## Contents

### Articles

- Evolutionary computation 1
- Evolutionary algorithm 4
- Mathematical optimization 7
- Nonlinear programming 19
- Combinatorial optimization 21
- Travelling salesman problem 24
- Constraint (mathematics) 37
- Constraint satisfaction problem 38
- Constraint satisfaction 41
- Heuristic (computer science) 45
- Multi-objective optimization 45
- Pareto efficiency 50
- Stochastic programming 55
- Parallel metaheuristic 57
- There ain't no such thing as a free lunch 61
- Fitness landscape 63
- Genetic algorithm 65
- Toy block 77
- Chromosome (genetic algorithm) 79
- Genetic operator 79
- Crossover (genetic algorithm) 80
- Mutation (genetic algorithm) 83
- Inheritance (genetic algorithm) 84
- Selection (genetic algorithm) 84
- Tournament selection 85
- Truncation selection 86
- Fitness proportionate selection 86
- Reward-based selection 87
- Edge recombination operator 88
- Population-based incremental learning 91
- Defining length 93
- Holland's schema theorem 94
- Genetic memory (computer science) 95
- Premature convergence 95
Schema (genetic algorithms) 96
Fitness function 97
Black box 98
Black box theory 100
Fitness approximation 101
Effective fitness 103
Speciation (genetic algorithm) 103
Genetic representation 104
Stochastic universal sampling 105
Quality control and genetic algorithms 106
Human-based genetic algorithm 108
Interactive evolutionary computation 110
Genetic programming 112
Gene expression programming 119
Grammatical evolution 120
Grammar induction 122
Java Grammatical Evolution 124
Linear genetic programming 125
Evolutionary programming 126
Gaussian adaptation 127
Differential evolution 133
Particle swarm optimization 135
Ant colony optimization algorithms 141
Artificial bee colony algorithm 153
Evolution strategy 155
Evolution window 157
CMA-ES 157
Cultural algorithm 168
Learning classifier system 170
Memetic algorithm 172
Meta-optimization 177
Cellular evolutionary algorithm 179
Cellular automaton 182
Artificial immune system 194
Evolutionary multi-modal optimization 198
Evolutionary music 201
Coevolution 203
Evolutionary art 208
Evolutionary computation

In computer science, **evolutionary computation** is a subfield of artificial intelligence (more particularly computational intelligence) that involves combinatorial optimization problems.

Evolutionary computation uses iterative progress, such as growth or development in a population. This population is then selected in a guided random search using parallel processing to achieve the desired end. Such processes are often inspired by biological mechanisms of evolution.

As evolution can produce highly optimised processes and networks, it has many applications in computer science.

**History**

The use of Darwinian principles for automated problem solving originated in the fifties. It was not until the sixties that three distinct interpretations of this idea started to be developed in three different places.

Evolutionary programming was introduced by Lawrence J. Fogel in the US, while John Henry Holland called his method a genetic algorithm. In Germany Ingo Rechenberg and Hans-Paul Schwefel introduced evolution strategies. These areas developed separately for about 15 years. From the early nineties on they are unified as different representatives ("dialects") of one technology, called evolutionary computing. Also in the early nineties, a fourth stream following the general ideas had emerged – genetic programming. Since the 1990s, evolutionary computation has largely become swarm-based computation, and nature-inspired algorithms are becoming an increasingly significant part.

These terminologies denote the field of evolutionary computing and consider evolutionary programming, evolution strategies, genetic algorithms, and genetic programming as sub-areas.

Simulations of evolution using evolutionary algorithms and artificial life started with the work of Nils Aall Barricelli in the 1960s, and was extended by Alex Fraser, who published a series of papers on simulation of artificial selection.[1] Artificial evolution became a widely recognised optimisation method as a result of the work of Ingo Rechenberg in the 1960s and early 1970s, who used evolution strategies to solve complex engineering problems.[2] Genetic algorithms in particular became popular through the writing of John Holland.[3] As academic interest grew, dramatic increases in the power of computers allowed practical applications, including the automatic evolution of computer programs.[4] Evolutionary algorithms are now used to solve multi-dimensional problems more efficiently than software produced by human designers, and also to optimise the design of systems.[5]

**Techniques**

Evolutionary computing techniques mostly involve metaheuristic optimization algorithms. Broadly speaking, the field includes:

**Evolutionary algorithms**
- Genetic algorithm
- Genetic programming
- Evolutionary programming
- Evolution strategy
- Differential evolution
- Eagle strategy

**Swarm intelligence**
- Ant colony optimization
- Particle swarm optimization
- Bees algorithm
• Cuckoo search
and in a lesser extent also:
• Artificial life (also see digital organism)
• Artificial immune systems
• Cultural algorithms
• Firefly algorithm
• Harmony search
• Learning classifier systems
• Learnable Evolution Model
• Parallel simulated annealing
• Self-organization such as self-organizing maps, competitive learning
• Self-Organizing Migrating Genetic Algorithm
• Swarm-based computing

Evolutionary algorithms
Evolutionary algorithms form a subset of evolutionary computation in that they generally only involve techniques implementing mechanisms inspired by biological evolution such as reproduction, mutation, recombination, natural selection and survival of the fittest. Candidate solutions to the optimization problem play the role of individuals in a population, and the cost function determines the environment within which the solutions "live" (see also fitness function). Evolution of the population then takes place after the repeated application of the above operators.

In this process, there are two main forces that form the basis of evolutionary systems: **Recombination** and **mutation** create the necessary diversity and thereby facilitate novelty, while **selection** acts as a force increasing quality.

Many aspects of such an evolutionary process are stochastic. Changed pieces of information due to recombination and mutation are randomly chosen. On the other hand, selection operators can be either deterministic, or stochastic. In the latter case, individuals with a higher fitness have a higher chance to be selected than individuals with a lower fitness, but typically even the weak individuals have a chance to become a parent or to survive.

Evolutionary computation practitioners
**Incomplete list:**
• Kalyanmoy Deb
• David E. Goldberg
• John Henry Holland
• John Koza
• Peter Nordin
• Ingo Rechenberg
• Hans-Paul Schwefel
• Peter J. Fleming
• Carlos M. Fonseca [6]
• Lee Graham
Major conferences and workshops

- IEEE Congress on Evolutionary Computation (CEC)
- Genetic and Evolutionary Computation Conference (GECCO)\textsuperscript{[7]}
- International Conference on Parallel Problem Solving From Nature (PPSN)\textsuperscript{[8]}

Bibliography


References


- Evolutionary Computing Research Community Europe (http://www.evolutionary-computing.eu)
- Evolutionary Computation Repository (http://www.fmi.uni-stuttgart.de/flk/evolalg/)
- Interactive illustration of Evolutionary Computation (http://userweb.eng.gla.ac.uk/yun.li/ga_demo/)
- VitaSCIENCES (http://www.vita-sciences.org/)
Evolutionary algorithm

In artificial intelligence, an evolutionary algorithm (EA) is a subset of evolutionary computation, a generic population-based metaheuristic optimization algorithm. An EA uses some mechanisms inspired by biological evolution: 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 environment within which the solutions "live" (see also cost function). Evolution of the population then takes place after the repeated application of the above operators. Artificial evolution (AE) describes a process involving individual evolutionary algorithms; EAs are individual components that participate in an AE.

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; this generality is shown by successes in fields as diverse as engineering, art, biology, economics, marketing, genetics, operations research, robotics, social sciences, physics, politics and chemistry.

Techniques from evolutionary algorithms applied to the modeling of biological evolution are generally limited to explorations of microevolutionary processes, however some computer simulations, such as Tierra and Avida, attempt to model macroevolutionary dynamics.

In most real applications of EAs, computational complexity is a prohibiting factor. 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; therefore, there may be no direct link between algorithm complexity and problem complexity.

Another possible limitation of many evolutionary algorithms is their lack of a clear genotype-phenotype distinction. In nature, the fertilized egg cell undergoes a complex process known as embryogenesis to become a mature phenotype. This indirect encoding is believed to make the genetic search more robust (i.e. reduce the probability of fatal mutations), and also may improve the evolvability of the organism.\(^1\)\(^2\) Such indirect (aka generative or developmental) encodings also enable evolution to exploit the regularity in the environment.\(^3\) Recent work in the field of artificial embryogeny, or artificial developmental systems, seeks to address these concerns. And gene expression programming successfully explores a genotype-phenotype system, where the genotype consists of linear multigenic chromosomes of fixed length and the phenotype consists of multiple expression trees or computer programs of different sizes and shapes.\(^4\)

Implementation of biological processes

Usually, an initial population of randomly generated candidate solutions comprise the first generation. The fitness function is applied to the candidate solutions and any subsequent offspring.

In selection, parents for the next generation are chosen with a bias towards higher fitness. The parents reproduce one or two offsprings (new candidates) by copying their genes, with two possible changes: crossover recombines the parental genes and mutation alters the genotype of an individual in a random way. These new candidates compete with old candidates for their place in the next generation (survival of the fittest).

This process can be repeated until a candidate with sufficient quality (a solution) is found or a previously defined computational limit is reached.
Evolutionary algorithm techniques

Similar techniques differ in the 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), 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

Related techniques

Swarm algorithms, including:

- Ant colony optimization - Based on the ideas of ant foraging by pheromone communication to form paths. Primarily suited for combinatorial optimization and graph problems.
- Bees algorithm is based on the foraging behaviour of honey bees. It has been applied in many applications such as routing and scheduling.
- Cuckoo search is inspired by the brooding parasitism of some cuckoo species. It also uses Lévy flights, and thus it suits for global optimization problems.
- Particle swarm optimization - Based on the ideas of animal flocking behaviour. Also primarily suited for numerical optimization problems.

Other population-based metaheuristic methods:

- Firefly algorithm is inspired by the behavior of fireflies, attracting each other by flashing light. This is especially useful for multimodal optimization.
- Invasive weed optimization algorithm - Based on the ideas of weed colony behavior in searching and finding a suitable place for growth and reproduction.
- Harmony search - Based on the ideas of musicians' behavior in searching for better harmonies. This algorithm is suitable for combinatorial optimization as well as parameter optimization.
- Gaussian adaptation - Based on information theory. Used for maximization of manufacturing yield, mean fitness or average information. See for instance Entropy in thermodynamics and information theory.
References


Bibliography


External links

• Evolutionary Computation Repository (http://www.fmi.uni-stuttgart.de/fk/evolalgonline/)
• Genetic Algorithms and Evolutionary Computation (http://www.talkorigins.org/faqs/genalg/genalg.html)
• An online interactive Evolutionary Algorithm demonstrator to practise or learn how exactly an EA works. (http://userweb.elec.gla.ac.uk/yunli/ga_demo/) Learn step by step or watch global convergence in batch, change population size, crossover rate, mutation rate and selection mechanism, and add constraints.
Mathematical optimization

In mathematics, computational science, or management science, \textbf{mathematical optimization} (alternatively, \textbf{optimization} or \textbf{mathematical programming}) refers to the selection of a best element from some set of available alternatives.\footnote{1}

In the simplest case, an optimization problem consists of maximizing or minimizing a real function by systematically choosing input values from within an allowed set and computing the value of the function. The generalization of optimization theory and techniques to other formulations comprises a large area of applied mathematics. More generally, optimization includes finding "best available" values of some objective function given a defined domain, including a variety of different types of objective functions and different types of domains.

\section*{Optimization problems}

An optimization problem can be represented in the following way

\begin{itemize}
  \item \textit{Given:} a function \( f : A \rightarrow \mathbb{R} \) from some set \( A \) to the real numbers
  \item \textit{Sought:} an element \( x_0 \) in \( A \) such that \( f(x_0) \leq f(x) \) for all \( x \) in \( A \) ("minimization") or such that \( f(x_0) \geq f(x) \) for all \( x \) in \( A \) ("maximization").
\end{itemize}

Such a formulation is called an \textbf{optimization problem} or a \textbf{mathematical programming problem} (a term not directly related to computer programming, but still in use for example in linear programming - see History below). Many real-world and theoretical problems may be modeled in this general framework. Problems formulated using this technique in the fields of physics and computer vision may refer to the technique as \textit{energy minimization}, speaking of the value of the function \( f \) as representing the energy of the system being modeled.

Typically, \( A \) is some subset of the Euclidean space \( \mathbb{R}^n \), often specified by a set of \textit{constraints}, equalities or inequalities that the members of \( A \) have to satisfy. The domain \( A \) of \( f \) is called the \textit{search space} or the \textit{choice set}, while the elements of \( A \) are called \textit{candidate solutions} or \textit{feasible solutions}.

The function \( f \) is called, variously, an \textbf{objective function}, \textbf{cost function} (minimization), \textbf{utility function} (maximization), or, in certain fields, \textbf{energy function}, or \textbf{energy functional}. A feasible solution that minimizes (or maximizes, if that is the goal) the objective function is called an \textit{optimal solution}.

By convention, the standard form of an optimization problem is stated in terms of minimization. Generally, unless both the objective function and the feasible region are convex in a minimization problem, there may be several local minima, where a \textit{local minimum} \( x^* \) is defined as a point for which there exists some \( \delta > 0 \) so that for all \( x \) such that

\[ \| x - x^* \| \leq \delta ; \]

the expression

\[ f(x^*) \leq f(x) \]

holds; that is to say, on some region around \( x^* \) all of the function values are greater than or equal to the value at that point. Local maxima are defined similarly.

A large number of algorithms proposed for solving non-convex problems – including the majority of commercially available solvers – are not capable of making a distinction between local optimal solutions and rigorous optimal solutions, and will treat the former as actual solutions to the original problem. The branch of applied mathematics and numerical analysis that is concerned with the development of deterministic algorithms that are capable of
guaranteeing convergence in finite time to the actual optimal solution of a non-convex problem is called global optimization.

**Notation**

Optimization problems are often expressed with special notation. Here are some examples.

### Minimum and maximum value of a function

Consider the following notation:

\[ \min_{x \in \mathbb{R}} (x^2 + 1) \]

This denotes the minimum value of the objective function \( x^2 + 1 \), when choosing \( x \) from the set of real numbers \( \mathbb{R} \). The minimum value in this case is 1, occurring at \( x = 0 \).

Similarly, the notation

\[ \max_{x \in \mathbb{R}} 2x \]

asks for the maximum value of the objective function \( 2x \), where \( x \) may be any real number. In this case, there is no such maximum as the objective function is unbounded, so the answer is "infinity" or "undefined".

### Optimal input arguments

Consider the following notation:

\[ \arg \min_{x \in (-\infty, -1]} x^2 + 1, \]

or equivalently

\[ \arg \min_{x} x^2 + 1, \quad \text{subject to: } x \in (-\infty, -1]. \]

This represents the value (or values) of the argument \( x \) in the interval \( (-\infty, -1] \) that minimizes (or minimize) the objective function \( x^2 + 1 \) (the actual minimum value of that function is not what the problem asks for). In this case, the answer is \( x = -1 \), since \( x = 0 \) is infeasible, i.e. does not belong to the feasible set.

Similarly,

\[ \arg \max_{x \in [-5, 5], \ y \in \mathbb{R}} x \cos(y), \]

or equivalently

\[ \arg \max_{x, \ y} x \cos(y), \quad \text{subject to: } x \in [-5, 5], \ y \in \mathbb{R}, \]

represents the \( (x, y) \) pair (or pairs) that maximizes (or maximize) the value of the objective function \( x \cos(y) \), with the added constraint that \( x \) lie in the interval \( [-5, 5] \) (again, the actual maximum value of the expression does not matter). In this case, the solutions are the pairs of the form \((5, 2k\pi)\) and \((-5,(2k+1)\pi)\), where \( k \) ranges over all integers.

\textbf{Arg min} and \textbf{arg max} are sometimes also written \textbf{argmin} and \textbf{argmax}, and stand for \textit{argument of the minimum} and \textit{argument of the maximum}. 
Mathematical optimization

History

Fermat and Lagrange found calculus-based formulas for identifying optima, while Newton and Gauss proposed iterative methods for moving towards an optimum. Historically, the first term for optimization was "linear programming", which was due to George B. Dantzig, although much of the theory had been introduced by Leonid Kantorovich in 1939. Dantzig published the Simplex algorithm in 1947, and John von Neumann developed the theory of duality in the same year.

The term *programming* in this context does not refer to computer programming. Rather, the term comes from the use of *program* by the United States military to refer to proposed training and logistics schedules, which were the problems Dantzig studied at that time.

Later important researchers in mathematical optimization include the following:

- Richard Bellman
- Ronald A. Howard
- Narendra Karmarkar
- William Karush
- Leonid Khachiyan
- Bernard Koopman
- Harold Kuhn
- Joseph Louis Lagrange
- László Lovász
- Arkadii Nemirovskii
- Yuri Nesterov
- Boris Polyak
- Lev Pontryagin
- James Renegar
- R. Tyrrell Rockafellar
- Cornelis Roos
- Naum Z. Shor
- Michael J. Todd
- Albert Tucker

Major subfields

- Convex programming studies the case when the objective function is convex (minimization) or concave (maximization) and the constraint set is convex. This can be viewed as a particular case of nonlinear programming or as generalization of linear or convex quadratic programming.
  - Linear programming (LP), a type of convex programming, studies the case in which the objective function $f$ is linear and the set of constraints is specified using only linear equalities and inequalities. Such a set is called a polyhedron or a polytope if it is bounded.
  - Second order cone programming (SOCP) is a convex program, and includes certain types of quadratic programs.
  - Semidefinite programming (SDP) is a subfield of convex optimization where the underlying variables are semidefinite matrices. It is generalization of linear and convex quadratic programming.
  - Conic programming is a general form of convex programming. LP, SOCP and SDP can all be viewed as conic programs with the appropriate type of cone.
  - Geometric programming is a technique whereby objective and inequality constraints expressed as posynomials and equality constraints as monomials can be transformed into a convex program.

- Integer programming studies linear programs in which some or all variables are constrained to take on integer values. This is not convex, and in general much more difficult than regular linear programming.
- Quadratic programming allows the objective function to have quadratic terms, while the feasible set must be specified with linear equalities and inequalities. For specific forms of the quadratic term, this is a type of convex programming.
- Fractional programming studies optimization of ratios of two nonlinear functions. The special class of concave fractional programs can be transformed to a convex optimization problem.
- Nonlinear programming studies the general case in which the objective function or the constraints or both contain nonlinear parts. This may or may not be a convex program. In general, whether the program is convex affects the...
difficulty of solving it.

- Stochastic programming studies the case in which some of the constraints or parameters depend on random variables.

- Robust programming is, like stochastic programming, an attempt to capture uncertainty in the data underlying the optimization problem. This is not done through the use of random variables, but instead, the problem is solved taking into account inaccuracies in the input data.

- Combinatorial optimization is concerned with problems where the set of feasible solutions is discrete or can be reduced to a discrete one.

- Infinite-dimensional optimization studies the case when the set of feasible solutions is a subset of an infinite-dimensional space, such as a space of functions.

- Heuristics and metaheuristics make few or no assumptions about the problem being optimized. Usually, heuristics do not guarantee that any optimal solution need be found. On the other hand, heuristics are used to find approximate solutions for many complicated optimization problems.

- Constraint satisfaction studies the case in which the objective function $f$ is constant (this is used in artificial intelligence, particularly in automated reasoning).
  - Constraint programming.
  - Disjunctive programming is used where at least one constraint must be satisfied but not all. It is of particular use in scheduling.

In a number of subfields, the techniques are designed primarily for optimization in dynamic contexts (that is, decision making over time):

- Calculus of variations seeks to optimize an objective defined over many points in time, by considering how the objective function changes if there is a small change in the choice path.

- Optimal control theory is a generalization of the calculus of variations.

- Dynamic programming studies the case in which the optimization strategy is based on splitting the problem into smaller subproblems. The equation that describes the relationship between these subproblems is called the Bellman equation.

- Mathematical programming with equilibrium constraints is where the constraints include variational inequalities or complementarities.

### Multi-objective optimization

Adding more than one objective to an optimization problem adds complexity. For example, to optimize a structural design, one would want a design that is both light and rigid. Because these two objectives conflict, a trade-off exists. There will be one lightest design, one stiffer design, and an infinite number of designs that are some compromise of weight and stiffness. The set of trade-off designs that cannot be improved upon according to one criterion without hurting another criterion is known as the Pareto set. The curve created plotting weight against stiffness of the best designs is known as the Pareto frontier.

A design is judged to be "Pareto optimal" (equivalently, "Pareto efficient" or in the Pareto set) if it is not dominated by any other design: If it is worse than another design in some respects and no better in any respect, then it is dominated and is not Pareto optimal.

The choice among "Pareto optimal" solutions to determine the "favorite solution" is delegated to the decision maker. In other words, defining the problem as multiobjective optimization signals that some information is missing: desirable objectives are given but not their detailed combination. In some cases, the missing information can be derived by interactive sessions with the decision maker.
Multi-modal optimization

Optimization problems are often multi-modal; that is they possess multiple good solutions. They could all be globally good (same cost function value) or there could be a mix of globally good and locally good solutions. Obtaining all (or at least some of) the multiple solutions is the goal of a multi-modal optimizer.

Classical optimization techniques due to their iterative approach do not perform satisfactorily when they are used to obtain multiple solutions, since it is not guaranteed that different solutions will be obtained even with different starting points in multiple runs of the algorithm. Evolutionary Algorithms are however a very popular approach to obtain multiple solutions in a multi-modal optimization task. See Evolutionary multi-modal optimization.

Classification of critical points and extrema

Feasibility problem

The satisfiability problem, also called the feasibility problem, is just the problem of finding any feasible solution at all without regard to objective value. This can be regarded as the special case of mathematical optimization where the objective value is the same for every solution, and thus any solution is optimal.

Many optimization algorithms need to start from a feasible point. One way to obtain such a point is to relax the feasibility conditions using a slack variable; with enough slack, any starting point is feasible. Then, minimize that slack variable until slack is null or negative.

Existence

The extreme value theorem of Karl Weierstrass states that a continuous real-valued function on a compact set attains its maximum and minimum value. More generally, a lower semi-continuous function on a compact set attains its minimum; an upper semi-continuous function on a compact set attains its maximum.

Necessary conditions for optimality

One of Fermat's theorems states that optima of unconstrained problems are found at stationary points, where the first derivative or the gradient of the objective function is zero (see first derivative test). More generally, they may be found at critical points, where the first derivative or gradient of the objective function is zero or is undefined, or on the boundary of the choice set. An equation (or set of equations) stating that the first derivative(s) equal(s) zero at an interior optimum is called a 'first-order condition' or a set of first-order conditions.

Optima of inequality-constrained problems are instead found by the Lagrange multiplier method. This method calculates a system of inequalities called the 'Karush–Kuhn–Tucker conditions' or 'complementary slackness conditions', which may then be used to calculate the optimum.

Sufficient conditions for optimality

While the first derivative test identifies points that might be extrema, this test does not distinguish a point that is a minimum from one that is a maximum or one that is neither. When the objective function is twice differentiable, these cases can be distinguished by checking the second derivative or the matrix of second derivatives (called the Hessian matrix) in unconstrained problems, or the matrix of second derivatives of the objective function and the constraints called the bordered Hessian in constrained problems. The conditions that distinguish maxima, or minima, from other stationary points are called 'second-order conditions' (see 'Second derivative test'). If a candidate solution satisfies the first-order conditions, then satisfaction of the second-order conditions as well is sufficient to establish at least local optimality.
Sensitivity and continuity of optima

The envelope theorem describes how the value of an optimal solution changes when an underlying parameter changes. The process of computing this change is called comparative statics.

The maximum theorem of Claude Berge (1963) describes the continuity of an optimal solution as a function of underlying parameters.

Calculus of optimization

For unconstrained problems with twice-differentiable functions, some critical points can be found by finding the points where the gradient of the objective function is zero (that is, the stationary points). More generally, a zero subgradient certifies that a local minimum has been found for minimization problems with convex functions and other locally Lipschitz functions.

Further, critical points can be classified using the definiteness of the Hessian matrix: If the Hessian is positive definite at a critical point, then the point is a local minimum; if the Hessian matrix is negative definite, then the point is a local maximum; finally, if indefinite, then the point is some kind of saddle point.

Constrained problems can often be transformed into unconstrained problems with the help of Lagrange multipliers. Lagrangian relaxation can also provide approximate solutions to difficult constrained problems.

When the objective function is convex, then any local minimum will also be a global minimum. There exist efficient numerical techniques for minimizing convex functions, such as interior-point methods.

Computational optimization techniques

To solve problems, researchers may use algorithms that terminate in a finite number of steps, or iterative methods that converge to a solution (on some specified class of problems), or heuristics that may provide approximate solutions to some problems (although their iterates need not converge).

Optimization algorithms

• Simplex algorithm of George Dantzig, designed for linear programming.
• Extensions of the simplex algorithm, designed for quadratic programming and for linear-fractional programming.
• Variants of the simplex algorithm that are especially suited for network optimization.
• Combinatorial algorithms

Iterative methods

The iterative methods used to solve problems of nonlinear programming differ according to whether they evaluate Hessians, gradients, or only function values. While evaluating Hessians (H) and gradients (G) improves the rate of convergence, such evaluations increase the computational complexity (or computational cost) of each iteration. In some cases, the computational complexity may be excessively high.

One major criterion for optimizers is just the number of required function evaluations as this often is already a large computational effort, usually much more effort than within the optimizer itself, which mainly has to operate over the N variables. The derivatives provide detailed information for such optimizers, but are even harder to calculate, e.g. approximating the gradient takes at least N+1 function evaluations. For approximations of the 2nd derivatives (collected in the Hessian matrix) the number of function evaluations is in the order of N^2. Newton's method requires the 2nd order derivates, so for each iteration the number of function calls is in the order of N^2, but for a simpler pure gradient optimizer it is only N. However, gradient optimizers need usually more iterations than Newton's algorithm. Which one is best wrt. number of function calls depends on the problem itself.

• Methods that evaluate Hessians (or approximate Hessians, using finite differences):
• Newton's method
  • Sequential quadratic programming: A Newton-based method for small-medium scale constrained problems. Some versions can handle large-dimensional problems.

• Methods that evaluate gradients or approximate gradients using finite differences (or even subgradients):
  • Quasi-Newton methods: Iterative methods for medium-large problems (e.g. N<1000).
  • Conjugate gradient methods: Iterative methods for large problems. (In theory, these methods terminate in a finite number of steps with quadratic objective functions, but this finite termination is not observed in practice on finite-precision computers.)
  • Interior point methods: This is a large class of methods for constrained optimization. Some interior-point methods use only (sub)gradient information, and others of which require the evaluation of Hessians.
  • Gradient descent (alternatively, "steepest descent" or "steepest ascent"): A (slow) method of historical and theoretical interest, which has had renewed interest for finding approximate solutions of enormous problems.
  • Subgradient methods - An iterative method for large locally Lipschitz functions using generalized gradients. Following Boris T. Polyak, subgradient–projection methods are similar to conjugate–gradient methods.
  • Bundle method of descent: An iterative method for small–medium sized problems with locally Lipschitz functions, particularly for convex minimization problems. (Similar to conjugate gradient methods)
  • Ellipsoid method: An iterative method for small problems with quasiconvex objective functions and of great theoretical interest, particularly in establishing the polynomial time complexity of some combinatorial optimization problems. It has similarities with Quasi-Newton methods.
  • Reduced gradient method (Frank–Wolfe) for approximate minimization of specially structured problems with linear constraints, especially with traffic networks. For general unconstrained problems, this method reduces to the gradient method, which is regarded as obsolete (for almost all problems).
  • Methods that evaluate only function values: If a problem is continuously differentiable, then gradients can be approximated using finite differences, in which case a gradient-based method can be used.
  • Interpolation methods
  • Pattern search methods, which have better convergence properties than the Nelder–Mead heuristic (with simplices), which is listed below.

Global convergence
More generally, if the objective function is not a quadratic function, then many optimization methods use other methods to ensure that some subsequence of iterations converges to an optimal solution. The first and still popular method for ensuring convergence relies on line searches, which optimize a function along one dimension. A second and increasingly popular method for ensuring convergence uses trust regions. Both line searches and trust regions are used in modern methods of non-differentiable optimization. Usually a global optimizer is much slower than advanced local optimizers (such as BFGS), so often an efficient global optimizer can be constructed by starting the local optimizer from different starting points.
Heuristics

Besides (finitely terminating) algorithms and (convergent) iterative methods, there are heuristics that can provide approximate solutions to some optimization problems:

- Memetic algorithm
- Differential evolution
- Dynamic relaxation
- Genetic algorithms
- Hill climbing
- Nelder-Mead simplicial heuristic: A popular heuristic for approximate minimization (without calling gradients)
- Particle swarm optimization
- Simulated annealing
- Tabu search
- Reactive Search Optimization (RSO) \(^\text{[2]}\) implemented in LIONsolver

Applications

Mechanics and engineering

Problems in rigid body dynamics (in particular articulated rigid body dynamics) often require mathematical programming techniques, since you can view rigid body dynamics as attempting to solve an ordinary differential equation on a constraint manifold; the constraints are various nonlinear geometric constraints such as "these two points must always coincide", "this surface must not penetrate any other", or "this point must always lie somewhere on this curve". Also, the problem of computing contact forces can be done by solving a linear complementarity problem, which can also be viewed as a QP (quadratic programming) problem.

Many design problems can also be expressed as optimization programs. This application is called design optimization. One subset is the engineering optimization, and another recent and growing subset of this field is multidisciplinary design optimization, which, while useful in many problems, has in particular been applied to aerospace engineering problems.

Economics

Economics is closely enough linked to optimization of agents that an influential definition relatedly describes economics \textit{qua} science as the "study of human behavior as a relationship between ends and scarce means" with alternative uses. \(^\text{[3]}\) Modern optimization theory includes traditional optimization theory but also overlaps with game theory and the study of economic equilibria. The \textit{Journal of Economic Literature} codes classify mathematical programming, optimization techniques, and related topics under JEL:C61-C63.

In microeconomics, the utility maximization problem and its dual problem, the expenditure minimization problem, are economic optimization problems. Insofar as they behave consistently, consumers are assumed to maximize their utility, while firms are usually assumed to maximize their profit. Also, agents are often modeled as being risk-averse, thereby preferring to avoid risk. Asset prices are also modeled using optimization theory, though the underlying mathematics relies on optimizing stochastic processes rather than on static optimization. Trade theory also uses optimization to explain trade patterns between nations. The optimization of market portfolios is an example of multi-objective optimization in economics.

Since the 1970s, economists have modeled dynamic decisions over time using control theory. For example, microeconomists use dynamic search models to study labor-market behavior. \(^\text{[4]}\) A crucial distinction is between deterministic and stochastic models. \(^\text{[5]}\) Macroeconomists build dynamic stochastic general equilibrium (DSGE) models that describe the dynamics of the whole economy as the result of the interdependent optimizing decisions of
workers, consumers, investors, and governments. [6][7]

**Operations research**

Another field that uses optimization techniques extensively is operations research. Operations research also uses stochastic modeling and simulation to support improved decision-making. Increasingly, operations research uses stochastic programming to model dynamic decisions that adapt to events; such problems can be solved with large-scale optimization and stochastic optimization methods.

**Control engineering**

Mathematical optimization is used in much modern controller design. High-level controllers such as Model predictive control (MPC) or Real-Time Optimization (RTO) employ mathematical optimization. These algorithms run online and repeatedly determine values for decision variables, such as choke openings in a process plant, by iteratively solving a mathematical optimization problem including constraints and a model of the system to be controlled.

**Notes**


- "numerical optimization methods in economics (http://www.dictionaryofeconomics.com/article?id=pde2008_N000148&edition=current&q=optimization&topicid=&result_number=1)" by Karl Schmedders
- "convex programming (http://www.dictionaryofeconomics.com/article?id=pde2008_C000348&edition=current&q=optimization&topicid=&result_number=4)" by Lawrence E. Blume

**Further reading**

**Comprehensive**

**Undergraduate level**

Graduate level


   • J. E. Dennis, Jr. and Robert B. Schnabel, A view of unconstrained optimization (pp. 1–72);
   • Donald Goldfarb and Michael J. Todd, Linear programming (pp. 73–170);
   • Philip E. Gill, Walter Murray, Michael A. Saunders, and Margaret H. Wright, Constrained nonlinear programming (pp. 171–210);
   • Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin, Network flows (pp. 211–369);
   • W. R. Pulleyblank, Polyhedral combinatorics (pp. 371–446);
   • George L. Nemhauser and Laurence A. Wolsey, Integer programming (pp. 447–527);
   • Claude Lemaréchal, Nondifferentiable optimization (pp. 529–572);
   • Roger J-B Wets, Stochastic programming (pp. 573–629);
   • A. H. G. Rinnooy Kan and G. T. Timmer, Global optimization (pp. 631–662);
   • P. L. Yu, Multiple criteria decision making: five basic concepts (pp. 663–699).


Continuous optimization


Combinatorial optimization


Journals


External links

- COIN-OR (http://www.coin-or.org/)—Computational Infrastructure for Operations Research
- Decision Tree for Optimization Software (http://plato.asu.edu/guide.html) Links to optimization source codes
- Global optimization (http://www.mat.univie.ac.at/~neum/glopt.html)
- Mathematical Programming Glossary (http://glossary.computing.society.informs.org/)
- Mathematical Programming Society (http://www.mathprog.org/)
- Optimization Online (http://www.optimization-online.org) A repository for optimization e-prints
- Optimization Related Links (http://www2.arnes.si/~ljc3m2/igor/links.html)
- Convex Optimization I (http://see.stanford.edu/see/courseinfo.aspx?coll=2db7ced4-39d1-4fdd-90e8-364129597c87) EE364a: Course from Stanford University
• Simplemax Online Optimization Services (http://simplemax.net) Web applications to access nonlinear optimization services

Solvers:
• APOPT (http://wiki.mcs.anl.gov/NEOS/index.php/APOPT) - large-scale nonlinear programming
• Free Optimization Software by Systems Optimization Laboratory, Stanford University (http://www.stanford.edu/group/SOL/software.html)
• MIDACO-Solver (http://www.midaco-solver.com/) General purpose (MINLP) optimization software based on Ant colony optimization algorithms (Matlab, Excel, C/C++, Fortran)
• Moocho (http://trilinos.sandia.gov/packages/moocho/) - a very flexible open-source NLP solver
• TANGO Project (http://www.ime.usp.br/~egbirgin/tango/) - Trustable Algorithms for Nonlinear General Optimization - Fortran

Libraries:
• The NAG Library (http://www.nag.co.uk/numeric/numerical_libraries.asp) is a collection of numerical routines developed by the Numerical Algorithms Group for multiple programming languages (including C, C++, Fortran, Visual Basic, Java and C#) and packages (for example, MATLAB, Excel, R, and LabVIEW) which contains several routines for both local and global optimization.
• ALGLIB (http://www.alglib.net/optimization/) Open-source optimization routines (unconstrained and bound-constrained optimization), C++, C#, Delphi, Visual Basic.
• IOptLib (Investigative Optimization Library) (http://www2.arnes.si/~ljc3m2/igor/ioptlib/) - a free, open-source library for optimization algorithms (ANSI C).
• OAT (Optimization Algorithm Toolkit) (http://optalgtoolkit.sourceforge.net/) - a set of standard optimization algorithms and problems in Java.
• Java Parallel Optimization Package (JPOP) (http://www5.informatik.uni-erlangen.de/research/software/java-parallel-optimization-package/) An open-source java package which allows the parallel evaluation of functions, gradients, and hessians.
• OOL (Open Optimization library) (http://ool.sourceforge.net/)-optimization routines in C.
• FuncLib (http://funclib.codeplex.com/) Open source non-linear optimization library in C# with support for non-linear constraints and automatic differentiation.
• JOptimizer (http://www.joptimizer.com/) Open source Java library for convex optimization.
Nonlinear programming

In mathematics, **nonlinear programming (NLP)** is the process of solving a system of equalities and inequalities, collectively termed constraints, over a set of unknown real variables, along with an objective function to be maximized or minimized, where some of the constraints or the objective function are nonlinear.[1]

**Applicability**

A typical nonconvex problem is that of optimising transportation costs by selection from a set of transportion methods, one or more of which exhibit economies of scale, with various connectivities and capacity constraints. An example would be petroleum product transport given a selection or combination of pipeline, rail tanker, road tanker, river barge, or coastal tankship. Owing to economic batch size the cost functions may have discontinuities in addition to smooth changes.

**Mathematical formulation of the problem**

The problem can be stated simply as:

$$\max_{x \in X} f(x)$$ to maximize some variable such as product throughput

or

$$\min_{x \in X} f(x)$$ to minimize a cost function

where

$$f : \mathbb{R}^n \to \mathbb{R}$$

$$X \subseteq \mathbb{R}^n.$$

**Methods for solving the problem**

If the objective function $f$ is linear and the constrained space is a polytope, the problem is a linear programming problem, which may be solved using well known linear programming solutions.

If the objective function is concave (maximization problem), or convex (minimization problem) and the constraint set is convex, then the program is called convex and general methods from convex optimization can be used in most cases.

If the objective function is a ratio of a concave and a convex function (in the maximization case) and the constraints are convex, then the problem can be transformed to a convex optimization problem using fractional programming techniques.

Several methods are available for solving nonconvex problems. One approach is to use special formulations of linear programming problems. Another method involves the use of branch and bound techniques, where the program is divided into subclasses to be solved with convex (minimization problem) or linear approximations that form a lower bound on the overall cost within the subdivision. With subsequent divisions, at some point an actual solution will be obtained whose cost is equal to the best lower bound obtained for any of the approximate solutions. This solution is optimal, although possibly not unique. The algorithm may also be stopped early, with the assurance that the best possible solution is within a tolerance from the best point found; such points are called $\varepsilon$-optimal. Terminating to $\varepsilon$-optimal points is typically necessary to ensure finite termination. This is especially useful for large, difficult problems and problems with uncertain costs or values where the uncertainty can be estimated with an appropriate reliability estimation.

Under differentiability and constraint qualifications, the Karush–Kuhn–Tucker (KKT) conditions provide necessary conditions for a solution to be optimal. Under convexity, these conditions are also sufficient.
Examples

2-dimensional example

A simple problem can be defined by the constraints
\[
x_1 \geq 0 \\
x_2 \geq 0 \\
x_1^2 + x_2^2 \geq 1 \\
x_1^2 + x_2^2 \leq 2
\]
with an objective function to be maximized
\[
f(x) = x_1 + x_2
\]
where \(x = (x_1, x_2)\). Solve 2-D Problem \([2]\).

3-dimensional example

Another simple problem can be defined by the constraints
\[
x_1^2 - x_2^2 + x_3^2 \leq 2 \\
x_1^2 + x_2^2 + x_3^2 \leq 10
\]
with an objective function to be maximized
\[
f(x) = x_1 x_2 + x_2 x_3
\]
where \(x = (x_1, x_2, x_3)\). Solve 3-D Problem \([3]\).

References


Further reading


Combinatorial optimization

In applied mathematics and theoretical computer science, combinatorial optimization is a topic that consists of finding an optimal object from a finite set of objects.[1] In many such problems, exhaustive search is not feasible. It operates on the domain of those optimization problems, in which the set of feasible solutions is discrete or can be reduced to discrete, and in which the goal is to find the best solution. Some common problems involving combinatorial optimization are the traveling salesman problem ("TSP") and the minimum spanning tree problem. Combinatorial optimization is a subset of mathematical optimization that is related to operations research, algorithm theory, and computational complexity theory. It has important applications in several fields, including artificial intelligence, machine learning, mathematics, auction theory, and software engineering.

Some research literature[2] considers discrete optimization to consist of integer programming together with combinatorial optimization (which in turn is composed of optimization problems dealing with graphs, matroids, and related structures) although all of these topics have closely intertwined research literature. It often involves determining the way to efficiently allocate resources used to find solutions to mathematical problems.

Methods

There is a large amount of literature on polynomial-time algorithms for certain special classes of discrete optimization, a considerable amount of it unified by the theory of linear programming. Some examples of combinatorial optimization problems that fall into this framework are shortest paths and shortest path trees, flows and circulations, spanning trees, matching, and matroid problems.

For NP-complete discrete optimization problems, current research literature includes the following topics:

- polynomial-time exactly-solvable special cases of the problem at hand (e.g. see fixed-parameter tractable)
- algorithms that perform well on "random" instances (e.g. for TSP)
- approximation algorithms that run in polynomial time and find a solution that is "close" to optimal
- solving real-world instances that arise in practice and do not necessarily exhibit the worst-case behavior inherent in NP-complete problems (e.g. TSP instances with tens of thousands of nodes[3]).

Combinatorial optimization problems can be viewed as searching for the best element of some set of discrete items, therefore, in principle, any sort of search algorithm or metaheuristic can be used to solve them. However, generic search algorithms are not guaranteed to find an optimal solution, nor are they guaranteed to run quickly (in polynomial time). Since some discrete optimization problems are NP-complete, such as the traveling salesman problem, this is expected unless P=NP.
Specific problems

- Vehicle routing problem
- Traveling salesman problem
- Minimum spanning tree problem
- Linear programming (if the solution space is the choice of which variables to make basic)
- Integer programming
- Eight queens puzzle - A constraint satisfaction problem. When applying standard combinatorial optimization algorithms to this problem, one would usually treat the goal function as the number of unsatisfied constraints (e.g. number of attacks) rather than whether the whole problem is satisfied or not.
- Knapsack problem
- Cutting stock problem
- Assignment problem
- Weapon target assignment problem

Lookahead

In artificial intelligence, lookahead is an important component of combinatorial search which specifies, roughly, how deeply the graph representing the problem is explored. The need for a specific limit on lookahead comes from the large problem graphs in many applications, such as computer chess and computer Go. A naive breadth-first search of these graphs would quickly consume all the memory of any modern computer. By setting a specific lookahead limit, the algorithm's time can be carefully controlled; its time increases exponentially as the lookahead limit increases.

More sophisticated search techniques such as alpha-beta pruning are able to eliminate entire subtrees of the search tree from consideration. When these techniques are used, lookahead is not a precisely defined quantity, but instead either the maximum depth searched or some type of average.

Further reading


References

[1] Schrijver, p. 1
[4] Take one city, and take all possible orders of the other 14 cities. Then divide by two because it does not matter in which direction in time they come after each other: 14!/2 = 43,589,145,600.
External links


Lecture notes

- Integer programming (http://people.brunel.ac.uk/~mastjjb/jeb/or/ip.html) notes, J E Beasley.

Source code

- Java Combinatorial Optimization Platform (http://sourceforge.net/projects/jcop/) open source project.

Others


- Pierluigi Crescenzi, Viggo Kann, Magnús Halldórsson, Marek Karpinski, Gerhard Woeginger, A Compendium of NP Optimization Problems (http://www.nada.kth.se/~viggo/wwwcompendium/).


- Arnab Das and Bikas K Chakrabarti, Rev. Mod. Phys. 80 1061 (2008)
Travelling salesman problem

The travelling salesman problem (TSP) is an NP-hard problem in combinatorial optimization studied in operations research and theoretical computer science. Given a list of cities and their pairwise distances, the task is to find the shortest possible route that visits each city exactly once and returns to the origin city. It is a special case of the travelling purchaser problem.

The problem was first formulated as a mathematical problem in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved.

The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept city represents, for example, customers, soldering points, or DNA fragments, and the concept distance represents travelling times or cost, or a similarity measure between DNA fragments. In many applications, additional constraints such as limited resources or time windows make the problem considerably harder.

In the theory of computational complexity, the decision version of the TSP (where, given a length $L$, the task is to decide whether any tour is shorter than $L$) belongs to the class of NP-complete problems. Thus, it is likely that the worst-case running time for any algorithm for the TSP increases exponentially with the number of cities.

History

The origins of the travelling salesman problem are unclear. A handbook for travelling salesmen from 1832 mentions the problem and includes example tours through Germany and Switzerland, but contains no mathematical treatment.[1] The travelling salesman problem was defined in the 1800s by the Irish mathematician W. R. Hamilton and by the British mathematician Thomas Kirkman. Hamilton’s Icosian Game was a recreational puzzle based on finding a Hamiltonian cycle.[2] The general form of the TSP appears to have been first studied by mathematicians during the 1930s in Vienna and at Harvard, notably by Karl Menger, who defines the problem, considers the obvious brute-force algorithm, and observes the non-optimality of the nearest neighbour heuristic:

We denote by messenger problem (since in practice this question should be solved by each postman, anyway also by many travelers) the task to find, for finitely many points whose pairwise distances are known, the shortest route connecting the points. Of course, this problem is solvable by finitely many trials. Rules which would push the number of trials below the number of permutations of the given points, are not known. The rule that one first should go from the starting point to the closest point, then to the point closest to this, etc., in general does not yield the shortest route.[3]

Hassler Whitney at Princeton University introduced the name travelling salesman problem soon after.[4]

In the 1950s and 1960s, the problem became increasingly popular in scientific circles in Europe and the USA. Notable contributions were made by George Dantzig, Delbert Ray Fulkerson and Selmer M. Johnson at the RAND
Corporation in Santa Monica, who expressed the problem as an integer linear program and developed the cutting plane method for its solution. With these new methods they solved an instance with 49 cities to optimality by constructing a tour and proving that no other tour could be shorter. In the following decades, the problem was studied by many researchers from mathematics, computer science, chemistry, physics, and other sciences.

Richard M. Karp showed in 1972 that the Hamiltonian cycle problem was NP-complete, which implies the NP-hardness of TSP. This supplied a mathematical explanation for the apparent computational difficulty of finding optimal tours.

Great progress was made in the late 1970s and 1980, when Grötschel, Padberg, Rinaldi and others managed to exactly solve instances with up to 2392 cities, using cutting planes and branch-and-bound.

In the 1990s, Applegate, Bixby, Chvátal, and Cook developed the program Concorde that has been used in many recent record solutions. Gerhard Reinelt published the TSPLIB in 1991, a collection of benchmark instances of varying difficulty, which has been used by many research groups for comparing results. In 2005, Cook and others computed an optimal tour through a 33,810-city instance given by a microchip layout problem, currently the largest solved TSPLIB instance. For many other instances with millions of cities, solutions can be found that are guaranteed to be within 1% of an optimal tour.

