CSC 323 EVOLUTIONARY COMPUTATION LECTURE NOTES 2

Fitness Landscape Analysis

Fitness landscape is a powerful metaphor which facilitates the visualization of the relationship between configuration space and fitness values. The concept of fitness landscape was first introduced by biologist Sewall Wright in 1932. Visualization of the fitness landscape offers an intuitive way of understanding a problem though caution is required in high dimensional spaces as it can be misleading. Formally, the fitness landscape is a triple (X;N;f), where X is the search space, N is the neighborhood operator function, and f is the objective function $f : X \to R$. The neighborhood operator represents how the solutions are connected in the landscape, and how one can move from one solution to another.

Features of Fitness Landscape

1. Modality

When trying to assess the difficulty of a problem, a feature of the fitness landscape that might seem to be the most obvious, is the number of local optima. One would think that a unimodal problem with single optima would be easier to search than a multimodal problem, but this is not always true. Horn and Goldberg have shown a unimodal problem that is hard to search and an extremely maximally multimodal problem, where half of the points in the search space are local optima, yet it is easy to search. Therefore, relying on the number of local optima alone as an indicator of problem hardness is neither sufficient nor necessary. Nevertheless, local optima and their number could still provide valuable information about the landscape. Examples of the information that can be studied about the local optima, alongside their frequency, are: the distribution of the optima over the search space, examining if fitter ones cluster together, examining if they form a valley in the search space and the difference in quality between the local and the global optimum.

2. Basins of attractions

Another important feature of an optimum is its basin of attraction, unlike modality; basins of attraction could provide more information about the landscape. For an optimum $x^* \in X$, is the set of points that leads to it after applying local search to them, $B(x^*) = \{x \in X \mid \text{localsearch}(x) = x^*\}$ Pitzer et al. note that a basin of attraction can be classified as strong or weak according to the set of points that belongs to it. A strong basin of attraction is a basin on which all the points in the basin converge exclusively to a certain optimum. A weak basin of attraction is a basin, which has some points that could converge to another optimum depending on the algorithm or operators used.

3. Ruggedness

Another feature of fitness landscapes that is related to local optima is ruggedness (rugged landscapes assumed to have many local optima). Ruggedness is one of the first proposed

methods to measure problem hardness. The first measure of ruggedness was introduced by Weinberger, where he defines the autocorrelation function and the correlation length. Autocorrelation function measures the correlation of fitness values of neighboring steps in a random walk. An important assumption for this measure to accurately characterize the correlation structure of the entire landscape is that the fitness landscape should be statistically isotropic. This means that the random walk is \representative" of the fitness landscape, regardless of the starting point. The correlation length can be defined as the maximal distance between two points in the walk where the correlation between them is still statistically significant. Smoother landscapes have larger correlation lengths. Autocorrelation and correlation length have been used successfully as a measure of problem hardness for some problem classes

4. Neutrality

Neutrality is another feature of fitness landscapes, which refers to the amount of neutral areas or plateaux in the landscape. The problem with neutrality is that it does not provide any guidance for search heuristics which could lead the search process to wander randomly in the neutral areas for a long time without much progress. Nevertheless, exploring neutral areas could be useful sometimes by allowing the search process to reach better quality solutions and escape nearly local optima [101]. A neutral walk is introduced which is a variation of a random walk to explore neutral areas. A neutral walk starts at a random point and continuously moves to neighboring points with equal fitness such that the distance to the starting point is maximized. The maximum distance obtained from this walk could then be used as a measure of neutrality. There are interesting features to study about neutral areas in a given landscape such as the size of neutral areas, types of neutral areas, the maximum and average distances between two points in the neutral area.

5. Position types and their distributions

We have seen how local optima and neutrality can be useful for characterizing landscapes. Another view that could help in gaining more insight about the structure of the landscape is to look at the distribution of different search position types in the search space. For a given point in the landscape, according to the topology and fitness of its direct neighborhood, it can belong to seven different types of search position. Collecting information about these types could be helpful in trying to understand the behaviour of local search

6. Fitness distance correlation

Fitness distance correlation (FDC) is perhaps one of the most popular measures of problem hardness. It was proposed by Jones and Forrest to measure the correlation between fitness values and the distances to the global optimum. The main motivation behind this measure is that the relation between distances and fitness values can be an attribute of problem difficulty. Considering a maximization problem, a large and positive correlation coefficient indicates a misleading problem; a correlation coefficient near to zero indicates a difficult problem, while a

large and negative correlation coefficient indicates a straightforward problem. An obvious drawback of this measure is that it requires the knowledge of the global optimum. This could be alleviated by considering the best known solution since in many situations when applying meta-heuristics, the goal is to find a good enough solution.

7. Epistasis

Epistasis is one of the earliest attempts to measure problem difficulty. It is computed based on the fitness function only. It studies the interaction between the solution components in an attempt to measure the amount of non-linearity in the fitness function.

One of the first attempts to quantify epistasis was made by researcher when he proposed epistasis variance. A high epistasis means that the variables depend on each other while low epistasis means that the variables are independent of each other. Problems with high epistasis are assumed to be hard to optimize. This method, however, is found to be difficult to interpret and works only for limited number of cases.

8. Information analysis

A different method to study the structure and ruggedness of the fitness landscapes is the information analysis. It is inspired by the concept that the information content of a system can be used as a measure of how difficult it is to describe that system. The main idea is to use the amount of information needed to describe a random walk in the fitness landscape of a problem as a difficulty measure of the problem. Difficult problems are assumed to require more information to describe a random walk in their landscape. This measure is derived from a sample of the search space and assumes that the landscape is statistically isotropic. A related idea is the concept of information landscapes

9. Evolvability

Evolvability involves studying the dynamic properties of certain meta-heuristics searching the landscape (e.g. evolving population). It studies the chances of improving a certain solution by measuring the correlation of successive solutions. The general concept is that a higher degree of evolvability indicates that the problem is easier for the metaheuristics.

Some of the methods that have been proposed to measure evolvability are:

evolvability portraits, fitness cloud and fitness-probability cloud

10. Spectral landscape analysis

Stadler was the first to introduce the study of isotropic fitness landscapes using Fourier Analysis, suggesting another approach to analyse fitness landscapes by decomposing the fitness landscape of an arbitrary problem into super positions of elementary landscape. An elementary landscape is a special type of landscape that can be described by Grover's wave equation. This method has been used extensively to analyse many popular problems.

