Global Optimization and Applications

Department of Computational & Applied Mathematics, Wits University,
Johannesburg, Republic of South Africa

Lecturer : **Dr M. Ali**

1 Simulated Annealing (SA)

Simulated annealing is a stochastic method for global optimization which has received much attention over the last two decades. The physical annealing process is known in condensed matter physics as a thermal process for obtaining low energy states of a solid in a heat bath. This has been successfully modeled as a Monte Carlo simulation. In the fifties, Metropolis et al [1] introduced a simple method, known as the Metropolis algorithm, to simulate a collection of atoms at a given temperature.

Statistical mechanics is the central discipline of condensed matter physics where annealing is known as a thermal process for obtaining low energy states of a solid in a heat bath. First the solid is heated until it melts and then it is cooled by slowly lowering the temperature of the heat bath. This is done by the following scheme.

1. Increase the temperature to a value at which the solid melts.
2. Decrease the temperature until the particles form a regular pattern.

In the liquid phase all particle of the solid arrange themselves randomly because of the high energy. But the ground state of the solid, which corresponds to the minimum energy configuration, will have a particular structure, such as seen in a crystal.

Kirkpatrick et al [2] showed how the model for simulating the annealing of solids proposed by Metropolis et al [1] could be used for optimization problems, where the objective function to be minimized corresponds to the energy of the states of the solid. The analogy can be seen in Table 1. The name of the algorithm, therefore, derives from

Table 1: Analogy between Physical Annealing and Global Optimization

initial state at high temperature	random selection of starting point (x)
atomic position	parameters or variables
definition of the molecular structure	definition of a vector
internal energy E	objective function f
average energy $\bar{E}$	mean value of the objective function $\bar{f}$
ground state E_o at low temperature	global optimal (configuration)

an analogy between solving an optimization problem and simulating the annealing of a solid. The Metropolis algorithm can be described briefly as follows:

Starting from an initial state, x, the system is perturbed at random to a new state, y, in the neighbourhood of x and the value of the objective function f_y is calculated. If the change, $\Delta f_{xy} = f_y - f_x$, represents a reduction in the value of the objective function, then the new state is accepted. If the change represents an increase in the objective function value, then the transformation (the new state) is accepted with probability $\exp(-\Delta f_{xy}/T)$, where T is a control parameter corresponding to temperature in the analogy. By repeating this process for a large enough number of iterations Metropolis, et al [1] showed that the equilibrium distribution (Boltzmann Distribution)[1] for a system of particles is approached for a given temperature. Thus the Metropolis procedure from statistical mechanics provides a mechanism which enables a simulated annealing algorithm to avoid becoming trapped in a local minimum in its search for the global minimum. The general SA algorithm is consists of two loops. In the outer loop the temperature is gradually decreased; in the inner loop a number of random moves (number of points in a MC) in the configuration space is proposed and from which some are accepted. The moves are accepted or not according to the Metropolis criterion.

The Basic SA algorithm

```
1. Compute the initial (high) temperature $T_o$ and initial configuration ($x_o$)
2. while stopping criterion is not satisfied do
     begin
     • while no complete MC (Markov Chain) do
     •    begin
     •    generate move ($x$); calculate $f(x)$
     •    if accept then update state ($x$) and $f(x)$
     •    end
     • Decrease $T$
   end
```

The above SA routine can be refined for both discrete and continuous problems. We will discuss here, the simulated annealing algorithm for continuous variables. Simulated annealing type algorithms for continuous variables have been proposed by fewer authors than for the discrete variable case. The convergence proof of their method is based on the equilibrium distribution of Markov chains (a Markov Chain (MC) in the simulated annealing algorithm is a sequence of trials).

[1]The thermal equilibrium in physics is determined by Boltzmann Distribution. This distribution gives the probability of the solid being in state i (there are only a finite number of state in physics) with energy E_i at temperature T, and is given by

$$\Pr\{X = i\} = \frac{1}{Z(T)} \exp\left(\frac{-E_i}{T}\right),$$

where X is a random variable and $Z(T)$ is the partition function and is given by

$$Z(T) = \sum_j \exp\left(\frac{-E_i}{T}\right)$$

The principle complication introduced in going from the discrete to the continuous application of SA lies in the point generation mechanism, g_{xy}, for generating a point y from a point x. Some proposed the following generation mechanism (**GM**):

$$g_{xy} = \begin{cases} LS(x), & \text{if } \omega \geq t_o, \\ \frac{1}{m(S)}, & \text{if } \omega < t_o, \end{cases} \tag{1}$$

where t_o is a fixed number in $(0,1)$, ω a uniform random number in $(0,1)$ and $m(S)$ is the Lebesgue measure of the feasible set S where the minimum of $f(x)$ is sought. Generating y from x by a uniform distribution on S is given by $g_{xy} = 1/m(S)$. $LS(x)$ denotes a local search that generates a point y in a descent direction from x. The local search from x is not a complete local search but only a few steps of some appropriate descent search. Thus $f_y < f_x$ but y is not necessarily a local minimum. The acceptance probability, $A_{xy}(T)$, (i.e., the probability of accepting a point y if x is the current point and y is generated as a possible new point) is known as the Metropolis algorithm, i.e.,

$$A_{xy}(T) = \min\{1, \exp(-(f_y - f_x)/T)\} \ . \tag{2}$$

In a simulated annealing algorithm a complete MC is executed at each temperature.

1.1 The Cooling Schedule

In any implementation of simulated annealing, a cooling schedule must be specified. The temperature parameter, T, is set to an initial value T_o; this is generally relatively high, so that most trials are accepted and there is little chance of the algorithm zooming in on a local minimum in the early stages. A scheme is then required for reducing T and for deciding how many trials are to be attempted at each value of T. Finally a stopping criterion is required to terminate the algorithm. The choice of a cooling schedule clearly has an important bearing on the performance of an SA algorithm. We now briefly summarise the cooling schedule.

Initial value of the temperature

The basic assumption underlying the calculation of the initial value of the control parameter T is that it should be sufficiently large, so that approximately all transitions are accepted at this value. This can be achieved by generating a number of trials, say m_o, and requiring that the initial acceptance ratio $\chi = \chi(T_o)$ is close to one, where $\chi(T)$ is defined as the ratio between the number of accepted transitions and the number of proposed transitions. This initial value of T is then obtained from the following expression

$$T_o = \overline{\Delta f^+} \left(\ln \frac{m_2}{m_2 \chi_o + (1 - \chi_o) m_1} \right)^{-1} \tag{3}$$

where m_1 and m_2 denote the number of trials $(m_1 + m_2 = m_o)$ with $\Delta f_{xy} \leq 0$ and $\Delta f_{xy} > 0$, respectively, and $\overline{\Delta f^+}$ the average value of those Δf_{xy}-values, for which $\Delta f_{xy} > 0$.

Length of the Markov chains

No generally acceptable solution has been presented for this value, which decides how many trials has to be generated at each temperature. The optimal value of this parameter, which has to depend on the problem size, can not be determined in a rigorous way. The chosen value could be

$$L_t = L_o \times n \ , \tag{4}$$

where n is the problem dimension. Note that this choice leads to a chain length which is constant for a given problem instance n.

Decrement of the control parameter

The new value of T_t (t is the temperature counter), say T_{t+1}, is calculated from the following expression

$$T_{t+1} = T_t \left(1 + \frac{T_t \ln(1+\delta)}{3\sigma(T_t)}\right)^{-1} , \qquad (5)$$

where $\sigma(T_t)$ denotes the standard deviation of the values of the cost function at the points in the Markov chain at T_t. The constant δ is known as the distance parameter and determines the rate of decrease of the control parameter.

Stopping Criterion

The last problem to be addressed is that of finding a criterion to terminate the annealing. The stopping criterion is based on the idea that the average function value $\bar{f}(T_t)$ over a MC decrease with T_t, so that $\bar{f}(T_t)$ converges to the optimal solution as $T_t \to 0$. If no changes have occurred in $\bar{f}(T_t)$ in two consecutive Markov Chains the procedure will stop. Therefore a simulated annealing algorithm is terminated if

$$\mid \frac{d\bar{f}_s(T_t)}{dT_t} \frac{T_t}{\bar{f}(T_o)} \mid < \varepsilon_s . \qquad (6)$$

Here $\bar{f}(T_o)$ is the mean value of f at the points in the initial Markov chain, $\bar{f}_s(T_t)$ is the smoothed value of $\bar{f}$ over a number of chains in order to reduce the fluctuations of $\bar{f}(T_t)$, ε_s is small positive number, called the stop parameter.

The cooling schedule described above is a finite-time realization as it implements the MC of finite length at a finite sequence of (decreasing) values of the control parameter T.

1.2 The SA Algorithm For The Continuous Problem

```
begin
    initialize ($T_o, x$);
    stop-criterion:=false;
    while not stop-criterion do
    begin
        for $i$:=1 to $L_t$ do
        begin
            generate $y$ from $x$
            if $f(y) - f(x) \leq 0$
            then accept
            else if $\exp(-(f(y) - f(x))/T_t) >$ random $[0,1)$ then accept;
            if accept then $x := y$;
        end
        lower $T_t$
    end
end.
```

2 Genetic Algorithm

The GA technique was introduced by Holland [6] in 1975 and since then has been applied to a wide range of global optimization problems. Its popularity is due both to its suitability for solving such problems and its ease of implementation. GA methodology is based on analogies to the current theory of biological evolution and hereditary. It mimics natural selection strategies from evolution, embracing the 'survival of the fittest' principle.

The idea behind genetic algorithm (GA) is to do what nature does. Let us take rabbits as an example : at any given time there is a population of rabbits. Some of them are faster and smarter than the other rabbits. These faster and smarter rabbits are less likely to be eaten by foxes, therefore more of them survive to do what rabbits do best : make more rabbits. Of course, some of slower and dumber will survive just because they are lucky. The breeding results in a good mixture of rabbit genetic material : some slow rabbits breed with fast rabbits, some fast with fast, some smart rabbits with dumb rabbits, and so on. The resulting baby rabbits will (on average) be faster and smarter than those in the original population because more faster, smarter parents survived the foxes.

A genetic algorithm follows a step by step procedure that closely matches the story of the rabbits. Genetic algorithm use a vocabulary borrowed from nature genetics. We would talk about individual (genotype) in a population; quite often these individuals are called also strings or chromosomes. Chromosome are made of units–genes (also features or characters)– arranged in linear succession; every gene controls the inheritance of one or several characters. Gene of certain characters are located in certain places of the chromosome, which are called loci (string position). Each genotype (a string or a chromosome) would represent a potential solution to a problem; an evolutionary process run on a population of chromosomes (say, a population consists of N chromosomes) corresponds to a search through a space of potential solutions. Such a search requires balancing two (apparently conflicting) objectives : exploiting the best solutions and exploring the search space.

GA has attracted a lot of interest in recent years. GAs have been quite successfully applied to optimization problems like wire routing, scheduling, adaptive control, game playing, cognitive modeling, transportation problems, travelling salesman problems, optimal control problems, in business, in telecommunication and other problem involving optimization in general. Many variants of GAs have been developed for the purposes of accommodating different applications. The standard GA method which constitutes the basis of these variants can be described as follows:

1. An initial population of parental individuals (feasible solutions) is randomly created. Each individual is represented by a chromosome, a string of characteristic genes.

2. All the individuals are evaluated and ranked with a fitness function appropriate to the problem in hand. The most fit of these passes directly to the following generation; a process referred to as 'elitism'.

3. A breeding population is formed by selecting top-ranking individuals from those that remain. This is the natural selection step.

4. These selected individuals undergo certain transformation *via* genetic operators to reproduce children for the next generation. Operators include recombination by crossover (two chromosomes mate by partly exchanging genes to form two new chromosomes) and mutation (one randomly chosen gene, or more, of a chromosome is altered). Mutation ensures a level of genetic diversity in the population.

The process (from 2 to 4) in repeated until a certain number of generations or some termination criterion is reached. As optimization continues, subsequent generations will consist of increasingly fit or 'superior' individuals. Relatively 'strong' individuals survive and reproduce, while relatively 'week' individuals die.

