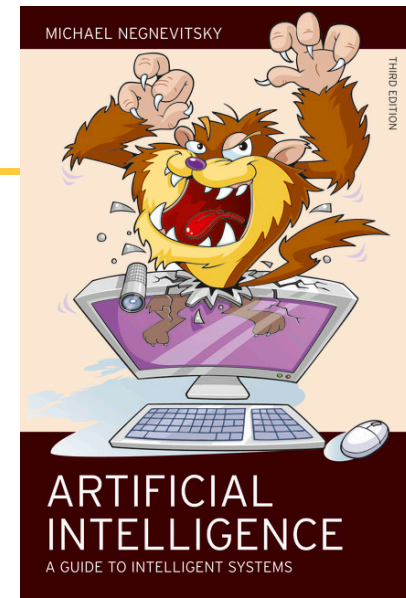


# Lecture 7



## Evolutionary Computation (EC)

[Michael Negnevitsky. 2011. *Artificial Intelligence A Guide to Intelligent Systems*. 3<sup>rd</sup> Ed. Pearson Education Canada. ISBN: 1408225743.]

## **1. Introduction**

2. Simulation of Natural Evolution

3. Genetic Algorithms (GAs)

4. Why Genetic Algorithms Work

5. Case Study

6. Evolutionary Strategies

7. Genetic Programming (GP)

# 1. Introduction

---

- Intelligence can be defined as the capability of a system to adapt its behavior to an ever-changing environment.
- According to Alan Turing, the form or appearance of a system is irrelevant to its intelligence.
- Evolutionary computation (EC) simulates evolution on a computer.
- The result of evolution simulation on a computer is a series of optimization algorithms, usually based on a simple set of rules.

# 1. Introduction

---

- Optimization iteratively improves the quality of solutions until an optimal, or at least feasible, solution is found.
- The evolutionary approach is based on computational models of natural selection and genetics.

# 1. Introduction

---

- We call these models **evolutionary computation** (EC), an umbrella term that combines **genetic algorithms** (GAs), **evolution strategies**, and **genetic programming** (GP).
- All these techniques simulate evolution by using the processes of **selection**, **mutation**, and **reproduction**.

# Contents

---

1. Introduction

**2. Simulation of Natural Evolution**

3. Genetic Algorithms (GAs)

4. Why Genetic Algorithms Work

5. Case Study

6. Evolutionary Strategies

7. Genetic Programming (GP)

## 2. Simulation of Natural Evolution

---

- On 1 July 1858, Charles Darwin presented his **theory of evolution** before the Linnean Society of London.
- This day marks the beginning of a revolution in biology.
- Darwin's classical **theory of evolution**, together with Weismann's **theory of natural selection** and Mendel's concept of **genetics**, now represent the neo-Darwinian paradigm.

## 2. Simulation of Natural Evolution

---

- **Neo-Darwinism** is based on processes of **reproduction**, **mutation**, **competition**, and **selection**.
- The power to reproduce appears to be an essential property of life.
- The power to mutate is also guaranteed in any living organism that reproduces itself in a continuously changing environment.



## 2. Simulation of Natural Evolution

---

- Processes of **competition** and **selection** normally take place in the natural world, where expanding populations of different species are limited by a finite space.
- Evolution can be seen as a process leading to the maintenance or increase of a population's ability to survive and reproduce in a specific environment. This ability is called **evolutionary fitness**.

## 2. Simulation of Natural Evolution

---

- **Evolutionary fitness** can also be viewed as a measure of the organism's ability to anticipate changes in its environment.
- The **fitness**, or the quantitative measure of the ability to predict environmental changes and respond adequately, can be considered as the quality that is being optimized in natural life.

## How is a Population with Increasing Fitness Generated?

---

- Let us consider a population of rabbits. Some rabbits are faster than others, and we may say that these rabbits possess superior fitness because they have a greater chance of avoiding foxes, surviving and then breeding.
- If two parents have superior fitness, there is a good chance that a combination of their genes will produce an offspring with even higher fitness.

## How is a Population with Increasing Fitness Generated?

---

- Over time the entire population of rabbits becomes faster to meet their environmental challenges in the face of foxes.

## 2. Simulation of Natural Evolution

---

- All methods of EC simulate natural evolution by **creating** a population of individuals, **evaluating** their fitness, **generating** a new population through **genetic operations** (**crossover** and **mutation**), and **repeating** this process a number of times.
- There are different ways of performing EC.

## 2. Simulation of Natural Evolution

---

- We will start with **genetic algorithms** (GAs) as most of the other evolutionary algorithms can be viewed as variations of GAs.
- In the early 1970s, John Holland introduced the concept of GAs.
- His aim was to make computers do what nature does.
- Holland was concerned with algorithms that manipulate strings of binary digits.

# Genetic Algorithms (GAs)

- Each artificial chromosome consists of a number of ‘genes’, and each gene is represented by 0 or 1.

1	0	1	1	0	1	0	0	0	0	0	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- Nature has an ability to adapt and learn without being told what to do.
- Nature finds good chromosomes blindly.
- GAs do the same as nature.

# Genetic Algorithms (GAs)

---

- Two mechanisms link a GA to the problem it is solving: **encoding** (i.e., representing chromosomes as bit strings) and **evaluation**.
- An **evaluation function** is used to measure the chromosome's performance (i.e., **fitness**) for the problem to be solved.
- The GA uses a measure of **fitness** of individual chromosomes to carry out **reproduction**.



# Genetic Algorithms (GAs)

---

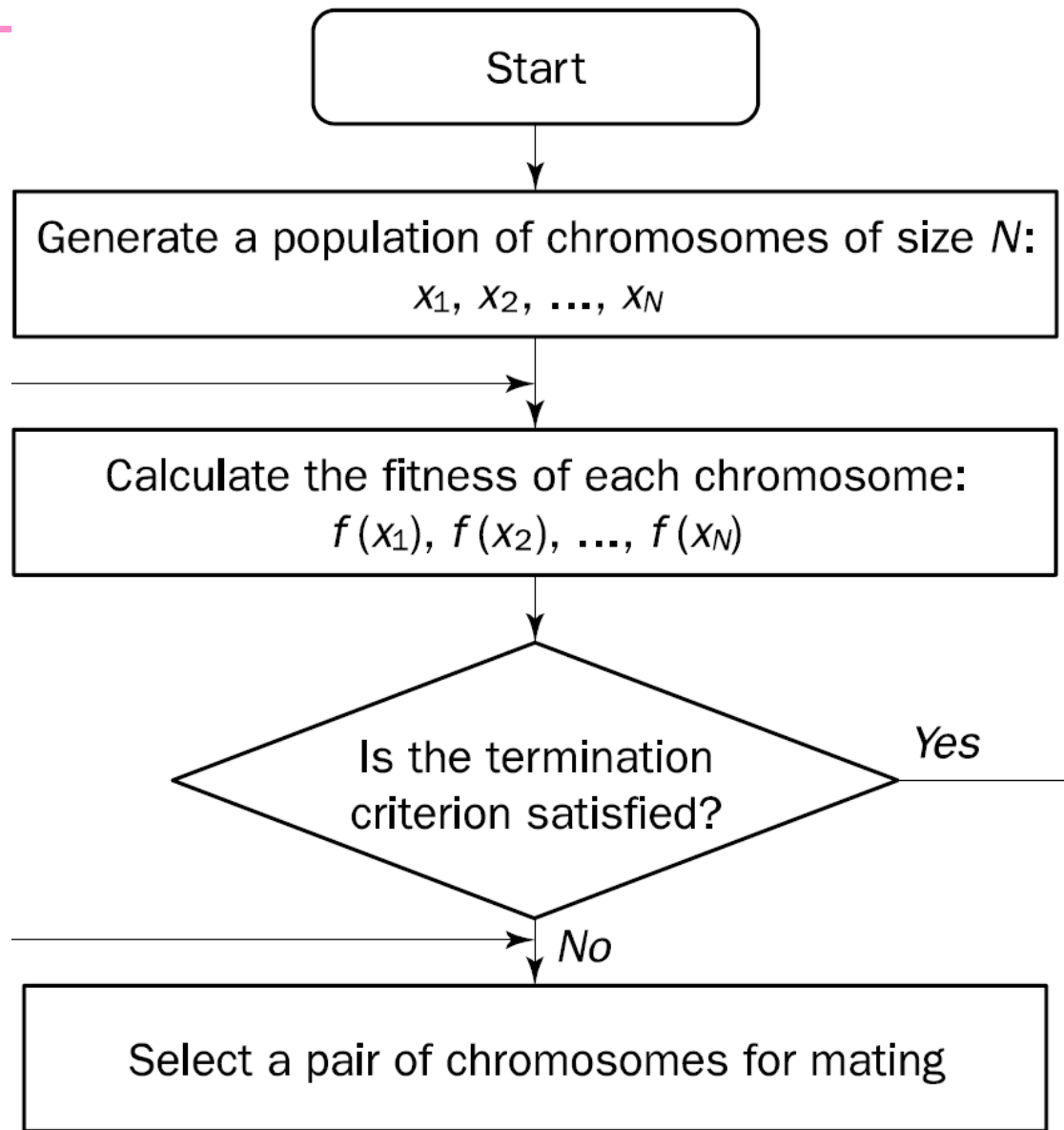
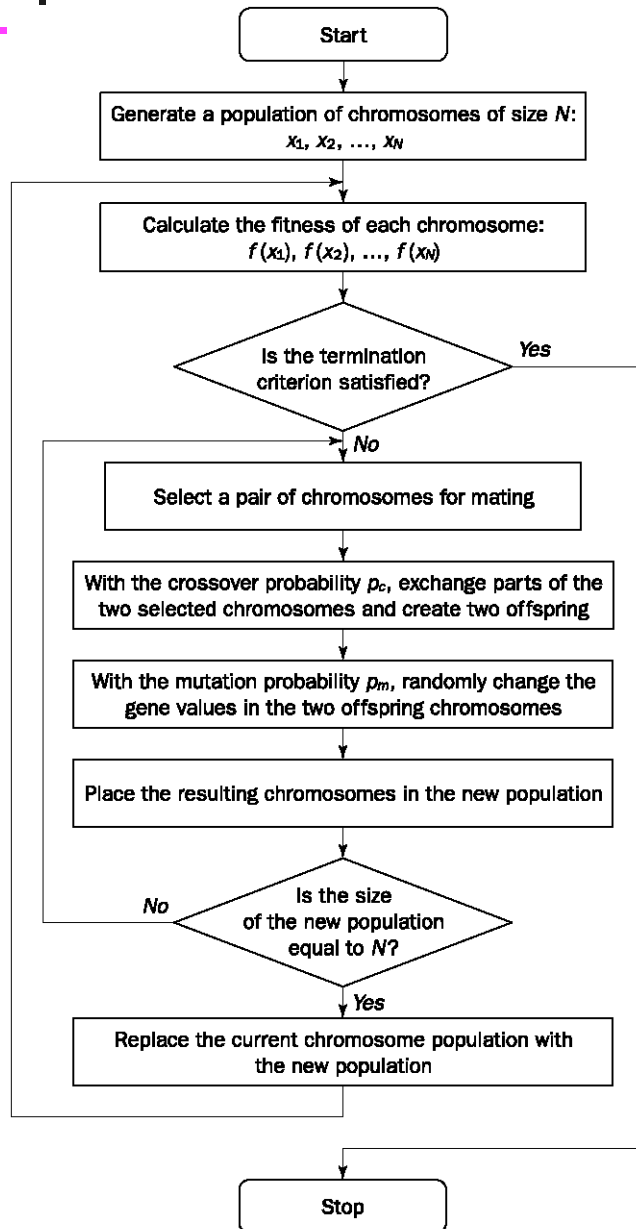
- As reproduction takes place, the **crossover** operator exchanges parts of two single chromosomes, and the **mutation** operator changes the gene value in some randomly chosen location of the chromosome.
- After a number of successive reproductions, the less fit chromosomes become extinct, while those best able to survive gradually come to dominate the population.

# Contents

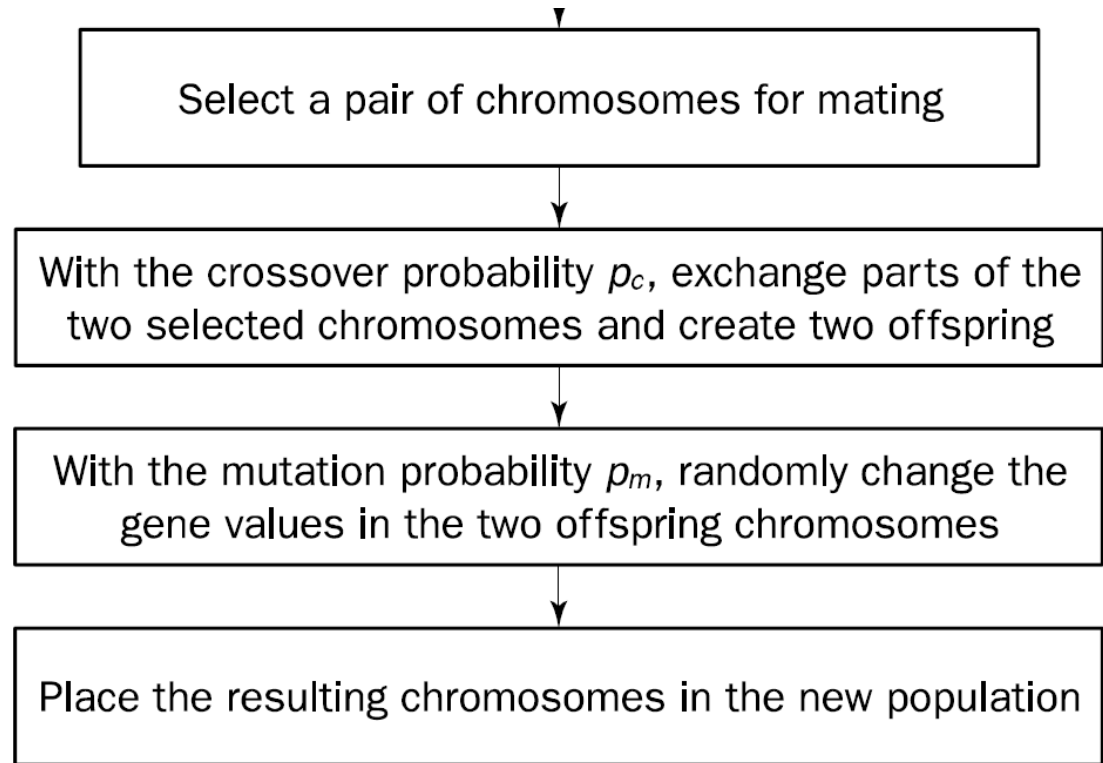
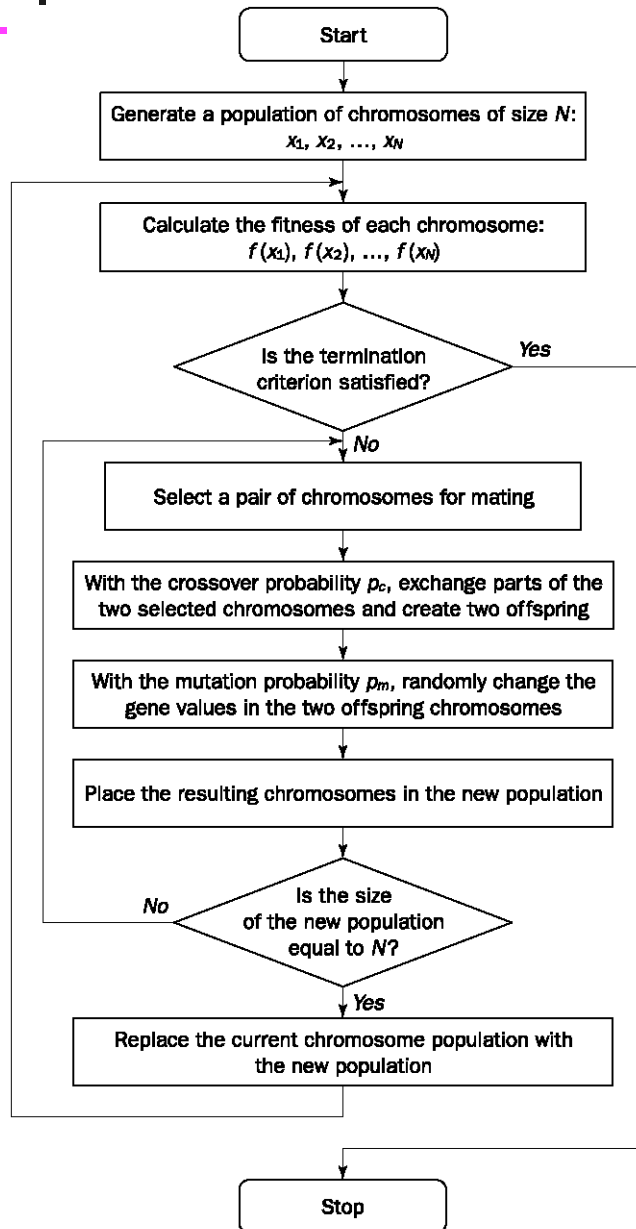
---