11. Network measures

Recent attempts to develop predictive models of problem difficulty utilized the fact that a fitness landscape can be represented as a network, and thus used complex network analysis tools to analyse the fitness landscape. A local optima network (LONs) was the first method that employed complex network analysis tools to combinatorial optimization problems.

Co-evolution

Cooperative Coevolution (CC) is an evolutionary computation method that divides a large problem into subcomponents and solves them independently in order to solve the large problem. The subcomponents are also called species. The subcomponents are implemented as subpopulations and the only interaction between subpopulations is in the cooperative evaluation of each individual of the subpopulations. The general CC framework is nature inspired where the individuals of a particular group of species mate amongst themselves; however, mating in between different species is not feasible. The cooperative evaluation of each individual in a subpopulation is algorithms in particular, since they are guided mainly by their direct neighborhood done by concatenating the current individual with the best individuals from the rest of the subpopulations as described by M. Potter.

The cooperative coevolution framework has been applied to real world problems such as pedestrian detection systems,[3] large-scale function optimization[4] and neural network training. It has also been further extended into another method, called Constructive cooperative coevolution.

MULTI-OBJECTIVE OPTIMIZATION

Multi-objective optimization (also known as multi-objective programming, vector optimization, multicriteria optimization or Pareto optimization) is an area of multiple criteria decision making that is concerned with mathematical optimization problems involving more than one objective function to be optimized simultaneously. Multi-objective optimization has been applied in many fields of science, including engineering, economics and logistics where optimal decisions need to be taken in the presence of trade-offs between two or more conflicting objectives. Minimizing cost while maximizing comfort while buying a car, and maximizing performance whilst minimizing fuel consumption and emission of pollutants of a vehicle are examples of multi-objective optimization problems involving two and three objectives, respectively. In practical problems, there can be more than three objectives.

For a nontrivial multi-objective optimization problem, no single solution exists that simultaneously optimizes each objective. In that case, the objective functions are said to be conflicting, and there exists a (possibly infinite) number of Pareto optimal solutions. A solution is called nondominated, Pareto optimal, Pareto efficient or noninferior, if none of the objective

functions can be improved in value without degrading some of the other objective values. Without additional subjective preference information, all Pareto optimal solutions are considered equally good. Researchers study multi-objective optimization problems from different viewpoints and, thus, there exist different solution philosophies and goals when setting and solving them. The goal may be to find a representative set of Pareto optimal solutions, and/or quantify the trade-offs in satisfying the different objectives, and/or finding a single solution that satisfies the subjective preferences of a human decision maker (DM).

Examples of applications

1. Economics

In economics, many problems involve multiple objectives along with constraints on what combinations of those objectives are attainable. For example, consumer's demand for various goods is determined by the process of maximization of the utilities derived from those goods, subject to a constraint based on how much income is available to spend on those goods and on the prices of those goods. This constraint allows more of one good to be purchased only at the sacrifice of consuming less of another good; therefore, the various objectives (more consumption of each good is preferred) are in conflict with each other. A common method for analyzing such a problem is to use a graph of indifference curves, representing preferences, and a budget constraint, representing the trade-offs that the consumer is faced with.

2. Finance

In finance, a common problem is to choose a portfolio when there are two conflicting objectives — the desire to have the expected value of portfolio returns be as high as possible, and the desire to have risk, often measured by the standard deviation of portfolio returns, be as low as possible. This problem is often represented by a graph in which the efficient frontier shows the best combinations of risk and expected return that are available, and in which indifference curves show the investor's preferences for various risk-expected return combinations. The problem of optimizing a function of the expected value (first moment) and the standard deviation (square root of the second central moment) of portfolio return is called a two-moment decision model.

3. Optimal control

In engineering and economics, many problems involve multiple objectives which are not describable as the-more-the-better or the-less-the-better; instead, there is an ideal target value for each objective, and the desire is to get as close as possible to the desired value of each objective. For example, energy systems typically have a trade-off between performance and cost[4][5] or one might want to adjust a rocket's fuel usage and orientation so that it arrives both at a specified

place and at a specified time; or one might want to conduct open market operations so that both the inflation rate and the unemployment rate are as close as possible to their desired values.

4. Optimal design

Product and process design can be largely improved using modern modeling, simulation and optimization techniques. The key question in optimal design is the measure of what is good or desirable about a design. Before looking for optimal designs it is important to identify characteristics which contribute the most to the overall value of the design. A good design typically involves multiple criteria/objectives such as capital cost/investment, operating cost, profit, quality and/or recovery of the product, efficiency, process safety, operation time etc. Therefore, in practical applications, the performance of process and product design is often measured with respect to multiple objectives. These objectives typically are conflicting, i.e. achieving the optimal value for one objective requires some compromise on one or more of other objectives.

5. Process optimization

Multi-objective optimization has been increasingly employed in chemical engineering and manufacturing. In 2009, Fiandaca and Fraga used the multi-objective genetic algorithm (MOGA) to optimize the pressure swing adsorption process (cyclic separation process). The design problem involved the dual maximization of nitrogen recovery and nitrogen purity. The results provided a good approximation of the Pareto frontier with acceptable trade-offs between the objectives.

In 2010, Sendín et al. solved a multi-objective problem for the thermal processing of food. They tackled two case studies (bi-objective and triple objective problems) with nonlinear dynamic models and used a hybrid approach consisting of the weighted Tchebycheff and the Normal Boundary Intersection approach. The novel hybrid approach was able to construct a Pareto optimal set for the thermal processing of foods.

6. Radio resource management

The purpose of radio resource management is to satisfy the data rates that are requested by the users of a cellular network.[27] The main resources are time intervals, frequency blocks, and transmit powers. Each user has its own objective function that, for example, can represent some combination of the data rate, latency, and energy efficiency. These objectives are conflicting since the frequency resources are very scarce, thus there is a need for tight spatial frequency reuse which causes immense inter-user interference if not properly controlled. Multi-user MIMO techniques are nowadays used to reduce the interference by adaptive precoding. The network operator would like to both bring great coverage and high data rates, thus the operator

would like to find a Pareto optimal solution that balance the total network data throughput and the user fairness in an appropriate subjective manner.