The evolutionary algorithms or GA as we have seen, imitate the process of evolution. In GA, a point in the feasible reason is represented by a string. Typically binary coding is used, but for continuous optimization problem many prefer the real coding. A good string has high fitness (function value). The algorithm starts with a first generation of strings (initial population). A pair of strings is selected for crossover to produce new offsprings. Those string with higher fitness have a better change of being selected and taking part in the crossover process. The resulting offsprings form part of the new generation, and this process repeats until the deviation within the population is less than a certain tolerance. During this process strings also have a small probability of being mutated. A typical genetic algorithm is as follows:

Step 1. Generate an initial population G of size N.

Step 2. Generate offspring.

- Selection : selecting $m \leq N$ strings as parents, with probability of each string being selected proportional to its fitness.
- Crossover : the parents are paired and generate m offspring, which replaces the m least fittest strings.
- Mutation : each string has a small probability β of being mutated.

Step 3. Repeat Step 2 until the function values of the points $(f(x_i), i = 1, 2, \cdots, N)$ are closed to each other, that is to say the standard deviation is very small.

There are many variants of the above representation of GA. For example, in the crossover phase of step 2, the m offspring may replace their parents, rather than m least fittest strings. During any iteration, say t, a genetic algorithm maintains a population of potential solutions (strings or chromosomes), $G(t) = \{x_1^t, x_2^t, \cdots, x_N^t\}$. Each solution x_i^t is evaluated to give its fitness. Then a new population (iteration $t + 1$) is formed by selecting the more fit individuals. Some members of the new population $G(t + 1)$ again undergo alterations by means of crossover and mutation, to form new solutions in G. **Crossover combines the features of two parent chromosomes to form two similar offspring by swapping corresponding segments of parents**. Mutation arbitrarily alters one or more genes of a selected chromosome, by a random chance with a probability equal to the mutation rate.

By now we clearly understand the fundamental steps of a GA. These are as follows:

1. Reproduction
2. Crossover
3. Mutation

We will now discuss these three steps one by one. Let us first consider the *reproduction*. Reproduction is a process in which individual string are copied according to their objective function values, f (biologist call this function the fitness function). Copying strings according to their fitness values means that strings with higher value (high function value) have a higher probability of contributing one or more offspring in the next generation. This operator, of course, is an artificial version of natural selection, Darwinian survival of the fittest among string creatures. To understand the reproduction let us consider $f(x) = x^2$ with $0 \leq x \leq 31$, and we generate randomly a population of size 4. We construct the following table:

String No.	Initial Population	x Value	$f(x)$	$pselect_i$ $\frac{f_i}{\sum f_i}$	% of Total
1	01101	13	169	0.14	14.4
2	11000	24	576	0.49	49.2
3	01000	8	64	0.06	5.5
4	10011	19	361	0.31	30.9
Total			1170	1	100.0

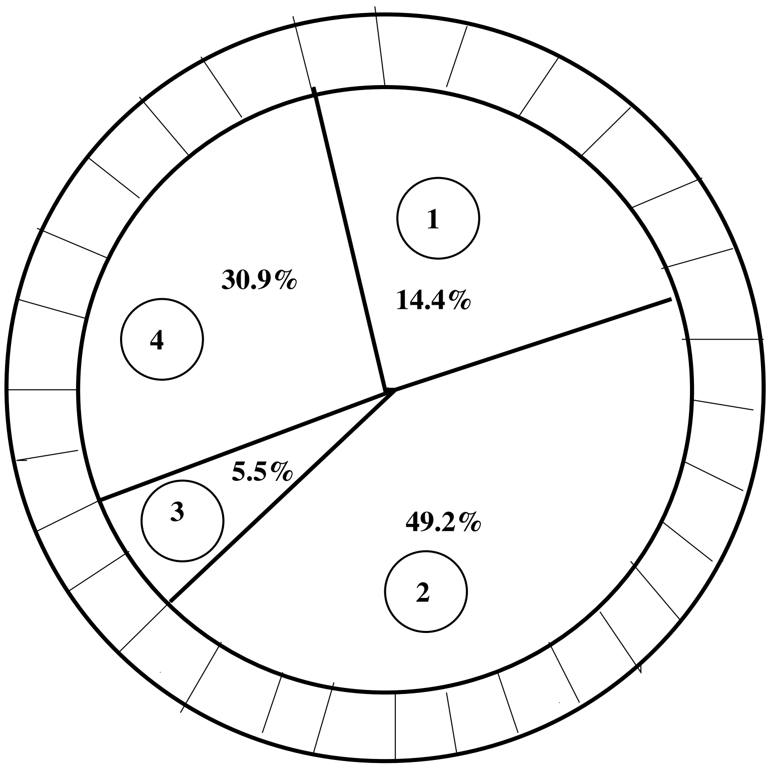

Figure 1: Reproduction by Roulette Wheel

The reproduction is implemented in algorithmic form by a roulette wheel where each current string in the population has a roulette wheel slot sized in proportion to its fitness. The percentage of population total fitness in shown in the above table. The corresponding weighted roulette wheel for this generation's reproduction is shown in Figure 1. To reproduce, we simply spin the weighted roulette wheel four times. For example in this problem, string number 1 has a fitness value of 169, which represent 14.4% of the total fitness. As a result, string 1 is given 14.4% of the roulette wheel, and each spin turns up string 1 with probability 0.144. Therefore, a simple spin of the weighted roulette wheel yields the reproduction candidate. In this way, more highly fit strings have a higher number of offspring in the succeeding generation.

After reproduction of two strings, the pair undergoes crossover as follows : an integer position k along the string is selected uniformly between 1 and the string length (l) less one, i.e., on $[1, l-1]$. Two new strings are created by swapping all characters between positions $k+1$ and l inclusively. For example, consider strings A_1 and A_2 from our example initial population:

$$\begin{aligned} A_1 &= 0\,1\,1\,0 \mid 1 \\ A_2 &= 1\,1\,0\,0 \mid 0 \end{aligned}$$

Suppose in choosing a random number 1 and 4, we obtain a $k = 4$ (as indicated by the separator symbol $|$). The resulting crossover yields two new strings where the prime means the strings (the offsprings) are part of the new generation:

$$\begin{aligned} A_1' &= 0\,1\,1\,0 \mid 0 \\ A_2' &= 1\,1\,0\,0 \mid 1 \end{aligned}$$

The mutation is the occasional (with small probability) random alteration of the value of the string position. In our example, this simply means changing 1 to 0 and vice versa. This operation is performed on a bit-by-bit basis. If we assume the mutation probability to be 0.001 then the mutation works as follows : we consider a single bit and generate a random number, if the random number is less than 0.001 the this bit position is changed (i.e., if it is 0 change it to 1 and vice versa).
Remark : If the function being optimized involves more than one variables, for each variable (i) a substring with length l_i is considered and in the population substring (i) with length l_i are added together to form a chromosome. For example if $P = 00101$ and $Q = 1111011$ (notice the length may differ and indeed this is the case for x_2 being defined by larger bounds) are two substring corresponding to variable x_1 and x_2 then their corresponding (single) chromosome in the population will be $PQ = 00101\ 1111011$.
Decode & Normalization : Decode is the decoded integer value (dvalue) of a binary string. If a variable (say, x_i) is defined in an interval $[L, M]$ then the decoded value is mapped in the interval by the rule

$$x_i = L + (M - L)\frac{dvalue}{2^{l_i} - 1}$$

where l_i is the length of the string.

A genetic algorithm for a particular problem must have the following five components:

- a genetic representation for potential solutions to the problem,
- a way to create an initial population of potential solutions,

- an evolution function that plays the role of the environment, rating solutions in terms of their fitness,
- genetic operators that alter the composition of children,
- values of the various parameters that the genetic algorithm uses (population size, probability of applying genetic operators ect.)

2.1 Optimization of a Simple Function

We now discuss the basic features of a genetic algorithm for optimization of a simple function of one variable, namely

$$f(x) = x \sin(10\pi x) + 1.0$$

The problem is to find x in the range $[-1, 2]$ which maximizes the function f.

2.1.1 Representation

We use a binary vector as a chromosome to represent real values of the variable x. The length of vector depends on the required precision, which, we want to be six places after the decimal point. The domain of the variable x has length 3; the precision requirement implies that the range $[-1, 2]$ should be divided into at least $3 \cdot 1000000$ equal size ranges. This means that 22 bits are required as a binary vector (chromosome):

$$2097152 = 2^{21} < 3000000 \leq 2^{22} = 4194304$$

The mapping of a string into a real number $x \in [-1, 2]$ can be done as follows:

$$x = -1.0 + x' \cdot \frac{3}{2^{22} - 1},$$

where -1.0 is the left limit of x, 3 is the length of the domain and x' is the unsigned binary integer (base 10) converted from any string. For example, a chromosome

$$(1000101110110101000111)$$

represents the number 0.637197, since

$$x' = (1000101110110101000111)_2 = 2288967$$

and

$$x = -1 + 2288967 \cdot \frac{3}{4194303} = 0.637197$$

The chromosome

$$(0000000000000000000000)$$

and

$$(1111111111111111111111)$$

represent the left and right limits of x, i.e., -1.0 and 2.

2.1.2 Initial Population

The initialization process is very simple : we create a population of chromosomes, where each chromosome is a binary vector of 22 bits. All 22 bits for each chromosome are initialized randomly.

2.1.3 Evaluation Function

Evaluation function eval for binary vector v is equivalent to the function f:

$$eval(v) = f(x)$$

where the chromosome v represents the real value x. As noted earlier, the evaluation function plays the role of the environment, rating potential solutions in terms of their fitness. For example, three chromosomes :

$$\begin{aligned} v_1 &= (1000101110110101000111) \\ v_2 &= (0000001110000000010000) \\ v_3 &= (1110000000111111000101) \end{aligned}$$

correspond to values $x_1 = 0.637197, x_2 = -0.958973$, and $x_3 = 1.627888$, respectively. Consequently the evaluation function would rate them as follows:

$$\begin{aligned} eval(v_1) &= f(x_1) = 1.586345, \\ eval(v_2) &= f(x_2) = 0.078878, \\ eval(v_3) &= f(x_3) = 2.250650. \end{aligned}$$

Clearly, the chromosome v_3 is the best of the three chromosomes, since its evaluation returns the highest value.

2.1.4 Genetic Operators

During the alteration phase of the genetic algorithm we would use two classical genetic operators: **mutation and crossover**.

As mentioned earlier, mutation alters one or more genes (positions in a chromosome) with a probability equal to the mutation rate. Assume that the fifth gene from the v_3 chromosome was selected for mutation. Since the fifth gene is 0, it would be flipped into 1. So the chromosome v_3 after the mutation would be

$$v_3' = (1110100000111111000101)$$

This chromosome represents the value $x_3' = 1.721638$ and $f(x_3') = -0.082257$. This means that this particular mutation resulted in a significant decrease of the value of the chromosome v_3. On the other hand if the 10th gene was selected for mutation in the chromosome v_3, then

$$v_3'' = (1110000001111111000101)$$

The corresponding values $x_3'' = 1.630818$ and $f(x_3'') = 2.343555$, an improvement over the original value of $f(x_3) = 2.250650$.

Let us now illustrate the crossover operator on chromosome v_2 and v_3. Assume the crossover point was selected after the 15th gene:

$$v_2 = (00000|01110000000100000)$$
$$v_3 = (11100|00000111111000101)$$

The two new offspring are

$$v_2' = (00000|00000111111000101)$$
$$v_3' = (11100|01110000000100000)$$

These offspring evaluate to:

$$eval(v_2') = f(x_2') = 0.940865,$$
$$eval(v_3') = f(x_3') = 2.459245.$$

2.1.5 Parameters

A set of values that have to be chosen by the user is set of values for mutation rate, population size and the probability of crossover etc.