Description

As a graph problem

TSP can be modelled as an undirected weighted graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's length. It is a minimization problem starting and finishing at a specified vertex after having visited each other vertex exactly once. Often, the model is a complete graph (i.e. each pair of vertices is connected by an edge). If no path exists between two cities, adding an arbitrarily long edge will complete the graph without affecting the optimal tour.

Asymmetric and symmetric

In the symmetric TSP, the distance between two cities is the same in each opposite direction, forming an undirected graph. This symmetry halves the number of possible solutions. In the asymmetric TSP, paths may not exist in both directions or the distances might be different, forming a directed graph. Traffic collisions, one-way streets, and airfares for cities with different departure and arrival fees are examples of how this symmetry could break down.

Related problems

- An equivalent formulation in terms of graph theory is: Given a complete weighted graph (where the vertices would represent the cities, the edges would represent the roads, and the weights would be the cost or distance of that road), find a Hamiltonian cycle with the least weight.
- The requirement of returning to the starting city does not change the computational complexity of the problem, see Hamiltonian path problem.
- Another related problem is the bottleneck travelling salesman problem (bottleneck TSP): Find a Hamiltonian cycle in a weighted graph with the minimal weight of the weightiest edge. The problem is of considerable practical importance, apart from evident transportation and logistics areas. A classic example is in printed circuit manufacturing: scheduling of a route of the drill machine to drill holes in a PCB. In robotic machining or drilling
applications, the "cities" are parts to machine or holes (of different sizes) to drill, and the "cost of travel" includes time for retooling the robot (single machine job sequencing problem).

- The generalized travelling salesman problem deals with "states" that have (one or more) "cities" and the salesman has to visit exactly one "city" from each "state". Also known as the "travelling politician problem". One application is encountered in ordering a solution to the cutting stock problem in order to minimise knife changes. Another is concerned with drilling in semiconductor manufacturing, see e.g. U.S. Patent 7054798 [5]. Surprisingly, Behzad and Modarres [6] demonstrated that the generalised travelling salesman problem can be transformed into a standard travelling salesman problem with the same number of cities, but a modified distance matrix.

- The sequential ordering problem deals with the problem of visiting a set of cities where precedence relations between the cities exist.

- The travelling purchaser problem deals with a purchaser who is charged with purchasing a set of products. He can purchase these products in several cities, but at different prices and not all cities offer the same products. The objective is to find a route between a subset of the cities, which minimizes total cost (travel cost + purchasing cost) and which enables the purchase of all required products.

**Computing a solution**

The traditional lines of attack for the NP-hard problems are the following:

- Devising algorithms for finding exact solutions (they will work reasonably fast only for small problem sizes).
- Devising "suboptimal" or heuristic algorithms, i.e., algorithms that deliver either seemingly or probably good solutions, but which could not be proved to be optimal.
- Finding special cases for the problem ("subproblems") for which either better or exact heuristics are possible.

**Computational complexity**

The problem has been shown to be NP-hard (more precisely, it is complete for the complexity class $\text{FP}^{\text{NP}}$; see function problem), and the decision problem version ("given the costs and a number $x$, decide whether there is a round-trip route cheaper than $x$") is NP-complete. The bottleneck travelling salesman problem is also NP-hard. The problem remains NP-hard even for the case when the cities are in the plane with Euclidean distances, as well as in a number of other restrictive cases. Removing the condition of visiting each city "only once" does not remove the NP-hardness, since it is easily seen that in the planar case there is an optimal tour that visits each city only once (otherwise, by the triangle inequality, a shortcut that skips a repeated visit would not increase the tour length).

**Complexity of approximation**

In the general case, finding a shortest travelling salesman tour is NPO-complete. If the distance measure is a metric and symmetric, the problem becomes APX-complete and Christofides's algorithm approximates it within 1.5. If the distances are restricted to 1 and 2 (but still are a metric) the approximation ratio becomes 7/6. In the asymmetric, metric case, only logarithmic performance guarantees are known, the best current algorithm achieves performance ratio $0.814 \log n$; it is an open question if a constant factor approximation exists.

The corresponding maximization problem of finding the longest travelling salesman tour is approximable within 63/38. If the distance function is symmetric, the longest tour can be approximated within 4/3 by a deterministic algorithm and within $(33 + \epsilon)/25$ by a randomised algorithm.
Exact algorithms

The most direct solution would be to try all permutations (ordered combinations) and see which one is cheapest (using brute force search). The running time for this approach lies within a polynomial factor of $O(n!)$, the factorial of the number of cities, so this solution becomes impractical even for only 20 cities. One of the earliest applications of dynamic programming is the Held–Karp algorithm that solves the problem in time $O(n^2 2^n)$.[14]

The dynamic programming solution requires exponential space. Using inclusion–exclusion, the problem can be solved in time within a polynomial factor of $2^n$ and polynomial space.[15]

Improving these time bounds seems to be difficult. For example, it has not been determined whether an exact algorithm for TSP that runs in time $O(1.9999^n)$ exists.[16]

Other approaches include:

- Various branch-and-bound algorithms, which can be used to process TSPs containing 40–60 cities.
- Progressive improvement algorithms which use techniques reminiscent of linear programming. Works well for up to 200 cities.
- Implementations of branch-and-bound and problem-specific cut generation; this is the method of choice for solving large instances. This approach holds the current record, solving an instance with 85,900 cities, see Applegate et al. (2006).

An exact solution for 15,112 German towns from TSPLIB was found in 2001 using the cutting-plane method proposed by George Dantzig, Ray Fulkerson, and Selmer M. Johnson in 1954, based on linear programming. The computations were performed on a network of 110 processors located at Rice University and Princeton University (see the Princeton external link). The total computation time was equivalent to 22.6 years on a single 500 MHz Alpha processor. In May 2004, the travelling salesman problem of visiting all 24,978 towns in Sweden was solved: a tour of length approximately 72,500 kilometers was found and it was proven that no shorter tour exists.[17]

In March 2005, the travelling salesman problem of visiting all 33,810 points in a circuit board was solved using Concorde TSP Solver: a tour of length 66,048,945 units was found and it was proven that no shorter tour exists. The computation took approximately 15.7 CPU-years (Cook et al. 2006). In April 2006 an instance with 85,900 points was solved using Concorde TSP Solver, taking over 136 CPU-years, see Applegate et al. (2006).

Heuristic and approximation algorithms

Various heuristics and approximation algorithms, which quickly yield good solutions have been devised. Modern methods can find solutions for extremely large problems (millions of cities) within a reasonable time which are with a high probability just 2–3% away from the optimal solution.

Several categories of heuristics are recognized.

Constructive heuristics

The nearest neighbour (NN) algorithm (or so-called greedy algorithm) lets the salesman choose the nearest unvisited city as his next move. This algorithm quickly yields an effectively short route. For N cities randomly distributed on a plane, the algorithm on average yields a path 25% longer than the shortest possible path.[18] However, there exist many specially arranged city distributions which make the NN algorithm give the worst route ( Gutin, Yeo, and Zverovich, 2002). This is true for both asymmetric and symmetric TSPs (Gutin and Yeo, 2007). Rosenkrantz et al. [1977] showed that the NN algorithm has the approximation factor $\Theta(\log |V|)$ for instances satisfying the triangle inequality.

Constructions based on a minimum spanning tree have an approximation ratio of 2. The Christofides algorithm achieves a ratio of 1.5.

The bitonic tour of a set of points is the minimum-perimeter monotone polygon that has the points as its vertices; it can be computed efficiently by dynamic programming.
Another constructive heuristic, Match Twice and Stitch (MTS) (Kahng, Reda 2004 [19]), performs two sequential matchings, where the second matching is executed after deleting all the edges of the first matching, to yield a set of cycles. The cycles are then stitched to produce the final tour.

**Iterative improvement**

Pairwise exchange, or Lin–Kernighan heuristics

The pairwise exchange or 2-opt technique involves iteratively removing two edges and replacing these with two different edges that reconnect the fragments created by edge removal into a new and shorter tour. This is a special case of the $k$-opt method. Note that the label Lin–Kernighan is an often heard misnomer for 2-opt. Lin–Kernighan is actually a more general method.

$k$-opt heuristic

Take a given tour and delete $k$ mutually disjoint edges. Reassemble the remaining fragments into a tour, leaving no disjoint subtours (that is, don't connect a fragment's endpoints together). This in effect simplifies the TSP under consideration into a much simpler problem. Each fragment endpoint can be connected to $2k - 2$ other possibilities: of $2k$ total fragment endpoints available, the two endpoints of the fragment under consideration are disallowed. Such a constrained $2k$-city TSP can then be solved with brute force methods to find the least-cost recombination of the original fragments. The $k$-opt technique is a special case of the $V$-opt or variable-opt technique. The most popular of the $k$-opt methods are 3-opt, and these were introduced by Shen Lin of Bell Labs in 1965. There is a special case of 3-opt where the edges are not disjoint (two of the edges are adjacent to one another). In practice, it is often possible to achieve substantial improvement over 2-opt without the combinatorial cost of the general 3-opt by restricting the 3-changes to this special subset where two of the removed edges are adjacent. This so-called two-and-a-half-opt typically falls roughly midway between 2-opt and 3-opt, both in terms of the quality of tours achieved and the time required to achieve those tours.

$V$-opt heuristic

The variable-opt method is related to, and a generalization of the $k$-opt method. Whereas the $k$-opt methods remove a fixed number ($k$) of edges from the original tour, the variable-opt methods do not fix the size of the edge set to remove. Instead they grow the set as the search process continues. The best known method in this family is the Lin–Kernighan method (mentioned above as a misnomer for 2-opt). Shen Lin and Brian Kernighan first published their method in 1972, and it was the most reliable heuristic for solving travelling salesman problems for nearly two decades. More advanced variable-opt methods were developed at Bell Labs in the late 1980s by David Johnson and his research team. These methods (sometimes called Lin–Kernighan–Johnson) build on the Lin–Kernighan method, adding ideas from tabu search and evolutionary computing. The basic Lin–Kernighan technique gives results that are guaranteed to be at least 3-opt. The Lin–Kernighan–Johnson methods compute a Lin–Kernighan tour, and then perturb the tour by what has been described as a mutation that removes at least four edges and reconnecting the tour in a different way, then $v$-opting the new tour. The mutation is often enough to move the tour from the local minimum identified by Lin–Kernighan. $V$-opt methods are widely considered the most powerful heuristics for the problem, and are able to address special cases, such as the Hamilton Cycle Problem and other non-metric TSPs that other heuristics fail on. For many years Lin–Kernighan–Johnson had identified optimal solutions for all TSPs where an optimal solution was known and had identified the best known solutions for all other TSPs on which the method had been tried.
Randomised improvement

Optimized Markov chain algorithms which use local searching heuristic sub-algorithms can find a route extremely close to the optimal route for 700 to 800 cities.

TSP is a touchstone for many general heuristics devised for combinatorial optimization such as genetic algorithms, simulated annealing, Tabu search, ant colony optimization, river formation dynamics (see swarm intelligence) and the cross entropy method.

Ant colony optimization

Artificial intelligence researcher Marco Dorigo described in 1997 a method of heuristically generating "good solutions" to the TSP using a simulation of an ant colony called ACS (Ant Colony System).\[^{[20]}\] It models behavior observed in real ants to find short paths between food sources and their nest, an emergent behaviour resulting from each ant's preference to follow trail pheromones deposited by other ants.

ACS sends out a large number of virtual ant agents to explore many possible routes on the map. Each ant probabilistically chooses the next city to visit based on a heuristic combining the distance to the city and the amount of virtual pheromone deposited on the edge to the city. The ants explore, depositing pheromone on each edge that they cross, until they have all completed a tour. At this point the ant which completed the shortest tour deposits virtual pheromone along its complete tour route (global trail updating). The amount of pheromone deposited is inversely proportional to the tour length: the shorter the tour, the more it deposits.

Special cases

Metric TSP

In the metric TSP, also known as delta-TSP or Δ-TSP, the intercity distances satisfy the triangle inequality.

A very natural restriction of the TSP is to require that the distances between cities form a metric, i.e., they satisfy the triangle inequality. This can be understood as the absence of "shortcuts", in the sense that the direct connection from $A$ to $B$ is never longer than the route via intermediate $C$:

$$d_{AB} \leq d_{AC} + d_{CB}$$

The edge lengths then form a metric on the set of vertices. When the cities are viewed as points in the plane, many natural distance functions are metrics, and so many natural instances of TSP satisfy this constraint.

The following are some examples of metric TSPs for various metrics.

- In the Euclidean TSP (see below) the distance between two cities is the Euclidean distance between the corresponding points.
In the rectilinear TSP the distance between two cities is the sum of the differences of their \(x\)- and \(y\)-coordinates. This metric is often called the Manhattan distance or city-block metric.

In the maximum metric, the distance between two points is the maximum of the absolute values of differences of their \(x\)- and \(y\)-coordinates.

The last two metrics appear for example in routing a machine that drills a given set of holes in a printed circuit board. The Manhattan metric corresponds to a machine that adjusts first one co-ordinate, and then the other, so the time to move to a new point is the sum of both movements. The maximum metric corresponds to a machine that adjusts both co-ordinates simultaneously, so the time to move to a new point is the slower of the two movements.

In its definition, the TSP does not allow cities to be visited twice, but many applications do not need this constraint. In such cases, a symmetric, non-metric instance can be reduced to a metric one. This replaces the original graph with a complete graph in which the inter-city distance \(d_{AB}\) is replaced by the shortest path between \(A\) and \(B\) in the original graph.

There is a constant-factor approximation algorithm for the metric TSP due to Christofides\(^{[21]}\) that always finds a tour of length at most 1.5 times the shortest tour. In the next paragraphs, we explain a weaker (but simpler) algorithm which finds a tour of length at most twice the shortest tour.

The length of the minimum spanning tree of the network is a natural lower bound for the length of the optimal route. In the TSP with triangle inequality case it is possible to prove upper bounds in terms of the minimum spanning tree and design an algorithm that has a provable upper bound on the length of the route. The first published (and the simplest) example follows:

1. Construct the minimum spanning tree.
2. Duplicate all its edges. That is, wherever there is an edge from \(u\) to \(v\), add a second edge from \(u\) to \(v\). This gives us an Eulerian graph.
3. Find a Eulerian cycle in it. Clearly, its length is twice the length of the tree.
4. Convert the Eulerian cycle into the Hamiltonian one in the following way: walk along the Eulerian cycle, and each time you are about to come into an already visited vertex, skip it and try to go to the next one (along the Eulerian cycle).

It is easy to prove that the last step works. Moreover, thanks to the triangle inequality, each skipping at Step 4 is in fact a shortcut; i.e., the length of the cycle does not increase. Hence it gives us a TSP tour no more than twice as long as the optimal one.

The Christofides algorithm follows a similar outline but combines the minimum spanning tree with a solution of another problem, minimum-weight perfect matching. This gives a TSP tour which is at most 1.5 times the optimal.

The Christofides algorithm was one of the first approximation algorithms, and was in part responsible for drawing attention to approximation algorithms as a practical approach to intractable problems. As a matter of fact, the term "algorithm" was not commonly extended to approximation algorithms until later; the Christofides algorithm was initially referred to as the Christofides heuristic.

In the special case that distances between cities are all either one or two (and thus the triangle inequality is necessarily satisfied), there is a polynomial-time approximation algorithm that finds a tour of length at most \(8/7\) times the optimal tour length.\(^{[22]}\) However, it is a long-standing (since 1975) open problem to improve the Christofides approximation factor of 1.5 for general metric TSP to a smaller constant. It is known that, unless \(P = NP\), there is no polynomial-time algorithm that finds a tour of length at most \(220/219=1.00456...\) times the optimal tour's length.\(^{[23]}\) In the case of bounded metrics it is known that there is no polynomial time algorithm that constructs a tour of length at most \(321/320\) times the optimal tour's length, unless \(P = NP\).\(^{[24]}\)
Travelling salesman problem

Euclidean TSP

The **Euclidean TSP**, or **planar TSP**, is the TSP with the distance being the ordinary Euclidean distance.

The Euclidean TSP is a particular case of the metric TSP, since distances in a plane obey the triangle inequality.

Like the general TSP, the Euclidean TSP (and therefore the general metric TSP) is NP-complete.\(^{[25]}\) However, in some respects it seems to be easier than the general metric TSP. For example, the minimum spanning tree of the graph associated with an instance of the Euclidean TSP is a Euclidean minimum spanning tree, and so can be computed in expected $O(n \log n)$ time for $n$ points (considerably less than the number of edges). This enables the simple 2-approximation algorithm for TSP with triangle inequality above to operate more quickly.

In general, for any $c > 0$, where $d$ is the number of dimensions in the Euclidean space, there is a polynomial-time algorithm that finds a tour of length at most $(1 + 1/c)$ times the optimal for geometric instances of TSP in $O\left(n(\log n)^{O(c\sqrt{d})}d^{d-1}\right)$ time; this is called a polynomial-time approximation scheme (PTAS).\(^{[26]}\) Sanjeev Arora and Joseph S. B. Mitchell were awarded the Gödel Prize in 2010 for their concurrent discovery of a PTAS for the Euclidean TSP.

In practice, heuristics with weaker guarantees continue to be used.

Asymmetric TSP

In most cases, the distance between two nodes in the TSP network is the same in both directions. The case where the distance from $A$ to $B$ is not equal to the distance from $B$ to $A$ is called asymmetric TSP. A practical application of an asymmetric TSP is route optimisation using street-level routing (which is made asymmetric by one-way streets, slip-roads, motorways, etc.).

**Solving by conversion to symmetric TSP**

Solving an asymmetric TSP graph can be somewhat complex. The following is a 3×3 matrix containing all possible path weights between the nodes $A$, $B$ and $C$. One option is to turn an asymmetric matrix of size $N$ into a symmetric matrix of size $2N$.\(^{[27]}\)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>6</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

\(^{1+}\) Asymmetric path weights

To double the size, each of the nodes in the graph is duplicated, creating a second *ghost node*. Using duplicate points with very low weights, such as $-\infty$, provides a cheap route "linking" back to the real node and allowing symmetric evaluation to continue. The original 3×3 matrix shown above is visible in the bottom left and the inverse of the original in the top-right. Both copies of the matrix have had their diagonals replaced by the low-cost hop paths, represented by $-\infty$. 
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A'</th>
<th>B'</th>
<th>C'</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>∞</td>
<td>6</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>∞</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>3</td>
<td>∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A'</td>
<td>∞</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B'</td>
<td>6</td>
<td>∞</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C'</td>
<td>5</td>
<td>4</td>
<td>∞</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|+ Symmetric path weights

The original 3×3 matrix would produce two Hamiltonian cycles (a path that visits every node once), namely A-B-C-A [score 9] and A-C-B-A [score 12]. Evaluating the 6×6 symmetric version of the same problem now produces many paths, including A-A’-B-B’-C-C’-A, A-B’-C-A’-A, A-A’-B-C’-A [all score 9 – ∞].

The important thing about each new sequence is that there will be an alternation between dashed (A’, B’, C’) and un-dashed nodes (A, B, C) and that the link to ”jump” between any related pair (A-A’) is effectively free. A version of the algorithm could use any weight for the A-A’ path, as long as that weight is lower than all other path weights present in the graph. As the path weight to ”jump” must effectively be ”free”, the value zero (0) could be used to represent this cost—if zero is not being used for another purpose already (such as designating invalid paths). In the two examples above, non-existent paths between nodes are shown as a blank square.

**Benchmarks**

For benchmarking of TSP algorithms, TSPLIB\cite{28} is a library of sample instances of the TSP and related problems is maintained, see the TSPLIB external reference. Many of them are lists of actual cities and layouts of actual printed circuits.

**Human performance on TSP**

The TSP, in particular the Euclidean variant of the problem, has attracted the attention of researchers in cognitive psychology. It is observed that humans are able to produce good quality solutions quickly. The first issue of the Journal of Problem Solving\cite{29} is devoted to the topic of human performance on TSP.

**TSP path length for random pointset in a square**

Suppose $N$ points are randomly distributed in a 1 x 1 square with $N$>>1. Consider many such squares. Suppose we want to know the average of the shortest path length (i.e. TSP solution) of each square.

**Lower bound**

\[
\frac{1}{2} \sqrt{N} \text{ is a lower bound obtained by assuming } i \text{ be a point in the tour sequence and } i \text{ has its nearest neighbor as its next in the path.}
\]

\[
\left(\frac{1}{4} + \frac{3}{8}\right) \sqrt{N} \text{ is a better lower bound obtained by assuming } is next is nearest, and } is previous is is second nearest.
\]

\[
\sqrt{\frac{N}{2}} \text{ is an even better lower bound obtained by dividing the path sequence into two parts as before}_i \text{ and after}_i \text{ with each part containing } N/2 \text{ points, and then deleting the before}_i \text{ part to form a diluted pointset (see discussion).}
\]
Travelling salesman problem

- David S. Johnson obtained a lower bound by computer experiment:
  \[0.7080\sqrt{N} + 0.522\] where 0.522 comes from the points near square boundary which have fewer neighbors.
- Christine L. Valenzuela and Antonia J. Jones obtained another lower bound by computer experiment:
  \[0.7078\sqrt{N} + 0.551\]

Upper bound

By applying Simulated Annealing method on samples of \(N=40000\), computer analysis shows an upper bound of

\[\left(\frac{\sqrt{N}}{2} + 0.72\right) \cdot 1.015\]

where 0.72 comes from the boundary effect.

Because the actual solution is only the shortest path, for the purposes of programmatic search another upper bound is the length of any previously discovered approximation.

Analyst's travelling salesman problem

There is an analogous problem in geometric measure theory which asks the following: under what conditions may a subset \(E\) of Euclidean space be contained in a rectifiable curve (that is, when is there a continuous curve that visits every point in \(E\))? This problem is known as the analyst's travelling salesman problem or the geometric travelling salesman problem.

Notes

[1] "Der Handlungsreisende — wie er sein soll und was er zu thun [sic] hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiß zu sein — von einem alten Commiss-Voyageur" (The traveling salesman — how he must be and what he should do in order to be sure to perform his tasks and have success in his business — by a high commis-voyageur)
[2] A discussion of the early work of Hamilton and Kirkman can be found in Graph Theory 1736–1936
[17] Work by David Applegate, AT&T Labs – Research, Robert Bixby, ILOG and Rice University, Vašek Chvátal, Concordia University, William Cook, Georgia Tech, and Keld Helsgaun, Roskilde University is discussed on their project web page hosted by Georgia Tech and last updated in June 2004, here (http://www.tsp.gatech.edu/sweden/)
Travelling salesman problem

References


**Further reading**


• Babin, Gilbert; Deneault, Stéphanie; Laportey, Gilbert (2005), "Improvements to the Or-opt Heuristic for the Symmetric Traveling Salesman Problem" (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.89.9953), Cahiers du GERAD, G-2005-02, Montreal: Group for Research in Decision Analysis.


External links
• Traveling Salesman Problem (http://www.tsp.gatech.edu/index.html) at Georgia Tech
• TSPLIB (http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/) at the University of Heidelberg
• optimap (http://www.gebweb.net/optimap/) an approximation using ACO on GoogleMaps with JavaScript
• tsp (http://travellingsalesmanproblem.appspot.com/) an exact solver using Constraint Programming on GoogleMaps
• Demo applet of a genetic algorithm solving TSPs and VRPTW problems (http://www.dna-evolutions.com/dnaappletsample.html)
• Source code library for the travelling salesman problem (http://www.adaptivebox.net/CILib/code/tspcodes_link.html)
• TSP solvers in R (http://tsp.r-forge.r-project.org/) for symmetric and asymmetric TSPs. Implements various insertion, nearest neighbor and 2-opt heuristics and an interface to Georgia Tech's Concorde and Chained Lin-Kernighan heuristics.
Constraint (mathematics)

In mathematics, a **constraint** is a condition that a solution to an optimization problem must satisfy. There are two types of constraints: **equality constraints** and **inequality constraints**. The set of solutions that satisfy all constraints is called the feasible set.

**Example**

The following is a simple optimization problem:

\[
\begin{align*}
\text{min } f(x) &= x_1^2 + x_2^4 \\
\text{subject to } & \quad x_1 \geq 1 \\
\text{and } & \quad x_2 = 1,
\end{align*}
\]

where \( x \) denotes the vector \((x_1, x_2)\).

In this example, the first line defines the function to be minimized (called the **objective** or **cost function**). The second and third lines define two constraints, the first of which is an inequality constraint and the second of which is an equality constraint. These two constraints define the feasible set of candidate solutions.

Without the constraints, the solution would be \( (0, 0) \) where \( f(x) \) has the lowest value. But this solution does not satisfy the constraints. The solution of the **constrained optimization problem** stated above but \( x = (1, 1) \), which is the point with the smallest value of \( f(x) \) that satisfies the two constraints.

**Terminology**

- If a constraint is an **equality** at a given point, the constraint is said to be **binding**, as the point cannot be varied in the direction of the constraint.
- If a constraint is an **inequality** at a given point, the constraint is said to be **non-binding**, as the point can be varied in the direction of the constraint.
- If a constraint is not satisfied, the point is said to be **infeasible**.

**External links**

- Nonlinear programming FAQ \(^1\)
- Mathematical Programming Glossary \(^2\)

**References**

\(^1\) http://www-unix.mcs.anl.gov/otc/Guide/faq/nonlinear-programming-faq.html
\(^2\) http://glossary.computing.society.informs.org/
Constraint satisfaction problem

Constraint satisfaction problems (CSPs) are mathematical problems defined as a set of objects whose state must satisfy a number of constraints or limitations. CSPs represent the entities in a problem as a homogeneous collection of finite constraints over variables, which is solved by constraint satisfaction methods. CSPs are the subject of intense research in both artificial intelligence and operations research, since the regularity in their formulation provides a common basis to analyze and solve problems of many unrelated families. CSPs often exhibit high complexity, requiring a combination of heuristics and combinatorial search methods to be solved in a reasonable time. The boolean satisfiability problem (SAT), the Satisfiability Modulo Theories (SMT) and answer set programming (ASP) can be roughly thought of as certain forms of the constraint satisfaction problem.

Examples of simple problems that can be modeled as a constraint satisfaction problem:

- Eight queens puzzle
- Map coloring problem
- Sudoku

Examples demonstrating the above are often provided with tutorials of ASP, boolean SAT and SMT solvers. In the general case, constraint problems can be much harder, and may not be expressible in some of these simpler systems.

Formal definition

Formally, a constraint satisfaction problem is defined as a triple \( \langle X, D, C \rangle \), where \( X \) is a set of variables, \( D \) is a domain of values, and \( C \) is a set of constraints. Every constraint is in turn a pair \( \langle t, R \rangle \) (usually represented as a matrix), where \( t \) is an \( n \)-tuple of variables and \( R \) is an \( n \)-ary relation on \( D \). An evaluation of the variables is a function from the set of variables to the domain of values, \( v : X \rightarrow D \). An evaluation \( v \) satisfies a constraint \( \langle (x_1, \ldots, x_n), R \rangle \) if \( v(x_1), \ldots, v(x_n) \in R \). A solution is an evaluation that satisfies all constraints.

Resolution of CSPs

Constraint satisfaction problems on finite domains are typically solved using a form of search. The most used techniques are variants of backtracking, constraint propagation, and local search.

Backtracking is a recursive algorithm. It maintains a partial assignment of the variables. Initially, all variables are unassigned. At each step, a variable is chosen, and all possible values are assigned to it in turn. For each value, the consistency of the partial assignment with the constraints is checked; in case of consistency, a recursive call is performed. When all values have been tried, the algorithm backtracks. In this basic backtracking algorithm, consistency is defined as the satisfaction of all constraints whose variables are all assigned. Several variants of backtracking exists. Backmarking improves the efficiency of checking consistency. Backjumping allows saving part of the search by backtracking "more than one variable" in some cases. Constraint learning infers and saves new constraints that can be later used to avoid part of the search. Look-ahead is also often used in backtracking to attempt to foresee the effects of choosing a variable or a value, thus sometimes determining in advance when a subproblem is satisfiable or unsatisfiable.

Constraint propagation techniques are methods used to modify a constraint satisfaction problem. More precisely, they are methods that enforce a form of local consistency, which are conditions related to the consistency of a group of variables and/or constraints. Constraint propagation has various uses. First, it turns a problem into one that is equivalent but is usually simpler to solve. Second, it may prove satisfiability or unsatisfiability of problems. This is not guaranteed to happen in general; however, it always happens for some forms of constraint propagation and/or for some certain kinds of problems. The most known and used form of local consistency are arc consistency, hyper-arc consistency, and path consistency. The most popular constraint propagation method is the AC-3 algorithm, which enforces arc consistency.
Local search methods are incomplete satisfiability algorithms. They may find a solution of a problem, but they may fail even if the problem is satisfiable. They work by iteratively improving a complete assignment over the variables. At each step, a small number of variables are changed value, with the overall aim of increasing the number of constraints satisfied by this assignment. The min-conflicts algorithm is a local search algorithm specific for CSPs and based in that principle. In practice, local search appears to work well when these changes are also affected by random choices. Integration of search with local search have been developed, leading to hybrid algorithms.

**Theoretical aspects of CSPs**

**Decision problems**

CSPs are also studied in computational complexity theory and finite model theory. An important question is whether for each set of relations, the set of all CSPs that can be represented using only relations chosen from that set is either in P or NP-complete. If such a dichotomy theorem is true, then CSPs provide one of the largest known subsets of NP which avoids NP-intermediate problems, whose existence was demonstrated by Ladner's theorem under the assumption that P ≠ NP. Schaefer's dichotomy theorem handles the case when all the available relations are boolean operators, that is, for domain size 2. Schaefer's dichotomy theorem was recently generalized to a larger class of relations.¹

Most classes of CSPs that are known to be tractable are those where the hypergraph of constraints has bounded treewidth (and there are no restrictions on the set of constraint relations), or where the constraints have arbitrary form but there exist essentially non-unary polymorphisms of the set of constraint relations. Every CSP can also be considered as a conjunctive query containment problem.²

**Function problems**

A similar situation exists between the functional classes FP and #P. By a generalization of Ladner's theorem, there are also problems in neither FP nor #P-complete as long as FP ≠ #P. As in the decision case, a problem in the #CSP is defined by a set of relations. Each problem takes as input a Boolean formula as input and the task is to compute the number of satisfying assignments. This can be further generalized by using larger domain sizes and attaching a weight to each satisfying assignment and computing the sum of these weights. It is known that any complex weighted #CSP problem is either in FP or #P-hard.³

**Variants of CSPs**

The classic model of Constraint Satisfaction Problem defines a model of static, inflexible constraints. This rigid model is a shortcoming that makes it difficult to represent problems easily.⁴ Several modifications of the basic CSP definition have been proposed to adapt the model to a wide variety of problems.

**Dynamic CSPs**

Dynamic CSPs⁵ (DCSPs) are useful when the original formulation of a problem is altered in some way, typically because the set of constraints to consider evolves because of the environment.⁶ DCSPs are viewed as a sequence of static CSPs, each one a transformation of the previous one in which variables and constraints can be added (restriction) or removed (relaxation). Information found in the initial formulations of the problem can be used to refine the next ones. The solving method can be classified according to the way in which information is transferred:

- Oracles: the solution found to previous CSPs in the sequence are used as heuristics to guide the resolution of the current CSP from scratch.
- Local repair: each CSP is calculated starting from the partial solution of the previous one and repairing the inconsistent constraints with local search.
• Constraint recording: new constraints are defined in each stage of the search to represent the learning of inconsistent group of decisions. Those constraints are carried over the new CSP problems.

**Flexible CSPs**

Classic CSPs treat constraints as hard, meaning that they are imperative (each solution must satisfy all them) and inflexible (in the sense that they must be completely satisfied or else they are completely violated). Flexible CSPs relax those assumptions, partially relaxing the constraints and allowing the solution to not comply with all them. This is similar to preferences in preference-based planning. Some types of flexible CSPs include:

• MAX-CSP, where a number of constraints are allowed to be violated, and the quality of a solution is measured by the number of satisfied constraints.

• Weighted CSP, a MAX-CSP in which each violation of a constraint is weighted according to a predefined preference. Thus satisfying constraint with more weight is preferred.

• Fuzzy CSP model constraints as fuzzy relations in which the satisfaction of a constraint is a continuous function of its variables' values, going from fully satisfied to fully violated.

**References**


**Further reading**


**External links**

• CSP Tutorial (http://4c.ucc.ie/web/outreach/tutorial.html)


• Tomás Feder, Constraint satisfaction: a personal perspective (http://theory.stanford.edu/~tomas/consmod.pdf), manuscript.

• Constraints archive (http://4c.ucc.ie/web/archive/index.jsp)
Constraint satisfaction problem

In artificial intelligence and operations research, constraint satisfaction is the process of finding a solution to a set of constraints that impose conditions that the variables must satisfy. A solution is therefore a vector of variables that satisfies all constraints.

The techniques used in constraint satisfaction depend on the kind of constraints being considered. Often used are constraints on a finite domain, to the point that constraint satisfaction problems are typically identified with problems based on constraints on a finite domain. Such problems are usually solved via search, in particular a form of backtracking or local search. Constraint propagation are other methods used on such problems; most of them are incomplete in general, that is, they may solve the problem or prove it unsatisfiable, but not always. Constraint propagation methods are also used in conjunction with search to make a given problem simpler to solve. Other considered kinds of constraints are on real or rational numbers; solving problems on these constraints is done via variable elimination or the simplex algorithm.

Constraint satisfaction originated in the field of artificial intelligence in the 1970s (see for example (Laurière 1978)). During the 1980s and 1990s, embedding of constraints into a programming language were developed. Languages often used for constraint programming are Prolog and C++.

Constraint satisfaction problem

As originally defined in artificial intelligence, constraints enumerate the possible values a set of variables may take. Informally, a finite domain is a finite set of arbitrary elements. A constraint satisfaction problem on such domain contains a set of variables whose values can only be taken from the domain, and a set of constraints, each constraint specifying the allowed values for a group of variables. A solution to this problem is an evaluation of the variables that satisfies all constraints. In other words, a solution is a way for assigning a value to each variable in such a way that all constraints are satisfied by these values.

In some circumstances, there may exist additional requirements: one may be interested not only in the solution (and in the fastest or most computationally efficient way to reach it) but in how it was reached; e.g. one may want the "simplest" solution ("simplest" in a logical, non computational sense that has to be precisely defined). This is often the case in logic games such as Sudoku.

In practice, constraints are often expressed in compact form, rather than enumerating all values of the variables that would satisfy the constraint. One of the most used constraints is the one establishing that the values of the affected variables must be all different.

Problems that can be expressed as constraint satisfaction problems are the Eight queens puzzle, the Sudoku solving problem, the Boolean satisfiability problem, scheduling problems and various problems on graphs such as the graph coloring problem.

While usually not included in the above definition of a constraint satisfaction problem, arithmetic equations and inequalities bound the values of the variables they contain and can therefore be considered a form of constraints.
Their domain is the set of numbers (either integer, rational, or real), which is infinite: therefore, the relations of these constraints may be infinite as well; for example, \( X = Y + 1 \) has an infinite number of pairs of satisfying values. Arithmetic equations and inequalities are often not considered within the definition of a "constraint satisfaction problem", which is limited to finite domains. They are however used often in constraint programming.

**Solving**

Constraint satisfaction problems on finite domains are typically solved using a form of search. The most used techniques are variants of backtracking, constraint propagation, and local search. These techniques are used on problems with nonlinear constraints.

In case there is a requirement on "simplicity", a pure logic, pattern based approach was first introduced for the Sudoku CSP in the book *The Hidden Logic of Sudoku*\(^1\). It has recently been generalized to any finite CSP in another book by the same author: *Constraint Resolution Theories*\(^2\).

Variable elimination and the simplex algorithm are used for solving linear and polynomial equations and inequalities, and problems containing variables with infinite domain. These are typically solved as optimization problems in which the optimized function is the number of violated constraints.

**Complexity**

Solving a constraint satisfaction problem on a finite domain is an NP complete problem with respect to the domain size. Research has shown a number of tractable subcases, some limiting the allowed constraint relations, some requiring the scopes of constraints to form a tree, possibly in a reformulated version of the problem. Research has also established relationship of the constraint satisfaction problem with problems in other areas such as finite model theory.

A very different aspect of complexity appears when one fixes the size of the domain. It is about the complexity distribution of minimal instances of a CSP of fixed size (e.g. Sudoku(9x9)). Here, complexity is measured according to the above-mentioned "simplicity" requirement (see *Unbiased Statistics of a CSP - A Controlled-Bias Generator*\(^3\) or *Constraint Resolution Theories*\(^2\)). In this context, a minimal instance is an instance with a unique solution such that if any given (or clue) is deleted from it, the resulting instance has several solutions (statistics can only be meaningful on the set of minimal instances).

**Constraint programming**

Constraint programming is the use of constraints as a programming language to encode and solve problems. This is often done by embedding constraints into a programming language, which is called the host language. Constraint programming originated from a formalization of equalities of terms in Prolog II, leading to a general framework for embedding constraints into a logic programming language. The most common host languages are Prolog, C++, and Java, but other languages have been used as well.

**Constraint logic programming**

A constraint logic program is a logic program that contains constraints in the bodies of clauses. As an example, the clause \( A(X) :- X > 0, B(X) \) is a clause containing the constraint \( X > 0 \) in the body. Constraints can also be present in the goal. The constraints in the goal and in the clauses used to prove the goal are accumulated into a set called constraint store. This set contains the constraints the interpreter has assumed satisfiable in order to proceed in the evaluation. As a result, if this set is detected unsatisfiable, the interpreter backtracks. Equations of terms, as used in logic programming, are considered a particular form of constraints which can be simplified using unification. As a result, the constraint store can be considered an extension of the concept of substitution that is used in regular logic programming. The most common kinds of constraints used in constraint logic programming are constraints over
integers/rational/real numbers and constraints over finite domains.

Concurrent constraint logic programming languages have also been developed. They significantly differ from non-concurrent constraint logic programming in that they are aimed at programming concurrent processes that may not terminate. Constraint handling rules can be seen as a form of concurrent constraint logic programming, but are also sometimes used within a non-concurrent constraint logic programming language. They allow for rewriting constraints or to infer new ones based on the truth of conditions.

**Constraint satisfaction toolkits**

Constraint satisfaction toolkits are software libraries for imperative programming languages that are used to encode and solve a constraint satisfaction problem.

- Cassowary constraint solver is an open source project for constraint satisfaction (accessible from C, Java, Python and other languages).
- Comet, a commercial programming language and toolkit
- Gecode, an open source portable toolkit written in C++ developed as a production-quality and highly efficient implementation of a complete theoretical background.
- JaCoP (solver) an open source Java constraint solver [4]

**Other constraint programming languages**

Constraint toolkits are a way for embedding constraints into an imperative programming language. However, they are only used as external libraries for encoding and solving problems. An approach in which constraints are integrated into an imperative programming language is taken in the Kaleidoscope programming language.

Constraints have also been embedded into functional programming languages.

**References**


External links
• CSP Tutorial (http://4c.ucc.ie/web/outreach/tutorial.html)
Heuristic (computer science)

In computer science and optimization a heuristic is a rule of thumb learned from experience but not always justified by an underlying theory. Heuristics are often used to improve efficiency or effectiveness of optimization algorithms, either by finding an approximate answer when the optimal answer would be prohibitively difficult or to make an algorithm faster. Usually, heuristics do not guarantee that an optimal solution is ever found. On the other hand, results about NP-hardness in theoretical computer science make heuristics the only viable alternative for many complex optimization problems which are significant in the real world.

An example of an approximation is one Jon Bentley described for solving the travelling salesman problem (TSP) where it was selecting the order to draw using a pen plotter. TSP is known to be NP-hard so an optimal solution for even moderate size problem is intractable. Instead the greedy algorithm can be used to to give a good but not optimal (it is an approximation to the optimal answer) in a short amount of time. The greedy algorithm heuristic says to pick whatever is currently the best next step regardless of whether that precludes good steps later. It is a heuristic in that practice says it is a good enough solution, theory says there are better solutions (and even can tell how much better in some cases).\[1\]

An example of making an algorithm faster occurs in certain search methods where it tries every possibility at each step but can stop the search if the current possibility is already worse than the best solution already found; in this sort of algorithm a heuristic can be used to try good choices first so that later it can eliminate bad paths early. (See alpha-beta pruning).

References


Multi-objective optimization

Multi-objective optimization (or multi-objective programming),\[1][2\] also known as multi-criteria or multi-attribute optimization, is the process of simultaneously optimizing two or more conflicting objectives subject to certain constraints.

Multiobjective optimization problems can be found in various fields: product and process design, finance, aircraft design, the oil and gas industry, automobile design, or wherever optimal decisions need to be taken in the presence of trade-offs between two or more conflicting objectives. Maximizing profit and minimizing the cost of a product; maximizing performance and minimizing fuel consumption of a vehicle; and minimizing weight while maximizing the strength of a particular component are examples of multi-objective optimization problems.

For nontrivial multiobjective problems, one cannot identify a single solution that simultaneously optimizes each objective. While searching for solutions, one reaches points such that, when attempting to improve an objective further, other objectives suffer as a result. A tentative solution is called non-dominated, Pareto optimal, or Pareto efficient if it cannot be eliminated from consideration by replacing it with another solution which improves an objective without worsening another one. Finding such non-dominated solutions, and quantifying the trade-offs in satisfying the different objectives, is the goal when setting up and solving a multiobjective optimization problem.
When the role of the decision maker (DM) is considered, one distinguishes between: *a priori* approaches that require all knowledge about the relative importance of the objectives before starting the solution process, *a posteriori* approaches that deliver a large representative set of Pareto-optimal solutions among which the DM chooses the preferred one, and *interactive* approaches which alternate the production of some Pareto-optimal solutions with the feedback by the DM, so that a better tuning of the preferred combination of objectives can be learned.\[^3\]

**Introduction**

In mathematical terms, the multiobjective problem can be written as:

\[
\min_x \left[ \mu_1(x), \mu_2(x), \ldots, \mu_n(x) \right]^T \\
\text{s.t.} \\
g(x) \leq 0 \\
h(x) = 0 \\
x_l \leq x \leq x_u
\]

where \( \mu_i \) is the \( i \)-th objective function, \( g \) and \( h \) are the inequality and equality constraints, respectively, and \( x \) is the vector of optimization or decision variables. The solution to the above problem is a set of Pareto points. Thus, instead of being a unique solution to the problem, the solution to a multiobjective problem is a possibly infinite set of Pareto points.

A design point in objective space is termed Pareto optimal if there does not exist another feasible design objective vector \( \mu^* \) such that \( \mu_i \leq \mu_i^* \) for all \( i \in \{1, 2, \ldots, n\} \), and \( \mu_j < \mu_j^* \) for at least one index of \( j \), \( j \in \{1, 2, \ldots, n\} \).

**Solution methods**

Some methods for finding a solution to a multiobjective optimization problem are summarized below.

**Constructing a single aggregate objective function (AOF)**

This is an intuitive approach to solving the multi-objective problem. The basic idea is to combine all of the objectives into a single objective function, called the AOF, such as the well-known weighted linear sum of the objectives. This objective function is optimized subject to technological constraints specifying how much of one objective must be sacrificed, from any given starting point, in order to gain a certain amount regarding the other objective. These technological constraints frequently come in the form \( f(x_1, x_2) \geq 0 \) for some function \( f \), where \( x_1 \) and \( x_2 \) are the objectives (e.g., strength and lightness of a product).

Often the aggregate objective function is not linear in the objectives, but rather is non-linear, expressing increasing marginal dissatisfaction with greater incremental sacrifices in the value of either objective. Furthermore, sometimes the aggregate objective function is additively separable, so that it is expressed as a weighted average of a non-linear function of one objective and a non-linear function of another objective. Then the optimal solution obtained will depend on the relative values of the weights specified. For example, if one is trying to maximize the strength of a machine component and minimize the production cost, and if a higher weight is specified for the cost objective compared to the strength, the solution will be one that favors lower cost over higher strength.

The weighted sum method, like any method of selecting a single solution as preferable to all others, is essentially subjective, in that a decision manager needs to supply the weights. Moreover, this approach may prove difficult to implement if the Pareto frontier is not globally convex and/or the objective function to be minimized is not globally concave.

The objective way of characterizing multi-objective problems, by identifying multiple Pareto optimal candidate solutions, requires a Pareto-compliant ranking method, favoring non-dominated solutions, as seen in current
multi-objective evolutionary approaches such as NSGA-II \cite{4} and SPEA2. Here, no weight is required and thus no \textit{a priori} information on the decision-maker's preferences is needed.\cite{5} However, to decide upon one of the Pareto-efficient options as the one to adopt requires information about the decision-maker's preferences. Thus the objective characterization of the problem is simply the first stage in a two-stage analysis, consisting of (1) identifying the non-dominated possibilities, and (2) choosing among them.

\textbf{The NBI, NC, SPO and DSD methods}

The Normal Boundary Intersection (NBI)\cite{6}\cite{7}, Normal Constraint (NC)\cite{8}\cite{9}, Successive Pareto Optimization (SPO)\cite{10}, and Directed Search Domain (DSD)\cite{11} methods solve the multi-objective optimization problem by constructing several AOFs. The solution of each AOF yields a Pareto point, whether locally or globally.

The NC and DSD methods suggest two different filtering procedures to remove locally Pareto points. The AOFs are constructed with the target of obtaining evenly distributed Pareto points that give a good impression (approximation) of the real set of Pareto points.

The DSD, NC and SPO methods generate solutions that represent some peripheral regions of the set of Pareto points for more than two objectives that are known to be not represented by the solutions generated with the NBI method.

According to Erfani and Utyuzhnikov, the DSD method works reasonably more efficiently than its NC and NBI counterparts on some difficult test cases in the literature.\cite{11}

\textbf{Evolutionary algorithms}

Evolutionary algorithms are popular approaches to solving multiobjective optimization. Currently most evolutionary optimizers apply Pareto-based ranking schemes. Genetic algorithms such as the Non-dominated Sorting Genetic Algorithm-II (NSGA-II) and Strength Pareto Evolutionary Algorithm 2 (SPEA-2) have become standard approaches, although some schemes based on particle swarm optimization and simulated annealing\cite{12} are significant. The main advantage of evolutionary algorithms, when applied to solve multi-objective optimization problems, is the fact that they typically optimize sets of solutions, allowing computation of an approximation of the entire Pareto front in a single algorithm run. The main disadvantage of evolutionary algorithms is the much lower speed.