7. Electric power systems

Reconfiguration, by exchanging the functional links between the elements of the system, represents one of the most important measures which can improve the operational performance of a distribution system. The problem of optimization through the reconfiguration of a power distribution system, in terms of its definition, is a historical single objective problem with constraints. Since 1975, when Merlin and Back introduced the idea of distribution system reconfiguration for active power loss reduction, until nowadays, a lot of researchers have proposed diverse methods and algorithms to solve the reconfiguration problem as a single objective problem. Some authors have proposed Pareto optimality based approaches (including active power losses and reliability indices as objectives). For this purpose, different artificial intelligence based methods have been used: microgenetic, branch exchange, particle swarm optimization [32] and non-dominated sorting genetic algorithm.

8. Inspection of Infrastructure

Autonomous inspection of infrastructure has the potential to reduce costs, risks and environmental impacts, as well as ensuring better periodic maintenance of inspected assets. Typically, planning such missions has been viewed as a single-objective optimization problem, where one aims to minimize the energy or time spent in inspecting an entire target structure.[For complex, real-world structures, however, covering 100% of an inspection target is not feasible, and generating an inspection plan may be better viewed as a multiobjective optimization problem, where one aims to both maximize inspection coverage and minimize time and costs. A recent study has indicated that multiobjective inspection planning indeed has the potential to outperform traditional methods on complex structures[

EVOLUTIONARY ALGORITHM

In computational intelligence (CI), an evolutionary algorithm (EA) is a subset of evolutionary computation,[1] a generic population-based metaheuristic optimization algorithm. An EA uses mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection. Candidate solutions to the optimization problem play the role of individuals in a population, and the fitness function determines the quality of the solutions (see also loss function). Evolution of the population then takes place after the repeated application of the above operators.

In computational intelligence (CI), an **evolutionary algorithm** (**EA**) is a subset of evolutionary computation,[1] a generic population-based metaheuristic optimization algorithm. An EA uses mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection. Candidate solutions to the optimization problem play the role of individuals in a population, and the fitness function determines the quality of the solutions (see also loss function). Evolution of the population then takes place after the repeated application of the above operators.

Evolutionary algorithms often perform well approximating solutions to all types of problems because they ideally do not make any assumption about the underlying fitness landscape. Techniques from evolutionary algorithms applied to the modeling of biological evolution are generally limited to explorations of microevolutionary processes and planning models based upon cellular processes. In most real applications of EAs, computational complexity is a prohibiting factor.[2] In fact, this computational complexity is due to fitness function evaluation. Fitness approximation is one of the solutions to overcome this difficulty. However, seemingly simple EA can solve often complex problems;[*citation needed*] therefore, there may be no direct link between algorithm complexity and problem complexity.

Implementation

The following is an example of a generic single-objective genetic algorithm.

Step One: Generate the initial population of individuals randomly. (First generation)

Step Two: Repeat the following regenerational steps until termination:

1. Evaluate the fitness of each individual in the population (time limit, sufficient fitness achieved, etc.)
2. Select the fittest individuals for reproduction. (Parents)
3. Breed new individuals through crossover and mutation operations to give birth to offspring.
4. Replace the least-fit individuals of the population with new individuals.

Types

Similar techniques differ in genetic representation and other implementation details, and the nature of the particular applied problem.

- Genetic algorithm – This is the most popular type of EA. One seeks the solution of a problem in the form of strings of numbers (traditionally binary, although the best representations are usually those that reflect something about the problem being

solved),[2] by applying operators such as recombination and mutation (sometimes one, sometimes both). This type of EA is often used in optimization problems.

- Genetic programming – Here the solutions are in the form of computer programs, and their fitness is determined by their ability to solve a computational problem.
- Evolutionary programming – Similar to genetic programming, but the structure of the program is fixed and its numerical parameters are allowed to evolve.
- Gene expression programming – Like genetic programming, GEP also evolves computer programs but it explores a genotype-phenotype system, where computer programs of different sizes are encoded in linear chromosomes of fixed length.
- Evolution strategy – Works with vectors of real numbers as representations of solutions, and typically uses self-adaptive mutation rates.
- Differential evolution – Based on vector differences and is therefore primarily suited for numerical optimization problems.
- Neuroevolution – Similar to genetic programming but the genomes represent artificial neural networks by describing structure and connection weights. The genome encoding can be direct or indirect.
- Learning classifier system – Here the solution is a set of classifiers (rules or conditions). A Michigan-LCS evolves at the level of individual classifiers whereas a Pittsburgh-LCS uses populations of classifier-sets. Initially, classifiers were only binary, but now include real, neural net, or S-expression types. Fitness is typically determined with either a strength or accuracy based reinforcement learning or supervised learning approach.

GENETIC PROGRAMMING

In artificial intelligence, **genetic programming** (**GP**) is a technique of evolving programs, starting from a population of unfit (usually random) programs, fit for a particular task by applying operations analogous to natural genetic processes to the population of programs. It is essentially a heuristic search technique often described as 'hill climbing', i.e. searching for an optimal or at least suitable program among the space of all programs.

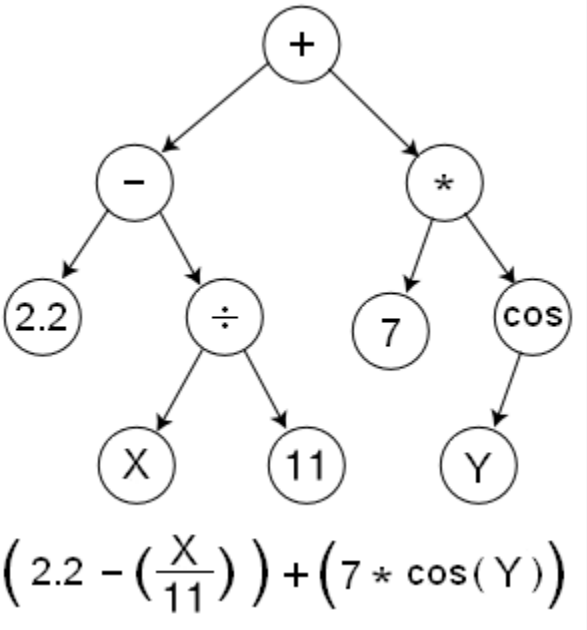
The operations are: selection of the fittest programs for reproduction (crossover) and mutation according to a predefined fitness measure, usually proficiency at the desired task. The crossover operation involves swapping random parts of selected pairs (parents) to produce new and different offspring that become part of the new generation of programs. Mutation involves substitution of some random part of a program with some other random part of a program. Some programs not selected for reproduction are copied from the current generation to the new generation. Then the selection and other operations are recursively applied to the new generation of programs.

