Evolutionary Computing Cheatsheet

Terminology

Phenotype External properties of an individual.
Genotype Encoding of the phenotype.
Gene Functional unit of inheritance.
Locus Gene, variable, placeholder within the genetic code.
Allele A value on a gen.
Generation Each timestep in the Evolutionary Algorithm.
μ The number of individuals within a generation.
λ The number of offsprings (new individuals) created at each generation.

Components of an Evolutionary Algorithm (EA)

Representation Codification as an array of bits, float values, permutations.
Population Set of individuals.
Fitness Function Score of each individual.
Parent Selection Selection mechanism for mating.
Variation Operators *Mutation* (unary) and *Recombination* or *Crossover* (binary, or n-ary).
Survivor Selection Replacement of individuals for next generation.
Terminal Condition Indicates when the algorithm finishes, based on a threshold value for the fitness function, CPU-time consumed, number of generations, number of fitness evaluation, population diversity.

Basic Algorithm

```
BEGIN
  random INITIALIZATION of population
  EVALUATE each individual

  DO WHILE( NOT termination_condition )
    SELECT parents
    RECOMBINATE pairs parents
    MUTATE offsprings
    EVALUATE new individuals
    SELECT individuals to survive
  END WHILE

  SELECT best individual
END
```

Techniques

Different techniques exist in the **EA** arena, all using most of the standard components. Each of them is designed for a particular problem, being the representation the of solution and the variation operators the main differences amongst them. We will cover Genetic Algorithms, Evolutionary Strategies and Genetic Programming.

Genetic Algorithms

Representation

Binary String of bits.
Integer Ordinal (ordered) attributes or cardinal (not ordered) attributes.
Real Values Each gene is a float/double number.
Permutation Genotype satisfies the permutation constraint, genes values cannot be repeated on the same genotype. Usually employed for order and constraint problems.

Mutation Operator

The mutation operator is executed at gen level, with a probability p_m. For some operators and representations it is also useful to specify p_i the probability of applying a mutation to an individual.

For *binary* representation:
Flipping Flipping the gen.

For *integer* representation:
Random Setting Set the value of each gen randomly.
Creep For ordinal attributes, add a small value to each gen.

For *floating* representation:
Uniform Set the value of each gen randomly using a uniform distribution.
Non-Uniform Set the value of each gen randomly using a given probability distribution.

For *Permutation* representation:
Swap Swap two gens randomly chosen.
Scramble Scramble a contiguous subset of gens randomly chosen.
Insertion Pick randomly two positions, put the latter after the former one, shifting the ones in between.
Inversion Invert gens on a contiguous randomly chosen subset.

Recombination Operator

Parents are represented as P_1, P_2 and offsprings as O_1, O_2.

For *binary* representation:
One Point X-over Pick a gen at random, crossover left and right parts of the genotype from each of the parents.
N Point X-over Same idea but the genotype is divided in N+1 regions to be crossed-over.
Uniform Cross-over genes from each of the parents with a given probability.

For *integer/floating* representation:
Discrete Recombination Similar to x-over operators for binary representation.
Arithmetic $O_1 = \alpha P_1 + (1-\alpha)P_2$, O_2 same, but swapping P_1, P_2

For *Permutation* representation, the operator must maintain the permutation constraint of non-repetition of alleles.
PMX Partially Map Crossover. It migh break lots of links.
Edge Conserve as far as possible edges presented in both parents.
Order Choose a contiguous subset of gens from P_1, put them in O_1 in the same position, and fill the others gens of O_1 with gens in P_2, keeping same order as in P_2 but avoiding repetitions.
Cycle Preserve cycles as much as possible, conserving relative positions of elements.

Population Models

Generational Model Usually $\mu = \lambda$, and the whole generation is replaced at once.
Steady-State $\lambda < \mu$, there is a generational gap $\frac{\lambda}{\mu}$ of the population replaced. Old individuals might live for several generations.

