



## Genetic Algorithms

---

22c: 145, Chapter 4.3



## Evolutionary Computing

---

- Genetic Algorithms
  - invented by John Holland (University of Michigan) in the 1960's
- Evolution Strategies
  - invented by Ingo Rechenberg (Technical University Berlin) in the 1960's
- Started out as individual developments, but converged in the later years



## What is Evolutionary Computation?

---

An abstraction from the theory of biological evolution that is used to create optimization procedures or methodologies, usually implemented on computers, that are used to solve problems.



## Natural Selection

---

- Limited number of resources
- Competition results in struggle for existence
- Success depends on fitness --
  - fitness of an individual: how well-adapted an individual is to their environment. This is determined by their genes (blueprints for their physical and other characteristics).
- Successful individuals are able to reproduce and pass on their genes



## The Argument

---

Evolution has optimized biological processes;

*therefore*

Adoption of the evolutionary paradigm to computation and other problems can help us find optimal solutions.



## When changes occur ...

---

- Previously "fit" (well-adapted) individuals will no longer be best-suited for their environment
- Some members of the population will have genes that confer different characteristics than "the norm". Some of these characteristics can make them more "fit" in the changing environment.

## Genetic Change in Individuals

- Mutation in genes
  - may be due to various sources (e.g. UV rays, chemicals, etc.)

Start:

1001001001001001001

After Mutation:

1001001001001001001

*Location of Mutation*

## The Nature of Computational Problems

- Require search through many possibilities to find a solution
  - (e.g. search through sets of rules for one set that best predicts the ups and downs of the financial markets)
- Search space too big -- search won't return within our lifetimes
- Require algorithm to be adaptive or to construct original solution
  - (e.g. interfaces that must adapt to idiosyncrasies of different users)

## Genetic Change in Individuals

- Recombination (Crossover)
  - occurs during reproduction -- sections of genetic material exchanged between two chromosomes

## Why Evolution Proves to be a Good Model for Solving these Types of Problems

- Evolution is a method of searching for an (almost) optimal solution
  - Possibilities -- all individuals
  - Best solution -- the most "fit" or well-adapted individual
- Evolution is a parallel process
  - Testing and changing of numerous species and individuals occur at the same time (or, in parallel)
- Evolution can be seen as a method that designs new (original) solutions to a changing environment

## Recombination (Crossover)

### Chromosome Crossing-over

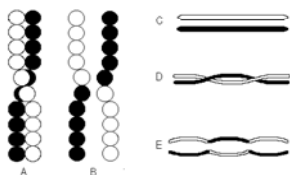


Image from <http://esg-www.mit.edu:8001/bio/mg/meiosis.html>

## The Metaphor

EVOLUTION

PROBLEM SOLVING

Individual	↔	Candidate Solution
Fitness	↔	Quality
Environment	↔	Problem

## Genetic Algorithms

- Closely follows a biological approach to problem solving
- A simulated population of randomly selected individuals is generated then allowed to evolve

## Fitness Function

- For each individual in the population, evaluate its relative fitness
- For a problem with  $m$  parameters, the fitness can be plotted in an  $m+1$  dimensional space

## Encoding the Problem

- Example: Looking for a new site which is closest to several nearby cities.
- Express the problem in terms of a bit string

$z = (1001010101011100)$

where the first 8 bits of the string represent the X-coordinate and the second 8 bits represent the Y-coordinate

## Sample Search Space

- A randomly generated population of individuals will be randomly distributed throughout the search space

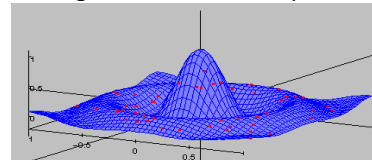


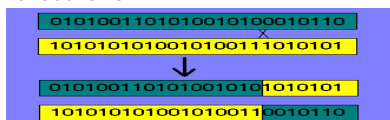
Image from <http://www2.informatik.uni-erlangen.de/~jacob/Evoluca/Java/MultiModalSearch/rats.017/Surface.gif>