2.2 Transforming The Objective Function Into Fitness Form

From our little practice of GA, we have noticed that the fitness function ($f(x)$) requires to be a) non-negative and b) in maximization form. In many problem the objective function are given in the minimization form rather than in maximization form. If the function $f(x)$ is naturally stated in the maximization form, even so this does not guarantee that $f(x) \geq 0$, $\forall x \in S$. As a result, it is often necessary to transfer the given function $f(x)$ to a fitness function form by some mapping. In traditional mathematical optimization problem a minimization problem (e.g., Min $f(x)$) is easily transformed to a maximization one (e.g., Max $F(x) = -f(x)$) by simply multiplying the function with a minus one (-1). However, in GA, this operation alone is insufficient because it does not guarantee the non-negativity criterion. In GA practice, for any given minimization function $f(x)$, the following function-to-fitness transformation is commonly used:

$$F(x) = \begin{cases} M_{max} - f(x) & \text{if } f(x) < M_{max} \\ 0 & \text{otherwise} \end{cases}$$

There are many different ways to choose the number M_{max}. It may be chosen as a input coefficient, as the largest f value in the current population, or the largest of the last k generations.

Even if the function $f(x)$ is stated as a maximization function, one may have negative function values ($f(x) < 0$). To overcome this we simply transform the function as given below:

$$F(x) = \begin{cases} M_{min} + f(x) & \text{if } f(x) + M_{min} > 0 \\ 0 & \text{otherwise} \end{cases}$$

We may choose M_{min} as an input coefficient, as the absolute value of the worst $f(x)$ value in the current or last k generations.

3 Controlled Random Search Algorithm (CRS)

In many applications the function f of interest is not differentiable and it is with this view in mind the CRS algorithm was initially developed. CRS is a 'direct search' technique and purely heuristic. It is a kind of contraction process where an initial sample of N points is iteratively contracted by replacing the worst point with a better point. The replacement point is either determined by a global or a local technique. The global technique is an iterative process in which a trial point, the 'new' point, is defined in terms of $n+1$ points selected from the whole current sample of N points until a replacement point is found. Some CRS algorithms also apply a local technique where the replacement point is searched for near a subset of best points in the current sample.

It starts by initially filling a set, called the population set, with a sample of N ($N \gg n$) points uniformly distributed over the search space S. The population set is then gradually contracted by replacing the current worst point in it with a better point, called a trial point. In the original crs1 [8], a trial point is obtained by forming a simplex, using $n+1$ distinct points chosen at random, with replacement, from the population set, and reflecting one of the points in the centroid of the remaining n points of the simplex, as in the Nelder and Mead algorithm [9].

In the original version of CRS of Price, therefore, a simplex is formed from the subset of $n+1$ points. One of the points of the simplex is reflected in the centroid of the remaining points to obtain a new trial point. This process of finding a trial point and replacing the current worst point in the population set, if the trial point is better than the current worst point in the population set, is repeated until a certain stopping condition is met. Below we present the crs1 algorithm.

The CRS1 algorithm

Step 1 **Initialize**. Generate N ($N \gg n$) uniformly distributed random points from the search region S and store the points and their corresponding function values in an array A. Set iteration counter $k = 0$.

Step 2 **Stopping rule based on best and worst points**. Find the best and worst points in the population set, x_l and x_h, where the best point x_l has the lowest function value f_l and the worst point x_h has the highest function value f_h. If the stopping condition (e.g., $f_h - f_l \leq \epsilon$) is achieved, then stop.

Step 3 **Generate a trial point $\tilde{x}$**. Choose randomly $n+1$ points $x_{p(1)}, x_{p(2)}, \cdots, x_{p(n+1)}$, with replacement, from the population set. Compute

$$\tilde{x} = 2G - x_{p(n+1)}, \tag{7}$$

where the centroid G is given by

$$G = \frac{1}{n}\sum_{j=1}^{n} x_{p(j)}. \tag{8}$$

If $\tilde{x} \notin S$ go to Step 3; otherwise compute $f(\tilde{x})$. If $f(\tilde{x}) \geq f_h$ then go to Step 3.

Step 4 **Update** the population set. Replace, in the population set, the point x_h and function value f_h by those of $\tilde{x}$ and $f(\tilde{x})$ respectively. Set $k = k+1$ and go to Step 2.

There are various modifications that have been suggested to the original crs1. The first two, crs2 and crs3, were suggested by Price [10, 11]. In crs2, the point with the least function value in population is always used in forming the simplex while in crs3, a Nelder-Mead-type [9] local search from the best $n+1$ points in population set is incorporated. Ali and Storey [12] modified crs2 by exploring the region around the best point using a β-distribution. This version is known as crs4. In crs4, every time a new best point x_l is found, $f(x)$ is evaluated for M points from a β-distribution using the current best point x_l as the mean and the distance between x_l and x_h as the standard deviation. Ali and Storey [12] also proposed crs5 which uses a few steps of a gradient based local search from the best point x_l instead of a β-distribution. Another modification, crsi, was introduced by Mohan and Shanker [13], who used coordinate-wise quadratic interpolation to find trial points instead of making use of simplexes. Ali et. al. [14] then proposed crs6 which uses the coordinate-wise quadratic interpolation to find trial points and a β-distribution for local exploration around the best point x_l just as in crs4.

The coordinate-wise quadratic interpolation scheme uses $x_1 = x_l$ and two other randomly selected points $\{x_2, x_3\}$ with replacement from population set to determine the coordinates of the trial point $\tilde{x} = (\tilde{x}^1, ..., \tilde{x}^n)$, where

$$\tilde{x}^i = \frac{1}{2}\left[\frac{(x_2^{i^2} - x_3^{i^2})f(x_1) + (x_3^{i^2} - x_1^{i^2})f(x_2) + (x_1^{i^2} - x_2^{i^2})f(x_3)}{(x_2^i - x_3^i)f(x_1) + (x_3^i - x_1^i)f(x_2) + (x_1^i - x_2^i)f(x_3)}\right], \tag{9}$$

for $i = 1, 2, \cdots, n$. If $\tilde{x} \notin S$, the denominator is zero or $f(\tilde{x}) \geq f_h$, then another two random points $\{x_2, x_3\}$ are chosen from population set and a new trial point is found by the above quadratic interpolation using $\{x_1, x_2, x_3\}$. Brachetti et. al. [15] also presented a modification to crs1. The main features of their version, ncrs, are the use of weighted centroid and weighted reflection in generating trial points. ncrs is similar to crs1 except for these features and the use of a quadratic model for local exploration using $2n+1$ best points in the population set.

It is clear from the above crs1 algorithm that the core step, that is, the operation through which the trial points are obtained in the crs algorithms, is Step 3. Therefore, most crs algorithms were derived by modifying this step. In particular, modifications were done by either replacing the point generation scheme in Step 3 and/or introducing a local search technique to obtain trial points locally every time a new best point x_l is obtained by a point generation scheme.

3.1 The CRS2 algorithm

crs2 gradually contracts the population set by replacing the current worst point in the population set with a better trial point through repeated use of simplexes. At each iteration, therefore, the current worst point x_h in the population set is targeted. A trial point is generated by forming a simplex. The simplex is formed using $n+1$ distinct points, $x_{p(1)}, x_{p(2)}, \cdots, x_{p(n+1)}$, where $x_{p(2)}, x_{p(3)}, \cdots, x_{p(n+1)}$ are chosen at random (with replacement) from the population set and the point $x_{p(1)}$ is always the current best point x_l in the population set. The trial point is obtained by reflecting the point $x_{p(n+1)}$ in the centroid of the remaining n points of the simplex, as in the Nelder and Mead algorithm [9]. The trial point $\tilde{x}$ is thus given as

$$\tilde{x} = 2G - x_{p(n+1)}, \tag{10}$$

where the centroid G is given by

$$G = \frac{1}{n} \sum_{j=1}^{n} x_{p(j)}. \tag{11}$$

The trial point generation in the original crs1 [8] is similar to crs2 except that in crs1 the point $x_{p(1)}$ is also randomly chosen. The process of finding a trial point and replacing the current worst point in population set, if the trial point is better than the current worst in the population set, is repeated until a certain stopping condition is met.

Computer implementation the the crs algorithm requires an array A for storing the points and their function values of the population set. A suggested value for N, the number of points in the array A, is $N = 10(n+1)$. It is however a heuristic choice and therefore could always be increased for obtaining the global minima with higher probability. The algorithm stops when all points in the array A are within agreement to a fixed number of decimal places, i.e. $|f_l - f_h| < \epsilon_o$, where ϵ_o is a preset small number. For all numerical calculation we took $\epsilon_o = 10^{-3}$.

3.2 Analysis of the crs2 algorithm

In crs2 the trial point $\tilde{x}$ (read x_{trial}) is given by

$$\tilde{x} = 2G - x_{p(n+1)}. \tag{12}$$

One can rewrite the above equation as

$$\tilde{x} = G + \left(G - x_{p(n+1)}\right). \tag{13}$$

Since the points $x_{p(2)}, \cdots, x_{p(n)}$ are random in S, G is also a random point in S. The position (feasibility) of $\tilde{x}$ is therefore determined by the positions of G and $x_{p(n+1)}$, and the length of the differential vector $G - x_{p(n+1)}$. Here $x_{p(n+1)}$ a randomly selected point from $N - n$ points given that the points $x_{p(1)}, \cdots, x_{p(n)}$ have been selected from N points. Therefore there are $N - n$ possibilities for $x_{p(n+1)}$ resulting in $N - n$ trial point after $x_{p(1)}, \cdots, x_{p(n)}$ have been fixed .

We visualize the location of $\tilde{x}$ using the following graph in two dimension ($n = 2$) and thereby motivate its feasibility. Suppose that we select $x_{p(1)}$ and $x_{p(2)}$ two points from the population set P of N points (in a two dimensional region S where the optimum is sought) and calculate the centroid G. We draw a reference axis through the centroid G. Since $x_{p(3)}$ is a random point from the remaining $N - 2$ points (and the fact that the points are scattered around) the location of $x_{p(3)}$ can be in any one of the four quadrants. For the sake of argument, we select four possible locations of $x_{p(3)}$ in around G. They are numbered as 1, 2, 3 and 4. If $x_{p(3)}$ is selected at location 1 then $\tilde{x}$ will fall in the first quadrant (as shown in the figure). If on the other hand, $x_{p(3)}$ is selected at location 2 then $\tilde{x}$ will fall in the 4th quadrant. Similarly the trial point can lie in the other quadrants. Therefore, there always exists n points $x_{p(2)}, \cdots, x_{p(n+1)}$ ($x_{p(1)} = x_l$) such that $\tilde{x}$ in a iteration will be feasible due to the following arguments:

- If G is feasible (which is always the case) then $\tilde{x}$ will be feasible at least in one of the quadrants.

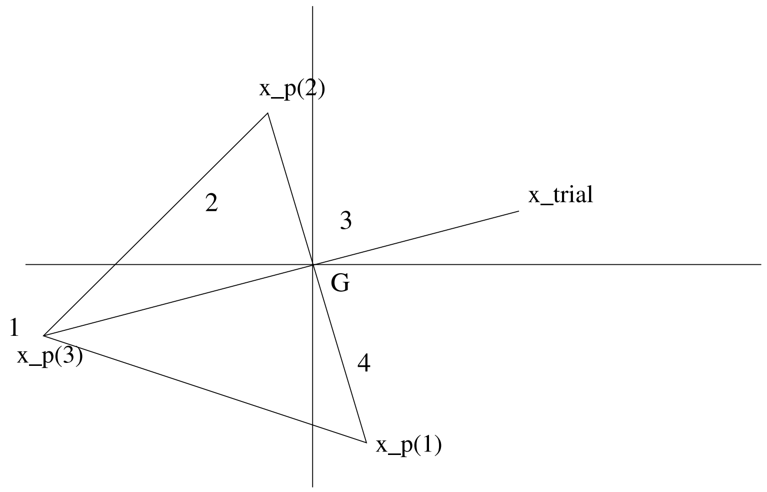

Figure 2: Simplex operation in crs2 in two dimension

- The points $x_{p(2)}, \cdots, x_{p(n)}$ are are selected at random from $N-1$ points (since $x_{p(1)} = x_l$ is fixed). The number of different ways in which these $n-1$ points can be chosen from $N-1$ points is ${}^{N-1}C_{n-1}$. Since $x_{p(n+1)}$ is random in the remaining $N-n$ points, the total number of equiprobable next trial point associated with the N points in the population set is given by $(N-n) \times {}^{N-1}C_{n-1}$. Given that $N \gg n$, probability of generating a feasible trial point will always be non-zero at each iteration.

