EE227
Genetic Algorithms and Machine Learning
Semester 2

Assoc. Prof. P. N. Suganthan
Email: epnsugan@ntu.edu.sg
Tel: 6790 5404
Office: S2-B2a-21

Introduction

- Textbooks:
  - **D. E. Goldberg, "Genetic algorithms in search, optimization, and machine learning".**
  - **Andries Engelbrecht, "Computational Intelligence: An Introduction," Wiley, 2nd edition or later.**
  - **Additional references are given in the notes.**
- There are two assignments for this subject.
- The notes and presentation slides are available from NTULearn. If you cannot access them, send me an email to receive them by email.

Outline for Weeks 1 - 7

- **Introduction to GAs:** Description of GA; brief discussion of optimization and optimization algorithms; a simple GA (SGA); simulation; similarity templates (schemata or schemas).
- **Review of Combinatorics and probability:** combinatorics; permutations and combinations; sample space and events; probability; conditional probability; Bayes rule; random variables.
- **Computer implementation:** General structure as flow chart; reproduction; crossover & mutation; the main program; test problem; mapping objective function to fitness form; coding; multivariable problems.
- **Function optimization:** historical applications; De Jong and function optimization; stochastic universal sampling; two-point and uniform crossover; replacement strategies; examples.

Outline for Weeks 1 - 7

- **Mathematical foundations of GAs:** the fundamental theorem; two-armed and *k*-armed bandit problem; the building block hypothesis; the minimal deceptive problem.
- Fitness scaling, real parameter-based operators, real parameter evolutionary algorithms such as the particle swarm optimization, differential evolution, etc.
- Constraint handling
- Multi-objective evolutionary algorithms
- Multi-modal optimization.

Natural Evolution

- Natural evolution is characterized by the *survival of the fittest*, as described by Charles Darwin.
- It also implies that the fitter individuals survive longer and reproduce more offsprings than less fitter individuals.
- When individuals reproduce, their genetic information is passed on to the offsprings (or children).
- If the parents are fitter individuals, then it is highly likely (but not always) that the offsprings will also be fitter individuals.

Some Real Life Observations

- Why some species become extinct (i.e. they cannot be found any more in this world)? Not fit enough to compete with fitter species and survive …
- Life style of olden time Kings with a large number of ...
- Influence of developments in the research front:
  - artificial evolution in agriculture & animal husbandry (genetic engineering).
  - decelerating the natural evolution of the humans.
- Other observations such as the social security in some countries, law, etc. ??

GAs & Evolution

- Genetic algorithms attempt to mimic (in an over-simplified manner) the natural evolution of species as described by Charles Darwin.
- GA was first studied by J Holland & his colleagues at the University of Michigan.
- A more generic name for approaches based on the natural evolution is *evolutionary computation*.
- Genetic algorithms and their variants are used to solve *difficult* optimization problems.
- GA is not at all related to life sciences!!

Principle of Employing (Artificial) Evolution for Optimization

- *The process of natural evolution has the ability to select the fitter individuals from populations.*
- *The above process of selecting better/fitter individuals has a close similarity to the process of optimization.*
- *If we represent solution instances to a particular problem by a population, and then if we apply an artificial version of the evolution to this population of candidate solutions, then this artificial evolution can be expected to make the population of solutions better as the number of generations increases.*

From Natural Evolution to GAs

- In real life, we can observe that the selection of 2 parents to reproduce offsprings as well as the characteristics of the offsprings are to some extend random.
- As evolutionary computations attempt to mimic natural evolution, randomness is appropriately (also frequently) used in evolutionary computation algorithms.
- GAs are basically *population-based guided random search*. This is because:
  - we work with a population of individuals to represent potential solutions to the problem to be solved
  - the search is guided by the evolution theory
  - the selection of parents to reproduce offsprings and the characteristics of the offsprings are somewhat random.

Evolutionary Computation Techniques

- GAs
  - First proposed by Dr J Holland and Ken DeJong in early 1970s
- Evolutionary programming
  - First proposed by Dr L Fogel in 1960s
- Evolution strategies
  - First proposed by Drs Rechenberg and Schwefel in 1960s
- Genetic Programming
- In this module (weeks 1 – 6), we will study GAs, particle swarm optimization (PSO), constraint handling, evolution strategy (ES), differential evolution (DE), multi-objective evolutionary algorithms and multimodal optimization.

GAs for Optimization

- Every individual string in the population represents an instance of a solution to the problem being solved.
- The solution is suitably coded so that evolutionary computation operators such as *crossover*, *mutation, reproduction*, etc. can be applied.
- Fitness of an individual population member (i.e. a solution instance) reflects how good the solution is to the problem being solved.
- Other issues: How many members in the population, (i.e. *population size*)? how many *generations* to evolve the populations?
- Details will be discussed later ….

Introduction to Optimization

- Optimization is the process of attempting to find the *best* or *optimal* solution to the problem under consideration.
- A large number of problems in various fields can be formulated as optimization problems.
- We usually tackle finite dimensional deterministic problems defined as follows:
  - A description of the *decision variables* (assume there are *n* variables) eg. $x_1, x_2, \ldots x_n$.
  - A description of the *admissible* or *permissible* or *feasible* region of the decision variables, eg. $\Omega \in \Re^n$. Typically Ω is defined implicitly via functional relations called *constraints or* explicitly by *upper/lower bounds*.
  - A measure of performance called an *objective function*.

Introduction to Optimization

The objective function can be described as follows: $f : \Omega \rightarrow \Re$

The problem can now be stated as follows:

Minimize *f(x)* over $x \in \Omega$

Note that maximization problems can be converted to minimization problems by multiplying by –1.

There are discrete, continuous, mixed, combinatorial optimization algorithms.

Discrete Optimization

- The decision variables take on finite number of discrete values.
- Transportation, assignment, and shortest path computation are of this type.
- We can approximate continuous problems by discrete ones by quantizing the decision variables.
- Consider a trivial continuous optimization problem:
  - minimize $f(x)=x^2$ $\quad -1 <= x <= 1$
  - This problem can be discretised for example using 32 bits plus a sign bit.
  - There are 2^{33} possible choices for *x* and the discrete solution can be considered an approximate solution to the continuous problem.

Discrete Optimization

- Discrete optimization problems can be solved by exhaustively searching all possible values of the decision variables. This is not a suitable approach for practical problems as they are likely to be a large number of enumerations.
- Hence, we need more efficient algorithms than *explicit enumeration*.
- Some traditional algorithms are simplex method, shortest path algorithms, maximum flow algorithms and so on (Refer to "Linear and Nonlinear Programming" book by D. G. Luenberger for details on the problems and traditional solution methods).

Global Optimum vs Computational Cost

- For some difficult optimization problems, we do not have efficient algorithm to find the optimal solution within a reasonable time.
- In practice, a good solution obtained within a reasonable time is better than the global optimum solution obtained by taking excessive time and/or computational resources.
- In these situations, heuristic methods such as the GA or simulated annealing can be used. *These approaches make use of randomness to escape from local optimal solutions*.
- Heuristic methods do not have firm mathematical basis (compared to mathematical programming methods) and they do not guarantee to produce the optimal solution.

Nonlinear Programming Problems

- Problems in which objective and equality/inequality constraints are described by continuous nonlinear functions (usually differentiable functions), can be solved by using methods of calculus. A general nonlinear programming problem:

$$\begin{aligned} &\text{minimise} && f(x), \quad x \in \Re^n \\ &\text{subject to} && h_j(x) = 0 \quad j = 1,2,\ldots,m \\ & && g_i(x) \le 0 \quad i = 1,2,\ \ldots,\ p \end{aligned}$$

- It is usually difficult to solve practical nonlinear programming problems analytically. In general, iterative numerical algorithms are developed to obtain solutions which may or may not produce the global optimal solution.

Nonlinear Programming Problems

- The most popular method is the sequential quadratic programming. (look at the Matlab optimization toolbox for several optimization algorithms).
- Most of the conventional methods require continuity of the objective function and the constraint functions. Sometimes, continuity of the 1st derivative may also be required.
- These are restrictive conditions and may not be always satisfied.
- There are also derivative free optimization algorithms (non-evolutionary).

Advantages of GAs

- Conceptually simple algorithms.
- As a population of solutions is used, GAs give a number of (distinct) solutions. Alternative solutions are useful in some applications such as planning, scheduling, multi-objective optimization and so on.
- The GAs do not require the search space to be
  - continuous
  - differentiable
  - unimodal (i.e. no locally optimal solutions, just the global optimum).

GAs & Traditional Optimization Algorithms

- GAs make use of a population of solution instances, not just a single point/solution in the search space. Population is a big advantage when multiple solutions are required.
- GAs use probabilistic transition rules, not deterministic rules used by most of the traditional algorithms.
- GAs work directly with the objective function, not with its gradient, nor approximated objectives nor other auxiliary information.
- *GAs usually operate on the coded variables, not directly on the decision variables themselves* (Real parameter evolutionary algorithms such as the PSO, DE, ES, can operate directly on the decision variables).

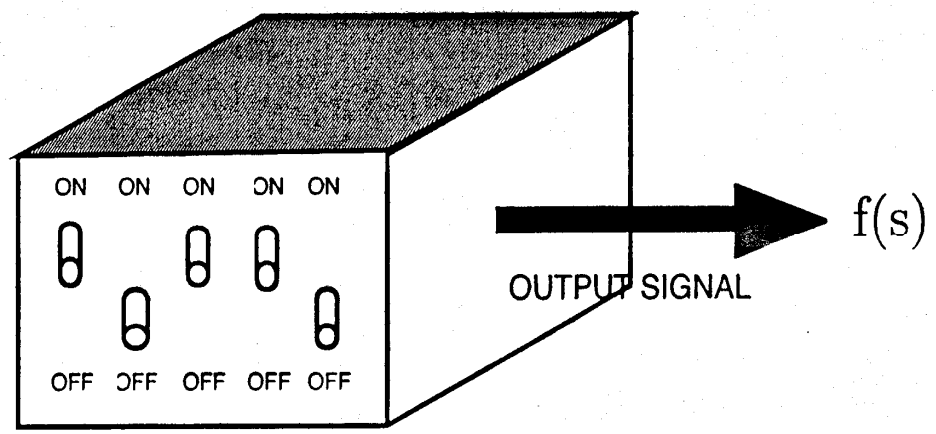
Terminology: Natural & Artificial

Natural	**Genetic Algorithm**
Chromosome	string /solution instance
Gene	feature/character/detector
Allele	feature value
Locus	string position
Genotype	structure
Phenotype	parameter set, alternative solution, a decoded structure

Not necessary to memorize all these terms.

A Simple Genetic Algorithm (SGA): A Simple Problem and solution coding

- Let us consider a simple black-box switching problem.
- There are 5 input switches and for every setting of the switches, there is an output signal *f(s)*.
- The objective is to set the switches to obtain maximum output signal value '*f(s)*'.
- The switch positions are either 'on' or 'off'. We can represent 'on' by 1 and 'off' by 0, i.e. coding of the solution.
- A particular solution of the form 11110 indicates that the first 4 switches are 'on' and the last one is 'off'.



SGA: The Initial Population

- We need to generate an initial population. We should also decide on the population size.
- The population size is generally kept constant for all generations.
- Let us assume that we choose 4 members in the population.
- The initial population is generally randomly generated, eg. By tossing a coin 20 times (4 members, each 5 bits long). Or calling a random number generator 20 times: if *rand <0.5* → *0.* If *rand >=0.5* →*1*, here *0<= rand <= 1.0.*
- Let the initial population be 01101, 11000, 01000, 10011
- Experimentally you can observe the (noticeable) influence of the initialization in the final solution.

SGA: Fitness (Objective) Values of the Initial Population

- Usually the next step is to compute the fitness (or objective) values of the randomly generated initial solution strings.
- This can be easily done by substituting each solution string into the objective function.
- Here, it is not necessary because our black-box gives the output which is also the objective function.