Typically, members of each new generation are on average more fit than the members of the previous generation, and the best-of-generation program is often better than the best-of-

generation programs from previous generations. Termination of the recursion is when some individual program reaches a predefined proficiency or fitness level.

It may and often does happen that a particular run of the algorithm results in premature convergence to some local maximum which is not a globally optimal or even good solution. Multiple runs (dozens to hundreds) are usually necessary to produce a very good result. It may also be necessary to increase the starting population size and variability of the individuals to avoid pathologies.

Program representation



A function represented as a tree structure

GP evolves computer programs, traditionally represented in memory as tree structures. Trees can be easily evaluated in a recursive manner. Every tree node has an operator function and every terminal node has an operand, making mathematical expressions easy to evolve and evaluate. Thus traditionally GP favors the use of programming languages that naturally embody tree structures (for example, Lisp; other functional programming languages are also suitable).

Non-tree representations have been suggested and successfully implemented, such as linear genetic programming which suits the more traditional imperative languages [see, for example, Banzhaf *et al.* (1998)].[30] The commercial GP software *Discipulus* uses automatic induction of binary machine code ("AIM")[31] to achieve better performance. *μGP*[32] uses directed multigraphs to generate programs that fully exploit the syntax of a given assembly language. Other program representations on which significant research and development have been conducted include programs for stack-based virtual machines,[33][34][35] and sequences of integers

that are mapped to arbitrary programming languages via grammars.[36][37] Cartesian genetic programming is another form of GP, which uses a graph representation instead of the usual tree based representation to encode computer programs.

Most representations have structurally noneffective code (introns). Such non-coding genes may seem to be useless because they have no effect on the performance of any one individual. However, they alter the probabilities of generating different offspring under the variation operators, and thus alter the individual's variational properties. Experiments seem to show faster convergence when using program representations that allow such non-coding genes, compared to program representations that do not have any non-coding genes.

Selection

Selection is a process whereby certain individuals are selected from the current generation that would serve as parents for the next generation. The individuals are selected probabilistically such that the better performing individuals have a higher chance of getting selected.[18] The most commonly used selection method in GP is tournament selection, although other methods such as fitness proportionate selection, lexicase selection,[40] and others have been demonstrated to perform better for many GP problems.

Elitism, which involves seeding the next generation with the best individual (or best *n* individuals) from the current generation, is a technique sometimes employed to avoid regression.

Crossover

Various genetic operators (i.e., crossover and mutation) are applied to the individuals selected in the selection step described above to breed new individuals. The rate at which these operators are applied determines the diversity in the population.

Mutation

Flip one or more bits from the previous offspring to generate new child or generation.

APPLICATIONS

GP has been successfully used as an automatic programming tool, a machine learning tool and an automatic problem-solving engine. GP is especially useful in the domains where the exact form of the solution is not known in advance or an approximate solution is acceptable (possibly because finding the exact solution is very difficult). Some of the applications of GP are curve fitting, data modeling, symbolic regression, feature selection, classification, etc. John R. Koza mentions 76 instances where Genetic Programming has been able to produce results that are competitive with human-produced results (called Human-competitive results).[41] Since 2004, the annual Genetic and Evolutionary Computation Conference (GECCO) holds Human Competitive

Awards (called Humies) competition,[42] where cash awards are presented to human-competitive results produced by any form of genetic and evolutionary computation. GP has won many awards in this competition over the years.

Evolutionary Algorithms

Evolutionary algorithms draw inspiration from nature. An evolutionary algorithm starts with a randomly initialized population. The population then evolves across several generations. In each generation, fit individuals are selected to become parent individuals. They cross-over with each other to generate new individuals, which are subsequently called offspring individuals. Randomly selected offspring individuals then undergo certain mutations. After that, the algorithm selects the optimal individuals for survival to the next generation according to the survival selection scheme designed in advance. For instance, if the algorithm is overlapping (De Jong, 2006), then both parent and offspring populations will participate in the survival selection. Otherwise, only the offspring population will participate in the survival selection.
The selected individuals then survive to the next generation. Such a procedure is repeated again and again until a certain termination condition is met (Wong, Leung, & Wong, 2010).

The design of evolutionary algorithm can be divided into several components: representation, parent selection, crossover operators, mutation operators, survival selection, and termination condition. Details can be found in the following sections.

- **Representation:** It involves genotype representation and genotype-phenotype mapping. (De Jong,
2006). For instance, we may represent an integer (phenotype) as a binary array (genotype): '19' as
'10011' and '106' as '1101010'. If we mutate the first bit, then we will get '3' (00011) and '42' (0101010). For those examples, even we have mutated one bit in the genotype, the phenotype may vary very much. Thus we can see that there are a lot of considerations in the mapping.
- **Parent Selection:** It aims at selecting good parent individuals for crossovers, where the goodness
of a parent individual is quantified by its fitness. Thus most parent selection schemes focus on giving more opportunities to the fitter parent individuals than the other individuals and vice versa such that “good” offspring individuals are likely to be generated.
- **Crossover Operators:** It resembles the reproduction mechanism in nature. Thus they, with mutation operators, are collectively called reproductive operators. In general, a crossover operator
combines two individuals to form a new individual. It tries to split an individual into parts and then assemble those parts into a new individual.
- **Mutation Operators:** It simulates the mutation mechanism in which some parts of a genome undergoes random changes in nature. Thus, as a typical modeling practice, a mutation operator

changes parts of the genome of an individual. On the other hand, mutations can be thought as an exploration mechanism to balance the exploitation power of crossover operators.

- **Survival Selection:** It aims at selecting a subset of good individuals from a set of individuals, where the goodness of individual is proportional to its fitness in most cases. Thus survival selection mechanism is somehow similar to parent selection mechanism. In a typical framework like 'EC4' (De Jong, 2006), most parent selection mechanisms can be re-applied in survival selection.
- **Termination Condition:** It refers to the condition at which an evolutionary algorithm should end.

Parent Selection

Parent selection aims at selecting "good" parent individuals for crossover, where the goodness of a parent individual is positively proportional to its fitness for most cases. Thus most parent selection schemes focus on giving more opportunities to the fitter parent individuals than the other individuals and vice versa. The typical methods are listed as follows:

- **Fitness Proportional Selection:** The scheme is sometimes called roulette wheel selection. In the scheme, the fitness values of all individuals are summed. Once summed, the fitness of each individual is divided by the sum. The ratio then becomes the probability for each individual to be selected.
- **Rank Proportional Selection:** Individuals with high ranks are given more chances to be selected.