4 Differential evolution (DE)

A genetic algorithm that is well adopted for a combinatorial (discrete) optimization problem or for problems which have both discrete as well as continuous variables. Although GA has a wide range of applicability, from engineering to commerce and from finance to banking, however while many problems can be solved successfully using GA, its credibility is frequently called into question for problem with continuous variables. Differential evolution (DE) [16] is a powerful yet simple evolutionary algorithm for optimizing real valued, multimodal (many minima) functions. It can be used when the underlying function is non-differentiable, noisy and highly nonlinear. It is therefore a direct search method of exceptionally simple in nature. As we know, guided by cost, GA attempts to transfer an initial population of randomly generated vectors into a solution vector through repeated cycle of mutation, recombination, and selection. DE also uses selection, mutation and recombination but in a different fashion. Experiments have shown that DE is much superior algorithm than GA and CRS.

The overall structure of the DE algorithm resembles that of most other population based searches, eg, GA and CRS. At an initial array the algorithm holds a population of number N, these are n dimensional vectors from the search space, generated randomly. In each generation, N competitions are hold to determine the composition of the next generation. In particular the i-th competition pits the i-th population vector, known as the *target*, against its adversary, the *trial vector*. Not only does the target vector compete against the trial vector, it is also one of the trial vector's parents. The trial vector's other parent is a randomly chosen population vector to which a weighted random difference vector has been added. DE, therefore, generates new parameter vectors by adding the weighted difference between two population vectors to a third vector. Let this operation be called *mutation*. Mating between this noisy random vector and the target vector is controlled by a crossover operator that determines which trial vector parameters (component or variable) are inherited from which parent. In other words, the mutated vector's parameters are then mixed with the parameters of another predetermined vector, the target vector, to yield the so-called trial vector. This parameter (variable) mixing is often referred to as *crossover*. If the trial vector yields a lower cost function (less function value) then the target vector, the trial vector replaces the target vector in the following generation. More specifically DE's basic strategy can be described as follows:

Initialisation : The initial population size of N are generated randomly within the search region, before DE is set to optimize the given function. At each iteration the DE algorithm calculates the i-th trial vector with which to replace the i-th target vector (in the old population). The process is continued until all members of the old population is considered, and at the end a new population is generated. Therefore, an n dimensional parameter vectors

$$x_{i,k}, \quad i = 1, 2, \cdots, N \tag{14}$$

is generated for each generation k. How this is done is given below.

Mutation (with Vector Differentials) : For each target vector $x_{i,k}, \quad i = 1, 2, \cdots, N$, a mutant vector is generated according to

$$v_{i,k+1} = x_{r1,k} + F \cdot (x_{r2,k} - x_{r3,k}) \tag{15}$$

with random indexes $r1, r2, r3 \in \{1, 2, \cdots, N\}$, integer, mutually different and $F > 0$. The randomly chosen integers $r1, r2$ and $r3$ are also chosen to be different from the

running index i, so that N must be greater or equal to to four to allow for this condition. F is a real and constant factor in $[0,2]$ which controls the amplification of the differential variation $(x_{r2,k} - x_{r3,k})$. The scalling factor, F, is a user-supplied constant whose value is found empirically. By mutating vectors with population -derived noise, DE ensures that the solution space will be efficiently searched in each dimension.
Crossover : In order to increase the diversity of the perturbed parameter vectors, crossover is introduced. To this end, the trial vector :

$$u_{i,k+1} = (u_{1i,k+1}, u_{2i,k+1}, \cdots, u_{ni,k+1}) \tag{16}$$

is formed, where

$$u_{ji,k+1} = \begin{cases} v_{ji,k+1} & \text{if } (\text{randb}(j) \leq \text{CR}) \text{ or } j = \text{rnbr}(i) \\ x_{ji,k} & \text{if } (\text{randb}(j) > \text{CR}) \text{ or } j \neq \text{rnbr}(i), \\ j = 1, 2, \cdots, n. & \end{cases} \tag{17}$$

In (17), randb(j) is the j-th uniform random number in $(0,1)$. CR is the crossover constant in $(0,1)$ which has to be determined by the user. rnbr(i) is a randomly chosen index in $\{1,2,\cdots,n\}$. In crossover, which parent contributes which trial vector parameter (variable) is determined by CR. The source of each trial vector parameter is determined by comparing CR to a uniformly distributed random number from within the interval $[0,1]$. If the random number greater than CR, the trial vector gets its parameter from the target vector, otherwise, the parameter comes from mutated vector.

Selection : To decides whether or not it $(u_{i,k+1})$ should become a member of generation $G+1$, the trial vector $u_{i,k+1}$ is compared to the target vector $x_{i,k}$. If the vector $u_{i,k+1}$ yields a smaller function value, then $u_{i,k+1}$ replaces $x_{i,k+1}$; otherwise, the old value $x_{i,k}$ is retained. Therefore, unlike many genetic algorithms, DE does not use proportional selection, ranking, or even an annealing criterion that would allow occasional uphill moves. Instead, the cost of each trial vector is compared to that of its parent target vector. The vector with the lower cost is rewarded by being allowed to go to the next generation.
Choice of the Control Parameters : It is interesting to note that DE's control variables, N, F and CR, are not difficult to choose in order to obtain good results. Numerical experience with suggests that a reasonable choice for N is between $6 \times n$ to $10 \times n$, N must be at least 4 to ensure that DE will have enough mutually different vectors with which to work. As for F, $F = 0.5$ is usually a good initial choice. If the population converges prematurely, then F and/or N should be increased. A good choice for CR is 0.2 but DE works well for CR$\in [0.2, 0.9]$.

The DE Algorithm

Step 1 : Generate a population $x_{i,k}$, $i = 1,2,\cdots,N$, of size N, store them in an array with their function values. Initialize this generation as $k = 0$.

Step 2 : For each target vector $x_{i,k}$, $i = 1,2,\cdots,N$, generate three independent vectors randomly (except $x_{i,k}$) $x_{r1,k}, x_{r2,k}$ and $x_{r3,k}$ from the array of population in generation k. Calculate the following mutated vector:

$$v_{i,k+1} = x_{r1,k} + F \cdot (x_{r2,k} - x_{r3,k})$$

Step 3 : Calculate the trial vector $u_{i,k+1} = (u_{1i,k+1}, u_{2i,k+1}, \cdots, u_{ni,k+1})$ to replace (if it gives better value) the target vector $x_{i,k}$ by the following rule:

$$u_{ji,k+1} = \begin{cases} v_{ji,k+1} & \text{if } (\text{randb}(j) \leq \text{CR}) \text{ or } j = \text{rnbr}(i) \\ x_{ji,k} & \text{if } (\text{randb}(j) > \text{CR}) \text{ or } j \neq \text{rnbr}(i), \\ j = 1, 2, \cdots, n. & \end{cases}$$

Step 4 : Continue Steps 2 and 3 for $i = 1, 2, \cdots N$ to get a new population for $k + 1$.

Step 5 : Continue Steps 2-4 until convergence is achieved.

5 Particle Swarm Optimization

The Particle Swarm Optimization Algorithm (PSO) is very recent method for global optimization.

Given the promising developments in the field of Global Optimization, it is only recently that we have found the means to handle problems which are exposed to analytical and numerical difficulties. These problems are dynamic in nature and to study their behaviours is still uncharted territory. Even though there are methods concerned with finding a solution to these problems, they still leave an open question to their efficiency.

Born out of the humble origins as a simulation of a simplified social system, the concept of a particle swarm was originally intended to graphically simulate the choreography of bird flocking or fish schooling. Suppose the following scenario: a flock of birds are randomly searching for food in a finite area. None of the birds have any knowledge of the locality of the food, but they know how far they are from the food, after each iteration. What is the best strategy for any bird to find the food?

Studies done on this behaviour inherent of the birds, found that the certain underlying rules enabled these large numbers of birds to flock synchronously, often changing direction suddenly, scattering and regrouping, etc [17]. It was found that these same rules dictate to human behaviour as well. The social sharing of information allows individual birds to profit from the discoveries and previous experience of the other birds during the search for food. This illustrated that the particle swarm model can be used as an optimizer.

The Particle Swarm Optimization algorithm (PSO) became one of the most recent additions to the list of global search methods, and is a member of the wide category of Swarm Intelligence methods, for solving Global Optimization problems. Motivated by the social behaviour of these bird flocks, the particle swarm optimization algorithm was proposed not only as a tool for optimization, but also a tool for representing socio-cognition of human and artificial agents, based on principles of social psychology. PSO as an optimization method provides a population-based stochastic search procedure in which individuals called particles change their position (state) with time. This simulates a bird flock's behavior where social sharing of information takes place. The particles in the population, which we now call the swarm, 'fly' around in a multi-dimensional search space in search of promising regions of the landscape. For the minimization case, such regions possess lower functional values than others visited previously. During flight, each particle is treated as a point in an n-dimensional hyper-cube. The particle adjusts its flying position according to its own experience, and according to the flying experience of neighbouring particles, making use of the best position encountered by itself and its neighbours.

Thus, as in modern GA and CRS algorithms, a PSO system combines local search methods with global search methods, attempting to balance exploration of the search space, and exploitation of neighbouring particles. It has relations with Artificial Life, which describes research into human-made systems that possess some of the essential properties of life. More specifically, it relates to swarming theories, and also with Evolutionary Computation, especially the Genetic Algorithm. PSO has been proved to be an efficient method for many Global Optimization problems, and, in some cases, it does not suffer the difficulties encountered by other GA technique.

The Particle Swarm Optimization (PSO) algorithms cater for all types of functions even those that are not continuous, those that have multiple local minima, and those that have numerical noise. It has fewer parameters and is not as computationally demanding compared to its other optimization counterparts. Moreover, it does not require gradient information of the objective function under consideration, but only its values, and it uses only primitive mathematical operators.

5.1 Applications

This algorithm can be successfully applied is in areas such as structural optimization, neural network training, control system analysis and design, and layout and scheduling problems, to name but a few [18].

The PSO algorithm can also cope with minimax problems which we encounter in numerous optimal control, engineering design, discrete optimization, Chebyshev approximation and game theory applications [19]. Operational Research problems, capital budgeting, portfolio analysis, network and VLSI circuit design, as well as automated production systems are some applications in which Integer Programming problems are met. The paper [20] investigates the performance of the PSO method in Integer Programming problems.

5.2 Description of PSO

In Global Optimization problems, the search for a solution is identified with the discovery of the global optimizer of a real-valued n-dimensional objective function $f(x)$. Given a real-valued objective function $f(x)$ defined on the set $x \in S$.

Let us assume, without loss of generality, that we are minimizing an n-dimensional objective function f. Now suppose we create a swarm of x_i, $i = 1, 2, \cdots, N$ particles. At any given point in time particle i has a current position, a record of the direction it followed previously, a record of its best ever position and a record of the best ever position by any particle in the swarm.

Each particle in represents a potential solution. PSO is concerned with moving each particle with a certain velocity across the solution space. This velocity is dependent on the two factors as discussed previously, that is, the tendency of the particle to follow its own experiences versus its tendency to follow the route leading towards the global best found thus far.

To create a kind of personal memory for a particle we introduce the point pb_i^k which stores the best ever position of particle i in iteration k. At the same time we store the fitness value corresponding to this temporary optimal position of particle i in $f(pb_i^k)$.

The algorithm then creates a global memory for the whole population of particles by recording the best ever position in g_b^k and the corresponding fitness value in $f(g_b^k)$. This is computed on every iteration of the algorithm.