Number	String	Fitness	% of Total	No. selected
1	01101	169	14.4	1
2	11000	576	49.2	2
3	01000	64	5.5	0
4	10011	361	30.9	1
Total		1170	100.0	4

Table 1.1 Sample problem strings and fitness values

SGA: Evolution and Genetic Operators

- Now we define and apply simple genetic operators to the initial population, thereby evolving the initial population into successive generations while expecting the overall fitness level of the population to improve.
- The basic genetic operations are:
  - Reproduction / selection (applied to the current population)
  - Crossover (applied to the result of reproduction operation)
  - Mutation (applied to the result of crossover operation)
- These operators are applied one after the other in the given order to obtain the next generation of solution strings (in the SGA).

SGA: Reproduction

- Reproduction operator copies solution strings from the current generation into a mating pool taking into consideration the fitness or objective values of individual solution strings.
- Strings with higher fitness values will likely be represented in higher numbers in the mating pool, thereby passing on their genetic information to future generations.
- This operator is an artificial version of the Darwinian process of *natural selection*.
- Mating pool is usually of the same size as the population size.

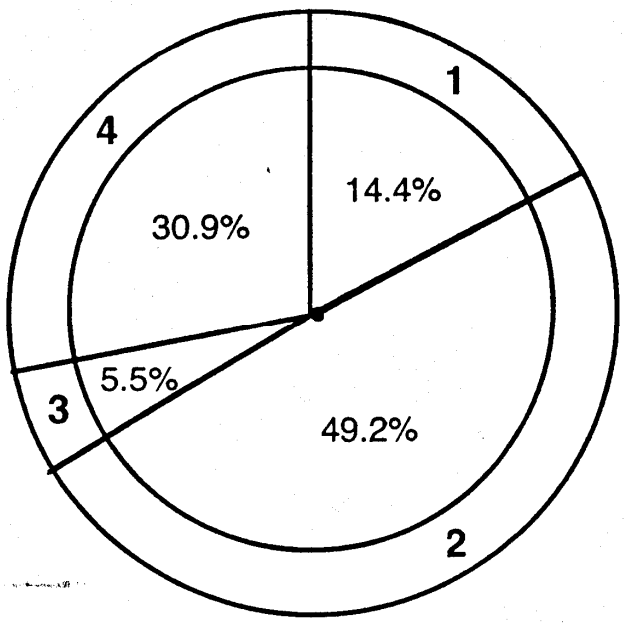
SGA: Implementation of Reproduction

- There are several alternative ways to implement the reproduction operator. In SGA, we'll use the *roulette wheel* method. We'll look at some others later on.
- In this approach, each string is assigned with a portion of the wheel's perimeter in proportion to its fitness level → fitness values should be +ve for this selection method.
- Then the wheel is spun *n* number of times to select *n* strings into the mating pool, *n* is population size. It is 4 in our simple example.
- In our simple example, assume that the fitness values have been observed from the output of the black-box and they are positive.

SGA: Roulette Wheel Selection

- Assume that the initial population has the fitness values (column 3) and the percentage fitness values (column 4) as shown in the Table 1.1 (slide 26).
- The roulette wheel shows the region assigned to every string in the population in proportion to the percentage fitness values.
- Next the roulette wheel is spun 4 times to select 4 strings into the mating pool (one particular selection is shown in column 5 in Table 1.1).

SGA: Roulette Wheel



^

| ← this is the marker

SGA: Computer Implementation of Roulette Wheel Selection

- We can use a random number (between 0 and 1) generator to simulate the operation of the roulette wheel.
- The concatenated percentage fitness values:
  - string 1: between 0 and 0.144
  - string 2: between 0.144 and 0.636
  - string 3: between 0.636 and 0.691
  - string 4: between 0.691 and 1.0
- Let the generated random numbers be: 0.127, 0.282, 0.832, 0.496. Hence, the selected strings: string 1, string 2, string 4, and string 2.

SGA: Crossover Operation

- The crossover operator is applied to the selected mating pool. According to Table 1.1, the following strings are in the mating pool: 01101, 11000, 11000, 10011.
- Crossover is performed between 2 parent strings in the mating pool and resulting in 2 offspring.
- Crossover probability p_c may be used to determine if there will be crossover between 2 parent strings.
- The 2 (usually distinct) parent strings are randomly selected from the mating pool.
- The crossover operation exchanges partial information in each string, hence the offspring have distinct parts of both parents. In this step also, there is an element of randomness.
- If the population is *n*, then repeat *n/2* times to get *n* offspring.

SGA: Crossover Operation

- Before we perform the crossover operation, we'll have to decide whether a crossover will take place.
- This decision is made by using the crossover probability p_c.
- As we'll perform crossover *n/2* times, we can generate a random vector with *n/2* elements each between 0 and 1.
- If the generated random number is smaller than p_c, we perform the crossover operation. Otherwise, no crossover is performed.
- If no crossover is performed, the selected parents are passed on as offpsrings to be processed by the mutation operator.

SGA: Crossover Operation

- After determining whether xover will take place, *One-point crossover* is performed as follows:
  - an integer position *k* is randomly selected along the string $1 <= k <= L-1$ where *L* is the length of the string. In our example, there are 4 potential one-point crossover points: 0.1.1.0.1
  - Two new strings (offsprings) are generated by breaking up the parent strings at position *k* and exchanging the information.
- If the two parent strings are **01101** and 11000 and the integer crossover position *k* is 4, then the results of crossover operation are 2 offsprings: **0110**0 and 1100**1**
- In the above example one string is underlined and the other is in bold face only to improve the understanding of the crossover operation.

Effectiveness of Reproduction & Xover

- Although the reproduction and crossover are simple operations, they can lead to a powerful search mechanism.
- We will investigate how this takes place later.

- An observation: In this SGA version, the parents are unlikely to appear in the next generation unaltered. We'll look at alternatives later on.

SGA: Mutation

- Mutation in general plays a minor role in the GAs.
- An appropriate mutation operation in our example will be to randomly change the bit values in the generated offspring strings from '0' to '1' OR from '1' to '0'.
- The mutation will introduce a degree of diversity to the population, prevent a premature convergence, help to sample unexplored regions of the search space.
- The probability of mutation operation, p_m, is small, eg. 0.001, 0.01.
- Usually the mutation probability is gradually reduced with the increase in the number of generations. In the early stages, it is necessary to sample the whole search space → need higher mutation. In the later stages, it is only a fine searching → need small mutation not to disturb the high quality solutions.

SGA: Implementation of Mutation

- There 4 strings in the population and each string is 5 bits long → 4*5 or 20 bits in total in the generated offsprings.
- The bit positions for mutation is selected randomly according to the mutation probability (0.1 for the example below).
- We can generate 20 random numbers between 0 and 1. If a generated value is smaller or equal to the mutation probability, we flip the bit value in the solution string at that position.
- Example: Let offsprings after x-over be: 01100, 11001, 11011, 10000. Random numbers: 0.26,0.34,0.46,0.59,**0.05**,0.27,0.87, 0.91,0.79,0.61,**0.09**,0.34,0.4,0.81,0.65,0.86,0.11,0.43,0.62,0.69. Offsprings after mutation: 0110**1**, 11001, **0**1011, 10000

Alternatives & Heuristic Decisions

- There are usually several alternatives available at every step, eg. reproduction, population replacement, solution representation / coding, crossover, mapping objective into fitness value, etc.
- The maximum number of generations, the size of the population, mutation / xover probability, etc. are chosen based on heuristics or experience or rules of thumb or trial-and-error.
- We'll learn some of these in the coming weeks. The best way to learn will be to perform computer simulations yourself **!!**
- Usually it is a good idea to study the traditional approaches to solving the particular problem in order to have a good start in the search for good choices in the GA-based search. Sometimes, hybrid methods (mixture of traditional & GAs) perform well.

SGA Continued … Example 2: $f(x)=x^2$

- We consider a problem of maximizing $f(x)= x^2$.
- Assume x is an integer between 0 and 31.
- We next review the SGA with more details and highlight the 6 major steps.

SGA-Step 1: Encoding the Decision Variable

- In this simple problem, we have just one integer variable restricted between 0 and 31.
- The obvious encoding will be to use 5 bits, as in the previous black-box example.

SGA - Step 2: Initial Population

- As indicated before, the initial population is generated randomly. Let us use the same initial population again.

SGA-Step 3: Computing Fitness Proportions

- The strings are converted to integers and *f(x)* are evaluated.
- The (initial) population, decoded decision variable *x*, fitness *f(x),* fitness proportions are all shown in Table 1.2.

SGA - Step 4: Reproduction

- We can use the roulette wheel to obtain the mating pool (shown in column 7 in Table 1.2)

SGA - Step 5: Crossover

- The 1st step is to select *n/2* pairs of parent strings for crossover. The randomly chosen pairs are shown in column 8 under "Mate".
- Then the one-point crossover sites are randomly chosen between 1 and 4. It is shown in column 9 as "C'over site".
- The result of crossover operation is given in column 10, as "New Popn".

String No.	Initial Popn.	x	$f(x)$	% of Total	No. Sel.	Mating Pool	Mate	C'over Site	New Popn.	x	$f(x)$
1	01101	13	169	14.4	1	01101	2	4	01100	12	144
2	11000	24	576	49.2	2	11000	1	4	11001	25	625
3	01000	8	64	5.5	0	11000	4	2	11011	27	729
4	10011	19	361	30.9	1	10011	3	2	10000	16	256
Sum			1170	100.0	4						1754
Average			293								439

Table 1.2 Simulation of a simple GA

SGA-Step 6: Mutation

- Here we assume that the mutation probability is 0.001. There are 20 bits in total in the population. Hence, 0.02 bits should mutate.
- Hence, no bits mutate and the final result (next generation population) is shown in column 10.

Fitness Values after 1st Generation

- We can compute the fitness values of the new population after decoding the strings.
- The decoded values and the fitness values are shown in column 11 and 12 in Table 1.2.
- We can observe that the average fitness has improved from 293 to 439, as the best string in the initial population appears two times in the mating pool.
- This example illustrates the process of the GA.

Evolution & Termination of the SGA

- In general, steps 1 & 2 performed just once in the beginning of the SGA.
- Steps 3, 4, 5 and 6 are repeated until a termination condition is satisfied.
- Possible termination conditions may be
  - A given number of generations has been completed
  - The fitness (i.e. the objective value of the optimization problem) of the best solution or the pool has reached an acceptable level.
  - The sequential increase in the fitness is negligible over several generations.
  - A combination of the above conditions.

Similarity Templates and Schemas

- Next we'll see how the GA performs the search effectively.
- From Table 1.1, we observe that the '1' bit in the 1st position is associated with high fitness values.
- This suggests that the 1st '1' bit has useful information to be used advantageously by the GA.
- We make use of the *similarity template* or the *schema* (plural form is schemata or schemas) to explain search process of the GA and how it makes use of such information.

Similarity Templates and Schemas

- A schema is a similarity template describing a subset of strings with similarities at certain string positions.
- We consider binary alphabet {0,1} with a wild card character *.
- We now create strings using alphabet {0, 1, *}.
- For example, schema *101* represents the subset {01010, 01011, 11010, 11011}.
- Using schema, we can represent similarities among various strings.

Similarity Templates

- If the binary string length is 5, then there are 3^5-1 different similarity templates. This is because we can have either 0 or 1 or * at each of the 5 positions.
- If there are *k* symbols in the alphabet (instead of just 0 and 1), then there are $(k+1)^l$-1 schemas.
- Obviously, there are far too many schemas than total number of actual strings. Still, schemas provide a useful way to analyze the workings of the GAs.

Similarity Templates

- A binary string of length 5, such as 10111 is a member of 2^5 schemas, as any particular position can be replaced by *.
- A binary string of length l contains 2^l schemas.
- Hence, a population of size n and binary string of length l may have somewhere between 2^l and $n.2^l$ schemas depending on the diversity of the population.
- How many of these schemas are usefully processed by the GA to improve the overall fitness level of the population?
- To answer this, we'll have to examine how various genetic operators (reproduction, crossover, mutation) affect the growth and decay of important schemas.