\textbf{Other methods}

- Multiobjective Optimization using Evolutionary Algorithms (MOEA)\cite{5}\cite{13}\cite{14}
- PGEN (Pareto surface generation for convex multiobjective instances)\cite{15}
- IOSO (Indirect Optimization on the basis of Self-Organization)
- SMS-EMOA (S-metric selection evolutionary multiobjective algorithm)\cite{16}
- Reactive Search Optimization (using machine learning for adapting strategies and objectives)\cite{17}\cite{18}, implemented in LIONsolver
- Benson's algorithm for linear vector optimization problems

\textbf{Applications}

\textbf{Economics}

In economics, the study of resource allocation under scarcity, many problems involve multiple objectives along with constraints on what combinations of those objectives are attainable.

For example, a consumer's demands for various goods are determined by the process of maximization of the utility 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
Multi-objective optimization

in conflict with each other according to this constraint. 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.

Another example involves the production possibilities frontier, which specifies what combinations of various types of goods can be produced by a society with certain amounts of various resources. The frontier specifies the trade-offs that the society is faced with — if the society is fully utilizing its resources, more of one good can be produced only at the expense of producing less of another good. A society must then use some process to choose among the possibilities on the frontier.

Macroeconomic policy-making is a context requiring multi-objective optimization. Typically a central bank must choose a stance for monetary policy that balances competing objectives — low inflation, low unemployment, low balance of trade deficit, etc. To do this, the central bank uses a model of the economy that quantitatively describes the various causal linkages in the economy; it simulates the model repeatedly under various possible stances of monetary policy, in order to obtain a menu of possible predicted outcomes for the various variables of interest. Then in principle it can use an aggregate objective function to rate the alternative sets of predicted outcomes, although in practice central banks use a non-quantitative, judgement-based, process for ranking the alternatives and making the policy choice.

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, 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 moment) of portfolio return is called a two-moment decision model.

Linear programming applications

In linear programming problems, a linear objective function is optimized subject to linear constraints. Typically multiple variables of concern appear in the objective function. A vast body of research has been devoted to methods of solving these problems. Because the efficient set, the set of combinations of values of the various variables of interest having the feature that none of the variables can be given a better value without hurting the value of another variable, is piecewise linear and not continuously differentiable, the problem is not dealt with by first specifying all the points on the Pareto-efficient set; instead, solution procedures utilize the aggregate objective function right from the start.

Many practical problems in operations research can be expressed as linear programming problems. Certain special cases of linear programming, such as network flow problems and multi-commodity flow problems are considered important enough to have generated much research on specialized algorithms for their solution. Linear programming is heavily used in microeconomics and company management, for dealing with such issues as planning, production, transportation, technology, and so forth.

Optimal control applications

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, 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
Multi-objective optimization

often such problems are subject to linear equality constraints that prevent all objectives from being simultaneously perfectly met, especially when the number of controllable variables is less than the number of objectives and when the presence of random shocks generates uncertainty. commonly a multi-objective quadratic objective function is used, with the cost associated with an objective rising quadratically with the distance of the objective from its ideal value. since these problems typically involve adjusting the controlled variables at various points in time and/or evaluating the objectives at various points in time, intertemporal optimization techniques are employed.

References


External links

- A tutorial on multiobjective optimization (http://www.calresco.org/lucas/pmo.htm)
- Evolutionary Multiobjective Optimization (http://demonstrations.wolfram.com/EvolutionaryMultiobjectiveOptimization/), The Wolfram Demonstrations Project
Pareto efficiency

Pareto efficiency, or Pareto optimality, is a concept in economics with applications in engineering and social sciences. The term is named after Vilfredo Pareto (1848–1923), an Italian economist who used the concept in his studies of economic efficiency and income distribution.

In a Pareto efficient economic system no allocation of given goods can be made without making at least one individual worse off. Given an initial allocation of goods among a set of individuals, a change to a different allocation that makes at least one individual better off without making any other individual worse off is called a Pareto improvement. An allocation is defined as "Pareto efficient" or "Pareto optimal" when no further Pareto improvements can be made.

Pareto efficiency is a minimal notion of efficiency and does not necessarily result in a socially desirable distribution of resources: it makes no statement about equality, or the overall well-being of a society.[1][2]

Pareto efficiency in short

An economic system that is not Pareto efficient implies that a certain change in allocation of goods (for example) may result in some individuals being made "better off" with no individual being made worse off, and therefore can be made more Pareto efficient through a Pareto improvement. Here "better off" is often interpreted as "put in a preferred position." It is commonly accepted that outcomes that are not Pareto efficient are to be avoided, and therefore Pareto efficiency is an important criterion for evaluating economic systems and public policies.

If economic allocation in any system is not Pareto efficient, there is potential for a Pareto improvement—an increase in Pareto efficiency: through reallocation, improvements to at least one participant's well-being can be made better without reducing any other participant's well-being.

In the real world ensuring that nobody is disadvantaged by a change aimed at improving economic efficiency may require compensation of one or more parties. For instance, if a change in economic policy dictates that a legally protected monopoly ceases to exist and that market subsequently becomes competitive and more efficient, the monopolist will be made worse off. However, the loss to the monopolist will be more than offset by the gain in efficiency. This means the monopolist can be compensated for its loss while still leaving an efficiency
Pareto efficiency

In real-world practice, the compensation principle often appealed to is hypothetical. That is, for the alleged Pareto improvement (say from public regulation of the monopolist or removal of tariffs) some losers are not (fully) compensated. The change thus results in distribution effects in addition to any Pareto improvement that might have taken place. The theory of hypothetical compensation is part of Kaldor–Hicks efficiency, also called Potential Pareto Criterion.\[^3\] Hicks-Kaldor compensation is what turns the utilitarian rule for the maximization of a function of all individual utilities postulated by Samuelson as a solution to the optimal public goods problem, into a rule that mimics Pareto efficiency. This is how Pareto-efficiency finds itself at the heart of modern Public Choice theory where under certain conditions Black's median voter opts for a Hick-Kaldor compensated Pareto efficient level of public goods.\[^4\]

Under certain idealized conditions, it can be shown that a system of free markets will lead to a Pareto efficient outcome. This is called the first welfare theorem. It was first demonstrated mathematically by economists Kenneth Arrow and Gérard Debreu. However, the result does not rigorously establish welfare results for real economies because of the restrictive assumptions necessary for the proof (markets exist for all possible goods, all markets are in full equilibrium, markets are perfectly competitive, transaction costs are negligible, there must be no externalities, and market participants must have perfect information). Moreover, it has since been demonstrated mathematically that, in the absence of perfect information or complete markets, outcomes will generically be Pareto inefficient (the Greenwald–Stiglitz theorem).\[^5\]

A competitive equilibrium may not be Pareto Optimal because of externalities, tax distortion, or use of monopoly power. A negative externality causes the firm to overproduce relative to Pareto efficiency, while a positive externality causes the firm to underproduce. Tax distortions cause a wedge between the marginal rate of substitution and marginal product of labour. Monopoly power occurs when firms may not be price-takers. If the firm is large relative to market size, it can use its monopoly power to restrict output, raise prices, and increase profits.\[^6\]

### Pareto improvements and microeconomic theory

Note that microeconomic analysis does not assume additive utility nor does it assume any interpersonal utility tradeoffs. To engage in interpersonal utility tradeoffs leads to greater good problems faced by earlier utilitarians. It also creates a question as to how weights are assigned and who assigns them, as well as questions regarding how to compare pleasure or pain across individuals.

Efficiency – in all of standard microeconomics – therefore refers to the absence of possible Pareto improvements. It does not in any way opine on the fairness of the allocation (in the sense of distributive justice or equity). An ‘efficient’ equilibrium could be one where one player has all the goods and other players have none (in an extreme example).
Weak and strong Pareto optimum

A "weak Pareto optimum" (WPO) nominally satisfies the same standard of not being Pareto-inferior to any other allocation, but for the purposes of weak Pareto optimization, an alternative allocation is considered to be a Pareto improvement only if the alternative allocation is strictly preferred by all individuals. In other words, when an allocation is WPO there are no possible alternative allocations whose realization would cause every individual to gain.

Weak Pareto-optimality is "weaker" than strong Pareto-optimality in the sense that the conditions for WPO status are "weaker" than those for SPO status: any allocation that can be considered an SPO will also qualify as a WPO, but a WPO allocation won't necessarily qualify as an SPO.

Under any form of Pareto-optimality, for an alternative allocation to be Pareto-superior to an allocation being tested—and, therefore, for the feasibility of an alternative allocation to serve as proof that the tested allocation is not an optimal one—the feasibility of the alternative allocation must show that the tested allocation fails to satisfy at least one of the requirements for SPO status. One may apply the same metaphor to describe the set of requirements for WPO status as being "weaker" than the set of requirements for SPO status. (Indeed, because the SPO set entirely encompasses the WPO set, with respect to any property the requirements for SPO status are of strength equal to or greater than the strength of the requirements for WPO status. Therefore, the requirements for WPO status are not merely weaker on balance or weaker according to the odds; rather, one may describe them more specifically and quite fittingly as "Pareto-weaker.")

- Note that when one considers the requirements for an alternative allocation's superiority according to one definition against the requirements for its superiority according to the other, the comparison between the requirements of the respective definitions is the opposite of the comparison between the requirements for optimality: To demonstrate the WPO-inferiority of an allocation being tested, an alternative allocation must falsify at least one of the particular conditions in the WPO subset, rather than merely falsify at least one of either these conditions or the other SPO conditions. Therefore, the requirements for weak Pareto-superiority of an alternative allocation are harder to satisfy (in other words, "stronger") than are the requirements for strong Pareto-superiority of an alternative allocation.

- It further follows that every SPO is a WPO (but not every WPO is an SPO): Whereas the WPO description applies to any allocation from which every feasible departure results in the NON-IMPROVEMENT of at least one individual, the SPO description applies to only those allocations that meet both the WPO requirement and the more specific ("stronger") requirement that at least one non-improving individual exhibit a specific type of non-improvement, namely doing worse.

- The "strong" and "weak" descriptions of optimality continue to hold true when one construes the terms in the context set by the field of semantics: If one describes an allocation as being a WPO, one makes a "weaker" statement than one would make by describing it as an SPO: If the statements "Allocation X is a WPO" and "Allocation X is a SPO" are both true, then the former statement is less controversial than the latter in that to defend the latter, one must prove everything to defend the former "and then some." By the same token, however, the former statement is less informative or contentful in that it "says less" about the allocation; that is, the former statement contains, implies, and (when stated) asserts fewer constituent propositions about the allocation.
Formal representation

Formally, a (strong/weak) Pareto optimum is a maximal element for the partial order relation of Pareto improvement/strict Pareto improvement: it is an allocation such that no other allocation is "better" in the sense of the order relation.

Pareto frontier

Given a set of choices and a way of valuing them, the Pareto frontier or Pareto set or Pareto front is the set of choices that are Pareto efficient. The Pareto frontier is particularly useful in engineering: by restricting attention to the set of choices that are Pareto-efficient, a designer can make tradeoffs within this set, rather than considering the full range of every parameter.

The Pareto frontier is defined formally as follows.

Consider a design space with $n$ real parameters, and for each design space point there are $m$ different criteria by which to judge that point. Let $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ be the function which assigns, to each design space point $x$, a criteria space point $f(x)$. This represents the way of valuing the designs. Now, it may be that some designs are infeasible; so let $X$ be a set of feasible designs in $\mathbb{R}^n$, which must be a compact set. Then the set which represents the feasible criterion points is $f(X)$, the image of the set $X$ under the action of $f$. Call this image $Y$.

Now construct the Pareto frontier as a subset of $Y$, the feasible criterion points. It can be assumed that the preferable values of each criterion parameter are the lesser ones, thus minimizing each dimension of the criterion vector. Then compare criterion vectors as follows: One criterion vector $y$ strictly dominates (or "is preferred to") a vector $y^*$ if each parameter of $y$ is not strictly greater than the corresponding parameter of $y^*$ and at least one parameter is strictly less: that is, $y_i \leq y^*_i$ for each $i$ and $y_i < y^*_i$ for some $i$. This is written as $y \prec y^*$ to mean that $y$ strictly dominates $y^*$. Then the Pareto frontier is the set of points from $Y$ that are not strictly dominated by another point in $Y$.

Formally, this defines a partial order on $Y$, namely the product order on $\mathbb{R}^m$ (more precisely, the induced order on $Y$ as a subset of $\mathbb{R}^m$), and the Pareto frontier is the set of maximal elements with respect to this order.

Algorithms for computing the Pareto frontier of a finite set of alternatives have been studied in computer science, being sometimes referred to as the maximum vector problem or the skyline query.\[7\][8]
Relationship to marginal rate of substitution

At a Pareto efficient allocation (on the Pareto frontier), the marginal rate of substitution is the same for all consumers. A formal statement can be derived by considering a system with \( m \) consumers and \( n \) goods, and a utility function of each consumer as \( z_i = f^i(x^i) \) where \( x^i = (x^i_1, x^i_2, \ldots, x^i_n) \) is the vector of goods, both for all \( i \).

The supply constraint is written \( \sum_{i=1}^{m} x^i_j = b^0_j \) for \( j = 1, \ldots, n \). To optimize this problem, the Lagrangian is used:

\[
L(x, \lambda, \Gamma) = f^1(x^1) + \sum_{i=2}^{m} \lambda_i(z^i_0 - f^i(x^i)) + \sum_{j=1}^{n} \Gamma_j(b^0_j - \sum_{i=1}^{m} x^i_j)
\]

where \( \lambda \) and \( \Gamma \) are Lagrange multipliers.

By taking the partial derivative of the Lagrangian with respect to consumer 1’s consumption of good \( j \), and then taking the partial derivative of the Lagrangian with respect to consumer \( i \)’s consumption of good \( j \), we have the following system of equations:

\[
\frac{\partial L}{\partial x^i_j} = f^i_j - \Gamma^0_j = 0 \quad \text{for} \quad j = 1, \ldots, n.
\]

\[
\frac{\partial L}{\partial x^i_j} = -\lambda_i f^i_j - \Gamma^0_j = 0 \quad \text{for} \quad i = 1, \ldots, m \quad \text{and} \quad j = 1, \ldots, n
\]

where \( f^i_j \) denotes consumer \( i \)’s marginal utility of consuming good \( j \) (the partial derivative of \( f^i \) with respect to \( x^i_j \)).

These equations combine to yield precisely the condition

\[
\frac{f^j_i}{f^j_k} = \frac{f^j_k}{f^j_s} \quad \text{for} \quad i, k = 1, \ldots, m \quad \text{and} \quad j, s = 1, \ldots, n.
\]

that requires that the marginal rate of substitution between each ordered pair of goods \((x^i_j, x^i_s)\) be equal across all consumers.

Notes


References


Stochastic programming

Stochastic programming is a framework for modeling optimization problems that involve uncertainty. Whereas deterministic optimization problems are formulated with known parameters, real world problems almost invariably include some unknown parameters. When the parameters are known only within certain bounds, one approach to tackling such problems is called robust optimization. Here the goal is to find a solution which is feasible for all such data and optimal in some sense. Stochastic programming models are similar in style but take advantage of the fact that probability distributions governing the data are known or can be estimated. The goal here is to find some policy that is feasible for all (or almost all) the possible data instances and maximizes the expectation of some function of the decisions and the random variables. More generally, such models are formulated, solved analytically or numerically, and analyzed in order to provide useful information to a decision-maker.[1]

As an example, consider two-stage linear programs. Here the decision maker takes some action in the first stage, after which a random event occurs affecting the outcome of the first-stage decision. A recourse decision can then be made in the second stage that compensates for any bad effects that might have been experienced as a result of the first-stage decision. The optimal policy from such a model is a single first-stage policy and a collection of recourse decisions (a decision rule) defining which second-stage action should be taken in response to each random outcome. Stochastic programming has applications in a broad range of areas ranging from finance to transportation to energy optimization.[2][3]

Biological Applications

Stochastic dynamic programming is frequently used to model animal behaviour in such fields as behavioural ecology.[4][5] Empirical tests of models of optimal foraging, life-history transitions such as fledging in birds and egg laying in parasitoid wasps have shown the value of this modelling technique in explaining the evolution of behavioural decision making. These models are typically many staged, rather than two-staged.

Economic Applications

Stochastic dynamic programming is a useful tool in understanding decision making under uncertainty. The accumulation of capital stock under uncertainty is one example, often it is used by resource economists to analyze bioeconomic problems[6] where the uncertainty enters in such as weather, etc.
Solvers

- FortSP - solver for stochastic programming problems

References


[3] Applications of stochastic programming are described at the following website, Stochastic Programming Community (http://stoprog.org).


Further reading


External links

- Stochastic Programming Community Home Page (http://stoprog.org)
Parallel metaheuristic

Parallel metaheuristic is a class of new advanced techniques that are able of reducing both the numerical effort and the run time of a metaheuristic. To this end, concepts and technologies from the field of parallelism in computer science are used to enhance and even completely modify the behavior of existing metaheuristics. Just as it exists a long list of metaheuristics like evolutionary algorithms, particle swarm, ant colony optimization, simulated annealing, etc. it also exists a large set of different techniques strongly or loosely based in these ones, whose behavior encompasses the multiple parallel execution of algorithm components that cooperate in some way to solve a problem on a given parallel hardware platform.

Background

In practice, optimization (and searching, and learning) problems are often NP-hard, complex, and time consuming. Two major approaches are traditionally used to tackle these problems: exact methods and metaheuristics. Exact methods allow to find exact solutions but are often impractical as they are extremely time-consuming for real-world problems (large dimension, hardly constrained, multimodal, time-varying, epistatic problems). Conversely, metaheuristics provide sub-optimal (sometimes optimal) solutions in a reasonable time. Thus, metaheuristics usually allow to meet the resolution delays imposed in the industrial field as well as they allow to study general problem classes instead that particular problem instances. In general, many of the best performing techniques in precision and effort to solve complex and real-world problems are metaheuristics. Their fields of application range from combinatorial optimization, bioinformatics, and telecommunications to economics, software engineering, etc. These fields are full of many tasks needing fast solutions of high quality. See [1] for more details on complex applications.

Metaheuristics fall in two categories: trajectory-based metaheuristics and population-based metaheuristics. The main difference of these two kind of methods relies in the number of tentative solutions used in each step of the (iterative) algorithm. A trajectory-based technique starts with a single initial solution and, at each step of the search, the current solution is replaced by another (often the best) solution found in its neighborhood. It is usual that trajectory-based metaheuristics allow to quickly find a locally optimal solution, and so they are called exploitation-oriented methods promoting intensification in the search space. On the other hand, population-based algorithms make use of a population of solutions. The initial population is in this case randomly generated (or created with a greedy algorithm), and then enhanced through an iterative process. At each generation of the process, the whole population (or a part of it) is replaced by newly generated individuals (often the best ones). These techniques are called exploration-oriented methods, since their main ability resides in the diversification in the search space.
Most basic metaheuristics are sequential. Although their utilization allows to significantly reduce the temporal complexity of the search process, this latter remains high for real-world problems arising in both academic and industrial domains. Therefore, parallelism comes as a natural way not to only reduce the search time, but also to improve the quality of the provided solutions.

For a comprehensive discussion on how parallelism can be mixed with metaheuristics see [2].

**Parallel trajectory-based metaheuristics**

Metaheuristics for solving optimization problems could be viewed as *walks through neighborhoods* tracing search trajectories through the solution domains of the problem at hands:

**Algorithm:** Sequential trajectory-based general pseudo-code

```plaintext
Generate(s(0)); // Initial solution

\[ t := 0; // Numerical step \]

\[ \textbf{while not} \ \text{Termination Criterion}(s(t)) \ \textbf{do} \]

\[ ...s'(t) := \text{SelectMove}(s(t)); // Exploration of the neighborhood \]

\[ \textbf{if} \ \text{AcceptMove}(s'(t)) \ \textbf{then} \]

\[ ...s(t) := \text{ApplyMove}(s'(t)); \]

\[ ...t := t+1; \]

\[ \textbf{endwhile} \]
```

Walks are performed by iterative procedures that allow moving from one solution to another one in the solution space (see the above algorithm). This kind of metaheuristics perform the moves in the neighborhood of the current solution, i.e., they have a perturbative nature. The walks start from a solution randomly generated or obtained from another optimization algorithm. At each iteration, the current solution is replaced by another one selected from the set of its neighboring candidates. The search process is stopped when a given condition is satisfied (a maximum number of generation, find a solution with a target quality, stuck for a given time, . . .).

A powerful way to achieve high computational efficiency with trajectory-based methods is the use of parallelism. Different parallel models have been proposed for trajectory-based metaheuristics, and three of them are commonly used in the literature: the parallel multi-start model, the parallel exploration and evaluation of the neighborhood (or parallel moves model), and the parallel evaluation of a single solution (or move acceleration model):

- **Parallel multi-start model:** It consists in simultaneously launching several trajectory-based methods for computing better and robust solutions. They may be heterogeneous or homogeneous, independent or cooperative, start from the same or different solution(s), and configured with the same or different parameters.

- **Parallel moves model:** It is a low-level master-slave model that does not alter the behavior of the heuristic. A sequential search would compute the same result but slower. At the beginning of each iteration, the master duplicates the current solution between distributed nodes. Each one separately manages their candidate/solution and the results are returned to the master.

- **Move acceleration model:** The quality of each move is evaluated in a parallel centralized way. That model is particularly interesting when the evaluation function can be itself parallelized as it is CPU time-consuming and/or I/O intensive. In that case, the function can be viewed as an aggregation of a certain number of partial functions that can be run in parallel.
Parallel metaheuristic

Population-based metaheuristics

Population-based metaheuristics are stochastic search techniques that have been successfully applied in many real and complex applications (epistatic, multimodal, multi-objective, and highly constrained problems). A population-based algorithm is an iterative technique that applies stochastic operators on a pool of individuals: the population (see the algorithm below). Every individual in the population is the encoded version of a tentative solution. An evaluation function associates a fitness value to every individual indicating its suitability to the problem. Iteratively, the probabilistic application of variation operators on selected individuals guides the population to tentative solutions of higher quality. The most well-known metaheuristic families based on the manipulation of a population of solutions are evolutionary algorithms (EAs), ant colony optimization (ACO), particle swarm optimization (PSO), scatter search (SS), differential evolution (DE), and estimation distribution algorithms (EDA).

Algorithm: Sequential population-based metaheuristic pseudo-code

Generate(P(0)); // Initial population

$ t := 0 $; // Numerical step

while not Termination Criterion(P(t)) do

...Evaluate(P(t)); // Evaluation of the population

...P′′(t) := Apply Variation Operators(P′(t)); // Generation of new solutions

...P(t + 1) := Replace(P(t), P′′(t)); // Building the next population

...$ t := t + 1 $;

endwhile

For non-trivial problems, executing the reproductive cycle of a simple population-based method on long individuals and/or large populations usually requires high computational resources. In general, evaluating a fitness function for every individual is frequently the most costly operation of this algorithm. Consequently, a variety of algorithmic issues are being studied to design efficient techniques. These issues usually consist of defining new operators, hybrid algorithms, parallel models, and so on.

Parallelism arises naturally when dealing with populations, since each of the individuals belonging to it is an independent unit (at least according to the Pittsburg style, although there are other approaches like the Michigan one which do not consider the individual as independent units). Indeed, the performance of population-based algorithms is often improved when running in parallel. Two parallelizing strategies are specially focused on population-based algorithms:

1. **Parallelization of computations**, in which the operations commonly applied to each of the individuals are performed in parallel, and

2. **Parallelization of population**, in which the population is split in different parts that can be simply exchanged or evolved separately, and then joined later.

In the beginning of the parallelization history of these algorithms, the well-known master-slave (also known as global parallelization or farming) method was used. In this approach, a central processor performs the selection operations while the associated slave processors (workers) run the variation operator and the evaluation of the fitness function. This algorithm has the same behavior as the sequential one, although its computational efficiency is improved, especially for time consuming objective functions. On the other hand, many researchers use a pool of processors to speed up the execution of a sequential algorithm, just because independent runs can be made more rapidly by using several processors than by using a single one. In this case, no interaction at all exists between the independent runs.

However, actually most parallel population-based techniques found in the literature utilize some kind of spatial disposition for the individuals, and then parallelize the resulting chunks in a pool of processors. Among the most widely known types of structured metaheuristics, the distributed (or coarse grain) and cellular (or fine grain) algorithms are very popular optimization procedures.
In the case of distributed ones, the population is partitioned in a set of subpopulations (islands) in which isolated serial algorithms are executed. Sparse exchanges of individuals are performed among these islands with the goal of introducing some diversity into the subpopulations, thus preventing search of getting stuck in local optima. In order to design a distributed metaheuristic, we must take several decisions. Among them, a chief decision is to determine the migration policy: topology (logical links between the islands), migration rate (number of individuals that undergo migration in every exchange), migration frequency (number of steps in every subpopulation between two successive exchanges), and the selection/replacement of the migrants.

In the case of a cellular method, the concept of neighborhood is introduced, so that an individual may only interact with its nearby neighbors in the breeding loop. The overlapped small neighborhood in the algorithm helps in exploring the search space because a slow diffusion of solutions through the population provides a kind of exploration, while exploitation takes place inside each neighborhood. See [3] for more information on cellular Genetic Algorithms and related models.

Also, hybrid models are being proposed in which a two-level approach of parallelization is undertaken. In general, the higher level for parallelization is a coarse-grained implementation and the basic island performs a a cellular, a master-slave method or even another distributed one.

See Also

- Cellular Evolutionary Algorithms
- Enrique Alba

References


- MALLBA (http://neo.lcc.uma.es/software/mallba/index.php)
- JGDS (http://neo.lcc.uma.es/software/jgds/index.php)
- DEME (http://neo.lcc.uma.es/software/deme/index.php)
- xxGA (http://neo.lcc.uma.es/software/xxga/index.php)
- Paradiseo
External links

- THE Page on Parallel Metaheuristics (http://mallba10.lcc.uma.es/PM/index.php/Parallel_Metaheuristics)
- The NEO group at the University of Málaga, Spain (http://neo.lcc.uma.es)

There ain't no such thing as a free lunch

"There ain't no such thing as a free lunch" (alternatively, "There's no such thing as a free lunch" or other variants) is a popular adage communicating the idea that it is impossible to get something for nothing. The acronyms TANSTAAFL and TINSTAAFL are also used. Uses of the phrase dating back to the 1930s and 1940s have been found, but the phrase's first appearance is unknown. The "free lunch" in the saying refers to the nineteenth century practice in American bars of offering a "free lunch" as a way to entice drinking customers. The phrase and the acronym are central to Robert Heinlein's 1966 libertarian science fiction novel The Moon is a Harsh Mistress, which popularized it. The free-market economist Milton Friedman also popularized the phrase by using it as the title of a 1975 book, and it often appears in economics textbooks; Campbell McConnell writes that the idea is "at the core of economics".

History and usage

"Free lunch"

The "free lunch" referred to in the acronym relates back to the once-common tradition of saloons in the United States providing a "free" lunch to patrons who had purchased at least one drink. All the foods on offer were high in salt (e.g. ham, cheese and salted crackers) so those who ate them ended up buying a lot of beer. Rudyard Kipling, writing in 1891, noted how he came upon a bar room full of bad Salon pictures, in which men with hats on the backs of their heads were wolfing food from a counter.

"It was the institution of the 'free lunch' I had struck. You paid for a drink and got as much as you wanted to eat. For something less than a rupee a day a man can feed himself sumptuously in San Francisco, even though he be a bankrupt. Remember this if ever you are stranded in these parts."

TANSTAAFL, on the other hand, indicates an acknowledgment that in reality a person or a society cannot get "something for nothing". Even if something appears to be free, there is always a cost to the person or to society as a whole even though that cost may be hidden or distributed. For example, as Heinlein has one of his characters point out, a bar offering a free lunch will likely charge more for its drinks.

Early uses

According to Robert Caro, Fiorello La Guardia, on becoming mayor of New York in 1934, said "È finita la cucagna!", meaning "No more free lunch"; in this context "free lunch" refers to graft and corruption. The earliest known occurrence of the full phrase, in the form "There ain't no such thing as free lunch", appears as the punchline of a joke related in an article in the El Paso Herald-Post of June 27, 1938, entitled "Economics in Eight Words".

In 1945 "There ain't no such thing as a free lunch" appeared in the Columbia Law Review, and "there is no free lunch" appeared in a 1942 article in the Oelwein Daily Register (in a quote attributed to economist Harley L. Lutz) and in a 1947 column by economist Merryle S. Rukeyser. In 1949 the phrase appeared in an article by Walter Morrow in the San Francisco News (published on 1 June) and in Pierre Dos Utt's monograph, "TANSTAAFL: a plan for a new economic world order", which describes an oligarchic political system based on his conclusions from "no free lunch" principles.

The 1938 and 1949 sources use the phrase in relating a fable about a king (Nebuchadrezzar in Dos Utt's retelling) seeking advice from his economic advisors. Morrow's retelling, which claims to derive from an earlier editorial
reported to be non-existent,[11] but closely follows the story as related in the earlier article in the El Paso Herald-Post, differs from Dos Utt's in that the ruler asks for ever-simplified advice following their original "eighty-seven volumes of six hundred pages" as opposed to a simple failure to agree on "any major remedy". The last surviving economist advises that "There ain't no such thing as a free lunch".

In 1950, a New York Times columnist ascribed the phrase to economist (and Army General) Leonard P. Ayres of the Cleveland Trust Company. "It seems that shortly before the General's death [in 1946]... a group of reporters approached the general with the request that perhaps he might give them one of several immutable economic truisms that he gathered from long years of economic study... 'It is an immutable economic fact,' said the general, 'that there is no such thing as a free lunch.'"[12]

MEANINGS

TANSTAAFL demonstrates opportunity cost. Greg Mankiw described the concept as: "To get one thing that we like, we usually have to give up another thing that we like. Making decisions requires trading off one goal against another."[13] The idea that there is no free lunch at the societal level applies only when all resources are being used completely and appropriately, i.e., when economic efficiency prevails. If not, a 'free lunch' can be had through a more efficient utilisation of resources. If one individual or group gets something at no cost, somebody else ends up paying for it. If there appears to be no direct cost to any single individual, there is a social cost. Similarly, someone can benefit for "free" from an externality or from a public good, but someone has to pay the cost of producing these benefits.

In the sciences, TANSTAAFL means that the universe as a whole is ultimately a closed system—there is no magic source of matter, energy, light, or indeed lunch, that does not draw resources from something else, and will not eventually be exhausted. Therefore the TANSTAAFL argument may also be applied to natural physical processes in a closed system (either the universe as a whole, or any system that does not receive energy or matter from outside). (See Second law of thermodynamics.) The bio-ecologist Barry Commoner used this concept as the last of his famous "Four Laws of Ecology".

In mathematical finance, the term is also used as an informal synonym for the principle of no-arbitrage. This principle states that a combination of securities that has the same cash flows as another security must have the same net price in equilibrium.

TANSTAAFL is sometimes used as a response to claims of the virtues of free software. Supporters of free software often counter that the use of the term "free" in this context is primarily a reference to a lack of constraint ("libre") rather than a lack of cost ("gratis"). Richard Stallman has described it as "free as in speech not as in beer".

The prefix "TANSTAA--" is used in numerous other contexts as well to denote some immutable property of the system being discussed. For example, "TANSTAAANFS" is used by Electrical Engineering professors to stand for "There Ain't No Such Thing As A Noise Free System".
References

  • American Notes by Rudyard Kipling (http://www.gutenberg.org/etext/977) at Project Gutenberg

Fitness landscape

In evolutionary biology, fitness landscapes or adaptive landscapes are used to visualize the relationship between genotypes (or phenotypes) and reproductive success. It is assumed that every genotype has a well-defined replication rate (often referred to as fitness). This fitness is the "height" of the landscape. Genotypes which are very similar are said to be "close" to each other, while those that are very different are "far" from each other.

The two concepts of height and distance are sufficient to form the concept of a "landscape". The set of all possible genotypes, their degree of similarity, and their related fitness values is then called a fitness landscape. The idea of a fitness landscape helps explain flawed forms in evolution, including exploits and glitches in animals like their reactions to supernormal stimuli.

In evolutionary optimization problems, fitness landscapes are evaluations of a fitness function for all candidate solutions (see below). The idea of studying evolution by visualizing the distribution of fitness values as a kind of landscape was first introduced by Sewall Wright in 1932.[11]

Fitness landscapes in biology

Fitness landscapes are often conceived of as ranges of mountains. There exist local peaks (points from which all paths are downhill, i.e. to lower fitness) and valleys (regions from which most paths lead uphill). A fitness landscape with many local peaks surrounded by deep valleys is called rugged. If all genotypes have the same replication rate, on the other hand, a fitness landscape is said to be flat. The shapes of fitness landscapes are also closely related to epistasis, as demonstrated by Stuart Kauffman's NK-Landscape model.

An evolving population typically climbs uphill in the fitness landscape, by a series of small genetic changes, until a local optimum is reached (Fig. 1). There it remains, unless a rare mutation opens a path to a new, higher fitness peak. Note, however, that at high mutation rates this picture is somewhat simplistic. A population may not be able to climb a very steep peak if the mutation rate is too high, or it may drift away from a peak it had already found;
consequently, reducing the fitness of the system. The process of drifting away from a peak is often referred to as Muller's ratchet.

The apparent lack of wheeled animals is an example of a fitness peak which is presently inaccessible due to a surrounding valley.

In general, the higher the connectivity the more rugged the system becomes. Thus, a simply connected system only has one peak and if part of the system is changed then there will be little, if any, effect on any other part of the system. A high connectivity implies that the variables or sub-systems interact far more and the system may have to settle for a level of 'fitness' lower than it might be able to attain. The system would then have to change its approach to overcoming whatever problems that confront it, thus, changing the 'terrain' and enabling it to continue.

**Fitness landscapes in evolutionary optimization**

Apart from the field of evolutionary biology, the concept of a fitness landscape has also gained importance in evolutionary optimization methods such as genetic algorithms or evolutionary strategies. In evolutionary optimization, one tries to solve real-world problems (e.g., engineering or logistics problems) by imitating the dynamics of biological evolution. For example, a delivery truck with a number of destination addresses can take a large variety of different routes, but only very few will result in a short driving time.

In order to use evolutionary optimization, one has to define for every possible solution $s$ to the problem of interest (i.e., every possible route in the case of the delivery truck) how 'good' it is. This is done by introducing a scalar-valued function $f(s)$ (scalar valued means that $f(s)$ is a simple number, such as 0.3, while $s$ can be a more complicated object, for example a list of destination addresses in the case of the delivery truck), which is called the fitness function or fitness landscape.

A high $f(s)$ implies that $s$ is a good solution. In the case of the delivery truck, $f(s)$ could be the number of deliveries per hour on route $s$. The best, or at least a very good, solution is then found in the following way: initially, a population of random solutions is created. Then, the solutions are mutated and selected for those with higher fitness, until a satisfying solution has been found.

Evolutionary optimization techniques are particularly useful in situations in which it is easy to determine the quality of a single solution, but hard to go through all possible solutions one by one (it is easy to determine the driving time for a particular route of the delivery truck, but it is almost impossible to check all possible routes once the number of destinations grows to more than a handful).

The concept of a scalar valued fitness function $f(s)$ also corresponds to the concept of a potential or energy function in physics. The two concepts only differ in that physicists traditionally think in terms of minimizing the potential function, while biologists prefer the notion that fitness is being maximized. Therefore, taking the inverse of a potential function turns it into a fitness function, and vice versa.

![Figure 1: Sketch of a fitness landscape. The arrows indicate the preferred flow of a population on the landscape, and the points A and C are local optima. The red ball indicates a population that moves from a very low fitness value to the top of a peak.](image)
References


Further reading


Genetic algorithm

In the computer science field of artificial intelligence, a genetic algorithm (GA) is a search heuristic that mimics the process of natural evolution. This heuristic is routinely used to generate useful solutions to optimization and search problems. Genetic algorithms belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover.

Methodology

In a genetic algorithm, a population of strings (called chromosomes or the genotype of the genome), which encode candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem, evolves toward better solutions. Traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible. The evolution usually starts from a population of randomly generated individuals and happens in generations. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are stochastically selected from the current population (based on their fitness), and modified (recombined and possibly randomly mutated) to form a new population. The new population is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. If the algorithm has terminated due to a maximum number of generations, a satisfactory solution may or may not have been reached.

Genetic algorithms find application in bioinformatics, phylogenetics, computational science, engineering, economics, chemistry, manufacturing, mathematics, physics and other fields.

A typical genetic algorithm requires:
1. a genetic representation of the solution domain,
2. a fitness function to evaluate the solution domain.

A standard representation of the solution is as an array of bits. Arrays of other types and structures can be used in essentially the same way. The main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size, which facilitates simple crossover operations. Variable length
representations may also be used, but crossover implementation is more complex in this case. Tree-like representations are explored in genetic programming and graph-form representations are explored in evolutionary programming.

The fitness function is defined over the genetic representation and measures the quality of the represented solution. The fitness function is always problem dependent. For instance, in the knapsack problem one wants to maximize the total value of objects that can be put in a knapsack of some fixed capacity. A representation of a solution might be an array of bits, where each bit represents a different object, and the value of the bit (0 or 1) represents whether or not the object is in the knapsack. Not every such representation is valid, as the size of objects may exceed the capacity of the knapsack. The fitness of the solution is the sum of values of all objects in the knapsack if the representation is valid, or 0 otherwise. In some problems, it is hard or even impossible to define the fitness expression; in these cases, interactive genetic algorithms are used.

Once the genetic representation and the fitness function are defined, a GA proceeds to initialize a population of solutions (usually randomly) and then to improve it through repetitive application of the mutation, crossover, inversion and selection operators.

**Initialization**

Initially many individual solutions are (usually) randomly generated to form an initial population. The population size depends on the nature of the problem, but typically contains several hundreds or thousands of possible solutions. Traditionally, the population is generated randomly, allowing the entire range of possible solutions (the search space). Occasionally, the solutions may be "seeded" in areas where optimal solutions are likely to be found.

**Selection**

During each successive generation, a proportion of the existing population is selected to breed a new generation. Individual solutions are selected through a fitness-based process, where fitter solutions (as measured by a fitness function) are typically more likely to be selected. Certain selection methods rate the fitness of each solution and preferentially select the best solutions. Other methods rate only a random sample of the population, as the latter process may be very time-consuming.

**Reproduction**

The next step is to generate a second generation population of solutions from those selected through genetic operators: crossover (also called recombination), and/or mutation.

For each new solution to be produced, a pair of "parent" solutions is selected for breeding from the pool selected previously. By producing a "child" solution using the above methods of crossover and mutation, a new solution is created which typically shares many of the characteristics of its "parents". New parents are selected for each new child, and the process continues until a new population of solutions of appropriate size is generated. Although reproduction methods that are based on the use of two parents are more "biology inspired", some research\[1][2] suggests that more than two "parents" generate higher quality chromosomes.

These processes ultimately result in the next generation population of chromosomes that is different from the initial generation. Generally the average fitness will have increased by this procedure for the population, since only the best organisms from the first generation are selected for breeding, along with a small proportion of less fit solutions, for reasons already mentioned above.

Although Crossover and Mutation are known as the main genetic operators, it is possible to use other operators such as regrouping, colonization-extinction, or migration in genetic algorithms.\[3]
Termination

This generational process is repeated until a termination condition has been reached. Common terminating conditions are:

- A solution is found that satisfies minimum criteria
- Fixed number of generations reached
- Allocated budget (computation time/money) reached
- The highest ranking solution's fitness is reaching or has reached a plateau such that successive iterations no longer produce better results
- Manual inspection
- Combinations of the above

Simple generational genetic algorithm procedure:

1. Choose the initial population of individuals
2. Evaluate the fitness of each individual in that population
3. Repeat on this generation until termination (time limit, sufficient fitness achieved, etc.):
   1. Select the best-fit individuals for reproduction
   2. Breed new individuals through crossover and mutation operations to give birth to offspring
   3. Evaluate the individual fitness of new individuals
   4. Replace least-fit population with new individuals

The building block hypothesis

Genetic algorithms are simple to implement, but their behavior is difficult to understand. In particular it is difficult to understand why these algorithms frequently succeed at generating solutions of high fitness when applied to practical problems. The building block hypothesis (BBH) consists of:

1. A description of a heuristic that performs adaptation by identifying and recombining "building blocks", i.e. low order, low defining-length schemata with above average fitness.
2. A hypothesis that a genetic algorithm performs adaptation by implicitly and efficiently implementing this heuristic.

Goldberg describes the heuristic as follows:

"Short, low order, and highly fit schemata are sampled, recombined [crossed over], and resampled to form strings of potentially higher fitness. In a way, by working with these particular schemata [the building blocks], we have reduced the complexity of our problem; instead of building high-performance strings by trying every conceivable combination, we construct better and better strings from the best partial solutions of past samplings.

"Because highly fit schemata of low defining length and low order play such an important role in the action of genetic algorithms, we have already given them a special name: building blocks. Just as a child creates magnificent fortresses through the arrangement of simple blocks of wood, so does a genetic algorithm seek near optimal performance through the juxtaposition of short, low-order, high-performance schemata, or building blocks."[4]
Observations

There are several general observations about the generation of solutions specifically via a genetic algorithm:

- Selection is clearly an important genetic operator, but opinion is divided over the importance of crossover versus mutation. Some argue that crossover is the most important, while mutation is only necessary to ensure that potential solutions are not lost. Others argue that crossover in a largely uniform population only serves to propagate innovations originally found by mutation, and in a non-uniform population crossover is nearly always equivalent to a very large mutation (which is likely to be catastrophic). There are many references in Fogel (2006) that support the importance of mutation-based search.

- As with all current machine learning problems it is worth tuning the parameters such as mutation probability, crossover probability and population size to find reasonable settings for the problem class being worked on. A very small mutation rate may lead to genetic drift (which is non-ergodic in nature). A recombination rate that is too high may lead to premature convergence of the genetic algorithm. A mutation rate that is too high may lead to loss of good solutions unless there is elitist selection. There are theoretical but not yet practical upper and lower bounds for these parameters that can help guide selection.

- Often, GAs can rapidly locate good solutions, even for large search spaces. The same is of course also true for evolution strategies and evolutionary programming.

Criticisms

There are several criticisms of the use of a genetic algorithm compared to alternative optimization algorithms:

- Repeated fitness function evaluation for complex problems is often the most prohibitive and limiting segment of artificial evolutionary algorithms. Finding the optimal solution to complex high dimensional, multimodal problems often requires very expensive fitness function evaluations. In real world problems such as structural optimization problems, one single function evaluation may require several hours to several days of complete simulation. Typical optimization methods cannot deal with such types of problem. In this case, it may be necessary to forgo an exact evaluation and use an approximated fitness that is computationally efficient. It is apparent that amalgamation of approximate models may be one of the most promising approaches to convincingly use GA to solve complex real life problems.

- Genetic algorithms do not scale well with complexity. That is, where the number of elements which are exposed to mutation is large there is often an exponential increase in search space size. This makes it extremely difficult to use the technique on problems such as designing an engine, a house or plane. In order to make such problems tractable to evolutionary search, they must be broken down into the simplest representation possible. Hence we typically see evolutionary algorithms encoding designs for fan blades instead of engines, building shapes instead of detailed construction plans, aerofoils instead of whole aircraft designs. The second problem of complexity is the issue of how to protect parts that have evolved to represent good solutions from further destructive mutation, particularly when their fitness assessment requires them to combine well with other parts. It has been suggested by some in the community that a developmental approach to evolved solutions could overcome some of the issues of protection, but this remains an open research question.

- The "better" solution is only in comparison to other solutions. As a result, the stop criterion is not clear in every problem.

- In many problems, GAs may have a tendency to converge towards local optima or even arbitrary points rather than the global optimum of the problem. This means that it does not "know how" to sacrifice short-term fitness to gain longer-term fitness. The likelihood of this occurring depends on the shape of the fitness landscape: certain problems may provide an easy ascent towards a global optimum, others may make it easier for the function to find the local optima. This problem may be alleviated by using a different fitness function, increasing the rate of mutation, or by using selection techniques that maintain a diverse population of solutions, although the No Free
Lunch theorem\(^5\) proves that there is no general solution to this problem. A common technique to maintain diversity is to impose a "niche penalty", wherein, any group of individuals of sufficient similarity (niche radius) have a penalty added, which will reduce the representation of that group in subsequent generations, permitting other (less similar) individuals to be maintained in the population. This trick, however, may not be effective, depending on the landscape of the problem. Another possible technique would be to simply replace part of the population with randomly generated individuals, when most of the population is too similar to each other. Diversity is important in genetic algorithms (and genetic programming) because crossing over a homogeneous population does not yield new solutions. In evolution strategies and evolutionary programming, diversity is not essential because of a greater reliance on mutation.

- Operating on dynamic data sets is difficult, as genomes begin to converge early on towards solutions which may no longer be valid for later data. Several methods have been proposed to remedy this by increasing genetic diversity somehow and preventing early convergence, either by increasing the probability of mutation when the solution quality drops (called triggered hypermutation), or by occasionally introducing entirely new, randomly generated elements into the gene pool (called random immigrants). Again, evolution strategies and evolutionary programming can be implemented with a so-called "comma strategy" in which parents are not maintained and new parents are selected only from offspring. This can be more effective on dynamic problems.

- GAs cannot effectively solve problems in which the only fitness measure is a single right/wrong measure (like decision problems), as there is no way to converge on the solution (no hill to climb). In these cases, a random search may find a solution as quickly as a GA. However, if the situation allows the success/failure trial to be repeated giving (possibly) different results, then the ratio of successes to failures provides a suitable fitness measure.