The very first step is to distribute the particles across the solution space randomly, and then search for optima by updating generations. If for example, we are working in a three dimensional space then we would randomly generate 3 co-ordinates for each particle and hence have an initial position for the particle. The next step would be to generate a random initial velocity so that we could put our swarm into motion. Our next position thus depends on the previous position and current velocity. From this we get our first equation, where i is the swarm index and k is the iteration index:

$$x_i^{k+1} = x_i^k + v^{k+1} \tag{18}$$

It is now apparent that the next step must enable us to calculate v^{k+1}. The updating of velocity is based on pb_i^k, g_b^k and the position x_i^k of the particle i in iteration k. The first firm of the velocity update is based on the current position of particle i, x_i^k, and its personal best, pb_i^k. This is a vector in the direction of pb_i^k originating from x_i^k i.e.

$$(pb_i^k - x_i^k)$$

By adding a fraction of this term to the velocity we ensure that the particle will partially move towards its personal best. It is not apparent how to weight this term though, but once again we look to our flock of birds for the answer.

Just as we have little idea of where each bird (or particle) might start, we also have little idea as to where it may decide to go next. Even after gaining public knowledge, will it still decide to continue to move towards its own objective? Each bird will in some way choose its own path, whether this path is influenced more by its own personal mission or the experience of others, is hard for us discover as an outside observer. Thus, to account for this uncertainty we multiply $(pb_i^k - x_i^k)$ by r_1, a random number chosen between 0 and 1. This ensures that the particle moves a fraction towards its personal best.

Another term which contributes to the velocity is the desire of the particle to try to move closer to the global best. To incorporate this aspect we consider x_i^k and g_b^k. Once again

$$(g_b^k - x_i^k)$$

indicates the direction of g_b^k relative to x_i^k. By employing a similar idea to that mentioned above we realize that each particle moves in a random fashion towards the current global minimum. We thus multiply $(g_b^k - x_i^k)$ by r_2, where once again r_2 is a random number chosen between 0 and 1.

Now, if we consider $r_1(pb_i^k - x_i^k)$ and $r_2(g_b^k - x_i^k)$, we notice that as r_1 and r_2 are between 0 and 1, we will have that the movement of particle i will be almost negligible. We thus need to rescale these terms in order to see any real effect to the velocity. We thus introduce two parameters, c_1 and c_2 for the terms $r_1(pb_i^k - x_i^k)$ and $r_2(g_b^k - x_i^k)$, respectively. We should notice that this implies that c_1 relates to the cognitive or personal component and c_2 relates to the social component. Kennedy and Eberhart proposed that these parameters can be selected in such a way that the product $c_1 r_1$ or $c_2 r_2$ will have a mean of 1. They are thus specifically chosen as $c_1 = c_2 = 2$. The result of using these proposed values is that the particles overshoot the target half the time, thereby maintaining separation within the group and allowing for a greater area to be searched [21]. And thus we have an equation for calculating our velocity

$$v_i^{k+1} = v_i^k + c_1 r_1 (pb_i^k - x_i^k) + c_2 r_2 (g_b^k - x_i^k) \tag{19}$$

To understand fully the idea behind the two equations (18 and 19) we have just derived, it may be profitable to consider it in a 2-dimensional space so that some sort of visual insight may be obtained. Suppose at a time k, we have a particle at position O, with its personal best pb_i^k represented at the point A, and a global best g_b^k as the point C. The particle has a current velocity v_i, which is point B. The resultant direction of the particle is the vector OD. Below is a graph for a clearer understanding.

The PSO algorithm seems to adhere to the five basic principles of swarm intelligence, as defined by Eberhart and Millonas. The first principle is that the swarm must be able to perform simple space and time computations. This is known as Proximity. The second principle is Quality and is the ability of the swarm to respond to quality factors in the environment. The third principle is the ability of the swarm to not commit its activities along excessively narrow channels and this is known as Diverse response. The swarm's ability to not change every time the environment changes is the fourth principle and is known as Stability. The final principle is that the swarm must be able to change its behaviour, when the computational cost is not prohibitive, and this is known as Adaptability [21]. Indeed, the swarm in PSO performs space calculations for several time steps. It responds to the quality factors implied by each particle's best position and the best particle in the swarm, allocating the responses in a way that ensures diversity. Moreover, the swarm alters its behavior only when the best particle in the swarm changes, thus, it is both adaptive and stable [21].

5.3 The Pseudocode

```
• Initialize each particle.
• For each particle $i$ do the following
    – Calculate fitness (function) value
    – If the fitness value is better than the best fitness value $pb_i$ in history set current value as the new $pb_i^k$
    – End
• Choose the particle with the best fitness value of all the particles as the $g_b^k$
• For each particle $i$ do the following
    – Calculate particle velocity according equation (19)
    – Update particle position according equation (18)
    – End
• While maximum iterations or minimum error criteria is not attained
```

5.4 Parameter Control

The number N of particles forming the swarm can be chosen as in GA and CRS or DE. The learning factors c_1 and c_2 are akin to the individuality and sociality parameters, respectively. Kennedy and Eberhart suggested default values of $c_1 = c_2 = 2$, to result in a mean of 1, since these parameters are multiplied by random number r_1 and r_2. However, other settings were also used in different papers. But usually c_1 equals to c_2 and ranges from (0,4). Parsopoulos [18] proposed that c1 = c2 = 0.5 provided better results, but more recent work reports that it might be even better to choose a larger cognitive parameter, c_1, than a social parameter, c_2, but with $c_1 + c_2 <= 4$.

This being said, the parameters c_1 and c_2, are not the most influential factors for a faster convergence of the PSO algorithm. However, proper fine-tuning may alleviate getting caught in a local minimum. An extended study of the acceleration parameters is presented in [22]. The parameters r_1 and r_2 are used to maintain the diversity of the population, and they are uniformly distributed in the range (0,1)

There are many methods to randomly select an initial position and velocity for a particle in an n-dimensional search space. As in GA and DE or CRS the initial population consisting of N particles can be generated in S. However, the suggests distributing the the velocities using a uniformly defined distribution, governed by the equation

$$v_i^0 = x_L + \frac{r_3(x_U - x_L)}{\Delta t}$$

where x_U x_L are coordinate depended upper and lower limits and r_3 is a random number between 0 and 1, and Δt is typically chosen to be greater than 1.

A stopping criterion should be inspected on each execution cycle. A stopping condition similar to GA or DE can be used.

5.5 Introduction of Constant Inertia Weight

The first modification to PSO came in 1998. Eberhart and Shi [22] found that the velocity of the particle needed to be scaled down a bit to avoid it drifting in an opposite direction to the optimum at a high speed. They did this by introducing an inertia term w, which performed a scaling operation on the velocity, analogous to introducing momentum to the particle. It controls the impact of the previous history of the velocities on the current velocity, thus resulting in a trade-off between global and local exploration abilities of the particles. Equation (19) this became

$$v_i^{k+1} = wv_i^k + c_1r_1(pb_i^k - x_i^k) + c_2r_2(g_b^k - x_i^k) \tag{20}$$

For high values of w, we have larger velocities, and thus the particles overshoot the target, resulting in a good global search characteristic. Low values of w result in the particles moving in short bursts which is a desirable property to facilitate more local behavior, and hence allow for a more refined search [23].

The major problem inherited with the introduction of a constant inertia is that the problem becomes highly dependent on w. Choosing an intermediate value for w still yields a bad optimum for both the global and local phases of the search, and this requires more iterations to find the optimum. Hence Shi and Eberhart once again added another modification.

5.6 Linear Inertia Weight Reduction

Time decreasing inertia weights was found to be better that a fixed inertia weight. This is because the larger inertia weights at the beginning help to find good seeds and the later small inertia weights facilitate fine search [24]. If w is kept constant then, the larger it is, the stronger its global searching abilities are. This may be profitable at first, but as we move closer to the global optimum, we need to start narrowing our search for its exact location by moving in smaller steps. By reducing w linearly, it allows us to start off with a more globally based search to initially locate the vicinity of the global optimum, and then move towards a precision search of the exact location within this space. Thus, decreasing w linearly as the search progresses prevents particles from oscillating around the solution and is more profitable than keeping it constant.

This variation attempts to eliminate some of the drawbacks of constant inertia, however, the optimum rate for reducing w is still problem dependent, and constitutes the main drawback of this variation.

5.7 Conclusion

Particle Swarm Optimization method has granted success to the various researchers in the field of Global Optimization. Kennedy was a social psychologist. Eberhart was an electrical engineer. The particle swarm optimizer serves both of these fields, and many others, equally well. Social behavior is ever-present in the animal kingdom because it optimizes. Modeling social behaviour is a good approach to solving the optimization problems in engineering.

The method has been developed and modifications have been suggested. Particle Swarm Optimization is still in its infancy, and much further research is required to fully comprehend the dynamics and the potential limits of this technique. New untold applications remain to be conducted. Still, the simplicity and robustness of the Particle Swarm Optimization algorithm make it an attractive Global Optimization method that would see its success in any application.

6 The Multistart (MS) and Multilevel Single Linkage (MSL) Algorithms

A global optimization algorithm aims at finding a global minimizer or its close approximation of a function $f : S \subset R^n \to R$. A point x^* is said to be a global minimizer of f if $f^* = f(x^*) \leq f(x), \forall x \in S$. We assume that the function f is essentially unconstrained, i.e., all of its minimizers are in the interior of S. For practical interests, especially in applied sciences and in engineering, it is required that an approximation $\hat{x}^*$ of a global minimizer of f be found. Thus the global optimization problem might be considered solved if, for some $\epsilon > 0$, an element of the following sets has been identified

$$A_x(\epsilon) = \{x \in S | \|x - x^*\| \leq \epsilon\}, \tag{21}$$

$$A_f(\epsilon) = \{x \in S \Big| |f^* - f(x)| < \epsilon \tag{22}$$

A disvantage of the sets A_x and A_f is that there exist continuous functions in S with maximum absolute difference in function values over S arbitrary small but with global optima wide apart. Another possibility is obtained by defining

$$\phi(y) = \frac{m(\{z \in S | f(z) \leq y\})}{m(S)}, \tag{23}$$

where $m(\cdot)$ is the Lebesgue measure and taking

$$A_\phi(\epsilon) = \{x \in S | \phi(f(x)) \leq \epsilon\}. \tag{24}$$

The multistart type global optimization methods are two phase methods : global and local phase. In the global phase, the function value is evaluated in a number of randomly sampled points (unifrom distribution) . In the local phase, the sample points are manipulated, e.g., by means of local search, to yield a candidate global minimum.

Two techniques are used to manipulate the sample points. They are known as concentration and reduction [26, 27]. The concentration technique dives the sample points using a few iteration of local technique (not a complete local search). The reduction eliminates a prespecified fraction of the sample points whose function values are relatively high. As a result of the above two processes, the sample consists of groups of mutually relatively close points that correspond to the regions with relatively small function values. Within each group the points are still distributed according to the original uniform distribution.

6.1 Multistart

The simplist stochastic method for global optimization consists only of a global phase –known as the pure random search.

Pure Random Search

Step 1: Evaluate $f(x)$ in N points, drawn from a uniform distribution over S. The smallest function value found is the candidate solution for f^*.

In spite of its simplicity, Pure random search offers an asymptotic guarantee in a probabilistic sense. The proof is based on the simple observation that the probability that a uniform sample of size N contains at least one point in a subset $A \subset S$ is equal to

$$1 - \left(1 - \frac{m(A)}{m(S)}\right)^N . \tag{25}$$

The simplest way to make use of a local search, say, L, occurs in the multistart algorithm (MS). The main idea of MS is to use repeated local search L.

6.2 The Multistart Algorithm

Step 1 Draw a point from a uniform distribution over the search region S.

Step 2 Apply L to the new sample point.

Step 3 A termination criterion indicates whether to stop or to return to Step 1.

Note : The local minima with the smallest function value found is considered as (a candidate for) the global minima.