Similarity Templates

- The effect of reproduction is simple: highly fit schemas are copied more times into the mating pool.
- To see the effect of crossover consider two schemas: 1***0 and **11*.
- Apparently, the 1st schema is more likely to be disrupted by one-point crossover than the 2nd schema.
- The 1st schema is said to have a large *defining length*.
- The mutation rate is usually very low and it does not disrupt the schemas in a significant way.

Similarity Templates

- The observation is that highly fit schemas with short defining lengths will get propagated in increasing numbers.
- Although the GAs do not explicitly consider schemas, this process takes place while the GA processes the population.
- These highly fit schemas with short defining lengths are called building blocks.
- The number of schemas that are usefully processed in each generation is approximately n^3 where n is the population size.
- This is called *implicit parallelism*.

Chapter 2
Computer Implementation

- We implement the SGA using Matlab codes.
- We consider two simple problems:
  - Optimization of simple function of 1 unsigned integer variable coded as unsigned binary.
  - Optimization of a function of 2 variables constrained between specified values.

The next slide shows a general structure for our **SIMPLE GA**

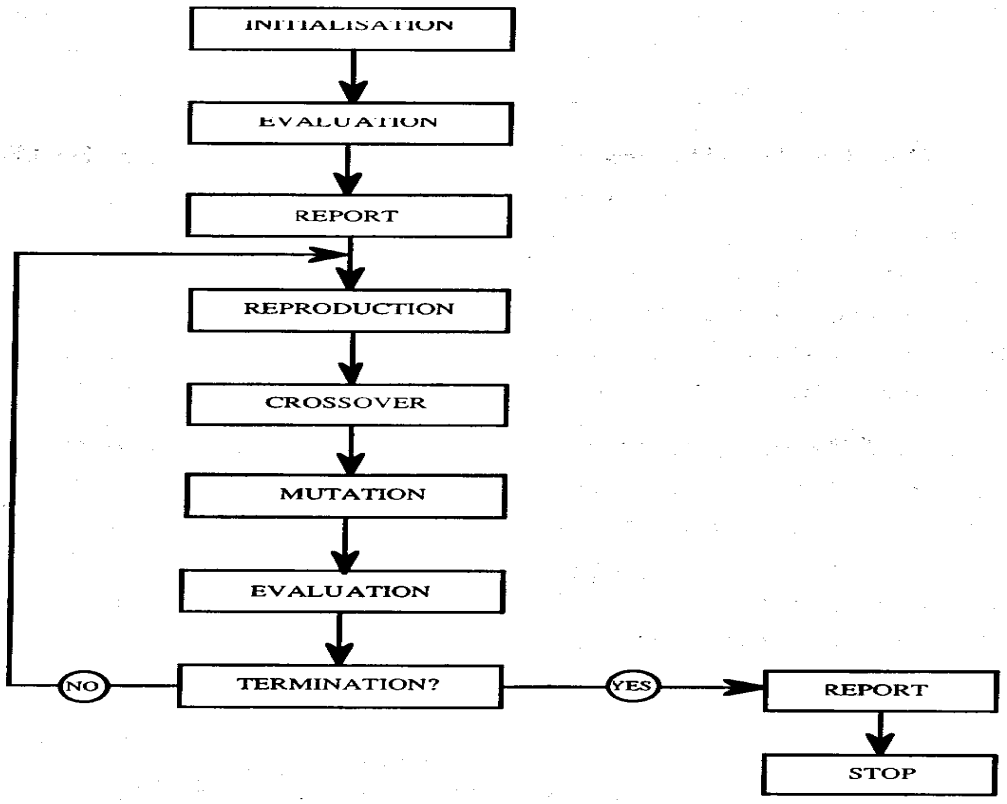


Figure 2.1 General Structure of the Simple GA

Initialisation

- In order to initialize the population we need values for
  - The number of bits in a solution candidate referred to as '*lbits*'
  - Total number of solution candidates, i.e. the population size referred to as '*popnsize*'
- Random initialization is commonly used (although there are several other alternatives)
- If discretisation is needed if variables are real valued. In this case, '*lbits*' can be computed depending on the required precision.
- The selection of '*popnsize*' influences the computation cost and the diversity of the population

Initialisation

```
function
  popn=initialise(popnsize, lbits)
% Generate the initial population

Popn=rand(popnsize, lbits) < 0.5;

% End initialise
```


Evaluation

- The next step is to evaluate the fitness of every candidate solution in the population.
- Candidate solutions have to be converted back to unsigned integers to evaluate the fitness.
- We may want to know the maximum fitness, mean fitness, the best current solution candidate, etc.
- These statistics can be saved in a file to be viewed or plotted later on, to examine the characteristics, convergence performance of the GA simulation run.

Evaluation

function [xpopn, fitness, xopt, maxf, meanf] = Evaluate(popn, lbits)

% Evaluate function returns the fitness of the current population.
% 1st this function calls the Decode() and Objfun() functions.

xpopn=Decode(popn);

fitness=Objfun(xpopn, lbits);

meanf=sum(fitness)/size(popn,1);

[maxf, imax]=max(fitness); % imax is the index of the fittest.

xopt=xpopn(imax) *%* xopt is the optimal value at current generation

% End of Evaluate function.

% We'll look at "*Decode*" and "*Objfun*" functions later.

Reproduction

- Reproduction uses the roulette wheel selection method.
- The selected strings are kept in "*select*" - the mating pool.
- The mating pool is randomly re-ordered.
- In the crossover stage, the 1st string is mated with the 2nd string and so on.

Reproduction

```
function [matingpairs, select] =
  Reproduce(popn, fitness)
% Reproduce function uses the roulette wheel selection to
%  produce mating pairs for crossover. "select" contains
%  the number of each string in the popn that had been
%  added to the mating pool.
normfit=fitness/sum(fitness);
partsum=0;
randnums=rand(size(fitness));
count(1)=0;      matepool=[];
```



```
for I=1:length(fitness)
  partsum=partsum+normfit(i);
  count(i+1)=length(find(randnums<partsum));
  select(i,1)=count(i+1)-count(i);
  matepool=[matepool;ones(select(i,1),1)*popn(i,:)];
End

% Next re-order the strings so that string 1 mated
% with string 2 and so on.

[junk, mating]=sort(rand(size(matepool,1),1));
matingpairs=matepool(mating,:);
% End of Reproduce function.
```

Crossover

- We use the simple one-point crossover as explained earlier.
- Crossover is applied to the ordered mating pool generated by the *Reproduce* function.
- Crossover sites are chosen randomly.
- There are half as many crossover sites as there are strings in the population.
- The probability of crossover taking place is p_c.

Crossover

```
function offspring=Crossover(popn,pc)
lbits=size(popn,2);
sites=ceil(rand(size(popn,1)/2,1) * (lbits-1));
sites=sites * (rand(size(sites)) < pc); % crossover probability
for j=1:length(sites),
offspring(2*j-1,:)=[popn(2*j-1,1:sites(j)) popn(2*j,
sites(j)+1:lbits)];
offspring(2*j,:)=[popn(2*j,1:sites(j)) popn(2*j-1,
sites(j)+1:lbits)];
end
% End of crossover function.
```

Mutation

* Only one bit is mutated.
* Mutation is done randomly with the mutation probability of p_m.

```
function newpopn=Mutation(offspring, pm)
 mutate=find(rand(size(offspring)) < pm);
% mutate has positions of genes to be mutated.
newpopn=offspring;
newpopn(mutate)=1-offspring(mutate);

% End of Mutation function.
```


Report

- This routine can be used to monitor the progress of the GA, and to obtain the statistics of the population as they evolve.
- This routine can be modified to suit the requirements of different applications.

```
function Report(gen, popn, xpopn, fitness,
mean, fhistory, maxfhistory, xopthistory)

disp(' '); disp('  '); disp(' ');
disp([sprintf('Generation %0.5f', gen)]);
disp('  ');
```

```
disp([‘      string        x    fitness’]);
popnstring =char(48+ popn);   % to compress the matrix
for i=1:size(popn,1),              % output current popn stats.
disp([popnstring(i,:) sprintf(%16.8g %16.8g’, xpopn(i), fitness(i))]);
end
disp(‘ ‘);
disp(‘     Fitness Histroy Statistics‘);
disp(‘Generation  Mean Fitness Max Fitness Optimal x);
history=[[0:gen]’meanfhistory maxfhistory xopthistory];
disp([sprintf(‘%5.0f %16.6g %16.6g \n’,history’)]);

% End of function Report.
```


Termination of the GA Evolutions

- The GA may be terminated after a specified number of generations
- Another possibility is the saturation of the mean or max fitness values over several generations.

Remaining Modules

- We will have to program the "*Decode*" function and the "*Objfun*".
- Both are problem specific.
- In our example, we code the single variable as an unsigned integer.

Decode function

```
function xpopn = Decode(popn)

% Decode function converts a population
% (popn) of strings from binary to integer
% assuming variables are non-negative.

lbits=size(popn,2);
twopowers=2.^(lbits-1:-1:0);
xpopn=popn*twopowers';

% End of Decode function.
```

Objective function - Objfun

- We consider the following objective function: $f(x)=(x/c)^{10}$.
- c is a scaling constant used to avoid getting too large function values.
- If *lbits*=30 & $c=2^{30}-1$; optimal $f(x)=1$ when $x=2^{30}-1$.

Objfun function

```
function fitness = Objfun(xpopn,lbits)
% Objfun evaluates the objective/fitness
% values of a decoded population
% This example is f(x)=(x/c)^10, where
% c=2^lbits-1;

c=2^lbits-1;
fitness = (xpopn/c).^10;

% End of Objfun
```

The Main Program

```
function [xpopn, fitness, meanf, maxf, xopt]=
  onevarSimpleGA(popnsize,lbits,pc,pm,numgens)
% onevarSimpleGA optimises simple function
  % of 1 variable coded as unsigned binary
  % integer
meanfhistory=[ ];
maxfhistory=[ ];
xopthistory=[ ];
% Generate the initial population
gen=0;
popn=Initialise(popnsize,lbits);
```



```
% Obtain fitness statistics for the initial population and  % save
   the history parameters for use in Report

[xpopn,fitness,xopt,maxf,meanf]=Evaluate(popn
  ,lbits);
xopthistory=[xopthistory;xopt];
maxfhistory=[maxfhistory;maxf];
meanfhistory=[meanfhistory;meanf];

% Call for a Report on the initial population
% and the initial population statistics

Report(gen,popn,xpopn,fitness,meanfhistory,ma
  xfhistory,xopthistory);
```

```
for gen=1:numgens   % Main generational loop
% Reproduce, crossover and mutate
matingpairs = Reproduce(popn,fitness);
offspring = crossover(matingpairs,pc);
popn = mutation(offspring,pm);
% Obtain statistics for the current population
% and save history parameters for Report.
[xpopn,fitness,xopt,maxf,meanf]=Evaluate(p
  opn,lbits);
xopthistory=[xopthistory;xopt];
maxfhistory=[maxfhistory;maxf];
meanfhistory=[meanfhistory;meanf];
End          % Main generational loop
```


The Main Program

```
% Call Report to printout the final
% population and a summary of the
% population stats over all generations
gen=numgens;
Report(gen,popn,xpopn,fitness,meanfhist
  ory,maxfhistory,xopthistory);
xopt=xopthistory;
maxf=maxfhistory;
meanf=meanfhistory;
% End of onevarSimpleGA
```

Running the Test Problem

- Maximise *f(x)=(x/c)^10, c=2^30-1*
- *x* is coded as unsigned binary integer.
- Some parameter values have to be specified:
- *popnsize=30, lbits=30, pc=0.6, pm=0.0333.*
- With *lbits*=30, there are 2^30 points in the solution space. Hence, exhaustive search is nearly impossible even for this simple problem.

Observations of the Results

- The random initialization gave one good candidate with fitness of 0.980787.
- Within the 10 generations, the best solution appeared in generation 7 with fitness 0.994064.
- The best solution was then destroyed by crossover, mutation, etc. in the following generation.