Parent Selection

FPS Fitness Proportional Selection, the probability of choosing individual i is proportional of its fitness value f_i. Main drawbacks are premature convergence, if individuals have similar fitness values it induces a low selection preassure, and it is senstivie to a transposition of the fitness function, i.e., for a new fitness function $f' = f + k$ does not behaves as f.
Ranking Selection It induces a constant selection preassure sorting the population based on the fitness value and allocating selection probabilities based on the rank, for example, the exponential distribution: $P_{exp-rank}(i) = \frac{1-e^{-ik}}{c}$, with i the rank, k a constant specifying the selection preassure and c a normalizing constant so that probabilities add up one.

Implementation of Selection Probabilities

These are algorithms to implement the selection of individuals based on a given selection rule.
Roulette Wheel A Spinning wheel with the size of each hole proportional to the probability. It does not provive good sampling distributions.
Stochastic Universal Sampling It solves the bad behaviour of the previous one.
Tournament Previous methods require knowledge of the whole population, not suitable for distributed computation. Tournament of size k, with a probability p of the best member being selected, select k individuals, compare them in a tournament and choose the best with probability p. If $p = 1.0$ we have deterministic tournament, otherwise, stochastic tournament. Tournaments can be with or without replacement.

Survivor Selection

From $\mu + \lambda$ individual, we have to end up with a new generation of μ. The selection can be done based on:

Age-Based Discard individuals living more than a given number of generations.

Fitness based As the parent selection method, based on rank, tournaments...

Replace Worst Discard the λ worst individual, it might show premature convergence unless $\lambda \ll \mu$.

Elitism Keep best α individuals.

Evolutionary Strategies

The technique is commonly used for continuous parameter optimization, i.e., to look for optimal parameter minimizing functions $\mathbb{R}^n \mapsto \mathbb{R}$. The parameters of the strategy are included on the cromosome, and coevolve themselves with the solutions, it is an initial attemp of **self-adaptation**.

Representation

Cromosomes are represented using real values, and they are split in the object part, representing the parameters to be optimized and the strategy parameter, which control the evolutionary process. The chromosomes take the form of $< \hat{x}, \hat{\sigma}, \hat{\alpha} >=< x_1, \cdots, x_n, \sigma_1, \cdots, \sigma_{n_\sigma}, \alpha_1, \cdots, \alpha_{n_\alpha} >$, where usually, $n_\sigma = 1$ or n, and $n_\alpha = 0$ or $\frac{n_\alpha(n_\alpha - 1)}{2}$.

Mutation

It is based on gaussian addition to the values of the parameters. Depending on the strategy parameters evolution, it can be:

Uncorrelated, one Step Size $\sigma' = min(\epsilon, \sigma e^{\tau N(0,1)})$, $x'_i = x_i + \sigma' N_i(0,1)$, with $\tau \propto \frac{1}{\sqrt{n}}$, the **learning rate**.

Correlated, n Step Size $\sigma'_i = min(\epsilon, \sigma_i e^{\tau' N(0,1) + \tau N_i(0,1)})$, $x'_i = x_i + \sigma'_i N_i(0,1)$, with $\tau' \propto \frac{1}{\sqrt{n}}$ and $\tau \propto \frac{1}{\sqrt{2\sqrt{n}}}$.

Correlated $\sigma'_i = min(\epsilon, \sigma_i e^{\tau' N(0,1) + \tau N_i(0,1)})$, $\alpha'_j = \alpha_j + \beta N(0,1)$, $\hat{x}' = \hat{x} + \hat{N}(\hat{0}, \hat{C})$, being $tan(2\alpha_{i,j}) = \frac{2c_{i,j}}{\sigma_i^2 - \sigma_j^2}$, $c_{i,j}$ the correlation between parameters σ_i, σ_j.

The main drawback of correlated strategies is that it has too many parameters, and so, the evolutionary process is slow and has convergence problems.

