and [110011]) are said to be instances of H (e.g. [1****1])

DEFINITION: a schemas is of order n iff n is the number of genes different from an asterisks (*); formally n=o(H)

DEFINITION: the defining length of a schemas H is the maximum distance in H between two defined genes; formally d(H)

Thus, in the example above, the schema H=[1****1] is said to have two defined bits (non-asterisks) or, equivalently, to be of order 2. Its defining length (the distance between its outermost defined bits) is 5.

A central belief of traditional GA theory is that schemas are (implicitly) the building blocks that the GA processes effectively under the operators of selection,

mutation, and single-point crossover.

How does the GA process schemas? In the case of a binary alphabet, any given bit string of length l is an instance of 2^l different schemas. For example, the string [11] is an instance of [**] (all four possible bit strings of length 2), [*1] , [1*] , and [11] (the schema that contains only one string, 11).

How do GA work?

Thus, any given population of n strings contains instances of between 2^l (maximal homogeneity) and $n \times 2^l$ (maximal diversity) different schemas. If all the strings are identical, then there are instances of exactly 2^l different schemas; otherwise, the number is less than or equal to $n \times 2^l$.

This means that, at a given generation, while the GA is explicitly evaluating the fitnesses of the n strings in the population, it is actually implicitly estimating the average fitness of a much larger number of schemas, where the average fitness of a schema is defined to be the average fitness of all possible instances of that schema.

For example, in a randomly generated population of n strings, on average half the strings will be instances of [1***...*] and half will be instances of [0***...*]. The evaluations of the approximately n/2 strings that are instances of [1***...*] give an estimate of the average fitness of that schema (this is an estimate because the instances evaluated in typical-size population are only a small sample of all possible instances).

The Schema Theorem

Just as schemas are not explicitly represented or evaluated by the GA, the estimates of schema average fitnesses are not calculated or stored explicitly by the GA. However, as will be seen below, the GA's behaviour, in terms of the increase and decrease in numbers of instances of given schemas in the population, can be described as though it actually were calculating and storing these averages.

We can calculate the approximate dynamics of this increase and decrease in schema instances as follows:

Let H be a schema with at least one instance x_i present in the population at time t, which retains N individuals.

Let $N(H,t)$ ($\leq N$) be the number of instances of H at time t, ...

and let $F(H,t)$ be the observed average fitness of H at time t (i.e., the average fitness of instances of H in the population at time t):

$$F(H,t) = \frac{\sum_{j=1 (x_j \in H)}^{N(H,t)} f(x_j)}{N(H,t)}$$

being f the fitness function.

We want to calculate $E[N(H,t+1)]$, the expected number of instances of H at time t+1.

Assume that the probability for a string x_i (a chromosome) to be selected is equal to

$$P(x_i) = \frac{f(x_i)}{\sum_j f(x_j)}$$

i.e. it is equal to the ratio among the fitness of x_i and the sum of the fitnesses of the population at time t.
Then, assuming x_i is in the population at time t, and x_i is an instance of H, and (for now) ignoring the effects of crossover and mutation, we have that the expected number of instances of H at time t+1 is

$$E[N(H,t+1)] = N \times \sum_{j=1}^{N} P(x_j)\delta_{(x_j \in H)} = N \times \frac{\sum_{j=1(x_j \in H)}^{N(H,t)} f(x_j)}{\sum_{j=1}^{N} f(x_j)} =$$

$$N \times N(H,t) \frac{\frac{1}{N(H,t)} \sum_{j=1(x_j \in H)}^{N(H,t)} f(x_j)}{\sum_{j=1}^{N} f(x_j)} = N(H,t) \frac{F(H,t)}{\frac{1}{N} \sum_{j=1}^{N} f(x_j)} \qquad (*)$$

Thus even though the GA does not calculate F(H,t) explicitly, the increases or decreases of schema instances in the population depend on this quantity: schemas with a greater average fitness will posses a greater number of instances as the generations evolve

By assuming that $F(H,t) = [\Sigma_j f(x_j)/N](1+c) > \Sigma_j f(x_j)/N$ it follows that

$$E[N(H,t+1)] = N(H,t)\frac{\left[\frac{1}{N}\sum_{j=1}^{N} f(x_j)\right](1+c)}{\frac{1}{N}\sum_{j=1}^{N} f(x_j)} = N(H,t)(1+c)$$

Then starting from t=0 and assuming c a constant we obtain:

$$E[N(H,t+1)] = N(H,0)(1+c)^t$$

which is a *geometric progression*, the discrete analogous of the exponential form.