Unlike fitness proportional scheme, the rank proportional scheme does not depend on the actual fitness values of the individuals. It is a double-edged sword. On the positive side, it can help us prevent the domination of very high fitness values. On the negative side, it imposes additional computational costs for ranking.

- **Uniform Deterministic Selection:** The scheme is the simplest among the other schemes. All individuals are selected, resulting in uniform selection.
- **Uniform Stochastic Selection:** The scheme is the probabilistic version of uniform deterministic selection. All individuals are given equal chances (equal probabilities) to be selected.
- **Binary Tournament:** Actually, there are other tournament selection schemes proposed in the past literature. In this book chapter, the most basic one, binary tournament, is selected and described. In each binary tournament, two individuals are randomly selected and competed with each other by fitness. The winner is then selected. Such a procedure is repeated until all vacancies are filled.
- **Truncation:** The top individuals are selected deterministically when there is a vacancy for selection. In other words, the bottom individuals are never selected. For example, if there are 100 individuals and 50 slots are available, then the top 50 fittest individuals will be selected.

Crossover Operators

Crossover operators resemble the reproduction mechanism in nature. Thus they, with mutation operators, are collectively called reproductive operators. In general, a crossover operator combines two individuals to form a new individual. It tries to partition an individual into parts and then assemble the parts of two individuals into a new individual. The partitioning is not a trivial task. It depends on the representation adopted. Thus it is not hard to imagine that crossover operators are representation-dependent.

Nevertheless, without loss of generality, a list of classic crossover operators is listed as follows:

- **One Point Crossover:** One point crossover is a commonly used crossover operator because of its simplicity. Given two individuals, it randomly chooses a cut point in their genomes. Then it swaps the parts after (or before) the cut point between the two genomes.
- **Two Points Crossover:** Two points crossover is another commonly used crossover operator because people argue that one point crossover has a positional bias toward the terminal positions. For instance, when making a one point crossover, the rightmost (or leftmost) part is always swapped. Thus people propose two point crossovers to avoid the positional bias.
- **Uniform Crossover:** Uniform crossover is a general one. Each gene is given an equal probability to be swapped.
- **Blend Crossover:** Blend crossover is commonly used in real number optimization. Instead of swapping genes, it tries to blend two genes together by arithmetic averaging to obtain the intermediate values. For instance, if we are going to make a crossover between two vectors [1 2 3] and [4 5 6], then the blended vector will be [2.5 3.5 4.5]. Weights can be applied here.

Mutation Operators

Mutation operators resemble the mutation mechanism in which some parts of genome undergo random changes in nature. Thus, as a typical modeling, a mutation operator changes parts of the genome of an individual probabilistically. Similar to crossover operators, mutation operators are representation dependent.

Nevertheless, without loss of generality, a list of commonly used mutation operators is shown below:

- **Bitflip Mutation:** It is commonly used in binary genomes. Specified by a pre-defined probability, each bit in a binary genome is probabilistically inverted.
- **Random Mutation:** Random mutation is generalized from bitflip mutation. It can be applied in many genomes. Specified by a pre-defined probability, each part in a genome is probabilistically changed to a random value within domain bounds.
- **Delta Mutation:** Delta mutation is commonly used in real number genomes. Specified by a predefined probability, each real number in a real number genome is probabilistically incremented/decremented by a certain step size (called delta), where the step size is pre-specified.

Nonetheless, it is straightforward to make the step size adaptive, similar to the trial vector generations in differential evolution (Storn & Price, 1997).

Gaussian Mutation: Gaussian mutation is also commonly used in real number genomes. Similar to delta mutation, each real number in a real number genome is probabilistically increased / decreased by a step size. The difference is that the step size is a Gaussian random number. (De Jong, 2006).

Survival Selection

Survival selection aims at selecting a subset of good individuals from a population, where the goodness of individual is proportional to its fitness for most cases. Thus survival selection mechanism is somehow similar to parent selection mechanism. In a typical framework like EC4 (De Jong, 2006), most parent selection mechanisms can be re-applied in survival selection. For example, the fitness proportional selection can be applied as survival selection.

Termination Condition

Termination condition refers to the condition at which an evolutionary algorithm should end. For historical reasons, the number of generations is often adopted as the termination measurement: an evolutionary algorithm terminates when a certain number of generations has been reached (e.g. 1000 generations). Nonetheless, it has been pointed out that fitness function evaluations are computationally expensive in certain domains. Thus the number of fitness function evaluations is also adopted in some problems. If computing resources are limited, CPU time is also adopted.
Nonetheless, convergence is not guaranteed. Thus people have calculated the fitness improvement of each generation as another condition for termination.

Examples

Genetic Algorithm: Genetic algorithm is the most classic evolutionary algorithm. It draws inspiration from the Darwin's Evolution Theory. The difference between genetic algorithm and evolutionary algorithm becomes blurred nowadays. The words 'genetic algorithm' and 'evolutionary algorithm' are sometimes interchanged in use. To clearly explain the working mechanism of a genetic algorithm, we chose the canonical genetic algorithm (Whitley, 1994) as a representative example.
In the canonical genetic algorithm, each individual has a fixed-length binary array as its genotype. Then the fitness of each individual is divided by the average fitness to calculate the normalized probability to be selected. The algorithm then adopts them to select parents for one point crossover to produce offspring individuals, which subsequently undergo mutations. The offspring individuals become the population in the next generation and so forth.

Genetic Programming: Genetic programming is indeed a special type of genetic algorithm. The difference lies in their representations. Genetic programming adopts trees as genotypes to represent programs or expressions. (Figure 2 depicts an example). The typical selection schemes of evolutionary algorithms can still be used as parent selection and survival selection in genetic

programming. The distinct features of genetic programming are their crossover and mutation operators. For instance, swapping sub-trees between two trees and random generation of sub-trees.

Why do we need optimization?