## Basic Genetic Algorithm

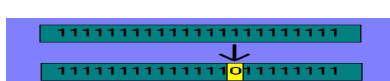
- Step 1. Generate a random population of  $n$  chromosomes
- Step 2. Assign a fitness value to each individual
- Step 3. Repeat until  $n$  children have been produced
  - Choose 2 parents based on fitness proportional selection
  - Apply genetic operators to copies of the parents
  - Produce new chromosomes

## Genetic Operators

- Cross-over

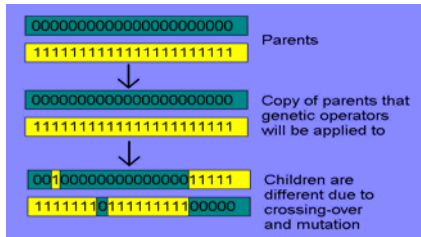


- Mutation



## Production of New Chromosomes

- 2 parents give rise to 2 children



## Ultimate Goal

- Each subsequent generation will evolve toward the global maximum
- After sufficient generations a near optimal solution will be present in the population of chromosomes

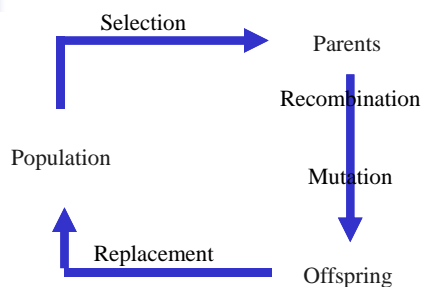
## Generations

- As each new generation of  $n$  individuals is generated, they replace their parent generation
- To achieve the desired results, typically 500 to 5000 generations are required

## Dynamic Evolution

- Genetic algorithms can adapt to a dynamically changing search space
- Seek out the moving maximum via a parasitic fitness function
  - as the chromosomes adapt to the search space, so does the fitness function

## The Evolutionary Cycle



## Basic Evolution Strategy

1. Generate some random individuals
2. Select the  $p$  best individuals based on some selection algorithm (fitness function)
3. Use these  $p$  individuals to generate  $c$  children
4. Go to step 2, until the desired result is achieved (i.e. little difference between generations)

## Encoding

- Individuals are encoded as vectors of real numbers (object parameters)
  - $op = (o_1, o_2, o_3, \dots, o_m)$
- The strategy parameters control the mutation of the object parameters
  - $sp = (s_1, s_2, s_3, \dots, s_m)$
- These two parameters constitute the individual's chromosome

## Mutation

- Needed to add new genes to the pool
  - optimal solution cannot be reached if a necessary gene is not present
  - bad genes filtered out by evolution
- Random changes to the chromosome
  - object parameter mutation
  - strategy parameter mutation
    - changes the step size used in object parameter mutation

## Fitness Functions

- Need a method for determining if one solution is more optimal than another
- Mathematical formula
- Main difference from genetic algorithms is that only the most fit individuals are allowed to reproduce (elitist selection)

## Discrete Recombination

- Similar to crossover of genetic algorithms
- Equal probability of receiving each parameter from each parent  
(8, 12, 31, ... , 5) (2, 5, 23, ... , 14)  
↓  
(2, 12, 31, ... , 14)

## Forming the Next Generation

- Number of individuals selected to be parents ( $\rho$ )
  - too many: lots of persistent bad traits
  - too few: stagnant gene pool
- Total number of children produced ( $c$ )
  - limited by computer resources
  - more children  $\Rightarrow$  faster evolution

## Intermediate Recombination

- Often used to adapt the strategy parameters
- Each child parameter is the mean value of the corresponding parent parameters  
(8, 12, 31, ... , 5) (2, 5, 23, ... , 14)  
↓  
(5, 8.5, 27, ... , 9.5)

## Example: Find the max value of $f(x_1, \dots, x_{100})$ .

- Population: real vectors of length 100.
- Mutation: randomly replace a value in a vector.
- Combination: Take the average of two vectors.