6.3 Multilevel Single Linkage Method (MSL)

MSL[27] is the major improvement over MS method which overcome some of the drawbacks that MS and clustering methods have. It has two phases (1) a global phase and (2) a local phase. In the global phase, the function is evaluated at a number N of random points. In the local phase, sample points are scrutinized to perform local searches in order to yield a candidate global minima. The algorithm initially discard less promising points from the N points and then local searches are carried out from some (not all) of the remaining points. For example if $N = 100$, initially 80% to 90% points are discarded and remaining (say 90% were discarded) 10 points are selected for local search. The local search will be applied to a subset of this 10 points. In MSL a r_k-descent sequence is considered. This is a sequence of sample points, such that any two successive points are within a distance r_k of each other and the function values in the sequence are monotonically decreasing.

6.4 The MSL Algorithm

The k-th iteration of the algorithm is as follows:

Step 1 Draw N points (and add them to the the previous $(k-1)N$ points) from a uniform distribution over the search region S and calculate their function values.

Step 2 Discard 80% to 90% points from kN points.

Step 3 Select starting points for local searches within the remaining points. Each point is selected as a starting point if it is not within a certain critical distance r_k of a point which was previously selected and which has a lower or equal function value.

Step 4 Perform local minimization from all starting points.

Step 5 A termination criterion indicates whether to stop or to return to Step 1.

Note : If confidence is sufficiently high that all the local minima have been found, regard the lowest local minima as global, otherwise repeat the steps.

The critical distance r_k is given by

$$r_k = \pi^{-\frac{1}{2}} \left[\sigma m(S) \Gamma(1 + \frac{n}{2}) \frac{\ln kN}{kN} \right]^{\frac{1}{n}}$$

where $m(S)$ is the measure of S, $\sigma = 4$ and Γ is the gamma function.

The MSL algorithm can also be written as follows:

Step 1 Sample N points from S and calculate $f(x)$ on these points. Add these points to the previously drawn points in all earlier iterations. Discard a percentage of worse points.

Step 2 Order the sample points such that $f(x_i) \leq f(x_{i+1}), 1 \leq i \leq M$, M being the number of remaining points. For every i, start a local search from x_i if it has not been used as a starting point at previous iteration or if there is another sample point, or previous detected local minimum within the critical distance r_k of x_i.

Step 3 If the stopping condition is satisfied then stop else go to step 1.

7 Optimization Technique Based on Learning Automata

The multitude of optimization techniques that have been developed so far are encouraging but of these techniques most are incapable of handling many problem situations where the problem concerned are extremely noisy and thus are exposed to analytical and numerical difficulties. Even so many problems are dynamic in nature and their behaviors are unknown. Therefore a need for optimization of many unknown dynamical system still remains an open question. To this end, we will learn a optimization technique based on Learning Automata.

The optimization algorithm we will be describing in this section is based learning systems which can take into consideration all the prior information concerning the process (a system or a problem) to be optimized and improve its performance or behavior with time.

The algorithm is concerned with the application of a stochastic automata with variable structure to solve a optimization problem in the presence of constraints, without making any assumption such as linearity of constraints or convexity of function etc. A learning automata operates in a random medium which models the optimization problem to be solved. The learning system collects pertinent information during its operation and process it according to a certain rule, so as to optimize a prespecified cost function under some constraints. A probability distribution is associated to the set of feasible solutions. Each probability represents the weight given by learning automata. The automaton processes and collects the costs of the process structures and updates the probability distribution by the use of a reinforcement scheme to achieve predefined goals (i.e., decrease the cost function). The goals to be reached are contained in a performance evaluation unit which evaluates the quality of a specific structure by a technique of reward or penalty. The reinforcement scheme uses this response (penalty or reward) to increase or decrease the weight (i.e., the probability) associated to this structure and generates a new probability distribution from the old one.

7.1 The Learning Algorithm

Let us search for an optimal plant design which minimizes a certain performance criterion in the presence of constraints. This is a discrete problem with the optimization variables u_i $(i = 1, 2, \cdots, N)$ represents a structure number. If the problem is continuous with continuous variables then the domain $[u_{min}, u_{max}]$ of the optimization variable is discretized into a set of N values, u_i, $(i = 1, 2, \cdots, N)$. Thus the optimization problem can be regarded as determination of the value(s), one one or more, of the discrete variable u_i which gives minimum function value $f(u)$ in the presence of constraints $g_j(u) \leq 0$, $j = 1, 2, \cdots, M$.

A learning system is composed of a stochastic automata and a performance evaluation unit. Let $P(k) = (p_1(k), p_2(k), p_3(k), \cdots, p_N(k))$ be the probability distribution at any iteration k (initially $k = 0$). A probability $p_i(k)$ is associated with u_i (the action u_i) and represents the weight (confidence in some sense) given by stochastic automaton to this action. Initially all action (u_i) have equal weight (i.e., probability).

Based on the probability distribution vector $P(k)$, at each iteration k, the automaton selects randomly an action $u_i(k)$ with associated cost function $f(u_i(k))$ and updates the overall probability distribution $(P(k))$ according to the response of the performance evaluation unit. This unit contains information or description of the objective concern-

ing the optimization problem. The performance evaluation unit generates a response corresponding to a reward if the selected action $u_i(k)$ agrees with the goals to be reached or a penalty in the opposite case.
Case I: Reward
(i) increase the probability $p_i(k)$,

$$p_i(k+1) = \{p_i(k) + \beta_o \cdot p_i(k)[1 - p_i(k)]\} / \sum_p; \tag{26}$$

(ii) decrease the other probabilities to constraint the sum to be one.

$$p_j(k+1) = \{p_j(k) - \beta_o \cdot p_i(k) p_j(k)\} / \sum_p, \qquad j \neq i (j = 1, 2, \cdots N). \tag{27}$$

The index j corresponding to discrete variables which have not been selected at iteration k. The quantity: $\sum_p = [p_1 + p_2 + \cdots + p_N]$ is introduced to avoid round-off errors and to guarantee that $p_i(k) \in [0, 1]$.
Case II: Penalty
(i) decrease the probability $p_i(k)$,

$$p_i(k+1) = \{p_i(k) - \beta_1 \cdot p_i(k)[1 - p_i(k)]\} / \sum_p; \tag{28}$$

(ii) increase the other probabilities to constraint the sum to be one.

$$p_j(k+1) = \{p_j(k) + \beta_1 \cdot p_i(k) p_j(k)\} / \sum_p, \qquad j \neq i (j = 1, 2, \cdots N). \tag{29}$$

The index j corresponding to discrete variables which have not been selected at iteration k.

The adaptation of parameters β_o and β_1 are such that $0 < \beta_o \leq 1$ and $0 \leq \beta_1 \leq 1$. In most cases $\beta_1 = 1$ is selected.

The algorithm selects randomly an action u_i at each iteration according to the following procedure, based on the probability distribution.

$$\sum_{j=1}^{i} p_j \geq Z,$$

where Z is a normally distributed number generated at each iteration, $Z \in [0, 1]$. The behavior of the learning system can be summarised by the following steps performed at each iteration.

Step 1 : Random selection of one of discrete u_i according to the probability distribution P.

Step 2 : Computation of the cost function.

Step 3 : Use of this cost value to determine if this action u_i has to be rewarded or penalized.

Step 4 : Updates the probability distribution according to this response.

Step 5 : Return to Step 1.

The Algorithm:

```
1. Repeat While $\max\{p_i \in P(k)\} < \eta (\eta = 0.5)$
2.     randomly select a $u_i$ according to the probability $p_i \in P(k)$
3.     evaluate $f(u_i)$
4.     evaluate $P(k+1)$ via (26)-(28).
5.     increament $k$
6. End Repeat.
```

8 Genetic Programming (GP)

8.1 Introduction

In recent years, applied researchers have increasingly interested in evolutionary strategies (ES) such as differential evolutions [16], genetic algorithm [7] and genetic programming [25], to tackle various practical problems. The power and effectiveness of GP in engineering design and system identification of many engineering problem are well known. The GP approach have been widely used to calibrate functions to empirical data. *The manipulation and optimization of these approximate functions that describe the physical process is achieved using GP.*

Our objective, here, is to use GP and sets of empirical data for system identification. Pioneering GP work by Koza [25] has produced impressive results for many engineering systems. Koza defined symbolic regression, a form of GP as follows:

'Symbolic regression (i.e., functional identification) involves finding a mathematical expression, in symbolic form, that provides a good, best or perfect fit between a given finite sampling of values of the independent variables and the associated values of the dependent variables. That is, symbolic regression involves finding both the functional form and the numeric coefficients for the model'.
He goes on to say, 'The real problem is often both the discovery of the correct functional form that fits the data and the discovery of the appropriate numeric coefficients that goes with that functional form·. It is data-to-function regression.'

8.2 The Genetic Programming Paradigm

The genetic programming (GP) paradigm continues the trend of dealing with the problem of representation in genetic algorithm (GA) by increasing the complexity of the structures undergoing adaptation. In particular, the structures undergoing adaptation in GP are general, hierarchical computer programs of dynamically varying size and shape. Problems in artificial intelligence, symbolic processing, and machine learning can be viewed as requiring discovery of computer program that produces some desired output for particular inputs.

GP starts with an initial population of randomly generated computer programs composed of **functions and terminals appropriate to the problem domain**. The functions may be standard arithmetic operations (e.g., +,-,*,/ etc), standard mathematical function (e.g., $\sin, \cos, \log, \exp$ etc), standard programming operators (e.g., AND, OR, NOT, DO Until etc), logical function (IF then ELSE). On the other hands, terminals are typically variable atoms (e.g., inputs, sensors, detectors, state variable, say, x for some system) or constant atoms (e.g., the number 3 or the boolean constant NIL). Therefore, depending on the particular problem, the computer program may be boolean, integer, real, complex, vector, symbolic, or multiple valued. The creation of the initial random population is a blind random search of the problem search space.

Each individual computer program (each member) in the population is measured in terms of how well it performs in the particular problem environment. This measured is called the fitness measure.

In summary, the GP paradigm breeds computer programs to solve problems by executing the following three steps:

- Generate an initial population of random compositions of the functions and terminals of the problem.

- Iteratively perform the following substeps until the termination criterion has been satisfied:

  a : Execute each program in the population and assign it a fitness value according to how well it solves the problem.

  b : Create a new population of computer programs by applying the following two primary operations. The operations are applied to computer program(s) in the population chosen with a probability based on fitness.

    (i) Copy existing computer programs to the new population.

    (ii) Create new computer programs by genetically recombining randomly chosen parts of two existing programs.

- The best computer program that appeared in any generation (i.e., the best so-called individual) is designated as the result of genetic programming.

The Structures Undergoing Adaptation :In any adaptive system or learning system, at least one structure is undergoing adaptation. In GP, the structures undergoing adaptation are a population of individual points from the search space, rather than a single point. The individual structures that undergo adaptation in GP are hierarchically structured computer programs. The size, the shape and the contents of these computer programs can dynamically change during the process.

The set of possible structures in GP is the set of all possible compositions of functions that can be composed recursively from the set of N_f functions from

$$F = \{f_1, f_2, f_3, \cdots, f_{N_f}\}$$

and the set of N_t terminals from

$$T = \{a_1, a_2, \cdots, a_{N_t}\}$$

Each particular function f_i the function set F takes a specific number $z(f_i)$ of arguments $z(f_1), z(f_2), \cdots, z(f_{N_f})$. That is function f_i has arity $z(f_i)$.

8.3 The Initial Structures

The initial structures in GP consists of individuals in the initial population of individual S-expressions for the problem.

The generation of each individual S-expression in the initial population is done by randomly generating a rooted, point labeled tree with ordered branches representing the S-expression.

We begin by selecting one of the functions from the set F at random (using uniform probability distribution) to be the Label of the root of the tree. We restrict the selection of the label for the root of the tree to the function set F because we want to generate a hierarchical structure.

Whenever a point of the tree is labeled with a function f from F, then $z(f)$ lines, where $z(f)$ is the number of arguments taken by the function f, are created to radiate out from that point. Then for each radiating line, an element from the combined set

$C = F \cup T$ of functions and terminals is randomly selected to be the label for the endpoint of that radiating line.