1. Introduction
2. Simulation of Natural Evolution
- 3. Genetic Algorithms (GAs)**
4. Why Genetic Algorithms Work
5. Case Study
6. Evolutionary Strategies
7. Genetic Programming (GP)

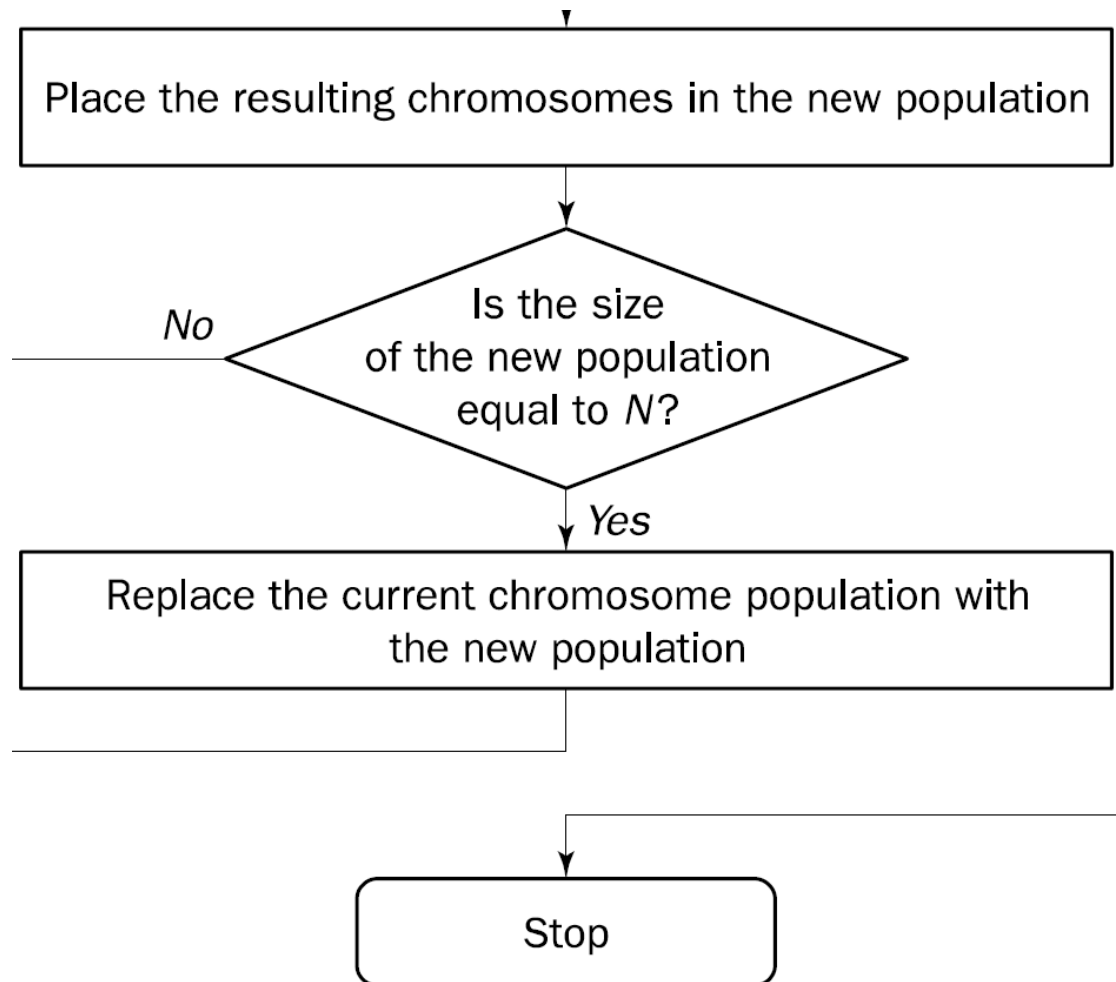
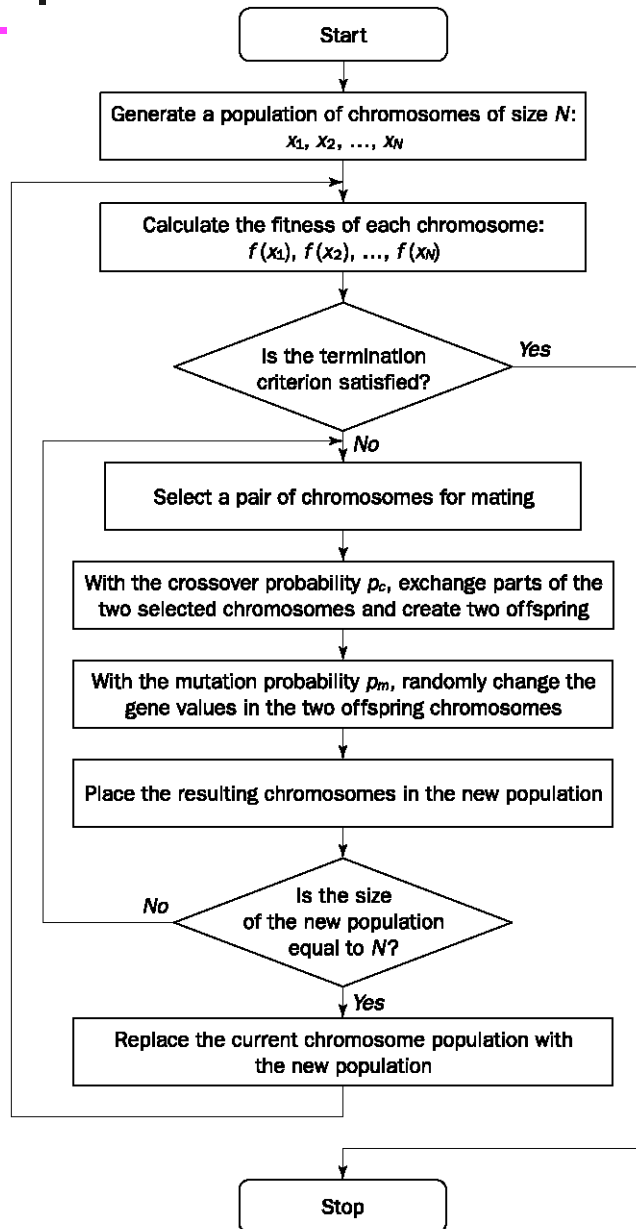
### 3. Genetic Algorithms (GAs)



# 3. Genetic Algorithms (GAs)



# 3. Genetic Algorithms (GAs)



### 3. Genetic Algorithms (GAs)

---

**Step 1:** Represent the problem variable domain as a chromosome of a **fixed** length, choose the size of a chromosome population  $N$ , the **crossover** probability  $p_c$ , and the **mutation** probability  $p_m$ .

**Step 2:** Define a **fitness function** to measure the performance (i.e., **fitness**) of an individual chromosome in the problem domain. The fitness function establishes the basis for selecting chromosomes that will be mated during reproduction.

### 3. Genetic Algorithms (GAs)

---

**Step 3:** Randomly generate an **initial population** of chromosomes of size  $N$ :

$$x_1, x_2, \dots, x_N$$

**Step 4:** Calculate the fitness of each individual chromosome:

$$f(x_1), f(x_2), \dots, f(x_N)$$

### 3. Genetic Algorithms (GAs)

---

**Step 5: Select** a pair of chromosomes for mating from the current population. Parent chromosomes are selected with a probability related to their fitness. Highly fit chromosomes have a higher probability of being selected for mating than less fit chromosomes.

**Step 6:** Create a pair of offspring chromosomes by applying the genetic operators - **crossover** and **mutation**.



### 3. Genetic Algorithms (GAs)

---

**Step 7:** Place the created offspring chromosomes in the **new population**.

**Step 8:** Repeat Step 5 until the size of the new chromosome population becomes equal to the size of the initial population  $N$ .

**Step 9:** Replace the initial (parent) chromosome population with the new (offspring) population.

**Step 10:** Go to Step 4, and repeat the process until the **termination criterion** is satisfied.

### 3. Genetic Algorithms (GAs)

---

- A GA represents an iterative process.
- Each iteration is called a **generation**.
- A typical number of generations for a simple GA can range from 50 to over 500.
- The entire set of generations is called a **run**.
- At the end of a run, we expect to find one or more highly fit chromosomes.

### 3. Genetic Algorithms (GAs)

---

- Because GAs use a stochastic search method, the fitness of a population may remain stable for a number of generations before a superior chromosome appears.
- A common practice is to terminate a GA after a specified number of generations and then examine the best chromosomes in the population.
- If no satisfactory solution is found, the GA is restarted.

# GA Example 1

---

- A simple example will help us to understand how a GA works.
- Let us find the maximum value of the function  $(15x - x^2)$  where parameter  $x$  varies between 0 and 15.
- For simplicity, we may assume that  $x$  takes only integer values. Thus, chromosomes can be built with only four genes:

# GA Example 1

Integer	Binary code	Integer	Binary code
0	0 0 0 0	8	1 0 0 0
1	0 0 0 1	9	1 0 0 1
2	0 0 1 0	10	1 0 1 0
3	0 0 1 1	11	1 0 1 1
4	0 1 0 0	12	1 1 0 0
5	0 1 0 1	13	1 1 0 1
6	0 1 1 0	14	1 1 1 0
7	0 1 1 1	15	1 1 1 1

# GA Example 1

---

- Suppose that the size of the chromosome population  $N$  is 6, the crossover probability  $p_c$  equals 0.7, and the mutation probability  $p_m$  equals 0.001.
- The fitness function in our example is defined by

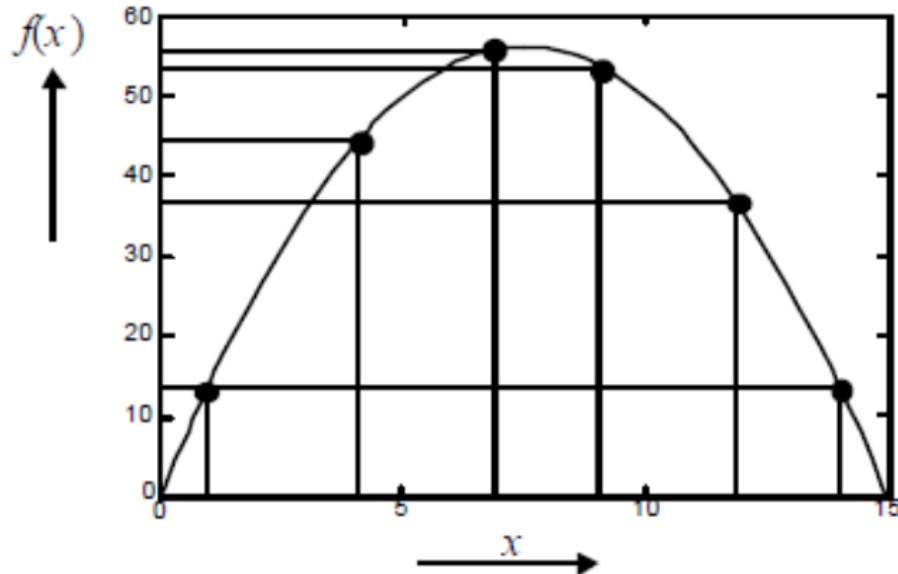
$$f(x) = 15x - x^2$$

- The GA creates an initial population of chromosomes by filling six 4-bit strings with randomly generated ones and zeros.

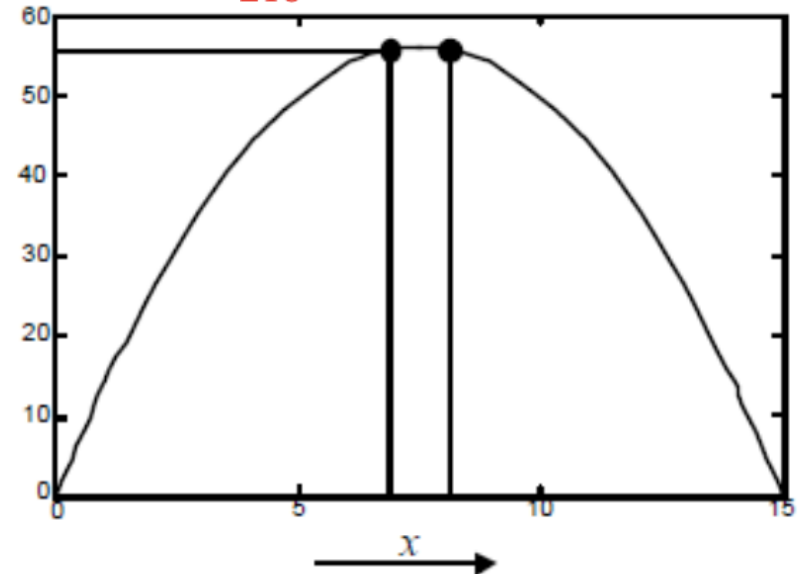
# GA Example 1

<i>Chromosome label</i>	<i>Chromosome string</i>	<i>Decoded integer</i>	<i>Chromosome fitness</i>	<i>Fitness ratio, %</i>
X1	1 1 0 0	12	36	16.5
X2	0 1 0 0	4	44	20.2
X3	0 0 0 1	1	14	6.4
X4	1 1 1 0	14	14	6.4
X5	0 1 1 1	7	56	25.7
X6	1 0 0 1	9	54	24.8

218



(a) Chromosome initial locations.



(b) Chromosome final locations.

# GA Example 1

---

- The **average fitness** of the initial population is 36 (i.e.,  $(36 + 44 + 14 + 14 + 56 + 54)/6 = 218/6 = 36.33 \approx 36$ ).
- To improve average fitness, initial population is modified by using genetic operators: **selection**, **crossover**, and **mutation**.
- In natural selection, only fittest species can survive, breed, and thereby pass their genes on to next generation.
- GAs use a similar approach, but unlike nature, the size of chromosome population remains unchanged from one generation to the next.