Generation 0

string	x	fitness
01111110101101010111011000111 0	5.3145537e+08	0.00088240779
10011111100100100010011000001	6.6928883e+08	0.0088539506
01000101011100101011111001010	2.9128699e+08	2.1587986e-06
10011100010011010110101000111 0	6.5557979e+08	0.0071987464
01010111001010011000101111010 01	3.6558513e+08	2.0935653e-05
01000110000110001010110010011 1	2.9400554e+08	2.3689532e-06
10110000110011111100110100011 11	7.4159886e+08	0.02469971
10110110110001100100011010010 0	7.6661188e+08	0.03441571
01111100000001101001001011110 0	5.202014e+08	0.00071238785
10001001000011101100111010111 1	5.7486226e+08	0.0019348141
00001011101001110001100001001 0	48875026	3.8183112e-14
01110110100111111100010100100 10	4.9754178e+08	0.00045634873
00001111100100011011000011111 1	65301567	6.9220915e-13
01010110011000001000001001010 0	3.6229135e+08	1.9124097e-05
10110000010011000000101001011 1	7.3944335e+08	0.023991112
10011110000101010110010001100 1	6.6305052e+08	0.0080624589
00000101000101110111001100001 1	21355715	9.6859555e-18
01101011101101000001110011001 1	4.5174149e+08	0.00017374052
11001011000111011110000011100 01	8.5193125e+08	0.098865961
01000001010110011000001100010 0	2.7409632e+08	1.174988e-06
11010111110010111001100001110 1	9.0511107e+08	0.18114238
10011110001011011100110000001 0	6.6345037e+08	0.0081112112
01000111010111000101110100000 1	2.9930886e+08	2.8326773e-06
10000110011110010110111010100 0	5.6402628e+08	0.0015995385
11111111100000001111101101000 0	1.0716608e+09	0.98078668
10001001001110110010010011010 1	5.7558866e+08	0.0019594023
11100011000011100111111111110 1	9.5234457e+08	0.30126066
00110011101000110010101101001 1	2.1658287e+08	1.1149164e-07
10001110000011001001001100010 1	5.9579719e+08	0.0027668512
10010001111100110010011000011 1	6.1215783e+08	0.0036277337

Fitness History Statistics

Generation	Mean Fitness	Max Fitness	Optimal x
0	0.056385	0.980787	1.07166e+09

Fitness functions

- The SGA requires the fitness values to be positive in order to be able to perform the roulette wheel selection or reproduction.
- If the objective function is -ve, then map the objective function into +ve fitness function.
- For maximization, if the objective function $u(x)$ can take –ve values, then define the fitness $f(x)$ as:

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

- $C_{\min}$ can be assigned a value initially or after the evaluation for example to the absolute value of the smallest $u(x)$ (which is –ve).

Fitness Functions

- For minimization problems, the fitness may be defined as below given *g(x)* is the objective function:

$$f(x) = \begin{cases} C_{\max} - g(x) & \text{if } C_{\max} - g(x) > 0 \\ 0 & \text{otherwise} \end{cases}$$

- Here the C_{max}, can be assigned a value initially or assigned to the largest of *g(x)* in the current population.

Fitness Scaling

- The simple GA may exhibit an undesirable behavior, in that, in the 1st few generations highly fit individuals to dominate the selection.
- This may lead to a premature convergence.
- After some generations, the maximum fitness and the mean fitness of the population will be similar leading to essentially a random selection & search.
- We can employ fitness scaling to deal with these problems.

Linear Fitness Scaling

- Given non-negative fitness $f(x)$, the scaled function: $g=af+b$
- Use heuristics to obtain values for a and b
  - maintaining average fitness value before and after scaling $g_{avg}=f_{avg}$.
  - $g_{max}=Cf_{avg}$ to restrict the fittest candidate to have for example, atmost 2 samples in the mating pool by setting $C=2$.

$$a = \frac{f_{avg}(C-1)}{f_{\max} - f_{avg}} \qquad b = f_{avg}(1-a)$$

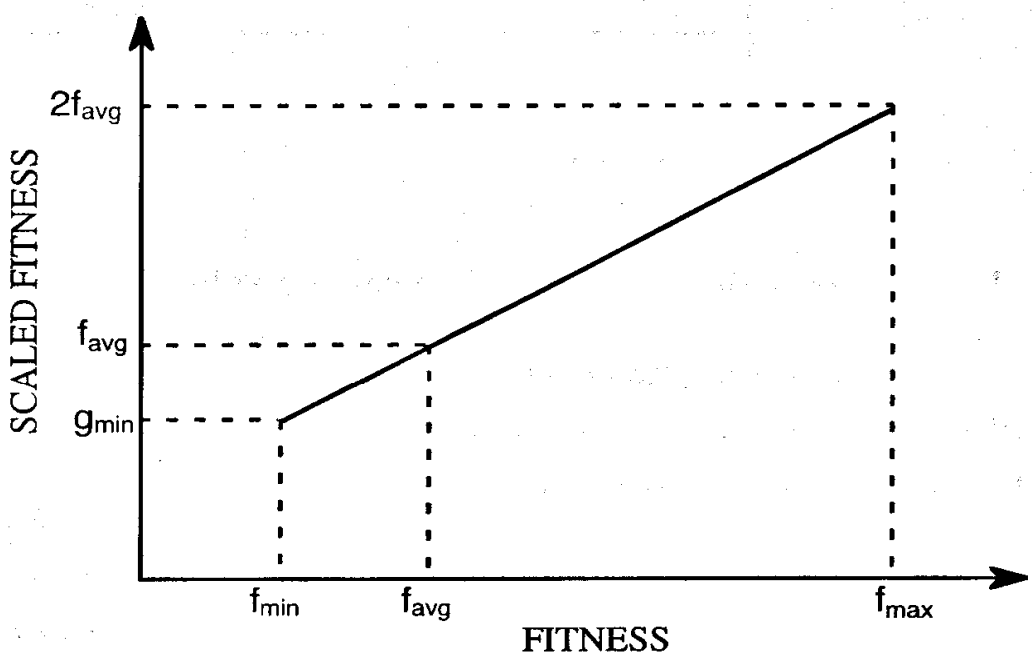


Figure 2.2 Linear scaling under normal conditions

Fitness Scaling

- The scaling may produce –ve fitness in later generations when many individuals have fitness values similar to the best individual.
- One possible solution is to maintain the average as the same before and after scaling, but map the min fitness to 0, instead of scaling the max fitness by *C*.

$$a = \frac{f_{avg}}{f_{avg} - f_{\min}} \qquad b = f_{avg}(1 - a)$$

- Fitness scaling is done within the Reproduce for selection only.

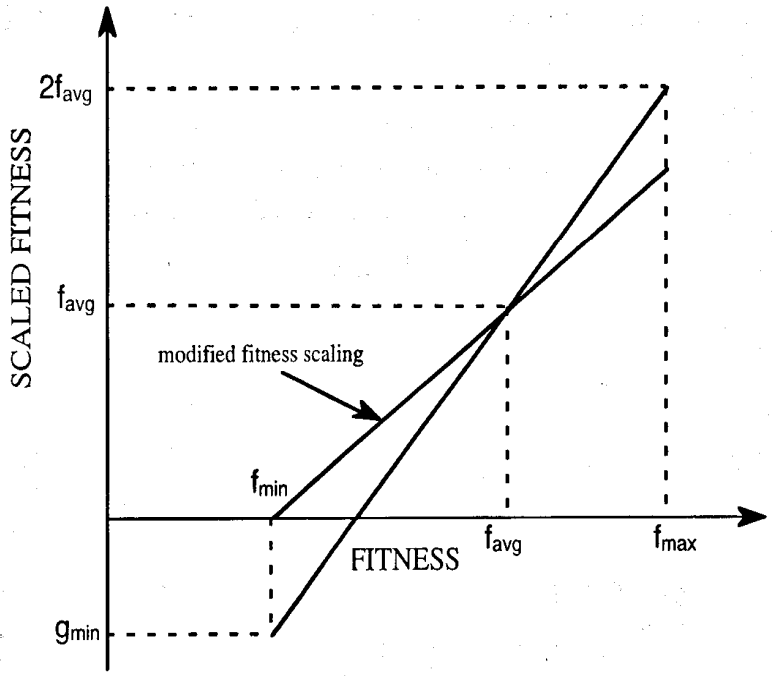


Figure 2.3 Linear scaling with negative scaling values and the modified function

```
Function
  scfitness=Scalefitness(fitness,fmultiple)
% To linearly scale the fitnesses. Returns the results
%  in scfitness. First calculate the parameters a and b
favg=sum(fitness)/length(fitness);
[fmax,i]=max(fitness);
[fmin,j]=min(fitness);
a=favg*(fmultiple-1)/(fmax-favg);
b=favg*(1-a);
if a*fmin+b<0
 a=favg/(favg-fmin);
 b=favg*(1-a);
end
scfitness=a*fitness+b*ones(size(fitness));
% End of Scalefitness function.
```


Sigma Truncation

- This method can be used before applying the linear scaling
- Makes use of variance and average of the fitness values as follows: $f' = f - (f_{avg} - c\sigma)$

  where f_{ave} is fitness average, σ is the variance, f' is the fitness after sigma truncation. c is a scale factor usually set around 2-3.
- Any –ve f' is set to zero.
- Linear scaling can be applied to f' .

Power Law Scaling

- The scaled fitness f' is a power of +ve raw fitness f, eg:

$$f' = g(f) = f^{\alpha}$$

.

- Alpha should be chosen appropriately:
  - If $0 \leq f < 1$ and if you wish to increase the separation, then $0 \leq \alpha < 1$.
  - If $1 < f$, and you wish to increase separation between fitness values, then $1 < \alpha$.
- During the early generations, maybe necessary to reduce the variations in the fitness values.
- Closer to termination, maybe necessary to increase the variations in the fitness values.

Coding Considerations

- Some problems may have natural coding.
- Generally there are many possible coding schemes.
- Selecting a suitable coding:
  - Give preference to short low-order schemas that are relevant to the problem
  - Use the smallest number of alphabets (0 & 1 for binary).
- The 1st objective is generally difficult to satisfy. We may attempt to re-order the string of coding.
- The 2nd point leads to naturally the binary coding – the smallest alphabet.

Coding Multivariable Problems

- Consider coding a decision variable

$$x = (x_1, x_2, ..., x_m) \text{ where } a_i \le x_i \le b_i$$

- Each component will be coded as a binary string of length l_i.
- The length is determined by the required precision.
- If we require 5 decimal point precision on variable x_i, then the l_i is chosen as the smallest integer satisfying $10^5 (b_i - a_i) \le 2^{l_i} - 1$

Multivariable Problems

- With this coding, the decoding is:

$$x_i = a_i + \text{decimal}(\text{string}_i)\left(\frac{b_i - a_i}{2^{l_i} - 1}\right)$$

- The simple GA has to be modified:
  - Te input *lbits* now a vector with components l_i, $i=1,...,m$
  - The Decode function has to be changed.
  - The main function should be able to have additional inputs in the form of upper and lower bounds as follows:

```
function [xopn,fitness,meanf,maxf,xopt]=
  multivarSimpleGA(popnsize,lbits,pc,pm,num
  gens,vlb,vub)
```

```
function xpopn =
 Decode(popn,lbits,vlb,vub)
% To convert a population (popn) of strings from  binary to real.
% Total string length is sum(lbits)  with m=length(lbits)substrings
%  each for  x_1,...,x_m variables. 1st decode each substring to
% an unsigned decimal  integer: xint
index1=1;   index2=0;
for i=1:length(lbits)
 index2=index2+lbits(i);
 twopowers=2.^(lbits(i)-1:-1:0);
 xint(:,i)=popn(:,index1:index2)*twopowers';
 index1=index1+lbits(i);
end
```


Decode Function

```
% Now calculate the x values
factor=(vub-vlb)./(2.^lbits-1);
xpopn=ones(size(popn,1),1)*vlb+xint*dia
  g(factor);
% End of Decode function with
% multiple variables
```

- **Some other functions may also have to be changed for multivariable case.**

Example

- Consider the following problem:

maximise $f(x_1, x_2) = 21.5 + x_1 \sin(2\pi x_1) + x_2 \sin(10\pi x_2)$

subject to $-3.0 \le x_1 \le 12.1$ and $4.1 \le x_2 \le 5.8$

- Assume the required precision of 4 decimal points to get l_1=18 and l_2=15.
- *popn=10,* p_c*=0.5,* p_m*=0.01 numgens=10*
- Better results can be obtained by increasing the numgens to 100 or so.