- For specific optimization problems and problem instances, other optimization algorithms may find better solutions than genetic algorithms (given the same amount of computation time). Alternative and complementary algorithms include evolution strategies, evolutionary programming, simulated annealing, Gaussian adaptation, hill climbing, and swarm intelligence (e.g.: ant colony optimization, particle swarm optimization) and methods based on integer linear programming. The question of which, if any, problems are suited to genetic algorithms (in the sense that such algorithms are better than others) is open and controversial.

**Variants**

The simplest algorithm represents each chromosome as a bit string. Typically, numeric parameters can be represented by integers, though it is possible to use floating point representations. The floating point representation is natural to evolution strategies and evolutionary programming. The notion of real-valued genetic algorithms has been offered but is really a misnomer because it does not really represent the building block theory that was proposed by John Henry Holland in the 1970s. This theory is not without support though, based on theoretical and experimental results (see below). The basic algorithm performs crossover and mutation at the bit level. Other variants treat the chromosome as a list of numbers which are indexes into an instruction table, nodes in a linked list, hashes, objects, or any other imaginable data structure. Crossover and mutation are performed so as to respect data element boundaries. For most data types, specific variation operators can be designed. Different chromosomal data types seem to work better or worse for different specific problem domains.

When bit-string representations of integers are used, Gray coding is often employed. In this way, small changes in the integer can be readily effected through mutations or crossovers. This has been found to help prevent premature convergence at so called Hamming walls, in which too many simultaneous mutations (or crossover events) must occur in order to change the chromosome to a better solution.

Other approaches involve using arrays of real-valued numbers instead of bit strings to represent chromosomes. Theoretically, the smaller the alphabet, the better the performance, but paradoxically, good results have been obtained from using real-valued chromosomes.
A very successful (slight) variant of the general process of constructing a new population is to allow some of the better organisms from the current generation to carry over to the next, unaltered. This strategy is known as elitist selection.

Parallel implementations of genetic algorithms come in two flavours. Coarse-grained parallel genetic algorithms assume a population on each of the computer nodes and migration of individuals among the nodes. Fine-grained parallel genetic algorithms assume an individual on each processor node which acts with neighboring individuals for selection and reproduction. Other variants, like genetic algorithms for online optimization problems, introduce time-dependence or noise in the fitness function.

Genetic algorithms with adaptive parameters (adaptive genetic algorithms, AGAs) is another significant and promising variant of genetic algorithms. The probabilities of crossover (pc) and mutation (pm) greatly determine the degree of solution accuracy and the convergence speed that genetic algorithms can obtain. Instead of using fixed values of pc and pm, AGAs utilize the population information in each generation and adaptively adjust the pc and pm in order to maintain the population diversity as well as to sustain the convergence capacity. In AGA (adaptive genetic algorithm),[6] the adjustment of pc and pm depends on the fitness values of the solutions. In CAGA (clustering-based adaptive genetic algorithm),[7] through the use of clustering analysis to judge the optimization states of the population, the adjustment of pc and pm depends on these optimization states.

It can be quite effective to combine GA with other optimization methods. GA tends to be quite good at finding generally good global solutions, but quite inefficient at finding the last few mutations to find the absolute optimum. Other techniques (such as simple hill climbing) are quite efficient at finding absolute optimum in a limited region. Alternating GA and hill climbing can improve the efficiency of GA while overcoming the lack of robustness of hill climbing.

This means that the rules of genetic variation may have a different meaning in the natural case. For instance – provided that steps are stored in consecutive order – crossing over may sum a number of steps from maternal DNA adding a number of steps from paternal DNA and so on. This is like adding vectors that more probably may follow a ridge in the phenotypic landscape. Thus, the efficiency of the process may be increased by many orders of magnitude. Moreover, the inversion operator has the opportunity to place steps in consecutive order or any other suitable order in favour of survival or efficiency. (See for instance [8] or example in travelling salesman problem, in particular the use of an edge recombination operator.)

A variation, where the population as a whole is evolved rather than its individual members, is known as gene pool recombination.

**Linkage-learning**

A number of variations have been developed to attempt to improve performance of GAs on problems with a high degree of fitness epistasis, i.e. where the fitness of a solution consists of interacting subsets of its variables. Such algorithms aim to learn (before exploiting) these beneficial phenotypic interactions. As such, they are aligned with the Building Block Hypothesis in adaptively reducing disruptive recombination. Prominent examples of this approach include the mGA,[9] GEMGA[10] and LLGA.[11]
**Problem domains**

Problems which appear to be particularly appropriate for solution by genetic algorithms include timetabling and scheduling problems, and many scheduling software packages are based on GAs. GAs have also been applied to engineering. Genetic algorithms are often applied as an approach to solve global optimization problems.

As a general rule of thumb genetic algorithms might be useful in problem domains that have a complex fitness landscape as mixing, i.e., mutation in combination with crossover, is designed to move the population away from local optima that a traditional hill climbing algorithm might get stuck in. Observe that commonly used crossover operators cannot change any uniform population. Mutation alone can provide ergodicity of the overall genetic algorithm process (seen as a Markov chain).

Examples of problems solved by genetic algorithms include: mirrors designed to funnel sunlight to a solar collector, antennae designed to pick up radio signals in space, and walking methods for computer figures. Many of their solutions have been highly effective, unlike anything a human engineer would have produced, and inscrutable as to how they arrived at that solution.

**History**

Computer simulations of evolution started as early as in 1954 with the work of Nils Aall Barricelli, who was using the computer at the Institute for Advanced Study in Princeton, New Jersey.\(^ {12}\)\(^ {13}\) His 1954 publication was not widely noticed. Starting in 1957,\(^ {14}\) the Australian quantitative geneticist Alex Fraser published a series of papers on simulation of artificial selection of organisms with multiple loci controlling a measurable trait. From these beginnings, computer simulation of evolution by biologists became more common in the early 1960s, and the methods were described in books by Fraser and Burnell (1970)\(^ {15}\) and Crosby (1973).\(^ {16}\) Fraser's simulations included all of the essential elements of modern genetic algorithms. In addition, Hans-Joachim Bremermann published a series of papers in the 1960s that also adopted a population of solution to optimization problems, undergoing recombination, mutation, and selection. Bremermann's research also included the elements of modern genetic algorithms.\(^ {17}\) Other noteworthy early pioneers include Richard Friedberg, George Friedman, and Michael Conrad. Many early papers are reprinted by Fogel (1998).\(^ {18}\)

Although Barricelli, in work he reported in 1963, had simulated the evolution of ability to play a simple game,\(^ {19}\) artificial evolution became a widely recognized optimization method as a result of the work of Ingo Rechenberg and Hans-Paul Schwefel in the 1960s and early 1970s – Rechenberg's group was able to solve complex engineering problems through evolution strategies.\(^ {20}\)\(^ {21}\)\(^ {22}\)\(^ {23}\) Another approach was the evolutionary programming technique of Lawrence J. Fogel, which was proposed for generating artificial intelligence. Evolutionary programming originally used finite state machines for predicting environments, and used variation and selection to optimize the predictive logics. Genetic algorithms in particular became popular through the work of John Holland in the early 1970s, and particularly his book *Adaptation in Natural and Artificial Systems* (1975). His work originated with studies of cellular automata, conducted by Holland and his students at the University of Michigan. Holland introduced a formalized framework for predicting the quality of the next generation, known as Holland's Schema Theorem. Research in GAs remained largely theoretical until the mid-1980s, when The First International Conference on Genetic Algorithms was held in Pittsburgh, Pennsylvania.

As academic interest grew, the dramatic increase in desktop computational power allowed for practical application of the new technique. In the late 1980s, General Electric started selling the world's first genetic algorithm product, a mainframe-based toolkit designed for industrial processes. In 1989, Axcelis, Inc. released Evolver, the world's first commercial GA product for desktop computers. The New York Times technology writer John Markoff wrote\(^ {24}\) about Evolver in 1990.
Related techniques

Parent fields
Genetic algorithms are a sub-field of:
- Evolutionary algorithms
- Evolutionary computing
  - Metaheuristics
    - Stochastic optimization
    - Optimization

Related fields

Evolutionary algorithms
Evolutionary algorithms is a sub-field of evolutionary computing.
- Evolution strategies (ES, see Rechenberg, 1994) evolve individuals by means of mutation and intermediate or discrete recombination. ES algorithms are designed particularly to solve problems in the real-value domain. They use self-adaptation to adjust control parameters of the search. De-randomization of self-adaptation has led to the contemporary Covariance Matrix Adaptation Evolution Strategy (CMA-ES).
- Evolutionary programming (EP) involves populations of solutions with primarily mutation and selection and arbitrary representations. They use self-adaptation to adjust parameters, and can include other variation operations such as combining information from multiple parents.
- Genetic programming (GP) is a related technique popularized by John Koza in which computer programs, rather than function parameters, are optimized. Genetic programming often uses tree-based internal data structures to represent the computer programs for adaptation instead of the list structures typical of genetic algorithms.
- Grouping genetic algorithm (GGA) is an evolution of the GA where the focus is shifted from individual items, like in classical GAs, to groups or subset of items.\textsuperscript{[25]} The idea behind this GA evolution proposed by Emanuel Falkenauer is that solving some complex problems, a.k.a. clustering or partitioning problems where a set of items must be split into disjoint group of items in an optimal way, would better be achieved by making characteristics of the groups of items equivalent to genes. These kind of problems include bin packing, line balancing, clustering with respect to a distance measure, equal piles, etc., on which classic GAs proved to perform poorly. Making genes equivalent to groups implies chromosomes that are in general of variable length, and special genetic operators that manipulate whole groups of items. For bin packing in particular, a GGA hybridized with the Dominance Criterion of Martello and Toth, is arguably the best technique to date.
- Interactive evolutionary algorithms are evolutionary algorithms that use human evaluation. They are usually applied to domains where it is hard to design a computational fitness function, for example, evolving images, music, artistic designs and forms to fit users' aesthetic preference.
Swarm intelligence
Swarm intelligence is a sub-field of evolutionary computing.

- Ant colony optimization (ACO) uses many ants (or agents) to traverse the solution space and find locally productive areas. While usually inferior to genetic algorithms and other forms of local search, it is able to produce results in problems where no global or up-to-date perspective can be obtained, and thus the other methods cannot be applied.

- Particle swarm optimization (PSO) is a computational method for multi-parameter optimization which also uses population-based approach. A population (swarm) of candidate solutions (particles) moves in the search space, and the movement of the particles is influenced both by their own best known position and swarm’s global best known position. Like genetic algorithms, the PSO method depends on information sharing among population members. In some problems the PSO is often more computationally efficient than the GAs, especially in unconstrained problems with continuous variables.\(^{[26]}\)

- Intelligent Water Drops or the IWD algorithm\(^{[27]}\) is a nature-inspired optimization algorithm inspired from natural water drops which change their environment to find the near optimal or optimal path to their destination. The memory is the river's bed and what is modified by the water drops is the amount of soil on the river's bed.

Other evolutionary computing algorithms
Evolutionary computation is a sub-field of the metaheuristic methods.

- Harmony search (HS) is an algorithm mimicking musicians' behaviours in the process of improvisation.

- Memetic algorithm (MA), also called hybrid genetic algorithm among others, is a relatively new evolutionary method where local search is applied during the evolutionary cycle. The idea of memetic algorithms comes from memes, which unlike genes, can adapt themselves. In some problem areas they are shown to be more efficient than traditional evolutionary algorithms.

- Bacteriologic algorithms (BA) inspired by evolutionary ecology and, more particularly, bacteriologic adaptation. Evolutionary ecology is the study of living organisms in the context of their environment, with the aim of discovering how they adapt. Its basic concept is that in a heterogeneous environment, you can't find one individual that fits the whole environment. So, you need to reason at the population level. It is also believed BAs could be successfully applied to complex positioning problems (antennas for cell phones, urban planning, and so on) or data mining.\(^{[28]}\)

- Cultural algorithm (CA) consists of the population component almost identical to that of the genetic algorithm and, in addition, a knowledge component called the belief space.

- Gaussian adaptation (normal or natural adaptation, abbreviated NA to avoid confusion with GA) is intended for the maximisation of manufacturing yield of signal processing systems. It may also be used for ordinary parametric optimisation. It relies on a certain theorem valid for all regions of acceptability and all Gaussian distributions. The efficiency of NA relies on information theory and a certain theorem of efficiency. Its efficiency is defined as information divided by the work needed to get the information.\(^{[29]}\) Because NA maximises mean fitness rather than the fitness of the individual, the landscape is smoothed such that valleys between peaks may disappear. Therefore it has a certain “ambition" to avoid local peaks in the fitness landscape. NA is also good at climbing sharp crests by adaptation of the moment matrix, because NA may maximise the disorder (average information) of the Gaussian simultaneously keeping the mean fitness constant.
Other metaheuristic methods

Metaheuristic methods broadly fall within stochastic optimisation methods.

• Simulated annealing (SA) is a related global optimization technique that traverses the search space by testing random mutations on an individual solution. A mutation that increases fitness is always accepted. A mutation that lowers fitness is accepted probabilistically based on the difference in fitness and a decreasing temperature parameter. In SA parlance, one speaks of seeking the lowest energy instead of the maximum fitness. SA can also be used within a standard GA algorithm by starting with a relatively high rate of mutation and decreasing it over time along a given schedule.

• Tabu search (TS) is similar to simulated annealing in that both traverse the solution space by testing mutations of an individual solution. While simulated annealing generates only one mutated solution, tabu search generates many mutated solutions and moves to the solution with the lowest energy of those generated. In order to prevent cycling and encourage greater movement through the solution space, a tabu list is maintained of partial or complete solutions. It is forbidden to move to a solution that contains elements of the tabu list, which is updated as the solution traverses the solution space.

• Extremal optimization (EO) Unlike GAs, which work with a population of candidate solutions, EO evolves a single solution and makes local modifications to the worst components. This requires that a suitable representation be selected which permits individual solution components to be assigned a quality measure ("fitness"). The governing principle behind this algorithm is that of emergent improvement through selectively removing low-quality components and replacing them with a randomly selected component. This is decidedly at odds with a GA that selects good solutions in an attempt to make better solutions.

Other stochastic optimisation methods

• The cross-entropy (CE) method generates candidates solutions via a parameterized probability distribution. The parameters are updated via cross-entropy minimization, so as to generate better samples in the next iteration.

• Reactive search optimization (RSO) advocates the integration of sub-symbolic machine learning techniques into search heuristics for solving complex optimization problems. The word reactive hints at a ready response to events during the search through an internal online feedback loop for the self-tuning of critical parameters. Methodologies of interest for Reactive Search include machine learning and statistics, in particular reinforcement learning, active or query learning, neural networks, and meta-heuristics.

References

[10] Gene expression: The missing link in evolutionary computation
Genetic algorithm

[17] 02.27.96 - UC Berkeley's Hans Bremermann, professor emeritus and pioneer in mathematical biology, has died at 69 (http://berkeley.edu/news/media/releases/96legacy/releases.96/4139.html)


Bibliography


• Hingston, Philip F.; Barone, Luigi C.; Michalewicz, Zbigniew (2008) Design by Evolution: Advances in Evolutionary Design:297

• Eiben, Agoston E.; Smith, James E. (2003) *Introduction to Evolutionary Computing*

### External links

#### Resources

- DigitalBiology.NET (http://www.digitalbiology.net/) Vertical search engine for GA/GP resources
- Genetic Algorithms Index (http://www.geneticprogramming.com/ga/index.htm) The site Genetic Programming Notebook provides a structured resource pointer to web pages in genetic algorithms field

#### Tutorials

- Genetic Algorithms Computer programs that "evolve" in ways that resemble natural selection can solve complex problems even their creators do not fully understand (http://www2.econ.iastate.edu/tesfatsi/holland.gaintro.htm) An excellent introduction to GA by John Holland and with an application to the Prisoner's Dilemma
- An online interactive GA demonstrator to practise or learn how a GA works. (http://userweb.elec.gla.ac.uk/y/unii/ga_demo/) Learn step by step or watch global convergence in batch, change population size, crossover rate, mutation rate and selection mechanism, and add constraints.
- A Genetic Algorithm Tutorial by Darrell Whitley Computer Science Department Colorado State University (http://samizdat.mines.edu/ga_tutorial/ga_tutorial.ps) An excellent tutorial with lots of theory
Examples

- Introduction to Genetic Algorithms with interactive Java applets. (http://www.obitko.com/tutorials/genetic-algorithms/) For experimenting with GAs online.
- Cross discipline example applications for GAs with references. (http://www.talkorigins.org/faqs/genalg/genalg.html)
- An interactive applet featuring evolving vehicles. (http://boxcar2d.com/)

Toy block

Toy blocks (also building bricks, building blocks, or simply blocks), are wooden, plastic or foam pieces of various shapes (square, cylinder, arch, triangle, etc.) and colors that are used as building toys. Sometimes toy blocks depict letters of the alphabet.

History

1693: One of the first references to Alphabet Nursery Blocks was made by English philosopher John Locke, in 1693, made the statement that "dice and playthings, with letters on them to teach children the alphabet by playing" would make learning to read a more enjoyable experience.[1]

1798: Witold Rybczynski has found that the earliest mention of building bricks for children appears in Maria and R.L. Edgeworth's Practical Education (1798). Called "rational toys," blocks were intended to teach children about gravity and physics, as well as spatial relationships that allow them to see how many different parts become a whole.[2]

1820: The first large-scale production of blocks was in the Williamsburg area of Brooklyn by S. L. Hill, who patented "ornamenting wood" a patent related to painting or coloring a block surface prior to the embossing process and then adding another color after the embossing to have multi-colored blocks.[3]

2003: National Toy Hall of Fame at the Strong Museum, inducted ABC blocks into their collection, granting it the title of one of America's toys of national significance.[4]

**Educational benefits**

- *Physical benefits*: toy blocks build strength in a child’s fingers and hands, and improve eye-hand coordination. They also help educate children in different shapes.
- *Social benefits*: block play encourages children to make friends and cooperate, and is often one of the first experiences a child has playing with others. Blocks are a benefit for the children because they encourage interaction and imagination. Creativity can be a combined action that is important for social play.
- *Intellectual benefits*: children can potentially develop their vocabularies as they learn to describe sizes, shapes, and positions. Math skills are developed through the process of grouping, adding, and subtracting, particularly with standardized blocks, such as unit blocks. Experiences with gravity, balance, and geometry learned from toy blocks also provide intellectual stimulation.
- *Creative benefits*: children receive creative stimulation by making their own designs with blocks.

**In popular culture**

Art Clokey, the creator of Gumby, has stated that Gumby's nemeses, the Block-heads, evolved from the blocks that appeared in the toy store that originally provided the setting for the stop-motion series.[5]

**References**


- *Block play: Building a child's mind* (http://www.woodentoy.com/html/BlocksGoodToy.html), the National Association for the Education of Young Children
Chromosome (genetic algorithm)

In genetic algorithms, a chromosome (also sometimes called a genome) is a set of parameters which define a proposed solution to the problem that the genetic algorithm is trying to solve. The chromosome is often represented as a simple string, although a wide variety of other data structures are also used.

Chromosome design

The article would also benefit from more relevant and clearer examples. The design of the chromosome and its parameters is by necessity specific to the problem to be solved. To give a trivial example, suppose the problem is to find the integer value of \( x \) between 0 and 255 that provides the maximal result for \( f(x) = x^2 \). (This isn’t the type of problem that is normally solved by a genetic algorithm, since it can be trivially solved using numeric methods. It is only used to serve as a simple example.) Our possible solutions are the integers from 0 to 255, which can all be represented as 8-digit binary strings. Thus, we might use an 8-digit binary string as our chromosome. If a given chromosome in the population represents the value 155, its chromosome would be 10011011.

A more realistic problem we might wish to solve is the travelling salesman problem. In this problem, we seek an ordered list of cities that results in the shortest trip for the salesman to travel. Suppose there are six cities, which we’ll call A, B, C, D, E, and F. A good design for our chromosome might be the ordered list we want to try. An example chromosome we might encounter in the population might be DFABEC.

The mutation operator and crossover operator employed by the genetic algorithm must take into account the chromosome’s design.

Genetic operator

A genetic operator is an operator used in genetic algorithms to maintain genetic diversity, known as Mutation (genetic algorithm) and to combine existing solutions into others, Crossover (genetic algorithm). The main difference between them is that the mutation operators operate on one chromosome, that is, they are unary, while the crossover operators are binary operators.

Genetic variation is a necessity for the process of evolution. Genetic operators used in genetic algorithms are analogous to those in the natural world: survival of the fittest, or selection; reproduction (crossover, also called recombination); and mutation.

Types of Operators

1. Mutation (genetic algorithm)
2. Crossover (genetic algorithm)
Crossover (genetic algorithm)

In genetic algorithms, crossover is a genetic operator used to vary the programming of a chromosome or chromosomes from one generation to the next. It is analogous to reproduction and biological crossover, upon which genetic algorithms are based. Cross over is a process of taking more than one parent solutions and producing a child solution from them. There are methods for selection of the chromosomes. Those are also given below.

Methods of selection of chromosomes for crossover

- **Roulette wheel selection** (SCX) It is also known as fitness proportionate selection. The individual is selected on the basis of fitness. The probability of an individual to be selected increases with the fitness of the individual greater or less than its competitor's fitness.
- **Boltzmann selection**
- **Tournament selection**
- **Rank selection**
- **Steady state selection**

Crossover techniques

Many crossover techniques exist for organisms which use different data structures to store themselves.

**One-point crossover**

A single crossover point on both parents' organism strings is selected. All data beyond that point in either organism string is swapped between the two parent organisms. The resulting organisms are the children:

![One-point crossover diagram]

**Two-point crossover**

Two-point crossover calls for two points to be selected on the parent organism strings. Everything between the two points is swapped between the parent organisms, rendering two child organisms:

![Two-point crossover diagram]

"Cut and splice"

Another crossover variant, the "cut and splice" approach, results in a change in length of the children strings. The reason for this difference is that each parent string has a separate choice of crossover point.

![Cut and splice diagram]
Uniform Crossover and Half Uniform Crossover

The Uniform Crossover uses a fixed mixing ratio between two parents. Unlike one- and two-point crossover, the Uniform Crossover enables the parent chromosomes to contribute the gene level rather than the segment level. If the mixing ratio is 0.5, the offspring has approximately half of the genes from first parent and the other half from second parent, although crossover points can be randomly chosen as seen below. Uniform Crossover evaluates each bit in the parent strings for exchange with a probability of 0.5. Even though the uniform crossover is a poor method, empirical evidence suggest that it is a more exploratory approach to crossover than the traditional exploitative approach that maintains longer schemata. This results in a more complete search of the design space with maintaining the exchange of good information. Unfortunately, no satisfactory theory exists to explain the discrepancies between the Uniform Crossover and the traditional approaches. In the uniform crossover scheme (UX) individual bits in the string are compared between two parents. The bits are swapped with a fixed probability, typically 0.5. In the half uniform crossover scheme (HUX), exactly half of the nonmatching bits are swapped. Thus first the Hamming distance (the number of differing bits) is calculated. This number is divided by two. The resulting number is how many of the bits that do not match between the two parents will be swapped.

Three parent crossover

In this technique, the child is derived from three parents. They are randomly chosen. Each bit of first parent is checked with bit of second parent whether they are same. If same then the bit is taken for the offspring otherwise the bit from the third parent is taken for the offspring. parent1 1 1 0 1 0 0 0 1 0 parent2 0 1 1 0 0 1 0 0 1 parent3 1 1 0 1 0 1 0 0 1 offspring 1 1 0 1 0 0 0 0 1

Crossover for Ordered Chromosomes

Depending on how the chromosome represents the solution, a direct swap may not be possible. One such case is when the chromosome is an ordered list, such as an ordered list of the cities to be travelled for the traveling salesman problem. There are many crossover methods for ordered chromosomes. The already mentioned N-point crossover can be applied for ordered chromosomes also, but this always need a corresponding repair process, actually, some ordered crossover methods are derived from the idea. However, sometimes a crossover of chromosomes produces recombinations which violate the constraint of ordering and thus need to be repaired. Several examples for crossover operators (also mutation operator) preserving a given order are given in:

1. partially matched crossover (PMX): In this method, two crossover points are selected at random and PMX proceeds by position wise exchanges. The two crossover points give matching selection. It affects cross by position-by-position exchange operations. In this method parents are mapped to each other, hence we can also call it partially mapped crossover.
2. cycle crossover (CX): Beginning at any gene \( i \) in parent 1, the \( i \)-th gene in parent 2 becomes replaced by it. The same is repeated for the displaced gene until the gene which is equal to the first inserted gene becomes replaced (cycle).
3. order crossover operator (OX1): A portion of one parent is mapped to a portion of the other parent. From the replaced portion on, the rest is filled up by the remaining genes, where already present genes are omitted and the order is preserved.
4. order-based crossover operator (OX2)
5. position-based crossover operator (POS)
6. voting recombination crossover operator (VR)
7. alternating-position crossover operator (AP)
8. sequential constrictive crossover operator (SCX) [6]

Other possible methods include the edge recombination operator and partially mapped crossover.

**Crossover biases**

For crossover operators which exchange contiguous sections of the chromosomes (e.g. k-point) the ordering of the variables may become important. This is particularly true when good solutions contain building blocks which might be disrupted by a non-respectful crossover operator.

**References**


[3] Introduction to genetic algorithms By S. N. Sivanandam, S. N. Deepa
[5] Introduction to genetic algorithms By S. N. Sivanandam, S. N. Deepa

**External links**

**Mutation (genetic algorithm)**

In genetic algorithms of computing, mutation is a genetic operator used to maintain genetic diversity from one generation of a population of algorithm chromosomes to the next. It is analogous to biological mutation. Mutation alters one or more gene values in a chromosome from its initial state. In mutation, the solution may change entirely from the previous solution. Hence GA can come to better solution by using mutation. Mutation occurs during evolution according to a user-definable mutation probability. This probability should be set low. If it is set to high, the search will turn into a primitive random search.

The classic example of a mutation operator involves a probability that an arbitrary bit in a genetic sequence will be changed from its original state. A common method of implementing the mutation operator involves generating a random variable for each bit in a sequence. This random variable tells whether or not a particular bit will be modified. This mutation procedure, based on the biological point mutation, is called single point mutation. Other types are inversion and floating point mutation. When the gene encoding is restrictive as in permutation problems, mutations are swaps, inversions and scrambles.

The purpose of mutation in GAs is preserving and introducing diversity. Mutation should allow the algorithm to avoid local minima by preventing the population of chromosomes from becoming too similar to each other, thus slowing or even stopping evolution. This reasoning also explains the fact that most GA systems avoid only taking the fittest of the population in generating the next but rather a random (or semi-random) selection with a weighting toward those that are fitter.\[1\]

For different genome types, different mutation types are suitable:

- **Bit string mutation**
  
  The mutation of bit strings ensue through bit flips at random positions.
  
  Example:

  \[
  \begin{array}{c}
  1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \\
  \downarrow \\
  1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0
  \end{array}
  \]

  The probability of a mutation of a bit is \( \frac{1}{l} \), where \( l \) is the length of the binary vector. Thus, a mutation rate of \( \frac{1}{l} \) per mutation and individual selected for mutation is reached.

- **Flip Bit**
  
  This mutation operator takes the chosen genome and inverts the bits. (i.e. if the genome bit is 1, it is changed to 0 and vice versa)

- **Boundary**
  
  This mutation operator replaces the genome with either lower or upper bound randomly. This can be used for integer and float genes.

- **Non-Uniform**
  
  The probability that amount of mutation will go to 0 with the next generation is increased by using non-uniform mutation operator. It keeps the population from stagnating in the early stages of the evolution. It tunes solution in later stages of evolution. This mutation operator can only be used for integer and float genes.

- **Uniform**
  
  This operator replaces the value of the chosen gene with a uniform random value selected between the user-specified upper and lower bounds for that gene. This mutation operator can only be used for integer and float genes.
• Gaussian
This operator adds a unit Gaussian distributed random value to the chosen gene. If it falls outside of the user-specified lower or upper bounds for that gene, the new gene value is clipped. This mutation operator can only be used for integer and float genes.

References

Bibliography

Inheritance (genetic algorithm)
In genetic algorithms, inheritance is the ability of modelled objects to mate, mutate and propagate their problem solving genes to the next generation, in order to produce an evolved solution to a particular problem.

Selection (genetic algorithm)
Selection is the stage of a genetic algorithm in which individual genomes are chosen from a population for later breeding (recombination or crossover).

A generic selection procedure may be implemented as follows:
1. The fitness function is evaluated for each individual, providing fitness values, which are then normalized.
   Normalization means dividing the fitness value of each individual by the sum of all fitness values, so that the sum of all resulting fitness values equals 1.
2. The population is sorted by descending fitness values.
3. Accumulated normalized fitness values are computed (the accumulated fitness value of an individual is the sum of its own fitness value plus the fitness values of all the previous individuals). The accumulated fitness of the last individual should be 1 (otherwise something went wrong in the normalization step).
4. A random number $R$ between 0 and 1 is chosen.
5. The selected individual is the first one whose accumulated normalized value is greater than $R$.

If this procedure is repeated until there are enough selected individuals, this selection method is called fitness proportionate selection or roulette-wheel selection. If instead of a single pointer spun multiple times, there are multiple, equally spaced pointers on a wheel that is spun once, it is called stochastic universal sampling. Repeatedly selecting the best individual of a randomly chosen subset is tournament selection. Taking the best half, third or another proportion of the individuals is truncation selection.

There are other selection algorithms that do not consider all individuals for selection, but only those with a fitness value that is higher than a given (arbitrary) constant. Other algorithms select from a restricted pool where only a certain percentage of the individuals are allowed, based on fitness value.

Retaining the best individuals in a generation unchanged in the next generation, is called elitism or elitist selection. It is a successful (slight) variant of the general process of constructing a new population.

See the main article on genetic algorithms for the context in which selection is used.
See Also

- Fitness proportionate selection
- Stochastic universal sampling
- Tournament selection
- Reward-based selection

External links

- Introduction to Genetic Algorithms [1]

References


Tournament selection

Tournament selection is a method of selecting an individual from a population of individuals in a genetic algorithm. Tournament selection involves running several "tournaments" among a few individuals chosen at random from the population. The winner of each tournament (the one with the best fitness) is selected for crossover. Selection pressure is easily adjusted by changing the tournament size. If the tournament size is larger, weak individuals have a smaller chance to be selected.

Tournament selection pseudo code:

```plaintext
choose k (the tournament size) individuals from the population at random
choose the best individual from pool/tournament with probability p
choose the second best individual with probability p*(1-p)
choose the third best individual with probability p*((1-p)^2)
and so on...
```

Deterministic tournament selection selects the best individual (when p=1) in any tournament. A 1-way tournament (k=1) selection is equivalent to random selection. The chosen individual can be removed from the population that the selection is made from if desired, otherwise individuals can be selected more than once for the next generation.

Tournament selection has several benefits: it is efficient to code, works on parallel architectures and allows the selection pressure to be easily adjusted.

See Also

- Fitness proportionate selection
- Reward-based selection

External links

Truncation selection

Truncation selection is a selection method used in genetic algorithms to select potential candidate solutions for recombination.

In truncation selection the candidate solutions are ordered by fitness, and some proportion, $p$, (e.g. $p=1/2$, $1/3$, etc.), of the fittest individuals are selected and reproduced $1/p$ times. Truncation selection is less sophisticated than many other selection methods, and is not often used in practice. It is used in Muhlenbein's Breeder Genetic Algorithm.[1]

References


Fitness proportionate selection

Fitness proportionate selection, also known as roulette-wheel selection, is a genetic operator used in genetic algorithms for selecting potentially useful solutions for recombination.

In fitness proportionate selection, as in all selection methods, the fitness function assigns a fitness to possible solutions or chromosomes. This fitness level is used to associate a probability of selection with each individual chromosome. If $f_i$ is the fitness of individual $i$ in the population, its probability of being selected is $p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$, where $N$ is the number of individuals in the population.

This could be imagined similar to a Roulette wheel in a casino. Usually a proportion of the wheel is assigned to each of the possible selections based on their fitness value. This could be achieved by dividing the fitness of a selection by the total fitness of all the selections, thereby normalizing them to 1. Then a random selection is made similar to how the roulette wheel is rotated.

While candidate solutions with a higher fitness will be less likely to be eliminated, there is still a chance that they may be. Contrast this with a less sophisticated selection algorithm, such as truncation selection, which will eliminate a fixed percentage of the weakest candidates. With fitness proportionate selection there is a chance some weaker solutions may survive the selection process; this is an advantage, as though a solution may be weak, it may include some component which could prove useful following the recombination process.

The analogy to a roulette wheel can be envisaged by imagining a roulette wheel in which each candidate solution represents a pocket on the wheel; the size of the pockets are proportionate to the probability of selection of the solution. Selecting $N$ chromosomes from the population is equivalent to playing $N$ games on the roulette wheel, as each candidate is drawn independently.
Other selection techniques, such as stochastic universal sampling\cite{1} or tournament selection, are often used in practice. This is because they have less stochastic noise, or are fast, easy to implement and have a constant selection pressure \cite{Blickle, 1996}.

Note performance gains can be achieved by using a binary search rather than a linear search to find the right pocket.

### See Also
- Stochastic universal sampling
- Tournament selection
- Reward-based selection

### External links
- C implementation \cite{2} (.tar.gz; see selector.cxx) WBL
- Example on Roulette wheel selection \cite{3}

### References
\[2\] http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/GProc-1.8b.tar.gz
\[3\] http://www.edc.ncl.ac.uk/highlight/rhjanuary2007/g02.php

---

**Reward-based selection** is a technique used in evolutionary algorithms for selecting potentially useful solutions for recombination. The probability of being selected for an individual is proportional to the cumulative reward, obtained by the individual. The cumulative reward can be computed as a sum of the individual reward and the reward, inherited from parents.

### Description

Reward-based selection can be used within Multi-armed bandit framework for Multi-objective optimization to obtain a better approximation of the Pareto front. \cite{1}

The newborn \(a'(g+1)\) and its parents receive a reward \(r(a)\), if \(a'(g+1)\) was selected for new population \(Q(g+1)\), otherwise the reward is zero. Several reward definitions are possible:

1. \(r(a) = \begin{cases} 1 & \text{if the newborn individual } a'(g+1) \text{ was selected for new population } Q(g+1) \end{cases}\)

2. \(r(a) = 1 - \frac{\text{rank}(a'(g+1))}{\mu} \) if \(a'(g+1) \in Q(g+1)\), where \(\text{rank}(a'(g+1))\) is the rank of newly inserted individual in the population of \(\mu\) individuals. Rank can be computed using a well-known non-dominated sorting procedure. \cite{2}

3. \(r(a) = \sum_{a \in Q(g+1)} \Delta H(a, Q(g+1)) - \sum_{a \in Q(g)} \Delta H(a, Q(g))\), where \(\Delta H(a, Q(g))\) is the hypervolume indicator contribution of the individual \(a\) to the population \(Q(g)\). The reward \(r(a) > 0\) if the newly inserted individual improves the quality of the population, which is measured as its hypervolume contribution in the objective space.

4. A relaxation of the above reward, involving a rank-based penalization for points for \(k\)-th dominated Pareto front: \(r(g) = \frac{1}{2^{k-1}} \left( \sum_{\text{ndom}_k(Q(g+1))} \Delta H(a, \text{ndom}_k(Q(g+1))) - \sum_{\text{ndom}_k(Q(g))} \Delta H(a, \text{ndom}_k(Q(g))) \right)\)
Reward-based selection can quickly identify the most fruitful directions of search by maximizing the cumulative reward of individuals.

References


Edge recombination operator

The edge recombination operator (ERO) is an operator that creates a path that is similar to a set of existing paths (parents) by looking at the edges rather than the vertices. The main application of this is for crossover in genetic algorithms when a genotype with non-repeating gene sequences is needed such as for the travelling salesman problem.

Algorithm

ERO is based on an adjacency matrix, which lists the neighbors of each node in any parent.

For example, in a travelling salesman problem such as the one depicted, the node map for the parents CABDEF and ABCEFD (see illustration) is generated by taking the first parent, say, ‘ABCEFD’ and recording its immediate neighbors, including those that roll around the end of the string.

Therefore;

\[ \ldots \rightarrow [A] \leftrightarrow [B] \leftrightarrow [C] \leftrightarrow [E] \leftrightarrow [F] \leftrightarrow [D] \leftrightarrow \ldots \]

... is converted into the following adjacency matrix by taking each node in turn, and listing its connected neighbors;

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>F</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>C</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>E</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With the same operation performed on the second parent (CABDEF), the following is produced:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>B</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>D</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>E</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Followed by making a union of these two lists, and ignoring any duplicates. This is as simple as taking the elements of each list and appending them to generate a list of unique link end points. In our example, generating this:

| A: B C D | = {B, D} ∪ {C, B} |
| B: A C D | = {A, C} ∪ {A, D} |
| C: A B E F | = {B, E} ∪ {F, A} |
| D: A B E F | = {F, A} ∪ {B, E} |
| E: C D F | = {C, F} ∪ {D, F} |
| F: C D E | = {E, D} ∪ {E, C} |

The result is another adjacency matrix, which stores the links for a network described by all the links in the parents. Note that more than two parents can be employed here to give more diverse links. However, this approach may result in sub-optimal paths.

Then, to create a path K, the following algorithm is employed:

Let K be the empty list
Let N be the first node of a random parent.

While Length(K) < Length(Parent):
    K := K, N   (append N to K)
    Remove N from all neighbor lists

If N's neighbor list is non-empty
    then let N* be the neighbor of N with the fewest neighbors in its list (or a random one, should there be multiple)
    else let N* be a randomly chosen node that is not in K

N := N*

To step through the example, we randomly select a node from the parent starting points, {A, C}.

- () -> A. We remove A from all the neighbor sets, and find that the smallest of B, C and D is B={C,D}.  
- AB. The smallest sets of C and D are C={E,F} and D={E,F}. We randomly select D.  
- ABD. Smallest are E={C,F}, F={C,E}. We pick F.  
- ABDF. C={E}, E={C}. We pick C.  
- ABDFC. The smallest set is E={}.  
- ABDFCE. The length of the child is now the same as the parent, so we are done.

Note that the only edge introduced in ABDFCE is AE.

**Comparison with other operators**

If one were to use an indirect representation for these parents (where each number in turn indexes and removes an element from an initially sorted set of nodes) and cross them with simple one-point crossover, one would get the following:
The parents:

31|1111  (CABDEF)
11|1211  (ABCEFD)

The children:

11|1111  (ABCDEF)
31|1211  (ABEDFC)

Both children introduce the edges CD and FA.

The reason why frequent edge introduction is a bad thing in these kinds of problem is that very few of the edges tend to be usable and many of them severely inhibit an otherwise good solution. The optimal route in the examples is ABDFEC, but swapping A for F turns it from optimal to far below an average random guess.

The difference between ERO and the indirect one-point crossover can be seen in the diagram. It takes ERO 25 generations of 500 individuals to reach 80% of the optimal path in a 29 point data set, something the indirect representation spends 150 generations on. Partially mapped crossover (PMX) ranks between ERO and indirect one-point crossover, with 80 generations for this particular target.\[1\]

References

\[1\] The traveling salesman and sequence scheduling: quality solutions using genetic edge recombination


Implementations

- "Edge Recombination Operator" (http://github.com/raunak/Travelling-Salesman-Problem/blob/master/edge_recombination.py) (Python)
Population-based incremental learning

In computer science and machine learning, population-based incremental learning (PBIL) is an optimization algorithm, and an estimation of distribution algorithm. This is a type of genetic algorithm where the genotype of an entire population (probability vector) is evolved rather than individual members[1]. The algorithm is proposed by Shumeet Baluja in 1994. The algorithm is simpler than a standard genetic algorithm, and in many cases leads to better results than a standard genetic algorithm[2][3][4].

Algorithm

In PBIL, genes are represented as real values in the range [0,1], indicating the probability that any particular allele appears in that gene.

The PBIL algorithm is as follows:

1. A population is generated from the probability vector.
2. The fitness of each member is evaluated and ranked.
3. Update population genotype (probability vector) based on fittest individual.
4. Mutate.
5. Repeat steps 1-4

Source code

This is a part of source code implemented in Java. In the paper, learnRate = 0.1, negLearnRate = 0.075, mutProb = 0.02, and mutShift = 0.05 is used. N = 100 and ITER_COUNT = 1000 is enough for a small problem.

```java
public void optimize() {
    final int totalBits = getTotalBits(domains);
    final double[] probVec = new double[totalBits];
    Arrays.fill(probVec, 0.5);
    bestCost = POSITIVE_INFINITY;

    for (int i = 0; i < ITER_COUNT; i++) {
        // Creates N genes
        final boolean[][] genes = new boolean[N][totalBits];
        for (boolean[] gene : genes) {
            for (int k = 0; k < gene.length; k++) {
                if (rand.nextDouble() < probVec[k])
                    gene[k] = true;
            }
        }

        // Calculate costs
        final double[] costs = new double[N];
        for (int j = 0; j < N; j++) {
            costs[j] = costFunc.cost(toRealVec(genes[j], domains));
        }

        // Find min and max cost genes
        boolean[] minGene = null, maxGene = null;
```
double minCost = POSITIVE_INFINITY, maxCost = NEGATIVE_INFINITY;
for (int j = 0; j < N; j++) {
    double cost = costs[j];
    if (minCost > cost) {
        minCost = cost;
        minGene = genes[j];
    }
    if (maxCost < cost) {
        maxCost = cost;
        maxGene = genes[j];
    }
}

// Compare with the best cost gene
if (bestCost > minCost) {
    bestCost = minCost;
    bestGene = minGene;
}

// Update the probability vector with max and min cost genes
for (int j = 0; j < totalBits; j++) {
    if (minGene[j] == maxGene[j]) {
        probVec[j] = probVec[j] * (1d - learnRate) +
                    (minGene[j] ? 1d : 0d) * learnRate;
    } else {
        final double learnRate2 = learnRate + negLearnRate;
        probVec[j] = probVec[j] * (1d - learnRate2) +
                    (minGene[j] ? 1d : 0d) * learnRate2;
    }
}

// Mutation
for (int j = 0; j < totalBits; j++) {
    if (rand.nextDouble() < mutProb) {
        probVec[j] = probVec[j] * (1d - mutShift) +
                    (rand.nextBoolean() ? 1d : 0d) * mutShift;
    }
}
}
References


Defining length

In genetic algorithms and genetic programming defining length $L(H)$ is the maximum distance between two defining symbols (that is symbols that have a fixed value as opposed to symbols that can take any value, commonly denoted as # or *) in schema $H$. In tree GP schemata, $L(H)$ is the number of links in the minimum tree fragment including all the non-= symbols within a schema $H$.\[1\]

Example

Schemata "00##0", "1###1", "01###", and "##0##" have defining lengths of 4, 4, 1, and 0, respectively. Lengths are computed by determining the last fixed position and subtracting from it the first fixed position.

In genetic algorithms as the defining length of a solution increases so does the susceptibility of the solution to disruption due to mutation or cross-over.

References

Holland's schema theorem

Holland's schema theorem is widely taken to be the foundation for explanations of the power of genetic algorithms. It was proposed by John Holland in the 1970s.

A schema is a template that identifies a subset of strings with similarities at certain string positions. Schemata are a special case of cylinder sets; and so form a topological space.

Description

For example, consider binary strings of length 6. The schema 1*10*1 describes the set of all strings of length 6 with 1’s at positions 1, 3 and 6 and a 0 at position 4. The * is a wildcard symbol, which means that positions 2 and 5 can have a value of either 1 or 0. The order of a schema is defined as the number of fixed positions in the template, while the defining length δ(H) is the distance between the first and last specific positions. The order of 1*10*1 is 4 and its defining length is 5. The fitness of a schema is the average fitness of all strings matching the schema. The fitness of a string is a measure of the value of the encoded problem solution, as computed by a problem-specific evaluation function. Using the established methods and genetic operators of genetic algorithms, the schema theorem states that short, low-order schemata with above-average fitness increase exponentially in successive generations. Expressed as an equation:

$$E(m(H, t + 1)) \geq \frac{m(H, t)f(H)}{a_t}[1 - p].$$

Here $m(H, t)$ is the number of strings belonging to schema $H$ at generation $t$. $f(H)$ is the observed fitness of schema $H$ and $a_t$ is the observed average fitness at generation $t$. The probability of disruption $p$ is the probability that crossover or mutation will destroy the schema $H$. It can be expressed as:

$$p = \frac{\delta(H)}{l-1}p_c + o(H)p_m$$

where $o(H)$ is the number of fixed positions, $l$ is the length of the code, $p_m$ is the probability of mutation and $p_c$ is the probability of crossover. So a schema with a shorter defining length $\delta(H)$ is less likely to be disrupted.

An often misunderstood point is why the Schema Theorem is an inequality rather than an equality. The answer is in fact simple: the Theorem neglects the small, yet non-zero, probability that a string belonging to the schema $H$ will be created "from scratch" by mutation of a single string (or recombination of two strings) that did not belong to $H$ in the previous generation.

References

Genetic memory (computer science)

In computer science, genetic memory refers to an artificial neural network combination of genetic algorithm and the mathematical model of sparse distributed memory. It can be used to predict weather patterns.\(^1\) Genetic memory and genetic algorithms have also gained an interest in the creation of artificial life.\(^2\)

References


Premature convergence

In genetic algorithms, the term of premature convergence means that a population for an optimization problem converged too early, resulting in being suboptimal. In this context, the parental solutions, through the aid of genetic operators, are not able to generate offsprings that are superior to their parents. Premature convergence can happen in case of loss of genetic variation (every individual in the population is identical, see convergence).

Strategies for preventing premature convergence

Strategies to regain genetic variation can be:

- a mating strategy called incest prevention,\(^1\)
- uniform crossover,
- favored replacement of similar individuals (preselection or crowding),
- segmentation of individuals of similar fitness (fitness sharing),
- increasing population size.

The genetic variation can also be regained by mutation though this process is highly random.