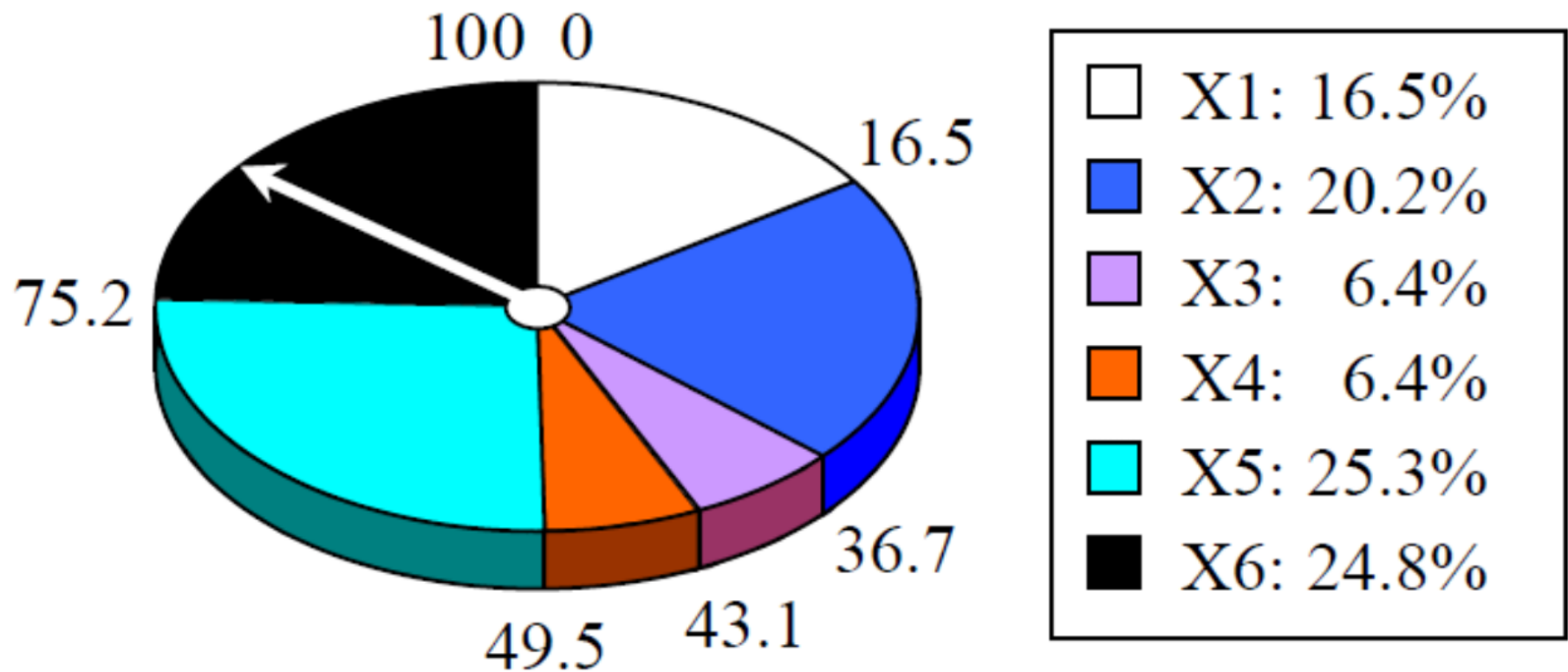
# GA Example 1

---

- The last column in previous Table shows the ratio of the individual chromosome's fitness to the population's total fitness (e.g.,  $36 \times 100 / 218 = 16.5$ ).
- This ratio determines the chromosome's chance of being selected for mating.
- The chromosomes X5 (25.7) and X6 (24.8) stand a fair chance, while the chromosomes X3 (6.4) and X4 (6.4) have a very low probability of being selected.
- The chromosome's **average fitness** improves from one generation to the next.

# Roulette Wheel Selection

- The most commonly used chromosome selection techniques is the **roulette wheel selection**.



# Crossover Operator

---

- In our example, we have an initial population of six chromosomes (i.e.,  $N = 6$ ).
- Thus, to establish the same population in the next generation, the roulette wheel would be spun six times.
- Once a pair of parent chromosomes is selected, the **crossover** operator is applied.

# Crossover Operator

---

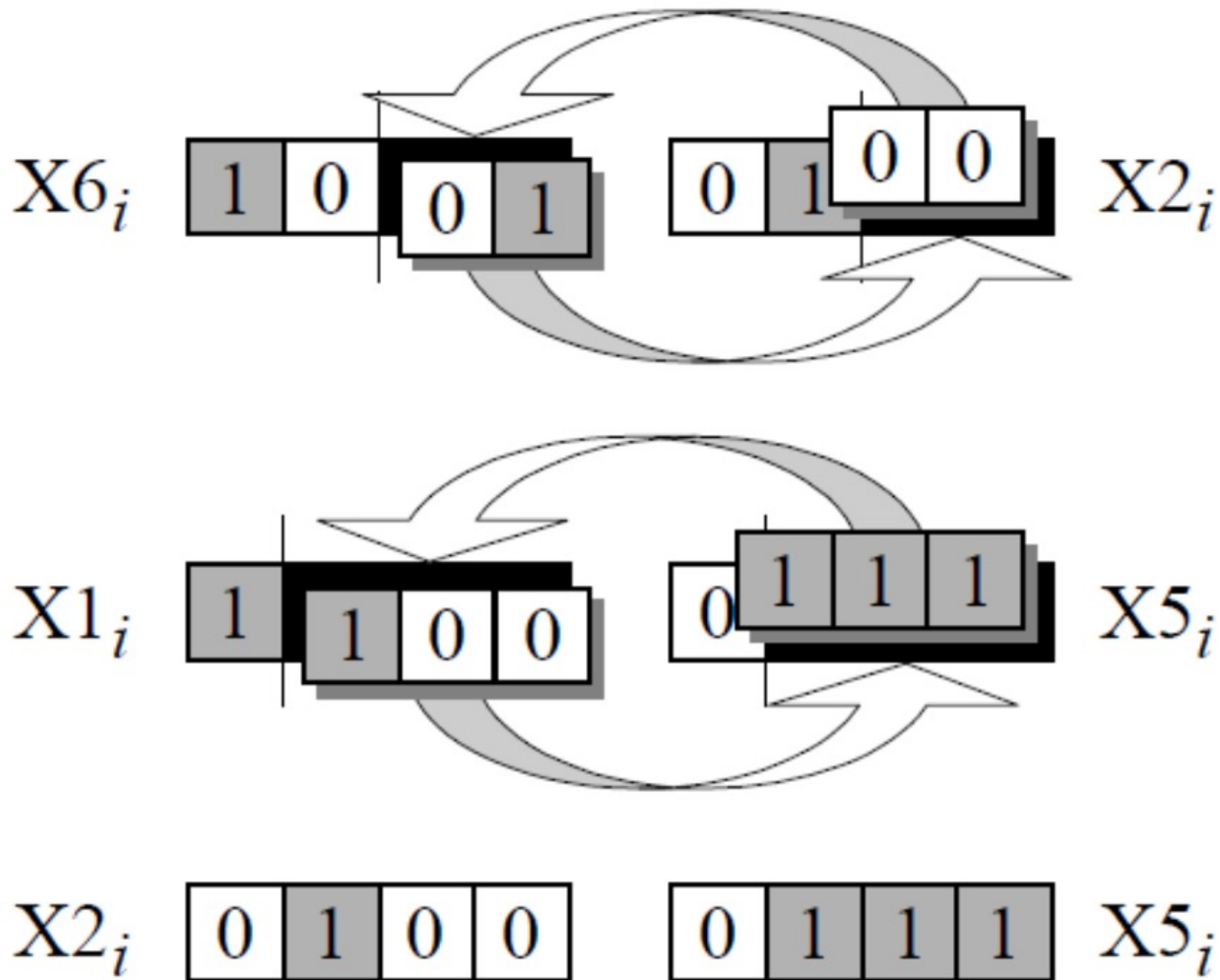
- Crossover operator randomly chooses a crossover point where two parent chromosomes ‘break’, and then exchanges chromosome parts after that point.
- After crossover, two new offspring are created.
- If a pair of chromosomes does not cross over, then chromosome **cloning** takes place, and the offspring are created as exact copies of each parent.

# Crossover Operator

---

- Crossover probability  $p_c$  falling within interval  $[0.7, 0.9]$  generally produces good results.
- After selection and crossover, **average fitness** of chromosome population is normally improved.

# Crossover Operator



# Mutation Operator

---

- **Mutation** represents a change in the gene. It may lead to a significant improvement in fitness, but more often has rather harmful results.
- Mutation is a background operator. Its role is to provide a guarantee that search algorithm is not trapped on a local optimum.
- Mutation operator flips a randomly selected gene in a chromosome.

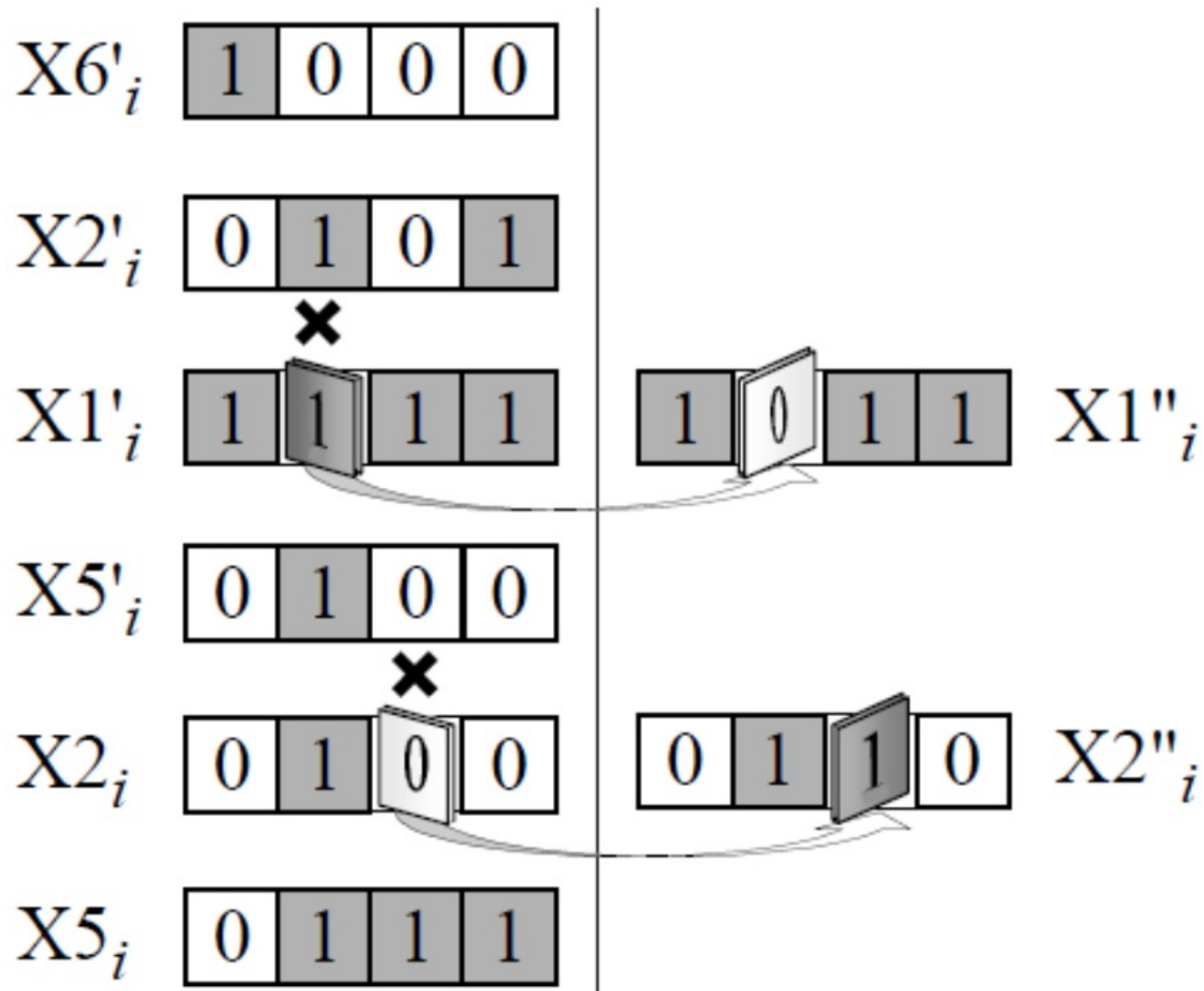
# Mutation Operator

---

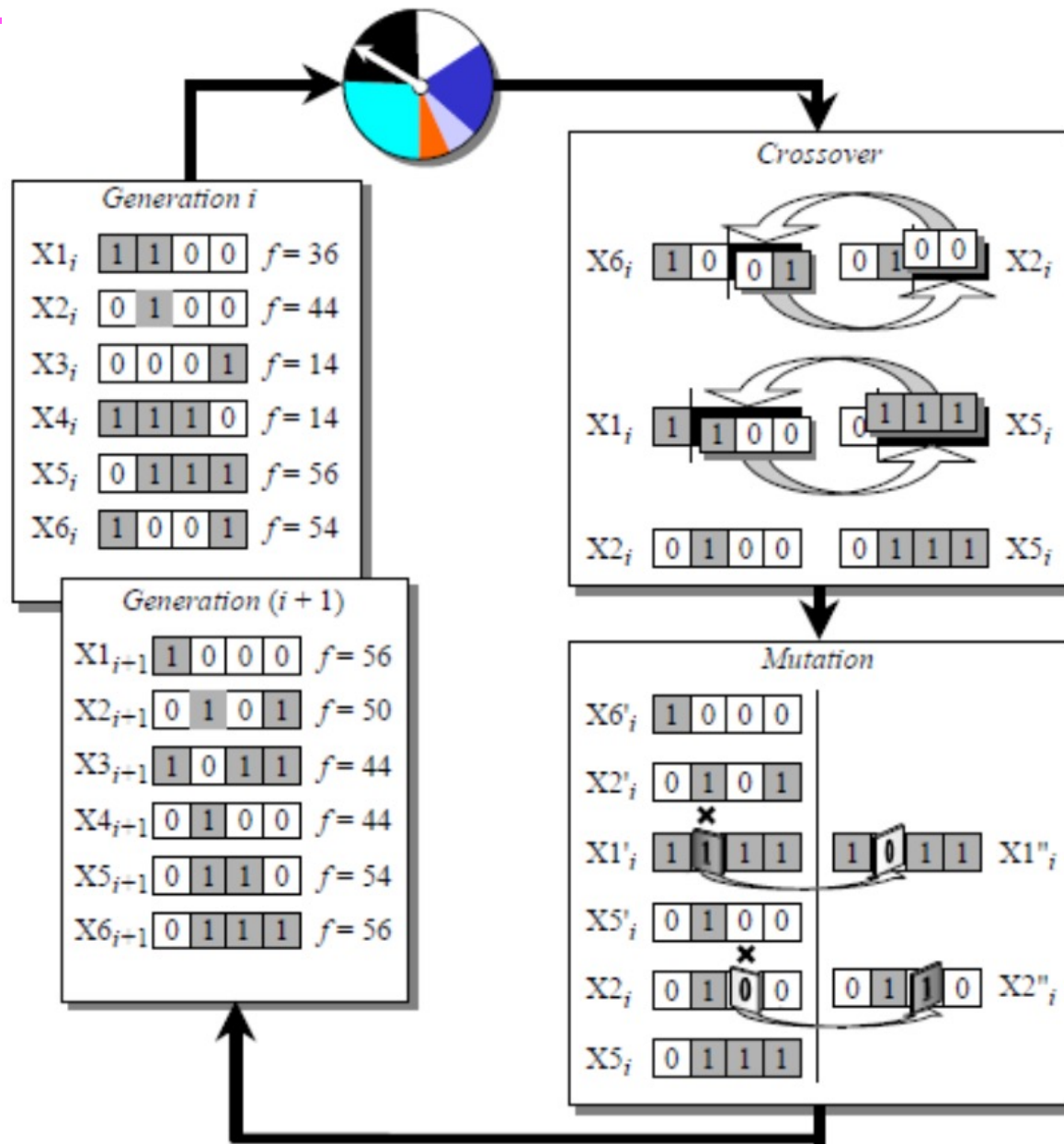
- Mutation probability  $p_m$  is quite small in nature, and is kept quite low for GAs, typically in the range between 0.001 and 0.01.
- GAs assure the continuous improvement of **average fitness** of the population, and after a number of generations (typically several hundred), population evolves to a **near-optimal** solution.



# Mutation Operator



# Genetic Algorithms Cycle



## GA Example 2

- Suppose that we want to find the maximum of the ‘peak’ function of two variables  $x$  and  $y$ :

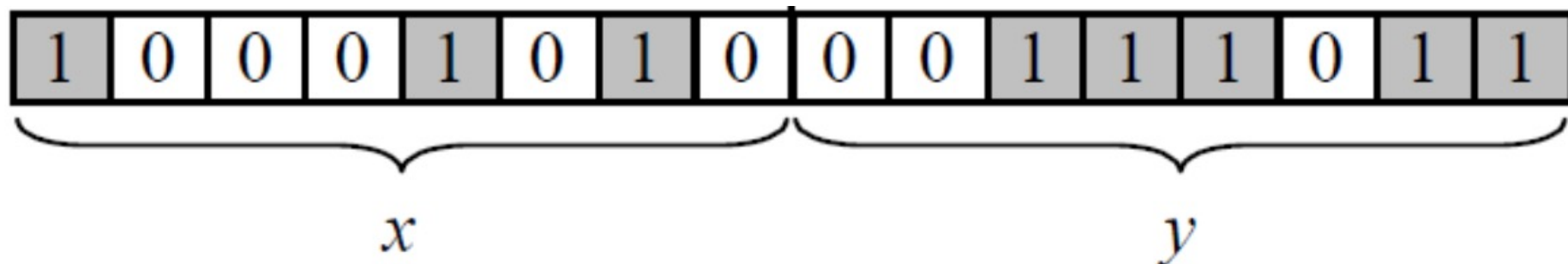
$$f(x, y) = (1 - x)^2 e^{-x^2 - (y+1)^2} - (x - x^3 - y^3) e^{-x^2 - y^2},$$

where parameters  $x$  and  $y$  are real numbers in  $[-3, 3]$ .