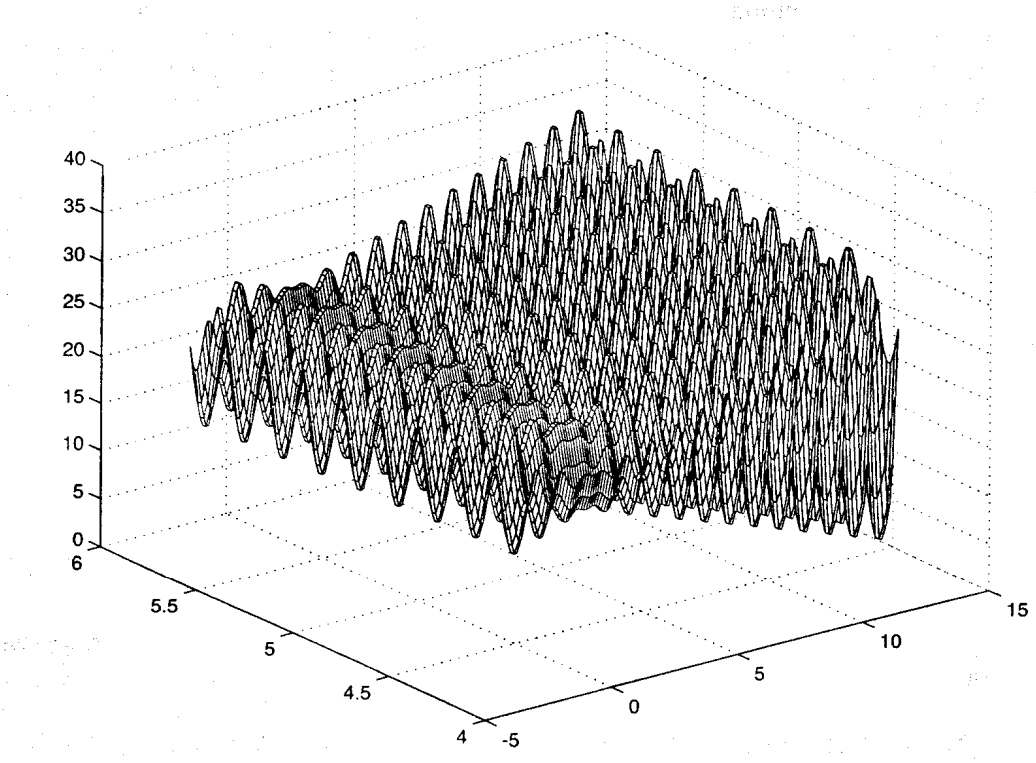


Figure 2.4 Mesh plot of the function $f(x_1, x_2)$

Further Investigations

- Investigate the effects of mutation rate, crossover, rate, etc. on the quality of the obtained solution.
- Need to keep all variables constant, except the one being investigated, changed gradually over some ranges.
- Need to evaluate and obtain statistics on several different random initialization for each particular parameter set.
- Keep the same initialization for different parameters sets.

Function Optimisation

- Improvements to ‘the simple GA’
- Function optimization
- Constrained optimization

Improvements to Selection Process

- The roulette wheel selection may not yield the expected numbers, if the population is small -- which is fairly common.
- It is possible that the fitter individuals may dominate at a much higher proportion → leading to premature convergence.
- Alternatively, stochastic universal selection (SUS) can be used.
- In SUS, the wheel is rotated once with *n* markers on the wheel. These *n* markers will give *n* strings. Gives good approximation within ± 1.

Stochastic Universal Selection

- This method can be implemented easily by replacing *randnums=rand(size(fitness));* by

  rr=rand;

  spacing=1/length(fitness);

  *randnums=sort(mod(rr:spacing:1+rr-0.5*spacing,1);*

Ranking Selection

- Ranking selection is another alternative where the population is sorted according to fitness values and each string is assigned with an offspring count.
- If we sort the population with the best first, the selection proportion to the k^{th} ranked string is:

$$p_k = q_{\max} - \left(\frac{k-1}{n-1}\right)(q_{\max} - q_{\min})$$

q_{max} and q_{min} are the allocation for the best and the worst strings respectively. q_{max} and q_{min} satisfy the following relationship because the summation of the proportions should be 1: $q_{max}+q_{min}=2/n$

Tournament Selection

- There are several variations to this method.
- In one version, 2 (or more) strings are randomly picked, their fitness values compared, the best one is included in the mating pool and all the strings are returned to the population.
- This process is repeated until the necessary number of strings are selected.
- Ranking and tournament are better than the roulette wheel method.

Improvements to Crossover

- The crossover operator is the main distinguishing feature of the GA.
- Its main function is to combine various building blocks.
- The one-point crossover used in the simple GA has a serious limitation -- it cannot combine all possible schemas.
- For instance, 1-point crossover cannot combine 11******1 and ****11*** to produce 11**11**1.

1 and 2 Point Crossover

- 1-point crossover is able to combine short low order schemas. But we may not know in advance what ordering of bits will group functionally related bits together.
- One possible solution is to use 2-point crossover.
- 2-point crossover is less likely to disrupt long schemas.

2-Point Crossover

- 2- point crossover can combine more schemas than the 1-point crossover.
- For example apply the 2-point crossover to the following two schemas with 2 randomly chosen crossover sites at 4 and 8: 11**|****|*1 and ****|1*1*|** to give 11**1*1**1 and ********** as the offspring schemas.
- However, there are schemas that may not be combined by even the 2-point crossover.

Uniform Crossover

- The uniform crossover has the ability combine almost all schemas.
- There is no crossover site, instead, for each bit position in the children, we randomly decide which parent to contribute:

Parent 1	1001011
Parent 2	*0101101*
Template:	1101001
Offspring 1	10*0*1*1*0*1*
Offspring 2	*01*0*1*01*1*

randomly generated 'Template' with equal probability to have 0 and 1.

Uniform Crossover

- For uniform crossover, the positions of the relevant bits (schema defining length) in the strings are not important.
- 1-point and 2-point crossovers are likely to preserve the good properties coded compactly.
- It is difficult to say which crossover is the best. It is problem & coding dependent.
- Generally for small populations, uniform crossover is more suitable.
- If the population is large, then 2-point crossover may be sufficient.

Improvements to Population Replacement

- In the simple GA, one entire generation is replaced by the offsprings.
- This is sometimes referred to as non-overlapping generations, ie. No overlapping between parents and children.
- It is possible sometimes the offsprings to be worse than their parents, as observed in some of the previous results.
- The main disadvantage is that some good building blocks in the parents may be lost and would not contribute to the evolution process.

Overlapping Generations and Elitism

- There are several population replacement methods proposed, essentially to overlap the new and old populations (i.e. offsprings and parents).
- Sometimes the best fitness value of the current generation can become smaller than the past generation(s).
- The *elitist* strategy fixes this problem by just copying the best or best few parents into the next generation to replace the worst children or some children randomly.
- Although the elitist strategy may lead to one or two fit strings dominating the population, still in most cases it improves the simple GA and gives better results.

Steady-State Reproduction

- Even with the elitist strategy, many best individual may not reproduce and the genes may be lost.
- So, replace just a few in the population in every generation – called steady-state reproduction.
- A small number of offsprings are generated and they replace the weak parents in the population.
- Steady-state reproduction is commonly used in evolving rule-based systems where incremental learning is important.

$\mu + \lambda$ Selection

- In this scheme, parents and the offsprings compete for survival into the next generation.
- μ offsprings and λ parents are chosen as parents in the next generation.
- If is λ very small, then it is the elitist strategy.
- Another special case may be equal number of μ and λ.

One advantage is that we can have a high mutation probability, a disadvantage is that this method cannot perform well in a changing environment,

De Jong and Function Optimisation

- De Jong did some pioneering work in function minimization problem domain.
- Function F1 has the minimum at origin ***x=0***.
- F2 (Rosenbrock's banana function) has global minimum at (1,1).
- F3 is a 3D step with the minimum at –15.
- F4 has the global optimum near origin ***x=0***, where *N(0,1)* is 0 mean 1 variance white noise.
- F5 has 25 very sharp troughs of similar depth. a_{ij} are randomly chosen between [-65.536, 65.536].

Number	Function	Limits
F1	$f_1(\boldsymbol{x}) = \sum_{i=1}^{3} x_i^2,$	$-5.12 \leq x_i \leq 5.12$
F2	$f_2(\boldsymbol{x}) = 100(x_1^2 - x_2)^2 + (1 - x_1)^2,$	$-2.048 \leq x_i \leq 2.048$
F3	$f_3(\boldsymbol{x}) = \sum_{i=1}^{3} \text{integer}(x_i),$	$-5.12 \leq x_i \leq 5.12$
F4	$f_4(\boldsymbol{x}) = \sum_{i=1}^{30} i x_i^4 + N(0,1),$	$-1.28 \leq x_i \leq 1.28$
F5	$f_5(\boldsymbol{x}) = 0.002 + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^{2} (x_i - a_{ij})^6},$	$-65.536 \leq x_i \leq 65.536$

Table 4.1 Test functions for minimisation

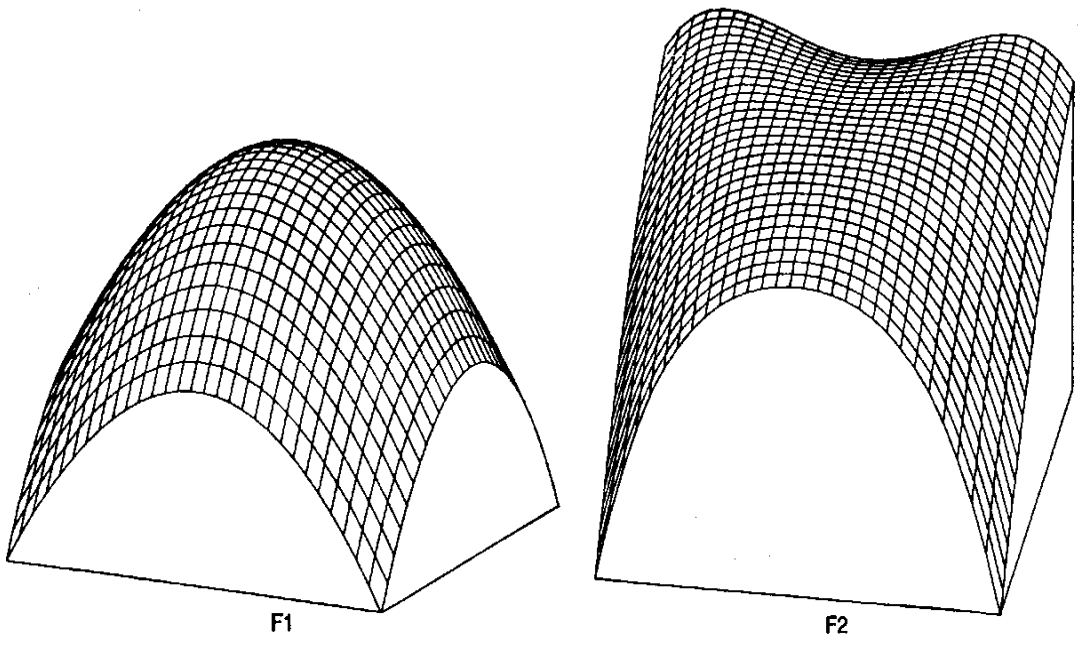
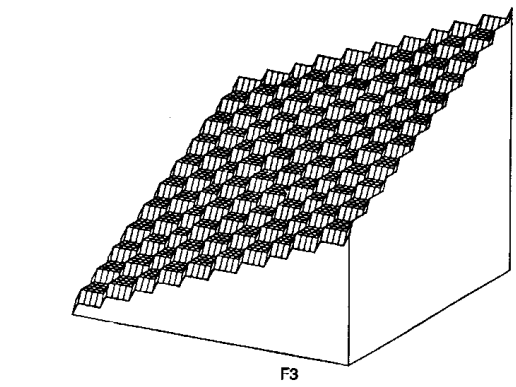


FIGURE 4.9 Inverted, two-dimensional versions of De Jong's (1975) test functions *F1* and *F2*. Reprinted by permission.