Thus, the selection operator assigns an increasing (decreasing) number of instances to schemas with high (low) idoneity following an exponential law.

Crossover and mutation can both destroy and create instances of H. For now let us include only the destructive effects of crossover and mutation, those that decrease the number of instances of H.

Including these effects, we modify the right side of the previous equation to give a lower bound on $E[m(H,t+1)]$.

Let P_c be the probability that single-point crossover will be applied to a string, and suppose that an instance of schema H is picked to be a parent. Schema H is said to "survive" under single-point crossover if one of the offspring is also an instance of schema H. We can give a lower bound on the probability $S_c(H)$ that H will survive to a single-point crossover:

$$S_c(H) \geq 1 - P_c \times \left(\frac{d(H)}{l - 1} \right)$$

where d(H) is the defining length of H and l is the length of bit strings in the search space. That is, crossovers occurring within the defining length of H can destroy H (i.e., can produce offspring that are not instances of H), so we multiply the fraction of the string that H occupies by the crossover probability to obtain an upper bound on the probability that it will be destroyed. (The value is an upper bound because some Crossovers inside a schema's defined positions will not destroy it, e.g., if two identical strings cross with each other.)
Subtracting this value from 1 gives a lower bound on the probability of survival $S_c(H)$. In short, the probability of survival under crossover is higher for shorter schemas.

The disruptive effects of mutation can be quantified as follows: Let P_m be the probability of any bit being mutated. Then $S_m(H)$, the probability that schema H will survive under mutation of an instance of H, is equal to $(1-P_m)^{o(H)}$, where $o(H)$ is the order of H (i.e., the number of defined bits in H).
That is, for each bit, the probability that the bit will not be mutated is $1-P_m$, so the probability that no defined bits of schema H will be mutated is this quantity multiplied by itself $o(H)$ times. In short, the probability of survival under mutation is higher for lower-order schemas.
These disruptive effects can be used to amend equation (*) :

$$E\left[N(H,t+1)\right] \geq N(H,t)\frac{F(H,t)}{\frac{1}{N}\sum_{j=1}^{N} f(x_j)}\left(1-P_c\frac{d(H)}{l-1}\right)\left(1-P_m\right)^{o(H)}$$

This is known as the Schema Theorem (Holland '75). It describes the growth of a schema from one generation to the next. The Schema Theorem is often interpreted as implying that short, low-order schemas whose average fitness remains above the mean will receive exponentially increasing numbers of samples (i.e., instances evaluated) over time.

The Schema Theorem is a lower bound, since it deals only with the destructive effects of crossover and mutation.
However, crossover is believed to be a major source of the GA's power, with the ability to recombine instances of good schemas to form instances of equally good or better higher-order schemas; this is known as the Building Block Hypothesis (Goldberg '89).
In evaluating a population of n strings, the GA is implicitly estimating the average fitnesses of all schemas that are present in the population, and increasing or decreasing their representation according to the Schema Theorem.
This simultaneous implicit evaluation of large numbers of schemas in a population of n strings is known as implicit paralelism (Holland '75).
The effect of selection is to gradually bias the sampling procedure toward instances of schemas whose fitness is estimated to be above average. The Schema Theorem and the Building Block Hypothesis deal primarily with the roles of selection and crossover in GAs. What is the role of mutation? Holland ('75) proposed that mutation is what prevents the loss of diversity at a given bit position.
In the end note that GAs can be seen as Markov processes!