References

Schema (genetic algorithms)

A schema is a template in computer science used in the field of genetic algorithms that identifies a subset of strings with similarities at certain string positions. Schemata are a special case of cylinder sets; and so form a topological space.[1]

Description

For example, consider binary strings of length 6. The schema 1**0*1 describes the set of all words of length 6 with 1’s at positions 1 and 6 and a 0 at position 4. The * is a wildcard symbol, which means that positions 2, 3 and 5 can have a value of either 1 or 0. The order of a schema is defined as the number of fixed positions in the template, while the defining length $\delta(H)$ is the distance between the first and last specific positions. The order of 1**0*1 is 3 and its defining length is 5. The fitness of a schema is the average fitness of all strings matching the schema. The fitness of a string is a measure of the value of the encoded problem solution, as computed by a problem-specific evaluation function.

Length

The length of a schema $H$, called $N(H)$, is defined as the total number of nodes in the schema. $N(H)$ is also equal to the number of nodes in the programs matching $H$. [2]

Disruption

If the child of an individual that matches schema H does not itself match H, the schema is said to have been disrupted.[2]

References

Fitness function

A **fitness function** is a particular type of objective function that is used to summarise, as a single figure of merit, how close a given design solution is to achieving the set aims.

In particular, in the fields of genetic programming and genetic algorithms, each design solution is represented as a string of numbers (referred to as a chromosome). After each round of testing, or simulation, the idea is to delete the 'n' worst design solutions, and to breed 'n' new ones from the best design solutions. Each design solution, therefore, needs to be awarded a figure of merit, to indicate how close it came to meeting the overall specification, and this is generated by applying the fitness function to the test, or simulation, results obtained from that solution.

The reason that genetic algorithms are not a lazy way of performing design work is precisely because of the effort involved in designing a workable fitness function. Even though it is no longer the human designer, but the computer, that comes up with the final design, it is the human designer who has to design the fitness function. If this is designed wrongly, the algorithm will either converge on an inappropriate solution, or will have difficulty converging at all.

Moreover, the fitness function must not only correlate closely with the designer's goal, it must also be computed quickly. Speed of execution is very important, as a typical genetic algorithm must be iterated many times in order to produce a usable result for a non-trivial problem.

Fitness approximation may be appropriate, especially in the following cases:

- Fitness computation time of a single solution is extremely high
- Precise model for fitness computation is missing
- The fitness function is uncertain or noisy.

Two main classes of fitness functions exist: one where the fitness function does not change, as in optimizing a fixed function or testing with a fixed set of test cases; and one where the fitness function is mutable, as in niche differentiation or co-evolving the set of test cases.

Another way of looking at fitness functions is in terms of a fitness landscape, which shows the fitness for each possible chromosome.

Definition of the fitness function is not straightforward in many cases and often is performed iteratively if the fittest solutions produced by GA are not what is desired. In some cases, it is very hard or impossible to come up even with a guess of what fitness function definition might be. Interactive genetic algorithms address this difficulty by outsourcing evaluation to external agents (normally humans).

**References**

- An Nice Introduction to Adaptive Fuzzy Fitness Granulation (AFFG) (http://profsite.um.ac.ir/~davarynej/Resources/CEC07-Draft.pdf) (PDF), A promising approach to accelerate the convergence rate of EAs. Available as a free PDF.
- The cyber shack of Adaptive Fuzzy Fitness Granulation (AFFG) (http://www.davarynejad.com/Mohsen/index.php?n=Main.AFFG) That is designed to accelerate the convergence rate of EAs.
- Fitness functions in evolutionary robotics: A survey and analysis (AFFG) (http://www.nelsonrobotics.org/paper_archive_nelson/nelson-jras-2009.pdf) (PDF), A review of fitness functions used in evolutionary robotics.
Black box

In science and engineering, a **black box** is a device, system or object which can be viewed solely in terms of its input, output and transfer characteristics without any knowledge of its internal workings, that is, its implementation is "opaque" (black). Almost anything might be referred to as a black box: a transistor, an algorithm, or the human mind.

The opposite of a black box is a system where the inner components or logic are available for inspection, which is sometimes known as a white box, a glass box, or a clear box.

**History**

The modern term "black box" seems to have entered the English language around 1945. The process of network synthesis from the transfer functions of black boxes can be traced to Wilhelm Cauer who published his ideas in their most developed form in 1941.[1] Although Cauer did not himself use the term, others who followed him certainly did describe the method as black-box analysis.[2] Vitold Belevitch[3] puts the concept of black-boxes even earlier, attributing the explicit use of two-port networks as black boxes to Franz Breisig in 1921 and argues that 2-terminal components were implicitly treated as black-boxes before that.

**Examples**

- In electronics, a sealed piece of replaceable equipment; see line-replaceable unit (LRU).
- In computer programming and software engineering, **black box testing** is used to check that the output of a program is as expected, given certain inputs.[4] The term "black box" is used because the actual program being executed is not examined.
- In computing in general, a **black box program** is one where the user cannot see its inner workings (perhaps because it is a closed source program) or one which has no side effects and the function of which need not be examined, a routine suitable for re-use.
- Also in computing, a black box refers to a piece of equipment provided by a vendor, for the purpose of using that vendor's product. It is often the case that the vendor maintains and supports this equipment, and the company receiving the black box typically are hands-off.
- In cybernetics a black box was described by Norbert Wiener as an unknown system that was to be identified using the techniques of system identification.[5] He saw the first step in Self-organization as being to be able to copy the output behaviour of a black box.
- In neural networking or heuristic algorithms (computer terms generally used to describe 'learning' computers or 'AI simulations') a black box is used to describe the constantly changing section of the program environment which cannot easily be tested by the programmers. This is also called a White box (software engineering) in the context that the program code can be seen, but the code is so complex that it might as well be a Black box.
- In finance many people trade with "black box" programs and algorithms designed by programmers.[6] These programs automatically trade user's accounts when certain technical market conditions suddenly exist (such as a SMA crossover).
- In physics, a black box is a system whose internal structure is unknown, or need not be considered for a particular purpose.
- In mathematical modelling, a limiting case.
• In philosophy and psychology, the school of behaviorism sees the human mind as a black box; see black box theory.[7]

• In neorealist international relations theory, the sovereign state is considered generally considered a black box: states are assumed to be unitary, rational, self-interested actors, and the actual decision-making processes of the state are disregarded as being largely irrelevant. Liberal and constructivist theorists often criticize neorealism for the "black box" model, and refer to much of their work on how states arrive at decisions as "breaking open the black box".

• In cryptography to capture the notion of knowledge obtained by an algorithm through the execution of a cryptographic protocol such as a zero-knowledge proof protocol. If the output of the algorithm when interacting with the protocol can be simulated by a simulator that interacts only the algorithm, this means that the algorithm 'cannot know' anything more than the input of the simulator. If the simulator can only interact with the algorithm in a black box way, we speak of a black box simulator.

• In aviation, a "black box" (they are actually bright orange, to facilitate their being found after a crash) is an audio or data recording device in an airplane or helicopter. The cockpit voice recorder records the conversation of the pilots and the flight data recorder logs information about controls and sensors, so that in the event of an accident investigators can use the recordings to assist in the investigation. Although these devices were originally called black boxes for a different reason, they are also an example of a black box according to the meaning above, in that it is of no concern how the recording is actually made.

• In amateur radio the term "black box operator" is a disparaging or self-deprecating description of someone who operates factory made radios without having a good understanding of how they work. Such operators don't build their own equipment (an activity called "homebrewing") or even repair their own "black boxes".[8]

References
Black box theory

Black box theories are things defined only in terms of their function. The term black box theory is applied to any field, philosophy and science or otherwise where some inquiry or definition is made into the relations between the appearance of something (exterior/outside), i.e. here specifically the things black box state, related to its characteristics and behaviour within(interior/inner). Specifically that the inquiry is focused upon a thing that has no immediately apparent characteristics and therefore has only factors for consideration held within itself hidden from immediate observation. The observer is assumed ignorant in the first instance as the majority of available datum is held in a inner situation away from facile investigations. The black box element of the definition is shown as being characterised by a system where observable elements enter a perhaps imaginary box with a set of different outputs emerging which are also observable.

Origin of term

The term black box was first recorded used by the RAF of approximately 1947 to describe the sealed containment used for apparatus of navigation, this usage becoming more widely applied after 1964. The identifier is therefore applied to objects known as the flight data recorder (FDR) and cockpit voice recorder (CVR). These function to record the radio transmissions occurring within an airplane, and are particularly important to persons who engage into an inquiry into the cause of a plane crashing, where the plane is caused to become wreckage. These boxes are in fact coloured orange in order that they be more easily located.

Examples

Considering a black box that could not be opened to "look inside" and see how it worked, all that would be possible would be to guess how it worked based on what happened when something was done to it (input), and what occurred as a result of that (output). If after putting an orange in on one side, an orange fell out the other, it would be possible to make educated guesses or hypotheses on what was happening inside the black box. It could be filled with oranges; it could have a conveyor belt to move the orange from one side to the other; it could even go through an alternate universe. Without being able to investigate the workings of the box, ultimately all we can do is guess.

However, occasionally strange occurrences will take place that change our understanding of the black box. Consider putting in an orange in and having a guava pop out. Now our "filled with oranges" and "conveyor belt" theories no longer work, and we may have to change our educated guess as to how the black box works.

The black box theory of consciousness, which states that the mind is fully understood once the inputs and outputs are well defined, and generally couples this with a radical skepticism regarding the possibility of ever successfully describing the underlying structure, mechanism, and dynamics of the mind.
**Uses**

One of the black box theories uses is as a method to describe/understand psychological factors in fields such as marketing where applied to an analyses of consumer behaviour.\[10][11][12]

**References**

[12] designing of websites (http://designshack.co.uk/articles/business-articles/using-the-black-box-model-to-design-better-websites/)


**Fitness approximation**

In function optimization, **fitness approximation** is a method for decreasing the number of fitness function evaluations to reach a target solution. It belongs to the general class of evolutionary computation or artificial evolution methodologies.

**Approximate models in function optimisation**

**Motivation**

In many real-world optimization problems including engineering problems, the number of fitness function evaluations needed to obtain a good solution dominates the optimization cost. In order to obtain efficient optimization algorithms, it is crucial to use prior information gained during the optimization process. Conceptually, a natural approach to utilizing the known prior information is building a model of the fitness function to assist in the selection of candidate solutions for evaluation. A variety of techniques for constructing of such a model, often also referred to as surrogates, metamodels or approximation models – for computationally expensive optimization problems have been considered.
Approaches

Common approaches to constructing approximate models based on learning and interpolation from known fitness values of a small population include:

- low-degree Polynomials and regression models
- Artificial neural networks including
  - Multilayer perceptrons
  - Radial basis function networks
  - Support vector machines

Due to the limited number of training samples and high dimensionality encountered in engineering design optimization, constructing a globally valid approximate model remains difficult. As a result, evolutionary algorithms using such approximate fitness functions may converge to local optima. Therefore, it can be beneficial to selectively use the original fitness function together with the approximate model.

Adaptive fuzzy fitness granulation

Adaptive fuzzy fitness granulation (AFFG) is a proposed solution to constructing an approximate model of the fitness function in place of traditional computationally expensive large-scale problem analysis like (L-SPA) in the Finite element method or iterative fitting of a Bayesian network structure.

In adaptive fuzzy fitness granulation, an adaptive pool of solutions, represented by fuzzy granules, with an exactly computed fitness function result is maintained. If a new individual is sufficiently similar to an existing known fuzzy granule, then that granule’s fitness is used instead as an estimate. Otherwise, that individual is added to the pool as a new fuzzy granule. The pool size as well as each granule’s radius of influence is adaptive and will grow/shrink depending on the utility of each granule and the overall population fitness. To encourage fewer function evaluations, each granule’s radius of influence is initially large and is gradually shrunk in latter stages of evolution. This encourages more exact fitness evaluations when competition is fierce among more similar and converging solutions. Furthermore, to prevent the pool from growing too large, granules that are not used are gradually eliminated.

Actually AFFG mirrors two features of human cognition: (a) granularity (b) similarity analysis. This granulation-based fitness approximation scheme is applied to solve various engineering optimization problems including detecting hidden information from a watermarked signal in addition to several structural optimization problems.

References

- The cyber shack of Adaptive Fuzzy Fitness Granulation (AFFG) (http://www.davarynejad.com/Mohsen/index.php?n=Main.AFFG) That is designed to accelerate the convergence rate of EAs.
- A complete list of references on Fitness Approximation in Evolutionary Computation (http://www.soft-computing.de/amec_n.html), by Yaochu Jin (http://www.soft-computing.de/jin.html).
Effective fitness

In natural evolution and artificial evolution (e.g. artificial life and evolutionary computation) the fitness (or performance or objective measure) of a schema is rescaled to give its *effective fitness* which takes into account crossover and mutation. That is *effective fitness* can be thought of as the fitness that the schema would need to have in order to increase or decrease as a fraction of the population as it actually does with crossover and mutation present but as if they were not.

References

• Foundations of Genetic Programming [1]

References


Speciation (genetic algorithm)

*Speciation* is a process that occurs naturally in evolution and is modeled explicitly in some genetic algorithms. Speciation in nature occurs when two similar reproducing beings evolve to become too dissimilar to share genetic information effectively or correctly. In the case of living organisms, they are incapable of mating to produce offspring. Interesting special cases of different species being able to breed exist, such as a horse and a donkey mating to produce a mule. However in this case the Mule is usually infertile, and so the genetic isolation of the two parent species is maintained.

In implementations of genetic search algorithms, the event of speciation is defined by some mathematical function that describes the similarity between two candidate solutions (usually described as individuals) in the population. If the result of the similarity is too low, the crossover operator is disallowed between those individuals.
Genetic representation

**Genetic representation** is a way of representing solutions/individuals in evolutionary computation methods. Genetic representation can encode appearance, behavior, physical qualities of individuals. Designing a good genetic representation that is expressive and evolvable is a hard problem in evolutionary computation. Difference in genetic representations is one of the major criteria drawing a line between known classes of evolutionary computation.

Genetic algorithms use linear binary representations. The most standard one is an array of bits. Arrays of other types and structures can be used in essentially the same way. The main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size. This facilitates simple crossover operation. Variable length representations were also explored in Genetic algorithms, but crossover implementation is more complex in this case.

Evolution strategy uses linear real-valued representations, e.g. an array of real values. It uses mostly gaussian mutation and blending/averaging crossover.

Genetic programming (GP) pioneered tree-like representations and developed genetic operators suitable for such representations. Tree-like representations are used in GP to represent and evolve functional programs with desired properties.[1]

Human-based genetic algorithm (HBGA) offers a way to avoid solving hard representation problems by outsourcing all genetic operators to outside agents, in this case, humans. The algorithm has no need for knowledge of a particular fixed genetic representation as long as there are enough external agents capable of handling those representations, allowing for free-form and evolving genetic representations.

**Common genetic representations**

- binary array
- binary tree
- genetic tree
- natural language
- parse tree

**References and notes**

Stochastic universal sampling


SUS is a development of fitness proportionate selection which exhibits no bias and minimal spread. Where fitness proportionate selection chooses several solutions from the population by repeated random sampling, SUS uses a single random value to sample all of the solutions by choosing them at evenly spaced intervals. Described as an algorithm, pseudocode for SUS looks like:

\begin{verbatim}
RWS(population, f)
    Ptr := 0
    for p in population
        if Ptr < f and Ptr + fitness of p > f
            return p
        Ptr := Ptr + fitness of p

SUS(population, N)
    F := total fitness of population
    Start := random number between 0 and F/N
    Ptrs := [Start + i*F/N "'i'" in [0..'N'"'i'"-1]
    return [RWS(i) | i in Ptrs]
\end{verbatim}

Here "RWS" describes the bulk of fitness proportionate selection (also known as "roulette wheel selection") - in true fitness proportional selection the parameter \( f \) is always a random number from 0 to \( F \). The algorithm above is very inefficient both for fitness proportionate and stochastic universal sampling, and is intended to be illustrative rather than canonical.

References
Quality control and genetic algorithms

The combination of quality control and genetic algorithms led to novel solutions of complex quality control design and optimization problems. Quality control is a process by which entities review the quality of all factors involved in production. Quality is the degree to which a set of inherent characteristics fulfills a need or expectation that is stated, general implied or obligatory. Genetic algorithms are search algorithms, based on the mechanics of natural selection and natural genetics.

Quality control

Alternative quality control (QC) procedures can be applied on a process to test statistically the null hypothesis, that the process conforms to the quality requirements, therefore that the process is in control, against the alternative, that the process is out of control. When a true null hypothesis is rejected, a statistical type I error is committed. We have then a false rejection of a run of the process. The probability of a type I error is called probability of false rejection. When a false null hypothesis is accepted, a statistical type II error is committed. We fail then to detect a significant change in the process. The probability of rejection of a false null hypothesis equals the probability of detection of the nonconformity of the process to the quality requirements.

The QC procedure to be designed or optimized can be formulated as:

\[ Q_1(n_1, X_1) \# Q_2(n_2, X_2) \# \ldots \# Q_q(n_q, X_q) \]  

where \( Q_i(n_i, X_i) \) denotes a statistical decision rule, \( n_i \) denotes the size of the sample \( S_i \), that is the number of the samples the rule is applied upon, and \( X_i \) denotes the vector of the rule specific parameters, including the decision limits. Each symbol \( \# \) denotes either the Boolean operator AND or the operator OR. Obviously, for \( \# \) denoting AND, and for \( n_1 < n_2 < \ldots < n_q \), that is for \( S_1 \subset S_2 \subset \ldots \subset S_q \), the \( (1) \) denotes a \( q \)-sampling QC procedure.

Each statistical decision rule is evaluated by calculating the respective statistic of a monitored variable of samples taken from the process. Then, if the statistic is out of the interval between the decision limits, the decision rule is considered to be true. Many statistics can be used, including the following: a single value of the variable of a sample, the range, the mean, and the standard deviation of the values of the variable of the samples, the cumulative sum, the smoothed mean, and the smoothed standard deviation. Finally, the QC procedure is evaluated as a Boolean proposition. If it is true, then the null hypothesis is considered to be false, the process is considered to be out of control, and the run is rejected.

A quality control procedure is considered to be optimum when it minimizes (or maximizes) a context specific objective function. The objective function depends on the probabilities of detection of the nonconformity of the process and of false rejection. These probabilities depend on the parameters of the quality control procedure \( (1) \) and on the probability density functions (see probability density function) of the monitored variables of the process.

Genetic algorithms

Genetic algorithms are robust search algorithms, that do not require knowledge of the objective function to be optimized and search through large spaces quickly. Genetic algorithms have been derived from the processes of the molecular biology of the gene and the evolution of life. Their operators, cross-over, mutation, and reproduction, are isomorphic with the synonymous biological processes. Genetic algorithms have been used to solve a variety of complex optimization problems. Additionally the classifier systems and the genetic programming paradigm have shown us that genetic algorithms can be used for tasks as complex as the program induction.
Quality control and genetic algorithms

In general, we can not use algebraic methods to optimize the quality control procedures. Usage of enumerative methods would be very tedious, especially with multi-rule procedures, as the number of the points of the parameter space to be searched grows exponentially with the number of the parameters to be optimized. Optimization methods based on the genetic algorithms offer an appealing alternative.

Furthermore, the complexity of the design process of novel quality control procedures is obviously greater than the complexity of the optimization of predefined ones.


References

\cite{Hoyle2001} Hoyle D. ISO 9000 quality systems handbook. Butterworth-Heineman 2001;p.654
\cite{Duncan1986} Duncan AJ. Quality control and industrial statistics. Irwin 1986;pp.1-1123.
\cite{Holland1975} Holland, JH. Adaptation in natural and artificial systems. The University of Michigan Press 1975;pp.1-228.
\cite{He2006} He D, Grigoryan A. Joint statistical design of double sampling x and s charts. European Journal of Operational Research 2006;168:122-142.

External links

- American Society for Quality (ASQ) (http://www.asq.org/index.html)
- Illinois Genetic Algorithms Laboratory (IlliGAL) (http://www.illigal.uiuc.edu/web/)
- Hellenic Complex Systems Laboratory (HCSL) (http://www.hcsl.com)
**Human-based genetic algorithm**

In evolutionary computation, a human-based genetic algorithm (HBGA) is a genetic algorithm that allows humans to contribute solution suggestions to the evolutionary process. For this purpose, a HBGA has human interfaces for initialization, mutation, and recombinant crossover. As well, it may have interfaces for selective evaluation. In short, a HBGA outsources the operations of a typical genetic algorithm to humans.

**Evolutionary genetic systems and human agency**

Among evolutionary genetic systems, HBGA is the computer-based analogue of genetic engineering (Allan, 2005). This table compares systems on lines of human agency:

<table>
<thead>
<tr>
<th>system</th>
<th>sequences</th>
<th>innovator</th>
<th>selector</th>
</tr>
</thead>
<tbody>
<tr>
<td>natural selection</td>
<td>nucleotide</td>
<td>nature</td>
<td>nature</td>
</tr>
<tr>
<td>artificial selection</td>
<td>nucleotide</td>
<td>nature</td>
<td>human</td>
</tr>
<tr>
<td>genetic engineering</td>
<td>nucleotide</td>
<td>human</td>
<td>human</td>
</tr>
<tr>
<td>human-based genetic algorithm</td>
<td>data</td>
<td>human</td>
<td>human</td>
</tr>
<tr>
<td>interactive genetic algorithm</td>
<td>data</td>
<td>computer</td>
<td>human</td>
</tr>
<tr>
<td>genetic algorithm</td>
<td>data</td>
<td>computer</td>
<td>computer</td>
</tr>
</tbody>
</table>

One obvious pattern in the table is the division between organic (top) and computer systems (bottom). Another is the vertical symmetry between autonomous systems (top and bottom) and human-interactive systems (middle).

Looking to the right, the selector is the agent that decides fitness in the system. It determines which variations will reproduce and contribute to the next generation. In natural populations, and in genetic algorithms, these decisions are automatic; whereas in typical HBGA systems, they are made by people.

The innovator is the agent of genetic change. The innovator mutates and recombines the genetic material, to produce the variations on which the selector operates. In most organic and computer-based systems (top and bottom), innovation is automatic, operating without human intervention. In HBGA, the innovators are people.

HBGA is roughly similar to genetic engineering. In both systems, the innovators and selectors are people. The main difference lies in the genetic material they work with: electronic data vs. polynucleotide sequences.

**Differences from a plain genetic algorithm**

- All four genetic operators (initialization, mutation, crossover, and selection) can be delegated to humans using appropriate interfaces (Kosorukoff, 2001).
- Initialization is treated as an operator, rather than a phase of the algorithm. This allows a HBGA to start with an empty population. Initialization, mutation, and crossover operators form the group of innovation operators.
- Choice of genetic operator may be delegated to humans as well, so they are not forced to perform a particular operation at any given moment.
Functional features

• HBGA is a method of collaboration and knowledge exchange. It merges competence of its human users creating a kind of symbiotic human-machine intelligence (see also distributed artificial intelligence).

• Human innovation is facilitated by sampling solutions from population, associating and presenting them in different combinations to a user (see creativity techniques).

• HBGA facilitates consensus and decision making by integrating individual preferences of its users.

• HBGA makes use of a cumulative learning idea while solving a set of problems concurrently. This allows to achieve synergy because solutions can be generalized and reused among several problems. This also facilitates identification of new problems of interest and fair-share resource allocation among problems of different importance.

• The choice of genetic representation, a common problem of genetic algorithms, is greatly simplified in HBGA, since the algorithm need not be aware of the structure of each solution. In particular, HBGA allows natural language to be a valid representation.

• Storing and sampling population usually remains an algorithmic function.

• A HBGA is usually a multi-agent system, delegating genetic operations to multiple agents (humans).

Applications

• Evolutionary knowledge management, integration of knowledge from different sources.

• Social organization, collective decision-making, and e-governance.

• Traditional areas of application of interactive genetic algorithms: computer art, user-centered design, etc.

• Collaborative problem solving using natural language as a representation.

The HBGA methodology was derived in 1999-2000 from analysis of the Free Knowledge Exchange project that was launched in the summer of 1998, in Russia (Kosorukoff, 1999). Human innovation and evaluation were used in support of collaborative problem solving. Users were also free to choose the next genetic operation to perform. Currently, several other projects implement the same model, the most popular being Yahoo! Answers, launched in December 2005.

Recent research suggests that human-based innovation operators are advantageous not only where it is hard to design an efficient computational mutation and/or crossover (e.g. when evolving solutions in natural language), but also in the case where good computational innovation operators are readily available, e.g. when evolving an abstract picture or colors (Cheng and Kosorukoff, 2004). In the latter case, human and computational innovation can complement each other, producing cooperative results and improving general user experience by ensuring that spontaneous creativity of users will not be lost.

References

• Kosorukoff, Alex (1999). Free knowledge exchange. internet archive [1]

• Kosorukoff, Alex (2000). Human-based genetic algorithm. online [2]


Interactive evolutionary computation

Interactive evolutionary computation (IEC) or aesthetic selection is a general term for methods of evolutionary computation that use human evaluation. Usually human evaluation is necessary when the form of fitness function is not known (for example, visual appeal or attractiveness; as in Dawkins, 1986) or the result of optimization should fit a particular user preference (for example, taste of coffee or color set of the user interface).

IEC design issues

The number of evaluations that IEC can receive from one human user is limited by user fatigue which was reported by many researchers as a major problem. In addition, human evaluations are slow and expensive as compared to fitness function computation. Hence, one-user IEC methods should be designed to converge using a small number of evaluations, which necessarily implies very small populations. Several methods were proposed by researchers to speed up convergence, like interactive constrain evolutionary search (user intervention) or fitting user preferences using a convex function (Takagi, 2001). IEC human-computer interfaces should be carefully designed in order to reduce user fatigue.

However IEC implementations that can concurrently accept evaluations from many users overcome the limitations described above. An example of this approach is an interactive media installation by Karl Sims that allows to accept preference from many visitors by using floor sensors to evolve attractive 3D animated forms. Some of these multi-user IEC implementations serve as collaboration tools, for example HBGA.

IEC types

IEC methods include interactive evolution strategy (Herdy, 1997), interactive genetic algorithm (Caldwell, 1991), interactive genetic programming (Sims, 1991; Unemi, 2000), and human-based genetic algorithm (Kosorukoff, 2001).

IGA

An interactive genetic algorithm (IGA) is defined as a genetic algorithm that uses human evaluation. These algorithms belong to a more general category of Interactive evolutionary computation. The main application of these techniques include domains where it is hard or impossible to design a computational fitness function, for example, evolving images, music, various artistic designs and forms to fit a user's aesthetic preferences. Interactive computation methods can use different representations, both linear (as in traditional genetic algorithms) and tree-like ones (as in genetic programming).
References


External links

- EndlessForms.com [3], Collaborative interactive evolution allowing you to evolve 3D objects and have them 3D printed.
- An online interactive demonstrator to do Evolutionary Computation step by step. [5]
- Galapagos by Karl Sims [7]
- E-volver [8]
- SBART, a program to evolve 2D images [9]
- GenJam (Genetic Jammer) [10]
- Evolutionary music [11]
- Darwin poetry [12]
- Takagi Lab at Kyushu University [13]
- idiofact.de [14], Webpage that uses interactive evolutionary computation with a generative design algorithm to generate 2d images.
- Picbreeder service [15], Collaborative interactive evolution allowing branching from other users' creations that produces pictures like faces and spaceships.
- Peer to Peer IGA [16] Using collaborative IGA sessions for floorplanning and document design.
Interactive evolutionary computation

References


Genetic programming

In artificial intelligence, genetic programming (GP) is an evolutionary algorithm-based methodology inspired by biological evolution to find computer programs that perform a user-defined task. It is a specialization of genetic algorithms (GA) where each individual is a computer program. It is a machine learning technique used to optimize a population of computer programs according to a fitness landscape determined by a program's ability to perform a given computational task.

History

In 1954, GP began with the evolutionary algorithms first used by Nils Aall Barricelli applied to evolutionary simulations. In the 1960s and early 1970s, evolutionary algorithms became widely recognized as optimization methods. Ingo Rechenberg and his group were able to solve complex engineering problems through evolution strategies as documented in his 1971 PhD thesis and the resulting 1973 book. John Holland was highly influential during the 1970s.

In 1964, Lawrence J. Fogel, one of the earliest practitioners of the GP methodology, applied evolutionary algorithms to the problem of discovering finite-state automata. Later GP-related work grew out of the learning classifier system community, which developed sets of sparse rules describing optimal policies for Markov decision processes. The first statement of modern “tree-based” Genetic Programming (that is, procedural languages organized in tree-based structures and operated on by suitably defined GA-operators) was given by Nichael L. Cramer (1985).[1] This work was later greatly expanded by John R. Koza, a main proponent of GP who has pioneered the application of genetic programming in various complex optimization and search problems.[2]

In the 1990s, GP was mainly used to solve relatively simple problems because it is very computationally intensive. Recently GP has produced many novel and outstanding results in areas such as quantum computing, electronic design, game playing, sorting, and searching, due to improvements in GP technology and the exponential growth in CPU power.[3] These results include the replication or development of several post-year-2000 inventions. GP has also been applied to evolvable hardware as well as computer programs.

Developing a theory for GP has been very difficult and so in the 1990s GP was considered a sort of outcast among search techniques. But after a series of breakthroughs in the early 2000s, the theory of GP has had a formidable and rapid development. So much so that it has been possible to build exact probabilistic models of GP (schema theories, Markov chain models and meta-optimization algorithms).
Genetic programming

Chromosome representation

GP evolves computer programs, traditionally represented in memory as tree structures.[4] 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)]. The commercial GP software Discipulus, uses AIM, automatic induction of binary machine code[5] to achieve better performance. µGP[6] uses directed multigraphs to generate programs that fully exploit the syntax of a given assembly language.

Genetic operators

The main operators used in evolutionary algorithms such as GP are crossover and mutation.

Crossover

Crossover is applied on an individual by simply switching one of its nodes with another node from another individual in the population. With a tree-based representation, replacing a node means replacing the whole branch. This adds greater effectiveness to the crossover operator. The expressions resulting from crossover are very much different from their initial parents.

Mutation

Mutation affects an individual in the population. It can replace a whole node in the selected individual, or it can replace just the node's information. To maintain integrity, operations must be fail-safe or the type of information the node holds must be taken into account. For example, mutation must be aware of binary operation nodes, or the operator must be able to handle missing values.

Other approaches

The basic ideas of genetic programming have been modified and extended in a variety of ways:

- Extended Compact Genetic Programming (ECGP)
- Embedded Cartesian Genetic Programming (ECGP)
- Probabilistic Incremental Program Evolution (PIPE)

MOSES

Meta-Optimizing Semantic Evolutionary Search (MOSES) is a meta-programming technique for evolving programs by iteratively optimizing genetic populations.[7] It has been shown to strongly outperform genetic and evolutionary program learning systems, and has been successfully applied to many real-world problems, including computational biology, sentiment evaluation, and agent control.[8] When applied to supervised classification problems, MOSES performs as well as, or better than support vector machines (SVM), while offering more insight into the structure of the data, as the resulting program demonstrates dependencies and is understandable in a way that a large vector of
MOSES is able to out-perform standard GP systems for two important reasons. One is that it uses estimation of
distribution algorithms (EDA) to determine the Markov blanket (that is, the dependencies in a Bayesian network)
between different parts of a program. This quickly rules out pointless mutations that change one part of a program
without making corresponding changes in other, related parts of the program. The other is that it performs reduction
to reduce programs to normal form at each iteration stage, thus making programs smaller, more compact, faster to
execute, and more human readable. Besides avoiding spaghetti code, normalization removes redundancies in
programs, thus allowing smaller populations of less complex programs, speeding convergence.

Meta-Genetic Programming

Meta-Genetic Programming is the proposed meta learning technique of evolving a genetic programming system
using genetic programming itself. It suggests that chromosomes, crossover, and mutation were themselves evolved,
therefore like their real life counterparts should be allowed to change on their own rather than being determined by a
human programmer. Meta-GP was formally proposed by Jürgen Schmidhuber in 1987,[9] but some earlier efforts
may be considered instances of the same technique, including Doug Lenat's Eurisko. It is a recursive but terminating
algorithm, allowing it to avoid infinite recursion.

Critics of this idea often say this approach is overly broad in scope. However, it might be possible to constrain the
fitness criterion onto a general class of results, and so obtain an evolved GP that would more efficiently produce
results for sub-classes. This might take the form of a Meta evolved GP for producing human walking algorithms
which is then used to evolve human running, jumping, etc. The fitness criterion applied to the Meta GP would
simply be one of efficiency.

For general problem classes there may be no way to show that Meta GP will reliably produce results more efficiently
than a created algorithm other than exhaustion. The same holds for standard GP and other search algorithms.

Implementations

Possibly most used:

• ECJ - Evolutionary Computation/Genetic Programming research system [10] (Java)
• Beagle - Open BEAGLE, a versatile EC framework [12] (C++ with STL)
• EO Evolutionary Computation Framework [13] (C++ with static polymorphism)

Other:

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Description</th>
<th>License</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>EvoJ [18]</td>
<td>Evolutionary computations framework</td>
<td>Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License [19]</td>
<td>Java</td>
</tr>
<tr>
<td>TinyGP [23]</td>
<td>A tiny genetic programming system.</td>
<td></td>
<td>C and Java</td>
</tr>
<tr>
<td>Software</td>
<td>Description</td>
<td>License</td>
<td>Language</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>pySTEP [27]</td>
<td>Python Strongly Typed Genetic Programming</td>
<td>MIT License [28]</td>
<td>Python</td>
</tr>
<tr>
<td>JAGA [31]</td>
<td>A Genetic Programming Package with support for Automatically Defined Functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RMIT GP [32]</td>
<td>Framework for conducting experiments in Genetic Programming</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPE [33]</td>
<td>simple Genetic Programming research system</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DGPF [34]</td>
<td>Java Genetic Algorithms and Genetic Programming, an open-source framework</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n-genes [36]</td>
<td>Java Genetic Algorithms and Genetic Programming (stack oriented) framework</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMDGP [37]</td>
<td>object oriented framework for solving genetic programming problems</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRP [38]</td>
<td>Directed Ruby Programming, Genetic Programming &amp; Grammatical Evolution Library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPLAB [39]</td>
<td>A Genetic Programming Toolbox for MATLAB</td>
<td></td>
<td>MATLAB</td>
</tr>
<tr>
<td>GPTIPS [40]</td>
<td>Genetic Programming Tool for MATLAB, aimed at performing multigene symbolic regression</td>
<td></td>
<td>MATLAB</td>
</tr>
<tr>
<td>PyRobot [41]</td>
<td>Evolutionary Algorithms (GA + GP) Modules, Open Source</td>
<td></td>
<td>Python</td>
</tr>
<tr>
<td>Discipulus [43]</td>
<td>Commercial Genetic Programming Software from RML Technologies, Inc</td>
<td></td>
<td>Generates code in most high level languages</td>
</tr>
<tr>
<td>GAlib [44]</td>
<td>Object oriented framework with 4 different GA implementations and 4 representation types (arbitrary derivations possible)</td>
<td>GAlib License [45]</td>
<td>C++</td>
</tr>
<tr>
<td>Java GALib [46]</td>
<td>Source Forge open source Java genetic algorithm library, complete with Javadocs and examples (see bottom of page)</td>
<td></td>
<td>Java</td>
</tr>
<tr>
<td>PushGP [48]</td>
<td>a strongly typed, stack-based genetic programming system that allows GP to manipulate its own code (auto-constructive evolution)</td>
<td></td>
<td>Java / C++ / Javascript / Scheme / Clojure / Lisp</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Language</td>
<td>License</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------</td>
<td>------------------</td>
</tr>
<tr>
<td>GEVA</td>
<td>Grammatical Evolution in Java</td>
<td>Java</td>
<td>GNU GPL [17]</td>
</tr>
<tr>
<td>jGE</td>
<td>Java Grammatical Evolution</td>
<td>Java</td>
<td>GNU GPL v3 [17]</td>
</tr>
<tr>
<td>ECF</td>
<td>Evolutionary Computation Framework, different genotypes, parallel algorithms, tutorial</td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>JCLEC</td>
<td>Evolutionary Computation Library in Java, expression tree encoding, syntax tree encoding</td>
<td>Java</td>
<td>GNU GPL [17]</td>
</tr>
<tr>
<td>HeuristicLab</td>
<td>A Paradigm-Independent and Extensible Environment for Heuristic Optimization, rich graphical user interface, open source, plugin-based architecture</td>
<td>C#</td>
<td></td>
</tr>
<tr>
<td>PolyGP</td>
<td>Strong typing and lambda abstractions</td>
<td>Haskell</td>
<td></td>
</tr>
<tr>
<td>PonyGE</td>
<td>a small, one source file implementation of GE, with an interactive graphics demo application</td>
<td>Python</td>
<td>GNU GPL v3 [17]</td>
</tr>
<tr>
<td>MicroGP (uGP)</td>
<td>General purpose tool, mostly exploited for assembly language generation</td>
<td>C++</td>
<td></td>
</tr>
</tbody>
</table>

NB. You should check the license and copyright terms on the program/library website before use.

References and notes

[19] http://creativecommons.org/licenses/by-nc-sa/3.0/legalcode
Bibliography

- Holland, John H (1975), *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor
used as a draft to his 1992 book.


**External links**

- DigitalBiology.NET (http://www.digitalbiology.net/) Vertical search engine for GA/GP resources
- Aymen S Saket & Mark C Sinclair (http://web.archive.org/web/20070813222058/http://uk.geocities.com/markcsinclair/abstracts.html#pro00a/)
- GP bibliography (http://www.cs.bham.ac.uk/~wbl/biblio/README.html)
- People who work on GP (http://www.cs.ucl.ac.uk/staff/W.Langdon/homepages.html)
Gene expression programming

Gene Expression Programming (GEP) is an evolutionary algorithm that evolves populations of computer programs in order to solve a user-defined problem. GEP has similarities, but is distinct from, the evolutionary computational method of genetic programming. In genetic programming the individuals comprising a population are typically symbolic expression trees; however, the individuals comprising a population of GEP are encoded as linear chromosomes, which are then translated into expression trees. The important difference is that the recombination operators of genetic programming operate directly on the tree structure (e.g. swapping sub-trees), whereas the recombination operators of gene expression programming operate directly on the linear encoding (i.e. before it is translated into a tree). As such, after recombination, the modified portions of the resulting expression trees often bear little semblance to their direct ancestors.

The expression trees are themselves computer programs evolved to solve a particular problem and are selected according to their performance/fitness in solving the problem at hand. After repeated iteration, populations of such computer programs will ideally discover new traits and become better adapted to a particular selection environment. The desired endpoint of the algorithm is that a good solution has been evolved by the evolutionary process.

Cândida Ferreira, the inventor of the technique, claims that GEP significantly surpasses the traditional genetic programming approach for a number of benchmark problems. She attributes the alleged speed increase to the separate genotype/phenotype representation and the inherently multigenic organization of GEP chromosomes.

For further details of GEP see the GEP paper published in *Complex Systems*, where the algorithm is described and applied to a set of problems including symbolic regression, Boolean concept learning, and cellular automata.

Further reading


References

- GEP home page [3]

References

Grammatical evolution


It is related to the idea of genetic programming in that the objective is to find an executable program or program fragment, that will achieve a good fitness value for the given objective function. In most published work on Genetic Programming, a LISP-style tree-structured expression is directly manipulated, whereas Grammatical Evolution applies genetic operators to an integer string, subsequently mapped to a program (or similar) through the use of a grammar. One of the benefits of GE is that this mapping simplifies the application of search to different programming languages and other structures.

Problem addressed

In type-free conventional, Koza/Cramer-style GP, the function set must meet the requirement of closure: all functions must be capable of accepting as their arguments the output of all other functions in the function set. Usually, this is implemented by dealing with a single data-type such as double-precision floating point. Whilst modern Genetic Programming frameworks supporting typing, such type-systems have limitations that Grammatical Evolution does not suffer from.

GE's solution

GE offers a solution to this issue by evolving solutions according to a user-specified grammar (usually a grammar in Backus-Naur form). Therefore the search space can be restricted, and domain knowledge of the problem can be incorporated. The inspiration for this approach comes from a desire to separate the "genotype" from the "phenotype": in GP, the objects the search algorithm operates on and what the fitness evaluation function interprets are one and the same. In contrast, GE's "genotypes" are ordered lists of integers which code for selecting rules from the provided context-free grammar. The phenotype, however, is the same as in Koza/Cramer-style GP: a tree-like structure that is evaluated recursively. This is more in line with how genetics work in nature, where there is a separation between an organism's genotype and that expression in proteins and the like.

GE has a modular approach to it. In particular, the search portion of the GE paradigm needn't be carried out by any one particular algorithm or method. Observe that the objects GE performs search on are the same as that used in genetic algorithms. This means, in principle, that any existing genetic algorithm package, such as the popular GAlib[44], can be used to carry out the search, and a developer implementing a GE system need only worry about carrying out the mapping from list of integers to program tree. It is also in principle possible to perform the search using some other method, such as particle swarm optimization (see the remark below); the modular nature of GE creates many opportunities for hybrids as the problem of interest to be solved dictates.

Brabazon and O'Hehir have successfully applied GE to predicting corporate bankruptcy, forecasting stock indices, bond credit ratings, and other financial applications.

It is possible to structure a GE grammar that for a given function/terminal set is equivalent to genetic programming.
Criticism

Despite its successes, GE has been the subject of some criticism. One issue is that as a result of its mapping operation, GE's genetic operators do not achieve high locality\cite{3}\cite{4} which is a highly regarded property of genetic operators in evolutionary algorithms.\cite{3}

Variants

Although GE is fairly new, there are already enhanced versions and variants that have been worked out. GE researchers have experimented with using particle swarm optimization to carry out the searching instead of genetic algorithms with results comparable to that of normal GE; this is referred to as a "grammatical swarm"; using only the basic PSO model it has been found that PSO is probably equally capable of carrying out the search process in GE as simple genetic algorithms are. (Although PSO is normally a floating-point search paradigm, it can be discretized, e.g., by simply rounding each vector to the nearest integer, for use with GE.)

Yet another possible variation that has been experimented with in the literature is attempting to encode semantic information in the grammar in order to further bias the search process.

Notes

\cite{1} http://www.grammaticalevolution.org/eurogp98.ps
\cite{2} http://bds.ul.ie
\cite{3} http://www.springerlink.com/content/0125627ls52766534/
\cite{4} http://www.cs.kent.ac.uk/pubs/2010/3004/index.html

Resources

- An Open Source C++ implementation (http://www.grammaticalevolution.org/libGE) of GE was funded by the Science Foundation of Ireland (http://www.sfi.ie).
- Grammatical Evolution in Java (http://ncra.ucd.ie/geva).
- jGE - Java Grammatical Evolution (http://www.hangor.ac.uk/~eep201/jge).
- DRP (http://drp.rubyforge.org/), Directed Ruby Programming, is an experimental system designed to let users create hybrid GE/GP systems. It is implemented in pure Ruby.
- GERET (http://geret.org/), Grammatical Evolution Ruby Exploratory Toolkit.
Grammar induction

Grammatical induction, also known as grammatical inference or syntactic pattern recognition, refers to the process in machine learning of learning a formal grammar (usually in the form of re-write rules or productions) from a set of observations, thus constructing a model which accounts for the characteristics of the observed objects. Grammatical inference is distinguished from traditional decision rules and other such methods principally by the nature of the resulting model, which in the case of grammatical inference relies heavily on hierarchical substitutions. Whereas a traditional decision rule set is geared toward assessing object classification, a grammatical rule set is geared toward the generation of examples. In this sense, the grammatical induction problem can be said to seek a generative model, while the decision rule problem seeks a descriptive model.

Methodologies

There are a wide variety of methods for grammatical inference. Two of the classic sources are Fu (1977) and Fu (1982). Duda, Hart & Stork (2001) also devote a brief section to the problem, and cite a number of references. The basic trial-and-error method they present is discussed below.

Grammatical inference by trial-and-error

The method proposed in Section 8.7 of Duda, Hart & Stork (2001) suggests successively guessing grammar rules (productions) and testing them against positive and negative observations. The rule set is expanded so as to be able to generate each positive example, but if a given rule set also generates a negative example, it must be discarded. This particular approach can be characterized as "hypothesis testing" and bears some similarity to Mitchel's version space algorithm. The Duda, Hart & Stork (2001) text provide a simple example which nicely illustrates the process, but the feasibility of such an unguided trial-and-error approach for more substantial problems is dubious.

Grammatical inference by genetic algorithms

Grammatical Induction using evolutionary algorithms is the process of evolving a representation of the grammar of a target language through some evolutionary process. Formal grammars can easily be represented as a tree structure of production rules that can be subjected to evolutionary operators. Algorithms of this sort stem from the genetic programming paradigm pioneered by John Koza. Other early work on simple formal languages used the binary string representation of genetic algorithms, but the inherently hierarchical structure of grammars couched in the EBNF language made trees a more flexible approach.

Koza represented Lisp programs as trees. He was able to find analogues to the genetic operators within the standard set of tree operators. For example, swapping sub-trees is equivalent to the corresponding process of genetic crossover, where sub-strings of a genetic code are transplanted into an individual of the next generation. Fitness is measured by scoring the output from the functions of the lisp code. Similar analogues between the tree structured lisp representation and the representation of grammars as trees, made the application of genetic programming techniques possible for grammar induction.

In the case of Grammar Induction, the transplantation of sub-trees corresponds to the swapping of production rules that enable the parsing of phrases from some language. The fitness operator for the grammar is based upon some measure of how well it performed in parsing some group of sentences from the target language. In a tree representation of a grammar, a terminal symbol of a production rule corresponds to a leaf node of the tree. Its parent nodes corresponds to a non-terminal symbol (e.g. a noun phrase or a verb phrase) in the rule set. Ultimately, the root node might correspond to a sentence non-terminal.
Grammar induction

Grammatical inference by greedy algorithms

Like all greedy algorithms, greedy grammar inference algorithms make, in iterative manner, decisions that seem to be the best at that stage. These made decisions deal usually with things like the making of a new or the removing of the existing rules, the choosing of the applied rule or the merging of some existing rules. Because there are several ways to define 'the stage' and 'the best', there are also several greedy grammar inference algorithms.

These context-free grammar generating algorithms make the decision after every read symbol:

- Lempel-Ziv-Welch algorithm creates a context-free grammar in a deterministic way such that it is necessary to store only the start rule of the generated grammar.
- Sequitur and its modifications.