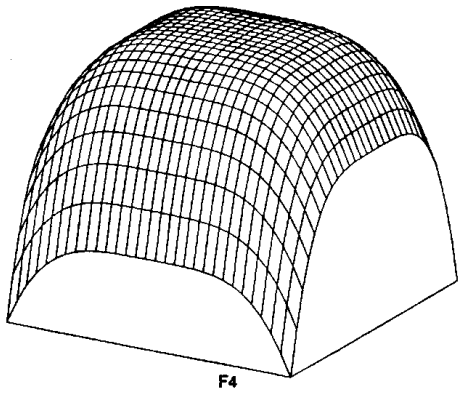


FIGURE 4.10 Inverted, two-dimensional versions of De Jong's (1975) test functions *F3* and *F4*. Reprinted by permission.

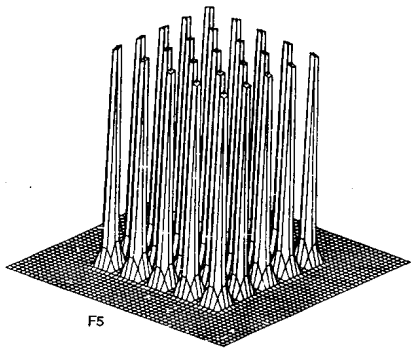


FIGURE 4.11 Inverted version of De Jong's (1975) test function *F5*. Reprinted by permission.

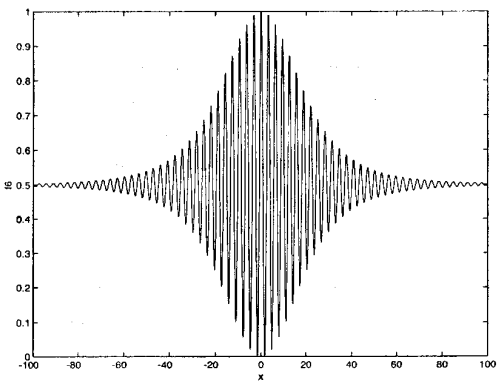


Figure 4.1 The test function $f6$ when y is constant ($y = 0$)

De Jong and Function Optimisation

- All the functions are positive.
- We should negate the functions to convert the minimization into a maximization problem.
- We should add an offset to make the fitnesses +ve.
- De Jong considered 2 performance measures:

$$J_{on-line} = \frac{1}{T}\sum_{t=0}^{T} \bar{f}(t) \quad [\bar{f}(t) \text{ is the mean fitness at generation } t]$$

$$J_{off-line} = \frac{1}{T}\sum_{t=0}^{T} f_{\max}(t) \quad [f_{\max}(t) \text{ is the max fitness at generation } t]$$

Experimental Results

- Results are given in the extended notes (page 6).
- Simple GA was tested first. Then crossover, selection, population replacement, etc. are improved to obtain better results.
- GA performs well in comparison to several conventional non-linear optimization algorithms on multimodal functions.
- Not so well on unimodal functions. GA should not be the choice for unimodal functions.
- You may run the Matlab programs and explore more !!

Davis' Function optimisation

- Davis' function:

$$f^6(x,y) = 0.5 - \frac{(\sin\sqrt{x^2+y^2})^2 - 0.5}{(1+0.001(x^2+y^2))^2}$$

- It has several hills and the maximum is at (0,0), the maximum occupies only a tiny fraction of the total area.
- Results are given for several cases: 1. The simple GA (Table 4.2), 2. SGA with fitness scaling (T. 4.3), 3. SGA with linear scaling and uniform xover (T 4.4), 4. SGA with the elitist replacement (T 4.5), 5. SGA with $\mu + \lambda$(T 4.6 & 4.7). Table numbers refer to the extended notes.

Fitness History Statistics

Generation	Mean Fitness	Max Fitness	Optimal x	
0	0.503697	0.867592	4.64513	-11.7565
1	0.507463	0.867592	4.64513	-11.7565
2	0.514698	0.820131	-15.0977	-4.47166
3	0.498547	0.820131	-15.0977	-4.47166
4	0.501326	0.821075	-15.0977	-4.40943
5	0.502437	0.821075	-15.0977	-4.40943
6	0.500878	0.921811	-8.65238	-3.71592
7	0.513566	0.921811	-8.65238	-3.71592
8	0.52791	0.921811	-8.65238	-3.71592
9	0.545438	0.921811	-8.65238	-3.71592
10	0.54032	0.921811	-8.65238	-3.71592
11	0.578093	0.921811	-8.65238	-3.71626
12	0.555383	0.921811	-8.65238	-3.71626
13	0.595485	0.921811	-8.65238	-3.71626
14	0.63703	0.921811	-8.65238	-3.71626
15	0.649057	0.921811	-8.65238	-3.71511
16	0.6869	0.960978	-2.63069	-5.74782
17	0.707684	0.960978	-2.63069	-5.74782
18	0.742898	0.921811	-8.65238	-3.71459
19	0.745931	0.921811	-8.65238	-3.71473
20	0.740784	0.921811	-8.65238	-3.71473
21	0.765929	0.921811	-8.65238	-3.71473
22	0.789503	0.921811	-8.65276	-3.71363
23	0.828096	0.921811	-8.65276	-3.71363
24	0.813929	0.921811	-8.65238	-3.71473
25	0.830831	0.921811	-8.65238	-3.71473
26	0.848045	0.921811	-8.65286	-3.71363
27	0.858572	0.921811	-8.65286	-3.71363
28	0.853368	0.921811	-8.65286	-3.71363
29	0.861227	0.921811	-8.65286	-3.71363
30	0.855864	0.921811	-8.65238	-3.71473
31	0.851469	0.921811	-8.65238	-3.71502
32	0.858286	0.921811	-8.65238	-3.71463
33	0.866796	0.921811	-8.65238	-3.71363
34	0.869285	0.921811	-8.65391	-3.71153
35	0.87443	0.921811	-8.65391	-3.71153
36	0.868587	0.921811	-8.65391	-3.71153
37	0.903035	0.921811	-8.65247	-3.71363
38	0.898683	0.921811	-8.65276	-3.71363
39	0.878667	0.921811	-8.65247	-3.71363
40	0.853164	0.921811	-8.65247	-3.71363

Table 4.2 Fitness statistics for 40 generations of the simple GA

Simple GA

Function *f6*

22 bits for *x*

22 bits for *y*

p_c=0.65;

P_m=0.004

40 generations

Population=100

Best soln in generation 16

Fitness History Statistics

Generation	Mean Fitness	Max Fitness	Optimal x	
0	0.503697	0.867592	4.64513	-11.7565
1	0.508326	0.867592	4.64513	-11.7565
2	0.509802	0.867592	4.64513	-11.7565
3	0.53039	0.868933	4.64513	-11.7443
4	0.561234	0.921625	5.82707	7.37765
5	0.59649	0.921625	5.82707	7.37765
6	0.672782	0.950617	5.21481	-3.69651
7	0.752719	0.921625	5.82707	7.37765
8	0.78568	0.921633	5.82707	7.37803
9	0.821143	0.921633	5.82707	7.37803
10	0.785656	0.921811	5.82707	7.39653
11	0.763477	0.950584	5.215	-3.69651
12	0.780457	0.950549	5.215	-3.6968
13	0.801794	0.950617	5.21481	-3.69651
14	0.852029	0.950617	5.21481	-3.69651
15	0.854064	0.953766	5.21481	-3.66867
16	0.822871	0.962424	5.21481	-3.52895
17	0.841983	0.962424	5.21481	-3.52895
18	0.818125	0.962424	5.21481	-3.52895
19	0.819909	0.962424	5.21481	-3.52895
20	0.831519	0.962427	5.21472	-3.52895
21	0.862256	0.962627	5.21481	-3.51675
22	0.904606	0.962775	5.1904	-3.52895
23	0.873061	0.962775	5.1904	-3.52895
24	0.86957	0.962775	5.1904	-3.52895
25	0.883483	0.962775	5.1904	-3.52895
26	0.899208	0.962776	5.19059	-3.52895
27	0.864422	0.962776	5.19059	-3.52895
28	0.85123	0.962776	5.19059	-3.52895
29	0.878127	0.962776	5.1904	-3.53048
30	0.867648	0.962776	5.1904	-3.53048
31	0.899257	0.962776	5.1904	-3.53048
32	0.892281	0.962776	5.1904	-3.53048
33	0.905438	0.962776	5.1904	-3.53048
34	0.880897	0.962776	5.1904	-3.53048
35	0.834948	0.962776	5.1904	-3.53048
36	0.901784	0.962776	5.1904	-3.53029
37	0.914267	0.962776	5.19097	-3.52933
38	0.907366	0.962776	5.19097	-3.52933
39	0.887017	0.962776	5.19097	-3.52933
40	0.885118	0.962776	5.19097	-3.52933

Table 4.6 Fitness statistics for 40 generations choosing the best offspring and parents to form the next generation - first run

121

Selection is

$\mu = \lambda$

$Pc=0.75,$

$Pm=0.01$

Fitness History Statistics

Generation	Mean Fitness	Max Fitness	Optimal x	
0	0.498652	0.890776	3.13227	-5.76427
1	0.49634	0.887869	3.14448	-5.76427
2	0.515782	0.890055	3.13532	-5.76427
3	0.542739	0.93341	3.14448	-5.63891
4	0.568121	0.960416	3.14448	-5.37436
5	0.583212	0.955109	3.14753	-1.0844
6	0.646628	0.960549	3.14448	-5.37603
7	0.744575	0.95005	3.14448	-5.5682
8	0.78392	0.956143	3.14457	-1.08421
9	0.842531	0.989737	3.14753	-0.303149
10	0.842104	0.990168	3.14753	-0.107837
11	0.839353	0.98977	3.14753	-0.29552
12	0.891083	0.990168	3.14753	-0.107074
13	0.867687	0.990168	3.14753	-0.107074
14	0.891209	0.990169	3.14753	-0.106311
15	0.858183	0.990265	3.13227	-0.107837
16	0.878565	0.99027	3.13227	-0.123954
17	0.927549	0.99027	3.13227	-0.123954
18	0.889455	0.990284	3.13456	-0.151134
19	0.938775	0.990284	3.13685	-0.107837
20	0.939114	0.990284	3.1369	-0.107837
21	0.935044	0.990284	3.13456	-0.156665
22	0.943409	0.990284	3.13685	-0.107837
23	0.930018	0.990284	3.1369	-0.100732
24	0.971233	0.990284	3.13685	-0.0987768
25	0.936272	0.999751	0.0120401	-0.0101805
26	0.92873	0.999751	0.0120401	-0.0101805
27	0.935169	0.990284	3.13685	-0.10097
28	0.935309	0.998658	0.0120401	-0.0345945
29	0.919122	0.999845	0.0120401	-0.0030756
30	0.938004	0.999751	0.0120401	-0.0101805
31	0.937849	0.99974	0.0120401	-0.010705
32	0.909272	0.999844	0.0120401	-0.00326634
33	0.928413	0.999861	0.00593662	-0.0101805
34	0.938213	0.999849	0.0116587	-0.00388622
35	0.934315	0.999845	0.0120401	-0.0030756
36	0.943323	0.999844	0.00841618	-0.0092268
37	0.943836	0.999849	0.0116587	-0.00388622
38	0.929207	0.999849	0.0116587	-0.00388622
39	0.946492	0.999907	0.00879765	-0.00388622
40	0.954651	0.999914	0.00841618	-0.00388622

Table 4.7 Fitness statistics for 40 generations choosing the best offspring and parents to form the next generation - second run

122

Chapter 4
Mathematical Analysis of SGA

- Similarity templates and schemas
- The fundamental theorem
- The two-armed and k-armed bandit problem
- Implicit parallelism and building blocks
- The minimal deceptive problem

Schema or Similarity Templates

- Schema or similarity template represents a subset strings with similarities at certain string positions.
- With the binary alphabet of 0 and 1, we introduce the wild card character '*'.
- Strings may be created now with {0,1,*}.
- Schema *101* represents the following strings: {01010, 11010, 01011, 11011}.
- Schema is a compact way to represent similarity between strings.