If such point is chosen to be the label for any such endpoint, the radiating process continues recursively. Figure $2(a)$ shows the beginning of the creation of a random program tree. For example in figure $2(a)$, let the function + (arity 2) was selected as the root from the set F, then in figure $2(b)$ the function $*$ (multiplication, arity 2) from the combined set $C = F \cup T$ was selected as the label of the internal nonroot point (point 2) at the end of the first leftmost line radiating from the point with the function + (point 1). Since a function was selected for point 2, it will be an internal nonroot point of the tree that will eventually be created. The function $*$ takes two arguments, so the figure shows two lines radiating out from the point 2.

If a terminal is chosen to be the label of any point, that point becomes an endpoint of the tree and the generating process is terminated for that point. For example, in the figure $2(c)$ below, the terminal A from the set T was selected to be label of the first line radiating from the point labeled with the function $*$. Similarly the terminal B and C were selected to be the labels of two other radisting lines. This generating process continues from left to right until a completely labeled tree has been created.

This generative process can be implemented in several different ways resulting in initial random trees of different sizes and shapes. The *full* and *grow* techniques are two of the basic methods creating trees having a wide variety of sizes and shapes.

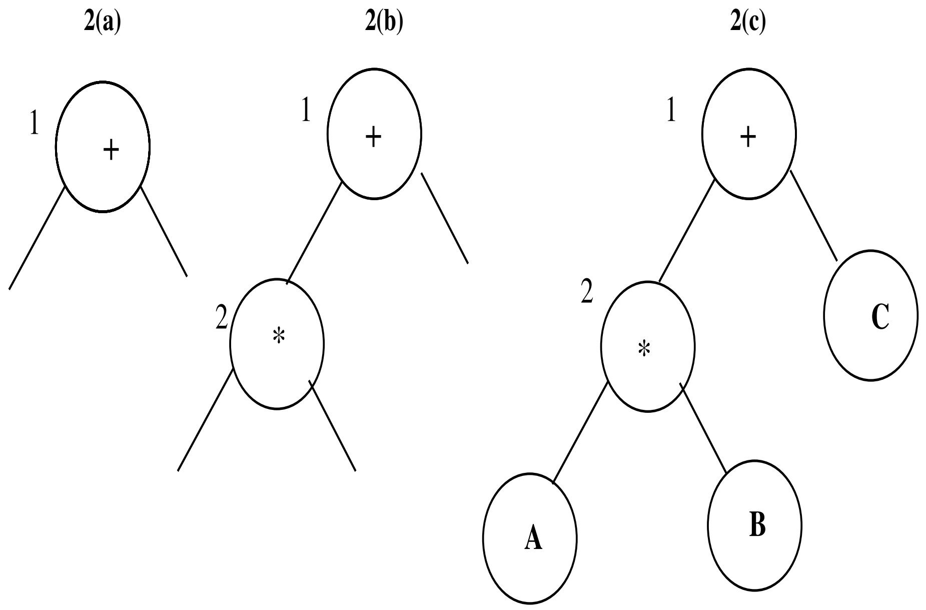

Figure 3: Initial Structure Generation

The full method of generating the initial random population involves creating trees for which the length of every nonbacktracking path between an endpoint and the root is equal to the specified maximum depth. This is accomplished by restricting the selection of the label for points at depth less than the maximum to t F, and restricting the selection of the labels for points at the maximum depth to the terminal set T.

The grow method of generating the initial random population involves growing trees that are variably shaped. The length of a path between an endpoint and the root is no greater than the specified maximum depth. This is accomplished by making the random selection of the label for points st depths less than the maximum from the combined set $C = F \cup T$ consisting of the function set F and the terminal set T, while restricting the the random selection of the labels for points at the maximum depth to the terminal set T.

In general, however, the ramped half-and-half method is used in GP. This method is a mixed method that incorporate both the full and the grow method. The method involves creating an equal number of trees using a depth parameter between 2 and the maximum specified depth. For example, if the maximum specified depth is 6, 20% of the trees will have depth 2, 20% will have depths 3, and so forth up to depth 6.

Note that, for the trees created with the full method for a given depth, all paths from the root of the tree to an endpoint are the same length and therefore have the same shape. In contrast, for the trees created via the grow method for a given value of depth, no path from the root of the tree to the endpoint has a depth greater than the given value of depth. Therefore, for a given value of depth, these trees vary considerably having a wide variety of sizes and shapes.

8.4 Closure of the Functional Set and the Terminal Set

The closure property requires that each of the functions in the function set is able to accept, as its arguments, any value and data type that may possibly be returned by any function in the functional set and any value and data type that may possibly be assumed by any terminal in the terminal set. That is, each function in the function set should be well defined and closed for any combination of arguments that it may encounter.

8.5 Computer Representation of Structures

The structures are coded from the tree structure into Reverse Polish Notation (RPN). As an example, suppose we have the following tree structure shown in Figure 3.

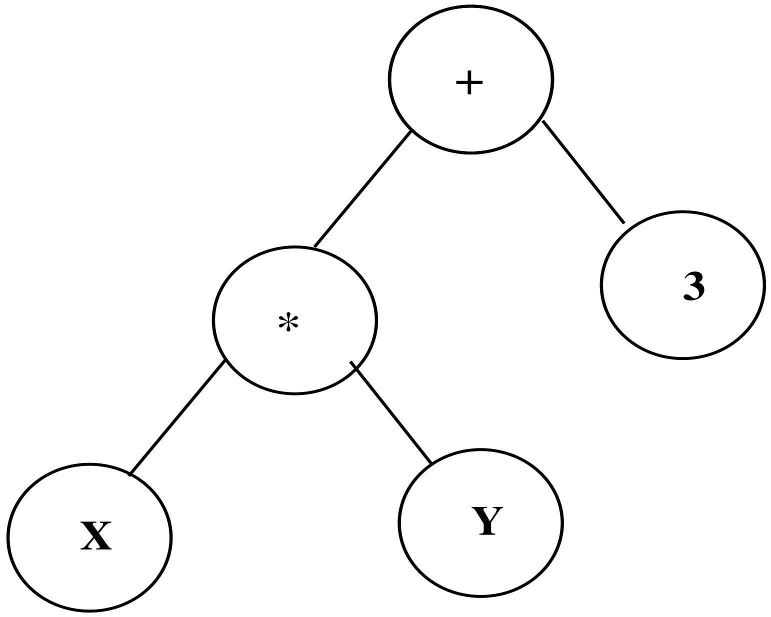

Figure 4: Example of S-expression tree

In standard notation the equation would be:

$$(X * Y) + 3$$

but in RPN the structure is

$$+ * XY3,$$

this has an immediate advantage that parenthesis are no longer necessary, and there is a one-to-one correspondence between the standard notation and RPN of algebraic formula.

Each element within the individual is represented by an unsigned character and a floating point number. Figure 4 shows an individual together with its coded values.

Whenever a value of 100 is received from the character array (a real numbered terminal) the location of the real number is the same point as the character of value 100 but in the floating number array. This requires additional memory for the real numbers but is the best method to date for coding.

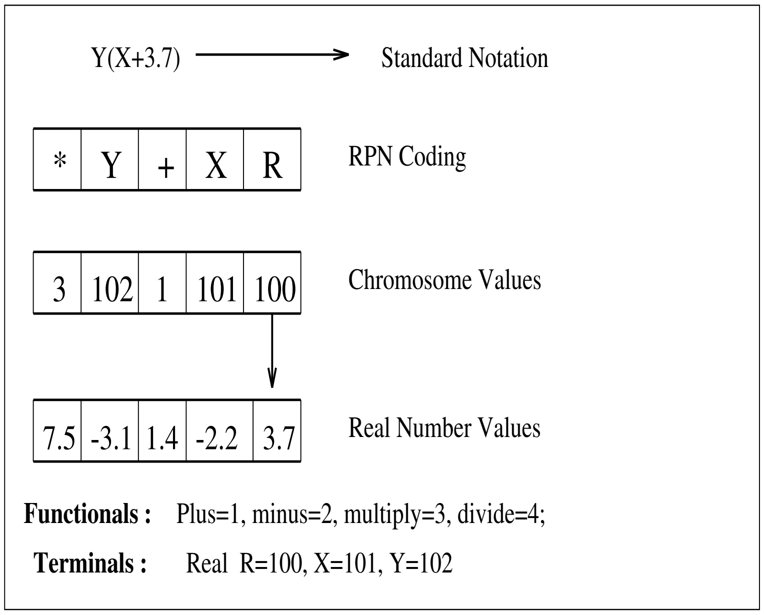

Figure 5: Representation of Structures

8.6 The Fitness

Fitness is the driving force in the natural selection and therefore, likewise, in both GA and GP. In nature, the fitness of an individual is the probability that it survives in the reproduction process. In the artificial world of mathematical algorithms, the one which we are considering, we measure fitness in some way and then use this measurement to control the application of the operation such as the selection and reproduction that modifies the structure of our artificial population.

Fitness can be measured in many different ways, sometimes it is measured implicitly and sometimes explicitly, according to the needs. However, in our consideration we will stick to the explicit from. Some of the explicit form of the fitness measure are as follows

1. Row Fitness

2. Standardized Fitness
3. Adjusted Fitness
4. Normalized Fitness.

8.6.1 The Row Fitness

The row fitness is the measurement of fitness that is stated in the natural terminology of the problem itself. The common example of row fitness could be the error. That is, the row fitness of an individual S-expression is the sum of the distances, taken over all the fitness cases, between the point in the range space returned by the S-expression for the set of arguments associated with particular fitness case and the correct point in the range space associated with the particular fitness case. When the row fitness is error, the row fitness $r(i,t)$ of an individual S-expression i in the population of size M at any generational time (iterations) step t

$$r(i,t) = \sum_{j=1}^{N_e} |S(i,j) - C(j)|$$

where $S(i,j)$ is the value returned by the S-expression i for the fitness case j (of N_e cases) and where $C(j)$ is the correct value of the fitness case j.

8.6.2 The Standardized Fitness (SF)

The SF $s(i,t)$ relates the row fitness so that a lower numerical value is always the better value. For example, in an optimal control problem, one may be trying to minimize the cost, so leaase value of row fitness is better. However, since GP and GA are inherently related to the maximization problem we calculates the SF in the following way.

$$s(i,t) = r_{max} - r(i,t)$$

where r_{max} is the maximum possible value of the row fitness.

8.6.3 The Adjusted Fitness (AF)

The AF measure $a(i,t)$ is computed from the standardized fitness in the following way.

$$a(i,t) = \frac{1}{1 + s(i,t)}$$

where $s(i,t)$ is the SF for individual i at time t. The AF lies between 0 and 1. This fitness is bigger for the better individual in the population.

8.6.4 The Normalized Fitness (NF)

The NF is normally calculated from AF value $a(i,t)$ and is as follows:

$$n(i,t) = \frac{a(i,t)}{\sum_{k=1}^{M} a(k,t)}$$

The NF has three desirable rules and they are as follows:

1. It ranges between 0 and 1.
2. It is larger for better individual in the population.
3. The sum of the normalized fitness value is 1.

8.7 The Reproduction

Reproduction is carried out in two steps : in the first step a single S-expression is selection according to its fitness. Second, the selected individual is copied to the new population with any alterations. This can be done for 10% times, i.e., if we have a population of 100 then in the new population 10 new individuals are according to above rule.

8.8 The Crossover

The crossover is very much similar to the crossover in GA, it starts with two S-expressions (parents). Both the parents are selected according to their fitnesses. The operation begins by selecting independently, using a uniform probability distribution, one random point in each parent to be the crossover point for that parent. Note that the two parents typically are unequal in size.

For example, consider the two parental S-expressions in the figure 5 below. The function appearing in these two are Boolean AND, OR, and NOT functions. The terminals appearing in the figure are the Boolean arguments D1 and D2. Assume that the second point in the (out of six of the first parent) is selected randomly as the crossover point for the first parent. The crossover point of the first parent is therefore the NOT function. Suppose also that the sixth point (out of ten points) is selected as the crossover point for the second parent. Here the crossover point is AND. Carefully look at the figures how the crossover is done.