These context-free grammar generating algorithms first read the whole given symbol-sequence and then start to make decisions:

- Byte pair encoding and its optimizations.

Applications

The principle of grammar induction has been applied to other aspects of natural language processing, and have been applied (among many other problems) to morpheme analysis, and even place name derivations. Grammar induction has also been used for lossless data compression and statistical inference via MML and MDL principles.

References

- Duda, Richard O.; Hart, Peter E.; Stork, David G. (2001), *Pattern Classification*[^1], New York: John Wiley & Sons
- *Syntactic Pattern Recognition, Applications*, Berlin: Springer-Verlag, 1977
- Gold, E Mark (1967), *Language Identification in the Limit*, Information and Control

References

[^2]: http://proquest.umi.com/pqdl ink/?ver=1&Exp=05-16-2013&FMT=7&DID=757518381&RQT=309&attempts=1&cfc=1
Java Grammatical Evolution

jGE Library

jGE Library is an implementation of Grammatical Evolution in the Java programming language. It was the first published implementation of Grammatical Evolution in this language [1]. Today, another one well-known published Java implementation exists, named GEVA [2]. GEVA developed at UCD’s Natural Computing Research & Applications group under the guidance of one of the inventors of Grammatical Evolution, Dr. Michael O’Neill. [3]

jGE Library aims to provide not only an implementation of Grammatical Evolution, but also a free, open-source, and extendable framework for experimentation in the area of evolutionary computation. Namely, it supports the implementation (through additions and extensions) of any evolutionary algorithm [4]. Furthermore, its extendable architecture and design facilitates the implementation and incorporation of new experimental implementation inspired by natural evolution and biology [5].

The jGE Library binary file, the source code, the documentation, and an extension for the NetLogo modeling environment [6], named jGE NetLogo extension, can be downloaded from the jGE Official Web Site [7].

Web Site

jGE Official Web Site [7]

License

The jGE Library is free software released under the GNU General Public License v3 [8].

jGE Publications


References

"Linear genetic programming" is unrelated to "linear programming".

Linear Genetic Programming (LGP) is a particular subset of genetic programming wherein computer programs in population are represented as a sequence of instructions from imperative programming language or machine language. The graph-based data flow that results from a multiple usage of register contents and the existence of structurally noneffective code (introns) are two main differences to more common tree-based genetic programming (TGP) variant.\[1\][2][3]

Examples of LGP programs

Because LGP programs are basically represented by a linear sequence of instructions, they are simpler to read and to operate on than their tree-based counterparts. For example, a simple program written in the LGP language Slash/A \[4\] looks like a series of instructions separated by a slash:

```
input/  # gets an input from user and saves it to register F
0/     # sets register I = 0
save/  # saves content of F into data vector D[I] (i.e. D[0] := F)
input/ # gets another input, saves to F
add/   # adds to F current data pointed to by I (i.e. D[0] := F)
output/. # outputs result from F
```

By representing such code in bytecode format, i.e. as an array of bytes each representing a different instruction, one can make mutation operations simply by changing an element of such an array.

See

Cartesian genetic programming

Notes


External links

- Slash/A (http://github.com/arturadib/slash-a) A programming language and C++ library specifically designed for linear GP
- DigitalBiology.NET (http://www.digitalbiology.net/) Vertical search engine for GA/GP resources
- Discipulus (http://www.aimlearning.com/) Genetic-Programming Software
- (http://www.genetic-programming.org)
Evolutionary programming

**Evolutionary programming** is one of the four major evolutionary algorithm paradigms. It is similar to genetic programming, but the structure of the program to be optimized is fixed, while its numerical parameters are allowed to evolve.

It was first used by Lawrence J. Fogel in the US in 1960 in order to use simulated evolution as a learning process aiming to generate artificial intelligence. Fogel used finite state machines as predictors and evolved them. Currently evolutionary programming is a wide evolutionary computing dialect with no fixed structure or (representation), in contrast with some of the other dialects. It is becoming harder to distinguish from evolutionary strategies.

Its main variation operator is mutation; members of the population are viewed as part of a specific species rather than members of the same species therefore each parent generates an offspring, using a (μ + μ) survivor selection.

**References**


**External links**

- Evolutionary Programming by Jason Brownlee (PhD)[^4]

**References**

[^2]: http://www.springer.de
[^3]: http://www.aip.de/~ast/EvolCompFAQ/Q1_2.htm
[^4]: http://www.cleveralgorithms.com/nature-inspired/evolution/evolutionary_programming.html
Gaussian adaptation

**Gaussian adaptation (GA)** is an evolutionary algorithm designed for the maximization of manufacturing yield due to statistical deviation of component values of signal processing systems. In short, GA is a stochastic adaptive process where a number of samples of an $n$-dimensional vector $x^T = (x_1, x_2, ..., x_n)$ are taken from a multivariate Gaussian distribution, $N(m, M)$, having mean $m$ and moment matrix $M$. The samples are tested for fail or pass. The first- and second-order moments of the Gaussian restricted to the pass samples are $m^*$ and $M^*$. The outcome of $x$ as a pass sample is determined by a function $s(x)$, $0 < s(x) < q \leq 1$, such that $s(x)$ is the probability that $x$ will be selected as a pass sample. The average probability of finding pass samples (yield) is

$$P(m) = \int s(x) N(x - m) \, dx$$

Then the theorem of GA states:

For any $s(x)$ and for any value of $P < q$, there always exist a Gaussian p. d. f. that is adapted for maximum dispersion. The necessary conditions for a local optimum are $m = m^*$ and $M$ proportional to $M^*$. The dual problem is also solved: $P$ is maximized while keeping the dispersion constant (Kjellström, 1991).

Proofs of the theorem may be found in the papers by Kjellström, 1970, and Kjellström & Taxén, 1981.

Since dispersion is defined as the exponential of entropy/disorder/average information it immediately follows that the theorem is valid also for those concepts. Altogether, this means that Gaussian adaptation may carry out a simultaneous maximisation of yield and average information (without any need for the yield or the average information to be defined as criterion functions).

**The theorem is valid for all regions of acceptability and all Gaussian distributions.** It may be used by cyclic repetition of random variation and selection (like the natural evolution). In every cycle a sufficiently large number of Gaussian distributed points are sampled and tested for membership in the region of acceptability. The centre of gravity of the Gaussian, $m$, is then moved to the centre of gravity of the approved (selected) points, $m^*$. Thus, the process converges to a state of equilibrium fulfilling the theorem. A solution is always approximate because the centre of gravity is always determined for a limited number of points.

It was used for the first time in 1969 as a pure optimization algorithm making the regions of acceptability smaller and smaller (in analogy to simulated annealing, Kirkpatrick 1983). Since 1970 it has been used for both ordinary optimization and yield maximization.

**Natural evolution and Gaussian adaptation**

It has also been compared to the natural evolution of populations of living organisms. In this case $s(x)$ is the probability that the individual having an array $x$ of phenotypes will survive by giving offspring to the next generation; a definition of individual fitness given by Hartl 1981. The yield, $P$, is replaced by the mean fitness determined as a mean over the set of individuals in a large population.

Phenotypes are often Gaussian distributed in a large population and a necessary condition for the natural evolution to be able to fulfill the theorem of Gaussian adaptation, with respect to all Gaussian quantitative characters, is that it may push the centre of gravity of the Gaussian to the centre of gravity of the selected individuals. This may be accomplished by the Hardy–Weinberg law. This is possible because the theorem of Gaussian adaptation is valid for any region of acceptability independent of the structure (Kjellström, 1996).

In this case the rules of genetic variation such as crossover, inversion, transposition etcetera may be seen as random number generators for the phenotypes. So, in this sense Gaussian adaptation may be seen as a genetic algorithm.
How to climb a mountain

Mean fitness may be calculated provided that the distribution of parameters and the structure of the landscape is known. The real landscape is not known, but figure below shows a fictitious profile (blue) of a landscape along a line \( x \) in a room spanned by such parameters. The red curve is the mean based on the red bell curve at the bottom of figure. It is obtained by letting the bell curve slide along the \( x \)-axis, calculating the mean at every location. As can be seen, small peaks and pits are smoothed out. Thus, if evolution is started at A with a relatively small variance (the red bell curve), then climbing will take place on the red curve. The process may get stuck for millions of years at B or C, as long as the hollows to the right of these points remain, and the mutation rate is too small.

If the mutation rate is sufficiently high, the disorder or variance may increase and the parameter(s) may become distributed like the green bell curve. Then the climbing will take place on the green curve, which is even more smoothed out. Because the hollows to the right of B and C have now disappeared, the process may continue up to the peaks at D. But of course the landscape puts a limit on the disorder or variability. Besides — dependent on the landscape — the process may become very jerky, and if the ratio between the time spent by the process at a local peak and the time of transition to the next peak is very high, it may as well look like a punctuated equilibrium as suggested by Gould (see Ridley).

Computer simulation of Gaussian adaptation

Thus far the theory only considers mean values of continuous distributions corresponding to an infinite number of individuals. In reality however, the number of individuals is always limited, which gives rise to an uncertainty in the estimation of \( m \) and \( M \) (the moment matrix of the Gaussian). And this may also affect the efficiency of the process. Unfortunately very little is known about this, at least theoretically.

The implementation of normal adaptation on a computer is a fairly simple task. The adaptation of \( m \) may be done by one sample (individual) at a time, for example

\[
m(i + 1) = (1 - a) \, m(i) + a \, x
\]

where \( x \) is a pass sample, and \( a < 1 \) a suitable constant so that the inverse of \( a \) represents the number of individuals in the population.

\( M \) may in principle be updated after every step \( y \) leading to a feasible point

\[
x = m + y \quad \text{according to:} \\
M(i + 1) = (1 - 2b) \, M(i) + 2 \, b \, y y^\top,
\]

where \( y^\top \) is the transpose of \( y \) and \( b \ll 1 \) is another suitable constant. In order to guarantee a suitable increase of average information, \( y \) should be normally distributed with moment matrix \( \mu^2 M \), where the scalar \( \mu > 1 \) is used to increase average information (information entropy, disorder, diversity) at a suitable rate. But \( M \) will never be used in the calculations. Instead we use the matrix \( W \) defined by \( W W^\top = M \).
Thus, we have $y = Wg$, where $g$ is normally distributed with the moment matrix $\mu U$, and $U$ is the unit matrix. $W$ and $W^T$ may be updated by the formulas

$$W = (1 - b)W + byg^T \quad \text{and} \quad W^T = (1 - b)W^T + by^T$$

because multiplication gives

$$M = (1 - 2b)M + 2byy^T,$$

where terms including $b^2$ have been neglected. Thus, $M$ will be indirectly adapted with good approximation. In practice it will suffice to update $W$ only

$$W(i + 1) = (1 - b)W(i) + byg^T.$$  

This is the formula used in a simple 2-dimensional model of a brain satisfying the Hebbian rule of associative learning; see the next section (Kjellström, 1996 and 1999).

The figure below illustrates the effect of increased average information in a Gaussian p.d.f. used to climb a mountain Crest (the two lines represent the contour line). Both the red and green cluster have equal mean fitness, about 65%, but the green cluster has a much higher average information making the green process much more efficient. The effect of this adaptation is not very salient in a 2-dimensional case, but in a high-dimensional case, the efficiency of the search process may be increased by many orders of magnitude.

The evolution in the brain

In the brain the evolution of DNA-messages is supposed to be replaced by an evolution of signal patterns and the phenotypic landscape is replaced by a mental landscape, the complexity of which will hardly be second to the former. The metaphor with the mental landscape is based on the assumption that certain signal patterns give rise to a better well-being or performance. For instance, the control of a group of muscles leads to a better pronunciation of a word or performance of a piece of music.

In this simple model it is assumed that the brain consists of interconnected components that may add, multiply and delay signal values.

- A nerve cell kernel may add signal values,
- a synapse may multiply with a constant and
- An axon may delay values.

This is a basis of the theory of digital filters and neural networks consisting of components that may add, multiply and delay signal values and also of many brain models, Levine 1991.

In the figure below the brain stem is supposed to deliver Gaussian distributed signal patterns. This may be possible since certain neurons fire at random (Kandel et al.). The stem also constitutes a disordered structure surrounded by more ordered shells (Bergström, 1969), and according to the central limit theorem the sum of signals from many neurons may be Gaussian distributed. The triangular boxes represent synapses and the boxes with the + sign are cell
Gaussian adaptation

In the cortex signals are supposed to be tested for feasibility. When a signal is accepted the contact areas in the synapses are updated according to the formulas below in agreement with the Hebbian theory. The figure shows a 2-dimensional computer simulation of Gaussian adaptation according to the last formula in the preceding section.

\[ m_1 = 0.9 m_1 + 0.1 x_1; \quad m_2 = 0.9 m_2 + 0.1 x_2; \]
\[ w_{11} = 0.9 w_{11} + 0.1 y_1 g_1; \quad w_{12} = 0.9 w_{12} + 0.1 y_1 g_2; \]
\[ w_{21} = 0.9 w_{21} + 0.1 y_2 g_1; \quad w_{22} = 0.9 w_{22} + 0.1 y_2 g_2; \]

As can be seen this is very much like a small brain ruled by the theory of Hebbian learning (Kjellström, 1996, 1999 and 2002).

**Gaussian adaptation and free will**

Gaussian adaptation as an evolutionary model of the brain obeying the Hebbian theory of associative learning offers an alternative view of free will due to the ability of the process to maximize the mean fitness of signal patterns in the brain by climbing a mental landscape in analogy with phenotypic evolution.

Such a random process gives us lots of freedom of choice, but hardly any will. An illusion of will may, however, emanate from the ability of the process to maximize mean fitness, making the process goal seeking. I. e., it prefers higher peaks in the landscape prior to lower, or better alternatives prior to worse. In this way an illusive will may appear. A similar view has been given by Zohar 1990. See also Kjellström 1999.

**A theorem of efficiency for random search**

The efficiency of Gaussian adaptation relies on the theory of information due to Claude E. Shannon (see information content). When an event occurs with probability \( P \), then the information \(-\log(P)\) may be achieved. For instance, if the mean fitness is \( P \), the information gained for each individual selected for survival will be \(-\log(P)\) – on the average - and the work/time needed to get the information is proportional to \(1/P\). Thus, if efficiency, \( E \), is defined as information divided by the work/time needed to get it we have:

\[ E = -P \log(P). \]

This function attains its maximum when \( P = 1/e = 0.37 \). The same result has been obtained by Gaines with a different method.

\( E = 0 \) if \( P = 0 \), for a process with infinite mutation rate, and if \( P = 1 \), for a process with mutation rate = 0 (provided that the process is alive). This measure of efficiency is valid for a large class of random search processes provided that certain conditions are at hand.

1 The search should be statistically independent and equally efficient in different parameter directions. This condition may be approximately fulfilled when the moment matrix of the Gaussian has been adapted for maximum average information to some region of acceptability, because linear transformations of the whole process do not have
an impact on efficiency.

2 All individuals have equal cost and the derivative at $P = 1$ is $< 0$.

Then, the following theorem may be proved:

All measures of efficiency, that satisfy the conditions above, are asymptotically proportional to $-P \log(P/q)$ when the number of dimensions increases, and are maximized by $P = q \exp(-1)$ (Kjellström, 1996 and 1999).

The figure above shows a possible efficiency function for a random search process such as Gaussian adaptation. To the left the process is most chaotic when $P = 0$, while there is perfect order to the right where $P = 1$.

In an example by Rechenberg, 1971, 1973, a random walk is pushed thru a corridor maximizing the parameter $x_1$. In this case the region of acceptability is defined as a $(n - 1)$-dimensional interval in the parameters $x_2, x_3, \ldots, x_n$, but a $x_1$-value below the last accepted will never be accepted. Since $P$ can never exceed 0.5 in this case, the maximum speed towards higher $x_1$-values is reached for $P = 0.5/e = 0.18$, in agreement with the findings of Rechenberg.

A point of view that also may be of interest in this context is that no definition of information (other than that sampled points inside some region of acceptability gives information about the extension of the region) is needed for the proof of the theorem. Then, because, the formula may be interpreted as information divided by the work needed to get the information, this is also an indication that $-\log(P)$ is a good candidate for being a measure of information.

**The Stauffer and Grimson algorithm**

Gaussian adaptation has also been used for other purposes as for instance shadow removal by "The Stauffer-Grimson algorithm" which is equivalent to Gaussian adaptation as used in the section "Computer simulation of Gaussian adaptation" above. In both cases the maximum likelihood method is used for estimation of mean values by adaptation at one sample at a time.

But there are differences. In the Stauffer-Grimson case the information is not used for the control of a random number generator for centering, maximization of mean fitness, average information or manufacturing yield. The adaptation of the moment matrix also differs very much as compared to "the evolution in the brain" above.
References

- Müller, Christian L. and Sbalzarini Ivo F. Gaussian Adaptation revisited - an entropic view on Covariance Matrix Adaptation. Institute of Theoretical Computer Science and Swiss Institute of Bioinformatics, ETH Zurich, CH-8092 Zurich, Switzerland.

Differential evolution

In computer science, differential evolution (DE) is a method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. Such methods are commonly known as metaheuristics as they make few or no assumptions about the problem being optimized and can search very large spaces of candidate solutions. However, metaheuristics such as DE do not guarantee an optimal solution is ever found.

DE is used for multidimensional real-valued functions but does not use the gradient of the problem being optimized, which means DE does not require for the optimization problem to be differentiable as is required by classic optimization methods such as gradient descent and quasi-newton methods. DE can therefore also be used on optimization problems that are not even continuous, are noisy, change over time, etc [1].

DE optimizes a problem by maintaining a population of candidate solutions and creating new candidate solutions by combining existing ones according to its simple formulae, and then keeping whichever candidate solution has the best score or fitness on the optimization problem at hand. In this way the optimization problem is treated as a black box that merely provides a measure of quality given a candidate solution and the gradient is therefore not needed. DE is originally due to Storn and Price [2][3]. Books have been published on theoretical and practical aspects of using DE in parallel computing, multiobjective optimization, constrained optimization, and the books also contain surveys of application areas [4][5][6].

Algorithm

A basic variant of the DE algorithm works by having a population of candidate solutions (called agents). These agents are moved around in the search-space by using simple mathematical formulae to combine the positions of existing agents from the population. If the new position of an agent is an improvement it is accepted and forms part of the population, otherwise the new position is simply discarded. The process is repeated and by doing so it is hoped, but not guaranteed, that a satisfactory solution will eventually be discovered.

Formally, let $f : \mathbb{R}^n \rightarrow \mathbb{R}$ be the cost function which must be minimized or fitness function which must be maximized. The function takes a candidate solution as argument in the form of a vector of real numbers and produces a real number as output which indicates the fitness of the given candidate solution. The gradient of $f$ is not known. The goal is to find a solution $m$ for which $f(m) \leq f(p)$ for all $p$ in the search-space, which would mean $m$ is the global minimum. Maximization can be performed by considering the function $h := -f$ instead.

Let $x \in \mathbb{R}^n$ designate a candidate solution (agent) in the population. The basic DE algorithm can then be described as follows:

• Initialize all agents $x$ with random positions in the search-space.
• Until a termination criterion is met (e.g. number of iterations performed, or adequate fitness reached), repeat the following:
For each agent \( \mathbf{x} \) in the population do:

- Pick three agents \( \mathbf{a}, \mathbf{b}, \text{ and } \mathbf{c} \) from the population at random, they must be distinct from each other as well as from agent \( \mathbf{x} \).
- Pick a random index \( R \in \{1, \ldots, n\} \) (\( n \) being the dimensionality of the problem to be optimized).
- Compute the agent’s potentially new position \( \mathbf{y} = [y_1, \ldots, y_n] \) as follows:
  - For each \( i \), pick a uniformly distributed number \( r_i \equiv U(0, 1) \)
  - If \( r_i < \text{CR} \) or \( i = R \) then set \( y_i = a_i + F(b_i - c_i) \) otherwise set \( y_i = x_i \)
- If \( f(\mathbf{y}) < f(\mathbf{x}) \) then replace the agent in the population with the improved candidate solution, that is, replace \( \mathbf{x} \) with \( \mathbf{y} \) in the population.
- Pick the agent from the population that has the highest fitness or lowest cost and return it as the best found candidate solution.

Note that \( F \in [0, 2] \) is called the *differential weight* and \( \text{CR} \in [0, 1] \) is called the *crossover probability*, both these parameters are selectable by the practitioner along with the population size \( \text{NP} \geq 4 \) see below.

### Parameter selection

The choice of DE parameters \( F, \text{CR} \text{ and } \text{NP} \) can have a large impact on optimization performance. Selecting the DE parameters that yield good performance has therefore been the subject of much research. Rules of thumb for parameter selection were devised by Storn et al. \([3][4]\) and Liu and Lampinen \([7]\). Mathematical convergence analysis regarding parameter selection was done by Zaharie \([8]\). Meta-optimization of the DE parameters was done by Pedersen \([9][10]\) and Zhang et al. \([11]\).

### Variants

Variants of the DE algorithm are continually being developed in an effort to improve optimization performance. Many different schemes for performing crossover and mutation of agents are possible in the basic algorithm given above, see e.g. \([3]\). More advanced DE variants are also being developed with a popular research trend being to perturb or adapt the DE parameters during optimization, see e.g. Price et al. \([4]\), Liu and Lampinen \([12]\), Qin and Suganthan \([13]\), and Brest et al. \([14]\).

### References


Particle swarm optimization

In computer science, particle swarm optimization (PSO) is a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. PSO optimizes a problem by having a population of candidate solutions, here dubbed particles, and moving these particles around in the search-space according to simple mathematical formulae over the particle's position and velocity. Each particle's movement is influenced by its own best known position in the search-space as well as the entire swarm's best known position. When improved positions are being discovered these will then come to guide the movements of the swarm. The process is repeated and by doing so it is hoped, but not guaranteed, that a satisfactory solution will

Algorithm
A basic variant of the PSO algorithm works by having a population (called a swarm) of candidate solutions (called particles). These particles are moved around in the search-space according to a few simple formulae. The movements of the particles are guided by their own best known position in the search-space as well as the entire swarm's best known position. When improved positions are being discovered these will then come to guide the movements of the swarm. The process is repeated and by doing so it is hoped, but not guaranteed, that a satisfactory solution will
Particle swarm optimization

136

eventually be discovered.

Formally, let \( f : \mathbb{R}^n \rightarrow \mathbb{R} \) be the fitness or cost function which must be minimized. The function takes a candidate solution as argument in the form of a vector of real numbers and produces a real number as output which indicates the fitness of the given candidate solution. The gradient of \( f \) is not known. The goal is to find a solution \( \mathbf{a} \) for which \( f(\mathbf{a}) \leq f(\mathbf{b}) \) for all \( \mathbf{b} \) in the search-space, which would mean \( \mathbf{a} \) is the global minimum. Maximization can be performed by considering the function \( h = -f \) instead.

Let \( S \) be the number of particles in the swarm, each having a position \( \mathbf{x}_i \in \mathbb{R}^n \) in the search-space and a velocity \( \mathbf{v}_i \in \mathbb{R}^n \). Let \( \mathbf{p}_i \) be the best known position of particle \( i \) and let \( \mathbf{g} \) be the best known position of the entire swarm. A basic PSO algorithm is then:

- For each particle \( i = 1, \ldots, S \) do:
  - Initialize the particle's position with a uniformly distributed random vector: \( \mathbf{x}_i \sim U(\mathbf{b}_{\text{lo}}, \mathbf{b}_{\text{up}}) \), where \( \mathbf{b}_{\text{lo}} \) and \( \mathbf{b}_{\text{up}} \) are the lower and upper boundaries of the search-space.
  - Initialize the particle’s best known position to its initial position: \( \mathbf{p}_i \leftarrow \mathbf{x}_i \)
  - If (\( f(\mathbf{p}_i) < f(\mathbf{g}) \)) update the swarm’s best known position: \( \mathbf{g} \leftarrow \mathbf{p}_i \)
  - Initialize the particle's velocity: \( \mathbf{v}_i \sim U(-|\mathbf{b}_{\text{up}}-\mathbf{b}_{\text{lo}}|, |\mathbf{b}_{\text{up}}-\mathbf{b}_{\text{lo}}|) \)
  - Until a termination criterion is met (e.g. number of iterations performed, or adequate fitness reached), repeat:
    - For each particle \( i = 1, \ldots, S \) do:
      - For each dimension \( d = 1, \ldots, n \) do:
        - Pick random numbers: \( r_p, r_g \sim U(0,1) \)
        - Update the particle’s velocity: \( \mathbf{v}_{i,d} \leftarrow \omega \mathbf{v}_{i,d} + \phi_p r_p (\mathbf{p}_{i,d} - \mathbf{x}_{i,d}) + \phi_g r_g (\mathbf{g}_d - \mathbf{x}_{i,d}) \)
        - Update the particle’s position: \( \mathbf{x}_{i} \leftarrow \mathbf{x}_{i} + \mathbf{v}_{i} \)
        - If (\( f(\mathbf{x}_i) < f(\mathbf{p}_i) \)) do:
          - Update the particle's best known position: \( \mathbf{p}_i \leftarrow \mathbf{x}_i \)
          - If (\( f(\mathbf{p}_i) < f(\mathbf{g}) \)) update the swarm’s best known position: \( \mathbf{g} \leftarrow \mathbf{p}_i \)
    - Now \( \mathbf{g} \) holds the best found solution.

The parameters \( \omega, \phi_p, \) and \( \phi_g \) are selected by the practitioner and control the behaviour and efficacy of the PSO method, see below.

**Parameter selection**

The choice of PSO parameters can have a large impact on optimization performance. Selecting PSO parameters that yield good performance has therefore been the subject of much research. \(^[7][8][9][10][11][12][13][14]\)

The PSO parameters can also be tuned by using another overlaying optimizer, a concept known as meta-optimization. \(^[15][16][17]\)

Parameters have also been tuned for various optimization scenarios. \(^ [18] \)

**Neighbourhoods and Topologies**

The basic PSO is easily trapped into a local minimum. This premature convergence can be avoided by not using any more the entire swarm's best known position \( \mathbf{g} \) but just the best known position \( \mathbf{l} \) of a sub-swarm "around" the particle that is moved. Such a sub-swarm can be a geometrical one - for example "the \( m \) nearest particles" - or, more often, a social one, i.e. a set of
particles that is not depending on any distance. In such a case, the PSO variant is said to be local best (vs global best for the basic PSO).

If we suppose there is an information link between each particle and its neighbours, the set of these links builds a graph, a communication network, that is called the topology of the PSO variant. A commonly used social topology is the ring, in which each particle has just two neighbours, but there are far more. The topology is not necessarily fixed, and can be adaptive (SPSO, stochastic star, TRIBES, Cyber Swarm).

**Inner workings**

There are several schools of thought as to why and how the PSO algorithm can perform optimization.

A common belief amongst researchers is that the swarm behaviour varies between exploratory behaviour, that is, searching a broader region of the search-space, and exploitative behaviour, that is, a locally oriented search so as to get closer to a (possibly local) optimum. This school of thought has been prevalent since the inception of PSO. This school of thought contends that the PSO algorithm and its parameters must be chosen so as to properly balance between exploration and exploitation to avoid premature convergence to a local optimum yet still ensure a good rate of convergence to the optimum. This belief is the precursor of many PSO variants, see below.

Another school of thought is that the behaviour of a PSO swarm is not well understood in terms of how it affects actual optimization performance, especially for higher dimensional search-spaces and optimization problems that may be discontinuous, noisy, and time-varying. This school of thought merely tries to find PSO algorithms and parameters that cause good performance regardless of how the swarm behaviour can be interpreted in relation to e.g. exploration and exploitation. Such studies have led to the simplification of the PSO algorithm, see below.

**Convergence**

In relation to PSO the word *convergence* typically means one of two things, although it is often not clarified which definition is meant and sometimes they are mistakenly thought to be identical.

- Convergence may refer to the swarm's best known position $g$ approaching (converging to) the optimum of the problem, regardless of how the swarm behaves.
- Convergence may refer to a swarm collapse in which all particles have converged to a point in the search-space, which may or may not be the optimum.

Several attempts at mathematically analyzing PSO convergence exist in the literature. These analyses have resulted in guidelines for selecting PSO parameters that are believed to cause convergence, divergence or oscillation of the swarm's particles, and the analyses have also given rise to several PSO variants. However, the analyses were criticized by Pedersen for being oversimplified as they assume the swarm has only one particle, that it does not use stochastic variables and that the points of attraction, that is, the particle's best known position $p$ and the swarm's best known position $g$, remain constant throughout the optimization process. Furthermore, some analyses allow for an infinite number of optimization iterations which is not possible in reality. This means that determining convergence capabilities of different PSO algorithms and parameters therefore still depends on empirical results.
Biases

As the basic PSO works dimension by dimension, the solution point is easier found when it lies on an axis of the search space, on a diagonal, and even easier if it is right on the centre.[24][25]

A first approach to avoid this bias, and for fair comparisons, is precisely to use non-biased benchmark problems, that are shifted or rotated.[26]

Another approach is to modify the algorithm itself so that it is not any more sensitive to the system of coordinates.[27][28]

Variants

Numerous variants of even a basic PSO algorithm are possible. For example, there are different ways to initialize the particles and velocities (e.g. start with zero velocities instead), how to dampen the velocity, only update $p_i$ and $g$ after the entire swarm has been updated, etc. Some of these choices and their possible performance impact have been discussed in the literature.[9]

New and more sophisticated PSO variants are also continually being introduced in an attempt to improve optimization performance. There are certain trends in that research; one is to make a hybrid optimization method using PSO combined with other optimizers.[29][30] another research trend is to try and alleviate premature convergence (that is, optimization stagnation) e.g. by reversing or perturbing the movement of the PSO particles,[14][31][32] another approach to deal with premature convergence is the use of multiple swarms (multi-swarm optimization), and then there are also attempts at adapting the behavioural parameters of PSO during optimization.[33]

Simplifications

Another school of thought is that PSO should be simplified as much as possible without impairing its performance; a general concept often referred to as Occam's razor. Simplifying PSO was originally suggested by Kennedy[3] and has been studied more extensively,[13][16][17][34] where it appeared that optimization performance was improved, and the parameters were easier to tune and they performed more consistently across different optimization problems.

Another argument in favour of simplifying PSO is that metaheuristics can only have their efficacy demonstrated empirically by doing computational experiments on a finite number of optimization problems. This means a metaheuristic such as PSO cannot be proven correct and this increases the risk of making errors in its description and implementation. A good example of this[35] presented a promising variant of a genetic algorithm (another popular metaheuristic) but it was later found to be defective as it was strongly biased in its optimization search towards similar values for different dimensions in the search space, which happened to be the optimum of the benchmark problems considered. This bias was because of a programming error, and has now been fixed.[36]

Initialization of velocities may require extra inputs. A simpler variant is the accelerated particle swarm optimization (APSO)[37], which does not need to use velocity at all and can speed up the convergence in many applications. A simple demo code of APSO is available.[38]
Multi-objective optimization

PSO has also been applied to multi-objective problems,\cite{59,40} in which the fitness comparison takes pareto dominance into account when moving the PSO particles and non-dominated solutions are stored so as to approximate the pareto front.

Binary, Discrete, and Combinatorial PSO

As the PSO equations given above work on real numbers, a commonly used method to solve discrete problems is to map the discrete search space to a continuous domain, to apply a classical PSO, and then to demap the result. Such a mapping can be very simple (for example by just using rounded values) or more sophisticated.\cite{41}

However, it can be noted that the equations of movement make use of operators that perform four actions:

- computing the difference of two positions. The result is a velocity (more precisely a displacement)
- multiplying a velocity by a numerical coefficient
- adding two velocities
- applying a velocity to a position

Usually a position and a velocity are represented by $n$ real numbers, and these operators are simply $\cdot$, $\ast$, $\div$, and again $\pm$. But all these mathematical objects can be defined in a completely different way, in order to cope with binary problems (or more generally discrete ones), or even combinatorial ones.\cite{42,43,44,45} One approach is to redefine the operators based on sets.\cite{46}

References

\begin{itemize}
  \item [14] Evers, G. (2009) (Master's thesis). \textit{An Automatic Regrouping Mechanism to Deal with Stagnation in Particle Swarm Optimization} (http://www.georgeevers.org/publications.htm). The University of Texas - Pan American, Department of Electrical Engineering.
\end{itemize}
Particle swarm optimization


[20] SPSO, Particle Swarm Central (http://www.particleswarm.info)


[28] SPSO 2011, Particle Swarm Central (http://www.particleswarm.info)


[38] http://www.mathworks.com/matlabcentral/fileexchange/?term=APSO


External links

- Particle Swarm Central (http://www.particleswarm.info) is a repository for information on PSO. Several source codes are freely available.

Ant colony optimization algorithms

In computer science and operations research, the ant colony optimization algorithm (ACO) is a probabilistic technique for solving computational problems which can be reduced to finding good paths through graphs.

This algorithm is a member of the ant colony algorithms family, in swarm intelligence methods, and it constitutes some metaheuristic optimizations. Initially proposed by Marco Dorigo in 1992 in his PhD thesis, \[1\][2] the first algorithm was aiming to search for an optimal path in a graph, based on the behavior of ants seeking a path between their colony and a source of food. The original idea has since diversified to solve a wider class of numerical problems, and as a result, several problems have emerged, drawing on various aspects of the behavior of ants.

Overview

Summary

In the natural world, ants (initially) wander randomly, and upon finding food return to their colony while laying down pheromone trails. If other ants find such a path, they are likely not to keep travelling at random, but to instead follow the trail, returning and reinforcing it if they eventually find food (see Ant communication).

Over time, however, the pheromone trail starts to evaporate, thus reducing its attractive strength. The more time it takes for an ant to travel down the path and back again, the more time the pheromones have to evaporate. A short path, by comparison, gets marched over more frequently, and thus the pheromone density becomes higher on shorter paths than longer ones. Pheromone evaporation also has the advantage of avoiding the convergence to a locally optimal solution. If there were no evaporation at all, the paths chosen by the first ants would tend to be excessively attractive to the following ones. In that case, the exploration of the solution space would be constrained.

Thus, when one ant finds a good (i.e., short) path from the colony to a food source, other ants are more likely to follow that path, and positive feedback eventually leads all the ants following a single path. The idea of the ant colony algorithm is to mimic this behavior with "simulated ants" walking around the graph representing the problem to solve.
Detailed

The original idea comes from observing the exploitation of food resources among ants, in which ants' individually limited cognitive abilities have collectively been able to find the shortest path between a food source and the nest.

1. The first ant finds the food source (F), via any way (a), then returns to the nest (N), leaving behind a trail pheromone (b)
2. Ants indiscriminately follow four possible ways, but the strengthening of the runway makes it more attractive as the shortest route.
3. Ants take the shortest route, long portions of other ways lose their trail pheromones.

In a series of experiments on a colony of ants with a choice between two unequal length paths leading to a source of food, biologists have observed that ants tended to use the shortest route. A model explaining this behaviour is as follows:

1. An ant (called "blitz") runs more or less at random around the colony;
2. If it discovers a food source, it returns more or less directly to the nest, leaving in its path a trail of pheromone;
3. These pheromones are attractive, nearby ants will be inclined to follow, more or less directly, the track;
4. Returning to the colony, these ants will strengthen the route;
5. If there are two routes to reach the same food source then, in a given amount of time, the shorter one will be traveled by more ants than the long route;
6. The short route will be increasingly enhanced, and therefore become more attractive;
7. The long route will eventually disappear because pheromones are volatile;
8. Eventually, all the ants have determined and therefore "chosen" the shortest route.

Ants use the environment as a medium of communication. They exchange information indirectly by depositing pheromones, all detailing the status of their "work". The information exchanged has a local scope, only an ant located where the pheromones were left has a notion of them. This system is called "Stigmergy" and occurs in many social animal societies (it has been studied in the case of the construction of pillars in the nests of termites). The mechanism to solve a problem too complex to be addressed by single ants is a good example of a self-organized system. This system is based on positive feedback (the deposit of pheromone attracts other ants that will strengthen it themselves) and negative (dissipation of the route by evaporation prevents the system from thrashing). Theoretically, if the quantity of pheromone remained the same over time on all edges, no route would be chosen. However, because of feedback, a slight variation on an edge will be amplified and thus allow the choice of an edge. The algorithm will move from an unstable state in which no edge is stronger than another, to a stable state where the route is composed of the strongest edges.

The basic philosophy of the algorithm involves the movement of a colony of ants through the different states of the problem influenced by two local decision policies, viz., trails and attractiveness. Thereby, each such ant incrementally constructs a solution to the problem. When an ant completes a solution, or during the construction phase, the ant evaluates the solution and modifies the trail value on the components used in its solution. This
pheromone information will direct the search of the future ants. Furthermore, the algorithm also includes two more mechanisms, viz., trail evaporation and daemon actions. Trail evaporation reduces all trail values over time thereby avoiding any possibilities of getting stuck in local optima. The daemon actions are used to bias the search process from a non-local perspective.

**Common extensions**

Here are some of most popular variations of ACO Algorithms

**Elitist ant system**

The global best solution deposits pheromone on every iteration along with all the other ants.

**Max-Min ant system (MMAS)**

Added Maximum and Minimum pheromone amounts $[\tau_{\text{max}}, \tau_{\text{min}}]$ Only global best or iteration best tour deposited pheromone. All edges are initialized to $\tau_{\text{max}}$ and reinitialized to $\tau_{\text{max}}$ when nearing stagnation. [5]

**Ant Colony System**

It has been presented above. [6]

**Rank-based ant system (ASrank)**

All solutions are ranked according to their length. The amount of pheromone deposited is then weighted for each solution, such that solutions with shorter paths deposit more pheromone than the solutions with longer paths.

**Continuous orthogonal ant colony (COAC)**

The pheromone deposit mechanism of COAC is to enable ants to search for solutions collaboratively and effectively. By using an orthogonal design method, ants in the feasible domain can explore their chosen regions rapidly and efficiently, with enhanced global search capability and accuracy.

The orthogonal design method and the adaptive radius adjustment method can also be extended to other optimization algorithms for delivering wider advantages in solving practical problems. [7]

**Convergence**

For some versions of the algorithm, it is possible to prove that it is convergent (i.e. it is able to find the global optimum in finite time). The first evidence of a convergence ant colony algorithm was made in 2000, the graph-based ant system algorithm, and then algorithms for ACS and MMAS. Like most metaheuristics, it is very difficult to estimate the theoretical speed of convergence. In 2004, Zlochin and his colleagues [8] showed that COA-type algorithms could be assimilated methods of stochastic gradient descent, on the cross-entropy and estimation of distribution algorithm. They proposed these metaheuristics as a "research-based model".

**Example pseudo-code and formulae**

```plaintext
procedure ACO_MetaHeuristic
  while (not_termination)
    generateSolutions()
    daemonActions()
    pheromoneUpdate()
  end while
```


Ant colony optimization algorithms

**End procedure**

Edge selection

An ant is a simple computational agent in the ant colony optimization algorithm. It iteratively constructs a solution for the problem at hand. The intermediate solutions are referred to as solution states. At each iteration of the algorithm, each ant moves from a state $x$ to state $y$, corresponding to a more complete intermediate solution. Thus, each ant $k$ computes a set $A_k(x)$ of feasible expansions to its current state in each iteration, and moves to one of these in probability. For ant $k$, the probability $p^k_{xy}$ of moving from state $x$ to state $y$ depends on the combination of two values, viz., the attractiveness $\eta_{xy}$ of the move, as computed by some heuristic indicating the *a priori* desirability of that move and the trail level $\tau_{xy}$ of the move, indicating how proficient it has been in the past to make that particular move.

The *trail level* represents a posteriori indication of the desirability of that move. Trails are updated usually when all ants have completed their solution, increasing or decreasing the level of trails corresponding to moves that were part of "good" or "bad" solutions, respectively.

In general, the $k$th ant moves from state $x$ to state $y$ with probability

$$p^k_{xy} = \frac{(\tau^\alpha_{xy})(\eta^\beta_{xy})}{\sum(\tau^\alpha_{xy})(\eta^\beta_{xy})}$$

where

$\tau_{xy}$ is the amount of pheromone deposited for transition from state $x$ to $y$, $0 \leq \alpha$ is a parameter to control the influence of $\tau_{xy}$, $\eta_{xy}$ is the desirability of state transition $xy$ (a *priori* knowledge, typically $1/d_{xy}$, where $d$ is the distance) and $\beta \geq 1$ is a parameter to control the influence of $\eta_{xy}$.

**Pheromone update**

When all the ants have completed a solution, the trails are updated by $\tau_{xy} \leftarrow (1 - \rho)\tau_{xy} + \sum_k \Delta \tau^k_{xy}$

where

$\tau_{xy}$ is the amount of pheromone deposited for a state transition $xy$, $\rho$ is the *pheromone evaporation coefficient* and $\Delta \tau^k_{xy}$ is the amount of pheromone deposited by $k$th ant, typically given for a TSP problem (with moves corresponding to arcs of the graph) by

$$\Delta \tau^k_{xy} = \begin{cases} Q/L_k & \text{if ant } k \text{ uses curve } xy \text{ in its tour} \\ 0 & \text{otherwise} \end{cases}$$

where $L_k$ is the cost of the $k$th ant's tour (typically length) and $Q$ is a constant.
Applications

Ant colony optimization algorithms have been applied to many combinatorial optimization problems, ranging from quadratic assignment to protein folding or routing vehicles and a lot of derived methods have been adapted to dynamic problems in real variables, stochastic problems, multi-targets and parallel implementations. It has also been used to produce near-optimal solutions to the travelling salesman problem. They have an advantage over simulated annealing and genetic algorithm approaches of similar problems when the graph may change dynamically; the ant colony algorithm can be run continuously and adapt to changes in real time. This is of interest in network routing and urban transportation systems.

The first ACO algorithm was called the Ant system \cite{9} and it was aimed to solve the travelling salesman problem, in which the goal is to find the shortest round-trip to link a series of cities. The general algorithm is relatively simple and based on a set of ants, each making one of the possible round-trips along the cities. At each stage, the ant chooses to move from one city to another according to some rules:

1. It must visit each city exactly once;
2. A distant city has less chance of being chosen (the visibility);
3. The more intense the pheromone trail laid out on an edge between two cities, the greater the probability that that edge will be chosen;
4. Having completed its journey, the ant deposits more pheromones on all edges it traversed, if the journey is short;
5. After each iteration, trails of pheromones evaporate.
Scheduling problem
- Job-shop scheduling problem (JSP)\[^{[10]}\]
- Open-shop scheduling problem (OSP)\[^{[11]}[12]\]
- Permutation flow shop problem (PFSP)\[^{[13]}\]
- Single machine total tardiness problem (SMTTP)\[^{[14]}\]
- Single machine total weighted tardiness problem (SMTWTP)\[^{[15]}[16][17]\]
- Resource-constrained project scheduling problem (RCPSP)\[^{[18]}\]
- Group-shop scheduling problem (GSP)\[^{[19]}\]
- Single-machine total tardiness problem with sequence dependent setup times (SMTTPDST)\[^{[20]}\]
- Multistage Flowshop Scheduling Problem (MFSP) with sequence dependent setup/changeover times\[^{[21]}\]

Vehicle routing problem
- Capacitated vehicle routing problem (CVRP)\[^{[22]}[23][24]\]
- Multi-depot vehicle routing problem (MDVRP)\[^{[25]}\]
- Period vehicle routing problem (PVRP)\[^{[26]}\]
- Split delivery vehicle routing problem (SDVRP)\[^{[27]}\]
- Stochastic vehicle routing problem (SVRP)\[^{[28]}\]
- Vehicle routing problem with pick-up and delivery (VRPPD)\[^{[29]}[30]\]
- Vehicle routing problem with time windows (VRPTW)\[^{[31]}[32][33]\]
- Time Dependent Vehicle Routing Problem with Time Windows (TDVRPTW)\[^{[34]}\]
- Vehicle Routing Problem with Time Windows and Multiple Service Workers (VRPTWMS)\[^{[35]}[36][37][38]\]

Assignment problem
- Quadratic assignment problem (QAP)\[^{[35]}\]
- Generalized assignment problem (GAP)\[^{[36]}[37]\]
- Frequency assignment problem (FAP)\[^{[38]}\]
- Redundancy allocation problem (RAP)\[^{[39]}\]

Set problem
- Set covering problem (SCP)\[^{[40]}[41]\]
- Set partition problem (SPP)\[^{[42]}\]
- Weight constrained graph tree partition problem (WCGTPP)\[^{[43]}\]
- Arc-weighted I-cardinality tree problem (AWICTP)\[^{[44]}\]
- Multiple knapsack problem (MKP)\[^{[45]}\]
- Maximum independent set problem (MIS)\[^{[46]}\]

Others
- Classification\[^{[47]}\]
- Connection-oriented network routing\[^{[48]}\]
- Connectionless network routing\[^{[49]}[50]\]
- Data mining\[^{[47]}[51][52][53]\]
- Discounted cash flows in project scheduling\[^{[54]}\]
- Distributed Information Retrieval\[^{[55]}[56]\]
- Grid Workflow Scheduling Problem\[^{[57]}\]
- Image processing\[^{[58]}[59]\]
- Intelligent testing system\[^{[60]}\]
Definition difficulty

With an ACO algorithm, the shortest path in a graph, between two points A and B, is built from a combination of several paths. It is not easy to give a precise definition of what algorithm is or is not an ant colony, because the definition may vary according to the authors and uses. Broadly speaking, ant colony algorithms are regarded as populated metaheuristics with each solution represented by an ant moving in the search space. Ants mark the best solutions and take account of previous markings to optimize their search. They can be seen as probabilistic multi-agent algorithms using a probability distribution to make the transition between each iteration. In their versions for combinatorial problems, they use an iterative construction of solutions. According to some authors, the thing which distinguishes ACO algorithms from other relatives (such as algorithms to estimate the distribution or particle swarm optimization) is precisely their constructive aspect. In combinatorial problems, it is possible that the best solution eventually be found, even though no ant would prove effective. Thus, in the example of the Travelling salesman problem, it is not necessary that an ant actually travels the shortest route: the shortest route can be built from the strongest segments of the best solutions. However, this definition can be problematic in the case of problems in real variables, where no structure of 'neighbours' exists. The collective behaviour of social insects remains a source of inspiration for researchers. The wide variety of algorithms (for optimization or not) seeking self-organization in biological systems has led to the concept of "swarm intelligence", which is a very general framework in which ant colony algorithms fit.