## Forming the Next Generation

- Similar operators as genetic algorithms
  - mutation is the most important operator (to uphold the principal of strong causality)
  - recombination needs to be used in cases where each child has multiple parents
- The parents can be included in the next generation
  - smoother fitness curve

## Evolution Process

- $p$  parents produce  $c$  children in each generation
- Four types of processes:
  - $p, c$
  - $p/r, c$
  - $p+c$
  - $p/r+c$

## $p+c$

- $p$  parents produce  $c$  children using mutation only (no recombination)
- The fittest  $p$  individuals (parents or children) become the parents of the next generation
- $p/r+c$  is the above with recombination

## $p, c$

- $p$  parents produce  $c$  children using mutation only (no recombination)
- The fittest  $p$  children become the parents for the next generation
- Parents are not part of the next generation
- $c \geq p$
- $p/r, c$  is the above with recombination

## Tuning a GA

- "Typical" tuning parameters for a small problem

Population size:	50 – 100
Children per generation:	= population size
Crossovers:	0 – 3
Mutations:	< 5%
Generations:	20 – 20,000

- Other concerns
  - population diversity
  - ranking policies
  - removal policies
  - role of random bias

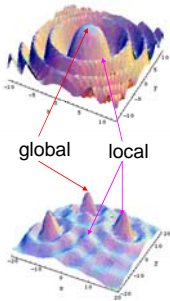
## Domains of Application

- Numerical, Combinatorial Optimization
- System Modeling and Identification
- Planning and Control
- Engineering Design
- Data Mining
- Machine Learning
- Artificial Life

## GA vs. Local Beam Search

- The starting part is the same
- GA uses both mutation and combination for next generations
- LBS uses all the neighbors (similar to mutation)
- Both keep best candidates for next round.

## Local Beam Search

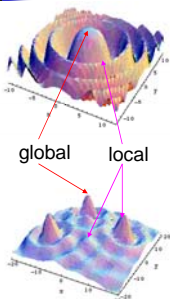


- A generalization of Hill-climbing
- Start with  $p$  candidates
- Keep the best  $p$  neighbors of these  $p$  candidates in the next round.

## GA vs. Stochastic Beam Search

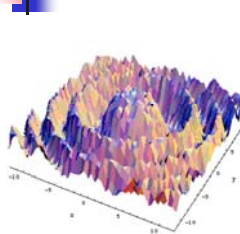
- The starting part is the same
- GA uses both mutation and combination for next generations
- SBS picks random neighbors (very much like mutation)
- GA keeps best candidates for next round.
- SBS may keep worse candidates for next round.

## Stochastic Beam Search



- Similar to Local Beam Search
- Randomly choose  $p$  neighbors in the next round.
- Keep better neighbors
- Keep worse neighbors with a probability

## GA suitable for Rugged Terrain

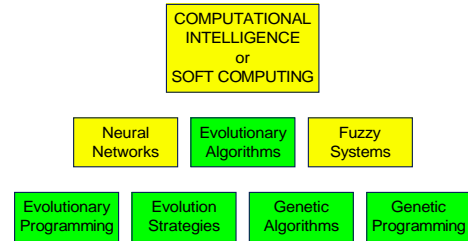


- More of a challenge to optimize
  - easy to get stuck in many local maxima
- Need to adjust mutation and crossover rates

## Drawbacks of GA

- Difficult to find an encoding for a problem
- Difficult to define a valid fitness function
- May not return the global maximum

## Taxonomy



## Why use a GA?

- requires little insight into the problem
- the problem has a very large solution space
- the problem is non-convex
- does not require derivatives
- objective function need not be smooth
- variables do not need to be scaled
- fitness function can be noisy (e.g. process data)
- when the goal is a *good* solution

## When NOT to use a GA?

- if *global* optimality is required
- if problem *insight* can:
  - significantly impact algorithm performance
  - simplify problem representation
- if the problem is highly constrained
- if the problem is smooth and convex
  - use a gradient-based optimizer
- if the search space is very small
  - use enumeration