- The first step is to represent the problem variables  $x$  and  $y$  as a chromosome.

## GA Example 2

- We represent parameters  $x$  and  $y$  as a concatenated binary string:



in which each parameter is represented by an 8-bit string.

- We choose the chromosome population size  $N = 6$  and randomly generate an initial population.

## GA Example 2

- The next step is to calculate the **fitness** of each chromosome. This is done in two stages.
- First stage: a chromosome (i.e., a 16-bit string) is partitioned into two 8-bitstrings.

1	0	0	0	1	0	1	0
---	---	---	---	---	---	---	---

 and 

0	0	1	1	1	0	1	1
---	---	---	---	---	---	---	---

- Second stage: the two strings are converted from binary (base 2) to decimal (base 10):

$$(10001010)_2 = (138)_{10}, \text{ and } (00111011)_2 = (59)_{10}$$

## GA Example 2

- Now the range of integers  $[0, 255]$  (i.e.,  $[0, 2^8 - 1]$ ) is mapped to the actual range of parameters  $x$  and  $y$  (i.e., real numbers in  $[-3, 3]$ ):

$$6 / 255 = 0.0235294$$

- To obtain the actual values of  $x$  and  $y$  (i.e., decoded values), we multiply their decimal values by 0.0235294 and subtract 3 from the results:

$$x = (138)_{10} \times 0.0235294 - 3 = 0.2470588$$

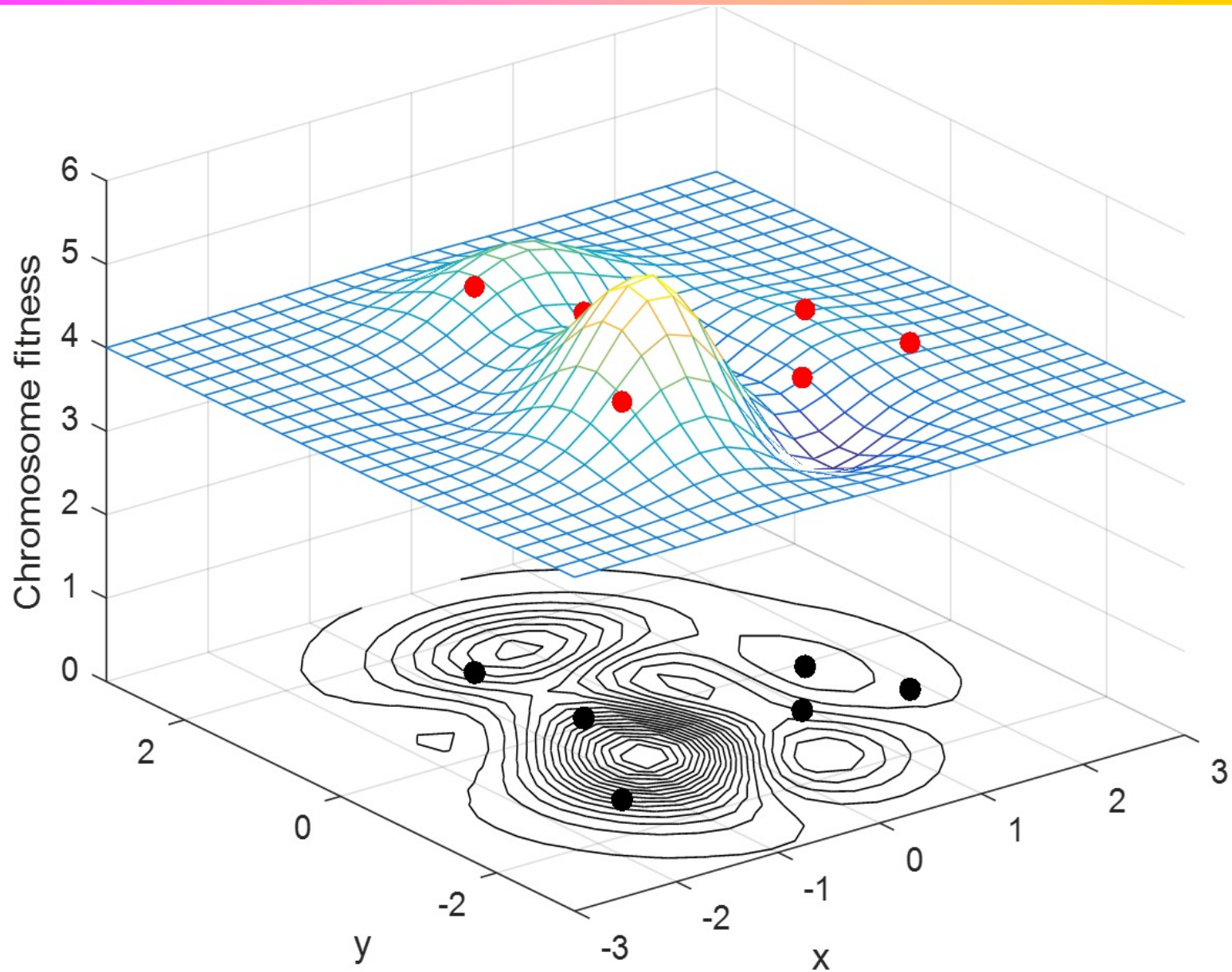
$$\text{and } y = (59)_{10} \times 0.0235294 - 3 = -1.6117647$$

## GA Example 2

---

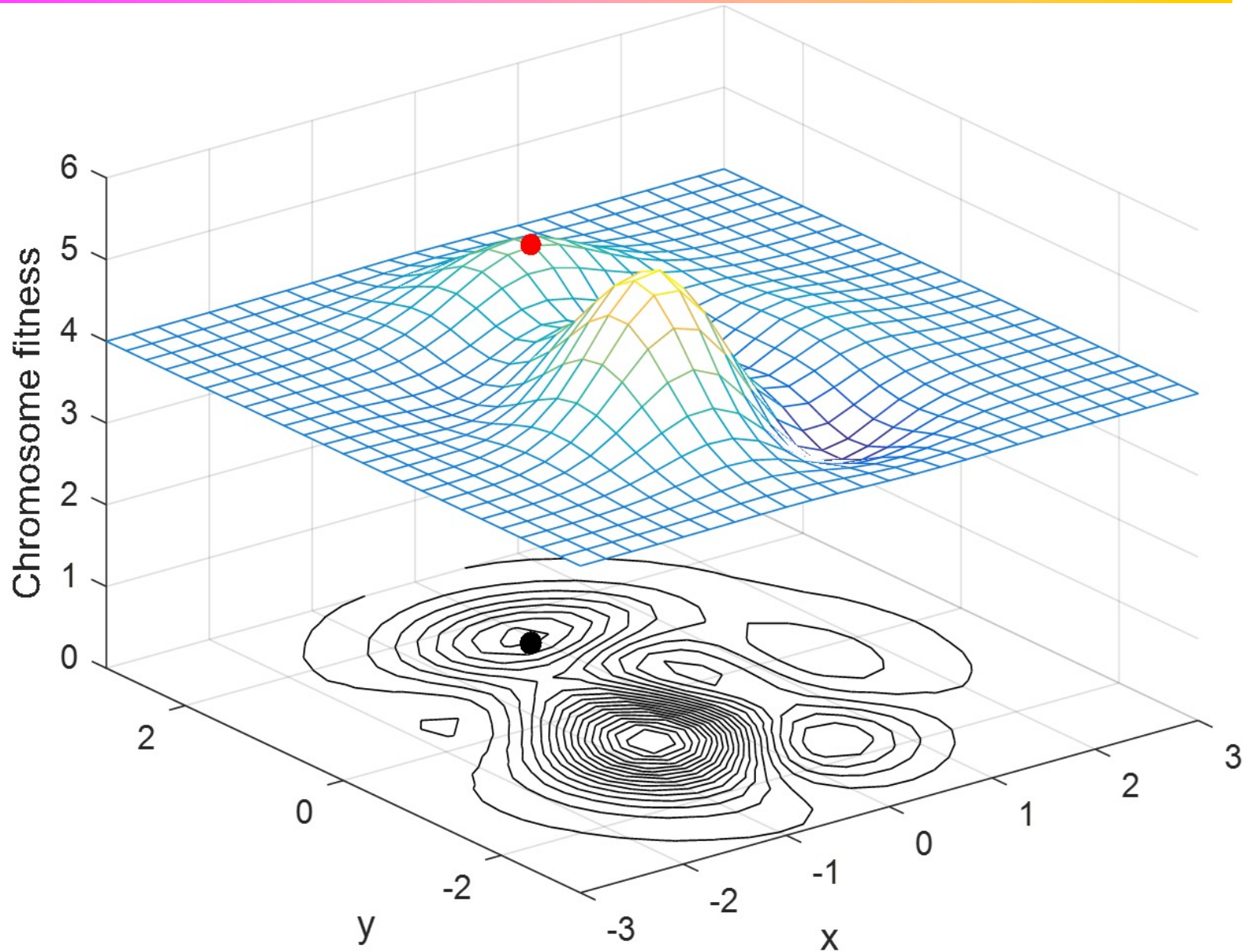
- Using decoded values of  $x$  and  $y$  as inputs in the mathematical function, the GA calculates the fitness of each chromosome.
- To find the maximum of the ‘peak’ function, we will use  $p_c = 0.7$  and  $p_m = 0.001$ .
- A common practice in GAs is to specify the number of generations.
- Suppose the desired number of generations is 100. That is, the GA will create 100 generations of 6 chromosomes before stopping.

# Initial Chromosome Population

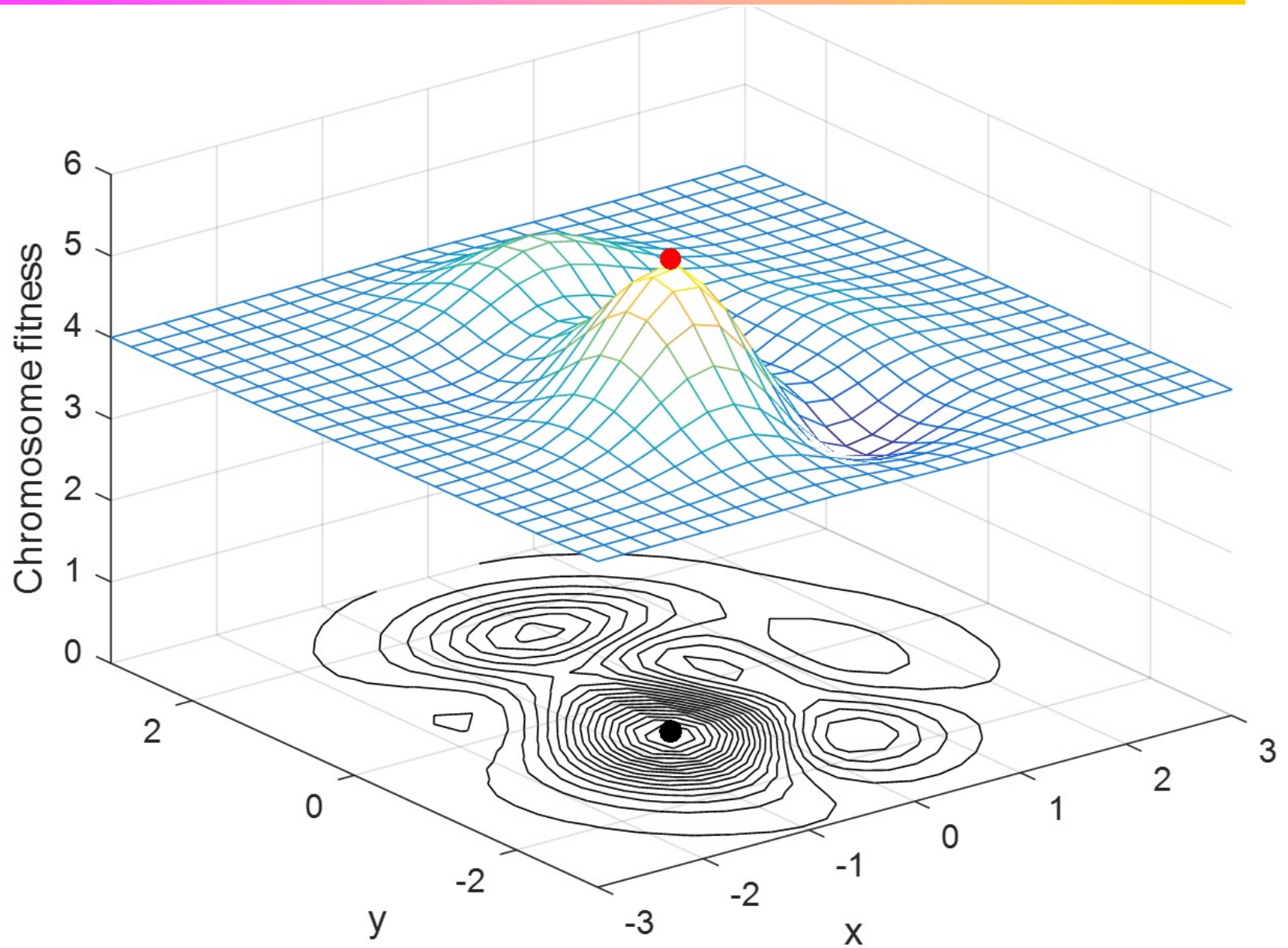




# Local Maximum (after 100 generations)



# Global Maximum (after 100 generations)



# Serious Problem with GAs

- The most serious problem in the use of GAs is the quality of a solution (i.e., whether or not an **optimal solution** is obtained.
- The quality of a solution can be improved by
  - increase  $p_m$  and rerun the GA. The results are less **steady**.
  - increase the population size. The results are more **steady**.

# Performance Graph

---

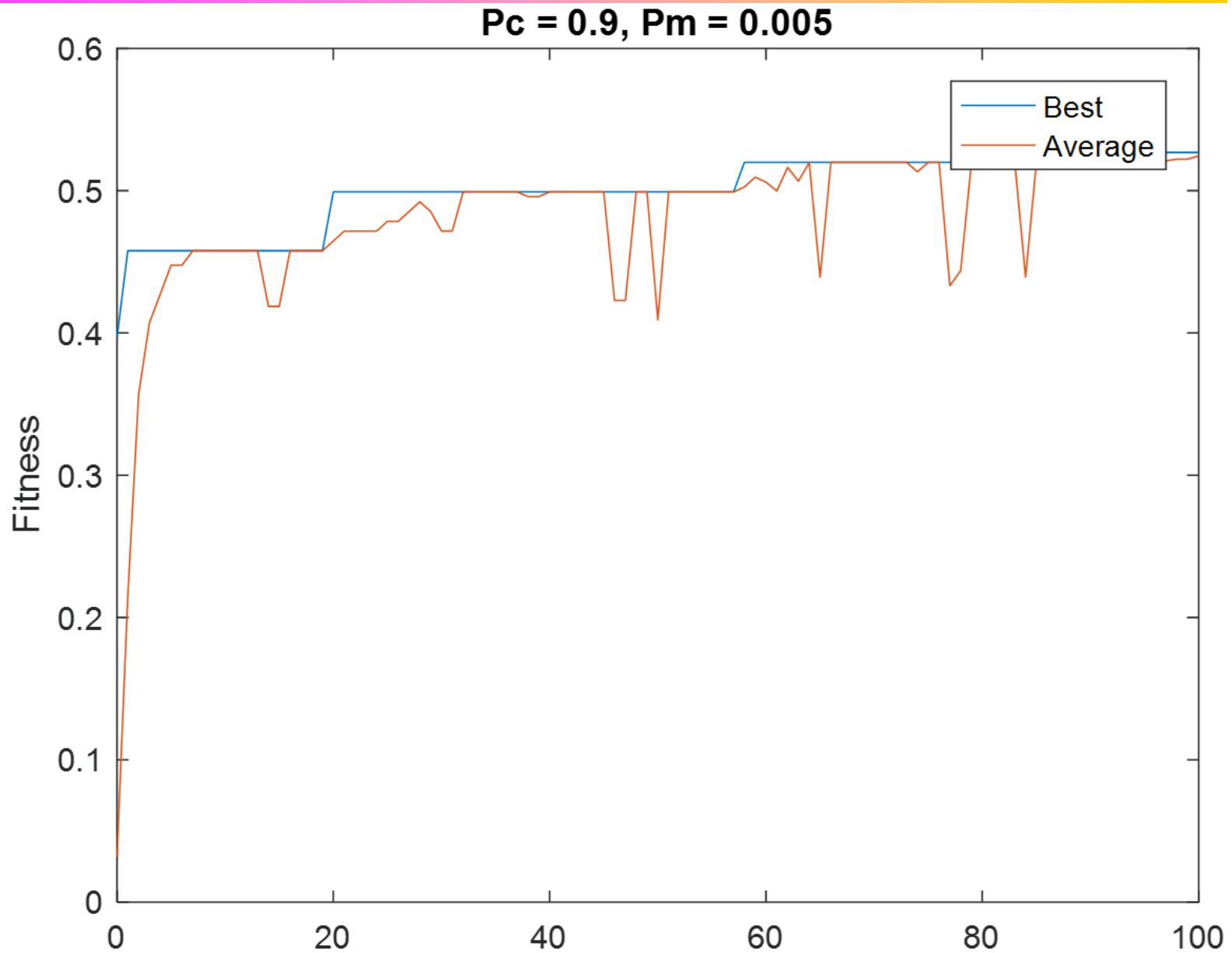
- A surface of a mathematical function can be displayed to show the GA's performance.
- Fitness functions for real world problems cannot be easily represented graphically. Instead, we can use **performance graphs**.