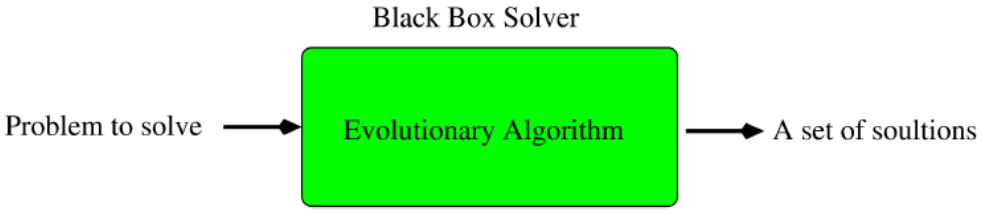
Environmental simulation models are used extensively to support decision-making processes in a variety of application areas, such as: the development and evaluation of national and international environmental regulations land use management, natural hazard management, the operation and management of reservoir systems, the assessment of environmental and human health, the management of river systems, the management of drains, the management of air pollution, the design of water distribution networks, the prediction of and adaption to natural hazards such as floods

EVOLUTIONARY COMPUTATION FOR DYNAMIC OPTIMIZATION PROBLEMS

What is Evolutionary Computation?

In computer science, evolutionary computation is a family of algorithms for global optimization inspired by biological evolution, and the subfield of artificial intelligence and soft computing studying these algorithms. In technical terms, they are a family of population-based trial and error problem solvers with a metaheuristic or stochastic optimization character.

EC encapsulates a class of stochastic optimization algorithms, dubbed
Evolutionary Algorithms (EAs)
An EA is an optimization algorithm that is
Generic: a black-box tool for many problems
Population-based: evolves a population of candidate solutions
Stochastic: uses probabilistic rules
Bio-inspired: uses principles inspired from biological evolution



EC Applications

EAs are easy-to-use: No strict requirements to problems
Widely used for optimisation and search problems

Financial and economical systems
Transportation and logistics systems
Industry engineering
Automatic programming, art and music design

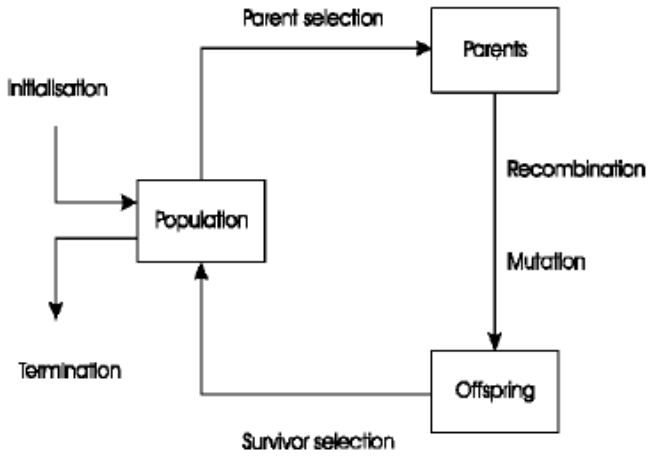
Design and Framework of EA

Given a problem to solve, first consider two key things:

- Representation of solution into individual
- Evaluation or fitness function

Then, design the framework of an EA:

- Initialization of population
- Evolve the population
    - Selection of parents
    - Variation operators (recombination & mutation)
    - Selection of offspring into next generation
- Termination condition: a given number of generations



- Traditionally, research on EAs has focused on static problems
    - Aim to find the optimum quickly and precisely

- But, many real-world problems are dynamic optimization problems (DOPs), where changes occur over time
    - In transport networks, travel time between nodes may change
    - In logistics, customer demands may change

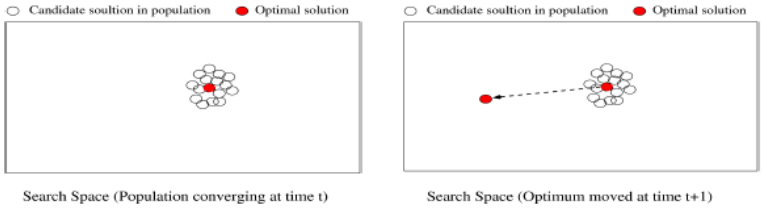
- In general terms, “optimization problems that change over time” are called *dynamic problems/time-dependent problems*

$$F = f(\vec{x}, \vec{\phi}, t)$$

  – $\vec{x}$: decision variable(s); $\vec{\phi}$: parameter(s); t: time
- DOPs: special class of dynamic problems that are solved online by an algorithm as time goes by
- Many real-world problems are DOPs
- EAs, once properly enhanced, are good choice
  - Inspired by natural/biological evolution, always in dynamic environments
  - Intrinsically, should be fine to deal with DOPs
- Many events on EC for DOPs recently

Why DOPs Challenge EC?

- For DOPs, optima may move over time in the search space
  - Challenge: need to track the moving optima over time



- DOPs challenge traditional EAs
  - Once converged, hard to escape from an old optimum

DOPs: Classification

Classification criteria:

- Time-linkage: Does the future behaviour of the problem depend on the current solution?
- Predictability: Are changes predictable?
- Visibility: Are changes visible or detectable
- Cyclicity: Are changes cyclic/recurrent in the search space?
- Factors that change: objective, domain/number of variables, constraints, and/or other parameters

Performance Measures

- For EC for stationary problems, 2 key performance measures
  - Convergence speed
  - Success rate of reaching optimality
- For EC for DOPs, over 20 measures (Nguyen et al., 2012)
  - Optimality-based performance measures
    - Collective mean fitness or mean best-of-generation
    - Accuracy
    - Adaptation
    - Offline error and offline performance
    - Mean distance to optimum at each generation
    - .....
  - Behaviour-based performance measures
    - Reactivity
    - Stability
    - Robustness
    - Satisficability
    - Diversity measures
    - .....

DOPs Common Characteristics

Common characteristics of DOPs in the literature:

- Most DOPs are non time-linkage problems
- For most DOPs, changes are assumed to be detectable
- In most cases, the objective function is changed
- Many DOPs have unpredictable changes
- Most DOPs have cyclic/recurrent changes

Performance Measures: Examples

- Collective mean fitness (mean best-of-generation):

$$\overline{F}_{BOG} = \frac{1}{G} \times \sum_{i=1}^{i=G} \left(\frac{1}{N} \times \sum_{j=1}^{j=N} F_{BOG_{ij}} \right)$$

  - G and N: number of generations and runs, resp.
  - $F_{BOG_{ij}}$: best-of-generation fitness of generation i of run j
- Adaptation performance (Mori et al., 1997)

$$Ada = \frac{1}{T} \sum_{t=1..T} (f_{best}(t)/f_{opt}(t))$$

- Accuracy (Trojanowski and Michalewicz, 1999)

LEARNING CLASSIFIER SYSTEM

Learning classifier systems, or LCS, are a paradigm of rule-based machine learning methods that combine a discovery component (e.g. typically a genetic algorithm) with a learning component (performing either supervised learning, reinforcement learning, or unsupervised learning). Learning classifier systems seek to identify a set of context-dependent rules that collectively store and apply knowledge in a piecewise manner in order to make predictions (e.g. behavior modeling, classification, data mining, regression, function approximation, or game strategy). This approach allows complex solution spaces to be broken up into smaller, simpler parts.