Stigmergy algorithms

There is in practice a large number of algorithms claiming to be "ant colonies", without always sharing the general framework of optimization by canonical ant colonies (COA). In practice, the use of an exchange of information between ants via the environment (a principle called "Stigmergy") is deemed enough for an algorithm to belong to the class of ant colony algorithms. This principle has led some authors to create the term "value" to organize methods and behavior based on search of food, sorting larvae, division of labour and cooperative transportation.[66]

Related methods

• Genetic algorithms (GA) maintain a pool of solutions rather than just one. The process of finding superior solutions mimics that of evolution, with solutions being combined or mutated to alter the pool of solutions, with solutions of inferior quality being discarded.
• Simulated annealing (SA) is a related global optimization technique which traverses the search space by generating neighboring solutions of the current solution. A superior neighbor is always accepted. An inferior neighbor is accepted probabilistically based on the difference in quality and a temperature parameter. The temperature parameter is modified as the algorithm progresses to alter the nature of the search.
• Reactive search optimization focuses on combining machine learning with optimization, by adding an internal feedback loop to self-tune the free parameters of an algorithm to the characteristics of the problem, of the
instance, and of the local situation around the current solution.

- Tabu search (TS) is similar to simulated annealing in that both traverse the solution space by testing mutations of an individual solution. While simulated annealing generates only one mutated solution, tabu search generates many mutated solutions and moves to the solution with the lowest fitness of those generated. To prevent cycling and encourage greater movement through the solution space, a tabu list is maintained of partial or complete solutions. It is forbidden to move to a solution that contains elements of the tabu list, which is updated as the solution traverses the solution space.

- Artificial immune system (AIS) algorithms are modeled on vertebrate immune systems.
- Particle swarm optimization (PSO), a Swarm intelligence method
- Intelligent Water Drops (IWD), a swarm-based optimization algorithm based on natural water drops flowing in rivers
- Gravitational Search Algorithm (GSA), a Swarm intelligence method
- Ant colony clustering method (ACCM), a method that make use of clustering approach, extending the ACO.
- Stochastic diffusion search (SDS), an agent-based probabilistic global search and optimization technique best suited to problems where the objective function can be decomposed into multiple independent partial-functions

### History

Chronology of COA algorithms

- 1959, Pierre-Paul Grassé invented the theory of Stigmergy to explain the behavior of nest building in termites;[67]
- 1983, Deneubourg and his colleagues studied the collective behavior of ants;[68]
- 1988, and Moyson Manderick have an article on self-organization among ants;[69]
- 1989, the work of Goss, Aron, Deneubourg and Pasteels on the collective behavior of Argentine ants, which will give the idea of Ant colony optimization algorithms;[3]
- 1989, implementation of a model of behavior for food by Ebling and his colleagues;[70]
- 1991, M. Dorigo proposed the Ant System in his doctoral thesis (which was published in 1992).[2] A technical report extracted from the thesis and co-authored by V. Maniezzo and A. Colomi[71] was published five years later;[9]
- 1996, publication of the article on Ant System;[9]
• 1996, Hoos and Stützle invent the MAX-MIN Ant System;[5]
• 1997, Dorigo and Gambardella publish the Ant Colony System;[6]
• 1997, Schoonwoerd and his colleagues developed the first application to telecommunication networks;[72]
• 1998, Dorigo launches first conference dedicated to the ACO algorithms;[73]
• 1998, Stützle proposes initial parallel implementations;[74]
• 1999, Bonabeau, Dorigo and Theraulaz publish a book dealing mainly with artificial ants[75]
• 2000, special issue of the Future Generation Computer Systems journal on ant algorithms[76]
• 2000, first applications to the scheduling, scheduling sequence and the satisfaction of constraints;
• 2000, Gutjahr provides the first evidence of convergence for an algorithm of ant colonies[77]
• 2001, the first use of COA Algorithms by companies (Eurobios [78] and AntOptima[79]);
• 2001, IREDA and his colleagues published the first multi-objective algorithm[80]
• 2002, first applications in the design of schedule, Bayesian networks;
• 2002, Bianchi and her colleagues suggested the first algorithm for stochastic problem;[81]
• 2004, Zlochin and Dorigo show that some algorithms are equivalent to the stochastic gradient descent, the cross-entropy and algorithms to estimate distribution[8]
• 2005, first applications to protein folding problems.

References

Ant colony optimization algorithms


Ant colony optimization algorithms

Publications (selected)

- C. Blum, 2005 "Ant colony optimization: Introduction and recent trends". Physics of Life Reviews, 2: 353-373

External links

- Ant Colony Optimization Home Page (http://www.aco-metaheuristic.org/)
- AntSim - Simulation of Ant Colony Algorithms (http://www.nightlab.ch/antsim)
- MIDACO-Solver (http://www.midaco-solver.com/) General purpose optimization software based on Ant Colony Optimization (Matlab, Excel, C/C++, Fortran, Python)
- University of Kaiserslautern, Germany, AG Wehn: Ant Colony Optimization Applet (http://ems.eit.uni-kl.de/index.php?id=156) Visualization of Traveling Salesman solved by Ant System with numerous options and parameters (Java Applet)
- Ant Farm Simulator (http://webspace.webring.com/people/br/raguirre/hormigas/antfarm/)
- Ant algorithm simulation (Java Applet) (http://www.djob.net/inde/ANTColony/applet.html)
Artificial bee colony algorithm

In computer science and operations research, the artificial bee colony algorithm (ABC) is an optimization algorithm based on the intelligent foraging behaviour of honey bee swarm, proposed by Karaboga in 2005.[1]

Algorithm

In the ABC model, the colony consists of three groups of bees: employed bees, onlookers and scouts. It is assumed that there is only one artificial employed bee for each food source. In other words, the number of employed bees in the colony is equal to the number of food sources around the hive. Employed bees go to their food source and come back to hive and dance on this area. The employed bee whose food source has been abandoned becomes a scout and starts to search for finding a new food source. Onlookers watch the dances of employed bees and choose food sources depending on dances. The main steps of the algorithm are given below:

- Initial food sources are produced for all employed bees
- REPEAT
  - Each employed bee goes to a food source in her memory and determines a neighbour source, then evaluates its nectar amount and dances in the hive
  - Each onlooker watches the dance of employed bees and chooses one of their sources depending on the dances, and then goes to that source. After choosing a neighbour around that, she evaluates its nectar amount.
  - Abandoned food sources are determined and are replaced with the new food sources discovered by scouts.
  - The best food source found so far is registered.
- UNTIL (requirements are met)

In ABC, a population based algorithm, the position of a food source represents a possible solution to the optimization problem and the nectar amount of a food source corresponds to the quality (fitness) of the associated solution. The number of the employed bees is equal to the number of solutions in the population. At the first step, a randomly distributed initial population (food source positions) is generated. After initialization, the population is subjected to repeat the cycles of the search processes of the employed, onlooker, and scout bees, respectively. An employed bee produces a modification on the source position in her memory and discovers a new food source position. Provided that the nectar amount of the new one is higher than that of the previous source, the bee memorizes the new source position and forgets the old one. Otherwise she keeps the position of the one in her memory. After all employed bees complete the search process, they share the position information of the sources with the onlookers on the dance area. Each onlooker evaluates the nectar information taken from all employed bees and then chooses a food source depending on the nectar amounts of sources. As in the case of the employed bee, she produces a modification on the source position in her memory and checks its nectar amount. Providing that its nectar is higher than that of the previous one, the bee memorizes the new position and forgets the old one. The sources abandoned are determined and new sources are randomly produced to be replaced with the abandoned ones by artificial scouts.

Application to real-world problems

References


10. Mustafa Sonmez, Discrete optimum design of truss structures using artificial bee colony algorithm, Structural and Multidisciplinary Optimization, Volume 43 Issue 1, January 2011

External links

- Artificial Bee Colony Algorithm (http://mf.erciyes.edu.tr/abc)
Evolution strategy

In computer science, **Evolution Strategy (ES)** is an optimization technique based on ideas of adaptation and evolution. It belongs to the general class of evolutionary computation or artificial evolution methodologies.

**History**

The **evolution strategy** optimization technique was created in the early 1960s and developed further in the 1970s and later by Ingo Rechenberg, Hans-Paul Schwefel and his co-workers.

**Methods**

Evolution strategies use natural problem-dependent representations, and primarily mutation and selection, as search operators. In common with evolutionary algorithms, the operators are applied in a loop. An iteration of the loop is called a generation. The sequence of generations is continued until a termination criterion is met.

As far as real-valued search spaces are concerned, mutation is normally performed by adding a normally distributed random value to each vector component. The step size or mutation strength (i.e. the standard deviation of the normal distribution) is often governed by self-adaptation (see evolution window). Individual step sizes for each coordinate or correlations between coordinates are either governed by self-adaptation or by covariance matrix adaptation (CMA-ES).

The (environmental) selection in evolution strategies is deterministic and only based on the fitness rankings, not on the actual fitness values. The resulting algorithm is therefore invariant with respect to monotonic transformations of the objective function. The simplest evolution strategy operates on a population of size two: the current point (parent) and the result of its mutation. Only if the mutant's fitness is at least as good as the parent one, it becomes the parent of the next generation. Otherwise the mutant is disregarded. This is a (1 + 1)-ES. More generally, \( \lambda \) mutants can be generated and compete with the parent, called (1 + \( \lambda \))-ES. In (1, \( \lambda \))-ES the best mutant becomes the parent of the next generation while the current parent is always disregarded. For some of these variants, proofs of linear convergence (in a stochastic sense) have been derived on unimodal objective functions.\(^{[1][2]}\)

Contemporary derivatives of evolution strategy often use a population of \( \mu \) parents and also recombination as an additional operator, called (\( \mu/\rho^+ \), \( \lambda \))-ES. This makes them less prone to get stuck in local optima.\(^{[3]}\)

**References**


**Bibliography**


Research centers
• Bionics & Evolutionstechnique at the Technical University Berlin (http://www.bionik.tu-berlin.de/institut/xstart.htm)
• Chair of Algorithm Engineering (Ls11) – University of Dortmund (http://ls11-www.cs.uni-dortmund.de/)
• Collaborative Research Center 531 – University of Dortmund (http://sfbci.cs.uni-dortmund.de/)

External links
• CMA Evolution Strategy (http://www.lri.fr/~hansen/cmaesintro.html) – a contemporary variant where the complete covariance matrix of the multivariate normal mutation distribution is adapted.
• Evolution Strategies Animations (http://www.bionik.tu-berlin.de/institut/xx2anima.html) - Some interesting animations and real world problems (such as format of lenses, bridges configurations, etc) solved through Evolution Strategies.
• Evolutionary Algorithms Demos (http://www.frankiedr.de/demos.html) – There are some applets with Evolution Strategies and Genetic Algorithms that the user can manipulate to solve problems. Very interesting for a comparison between the two Evolutionary Algorithms.
• Professor Hans-Paul Schwefel talks to EvoNews (http://evonet.lri.fr/evoweb/news_events/news_features/article.php?id=5) – An interview with Professor Hans-Paul Schwefel, one of the Evolution Strategy pioneers.
Evolution window

It was observed in evolution strategies that significant progress toward the fitness/objective function's optimum, generally, can only happen in a narrow band of the mutation step size $\sigma$. That narrow band is called **evolution window**.

There are three well-known methods to adapt the mutation step size $\sigma$ in evolution strategies:

- (1/5-th) Success Rule
- Self-Adaptation (for example through log-normal mutations)
- Cumulative Step Size Adaptation (CSA)

On simple functions all of them have been empirically shown to keep the step size within the evolution window.

**References**


**CMA-ES**

**CMA-ES** stands for Covariance Matrix Adaptation Evolution Strategy. Evolution strategies (ES) are stochastic, derivative-free methods for numerical optimization of non-linear or non-convex continuous optimization problems. They belong to the class of evolutionary algorithms and evolutionary computation. An evolutionary algorithm is broadly based on the principle of biological evolution, namely the repeated interplay of variation (via mutation and recombination) and selection: in each generation (iteration) new individuals (candidate solutions, denoted as $x$) are generated by variation, usually in a stochastic way, and then some individuals are selected for the next generation based on their fitness or objective function value $f(x)$. Like this, over the generation sequence, individuals with better and better $f$-values are generated.

In an evolution strategy, new candidate solutions are sampled according to a multivariate normal distribution in the $\mathbb{R}^n$. Pairwise dependencies between the variables in this distribution are represented by a covariance matrix. The covariance matrix adaptation (CMA) is a method to update the covariance matrix of this distribution. This is particularly useful, if the function $f$ is ill-conditioned.

Adaptation of the covariance matrix amounts to learning a second order model of the underlying objective function similar to the approximation of the inverse Hessian matrix in the Quasi-Newton method in classical optimization. In contrast to most classical methods, fewer assumptions on the nature of the underlying objective function are made. Only the ranking between candidate solutions is exploited for learning the sample distribution and neither derivatives nor even the function values themselves are required by the method.
**Principles**

Two main principles for the adaptation of parameters of the search distribution are exploited in the CMA-ES algorithm.

First, a maximum-likelihood principle, based on the idea to increase the probability of successful candidate solutions and search steps. The mean of the distribution is updated such that the likelihood of previously successful candidate solutions is maximized. The covariance matrix of the distribution is updated (incrementally) such that the likelihood of previously successful search steps is increased. Both updates can be interpreted as a natural gradient descent. Also, in consequence, the CMA conducts an iterated principal components analysis of successful search steps while retaining all principal axes. Estimation of distribution algorithms and the Cross-Entropy Method are based on very similar ideas, but estimate (non-incrementally) the covariance matrix by maximizing the likelihood of successful solution points instead of successful search steps.

Second, two paths of the time evolution of the distribution mean of the strategy are recorded, called search or evolution paths. These paths contain significant information about the correlation between consecutive steps. Specifically, if consecutive steps are taken in a similar direction, the evolution paths become long. The evolution paths are exploited in two ways. One path is used for the covariance matrix adaptation procedure in place of single successful search steps and facilitates a possibly much faster variance increase of favorable directions. The other path is used to conduct an additional step-size control. This step-size control aims to make consecutive movements of the distribution mean orthogonal in expectation. The step-size control effectively prevents premature convergence yet allowing fast convergence to an optimum.

**Algorithm**

In the following the most commonly used \( (\mu/\mu_w, \lambda) \)-CMA-ES is outlined, where in each iteration step a weighted combination of the \( \mu \) best out of \( \lambda \) new candidate solutions is used to update the distribution parameters. The main loop consists of three main parts: 1) sampling of new solutions, 2) re-ordering of the sampled solutions based on their fitness, 3) update of the internal state variables based on the re-ordered samples. A pseudocode of the algorithm looks as follows.

```plaintext
set \( \lambda \) // number of samples per iteration, at least two, generally > 4
initialize \( m, \sigma, C = I, p_\sigma = 0, p_c = 0 \) // initialize state variables
while not terminate // iterate
  for \( i \) in \( \{1...\lambda\} \) // sample \( \lambda \) new solutions and evaluate them
    \( x_i = \text{sample}\_\text{multivariate}\_\text{normal} (\text{mean} = m, \text{covariance}\_\text{matrix} = \sigma^2C) \)
    \( f_i = \text{fitness}\_\text{with} (x_i) \)
    \( x_{1...\lambda} = x_{s(1)...s(\lambda)} \) with \( s(i) = \text{argsort}(f_{1...\lambda}, i) \) // sort solutions
    \( m' = m \) // we need later \( m - m' \) and \( x_i - m' \)
    \( m \leftarrow \text{update}_m(x_1, ..., x_\lambda) \) // move mean to better solutions
    \( p_\sigma \leftarrow \text{update}\_ps(p_\sigma, \sigma^{-1}C^{-1/2}(m - m')) \) // update isotropic evolution path
    \( p_c \leftarrow \text{update}\_pc(p_c, \sigma^{-1}(m - m'), ||p_\sigma||) \) // update anisotropic evolution path
    \( C \leftarrow \text{update}\_C(C, p_\sigma, (x_1 - m')/\sigma, ..., (x_\lambda - m')/\sigma) \) // update covariance matrix
```
\[ \sigma \leftarrow \text{update}_\sigma (\sigma, \|p_\sigma\|) \] // update step-size using isotropic path length

return \( m \) or \( x_1 \)

The order of the five update assignments is relevant. In the following, the update equations for the five state variables are specified.

Given are the search space dimension \( n \) and the iteration step \( k \). The five state variables are

\[
m_k \in \mathbb{R}^n, \text{ the distribution mean and current favorite solution to the optimization problem,}
\]

\[
\sigma_k > 0, \text{ the step-size,}
\]

\[
C_k, \text{ a symmetric and positive definite } n \times n \text{ covariance matrix with } C_0 = I \text{ and}
\]

\[
p_\sigma \in \mathbb{R}^n, p_c \in \mathbb{R}^n, \text{ two evolution paths, initially set to the zero vector.}
\]

The iteration starts with sampling \( \lambda > 1 \) candidate solutions \( x_i \in \mathbb{R}^n \) from a multivariate normal distribution \( \mathcal{N}(m_k, \sigma_k^2 C_k) \), i.e. for \( i = 1, \ldots, \lambda \)

\[
x_i \sim \mathcal{N}(m_k, \sigma_k^2 C_k) \\
\approx m_k + \sigma_k \times \mathcal{N}(0, C_k)
\]

The second line suggests the interpretation as perturbation (mutation) of the current favorite solution vector \( m_k \) (the distribution mean vector). The candidate solutions \( x_i \) are evaluated on the objective function \( f : \mathbb{R}^n \rightarrow \mathbb{R} \) to be minimized. Denoting the \( f \)-sorted candidate solutions as

\[
\{x_{i: \lambda} \mid i = 1 \ldots \lambda \} = \{x_i \mid i = 1 \ldots \lambda \} \text{ and } f(x_{1: \lambda}) \leq \cdots \leq f(x_{\mu: \lambda}) \leq f(x_{\mu+1: \lambda}) \cdots,
\]

the new mean value is computed as

\[
m_{k+1} = \sum_{i=1}^{\mu} w_i x_{i: \lambda}
\]

\[
= m_k + \sum_{i=1}^{\mu} w_i (x_{i: \lambda} - m_k)
\]

where the positive (recombination) weights \( w_1 \geq w_2 \geq \ldots \geq w_\mu > 0 \) sum to one. Typically, \( \mu \leq \lambda/2 \) and the weights are chosen such that \( \mu_w := 1/\sum_{i=1}^{\mu} w_i^2 \approx \lambda/4 \). The only feedback used from the objective function here and in the following is an ordering of the sampled candidate solutions due to the indices \( i : \lambda \).

The step-size \( \sigma_k \) is updated using cumulative step-size adaptation (CSA), sometimes also denoted as path length control. The evolution path (or search path) \( p_\sigma \) is updated first.

\[
p_\sigma \leftarrow \left( 1 - c_\sigma \right) p_\sigma + \sqrt{1 - (1 - c_\sigma)^2} \sqrt{\mu_w} \ C_k^{-1/2} \frac{m_{k+1} - m_k}{\sigma_k}
\]

complements for discounted variance

\[
displacement \ of \ m
\]

\[
distributed \ as \ \mathcal{N}(0, I) \ \text{under neutral selection}
\]

\[
\sigma_{k+1} = \sigma_k \times \exp \left( c_\sigma \ E \left( \frac{\|p_\sigma\|}{E[\mathcal{N}(0, I)]} - 1 \right) \right)
\]

unbiased about 0 under neutral selection

where

\[
c_\sigma^{-1} \approx n/3 \text{ is the backward time horizon for the evolution path } p_\sigma \text{ and larger than one,}
\]

\[
\mu_w = \left( \frac{1}{\sum_{i=1}^{\mu} w_i^2} \right)^{-1} \text{ is the variance effective selection mass and } 1 \leq \mu_w \leq \mu \text{ by definition of } w_i,
\]

\[
C_k^{-1/2} = \sqrt{C_k^{-1}} = \sqrt{C_k^{-1}} \text{ is the unique symmetric square root of the inverse of } C_k, \text{ and}
\]

\[
d_\sigma \text{ is the damping parameter usually close to one. For } d_\sigma = \infty \text{ or } c_\sigma = 0 \text{ the step-size remains unchanged.}
The step-size \( \sigma_k \) is increased if and only if \( \|p_\sigma\| \) is larger than the expected value
\[
E\|N(0,1)\| = \sqrt{2} \Gamma((n+1)/2)/\Gamma(n/2) \\
\approx \sqrt{n} (1 - 1/(4n) + 1/(21n^2))
\]
and decreased if it is smaller. For this reason, the step-size update tends to make consecutive steps \( C_k^{-1} \)-conjugate, in that after the adaptation has been successful \( \left( \frac{m_{k+1} - m_k}{\sigma_{k+1}} \right)^T C_k^{-1} \frac{m_{k+1} - m_k}{\sigma_k} \approx 0 \).

Finally, the covariance matrix is updated, where again the respective evolution path is updated first.

\[
P_c \leftarrow (1 - c_c) P_c + \mathbf{1}_{[0,\alpha \sqrt{n}]}(\|P_\sigma\|) \sqrt{1 - (1 - c_c)^2} \frac{m_{k+1} - m_k}{\sigma_k}
\]
\[
C_{k+1} = (1 - c_1 - c_\mu + c_s) C_k + c_1 p_c p_c^T + c_\mu \sum_{i=1}^\mu w_i \frac{x_{i;\lambda} - m_k}{\sigma_k} \left( \frac{x_{i;\lambda} - m_k}{\sigma_k} \right)^T
\]

where \( T \) denotes the transpose and
\[
c_c^{-1} \approx n/4
\]
is the backward time horizon for the evolution path \( P_c \) and larger than one,
\[
\alpha \approx 1.5 \quad \text{and the indicator function } \mathbf{1}_{[0, \alpha \sqrt{n}]}(\|P_\sigma\|) \quad \text{evaluates to one iff } \|P_\sigma\| \in [0, \alpha \sqrt{n}]
\]
or, in other words, \( \|P_\sigma\| \leq \alpha \sqrt{n} \), which is usually the case,
\[
c_s = (1 - \mathbf{1}_{[0, \alpha \sqrt{n}]}(\|P_\sigma\|)^2) c_1 c_c (2 - c_c)
\]
makes partly up for the small variance loss in case the indicator is zero,
\[
c_1 \approx 2/n^2
\]
is the learning rate for the rank-one update of the covariance matrix and
\[
c_\mu \approx \mu_w/n^2
\]
is the learning rate for the rank- \( \mu \) update of the covariance matrix and must not exceed
\[
1 - c_1.
\]
The covariance matrix update tends to increase the likelihood for \( p_c \) and for \( (x_{i;\lambda} - m_k)/\sigma_k \) to be sampled from \( N(0, C_{k+1}) \). This completes the iteration step.

The number of candidate samples per iteration, \( \lambda \), is not determined a priori and can vary in a wide range. Smaller values, for example \( \lambda = 10 \), lead to more local search behavior. Larger values, for example \( \lambda = 100 \) with default value \( \mu_w \approx \lambda/4 \), render the search more global. Sometimes the algorithm is repeatedly restarted with increasing \( \lambda \) by a factor of two for each restart.\([2]\) Besides of setting \( \lambda \) (or possibly \( \mu \) instead, if for example \( \lambda \) is predetermined by the number of available processors), the above introduced parameters are not specific to the given objective function and therefore not meant to be modified by the user.

**Example code in Matlab/Octave**

```matlab
function xmin=purecmaes   % (mu/mu_w, lambda)-CMA-ES

% Initialization

strfitnessfct = 'frosenbrock'; % name of objective/fitness function
N = 20;   % number of objective variables/problem dimension
xmean = rand(N,1); % objective variables initial point
sigma = 0.5; % coordinate wise standard deviation (step size)

% User defined input parameters (need to be edited)
% User defined input parameters (need to be edited)
```

[1] CMA-ES

[2] CMA-ES
stopfitness = 1e-10; % stop if fitness < stopfitness (minimization)
stopeval = 1e3*N^2; % stop after stopeval number of function evaluations

% Strategy parameter setting: Selection
lambda = 4+floor(3*log(N)); % population size, offspring number
mu = lambda/2; % number of parents/points for recombination
weights = log(mu+1/2)-log(1:mu)'; % muXone array for weighted recombination
mu = floor(mu);
weights = weights/sum(weights); % normalize recombination weights array
mueff=sum(weights).^2/sum(weights.^2); % variance-effectiveness of sum w_i x_i

% Strategy parameter setting: Adaptation
cc = (4+mueff/N) / (N+4 + 2*mueff/N); % time constant for cumulation for C
for C
    cs = (mueff+2) / (N+mueff+5); % t-const for cumulation for sigma control
    c1 = 2 / ((N+1.3)^2+mueff); % learning rate for rank-one update of C
    cmu = 2 * (mueff-2+1/mueff) / ((N+2)^2+mueff); % and for rank-mu update
damps = 1 + 2*max(0, sqrt((mueff-1)/(N+1))-1) + cs; % damping for sigma
% usually close to 1

% Initialize dynamic (internal) strategy parameters and constants
pc = zeros(N,1); ps = zeros(N,1); % evolution paths for C and sigma
B = eye(N,N); % B defines the coordinate system
D = ones(N,1); % diagonal D defines the scaling
C = B * diag(D.^2) * B'; % covariance matrix C
invsqrtC = B * diag(D.^-1) * B'; % C^-1/2
eigeneval = 0; % track update of B and D
chiN=N^0.5*(1-1/(4*N)+1/(21*N^2)); % expectation of
% | |N(0,I)| | ==
norm(randn(N,1))

% --------------- Generation Loop
-----------------------------------
counteval = 0; % the next 40 lines contain the 20 lines of interesting code
while counteval < stopeval
    % Generate and evaluate lambda offspring
for k=1:lambda,
    arx(:,k) = xmean + sigma * B * (D .* randn(N,1)); % m + sig * Normal(0,C)
    arfitness(k) = feval(strfitnessfct, arx(:,k)); % objective
end

% Sort by fitness and compute weighted mean into xmean
[arfitness, arindex] = sort(arfitness); % minimization
xold = xmean;
xmean = arx(:,arindex(1:mu))*weights; % recombination, new mean value

% Cumulation: Update evolution paths
ps = (1-cs)*ps ...
    + sqrt(cs*(2-cs)*mueff) * invsqrC * (xmean-xold) / sigma;
hsig = norm(ps)/sqrt(1-(1-cs)^(2*counteval/lambda))/chiN < 1.4 + 2/(N+1);
if hsig=0
    + cmu * artmp * diag(weights) * artmp'; % plus rank mu update
end

% Adapt step size sigma
sigma = sigma * exp((cs/damps)*(norm(ps)/chiN - 1));

% Decomposition of C into B*diag(D.^2)*B' (diagonalization)
if counteval - eigeneval > lambda/(c1+cmu)/N/10 % to achieve O(N^2)
    eigeneval = counteval;
    C = triu(C) + triu(C,1)'; % enforce symmetry
    [B,D] = eig(C); % eigen decomposition, B==normalized eigenvectors
end
D = sqrt(diag(D)); % D is a vector of standard deviations now
invsqrC = B * diag(D.^-1) * B';

% Break, if fitness is good enough or condition exceeds 1e14, better termination methods are advisable
if arfitness(1) <= stopfitness || max(D) > 1e7 * min(D)
    break;
end

end % while, end generation loop

xmin = arx(:, arindex(1)); % Return best point of last iteration.
% Notice that xmean is expected to be even % better.

% ---------------------------------------------------------------
function f=frosenbrock(x)
    if size(x,1) < 2 error('dimension must be greater one'); end
    f = 100 * sum((x(1:end-1).^2 - x(2:end)).^2) +
    sum((x(1:end-1)-1).^2);

Theoretical Foundations

Given the distribution parameters—mean, variances and covariances—the normal probability distribution for sampling new candidate solutions is the maximum entropy probability distribution over \( \mathbb{R}^n \), that is, the sample distribution with the minimal amount of prior information built into the distribution. More considerations on the update equations of CMA-ES are made in the following.

Variable Metric

The CMA-ES implements a stochastic variable-metric method. In the very particular case of a convex-quadratic objective function

\[
f(x) = \frac{1}{2}(x - x^*)^T H(x - x^*)
\]

the covariance matrix \( C_k \) adapts to the inverse of the Hessian matrix \( H \), up to a scalar factor and small random fluctuations. More general, also on the function \( g \circ f \), where \( g \) is strictly increasing and therefore order preserving and \( f \) is convex-quadratic, the covariance matrix \( C_k \) adapts to \( H^{-1} \), up to a scalar factor and small random fluctuations.

Maximum-Likelihood Updates

The update equations for mean and covariance matrix maximize a likelihood while resembling an expectation-maximization algorithm. The update of the mean vector \( \mu \) maximizes a log-likelihood, such that

\[
m_{k+1} = \arg \max_{\mu} \sum_{i=1}^{\mu} w_i \log p_N(x_i; \mu | \mu)
\]

where

\[
\log p_N(x) = -\frac{1}{2} \log \det(2\pi C) - \frac{1}{2} (x - \mu)^T C^{-1} (x - \mu)
\]

denotes the log-likelihood of \( x \) from a multivariate normal distribution with mean \( \mu \) and any positive definite covariance matrix \( C \). To see that \( m_{k+1} \) is independent of \( C \) remark first that this is the case for any diagonal matrix \( C \), because the coordinate-wise maximizer is independent of a scaling factor. Then, rotation of the data points or choosing \( C \) non-diagonal are equivalent.

The rank- \( \mu \) update of the covariance matrix, that is, the right most summand in the update equation of \( C_k \), maximizes a log-likelihood in that
for $\mu \geq n$ (otherwise $C$ is singular, but substantially the same result holds for $\mu < n$). Here, $p_N(x|C)$ denotes the likelihood of $x$ from a multivariate normal distribution with zero mean and covariance matrix $C$. Therefore, for $c_1 = 0$ and $c_\mu = 1$, $C_{k+1}$ is the above maximum-likelihood estimator. See estimation of covariance matrices for details on the derivation.

**Natural Gradient Descent in the Space of Sample Distributions**

Akimoto *et al.*\cite{3} recently found that the update of the distribution parameters resembles the descend in direction of a sampled natural gradient of the expected objective function value $E f(x)$ (to be minimized), where the expectation is taken under the sample distribution. With the parameter setting of $c_\sigma = 0$ and $c_1 = 0$, i.e. without step-size control and rank-one update, CMA-ES can thus be viewed as an instantiation of Natural Evolution Strategies (NES).\cite{3,4} The natural gradient is independent of the parameterization of the distribution. Taken with respect to the parameters $\theta$ of the sample distribution $p$, the gradient of $E f(x)$ can be expressed as

$$\nabla_\theta E(f(x)|\theta) = \nabla_\theta \int_{\mathbb{R}^n} f(x)p(x)dx$$

$$= \int_{\mathbb{R}^n} f(x)\frac{p(x)}{p(x)}\nabla_\theta p(x)dx$$

$$= \int_{\mathbb{R}^n} f(x)p(x)\nabla_\theta \ln p(x)dx$$

$$= E(f(x)\nabla_\theta \ln p(x|\theta))$$

where $p(x) = p(x|\theta)$ depends on the parameter vector $\theta$, the so-called score function, $\nabla_\theta \ln p(x|\theta) = \frac{\nabla_\theta p(x)}{p(x)}$, indicates the relative sensitivity of $p$ w.r.t. $\theta$, and the expectation is taken with respect to the distribution $p$. The natural gradient of $E f(x)$, complying with the Fisher information metric (an informational distance measure between probability distributions and the curvature of the relative entropy), now reads

$$\hat{\nabla} E(f(x)|\theta) = F_\theta^{-1}\nabla_\theta E(f(x)|\theta)$$

where the Fisher information matrix $F_\theta$ is the expectation of the Hessian of $-\ln p$ and renders the expression independent of the chosen parameterization. Combining the previous equalities we get

$$\hat{\nabla} E(f(x)|\theta) = F_\theta^{-1}E(f(x)\nabla_\theta \ln p(x|\theta))$$

$$= E(f(x)F_\theta^{-1}\nabla_\theta \ln p(x|\theta))$$

A Monte Carlo approximation of the latter expectation takes the average over $\lambda$ samples from $p$

$$\hat{\nabla} \tilde{E}_\theta(f) := -\sum_{i=1}^\lambda w_i F_\theta^{-1}\nabla_\theta \ln p(x_{i\lambda}|\theta) \quad \text{with} \quad w_i = -f(x_{i\lambda})/\lambda$$

where the notation $i : \lambda$ from above is used and therefore $w_i$ are monotonously decreasing in $i$. We might use, for a more robust approximation, rather $w_i$ as defined in the CMA-ES and zero for $i > \mu$ and let

$$\theta = [m_k^T \text{vec}(\sigma_k^2 C_k)]^T \in \mathbb{R}^{n^2}$$

such that $p(\cdot|\theta)$ is the density of the multivariate normal distribution $N(m_k, \sigma_k^2 C_k)$. Then, we have an explicit expression for

$$F_\theta^{-1} = \begin{bmatrix}
\sigma_k^2 C_k & 0 \\
0 & 2\sigma_k^2 C_k \otimes \sigma_k^2 C_k
\end{bmatrix}$$

and for
\[
\ln p(x|\theta) = \ln p(x|m_k, \sigma_k^2 C_k) = -\frac{1}{2}(x-m_k)^T \sigma_k^{-2} C_k^{-1} (x-m_k) - \frac{1}{2} \ln \det(2\pi \sigma_k^2 C_k)
\]
and, after some calculations, the updates in the CMA-ES turn out as
\[
\begin{align}
m_{k+1} &= m_k - \underbrace{[\tilde{\nabla} \widehat{E}_\theta(f)]_{1,\ldots,n}}_{\text{natural gradient for mean}} \\
&= m_k + \sum_{i=1}^\lambda w_i (x_{i:\lambda} - m_k)
\end{align}
\]
and
\[
C_{k+1} = C_k + c_1 (p_c p_c^T - C_k) - \frac{c_\mu}{\sigma_k^2} \text{mat}([\nabla \widehat{E}_\theta(f)]_{n+1,\ldots,n+n^2})
\]
\[
= C_k + c_1 (p_c p_c^T - C_k) + \frac{c_\mu}{\sigma_k^2} \sum_{i=1}^\lambda w_i \left((x_{i:\lambda} - m_k)(x_{i:\lambda} - m_k)^T - \sigma_k^2 C_k\right)
\]
where \text{mat} forms the proper matrix from the respective natural gradient sub-vector. That means, setting \(c_1 = c_\sigma = 0\), the CMA-ES updates descend in direction of the approximation \(\nabla \widehat{E}_\theta(f)\) of the natural gradient while using different step-sizes (learning rates) for the orthogonal parameters \(m\) and \(C\) respectively.

**Stationarity or Unbiasedness**

It is comparatively easy to see that the update equations of CMA-ES satisfy some stationarity conditions, in that they are essentially unbiased. Under neutral selection, where \(x_{i:\lambda} \sim \mathcal{N}(m_k, \sigma_k^2 C_k)\), we find that
\[
E(m_{k+1} | m_k) = m_k
\]
and under some mild additional assumptions on the initial conditions
\[
E(\log \sigma_{k+1} | \sigma_k) = \log \sigma_k
\]
and with an additional minor correction in the covariance matrix update for the case where the indicator function evaluates to zero, we find
\[
E(C_{k+1} | C_k) = C_k
\]

**Invariance**

Invariance properties imply uniform performance on a class of objective functions. They have been argued to be an advantage, because they allow to generalize and predict the behavior of the algorithm and therefore strengthen the meaning of empirical results obtained on single functions. The following invariance properties have been established for CMA-ES.

- Invariance under order-preserving transformations of the objective function value \(f\), in that for any \(h : \mathbb{R}^n \to \mathbb{R}\) the behavior is identical on \(f : x \mapsto g(h(x))\) for all strictly increasing \(g : \mathbb{R} \to \mathbb{R}\). This invariance is easy to verify, because only the \(f\)-ranking is used in the algorithm, which is invariant under the choice of \(g\).
- Scale-invariance, in that for any \(h : \mathbb{R}^n \to \mathbb{R}\) the behavior is independent of \(\alpha > 0\) for the objective function \(f : x \mapsto h(\alpha x)\) given \(\sigma_0 \propto 1/\alpha\) and \(m_0 \propto 1/\alpha\).
- Invariance under rotation of the search space in that for any \(h : \mathbb{R}^n \to \mathbb{R}\) and any \(z \in \mathbb{R}^n\) the behavior on \(f : x \mapsto h(Rx)\) is independent of the orthogonal matrix \(R\), given \(m_0 = R^{-1}z\). More general, the algorithm is also invariant under general linear transformations \(R\) when additionally the initial covariance matrix is chosen as \(R^{-1}R^{-1}^T\).

Any serious parameter optimization method should be translation invariant, but most methods do not exhibit all the above described invariance properties. A prominent example with the same invariance properties is the
Nelder–Mead method, where the initial simplex must be chosen respectively.

**Convergence**

Conceptual considerations like the scale-invariance property of the algorithm, the analysis of simpler evolution strategies, and overwhelming empirical evidence suggest that the algorithm converges on a large class of functions fast to the global optimum, denoted as $x^*$. On some functions, convergence occurs independently of the initial conditions with probability one. On some functions the probability is smaller than one and typically depends on the initial $m_0$ and $\sigma_0$. Empirically, the fastest possible convergence rate in $k$ for rank-based direct search methods can often be observed (depending on the context denoted as linear or log-linear or exponential convergence). Informally, we can write

$$\|m_k - x^*\| \approx \|m_0 - x^*\| \times e^{-ck}$$

for some $c > 0$, and more rigorously

$$\frac{1}{k} \sum_{i=1}^{k} \log \frac{\|m_i - x^*\|}{\|m_0 - x^*\|} = \frac{1}{k} \log \frac{\|m_k - x^*\|}{\|m_0 - x^*\|} \to -c < 0 \quad \text{for } k \to \infty,$$

or similarly,

$$E \log \frac{\|m_k - x^*\|}{\|m_{k-1} - x^*\|} \to -c < 0 \quad \text{for } k \to \infty.$$

This means that on average the distance to the optimum is decreased in each iteration by a "constant" factor, namely by $\exp(-c)$. The convergence rate $c$ is roughly $0.1\lambda/n$, given $\lambda$ is not much larger than the dimension $n$.

Even with optimal $\sigma$ and $C$, the convergence rate $c$ cannot largely exceed $0.25\lambda/n$, given the above recombination weights $w_i$ are all non-negative. The actual linear dependencies in $\lambda$ and $n$ are remarkable and they are in both cases the best one can hope for in this kind of algorithm. Yet, a rigorous proof of convergence is missing.

**Interpretation as Coordinate System Transformation**

Using a non-identity covariance matrix for the multivariate normal distribution in evolution strategies is equivalent to a coordinate system transformation of the solution vectors, primarily because the sampling equation

$$x_i \sim m_k + \sigma_k \times N(0, C_k)$$

$$\sim m_k + \sigma_k \times C_k^{-1/2} N(0, I)$$

can be equivalently expressed in an "encoded space" as

$$C_k^{-1/2} x_i \sim C_k^{-1/2} m_k + \sigma_k \times N(0, I)$$

represented in the encoded space.

The covariance matrix defines a bijective transformation (encoding) for all solution vectors into a space, where the sampling takes place with identity covariance matrix. Because the update equations in the CMA-ES are invariant under coordinate system transformations (general linear transformations), the CMA-ES can be re-written as an adaptive encoding procedure applied to a simple evolution strategy with identity covariance matrix. This adaptive encoding procedure is not confined to algorithms that sample from a multivariate normal distribution (like evolution strategies), but can in principle be applied to any iterative search method.
Performance in Practice

In contrast to most other evolutionary algorithms, the CMA-ES is, from the users perspective, quasi parameter-free. However, the number of candidate samples \( \lambda \) (population size) can be adjusted by the user in order to change the characteristic search behavior (see above). CMA-ES has been empirically successful in hundreds of applications and is considered to be useful in particular on non-convex, non-separable, ill-conditioned, multi-modal or noisy objective functions. The search space dimension ranges typically between two and a few hundred. Assuming a black-box optimization scenario, where gradients are not available (or not useful) and function evaluations are the only considered cost of search, the CMA-ES method is likely to be outperformed by other methods in the following conditions:

- on low-dimensional functions, say \( n < 5 \), for example by the downhill simplex method or surrogate-based methods (like kriging with expected improvement);
- on separable functions without or with only negligible dependencies between the design variables in particular in the case of multi-modality or large dimension, for example by differential evolution;
- on (nearly) convex-quadratic functions with low or moderate condition number of the Hessian matrix, where BFGS or NEWUOA are typically ten times faster;
- on functions that can already be solved with a comparatively small number of function evaluations, say no more than \( 10n \), where CMA-ES is often slower than, for example, NEWUOA or Multilevel Coordinate Search (MCS).

On separable functions the performance disadvantage is likely to be most significant, in that CMA-ES might not be able to find at all comparable solutions. On the other hand, on non-separable functions that are ill-conditioned or rugged or can only be solved with more than \( 100n \) function evaluations, the CMA-ES shows most often superior performance.

Variations and Extensions

The (1+1)-CMA-ES \([6]\) generates only one candidate solution per iteration step which only becomes the new distribution mean, if it is better than the old mean. For \( c_e = 1 \) it is a close variant of Gaussian adaptation. The CMA-ES has also been extended to multiobjective optimization as MO-CMA-ES \([7]\). Another remarkable extension has been the addition of a negative update of the covariance matrix with the so-called active CMA \([8]\).

References


Bibliography


External links

- A short introduction to CMA-ES by N. Hansen (http://www.lri.fr/~hansen/cmaesintro.html)
- CMA-ES source code page (http://www.lri.fr/~hansen/cmaes_inmatlab.html)

Cultural algorithm

*Cultural algorithms* (CA) are a branch of evolutionary computation where there is a knowledge component that is called the belief space in addition to the population component. In this sense, cultural algorithms can be seen as an extension to a conventional genetic algorithm. Cultural algorithms were introduced by Reynolds (see references).

Belief space

The belief space of a cultural algorithm is divided into distinct categories. These categories represent different domains of knowledge that the population has of the search space.

The belief space is updated after each iteration by the best individuals of the population. The best individuals can be selected using a fitness function that assesses the performance of each individual in population much like in genetic algorithms.

List of belief space categories

- **Normative knowledge** A collection of desirable value ranges for the individuals in the population component e.g. acceptable behavior for the agents in population.
- **Domain specific knowledge** Information about the domain of the problem CA is applied to.
- **Situational knowledge** Specific examples of important events - e.g. successful/unsuccessful solutions
- **Temporal knowledge** History of the search space - e.g. the temporal patterns of the search process
- **Spatial knowledge** Information about the topography of the search space
Cultural algorithm

Population
The population component of the cultural algorithm is approximately the same as that of the genetic algorithm.

Communication protocol
Cultural algorithms require an interface between the population and belief space. The best individuals of the population can update the belief space via the update function. Also, the knowledge categories of the belief space can affect the population component via the influence function. The influence function can affect population by altering the genome or the actions of the individuals.

Pseudo-code for cultural algorithms
1. Initialize population space (choose initial population)
2. Initialize belief space (e.g. set domain specific knowledge and normative value-ranges)
3. Repeat until termination condition is met
   1. Perform actions of the individuals in population space
   2. Evaluate each individual by using the fitness function
   3. Select the parents to reproduce a new generation of offspring
   4. Let the belief space alter the genome of the offspring by using the influence function
   5. Update the belief space by using the accept function (this is done by letting the best individuals to affect the belief space)

Applications
• Various optimization problems
• Social simulation

References

References
Learning classifier system

A learning classifier system, or LCS, is a machine learning system with close links to reinforcement learning and genetic algorithms. First described by John Holland, his LCS consisted of a population of binary rules on which a genetic algorithm altered and selected the best rules. Rule fitness was based on a reinforcement learning technique.

Learning classifier systems can be split into two types depending upon where the genetic algorithm acts. A Pittsburgh-type LCS has a population of separate rule sets, where the genetic algorithm recombines and reproduces the best of these rule sets. In a Michigan-style LCS there is only a single set of rules in a population and the algorithm's action focuses on selecting the best classifiers within that set. Michigan-style LCSs have two main types of fitness definitions, strength-based (e.g. ZCS) and accuracy-based (e.g. XCS). The term "learning classifier system" most often refers to Michigan-style LCSs.

Initially the classifiers or rules were binary, but recent research has expanded this representation to include real-valued, neural network, and functional (S-expression) conditions.

Learning classifier systems are not fully understood mathematically and doing so remains an area of active research. Despite this, they have been successfully applied in many problem domains.

Overview

A learning classifier system (LCS) is an adaptive system that learns to perform the best action given its input. By "best" is generally meant the action that will receive the most reward or reinforcement from the system's environment. By "input" is meant the environment as sensed by the system, usually a vector of numerical values. The set of available actions depends on the decision context, for instance a financial one, the actions might be "buy", "sell", etc. In general, an LCS is a simple model of an intelligent agent interacting with an environment.

An LCS is "adaptive" in the sense that its ability to choose the best action improves with experience. The source of the improvement is reinforcement—technically, payoff—provided by the environment. In many cases, the payoff is arranged by the experimenter or trainer of the LCS. For instance, in a classification context, the payoff may be 1.0 for "correct" and 0.0 for "incorrect". In a robotic context, the payoff could be a number representing the change in distance to a recharging source, with more desirable changes (getting closer) represented by larger positive numbers, etc. Often, systems can be set up so that effective reinforcement is provided automatically, for instance via a distance sensor. Payoff received for a given action is used by the LCS to alter the likelihood of taking that action, in those circumstances, in the future. To understand how this works, it is necessary to describe some of the LCS mechanics.