# Performance Graph

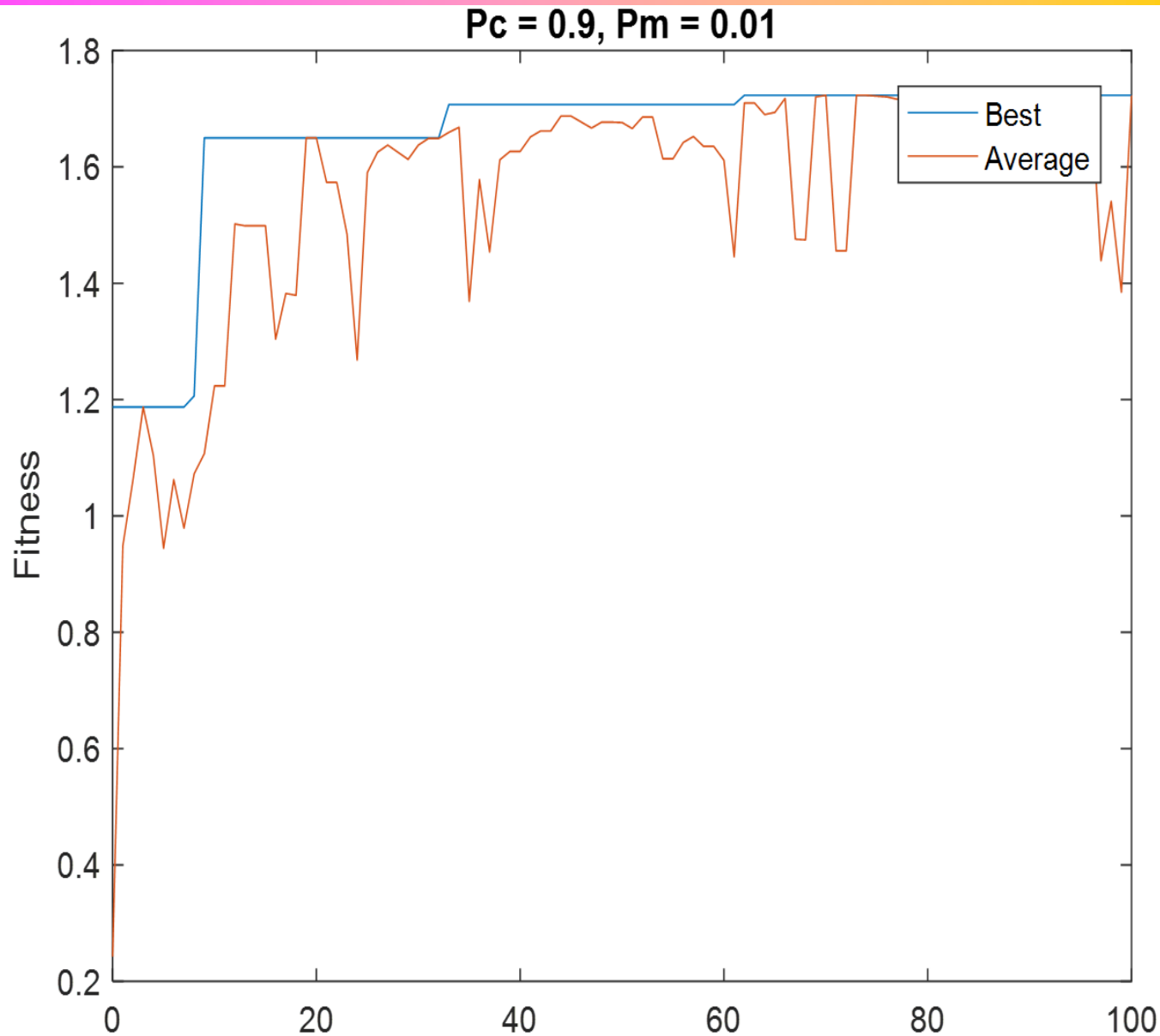
---

- Performance graph is a useful way of examining the behavior of a GA over the chosen number of generations.
- Performance graph includes
  - a curve shows the **average performance** of the entire population of chromosomes (i.e., the best values of the fitness function across generations)
  - a curve shows the performance of the **best** individual in the population (i.e., the average values of the fitness function across generations).

# Performance Graph (Local Maximum)



# Performance Graph (Global Maximum)



### 3. Genetic Algorithms (GAs)

---

- **Mutation** operator allows a GA to explore the landscape in a random manner.
- Mutation may lead to significant improvement in the population fitness, but more often decreases it.



# Binary Encoding vs. Gray Encoding

- Instead of using binary coding, we can also use **Gray coding**.  $x_{gi} = x_{bi}$  if  $i = 1$ ; otherwise,  $x_{gi} = x_{bi-1} \oplus x_{bi}$ . where  $\oplus$  denotes addition modulo 2.

$N$	Binary encoding	Gray encoding
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

# Contents

---

1. Introduction
2. Simulation of Natural Evolution
3. Genetic Algorithms (GAs)
- 4. Why Genetic Algorithms Work**
5. Case Study
6. Evolutionary Strategies
7. Genetic Programming (GP)

## 4. Why Genetic Algorithms Work

---

- GA techniques have a solid theoretical foundation that is based on the **Schema Theorem**.
- John Holland introduced the notation of **schema**, which came from the Greek word meaning ‘form’.
- A **schema** is a set of bit strings of 1’s, 0’s and \* (asterisks), where each \* can be 1 or 0.
- 1’s and 0’s represent the **fixed positions of a schema**, while \* represent a wild cards.

# Schema

- The schema  $H$  1 \* \* 0 stands for a set of 4-bit strings

1 1 1 0, 1 1 0 0, 1 0 1 0, and 1 0 0 0

- Each string in this set begins with 1 and ends with 0. These strings (chromosomes) are called **instances** of the schema  $H$ .
- A chromosome matches a schema when the **fixed positions** in the schema match the corresponding positions in the chromosome.

# Order of a Schema

- The number of **defined bits** (i.e., non-asterisks) in a schema is called the **order** of the schema.
- The schema  $H 1 * * 0$  has two defined bits so its order is 2.
- GAs manipulate **schemata** (schemata is the plural of the word schema) when they run.

# Schema Theorem

---

- According to the **Schema Theorem**, if GAs use a technique that makes the probability of reproduction proportional to chromosome fitness, then we can predict the presence of a given schema in the next chromosome generation.
- According to the **Schema Theorem**, we can describe the GA's behavior in terms of the **increase** or **decrease** in the number of instances of a given schema.

## Compute $m_H(i + 1)$

- Let us assume that at least one instance of the schema  $H$  is present in the chromosome initial generation  $i$ .
- Let  $m_H(i)$  be the number of instances of the schema  $H$  in the generation  $i$ , and  $\hat{f}_H(i)$  be the **average fitness** of these instances.
- We want to calculate **the number of instances** of the schema  $H$  in the next generation  $m_H(i + 1)$ .

## Compute $m_H(i + 1)$

- As the probability of reproduction is proportional to chromosome fitness, we can easily calculate the **expected number of offspring** of a chromosome  $x$  in the next generation by

$$m_x(i + 1) = f_x(i) / \hat{f}(i), (7.1)$$

where  $f_x(i)$  is the **fitness** of the chromosome  $x$ , and  $\hat{f}(i)$  is the **average fitness** of the chromosome initial generation  $i$ .



## Compute $m_H(i + 1)$

- Assume that the chromosome  $x$  is an instance of the schema  $H$ , we obtain

$$m_H(i + 1) = \sum_{x=1}^{m_H(i)} f_x(i) / \hat{f}(i), x \in H \quad (7.2)$$


- By definition of  $\hat{f}_H(i)$ , we have

$$\hat{f}_H(i) = \sum_{x=1}^{m_H(i)} f_x(i) / m_H(i),$$

## Compute $m_H(i + 1)$

- Finally, we obtain

$$m_H(i + 1) = (\hat{f}_H(i) / \hat{f}(i)) \times m_H(i) \quad (7.3)$$

- A schema  $H$  with **above-average fitness** (i.e., average fitness of its instances greater than average fitness of chromosome population at generation  $i$ ) tends to occur more frequently in the next generation  $(i + 1)$  of chromosomes. 
- A schema  $H$  with **below-average fitness** tends to occur less frequently in the next generation  $(i + 1)$  of chromosomes.

# Effects of Crossover and Mutation on Schema

---

- **Crossover** and **mutation** can both create and destroy instances of a schema.
- We consider only destructive effects of crossover and mutation. That is effects that **decrease** the number of instances of the schema  $H$ .
- The schema  $H$  will survive after **crossover** if at least one of its offspring is also its instance. This is the case when **crossover** does not occur within the **defining length of the schema**.

# Defining Length of a Schema

- The distance between the outermost defined bits of a schema is called **defining length**.
- The defining length of  $* * * * 1 0 1 1$  is 3, of  $* 0 * 1 * 1 0 *$  is 5, and of  $1 * * * * * 0$  is 7.
- If **crossover** takes place within the defining length, the schema  $H$  can be destroyed and offspring that are not instances of  $H$  can be created.

# Effects of Crossover on Schema

- The probability that the schema  $H$  will survive after **crossover** is defined as

$$P_H^{(c)} = 1 - p_c [\ell_d / (\ell - 1)] \quad (7.4)$$

where  $p_c$  is the crossover probability, and  $\ell$  and  $\ell_d$  are, respectively, the length and the defining length of the schema  $H$ .

- It is clear that the probability of survival under crossover is higher for short schemata rather than for long ones.

# Effects of Mutation on Schema

- Let  $p_m$  be mutation probability for any bit of schema  $H$ , and  $n$  be **order of the schema**  $H$ .
- Probability that a bit will not be mutated is  $(1 - p_m)$ .
- Probability that schema  $H$  will survive after **mutation** is determined as

$$P_H^{(m)} = (1 - p_m)^n \quad (7.5)$$

- It is clear that probability of survival under **mutation** is higher for low-order schemata than for high-order ones.

# Effects of Crossover and Mutation on Schema

- Equation (7.3) is modified to take into account the destructive effects of crossover and mutation as follows.

$$m_H(i + 1) = \frac{\hat{f}_H(i)}{\hat{f}(i)} m_H(i) \left[ 1 - p_c \left( \frac{\ell_d}{\ell - 1} \right) \right] (1 - p_m)^n \quad (7.6)$$

- Equation (7.6) describes the growth of a schema from one generation to the next.

# Effects of Crossover and Mutation on Schema

$$m_H(i + 1) = \frac{\hat{f}_H(i)}{\hat{f}(i)} m_H(i) \left[ 1 - p_c \left( \frac{\ell_d}{\ell - 1} \right) \right] (1 - p_m)^n \quad (7.6)$$

- Equation (7.6) is known as the **Schema Theorem**.
- It considers only the destructive effects of crossover and mutation.
- The above equation gives us a lower bound on the number of instances of the schema  $H$  in the next generation.



## 4. Why Genetic Algorithms Work

---

- Despite **crossover** arguably representing a major advantage of GAs, there is no theoretical basis to support the view that a GA will outperform other search and optimization techniques just because **crossover** allows the combination of partial solutions.
- GAs are a very powerful tool, but need to be applied intelligently.

# Contents

---

1. Introduction
2. Simulation of Natural Evolution
3. Genetic Algorithms (GAs)
4. Why Genetic Algorithms Work
- 5. Case Study**
6. Evolutionary Strategies
7. Genetic Programming (GP)

## 5. Case Study

---

- Self-study

# Contents

---

1. Introduction
2. Simulation of Natural Evolution
3. Genetic Algorithms (GAs)
4. Why Genetic Algorithms Work
5. Case Study
- 6. Evolutionary Strategies**
7. Genetic Programming (GP)

## 6. Evolutionary Strategies

---

- Another approach to simulating natural evolution was proposed in Germany in the early 1960s.
- Unlike GAs, this approach - called an **evolution strategy** (ES) - was designed to solve technical optimization problems.
- In 1963 two students of the Technical University of Berlin, Ingo Rechenberg and Hans-Paul Schwefel, were working on the search for the optimal shapes of bodies in a flow.

## 6. Evolutionary Strategies

---

- They decided to try random changes in the parameters defining the shape following the example of natural mutation. As a result, the evolution strategy (ES) was born.
- Evolution strategies (ESs) were developed as an alternative to the engineer's intuition.
- Unlike GAs, ESs use only a mutation operator.

# Basic Evolutionary Strategies

---

- In its simplest form, termed as a  $(1+1)$ -evolution strategy, one parent generates one offspring per generation by applying **normally distributed** mutation (bell-shaped curve).
- The  $(1+1)$ -evolution strategy can be implemented as follows:

# Basic Evolutionary Strategies

**Step 1:** Choose the number of parameters  $N$  to represent the problem, and then determine a feasible range for each parameter:

$$\{x_{1min}, x_{1max}\}, \{x_{2min}, x_{2max}\}, \dots, \{x_{Nmin}, x_{Nmax}\}.$$

Define a standard deviation for each parameter and the function to be optimized.



# Basic Evolutionary Strategies

**Step 2:** Randomly select an initial value for each parameter from the respective feasible range. The set of these parameters will constitute the initial population of parent parameters:  $x_1, x_2, \dots, x_N$

**Step 3:** Calculate the solution associated with the parent parameters:  $X = f(x_1, x_2, \dots, x_N)$

# Basic Evolutionary Strategies

**Step 4:** Create a new (offspring) parameter by adding a normally distributed random variable  $a$  with mean zero and pre-selected standard deviation  $\delta$  to each parent parameter:

$$x'_i = x_i + a(0, \delta), \quad i = 1, 2, \dots, N \quad (7.7)$$

Normally distributed mutations with mean zero reflect the natural process of evolution where smaller changes occur more frequently than larger ones.