Recombination

Two or more parents creates just one offsprings using **discrete** or **intermediary** crossover. Usually, we use discrete recombination for the object variables, taking values either from P_1 or P_2, and intermediary recombination for the strategy parameters, $O_i = (P_1^i + P_2^i)/2$.

Parent Selection

Parents are choosen randomly, not biased by fitness.

Survivor Selection

Usually two schemas, (μ, λ), where the best μ individuals are chosen from the new offsprings, or $(\mu + \lambda)$, where the best μ individuals are chosen from population of parents and new offsprings. Usually, (μ, λ) is preferred.

Genetic Programming

While other evolutionary techniques are employed for optimization problems, this one is more a machine learning method.

Representation

The encoding of individuals is done using Parse trees as chromosomes, being able to encode arithmetic formulae, logical formulae or computer programs. Each tree represents a symbolic expression with a **function set** making up the **nodes** (for example operators $+, -, \times, /$) and a **terminal set** (number, or constants) making up the leaves of the tree. Each function on the function set has an arity, a function with arity n, will have associated a node in the three with n branches hanging from it.

Mutation

Usually is performed by replacing a node by a random subtree. Other alternatives can be changing a function on a node by another random function of the same arity, or by changing terminal values by others.

Recombination

Subtree crossover, given two parents, choose randomly two nodes from each of them, and swap everything under the two nodes between the two parents.

Parent Selection

Usually based on fitness.

Survivor Selection

Usually generational, but many other schema are possible.

Initialization

Creation of random trees with a given maximum depth D_{max}.

Bloat

On genetic programming, usually bigger trees takes over the whole population (survival of the fattest) and trees tend to grow to infinity (**bloat effect**). There are many methods of fixing this, such as limiting the size of the resulting trees on mutation and recombination operators, as well as prunning.

Learning Classifier Systems

As before, they are more a machine learning technique. They use evolutionary techniques to create and evolve new rules which best adapt to an external environment. The rules are encoded, mutated and recombinated similarly to previous method, and each rule has an associated credit depending on how well it has performed previously, equivalent to a fitness function. Each individual encodes a rule, which for example can be of the form: if input a is on and input b is off, then do some action: `condition : action - credit`

When the System receives a message (sensor readings for example) from the external world, it looks within the population of rules (Rule Set) the ones whose condition part match the given input (Match Set). Then, it estimates the credit it would receive by executing the associated action, choosing the best rule. Depending on the outcome of performing that action, the rule, or the rules with the same actions (Action Set) will be allocated a credit. The evolutionary methods for creating new rules uses this credit as a way equivalent to fitness functions.

Parameter Control in Evolutionary Algorithms

The parameters which control the evolutionary process, such as mutation probability, crossover probability, internal parameters for particular mutation and crossover methods, parameters which control the selection preassure on the parent and survival selection... can be predefined to achieve optimal behaviour (**parameter tuning**). On the general case, these optimal values are unknown and difficult to deduce, because of interdependencies and the large number of trials one must do. **Parameter control** techniques modify these parameters during the execution of the evolutionary process in order to achieve better performance.

Parameter control techniques tackle the problem on different fronts:

- *What is changed?:* Most frequent solution is to change variation operators and their probabilities, the selection operators (for mating and replacement) and the population size and topology. Things such as representation of individuals and evaluation functions are not usually changed.
- *How the change is made?:* We can change it using a **deterministic rule** (function of number of generation for example, or diversity), with an **adaptive parameter control**, controlled by feedback (mean fitness, diversity), or with a **self-adaptive parameter control**, where the parameters evolve along with the individual, as in the evolutionary strategies.

Multimodal problems and Spacial Distributions

Simplest evolutionary techniques treat the population as a unique entity. Evolution in nature usually takes places in smaller population groups, which are highly interconected within the group, and communicate with other groups often. For **multimodal problems**, where there might be many local maxima worse than the global optimum point, dividing the population into subgroups, each of one looks on a smaller region of the solution space, is a good idea.