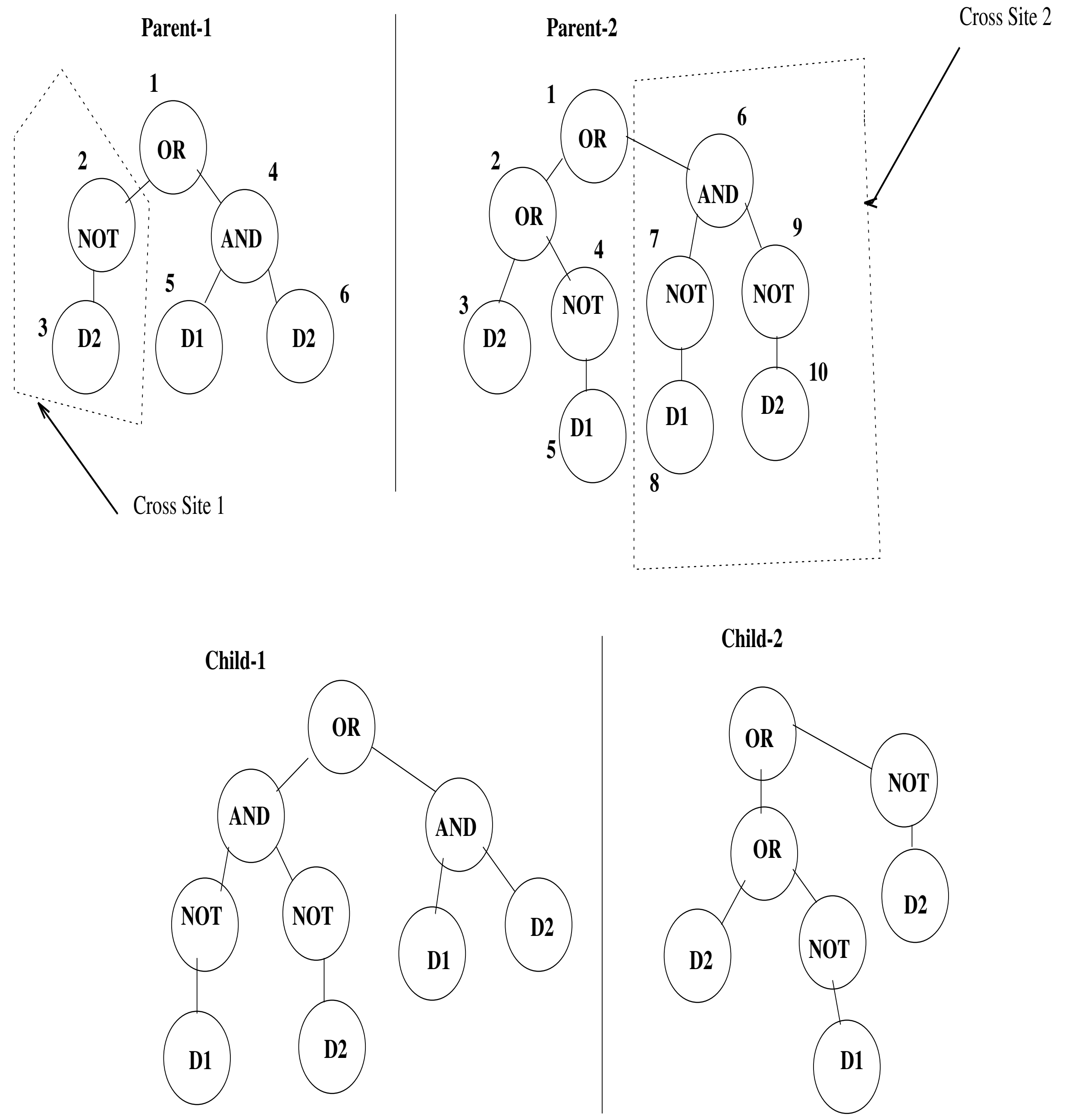

Figure 6: Crossover Operation

8.9 The Mutation

As in GA the mutation operation brings random changes in the population. Each individual is selected with a very small probability for mutation. Mutation introduces diversity in the population that may be tending to converge prematurely. In GP, if a S-expression is selected for mutation (usually with small probability) then the mutation operation begins by selected a point at random within the S-expression. This mutation point can be a function or a terminal point of the tree. The mutation operation then removes whatever is currently at the mutation point and what is below the mutation point. A newly created subtree is then inserted at this point. The subtree is controlled by the parameter that specifies the maximum size of a tree. For example, in the figure 6 below the point 3 (i.e., D1) is selected as the mutation point. The subexpression (NOT D2) is then randomly generated and then inserted at that point to produce a new S-expression.

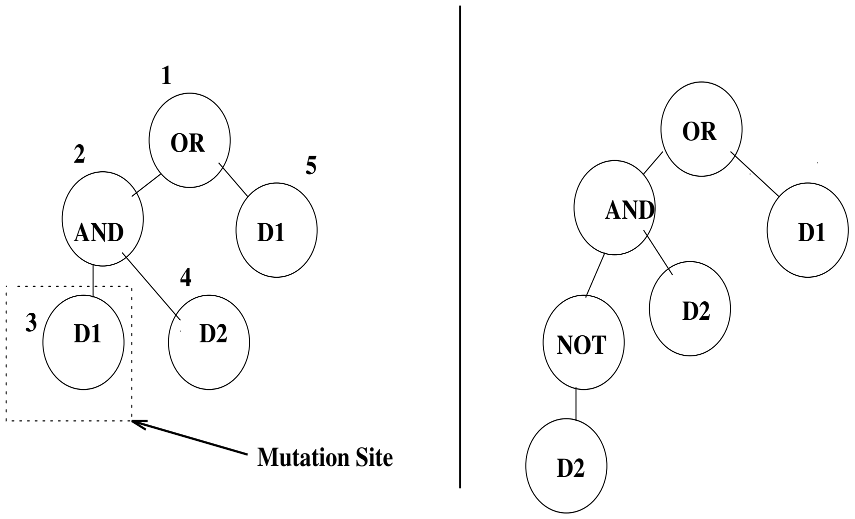

Figure 7: Mutation Operation

References

[1] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller and E. Teller, Equation of State Calculation by Fast Computing Machines, *Journal of Chemical Physics*, Vol.2, No. 6, pp.1087-1091, 1953.

[2] S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, Optimization by Simulated Annealing, *Science*, Vol.220, No. 4598, pp.671-680, 1983.

[3] E. H. L. Aarts and J. H. M. Korst, *Simulated Annealing and Boltzmann Machines*, Wileym Chichester, 1988.

[4] A. Dekkers and E. Aarts, Global Optimization and Simulated Annealing, *Mathematical Programming*, 50, pp.367-393, 1991.

[5] R.W. Eglese, Simulated Annealing, A tool for Operational Research, *European Journal of Operational Research*, Vol.46, pp.271-281, 1990.

[6] J. H. Holland, *Adaptation in Natural and Artificial Syayems*, University of Michigan Press, Ann Arbor, MI., 1975.

[7] D. E. Goldberg, *Genetic Algorithms : in search, Optimization & Machine Learning*, Addition-Wesley Publishing Company, Inc., 1989.

[8] W. L. Price, A Controlled Random Search Procedure for Global Optimization, Computer Journal, Vol. 20, pp.367-370, 1977.

[9] J. A. Nelder and R. Mead, A Simplex Method for Function Minimization, Computer Journal, Vol. 7, pp.308-313, 1965.

[10] W. L. Price, Global Optimization by Controlled Random Search, Journal of Optimization Theory and Applications, Vol. 40, pp.333-348, 1983.

[11] W. L. Price, Global Optimization Algorithms for a CAD Workstation, Journal of Optimization Theory and Applications, Vol. 55, pp.133-146, 1987.

[12] M. M. Ali and C. Storey, Modified Controlled Random Search Algorithms, International Journal of Computer Mathematics, Vol. 54, pp. 229-235, 1994.

[13] Mohan, C. and Shanker, K., A Controlled Random Search technique for Global Optimization using quadratic approximation, Asia-Pacific Journal of Operational Research Vol.11, pp.93-101, 1994.

[14] M. M. Ali, A. Törn and S. Viitanen, A Numerical Comparison of Some Modified Controlled Random Search Algorithms, Journal of Global Optimization, Vol. 11, pp. 377-385, 1997.

[15] P. Brachetti, M. De Felice Ciccoli, G. Di Pillo and S. Lucidi, A new version of the Price's Algorithm for Global Optimization, Journal of Global Optimization, Vol.10, pp.165-184, 1997.

[16] R. Storn and K. Price, *Differential Evolution : A simple and efficient heuristic for global optimization over continuous spaces*, Journal of global optimization, volume 11, pp.341-359, 1997.

[17] Holland J.H.; (1992); Adaptation in Natural and Artificial Systems; MIT Press, Cambridge, MA

[18] Schutte J.F., Reinbolt J.A., Fregly B.J., Haftka R.T. and George A.D.; (2004); Parallel Global Optimization with the Particle Swarm Algorithm; International Journal of Numerical Methods in Engineering (in press)

[19] Laskari E.C., Parsopoulos K.E., and Vrahatis M.N.; (2002); Particle Swarm Optimization for Minimax Problems; IEEE Congress on Evolutionary Computation; 1576-1581 25

[20] Laskari E.C., Parsopoulos K.E., and Vrahatis M.N.; (2002); Particle Swarm Optimization for Integer Programming; IEEE Congress on Evolutionary Computation; 1582-1587

[21] Parsopoulos K.E. and Vrahatis M.N.; (2002); Recent Approaches to Global Optimization Problems through Particle Swarm Optimization; Natural Computing 1: 235-306; Netherlands: Kluwer Academic Publishers

[22] Eberhart R.C. and Shi Y.; (2000); Comparing Inertia Weights and Constriction Factors in Particle Swarm Optimization; Proceedings of the Congress on Evolutionary Computing; 84-88

[23] Parsopoulos K.E. and Vrahatis M.N.; (2001); Modification of the Particle Swarm Optimizer for locating all the global minima; Artificial Neural Networks and Genetic Algorithms; 324-327; Springer, Wien

[24] , Shi Y. and Eberhart R.C.; (1998); Parameter Selection in Particle Swarm Optimization; Evolutionary Programming VII; 611-616; Springer

[25] John R. Koza, *Genetic Programming : On the programming of computers by means of natural selection*, The MIT Press, 1994.

[26] A. H. G Rinnoy Kan and G. T. Timmer, *Stochastic global optimization methods; Part-I : Clustering methods*, mathematical programming, volume 39, pp.27-56, 1987.

[27] A. H. G Rinnoy Kan and G. T. Timmer, *Stochastic global optimization methods; Part-II : Multilevel methods*, mathematical programming, volume 39, pp.57-78, 1987.

[28] W. L. Price, A Controlled Random Search Procedure for Global Optimization, *Towards Global Optimization 2*, Edited by L.C.W. Dixon and G.P. Szegö, North-Holland, Amsterdam, Holland, pp.71-84, 1978.

[29] W. L. Price, Global Optimization by Controlled Random Search, *Journal of Optimization Theory and Applications*, 40, pp.333-348, 1983.

[30] W. L. Price, Global Optimization Algorithms for a CAD Workstation, *Journal of Optimization Theory and Applications*, 55, pp.133-146, 1987.

[31] , K. Najim, L. Pibouleau and M. V. Le Lann, *Optimization Technique based on Learing Automata*, Journal of Optimization Theory and Applications, Volume 64, Number 2, 1990.

[32] A. Törn and A. Žilinskas, *Global Optimization*, Springer-Verlag, Berlin, 1989.

[33] M. M. Ali, Some Modified Stochastic Global Optimization Algorithms with Applications, Loughborough University of Technology, *Ph.D Thesis*, 1994.