# Basic Evolutionary Strategies

**Step 5:** Calculate the solution associated with the offspring parameters:  $X' = f(x'_1, x'_2, \dots, x'_N)$

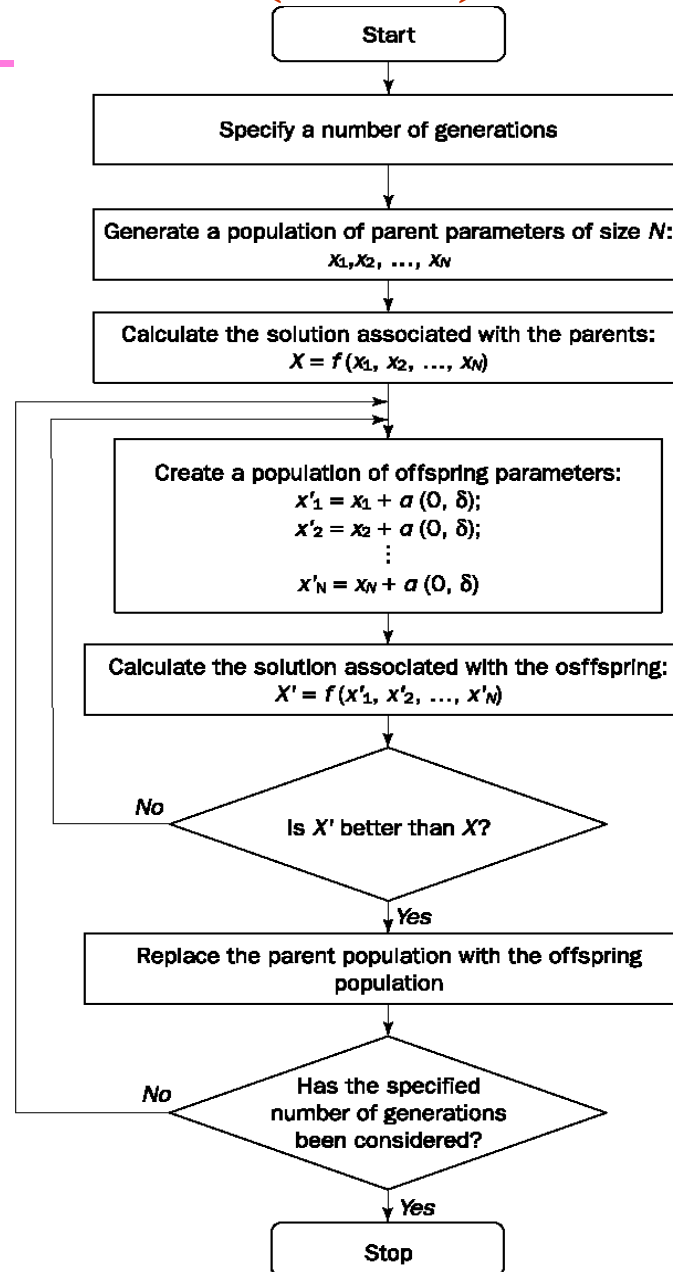
**Step 6:** Compare the solution associated with the offspring parameters with the one associated with the parent parameters. If the solution for the offspring is better than that for the parents, replace the parent population with the offspring population. Otherwise, keep the parent parameters.

# Basic Evolutionary Strategies

---

**Step 7:** Go to Step 4, and repeat the process until a satisfactory solution is reached, or a specified number of generations is considered.

# Block-Diagram of (1 + 1)-Evolution Strategy



## 6. Evolutionary Strategies (ESs)

---

- An evolution strategy reflects the nature of a chromosome.
- A single gene may simultaneously affect several characteristics of the living organism.
- On the other hand, a single characteristic of an individual may be determined by the simultaneous interactions of several genes.
- The natural selection acts on a collection of genes, not on a single gene in isolation.

## 6. Evolutionary Strategies

---

- ESs can solve a wide range of constrained and unconstrained non-linear optimization problems and produce better results than many conventional, highly complex, non-linear optimization techniques.
- Experiments also suggest that the simplest version of ESs that uses a single parent - single offspring search works best.

# Differences between GAs and ESs

---

- The principal difference between a GA and an ES is that the former uses both crossover and mutation whereas the latter uses only mutation.
- In addition, when we use an ES, we do not need to represent the problem in a coded form.



# Differences between GAs and ESs

---

- An ES uses a purely numerical optimization procedure.
- GAs are capable of more general applications, but the hardest part of applying a GA is coding the problem.

# Contents

---

1. Introduction
2. Simulation of Natural Evolution
3. Genetic Algorithms (GAs)
4. Why Genetic Algorithms Work
5. Case Study
6. Evolutionary Strategies
- 7. Genetic Programming (GP)**

## 7. Genetic Programming (GP)

---

- One of the central problems in computer science is how to make computers solve problems without being explicitly programmed to do so.
- GP offers a solution through the evolution of computer programs by methods of natural selection.
- In fact, GP is an extension of the conventional GA, but the goal of GP is not just to evolve a bit-string representation of some problem but the **computer code** that solves the problem.

## 7. Genetic Programming (GP)

---

- GP creates **computer programs** as the solution, while GAs create a string of binary numbers that represent the solution.
- GP is a recent development in the area of evolutionary computation (EC).
- GP was greatly stimulated in the 1990s by John Koza.
- According to Koza, GP searches the space of possible **computer programs** for a program that is highly fit for solving the problem at hand.

## 7. Genetic Programming (GP)

---

- Any computer program is a sequence of operations (functions) applied to values (arguments), but different programming languages may include different types of statements and operations, and have different syntactic restrictions.

## 7. Genetic Programming (GP)

---

- Since GP manipulates programs by applying genetic operators, a programming language should permit a computer program to be manipulated as data and the newly created data to be executed as a program. For these reasons, **LISP** was chosen as the main language for GP.

# LISP

---

- LISP (**List Processor**) has a highly symbol-oriented structure.
- LISP basic data structures are **atoms** and **lists**.
- An atom is the smallest indivisible element of the LISP syntax.
- The number *21*, the symbol *X* and the string '*This is a string*' are examples of LISP atoms.
- A list is an object composed of atoms and/or other lists.

# LISP

- LISP lists are written as an ordered collection of items inside a pair of parentheses.
- For example, the list `(-(* A B) C)` calls for the application of the subtraction function `(-)` to two arguments, namely the list `(*AB)` and the atom `C`.
- First, LISP applies the multiplication function `(*)` to the atoms `A` and `B`. Once the list `(*AB)` is evaluated, LISP applies the subtraction function `(-)` to the two arguments, and thus evaluates the entire list `(-(*AB) C)`.



# LISP

---

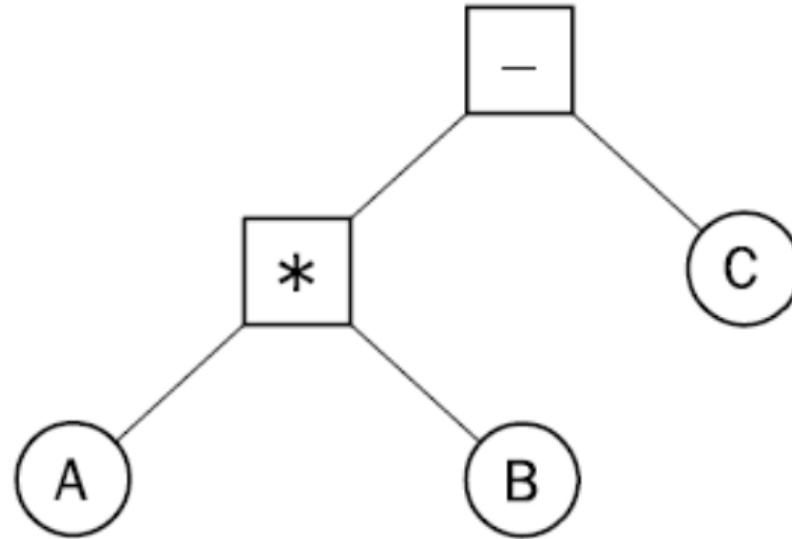
- Both atoms and lists are called symbolic expressions (**S-expressions**).
- In LISP, all data and all programs are **S-expressions**. This gives LISP the ability to operate on programs as if they were data.

# LISP

---

- LISP programs can modify themselves or even write other LISP programs. This remarkable property of LISP makes it very attractive for GP.
- Any LISP S-expression can be depicted as a rooted point-labelled tree with ordered branches.

# Graphical Representation of $-(AB) C$



- Terminals: A, B, C
- Functions:  $-$ ,  $*$

# How to Apply GP to a Problem

---

- Before applying GP to a problem, we must accomplish **five preparatory steps**:

1. Determine the set of terminals.
2. Select the set of primitive functions.
3. Define the fitness function.
4. Decide on the parameters for controlling the run.
5. Choose the method for designating a result of the run.

# Pythagorean Theorem Example

- The Pythagorean Theorem helps us to illustrate these preparatory steps and demonstrate the potential of GP. The theorem says that the hypotenuse  $c$  of a right triangle with short sides  $a$  and  $b$  is given by

$$c = \sqrt{a^2 + b^2}$$

- The aim of GP is to discover a program that matches this function.

# An Example

---

- To measure the performance of the as-yet-undiscovered computer program, we will use a number of different **fitness cases**.
- The fitness cases for the Pythagorean Theorem are represented by the samples of right triangles in Table. These fitness cases are chosen at random over a range of values of variables  $a$  and  $b$ .

# An Example

Side $a$	Side $b$	Hypotenuse $c$	Side $a$	Side $b$	Hypotenuse $c$
3	5	5.830952	12	10	15.620499
8	14	16.124515	21	6	21.840330
18	2	18.110770	7	4	8.062258
32	11	33.837849	16	24	28.844410
4	3	5.000000	2	9	9.219545

# An Example

---

## Step 1: *Determine the set of terminals*

The terminals correspond to the inputs of the computer program to be discovered. Our program takes two inputs,  $a$  and  $b$ .



# An Example

---

## Step 2: *Select the set of primitive functions*

The functions can be presented by standard arithmetic operations, standard programming operations, standard mathematical functions, logical functions or domain-specific functions. Our program will use four standard arithmetic operations  $+$ ,  $-$ ,  $*$  and  $/$ , and one mathematical function *sqrt*.

# An Example

---

## Step 3: *Define the fitness function*

A fitness function evaluates how well a particular computer program can solve the problem.

The choice of the fitness function depends on the problem, and may vary greatly from one problem to the next.

# An Example

---

## Step 3: *Define the fitness function*

For our problem, the fitness of the computer program can be measured by the error between the actual result produced by the program and the correct result given by the fitness case. Typically, the error is not measured over just one fitness case, but instead calculated as a sum of the absolute errors over a number of fitness cases. The closer this sum is to zero, the better the computer program.

# An Example

---

## Step 4: *Decide on the parameters for controlling the run*

For controlling a run, GP uses the same primary parameters as those used for GAs. They include the population size and the maximum number of generations to be run.

# An Example

---

**Step 5: *Choose the method for designating a result of the run***

It is common practice in GP to designate the best-so-far generated program as the result of a run.

## An Example

---

- Once these five steps are complete, a run can be made.
- The run of GP starts with a random generation of an initial population of computer programs.
- Each program is composed of functions  $+$ ,  $-$ ,  $*$ ,  $/$  and  $\sqrt{\phantom{x}}$ , and terminals  $a$  and  $b$ .

# An Example

---

- In the initial population, all computer programs usually have poor fitness, but some individuals are more fit than others. Just as a fitter chromosome is more likely to be selected for reproduction, so a fitter computer program is more likely to survive by copying itself into the next generation.

# An Example

---

- In GP, crossover operates on two computer programs which are selected on the basis of their fitness.
- These programs can have different sizes and shapes.
- The two offspring programs are composed by recombining randomly chosen parts of their parents.



## An Example of S-expressions

- For example, consider the following two LISP S-expressions:  $(/ (- (sqrt (+ (* a a) (-a b))) a) (* a b))$ , which is equivalent to

$$\frac{\sqrt{a^2 + (a - b)} - a}{ab},$$

- and  $(+ (- (sqrt (- (* b b) a)) b) (sqrt (/ a b)))$ , which is equivalent to

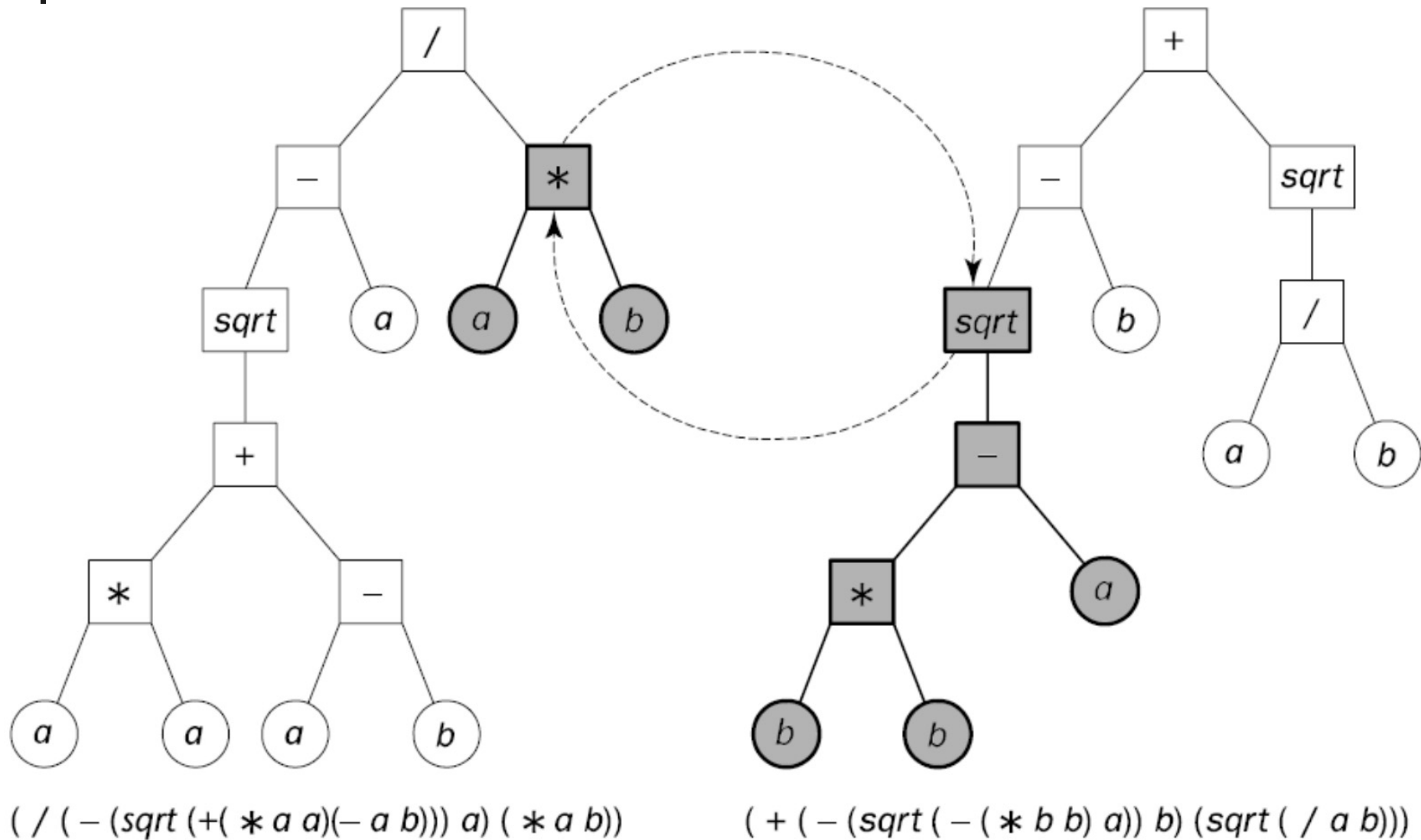
$$(\sqrt{b^2 - a} - b) + \sqrt{\frac{a}{b}}.$$

# An Example of S-expressions

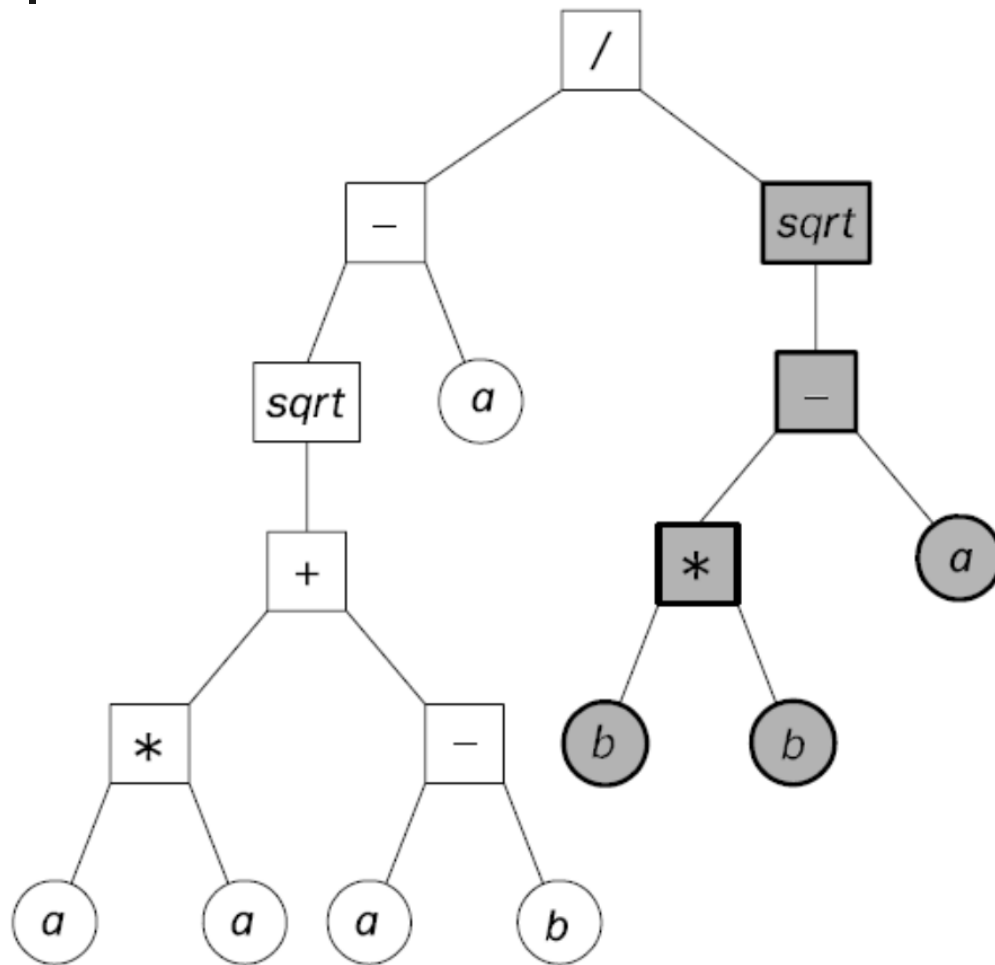
---

- These two S-expressions can be presented as rooted, point-labelled trees with ordered branches.