The underlying idea is that the population is divided into **demes** or subgroups, where all individuals communicate with each other, and occasionally, individuals from other close

demes migrate and provide new genetic material. So, each **deme** is specialized on a region of the solution space and oftenly incorporate knowledge from other **demes**. This idea follows the theory of **Puntuated Equilibria** which states that periods of evolutinary stability are interrumped by rapid growth when the population is invaded by individuals from other isolated groups of the same specie.
This spacial distribution can be implemented in several ways, which exploit very well the paralell computation capacity of the machines. Each subgroup evolves in a different process, and after a given number of **epoches**, a migration of some individuals (best from the group, randomly chosen...) is forced, adding variety to the subgroups. This migration can be forced after a fixed number of generations or when diversity of subgroups drops below a threshold.
Other schema simulates a **diffusion process** where each cell is occupied typically by one or just a few individuals, and can only communicate with neighbours.

Hybridization with other techniques

To boost the performance of evolutionary algorithms, we can mix these algorithms with other techniques adapted to a particular problem. Evolutionary Algorithm explore a wide range of the solution space in order to find the optimal solution, however, other methods (*local search*, *heuristics*) can guide the evolutionary process to the best candidate solutions, reducing the search space.
There are several ways to guide the evolutionary process to build **hybrid Evolutionary Algorithms**.

- Initialization: We initialize the individuals of the population with non-random individuals with good fitness, provided by previous known approximate solutions or heuristic method. However, this might bias the evolutionary search process to non-optimal solutions.
- Intelligent Variation Operators: Defining mutation and recombination operators adapted to the problem being solved, using specific knowledge such as heuristics which bias the operators to certain kind of individuals.

Constraint handling

EA has been successfully applied to **free optimisation problems** where the domain to search for the solution is the entire domain. **Constrain Satisfaction Problems, CSP** are problems with no objective function, but with constrains, so the solution to find is one which simply satisfies all constrains. **Constrained Optimization Problems, COP** are those with an objective function to minimize where the search space is constrained on some limits.
One way to solve constrained problems is to transform **CSP, COP** to free optimization problems, with **penalty function** for example. If the problem cannot be transformed to a free optimization problem, we can still apply EA, for example, incorporating the constrains on the representation of the individual, forcing all individuals to satisfy the constrain by design, limiting thus search to the **feasible region**. This method forces us to use mutation and recombination operators which enforces the constrains. Another popular method is to run the evolutioary strategy as usual and then fix the individuals which do not satisfy the constraints.

Working with Evolutionary Algorithms

The steps to follow to solve a problem with EA are: defining the representation of individuals, finding good variational operators and finding optimal parameters. Once we have the algorithm defined, we have to test its performance.
Since EA are by nature stochastic algorithm, usually we have to run the simulation several times in order to see if the solutions provided are stable and consistent.

Performance Measures

When the solution of the problem is known, for example, in constraint satisfaction problem, the performance is usually measured using **SR, success rate**, the number of successful execution over all executions. For those problems whose solution is not known, we use **MBF, Mean Best Fitness**, averaging the fitness value of the best individual on each execution. In order to compare performance, we can use metrics such as **AES, Average number of Evaluations to a Solution**. When comparing algorithms, we have to decide if we want **Peak or Average Performance**. The first case will be interesting for one-off problems, where a single solution is desired. For repetitive on-line problems where the algorithm must be run many times, we might be more interested on **Average Performance**.

Test Functions

Appart from the classical problems, E.A. uses quite often a batch of test function to evaluate performance of the algorithms. Classical test function are *sphere model*, *Ackley functions*, *Schwefel function*, *Rastrigin function*, *DeJong functions*. Each of them are specific to a particular class of problems, such as multimodal or unimodal problems.

Bibliography

- A. E. Eiben and J. E. Smith, Introduction to evolutionary computation, Springer-Verlag, 2003.
- David Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, 1989.
- John R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, 1992.

Copyright © 2012 Victor Montiel

`$Revision: 1.0 $, $Date: 2012/10/02 21:34:00 $.`
http://www.victormontielargaiz.net/