The founding concepts behind learning classifier systems came from attempts to model complex adaptive systems, using rule-based agents to form an artificial cognitive system (i.e. artificial intelligence).

Elements of a generic LCS algorithm

Keeping in mind that LCS is a paradigm for genetic-based machine learning rather than a specific method, the following outlines key elements of a generic, modern (i.e. post-XCS) LCS algorithm. For simplicity let us focus on Michigan-style architecture with supervised learning.

Environment

The environment is the source of data upon which an LCS learns. It can be an offline, finite training dataset (characteristic of a data mining, classification, or regression problem), or an online sequential stream of live training instances. Each training instance is assumed to include some number of *features* (also referred to as *attributes*, or *independent variables*), and a single *endpoint* of interest (also referred to as the class, *action*, *phenotype*, *prediction*, or *dependent variable*). Part of LCS learning can involve feature selection, therefore not all of the features in the training data need be informative. The set of feature values of an instance is commonly referred to as the *state*. For simplicity let's assume an example problem domain with Boolean/binary features and a Boolean/binary class. For Michigan-style systems, one instance from the environment is trained on each learning cycle (i.e. incremental learning). Pittsburgh-style systems perform batch learning, where rule-sets are evaluated each iteration over much or all of the training data.

Rule/classifier/population

A rule is a context dependent relationship between state values and some prediction. Rules typically take the form of an {IF:THEN} expression, (e.g. {*IF 'condition' THEN 'action'},* or as a more specific example, *{IF 'red' AND 'octagon' THEN 'stop-sign'}*). A critical concept in LCS and rule-based machine learning alike, is that an individual rule is not in itself a model, since the

rule is only applicable when its condition is satisfied. Think of a rule as a "local-model" of the solution space.

Rules can be represented in many different ways to handle different data types (e.g. binary, discrete-valued, ordinal, continuous-valued). Given binary data LCS traditionally applies a ternary rule representation (i.e. rules can include either a 0, 1, or '#' for each feature in the data). The 'don't care' symbol (i.e. '#') serves as a wild card within a rule's condition allowing rules, and the system as a whole to generalize relationships between features and the target endpoint to be predicted. Consider the following rule (#1###0 ~ 1) (i.e. condition ~ action). This rule can be interpreted as: IF the second feature = 1 AND the sixth feature = 0 THEN the class prediction = 1. We would say that the second and sixth features were specified in this rule, while the others were generalized. This rule, and the corresponding prediction are only applicable to an instance when the condition of the rule is satisfied by the instance. This is more commonly referred to as matching. In Michigan-style LCS, each rule has its own fitness, as well as a number of other rule-parameters associated with it that can describe the number of copies of that rule that exist (i.e. the *numerosity*), the age of the rule, its accuracy, or the accuracy of its reward predictions, and other descriptive or experiential statistics. A rule along with its parameters is often referred to as a *classifier*. In Michigan-style systems, classifiers are contained within a *population* [P] that has a user defined maximum number of classifiers. Unlike most stochastic search algorithms (e.g. evolutionary algorithms), LCS populations start out empty (i.e. there is no need to randomly initialize a rule population). Classifiers will instead be initially introduced to the population with a covering mechanism.

In any LCS, the trained model is a set of rules/classifiers, rather than any single rule/classifier. In Michigan-style LCS, the entire trained (and optionally, compacted) classifier population forms the prediction model.

Matching

One of the most critical and often time-consuming elements of an LCS is the matching process. The first step in an LCS learning cycle takes a single training instance from the environment and passes it to [P] where matching takes place. In step two, every rule in [P] is now compared to the training instance to see which rules match (i.e. are contextually relevant to the current instance). In step three, any matching rules are moved to a *match set* [M]. A rule matches a training instance if all feature values specified in the rule condition are equivalent to the corresponding feature value in the training instance. For example, assuming the training instance is (001001 ~ 0), these rules would match: (###0## ~ 0), (00###1 ~ 0), (#01001 ~ 1), but these rules would not (1##### ~ 0), (000##1 ~ 0), (#0#1#0 ~ 1). Notice that in matching, the endpoint/action specified by the rule is not taken into consideration. As a result, the match set may contain classifiers that propose conflicting actions. In the fourth step, since we are performing supervised learning, [M] is divided into a correct set [C] and an incorrect set [I]. A matching rule goes into the correct set if it proposes the correct action (based on the known action of the training instance), otherwise it

goes into [I]. In reinforcement learning LCS, an action set [A] would be formed here instead, since the correct action is not known.

Covering

At this point in the learning cycle, if no classifiers made it into either [M] or [C] (as would be the case when the population starts off empty), the covering mechanism is applied (fifth step). Covering is a form of *online smart population initialization*. Covering randomly generates a rule that matches the current training instance (and in the case of supervised learning, that rule is also generated with the correct action. Assuming the training instance is (001001 ~ 0), covering might generate any of the following rules: (#0#0## ~ 0), (001001 ~ 0), (#010## ~ 0). Covering not only ensures that each learning cycle there is at least one correct, matching rule in [C], but that any rule initialized into the population will match at least one training instance. This prevents LCS from exploring the search space of rules that do not match any training instances.

Parameter updates/credit assignment/learning

In the sixth step, the rule parameters of any rule in [M] are updated to reflect the new experience gained from the current training instance. Depending on the LCS algorithm, a number of updates can take place at this step. For supervised learning, we can simply update the accuracy/error of a rule. Rule accuracy/error is different than model accuracy/error, since it is not calculated over the entire training data, but only over all instances that it matched. Rule accuracy is calculated by dividing the number of times the rule was in a correct set [C] by the number of times it was in a match set [M]. Rule accuracy can be thought of as a 'local accuracy'. Rule fitness is also updated here, and is commonly calculated as a function of rule accuracy. The concept of fitness is taken directly from classic genetic algorithms. Be aware that there are many variations on how LCS updates parameters in order to perform credit assignment and learning.