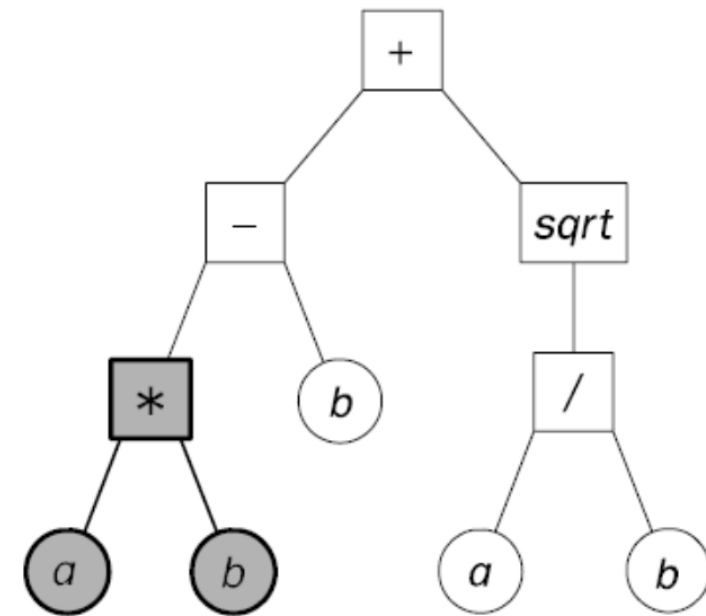
# Crossover in GP: Two Parental S-expressions



# Crossover in GP: Two Offspring S-expressions




$( / ( - ( \text{sqrt} ( + ( * a a ) ( - a b ) ) ) a ) ( \text{sqrt} ( - ( * b b ) a ) ) )$



$( + ( - ( * a b ) b ) ( \text{sqrt} ( / a b ) ) )$

## Two Offspring S-expressions

- These offspring are equivalent to

$$\frac{\sqrt{a^2 + (a - b)} - a}{ab} \text{ and } (\sqrt{b^2 - a} - b) + \sqrt{\frac{a}{b}}.$$
$$\frac{\sqrt{a^2 + (a - b)} - a}{\sqrt{b^2 - a}} \text{ and } (ab - b) + \sqrt{\frac{a}{b}}.$$


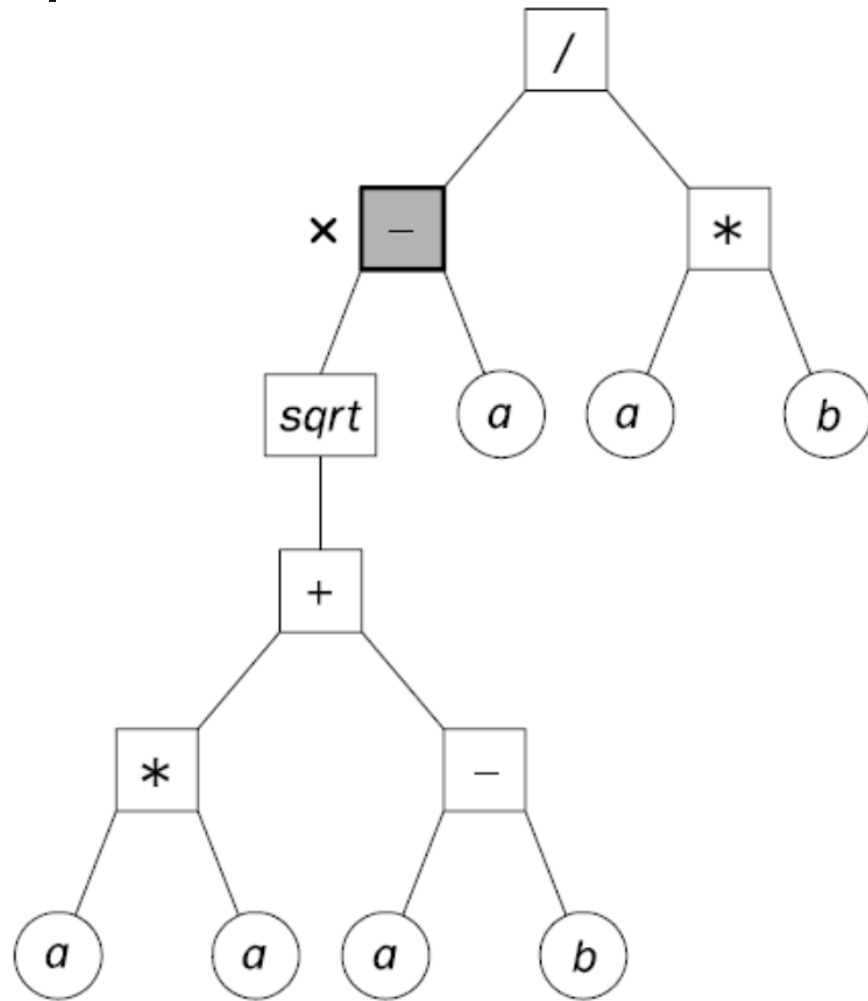
- The crossover operator produces valid offspring computer programs regardless of the choice of crossover points.

# Mutation in GP

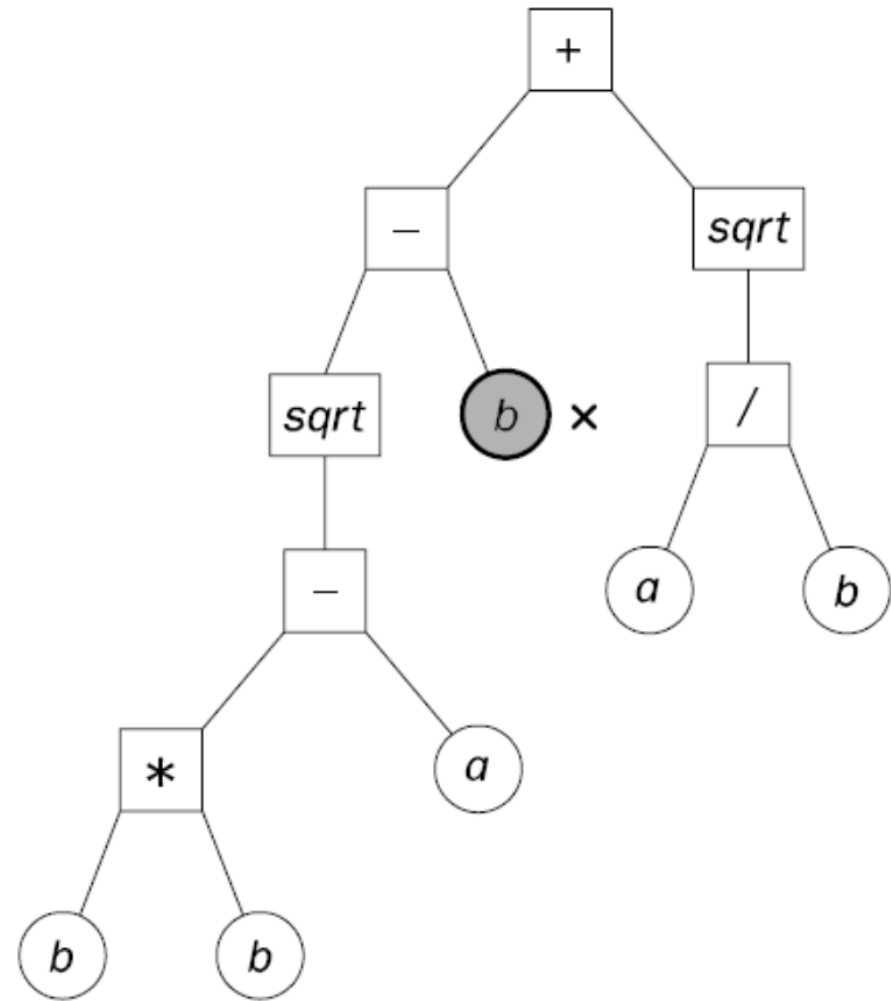
---

- A mutation operator can randomly change any function or any terminal in the LISP S-expression.
- Under mutation, a function can only be replaced by a function and a terminal can only be replaced by a terminal.

# Mutation in GP: Original S-expressions

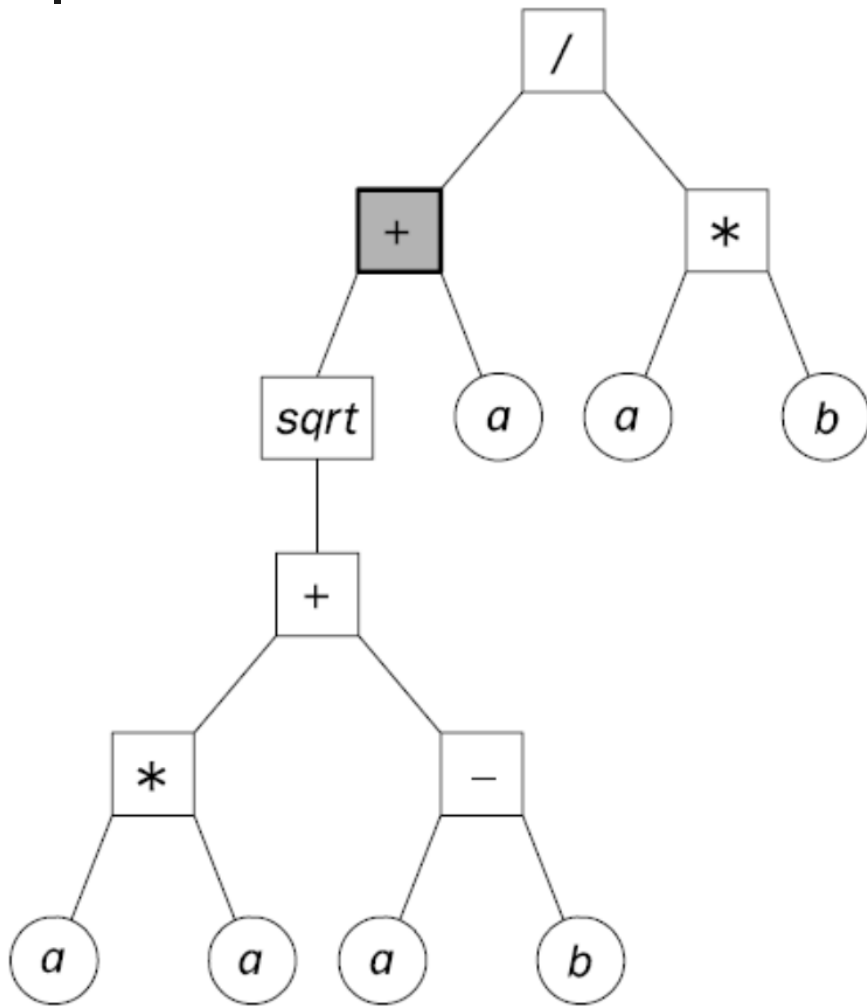


$( / ( - ( \text{sqrt} ( + ( * a a ) ( - a b ) ) ) a ) ( * a b ) )$

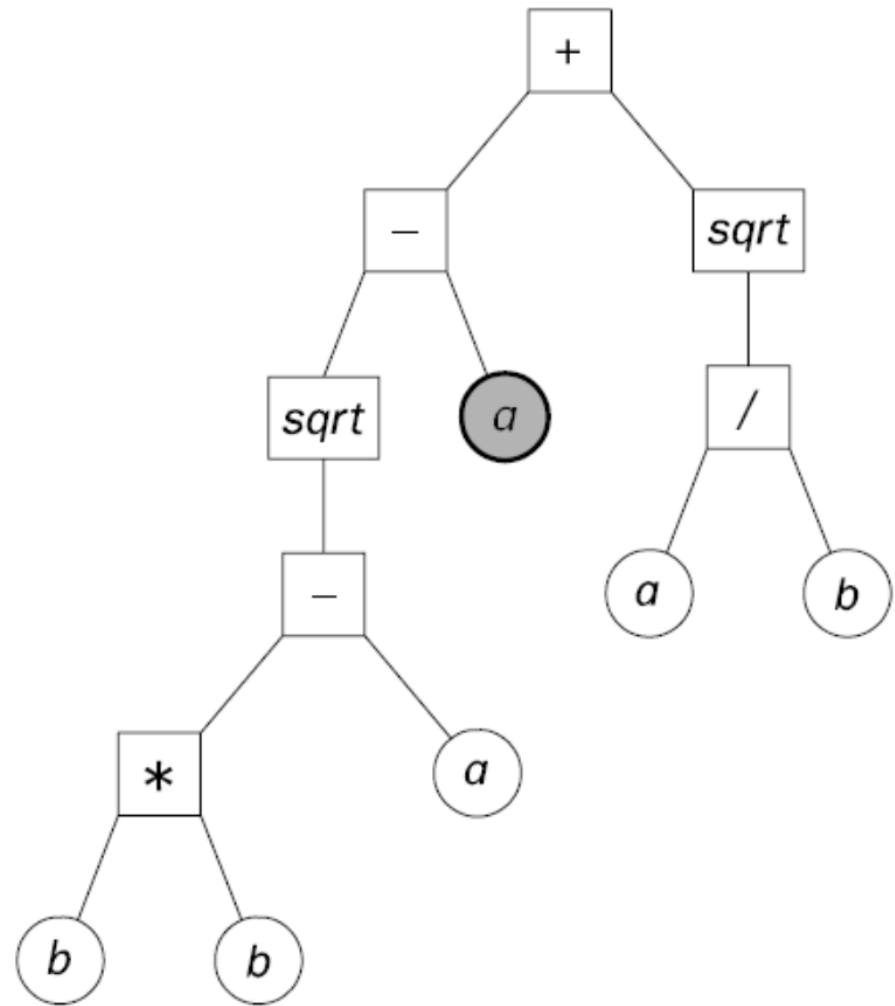


$( + ( - ( \text{sqrt} ( - ( * b b ) a ) ) b ) ( \text{sqrt} ( / a b ) ) )$

# Mutation in GP: Mutated S-expressions



$( / ( + ( \text{sqrt} ( + ( * a a ) ( - a b ) ) ) a ) ( * a b )$



$( + ( - ( \text{sqrt} ( - ( * b b ) a ) ) a ) ( \text{sqrt} ( / a b ) )$



# Steps of Genetic Programming

---

- In summary, GP creates computer programs by executing the following steps:

**Step 1:** Assign the maximum number of generations to be run and probabilities for cloning, crossover and mutation. Note that the sum of the probability of cloning, the probability of crossover and the probability of mutation must be equal to one.

# Steps of Genetic Programming

---

**Step 2:** Generate an initial population of computer programs of size  $N$  by combining randomly selected functions and terminals.

**Step 3:** Execute each computer program in the population and calculate its fitness with an **appropriate fitness function**. Designate the best-so-far individual as the result of the run.