Schemas

- If the string length, *l*, is 5, then there are 3^5 different similarity templates, larger than the total number of strings: 2^5.
- For alphabets with *k* symbols, there are (*k*+1)^*l* schemas.
- Any particular binary string of length *l* is a member of 2^*l* schemas. The string 10111 may take its actual value or '*' at each of the 5 position giving 2^5 schemas.
- A population with *n* members and length *l* may have schemas between 2^*l* and *n*.2^*l* depending on the population diversity.

Survival & Propagation of Schemas

- Due to reproduction highly fit schemas are likely to be reproduced in increasing numbers.
- 1-point Crossover is likely to destroy schema 1****1, than ***11*.
- Hence shorter schemas which are also highly fit get propagated generation to generation in increasing numbers.
- These short highly fit schemas are known as the building blocks.

Order of Schema

- If *H* is a schema taken from the 3 letter alphabet {0,1,*}, then the order of *H* is denoted by *o(H)* which is the number of fixed positions in the string.
- Order of schema 0 1 1 * 1 1 * is 5 and the order of schema 0 * * * * * * * is 1.
- An obvious observation is that a lower order schema is more likely to survive crossover and mutation.

Defining Length of *H*

- Defining length of schema *H* is the distance between the first and last fixed string positions denoted as $\delta(H)$.
- Eg. The defining length of 011*11** is $\delta(H)$ =6-1=5 and the defining length of 1******* is $\delta(H)$ =1-1=0.
- Let *m(H,t)* represent the number of examples of a particular schema *H* in the population *A(t)* at time *t*.

The Fundamental Theorem

- Consider the individual and combined effects of reproduction, crossover and mutation on schemas in a population of strings.
- The combined result is known as the *Schema Theorem* or *The Fundamental Theorem of Genetic Algorithms*.
- Short & low order schemas with above population-average fitness will receive exponentially increasing numbers in the subsequent generations.

Two-armed Bandit Problem

- Suppose a gambler has N coins to play a slot machine with 2 arms.
- The arms have mean payoffs and variances as follows:

$$\mu_1, \mu_2, \sigma_1, \sigma_2$$

- How to maximise the total payoff given the gambler does not know the values of these parameters before hand.
- This is an important problem in statistical decision theory.

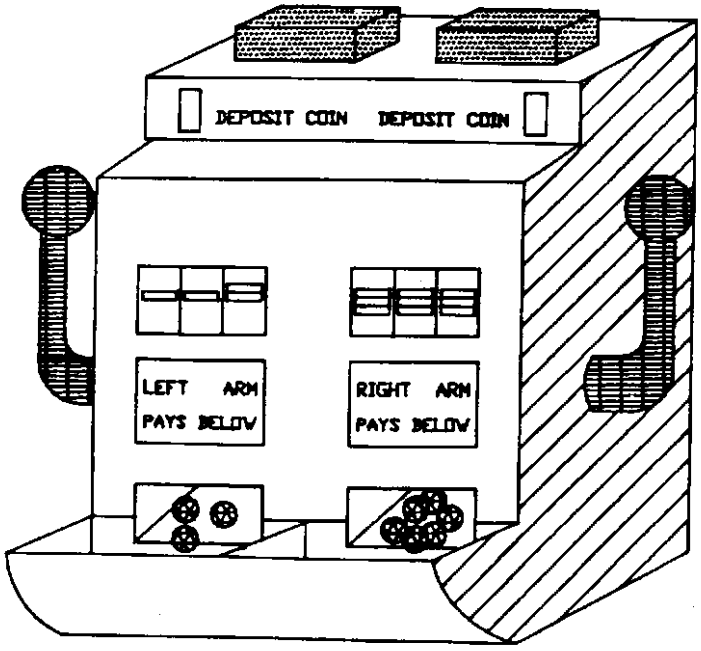


FIGURE 2.1 The two-armed bandit problem poses a dilemma: how do we search for the right answer (exploration) at the same time we use that information (exploitation)?

Two-armed Bandit Problem

- Let us allocate *n* coins to each arm in order to identify the best arm.
- Allocate the remaining *N-2n* coins to the best arm.
- However, there is a probability *q(n)* that our identification of the best arm was incorrect.
- The expected loss after *N* trials, in relation to allocating all *N* to the best arm:

$$L(N,n) = n\,|\,\mu_1 - \mu_2\,| + q(n)(N-2n)\,|\,\mu_1 - \mu_2\,|$$

Two-armed Bandit Problem

- The above expression has two components: (a) *n* trials have been definitely allocated to the worst arm. (b) the remaining *N-2n* trials are allocated to the best arm. *q(n)* is the probability that our selection of the "best arm" was incorrect.
- Determine the *n* to minimise the *L(N,n)*. The differential:

$$\frac{dL}{dn} = [1 - 2\,q(n) + (N - 2n)\frac{dq(n)}{dn}] \mid \mu_1 - \mu_2 \mid = 0$$

- To solve this we need to know the *q(n)*.

Two-armed Bandit Problem

- It is a bit complicated. So we go for the final result.
- The optimal *n* referred to as *n** is approximated by:

$$n^* \approx c \ln\left(\frac{N^2}{8\pi c^2 \ln(N^2)}\right) \quad \text{where } c = \left(\frac{\sigma_1}{\mu_1 - \mu_2}\right)^2$$

- From the above expression we obtain:

$$N \approx e^{\frac{n^*}{2c}} \sqrt{8\pi c^2 \ln(N^2)}$$

Two-armed Bandit Problem

- The number of trials given to the best arm:

$$N - n^* \approx e^{\frac{n^*}{2c}} \sqrt{8\pi c^2 \ln(N^2)}$$

- *The interpretation is that the the best arm to be allocated exponentially increasing number of trials.*
- The results of *k*-armed bandit problem is similar to that of 2-armed bandit problem.
- In GA there are several multi-armed bandit problems being solved simultaneously.

Multi-armed Bandit Problem and GAs

- Consider the following 8 schemas:

  * 0 0 * 0 * *
  * 0 0 * 1 * *
  * 0 1 * 0 * *
  * 0 1 * 1 * *
  * 1 0 * 0 * *
  * 1 0 * 1 * *
  * 1 1 * 0 * *
  * 1 1 * 1 * *

  This can be compared with an 8-armed bandit problem.

Multi-armed Bandit Problem and GAs

- These 2^3=8 schemas compete with one another to be selected in the reproduction stage.
- Just as in the *k*-armed bandit problem, we allocate exponentially increasing numbers to the best schema.
- In GA several problems proceed in parallel.
- For example, with string length 7 and schema order 3, there are $\binom{7}{3}$=35 different 8-armed problems.
- In general, for schema order *j* and string length *l*, there are $\binom{l}{j}$ different *2^j*-armed bandit problems.

Implicit Parallelism

- Each individual with string length *l* has *2^l* schemas.
- The population may have schemas between *2^l* and *n.2^l*.
- Hence, in each generation up to *n.2^l* schemas are processed in parallel – this is called *implicit parallelism*.
- Although the GA processes the *n* string in the population at each generation, it is estimated that approximately *n^3* schemas are processed in each generation.
- The computational power of GAs comes from this implicit parallelism.

Building Block Hypothesis

- Schemas with long defining length are likely to be destroyed by crossover.
- Short, lower order and highly fit schemas get sampled at exponentially increasing rates.
- These schemas are known as the *building blocks*.
- The GA performs computation by recombining the building blocks to form better and better strings – this is called the *building block hypothesis*.
- There are problems where this hypothesis may be violated.

GA-Deceptive Problems

- If good points/solutions are surrounded by bad points / solutions, then it will be difficult for the GA to achieve the optimum.
- These functions & codings are known as *GA-deceptive*.
- It is possible that a particular problem may have GA-deceptive coding as well as non-deceptive.
- Next, we will attempt to construct a problem that violates the building block hypothesis. That is short, lower order highly fit schemas to lead to incorrect longer high-order schemas.

The minimal Deceptive Problem

- No order –1 problems are GA-deceptive
- The smallest possible GA-deceptive problem is an order – 2 problem – called the *minimal deceptive problem* (MDP).
- Consider a set of 4 order – 2 schemas shown below:

* * * 0 * * * * * 0 * f_{00} is the schema average fitness value.

* * * 0 * * * * * 1 * f_{01} is the schema average fitness value.

* * * 1 * * * * * 0 * f_{10} is the schema average fitness value.

* * * 1 * * * * * 1 * f_{11} is the schema average fitness value.

- **Defining length is $\delta(H)$.**

The minimal Deceptive Problem

- Now our objective is to try to deceive the GA to give a wrong answer.
- Assume that f_{11} is the schema associated with the global optimal solution and the fitness f_{0*} of the order – one schema 0* is greater than the fitness f_{1*} of the order – one schema 1*, that is

$$\frac{f_{00} + f_{01}}{2} > \frac{f_{10} + f_{11}}{2}$$

The minimal Deceptive Problem

- Given f_{11} corresponds to the global optimum solution, we can deduce the following:

$$f_{10} < f_{01} \quad \text{and} \quad f_{10} < f_{00}$$

- There are 2 possible relationships between f_{01} and f_{00}:

$$\text{Type I}: \quad f_{01} > f_{00}$$

$$\text{Type II}: \quad f_{01} \leq f_{00}$$

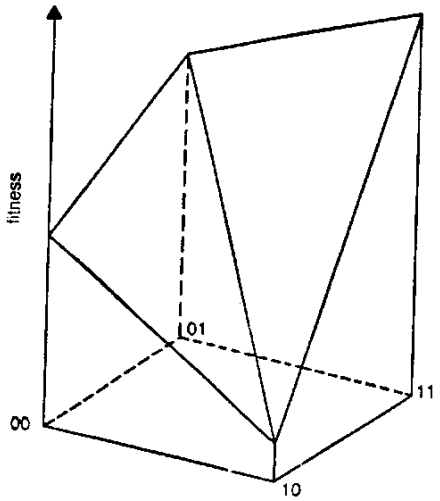


FIGURE 2.8 Sketch of Type I, minimal deceptive problem (MDP) $f_{01} > f_{00}$.

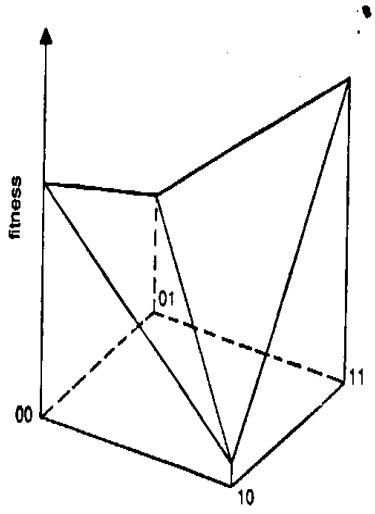


FIGURE 2.9 Sketch of Type II, minimal deceptive problem (MDP) $f_{00} > f_{01}$.

The minimal Deceptive Problem

- Next we will construct difference equation relationships for the expected portions of each schema with 0.0 mutation probability.
- The schema theorem gives only a lower bound assuming all crossovers may destroy the schema.
- Here we can calculate a better estimate by considering all crossover pairs as shown in the table:

Table: Results of 1-Point Crossover

- ‘S’ means that both schemas survive the 1-point crossover, irrespective of the crossover point.
- Both schemas can be destroyed only if (a) the schemas are complement of each other and (b) the crossover site is in between the fixed positions.
- The new schemas created as shown in the table along the diagonal is for the case (b) above.