Subsumption

In the seventh step, a *subsumption* mechanism is typically applied. Subsumption is an explicit generalization mechanism that merges classifiers that cover redundant parts of the problem space. The subsuming classifier effectively absorbs the subsumed classifier (and has its numerosity increased). This can only happen when the subsuming classifier is more general, just as accurate, and covers all of the problem space of the classifier it subsumes.

Rule discovery/genetic algorithm

In the eighth step, LCS adopts a highly elitist genetic algorithm (GA) which will select two parent classifiers based on fitness (survival of the fittest). Parents are selected from [C] typically using tournament selection. Some systems have applied roulette wheel selection or deterministic

selection, and have differently selected parent rules from either [P] - panmictic selection, or from [M]). Crossover and mutation operators are now applied to generate two new offspring rules. At this point, both the parent and offspring rules are returned to [P]. The LCS genetic algorithm is highly elitist since each learning iteration, the vast majority of the population is preserved. Rule discovery may alternatively be performed by some other method, such as an estimation of distribution algorithm, but a GA is by far the most common approach. Evolutionary algorithms like the GA employ a stochastic search, which makes LCS a stochastic algorithm. LCS seeks to cleverly explore the search space, but does not perform an exhaustive search of rule combinations, and is not guaranteed to converge on an optimal solution.

Deletion

The last step in a generic LCS learning cycle is to maintain the maximum population size. The deletion mechanism will select classifiers for deletion (commonly using roulette wheel selection). The probability of a classifier being selected for deletion is inversely proportional to its fitness. When a classifier is selected for deletion, its numerosity parameter is reduced by one. When the numerosity of a classifier is reduced to zero, it is removed entirely from the population.

Training

LCS will cycle through these steps repeatedly for some user defined number of training iterations, or until some user defined termination criteria have been met. For online learning, LCS will obtain a completely new training instance each iteration from the environment. For offline learning, LCS will iterate through a finite training dataset. Once it reaches the last instance in the dataset, it will go back to the first instance and cycle through the dataset again.

Rule compaction

Once training is complete, the rule population will inevitably contain some poor, redundant and inexperienced rules. It is common to apply a *rule compaction*, or *condensation* heuristic as a post-processing step. This resulting compacted rule population is ready to be applied as a prediction model (e.g. make predictions on testing instances), and/or to be interpreted for knowledge discovery.

Prediction

Whether or not rule compaction has been applied, the output of an LCS algorithm is a population of classifiers which can be applied to making predictions on previously unseen instances. The prediction mechanism is not part of the supervised LCS learning cycle itself, however it would play an important role in a reinforcement learning LCS learning cycle. For now we consider how

the prediction mechanism can be applied for making predictions to test data. When making predictions, the LCS learning components are deactivated so that the population does not continue to learn from incoming testing data. A test instance is passed to [P] where a match set [M] is formed as usual. At this point the match set is differently passed to a prediction array. Rules in the match set can predict different actions, therefore a voting scheme is applied. In a simple voting scheme, the action with the strongest supporting 'votes' from matching rules wins, and becomes the selected prediction. All rules do not get an equal vote. Rather the strength of the vote for a single rule is commonly proportional to its numerosity and fitness. This voting scheme and the nature of how LCS's store knowledge, suggests that LCS algorithms are implicitly *ensemble learners*.

Interpretation

Individual LCS rules are typically human readable IF:THEN expression. Rules that constitute the LCS prediction model can be ranked by different rule parameters and manually inspected. Global strategies to guide knowledge discovery using statistical and graphical have also been proposed.[12][13] With respect to other advanced machine learning approaches, such as artificial neural networks, random forests, or genetic programming, learning classifier systems are particularly well suited to problems that require interpretable solutions.

Evolutionary Computation models

There are four main evolutionary computation models:

- Genetic Algorithms,
- Evolutionary Programming,
- Evolutionary Strategies,
- Genetic Programming.

These models differ basically in the way they encode the individuals and application of the genetic operators. These will be discussed in the following subsections.

Genetic Algorithms

A genetic algorithm(GA) has the following components:

- a mechanism to encode solutions to problems as strings,
- a population of solutions represented as strings,
- a problem dependent fitness function,
- a selection mechanism,
- crossover and mutation operators.

Generational Cycle

The generational cycle consists of repeated application of selection and the genetic operators to the population. Typically a population size of 30 to 200 is used. Crossover rates are chosen in the interval [0.5, 1.0] whereas mutation rates in the interval [0.001, 0.05]. These parameters are called the *control parameters* of the GA and are specified before the execution starts.

The generational cycle is terminated using a stopping criterion such as:

- reaching a fixed number of iterations,
- evolving a string with a high fitness value,
- creation of a certain degree of homogeneity within the population.

Applications of Evolutionary Algorithms

EAs should be used when there is no other known problem solving strategy, and the problem domain is NP-complete. That's where EAs come into play: heuristically finding solutions where all else fails.

Some of the EA applications are listed below:

- *Time-tabling:* This has been addressed quite successfully with GAs. A very common example of this kind of problem is the time-tabling of exams or classes in Universities, etc. At the Department of Artificial Intelligence, University of Edinburgh, time-tabling the MSc exams is now done using a GA
- *Job-shop scheduling*: The Job-Shop Scheduling Problem is a very difficult NP-complete problem which, so far, seems best addressed by branch and bound search techniques. GA researchers, however, are continuing to make progress on it.
- *Game playing*: GAs can be used to evolve behaviors for playing games. Work in evolutionary game theory typically surrounds the evolution of a population of players who meet randomly to play a game in which they each must adopt one of a limited number of moves
- *Computer Aided Design (CAD):* General Electric developed EnGENEous a CAD tool. It is a hybrid system, combining numerical optimization tools, expert systems and genetic algorithms
- *Face Recognition*: Faceprints was developed by New Mexico State University. It helps to draw a suspect's face from a witness's description. The GA generates 20 faces on a computer screen. The witness provides the fitness function by assigning scores to faces on the screen on a 10-point scale. Using this information together with the usual genetic operators additional faces are generated
- *Financial Time-Series Prediction*: Prediction Company, has developed a set of time-series prediction and trading tools for currency trading tools for currency trading in which GA constitute an important part