**Step 4:** With the assigned probabilities, select a genetic operator to perform cloning, crossover or mutation.

# Steps of Genetic Programming

---

**Step 5:** If the cloning operator is chosen, select one computer program from the current population of programs and copy it into a new population.

- If crossover is chosen, select a pair of computer programs from the current population, create a pair of offspring programs and place them into the new population.

# Steps of Genetic Programming

---

- If mutation is chosen, select one computer program from the current population, perform mutation and place the mutant into the new population.
- All programs are selected with a probability based on their fitness (i.e., the higher the fitness, the more likely the program is to be selected).

# Steps of Genetic Programming

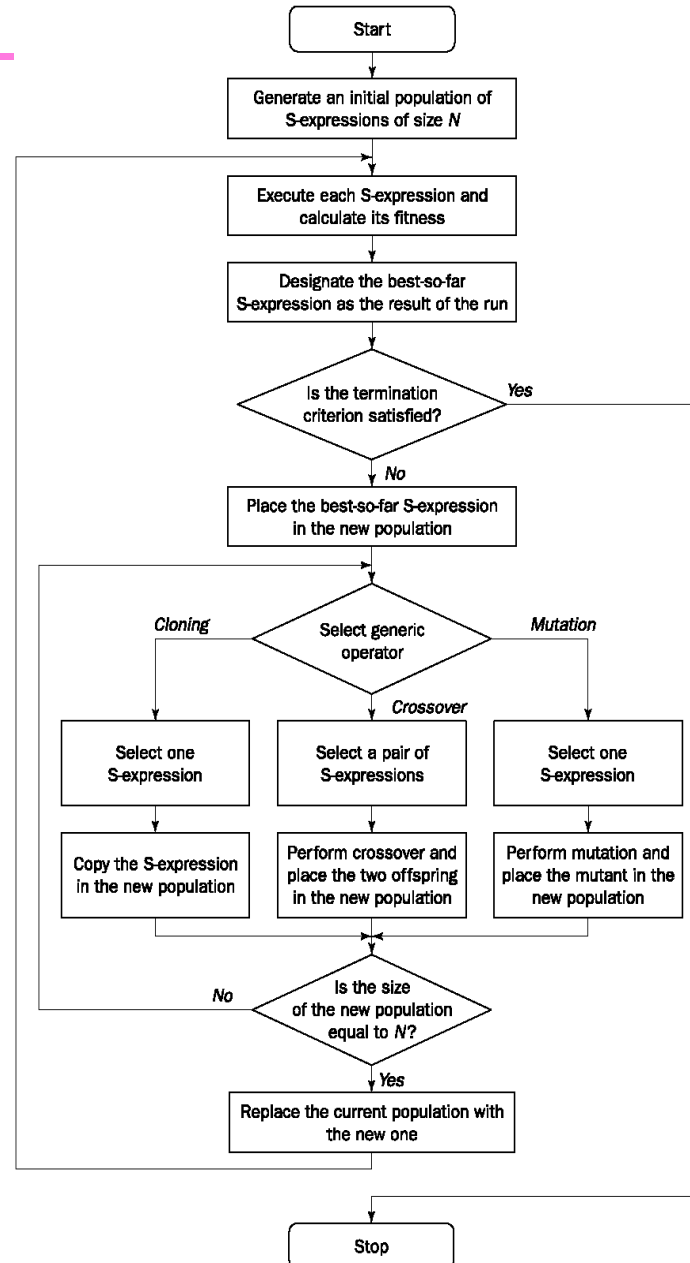
---

**Step 6:** Repeat Step 4 until the size of the new population of computer programs becomes equal to the size of the initial population  $N$ .

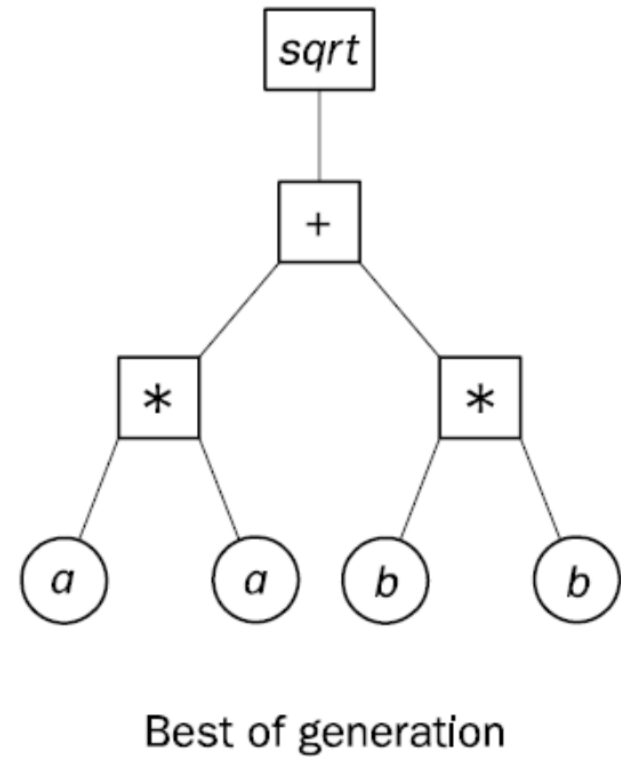
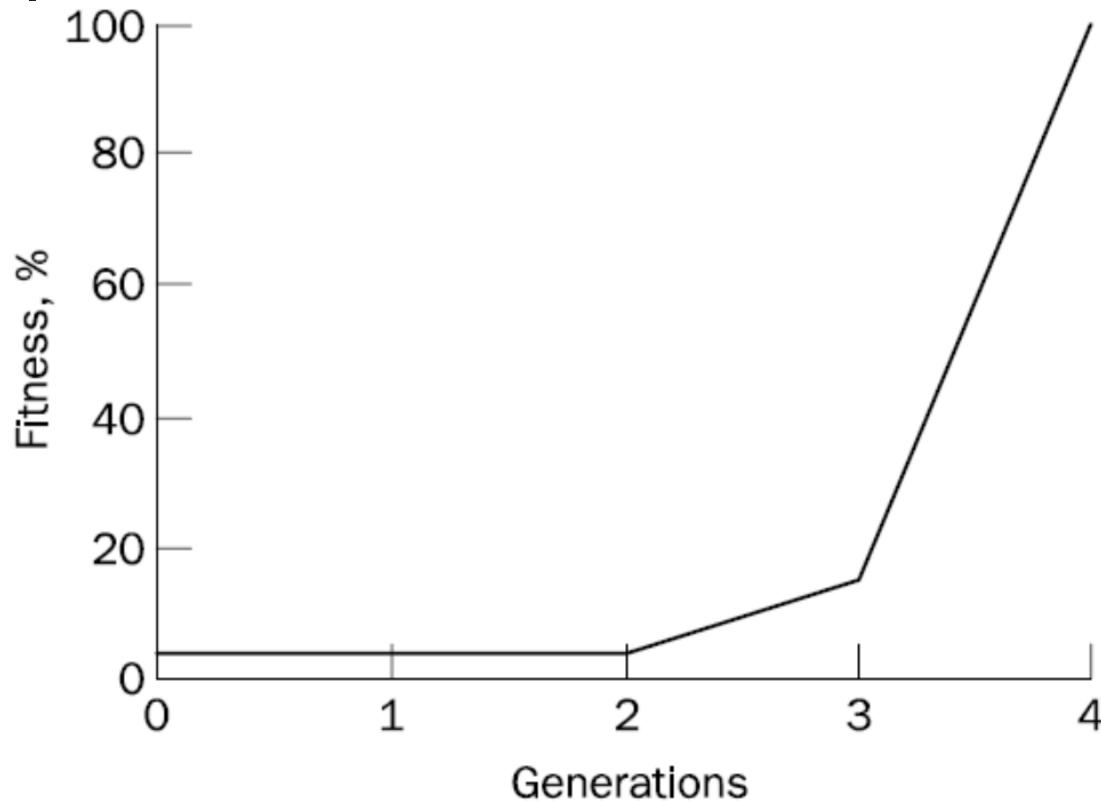
**Step 7:** Replace the current (parent) population with the new (offspring) population.

**Step 8:** Go to Step 3 and repeat the process until the termination criterion is satisfied.

# Flowchart of Steps of GP



# Fitness History of the Best S-expressions



- Initially, best S-expression has very poor fitness.
- At the 4<sup>th</sup> generation, correct S-expression is reproduced.

# Genetic Programming

---

- The simple Pythagorean Theorem example demonstrates that GP offers a general and robust method of evolving computer programs.
- In Pythagorean Theorem example, we used LISP S-expressions but there is no reason to restrict GP only to LISP S-expressions.
- GP can also be implemented in other programming languages (e.g., C, C++, Pascal, FORTRAN, Mathematica, Smalltalk) and can be applied more generally.



# Advantages of GP compared to GAs

---

- GP applies the same evolutionary approach as a GA does.
- However, GP is no longer breeding bit strings that represent coded solutions but complete computer programs that solve a particular problem.
- The fundamental difficulty of GAs lies in the problem representation. That is, in the fixed-length coding. A poor representation limits the power of a GA, and even worse, may lead to a false solution.

# Advantages of GP compared to GAs

---

- A fixed-length coding is rather artificial. As it cannot provide a dynamic variability in length, such a coding often causes considerable redundancy and reduces the efficiency of genetic search.
- In contrast, GP uses high-level building blocks of variable length. Their size and complexity can change during breeding.
- GP works well in a large number of different cases and has many potential applications.

# Difficulties of GP

---

- Despite many successful applications, there is still no proof that GP will scale up to more complex problems that require larger computer programs. And even if it scales up, extensive computer run times may be needed.

# Summary

---

- The evolutionary approach to AI is based on the computational models of natural selection and genetics known as evolutionary computation (EC). EC combines GAs, evolution strategies and GP.
- All methods of EC work as follows: create a population of individuals, evaluate their fitness, generate a new population by applying genetic operators, and repeat this process a number of times.

# Summary

---

- GAs use fitness values of individual chromosomes to carry out reproduction. As reproduction takes place, the crossover operator exchanges parts of two single chromosomes, and the mutation operator changes the gene value in some randomly chosen location of the chromosome. After a number of successive reproductions, the less fit chromosomes become extinct, while those best fit gradually come to dominate the population.

# Summary

---

- Solving a problem using GAs involves defining constraints and optimum criteria, encoding the problem solutions as chromosomes, defining a fitness function to evaluate a chromosome's performance, and creating appropriate crossover and mutation operators.

# Summary

---

- Evolution strategies (ESs) were developed by Ingo Rechenberg and Hans-Paul Schwefel in the early 1960s as an alternative to the engineer's intuition. ESs are used in technical/numerical optimization problems when no analytical objective function is available, and no conventional optimization method exists - only the engineer's intuition.

# Summary

---

- Unlike GAs, ESs use only a mutation operator. In addition, the representation of a problem in a coded form is not required.



# Summary

---

- GP is a recent development in the area of EC. It was greatly stimulated in the 1990s by John Koza. GP applies the same evolutionary approach as GAs. However, GP is no longer breeding bit strings that represent coded solutions but complete computer programs that solve a problem at hand.

# Summary

---

- Solving a problem by GP involves determining the set of arguments, selecting the set of functions, defining a fitness function to evaluate the performance of created computer programs, and choosing the method for designating a result of the run.

# Exercises

---

1. Describe the major steps of the basic genetic algorithm.
2. Present the main steps of the (1+1)-evolutionary strategy.
3. Find the maximum value of the function  $f(x)$ , where integer parameter  $x \in [a, b]$ . Describe how to encode the problem variable  $x$  as a chromosome and calculate the fitness ratio of each chromosome to the total fitness of the population of  $N$  chromosomes (e.g.,  $N = 6$ ).

## Exercises

---

4. Find the maximum value of the function  $f(x, y)$ , where real parameter  $x, y \in [-a, a]$ . Describe how to encode the problem variables  $x$  and  $y$  as a chromosome and calculate the fitness ratio of each chromosome to the total fitness of the population of  $N$  chromosomes (e.g.,  $N = 6$ ).

# Problem Encoding/Representation

---

- Binary representation
- Real representation
- Integer representation
- Order based representation
- Tree representation

# Selection

---

- Roulette wheel selection
- Elitist selection: simply takes the upper 50% of the most fit chromosomes for recombination.

# Crossover (Recombination)

---

- Single-point crossover
- Double-point crossover
- Multi-point crossover
- Uniform crossover

# Mutation

-



# Evolutionary Strategies

- $(1 + 1)$ -evolutionary strategy: one parent produces a single offspring (i.e., one child)
- $(\mu + \lambda)$ -evolutionary strategy:  $\mu$  parents are selected and  $\lambda$  children are produced. The new population size is  $\mu + \lambda$  (all parents and children compete for survival in the next generation).
- $(\mu, \lambda)$ -evolutionary strategy:  $\mu$  parents are selected and  $\lambda$  children are produced. The new population size is  $\lambda$  (only  $\lambda$  children compete for survival in the next generation).

# Differential Evolution (DE)



# Standard Deviation (SD)

- Given a data set  $X = x_1, x_2, \dots, x_n$ .
- Mean or average of  $X$ :  $\mu = \frac{1}{n} \sum_{i=1}^n x_i$  (mu)
- Variance of  $X$ :  $var(X) = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2$
- SD of  $X$ :
$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2}$$

# Normally Distributed Random Variable $a$

- The value of the normally distributed random variable  $a$  is computed as

$$a = \varphi(\mu, \sigma, x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2},$$

where  $x$  is a normal random variable.

$$\mu = 0: a = \varphi(0, \sigma, x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x}{\sigma}\right)^2}.$$

# References

---

1. Michael Negnevitsky. 2011. *Artificial Intelligence A Guide to Intelligent Systems*. 3<sup>rd</sup> Ed. Pearson. ISBN: 1408225743.
2. Crina Grosan, Ajith Abraham. 2011. *Intelligent Systems A Modern Approach*. Springer. ISBN: 3642269397.

# References

---

3. M. Tim Jones. 2008. *Artificial Intelligence A Systems Approach*. Jones & Bartlett Learning. ISBN: 0763773379.

4. Ben Coppin. 2004. *Artificial Intelligence Illuminated*. Jones & Bartlett Learning. ISBN: 0763732303.

# References

---

5. George F Luger. 2008. *Artificial Intelligence Structures and Strategies for Complex Problem Solving*. 6<sup>th</sup> Ed. Pearson. ISBN: 0321545893.
6. Stuart J. Russell, Peter Norvig. 2009. *Artificial Intelligence A Modern Approach*. 3<sup>rd</sup> Ed. Pearson. ISBN: 0136042597.

# References

---

7. Adrian A. Hopgood. 2011. *Intelligent Systems for Engineers and Scientists*. 3<sup>rd</sup> Ed. CRC Press. ISBN: 0300097603.
8. Robert J. Schalkoff. 2009. *Intelligent Systems: Principles, Paradigms, and Pragmatics*. Jones & Bartlett Learning. ISBN: 0763780170.



## Extra Slides

**function** Genetic-Algorithm(*population*,  
FITNESS-FN)

**returns** an individual // Stuart J. Russell

**inputs:** *population*, a set of individuals

FITNESS-FN, a function that measures  
the fitness of an individual

**repeat**

*new\_population*  $\leftarrow$  empty set

## Extra Slides

```
for  $i = 1$  to SIZE( $population$ ) do  
     $x \leftarrow$  RANDOM-SELECTION( $population$ ,  
                                FITNESS-FN)  
     $y \leftarrow$  RANDOM-SELECTION( $population$ ,  
                                FITNESS-FN)  
     $child \leftarrow$  REPRODUCE( $x, y$ )  
    if (small random probability) then  
         $child \leftarrow$  MUTATE( $child$ )
```

## Extra Slides

add *child* to *new\_population*

*population*  $\leftarrow$  *new\_population*

**until** some individual is fit enough, or

enough time has elapsed

**return** the best individual in *population*,

according to FITNESS-FN

## Extra Slides

**function** REPRODUCE( $x, y$ )

**returns** an individual

**inputs:**  $x, y$ , parent individuals

$n \leftarrow \text{LENGTH}(x);$

$c \leftarrow$  random number from 1 to  $n$

**return** APPEND(SUBSTRING( $x, 1, c$ ),  
SUBSTRING( $y, c + 1, n$ ))

# Extra Slides

-