Diagonal shows the result of 1-point xover when the xover site is between the fixed positions.

S – the same schema(s) obtained by 1-point crossover irrespective of the crossover site.

	0 0	0 1	1 0	1 1
0 0	S	S	S	0 1 1 0
0 1	S	S	0 0 1 1	S
1 0	S	0 0 1 1	S	S
1 1	0 1 1 0	S	S	S

The minimal Deceptive Problem

- The proportions of schemas at time t is denoted as: $P_{00}^t,\ P_{01}^t,\ P_{10}^t,\ P_{11}^t$
- The probability that a crossover site falling between the two fixed bit positions is: $p_c \dfrac{\delta(H)}{l-1}$
- If we assume the crossover probability to be 1, then: $p_c \dfrac{\delta(H)}{l-1} = 0.6$

Deriving Population Proportions at time *t+1*

- Consider the proportion of schema '00'.
- This proportion always survive except (a) when mated with schema '11' with the crossover site between the fixed positions, this '00' schema is destroyed, and (b) when schemas '10' and '01' mate with the crossover site between the fixed positions, this '00' schema is created. Gives:

$$P_{00}^{t+1} = \frac{f_{00}}{\bar{f}} P^{t}{}_{00} - 0.6 \frac{f_{00}}{\bar{f}} P_{00}^{t} \frac{f_{11}}{\bar{f}} P_{11}^{t} + 0.6 \frac{f_{01} f_{10}}{\bar{f}^{2}} P_{01}^{t} P_{10}^{t}$$

Deriving Population Proportions at time *t+1*

- Alternatively, consider all pairs of schemas and identify the proportion of schema '00' after x-over:

00	00	$2 \frac{f_{00}}{\bar{f}} P^{t}{}_{00} \frac{f_{00}}{\bar{f}} P^{t}{}_{00}$
00	01	$\frac{f_{00}}{\bar{f}} P^{t}{}_{00} \frac{f_{01}}{\bar{f}} P^{t}{}_{01}$
00	10	$\frac{f_{00}}{\bar{f}} P^{t}{}_{00} \frac{f_{10}}{\bar{f}} P^{t}{}_{10}$
01	10	$0.6 \frac{f_{01}}{\bar{f}} P^{t}{}_{01} \frac{f_{10}}{\bar{f}} P^{t}{}_{10}$
00	11	$\frac{f_{00}}{\bar{f}} P^{t}{}_{00} \frac{f_{11}}{\bar{f}} P^{t}{}_{11} - 0.6 \frac{f_{00}}{\bar{f}} P^{t}{}_{00} \frac{f_{11}}{\bar{f}} P^{t}{}_{11}$

Continued .....

Deriving Population Proportions at time *t+1*

- Other pairs to be considered to compute the proportion of schema '00' at time *t+1* are: (01 00), (10 00), (10 01), (11 00)
- Also remember that for a population of *n*, the crossover operation is performed only *n/2* number of times.
- Hence, the factor 2 will be cancelled when the proportions are computed.

The minimal Deceptive Problem

- The proportions of 4 schemas in generation *t+1* are:

$$P_{00}^{t+1} = \frac{f_{00}}{\bar{f}} P_{00}^{t} (1 - 0.6 \frac{f_{11}}{\bar{f}} P_{11}^{t}) + 0.6 \frac{f_{01} f_{10}}{\bar{f}^2} P_{01}^{t} P_{10}^{t}$$

$$P_{01}^{t+1} = \frac{f_{01}}{\bar{f}} P_{01}^{t} (1 - 0.6 \frac{f_{10}}{\bar{f}} P_{10}^{t}) + 0.6 \frac{f_{00} f_{11}}{\bar{f}^2} P_{00}^{t} P_{11}^{t}$$

$$P_{10}^{t+1} = \frac{f_{10}}{\bar{f}} P_{10}^{t} (1 - 0.6 \frac{f_{01}}{\bar{f}} P_{01}^{t}) + 0.6 \frac{f_{00} f_{11}}{\bar{f}^2} P_{00}^{t} P_{11}^{t}$$

$$P_{11}^{t+1} = \frac{f_{11}}{\bar{f}} P_{11}^{t} (1 - 0.6 \frac{f_{00}}{\bar{f}} P_{00}^{t}) + 0.6 \frac{f_{01} f_{10}}{\bar{f}^2} P_{01}^{t} P_{10}^{t}$$

The minimal Deceptive Problem

- $\bar{f}$ is the average population fitness in generation *t* and given by:

$$\bar{f} = f_{00}P_{00}^t + f_{01}P_{01}^t + f_{10}P_{10}^t + f_{11}P_{11}^t$$

- Simulate the expressions on the previous slide for the variations in the schema numbers over generations.
- Normalise schema fitnesses with respect to f_{00}. For example when we specify f_{11}=1.1, it means $f_{11}=1.1f_{00}$.

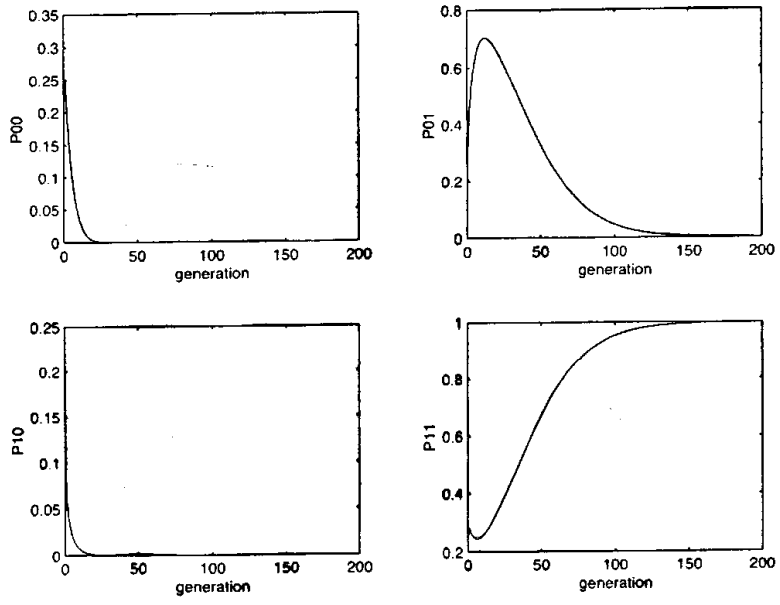


Figure 3.1 Type 1: $f_{01} = 1.05$, $f_{10} = 0$, $f_1 = 1.1$ and equal initial proportions.

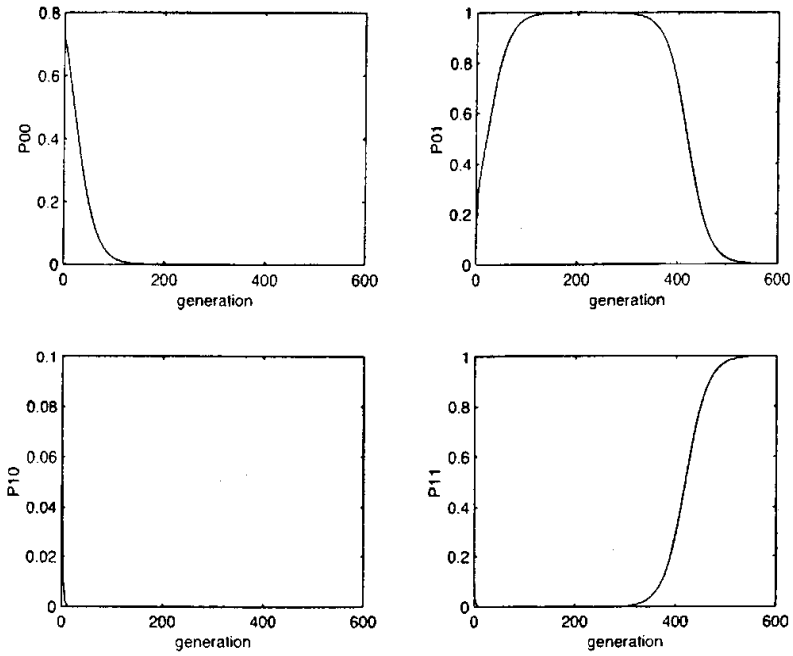


Figure 3.2 Type 1: $f_{01} = 1.05$, $f_{10} = 0$, $f_1 = 1.1$ and unequal initial proportions.

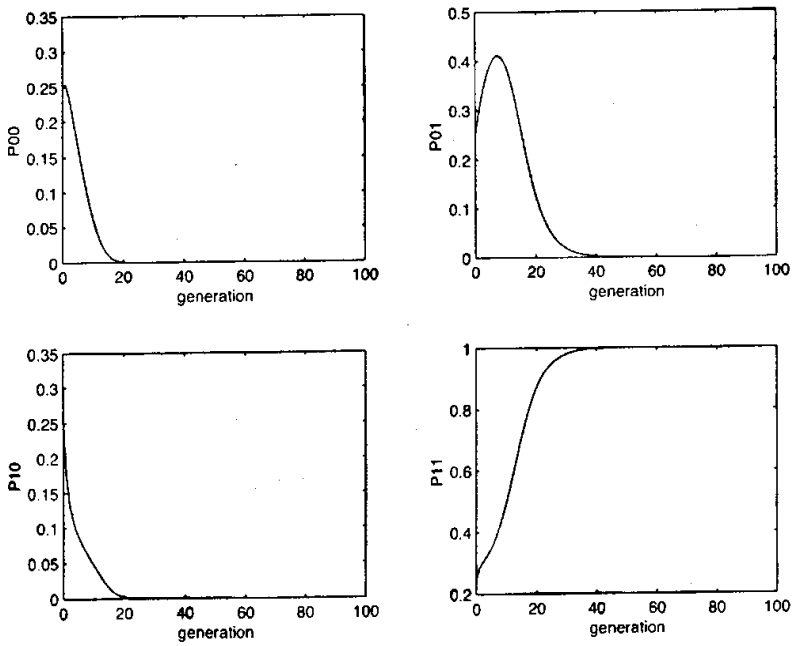


Figure 3.3 Type 2: $f_{01} = 0.9$, $f_{10} = 0.5$, $f_1 = 1.1$ and equal initial proportions.

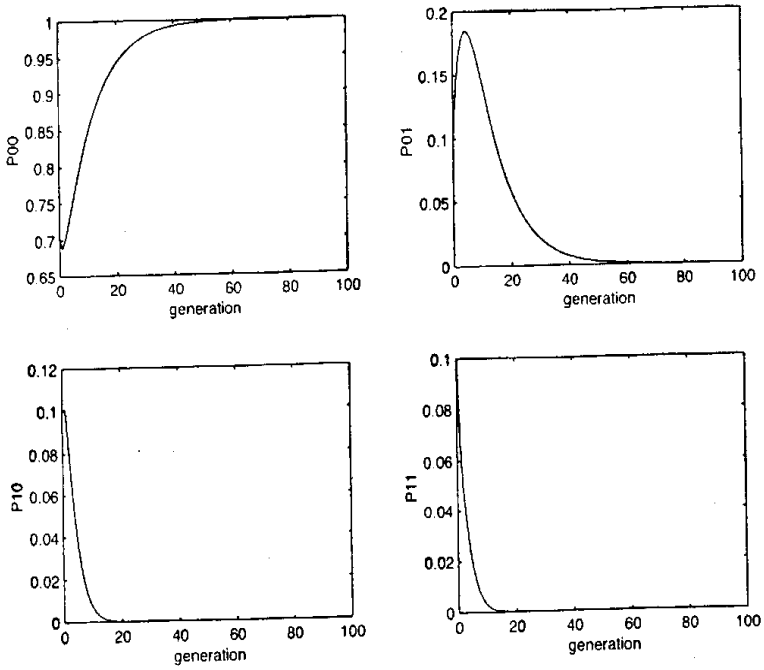


Figure 3.4 Type 2: $f_{01} = 0.9$, $f_{10} = 0.5$, $f_1 = 1.1$ and unequal initial proportions.