Inside the LCS is a set—technically, a population—of "condition-action rules" called classifiers. There may be hundreds of classifiers in the population. When a particular input occurs, the LCS forms a so-called match set of classifiers whose conditions are satisfied by that input. Technically, a condition is a truth function \( t(x) \) which is satisfied for certain input vectors \( x \). For instance, in a certain classifier, it may be that \( t(x) = 1 \) (true) for \( 43 < x_3 < 54 \), where \( x_3 \) is a component of \( x \), and represents, say, the age of a medical patient. In general, a classifier's condition will refer to more than one of the input components, usually all of them. If a classifier's condition is satisfied, i.e. its \( t(x) = 1 \), then that classifier joins the match set and influences the system's action decision. In a sense, the match set consists of classifiers in the population that recognize the current input.

Among the classifiers—the condition-action rules—of the match set will be some that advocate one of the possible actions, some that advocate another of the actions, and so forth. Besides advocating an action, a classifier will also contain a prediction of the amount of payoff which, speaking loosely, "it thinks" will be received if the system takes that action. How can the LCS decide which action to take? Clearly, it should pick the action that is likely to receive the highest payoff, but with all the classifiers making (in general) different predictions, how can it decide? The technique adopted is to compute, for each action, an average of the predictions of the classifiers advocating that action—and then choose the action with the largest average. The prediction average is in fact weighted by another
classifier quantity, its fitness, which will be described later but is intended to reflect the reliability of the classifier's prediction.

The LCS takes the action with the largest average prediction, and in response the environment returns some amount of payoff. If it is in a learning mode, the LCS will use this payoff, \( P \), to alter the predictions of the responsible classifiers, namely those advocating the chosen action; they form what is called the action set. In this adjustment, each action set classifier's prediction \( p \) is changed mathematically to bring it slightly closer to \( P \), with the aim of increasing its accuracy. Besides its prediction, each classifier maintains an estimate \( \epsilon \) of the error of its predictions. Like \( p \), \( \epsilon \) is adjusted on each learning encounter with the environment by moving \( \epsilon \) slightly closer to the current absolute error \( |p - P| \). Finally, a quantity called the classifier's fitness is adjusted by moving it closer to an inverse function of \( \epsilon \), which can be regarded as measuring the accuracy of the classifier. The result of these adjustments will hopefully be to improve the classifier's prediction and to derive a measure—the fitness—that indicates its accuracy.

The adaptivity of the LCS is not, however, limited to adjusting classifier predictions. At a deeper level, the system treats the classifiers as an evolving population in which accurate—i.e. high fitness—classifiers are reproduced over less accurate ones and the "offspring" are modified by genetic operators such as mutation and crossover. In this way, the population of classifiers gradually changes over time, that is, it adapts structurally. Evolution of the population is the key to high performance since the accuracy of predictions depends closely on the classifier conditions, which are changed by evolution.

Evolution takes place in the background as the system is interacting with its environment. Each time an action set is formed, there is finite chance that a genetic algorithm will occur in the set. Specifically, two classifiers are selected from the set with probabilities proportional to their fitnesses. The two are copied and the copies (offspring) may, with certain probabilities, be mutated and recombined ("crossed"). Mutation means changing, slightly, some quantity or aspect of the classifier condition; the action may also be changed to one of the other actions. Crossover means exchanging parts of the two classifiers. Then the offspring are inserted into the population and two classifiers are deleted to keep the population at a constant size. The new classifiers, in effect, compete with their parents, which are still (with high probability) in the population.

The effect of classifier evolution is to modify their conditions so as to increase the overall prediction accuracy of the population. This occurs because fitness is based on accuracy. In addition, however, the evolution leads to an increase in what can be called the "accurate generality" of the population. That is, classifier conditions evolve to be as general as possible without sacrificing accuracy. Here, general means maximizing the number of input vectors that the condition matches. The increase in generality results in the population needing fewer distinct classifiers to cover all inputs, which means (if identical classifiers are merged) that populations are smaller, and also that the knowledge contained in the population is more visible to humans—which is important in many applications. The specific mechanism by which generality increases is a major, if subtle, side-effect of the overall evolution.

Summarizing, a learning classifier system is a broadly-applicable adaptive system that learns from external reinforcement and through an internal structural evolution derived from that reinforcement. In addition to adaptively increasing its performance, the LCS develops knowledge in the form of rules that respond to different aspects of the environment and capture environmental regularities through the generality of their conditions.

Many important aspects of LCS were omitted in the above presentation, including among others: use in sequential (multi-step) tasks, modifications for non-Markov (locally ambiguous) environments, learning in the presence of noise, incorporation of continuous-valued actions, learning of relational concepts, learning of hyper-heuristics, and use for on-line function approximation and clustering. An LCS appears to be a widely applicable cognitive/agent model that can act as a framework for a diversity of learning investigations and practical applications.
Memetic algorithm

Memetic algorithms (MA) represent one of the recent growing areas of research in evolutionary computation. The term MA is now widely used as a synergy of evolutionary or any population-based approach with separate individual learning or local improvement procedures for problem search. Quite often, MA are also referred to in the literature as Baldwinian Evolutionary algorithms (EA), Lamarckian EAs, cultural algorithms or genetic local search.

Introduction

The theory of "Universal Darwinism" was coined by Richard Dawkins in 1983\(^1\) to provide a unifying framework governing the evolution of any complex system. In particular, "Universal Darwinism" suggests that evolution is not exclusive to biological systems, i.e., it is not confined to the narrow context of the genes, but applicable to any complex system that exhibit the principles of inheritance, variation and selection, thus fulfilling the traits of an evolving system. For example, the new science of memetics represents the mind-universe analogue to genetics in culture evolution that stretches across the fields of biology, cognition and psychology, which has attracted significant attention in the last decades. The term "meme" was also introduced and defined by Dawkins in 1976\(^2\) as "the basic unit of cultural transmission, or imitation", and in the Oxford English Dictionary as "an element of culture that may be considered to be passed on by non-genetic means".

Inspired by both Darwinian principles of natural evolution and Dawkins’ notion of a meme, the term "Memetic Algorithm" (MA) was first introduced by Moscato in his technical report\(^3\) in 1989 where he viewed MA as being close to a form of population-based hybrid genetic algorithm (GA) coupled with an individual learning procedure capable of performing local refinements. The metaphorical parallels, on the one hand, to Darwinian evolution and, on the other hand, between memes and domain specific (local search) heuristics are captured within memetic algorithms thus rendering a methodology that balances well between generality and problem specificity. In a more diverse context, memetic algorithms are now used under various names including Hybrid Evolutionary Algorithms, Baldwinian Evolutionary Algorithms, Lamarckian Evolutionary Algorithms, Cultural Algorithms or Genetic Local Search. In the context of complex optimization, many different instantiations of memetic algorithms have been reported across a wide range of application domains, in general, converging to high quality solutions more efficiently than their conventional evolutionary counterparts.

In general, using the ideas of memetics within a computational framework is called "Memetic Computing" (MC).\(^4\)\(^5\) With MC, the traits of Universal Darwinism are more appropriately captured. Viewed in this perspective, MA is a more constrained notion of MC. More specifically, MA covers one area of MC, in particular dealing with areas of evolutionary algorithms that marry other deterministic refinement techniques for solving optimization problems. MC extends the notion of memes to cover conceptual entities of knowledge-enhanced procedures or
The development of MAs

1st generation

The first generation of MA refers to hybrid algorithms, a marriage between a population-based global search (often in the form of an evolutionary algorithm) coupled with a cultural evolutionary stage. This first generation of MA although encompasses characteristics of cultural evolution (in the form of local refinement) in the search cycle, it may not qualify as a true evolving system according to Universal Darwinism, since all the core principles of inheritance/memetic transmission, variation and selection are missing. This suggests why the term MA stirred up criticisms and controversies among researchers when first introduced.\(^3\)

Pseudo code:

```
Procedure Memetic Algorithm
  Initialize: Generate an initial population;
  while Stopping conditions are not satisfied do
    Evaluate all individuals in the population.
    Evolve a new population using stochastic search operators.
    Select the subset of individuals, ΩIu, that should undergo the individual improvement procedure.
      for each individual in ΩIu do
        Perform individual learning using meme(s) with frequency or probability of \( f_{iu} \), for a period of \( t_{iu} \).
        Proceed with Lamarckian or Baldwinian learning.
      end for
  end while
```

2nd generation

Multi-meme,\(^6\) Hyper-heuristic\(^7\) and Meta-Lamarckian MA\(^8\) are referred to as second generation MA exhibiting the principles of memetic transmission and selection in their design. In Multi-meme MA, the memetic material is encoded as part of the genotype. Subsequently, the decoded meme of each respective individual / chromosome is then used to perform a local refinement. The memetic material is then transmitted through a simple inheritance mechanism from parent to offspring(s). On the other hand, in hyper-heuristic and meta-Lamarckian MA, the pool of candidate memes considered will compete, based on their past merits in generating local improvements through a reward mechanism, deciding on which meme to be selected to proceed for future local refinements. Memes with a higher reward have a greater chance of being replicated or copied. For a review on second generation MA, i.e., MA considering multiple individual learning methods within an evolutionary system, the reader is referred to.\(^9\)

3rd generation

Co-evolution\(^10\) and self-generating MAs\(^11\) may be regarded as 3rd generation MA where all three principles satisfying the definitions of a basic evolving system have been considered. In contrast to 2nd generation MA which assumes that the memes to be used are known a priori, 3rd generation MA utilizes a rule-based local search to supplement candidate solutions within the evolutionary system, thus capturing regularly repeated features or patterns in the problem space.
Some design notes
The frequency and intensity of individual learning directly define the degree of evolution (exploration) against individual learning (exploitation) in the MA search, for a given fixed limited computational budget. Clearly, a more intense individual learning provides greater chance of convergence to the local optima but limits the amount of evolution that may be expended without incurring excessive computational resources. Therefore, care should be taken when setting these two parameters to balance the computational budget available in achieving maximum search performance. When only a portion of the population individuals undergo learning, the issue on which subset of individuals to improve need to be considered to maximize the utility of MA search. Last but not least, the individual learning procedure/meme used also favors a different neighborhood structure, hence the need to decide which meme or memes to use for a given optimization problem at hand would be required.

How often should individual learning be applied?
One of the first issues pertinent to memetic algorithm design is to consider how often the individual learning should be applied, i.e., individual learning frequency. In one case,[12] the effect of individual learning frequency on MA search performance was considered where various configurations of the individual learning frequency at different stages of the MA search were investigated. Conversely, it was shown elsewhere[13] that it may be worthwhile to apply individual learning on every individual if the computational complexity of the individual learning is relatively low.

On which solutions should individual learning be used?
On the issue of selecting appropriate individuals among the EA population that should undergo individual learning, fitness-based and distribution-based strategies were studied for adapting the probability of applying individual learning on the population of chromosomes in continuous parametric search problems with Land[14] extending the work to combinatorial optimization problems. Bambha et al. introduced a simulated heating technique for systematically integrating parameterized individual learning into evolutionary algorithms to achieve maximum solution quality.[15]

How long should individual learning be run?
Individual learning intensity, $t_{IL}$, is the amount of computational budget allocated to an iteration of individual learning, i.e., the maximum computational budget allowable for individual learning to expend on improving a single solution.

What individual learning method or meme should be used for a particular problem or individual?
In the context of continuous optimization, individual learning/individual learning exists in the form of local heuristics or conventional exact enumerative methods.[16] Examples of individual learning strategies include the hill climbing, Simplex method, Newton/Quasi-Newton method, interior point methods, conjugate gradient method, line search and other local heuristics. Note that most of common individual learninger are deterministic.

In combinatorial optimization, on the other hand, individual learning methods commonly exists in the form of heuristics (which can be deterministic or stochastic), that are tailored to serve a problem of interest well. Typical heuristic procedures and schemes include the k-gene exchange, edge exchange, first-improvement, and many others.
Applications

Memetic algorithms are the subject of intense scientific research (a scientific journal devoted to their research is going to be launched) and have been successfully applied to a multitude of real-world problems. Although many people employ techniques closely related to memetic algorithms, alternative names such as hybrid genetic algorithms are also employed. Furthermore, many people term their memetic techniques as genetic algorithms. The widespread use of this misnomer hampers the assessment of the total amount of applications.

Researchers have used memetic algorithms to tackle many classical NP problems. To cite some of them: graph partitioning, multidimensional knapsack, travelling salesman problem, quadratic assignment problem, set cover problem, minimal graph colouring, max independent set problem, bin packing problem and generalized assignment problem.

More recent applications include (but are not limited to): training of artificial neural networks, pattern recognition, robotic motion planning, beam orientation, circuit design, electric service restoration, medical expert systems, single machine scheduling, automatic timetabling (notably, the timetable for the NHL), manpower scheduling, nurse rostering and function optimisation, processor allocation, maintenance scheduling (for example, of an electric distribution network), multidimensional knapsack problem, VLSI design, clustering of gene expression profiles, feature/gene selection, and multi-class, multi-objective feature selection.

Recent Activities in Memetic Algorithms

- IEEE Workshop on Memetic Algorithms (WOMA 2009). Program Chairs: Jim Smith, University of the West of England, U.K.; Yew-Soon Ong, Nanyang Technological University, Singapore; Gustafson Steven, University of Nottingham; U.K.; Meng Hiot Lim, Nanyang Technological University, Singapore; Natalio Krasnogor, University of Nottingham, U.K.
- 2008 IEEE World Congress on Computational Intelligence (WCCI 2008), Hong Kong, Special Session on Memetic Algorithms.
- IEEE Computational Intelligence Society Emergent Technologies Task Force on Memetic Computing.
- IEEE Congress on Evolutionary Computation (CEC 2007), Singapore, Special Session on Memetic Algorithms.
- ‘Memetic Computing’ by Thomson Scientific’s Essential Science Indicators as an Emerging Front Research Area.
References


Meta-optimization

In numerical optimization, meta-optimization is the use of one optimization method to tune another optimization method. Meta-optimization is reported to have been used as early as in the late 1970s by Mercer and Sampson[^1^] for finding optimal parameter settings of a genetic algorithm. Meta-optimization is also known in the literature as meta-evolution, super-optimization, automated parameter calibration, hyper-heuristics, etc.

Motivation

Optimization methods such as genetic algorithm and differential evolution have several parameters that govern their behaviour and efficacy in optimizing a given problem and these parameters must be chosen by the practitioner to achieve satisfactory results. Selecting the behavioural parameters by hand is a laborious task that is susceptible to human misconceptions of what makes the optimizer perform well.

The behavioural parameters of an optimizer can be varied and the optimization performance plotted as a landscape. This is computationally feasible for optimizers with few behavioural parameters and optimization problems that are fast to compute, but when the number of behavioural parameters increases the time usage for computing such a performance landscape increases exponentially. This is the curse of dimensionality for the search-space consisting of an optimizer’s behavioural parameters. An efficient method is therefore needed to search the space of behavioural parameters.


Methods

A simple way of finding good behavioural parameters for an optimizer is to employ another overlaying optimizer, called the meta-optimizer. There are different ways of doing this depending on whether the behavioural parameters to be tuned are real-valued or discrete-valued, and depending on what performance measure is being used, etc.

Meta-optimizing the parameters of a genetic algorithm was done by Grefenstette [2] and Keane, [3] amongst others, and experiments with meta-optimizing both the parameters and the genetic operators were reported by Bäck. [4] Meta-optimization of particle swarm optimization was done by Meissner et al. [5] as well as by Pedersen and Chipperfield, [6] who also meta-optimized differential evolution. Birattari et al. [7][8] meta-optimized ant colony optimization. Statistical models have also been used to reveal more about the relationship between choices of behavioural parameters and optimization performance, see for example Francois and Lavergne, [9] and Nannen and Eiben. [10] A comparison of various meta-optimization techniques was done by Smit and Eiben. [11]

References

Cellular evolutionary algorithm

A Cellular Evolutionary Algorithm (cEA) is a kind of evolutionary algorithm (EA) in which individuals cannot mate arbitrarily, but every one interacts with its closer neighbors on which a basic EA is applied (selection, variation, replacement).

The cellular model simulates Natural evolution from the point of view of the individual, which encodes a tentative (optimization, learning, search) problem solution. The essential idea of this model is to provide the EA population with a special structure defined as a connected graph, in which each vertex is an individual who communicates with his nearest neighbors. Particularly, individuals are conceptually set in a toroidal mesh, and are only allowed to recombine with close individuals. This leads us to a kind of locality known as isolation by distance. The set of potential mates of an individual is called its neighborhood. It is known that, in this kind of algorithm, similar individuals tend to cluster creating niches, and these groups operate as if they were separate sub-populations (islands). Anyway, there is no clear borderline between adjacent groups, and close niches could be easily colonized by competitive niches and maybe merge solution contents during the process. Simultaneously, farther niches can be affected more slowly.
**Introduction**

A Cellular Evolutionary Algorithm (cEA) usually evolves a structured bidimensional grid of individuals, although other topologies are also possible. In this grid, clusters of similar individuals are naturally created during evolution, promoting exploration in their boundaries, while exploitation is mainly performed by direct competition and merging inside them.

The grid is usually 2D toroidal structure, although the number of dimensions can be easily extended (to 3D) or reduced (to 1D, e.g. a ring). The neighborhood of a particular point of the grid (where an individual is placed) is defined in terms of the Manhattan distance from it to others in the population. Each point of the grid has a neighborhood that overlaps the neighborhoods of nearby individuals. In the basic algorithm, all the neighborhoods have the same size and identical shapes. The two most commonly used neighborhoods are L5, also called Von Neumann or NEWS (North, East, West and South), and C9, also known as Moore neighborhood. Here, L stands for Linear while C stands for Compact.

In cEAs, the individuals can only interact with their neighbors in the reproductive cycle where the variation operators are applied. This reproductive cycle is executed inside the neighborhood of each individual and, generally, consists in selecting two parents among its neighbors according to a certain criterion, applying the variation operators to them (recombination and mutation for example), and replacing the considered individual by the recently created offspring following a given criterion, for instance, replace if the offspring represents a better solution than the considered individual.

**Synchronous versus Asynchronous cEAs**

In a regular synchronous cEA, the algorithm proceeds from the very first top left individual to the right and then to the several rows by using the information in the population to create a new temporary population. After finishing with the bottom-right last individual the temporary population is full with the newly computed individuals, and the replacement step starts. In it, the old population is completely and synchronously replaced with the newly computed one according to some criterion. Usually, the replacement keeps the best individual in the same position of both populations, that is, elitism is used.

We must notice that according to the update policy of the population used, we could also define an asynchronous cEA. This is also a well-known issue in cellular automata. In asynchronous cEAs the order in which the individuals in the grid are update changes depending on the criterion used: line sweep, fixed random sweep, new random sweep, and uniform choice. These are the four most usual ways of updating the population. All of them keep using the newly computed individual (or the original if better) for the computations of its neighbors immediately. This makes the population to hold at any time individual in different states of evolution, defining a very interesting new line of research.
The overlap of the neighborhoods provides an implicit mechanism of solution migration to the cEA. Since the best solutions spread smoothly through the whole population, genetic diversity in the population is preserved longer than in non-structured EAs. This soft dispersion of the best solutions through the population is one of the main issues of the good tradeoff between exploration and exploitation that cEAs perform during the search. It is then easy to see that we could tune this tradeoff (and hence, tune the genetic diversity level along the evolution) by modifying (for instance) the size of the neighborhood used, as the overlap degree between the neighborhoods grows according to the size of the neighborhood.

A cEA can be seen as a cellular automaton (CA) with probabilistic rewritable rules, where the alphabet of the CA is equivalent to the potential number of solutions of the problem. Hence, if we see cEAs as a kind of CA, it is possible to import knowledge from the field of CAs to cEAs, and in fact this is an interesting open research line.

**Parallelism and cEAs**

Cellular EAs are very amenable to parallelism, thus usually found in the literature of parallel metaheuristics. In particular, fine grain parallelism can be use to assign independent threads of execution to every individual, thus allowing the whole cEA to run on a concurrent or actually parallel hardware platform. In this way, large time reductions can be got when running cEAs on FPGAs or GPUs. However, it is important to stress that cEAs are a model of search, in many senses different to traditional EAs. Also, they can be run in sequential and parallel platforms, reinforcing the fact that the model and the implementation are two different concepts.

See here [3] for a complete description on the fundamentals for the understanding, design, and application of cEAs.

**References**

External links

• THE site on Cellular Evolutionary Algorithms (http://neo.lcc.uma.es/cEA-web/)
• NEO Research Group at University of Málaga, Spain (http://neo.lcc.uma.es)

Cellular automaton

A **cellular automaton** (pl. **cellular automata**, abbrev. CA) is a discrete model studied in computability theory, mathematics, physics, complexity science, theoretical biology and microstructure modeling. It consists of a regular grid of **cells**, each in one of a finite number of **states**, such as "On" and "Off" (in contrast to a coupled map lattice). The grid can be in any finite number of dimensions. For each cell, a set of cells called its **neighborhood** (usually including the cell itself) is defined relative to the specified cell. For example, the neighborhood of a cell might be defined as the set of cells a distance of 2 or less from the cell. An initial state (time \( t=0 \)) is selected by assigning a state for each cell. A new **generation** is created (advancing \( t \) by 1), according to some fixed rule (generally, a mathematical function) that determines the new state of each cell in terms of the current state of the cell and the states of the cells in its neighborhood. For example, the rule might be that the cell is "On" in the next generation if exactly two of the cells in the neighborhood are "On" in the current generation, otherwise the cell is "Off" in the next generation. Typically, the rule for updating the state of cells is the same for each cell and does not change over time, and is applied to the whole grid simultaneously, though exceptions are known.

Cellular automata are also called "cellular spaces", "tessellation automata", "homogeneous structures", "cellular structures", "tessellation structures", and "iterative arrays".[2]

**Overview**

One way to simulate a two-dimensional cellular automaton is with an infinite sheet of graph paper along with a set of rules for the cells to follow. Each square is called a "cell" and each cell has two possible states, black and white. The "neighbors" of a cell are the 8 squares touching it. For such a cell and its neighbors, there are 512 (= \( 2^9 \)) possible patterns. For each of the 512 possible patterns, the rule table would state whether the center cell will be black or white on the next time interval. Conway's Game of Life is a popular version of this model.

It is usually assumed that every cell in the universe starts in the same state, except for a finite number of cells in other states, often called a **configuration**. More generally, it is sometimes assumed that the universe starts out covered with a periodic pattern, and only a finite number of cells violate that pattern. The latter assumption is common in one-dimensional cellular automata.
Cellular automata are often simulated on a finite grid rather than an infinite one. In two dimensions, the universe would be a rectangle instead of an infinite plane. The obvious problem with finite grids is how to handle the cells on the edges. How they are handled will affect the values of all the cells in the grid. One possible method is to allow the values in those cells to remain constant. Another method is to define neighbourhoods differently for these cells. One could say that they have fewer neighbours, but then one would also have to define new rules for the cells located on the edges. These cells are usually handled with a toroidal arrangement: when one goes off the top, one comes in at the corresponding position on the bottom, and when one goes off the left, one comes in on the right. (This essentially simulates an infinite periodic tiling, and in the field of partial differential equations is sometimes referred to as periodic boundary conditions.) This can be visualized as taping the left and right edges of the rectangle to form a tube, then taping the top and bottom edges of the tube to form a torus (doughnut shape). Universes of other dimensions are handled similarly. This is done in order to solve boundary problems with neighborhoods, but another advantage of this system is that it is easily programmable using modular arithmetic functions. For example, in a 1-dimensional cellular automaton like the examples below, the neighborhood of a cell $x_i^t$—where $i$ is the time step (vertical), and $t$ is the index (horizontal) in one generation—is $\{x_{i-1}^{t-1}, x_i^{t-1}, x_{i+1}^{t-1}\}$. There will obviously be problems when a neighbourhood on a left border references its upper left cell, which is not in the cellular space, as part of its neighborhood.

**History**

Stanisław Ulam, while working at the Los Alamos National Laboratory in the 1940s, studied the growth of crystals, using a simple lattice network as his model. At the same time, John von Neumann, Ulam's colleague at Los Alamos, was working on the problem of self-replicating systems. Von Neumann's initial design was founded upon the notion of one robot building another robot. This design is known as the kinematic model. As he developed this design, von Neumann came to realize the great difficulty of building a self-replicating robot, and of the great cost in providing the robot with a "sea of parts" from which to build its replicant. Ulam suggested that von Neumann develop his design around a mathematical abstraction, such as the one Ulam used to study crystal growth. Thus was born the first system of cellular automata. Like Ulam's lattice network, von Neumann's cellular automata are two-dimensional, with his self-replicator implemented algorithmically. The result was a universal copier and constructor working within a CA with a small neighborhood (only those cells that touch are neighbors; for von Neumann's cellular automata, only orthogonal cells), and with 29 states per cell. Von Neumann gave an existence proof that a particular pattern would make endless copies of itself within the given cellular universe. This design is known as the tessellation model, and is called a von Neumann universal constructor.

Also in the 1940s, Norbert Wiener and Arturo Rosenblueth developed a cellular automaton model of excitable media. Their specific motivation was the mathematical description of impulse conduction in cardiac systems. Their original work continues to be cited in modern research publications on cardiac arrhythmia and excitable systems.

In the 1960s, cellular automata were studied as a particular type of dynamical system and the connection with the mathematical field of symbolic dynamics was established for the first time. In 1969, Gustav A. Hedlund compiled
many results following this point of view\textsuperscript{[7]} in what is still considered as a seminal paper for the mathematical study of cellular automata. The most fundamental result is the characterization in the Curtis–Hedlund–Lyndon theorem of the set of global rules of cellular automata as the set of continuous endomorphisms of shift spaces.

In the 1970s a two-state, two-dimensional cellular automaton named Game of Life became very widely known, particularly among the early computing community. Invented by John Conway and popularized by Martin Gardner in a \textit{Scientific American} article\textsuperscript{[8]}, its rules are as follows: If a cell has 2 black neighbours, it stays the same. If it has 3 black neighbours, it becomes black. In all other situations it becomes white. Despite its simplicity, the system achieves an impressive diversity of behavior, fluctuating between apparent randomness and order. One of the most apparent features of the Game of Life is the frequent occurrence of \textit{gliders}, arrangements of cells that essentially move themselves across the grid. It is possible to arrange the automaton so that the gliders interact to perform computations, and after much effort it has been shown that the Game of Life can emulate a universal Turing machine\textsuperscript{[9]}. Possibly because it was viewed as a largely recreational topic, little follow-up work was done outside of investigating the particularities of the Game of Life and a few related rules.

In 1969, however, German computer pioneer Konrad Zuse published his book \textit{Calculating Space}, proposing that the physical laws of the universe are discrete by nature, and that the entire universe is the output of a deterministic computation on a giant cellular automaton. This was the first book on what today is called digital physics.

In 1983 Stephen Wolfram published the first of a series of papers systematically investigating a very basic but essentially unknown class of cellular automata, which he terms \textit{elementary cellular automata} (see below). The unexpected complexity of the behavior of these simple rules led Wolfram to suspect that complexity in nature may be due to similar mechanisms. Additionally, during this period Wolfram formulated the concepts of intrinsic randomness and computational irreducibility, and suggested that rule 110 may be universal—a fact proved later by Wolfram’s research assistant Matthew Cook in the 1990s.

In 2002 Wolfram published a 1280-page text \textit{A New Kind of Science}, which extensively argues that the discoveries about cellular automata are not isolated facts but are robust and have significance for all disciplines of science. Despite much confusion in the press and academia, the book did not argue for a fundamental theory of physics based on cellular automata, and although it did describe a few specific physical models based on cellular automata, it also provided models based on qualitatively different abstract systems.

**Elementary cellular automata**

The simplest nontrivial CA would be one-dimensional, with two possible states per cell, and a cell's neighbors defined to be the adjacent cells on either side of it. A cell and its two neighbors form a neighborhood of 3 cells, so there are $2^3=8$ possible patterns for a neighborhood. A rule consists of deciding, for each pattern, whether the cell will be a 1 or a 0 in the next generation. There are then $2^8=256$ possible rules. These 256 CAs are generally referred to by their Wolfram code, a standard naming convention invented by Stephen Wolfram which gives each rule a number from 0 to 255. A number of papers have analyzed and compared these 256 CAs. The rule 30 and rule 110 CAs are particularly interesting. The images below show the history of each when the starting configuration consists of a 1 (at the top of each image) surrounded by 0's. Each row of pixels represents a generation in the history of the automaton, with $t=0$ being the top row. Each pixel is colored white for 0 and black for 1.
Rule 30 cellular automaton

<table>
<thead>
<tr>
<th>current pattern</th>
<th>111</th>
<th>110</th>
<th>101</th>
<th>100</th>
<th>011</th>
<th>010</th>
<th>001</th>
<th>000</th>
</tr>
</thead>
<tbody>
<tr>
<td>new state for center cell</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Rule 110 cellular automaton

<table>
<thead>
<tr>
<th>current pattern</th>
<th>111</th>
<th>110</th>
<th>101</th>
<th>100</th>
<th>011</th>
<th>010</th>
<th>001</th>
<th>000</th>
</tr>
</thead>
<tbody>
<tr>
<td>new state for center cell</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Rule 30 exhibits *class 3* behavior, meaning even simple input patterns such as that shown lead to chaotic, seemingly random histories.

Rule 110, like the Game of Life, exhibits what Wolfram calls *class 4* behavior, which is neither completely random nor completely repetitive. Localized structures appear and interact in various complicated-looking ways. In the course of the development of *A New Kind of Science*, as a research assistant to Stephen Wolfram in 1994, Matthew Cook proved that some of these structures were rich enough to support universality. This result is interesting because rule 110 is an extremely simple one-dimensional system, and one which is difficult to engineer to perform specific behavior. This result therefore provides significant support for Wolfram's view that class 4 systems are inherently likely to be universal. Cook presented his proof at a Santa Fe Institute conference on Cellular Automata in 1998, but
Wolfram blocked the proof from being included in the conference proceedings, as Wolfram did not want the proof to be announced before the publication of *A New Kind of Science*. In 2004, Cook's proof was finally published in Wolfram's journal Complex Systems [10] (Vol. 15, No. 1), over ten years after Cook came up with it. Rule 110 has been the basis over which some of the smallest universal Turing machines have been built, inspired on the breakthrough concepts that the development of the proof of rule 110 universality produced.

Reversible

A cellular automaton is said to be reversible if for every current configuration of the cellular automaton there is exactly one past configuration (preimage). If one thinks of a cellular automaton as a function mapping configurations to configurations, reversibility implies that this function is bijective. If a cellular automaton is reversible, its time-reversed behavior can also be described as a cellular automaton; this fact is a consequence of the Curtis–Hedlund–Lyndon theorem, a topological characterization of cellular automata. [11][12] For cellular automata in which not every configuration has a preimage, the configurations without preimages are called Garden of Eden patterns.

For one dimensional cellular automata there are known algorithms for deciding whether a rule is reversible or irreversible. [13][14] However, for cellular automata of two or more dimensions reversibility is undecidable; that is, there is no algorithm that takes as input an automaton rule and is guaranteed to determine correctly whether the automaton is reversible. The proof by Jarkko Kari is related to the tiling problem by Wang tiles.[15]

Reversible CA are often used to simulate such physical phenomena as gas and fluid dynamics, since they obey the laws of thermodynamics. Such CA have rules specially constructed to be reversible. Such systems have been studied by Tommaso Toffoli, Norman Margolus and others. Several techniques can be used to explicitly construct reversible CA with known inverses. Two common ones are the second order cellular automaton and the block cellular automaton, both of which involve modifying the definition of a CA in some way. Although such automata do not strictly satisfy the definition given above, it can be shown that they can be emulated by conventional CAs with sufficiently large neighborhoods and numbers of states, and can therefore be considered a subset of conventional CA. Conversely, it has been shown that every reversible cellular automaton can be emulated by a block cellular automaton.[16]

Totalistic

A special class of CAs are totalistic CAs. The state of each cell in a totalistic CA is represented by a number (usually an integer value drawn from a finite set), and the value of a cell at time $t$ depends only on the sum of the values of the cells in its neighborhood (possibly including the cell itself) at time $t-1$. [17][18] If the state of the cell at time $t$ does depend on its own state at time $t-1$ then the CA is properly called outer totalistic. [18] Conway's Game of Life is an example of an outer totalistic CA with cell values 0 and 1; outer totalistic cellular automata with the same Moore neighborhood structure as Life are sometimes called life-like cellular automata. [19]
Classification

Stephen Wolfram, in *A New Kind of Science* and in several papers dating from the mid-1980s, defined four classes into which cellular automata and several other simple computational models can be divided depending on their behavior. While earlier studies in cellular automata tended to try to identify type of patterns for specific rules, Wolfram's classification was the first attempt to classify the rules themselves. In order of complexity the classes are:

- **Class 1**: Nearly all initial patterns evolve quickly into a stable, homogeneous state. Any randomness in the initial pattern disappears.
- **Class 2**: Nearly all initial patterns evolve quickly into stable or oscillating structures. Some of the randomness in the initial pattern may filter out, but some remains. Local changes to the initial pattern tend to remain local.
- **Class 3**: Nearly all initial patterns evolve in a pseudo-random or chaotic manner. Any stable structures that appear are quickly destroyed by the surrounding noise. Local changes to the initial pattern tend to spread indefinitely.
- **Class 4**: Nearly all initial patterns evolve into structures that interact in complex and interesting ways. Class 2 type stable or oscillating structures may be the eventual outcome, but the number of steps required to reach this state may be very large, even when the initial pattern is relatively simple. Local changes to the initial pattern may spread indefinitely. Wolfram has conjectured that many, if not all class 4 cellular automata are capable of universal computation. This has been proven for Rule 110 and Conway's game of Life.

These definitions are qualitative in nature and there is some room for interpretation. According to Wolfram, "...with almost any general classification scheme there are inevitably cases which get assigned to one class by one definition and another class by another definition. And so it is with cellular automata: there are occasionally rules...that show some features of one class and some of another."[20] Wolfram's classification has been empirically matched to a clustering of the compressed lengths of the outputs of cellular automata.[21]

There have been several attempts to classify CA in formally rigorous classes, inspired by the Wolfram's classification. For instance, Culik and Yu proposed three well-defined classes (and a fourth one for the automata not matching any of these), which are sometimes called Culik-Yu classes; membership in these proved to be undecidable.[22][23][24]
Evolving cellular automata using genetic algorithms

Recently there has been a keen interest in building decentralized systems, be they sensor networks or more sophisticated micro level structures designed at the network level and aimed at decentralized information processing. The idea of emergent computation came from the need of using distributed systems to do information processing at the global level. The area is still in its infancy, but some people have started taking the idea seriously. Melanie Mitchell who is Professor of Computer Science at Portland State University and also Santa Fe Institute External Professor has been working on the idea of using self-evolving cellular arrays to study emergent computation and distributed information processing. Mitchell and colleagues are using evolutionary computation to program cellular arrays. Computation in decentralized systems is very different from classical systems, where the information is processed at some central location depending on the system's state. In decentralized system, the information processing occurs in the form of global and local pattern dynamics.

The inspiration for this approach comes from complex natural systems like insect colonies, nervous system and economic systems. The focus of the research is to understand how computation occurs in an evolving decentralized system. In order to model some of the features of these systems and study how they give rise to emergent computation, Mitchell and collaborators at the SFI have applied Genetic Algorithms to evolve patterns in cellular automata. They have been able to show that the GA discovered rules that gave rise to sophisticated emergent computational strategies. Mitchell's group used a single dimensional binary array where each cell has six neighbors. The array can be thought of as a circle where the first and last cells are neighbors. The evolution of the array was tracked through the number of ones and zeros after each iteration. The results were plotted to show clearly how the network evolved and what sort of emergent computation was visible.

The results produced by Mitchell's group are interesting, in that a very simple array of cellular automata produced results showing coordination over global scale, fitting the idea of emergent computation. Future work in the area may include more sophisticated models using cellular automata of higher dimensions, which can be used to model complex natural systems.

Cryptography use

Rule 30 was originally suggested as a possible Block cipher for use in cryptography (See CA-1.1).

Cellular automata have been proposed for public key cryptography. The one way function is the evolution of a finite CA whose inverse is believed to be hard to find. Given the rule, anyone can easily calculate future states, but it appears to be very difficult to calculate previous states. However, the designer of the rule can create it in such a way as to be able to easily invert it. Therefore, it is apparently a trapdoor function, and can be used as a public-key cryptosystem. The security of such systems is not currently known.
Related automata

There are many possible generalizations of the CA concept.

One way is by using something other than a rectangular (cubic, etc.) grid. For example, if a plane is tiled with regular hexagons, those hexagons could be used as cells. In many cases the resulting cellular automata are equivalent to those with rectangular grids with specially designed neighborhoods and rules.

Also, rules can be probabilistic rather than deterministic. A probabilistic rule gives, for each pattern at time \( t \), the probabilities that the central cell will transition to each possible state at time \( t+1 \). Sometimes a simpler rule is used; for example: "The rule is the Game of Life, but on each time step there is a 0.001% probability that each cell will transition to the opposite color."

The neighborhood or rules could change over time or space. For example, initially the new state of a cell could be determined by the horizontally adjacent cells, but for the next generation the vertical cells would be used.

The grid can be finite, so that patterns can "fall off" the edge of the universe.

In CA, the new state of a cell is not affected by the new state of other cells. This could be changed so that, for instance, a 2 by 2 block of cells can be determined by itself and the cells adjacent to itself.

There are continuous automata. These are like totalistic CA, but instead of the rule and states being discrete (e.g. a table, using states \{0,1,2\}), continuous functions are used, and the states become continuous (usually values in \([0,1]\)). The state of a location is a finite number of real numbers. Certain CA can yield diffusion in liquid patterns in this way.

Continuous spatial automata have a continuum of locations. The state of a location is a finite number of real numbers. Time is also continuous, and the state evolves according to differential equations. One important example is reaction-diffusion textures, differential equations proposed by Alan Turing to explain how chemical reactions could create the stripes on zebras and spots on leopards. When these are approximated by CA, such CAs often yield similar patterns. MacLennan considers continuous spatial automata as a model of computation.

There are known examples of continuous spatial automata which exhibit propagating phenomena analogous to gliders in the Game of Life.

Biology

Some biological processes occur—or can be simulated—by cellular automata.

Patterns of some seashells, like the ones in Conus and Cymbiola genus, are generated by natural CA. The pigment cells reside in a narrow band along the shell's lip. Each cell secretes pigments according to the activating and inhibiting activity of its neighbour pigment cells, obeying a natural version of a mathematical rule. The cell band leaves the colored pattern on the shell as it grows slowly. For example, the widespread species Conus textile bears a pattern resembling Wolfram's rule 30 CA.
Plants regulate their intake and loss of gases via a CA mechanism. Each stoma on the leaf acts as a cell.\textsuperscript{[32]}

Moving wave patterns on the skin of cephalopods can be simulated with a two-state, two-dimensional cellular automata, each state corresponding to either an expanded or retracted chromatophore.\textsuperscript{[33]}

Threshold automata have been invented to simulate neurons, and complex behaviors such as recognition and learning can be simulated.

Fibroblasts bear similarities to cellular automata, as each fibroblast only interacts with its neighbors.\textsuperscript{[34]}

### Chemical types

The Belousov–Zhabotinsky reaction is a spatio-temporal chemical oscillator which can be simulated by means of a cellular automaton. In the 1950s A. M. Zhabotinsky (extending the work of B. P. Belousov) discovered that when a thin, homogenous layer of a mixture of malonic acid, acidified bromate, and a ceric salt were mixed together and left undisturbed, fascinating geometric patterns such as concentric circles and spirals propagate across the medium. In the "Computer Recreations" section of the August 1988 issue of \textit{Scientific American},\textsuperscript{[35]}
A. K. Dewdney discussed a cellular automaton\textsuperscript{[36]} which was developed by Martin Gerhardt and Heike Schuster of the University of Bielefeld (West Germany). This automaton produces wave patterns resembling those in the Belousov-Zhabotinsky reaction.

### Computer processors

CA processors are physical (not computer simulated) implementations of CA concepts, which can process information computationally. Processing elements are arranged in a regular grid of identical cells. The grid is usually a square tiling, or tessellation, of two or three dimensions; other tilings are possible, but not yet used. Cell states are determined only by interactions with adjacent neighbor cells. No means exists to communicate directly with cells farther away.

One such CA processor array configuration is the systolic array.

Cell interaction can be via electric charge, magnetism, vibration (phonons at quantum scales), or any other physically useful means. This can be done in several ways so no wires are needed between any elements.

This is very unlike processors used in most computers today, von Neumann designs, which are divided into sections with elements that can communicate with distant elements over wires.

### Error correction coding

CA have been applied to design error correction codes in the paper "Design of CAECC – Cellular Automata Based Error Correcting Code", by D. Roy Chowdhury, S. Basu, I. Sen Gupta, P. Pal Chaudhuri. The paper defines a new scheme of building SEC-DED codes using CA, and also reports a fast hardware decoder for the code.

### CA as models of the fundamental physical reality

As Andrew Ilachinski points out in his \textit{Cellular Automata}, many scholars have raised the question of whether the universe is a cellular automaton.\textsuperscript{[37]} Ilachinsky argues that the importance of this question may be better appreciated with a simple observation, which can be stated as follows. Consider the evolution of rule 110: if it were some kind of "alien physics", what would be a reasonable description of the observed patterns?\textsuperscript{[38]} If you didn't know how the images were generated, you might end up conjecturing about the movement of some particle-like objects (indeed, physicist James Crutchfield made a rigorous mathematical theory out of this idea proving the statistical emergence of "particles" from CA).\textsuperscript{[39]} Then, as the argument goes, one might wonder if \textit{our} world, which is currently well described by physics with particle-like objects, could be a CA at its most fundamental level.

While a complete theory along this line is still to be developed, entertaining and developing this hypothesis led scholars to interesting speculation and fruitful intuitions on how can we make sense of our world within a discrete
framework. Marvin Minsky, the AI pioneer, investigated how to understand particle interaction with a four-dimensional CA lattice; Konrad Zuse—the inventor of the first working computer, the Z3—developed an irregularly organized lattice to address the question of the information content of particles. More recently, Edward Fredkin exposed what he terms the "finite nature hypothesis", i.e., the idea that "ultimately every quantity of physics, including space and time, will turn out to be discrete and finite." Fredkin and Stephen Wolfram are strong proponents of a CA-based physics.

In recent years, other suggestions along these lines have emerged from literature in non-standard computation. Stephen Wolfram's A New Kind of Science considers CA to be the key to understanding a variety of subjects, physics included. The Mathematics Of the Models of Reference—created by iLabs founder Gabriele Rossi and developed with Francesco Berto and Jacopo Tagliabue—features an original 2D/3D universe based on a new "rhombic dodecahedron-based" lattice and a unique rule. This model satisfies universality (it is equivalent to a Turing Machine) and perfect reversibility (a desideratum if one wants to conserve various quantities easily and never lose information), and it comes embedded in a first-order theory, allowing computable, qualitative statements on the universe evolution.[44]

In popular culture

- One-dimensional cellular automata were mentioned in the Season 2 episode of NUMB3RS "Better or Worse".[45]
- The Hacker Emblem, a symbol for hacker culture proposed by Eric S. Raymond, depicts a glider from Conway's Game of Life.[46]
- The Autoverse, an artificial life simulator in the novel Permutation City, is a cellular automaton.[47]
- Cellular automata are discussed in the novel Bloom.[48]
- Cellular automata are central to Robert J. Sawyer's trilogy WWW in an attempt to explain how Websmind spontaneously attained consciousness.[49]

Reference notes

References

- "History of Cellular Automata" (http://www.wolframscience.com/reference/notes/876b) from Stephen Wolfram's A New Kind of Science
- Cellular automaton FAQ (http://cafaq.com/) from the newsgroup comp.theory.cell-automata
- Neighbourhood survey (http://cell-auto.com/neighbourhood/index.html) includes discussion on triangular grids, and larger neighbourhood CAs.
- Cosma Shalizi’s Cellular Automata Notebook (http://cscs.umich.edu/~crshalizi/notebooks/cellular-automata.html) contains an extensive list of academic and professional reference material.
- Wolfram's papers on CAs (http://www.stephenwolfram.com/publications/articles/ca/)
- The Evolution of Emergent Computation, James P. Crutchfield and Melanie Mitchell (SFI Technical Report 94-03-012)
- Ganguly, Sikdar, Deutsch and Chaudhuri “A Survey on Cellular Automata” (http://www.wepapers.com/Papers/16352/files/swf/15001To20000/16352.swf)

External links

- Cellular Automata modelling of landslides and avalanches (http://www.nhazca.it/?page_id=1331&lang=en)
- Mirek’s Cellebration (http://www.mirekw.com/ca/index.html) – Home to free MCell and MJCell cellular automata explorer software and rule libraries. The software supports a large number of 1D and 2D rules. The site provides both an extensive rules lexicon and many image galleries loaded with examples of rules. MCell is a Windows application, while MJCell is a Java applet. Source code is available.
- Modern Cellular Automata (http://www.collidoscope.com/modernca/) – Easy to use interactive exhibits of live color 2D cellular automata, powered by Java applet. Included are exhibits of traditional, reversible, hexagonal, multiple step, fractal generating, and pattern generating rules. Thousands of rules are provided for viewing. Free software is available.
• Self-replication loops in Cellular Space (http://necsi.edu/postdocs/sayama/sdsr/java/) – Java applet powered exhibits of self replication loops.
• A collection of over 10 different cellular automata applets (http://vlab.infotech.monash.edu.au/simulations/cellular-automata/) (in Monash University's Virtual Lab)
• Golly (http://www.sourceforge.net/projects/golly) supports von Neumann, Nobili, GOL, and a great many other systems of cellular automata. Developed by Tomas Rokicki and Andrew Trevorrow. This is the only simulator currently available which can demonstrate von Neumann type self-replication.
• Conway Life (http://www.conwaylife.com/)
• First replicating creature spawned in life simulator (http://www.newscientist.com/article/mg20627653.800-first-replicating-creature-spawned-in-life-simulator.html)
• The Mathematics of the Models of Reference (http://www.mmdr.it/provaEN.asp), featuring a general tutorial on CA, interactive applet, free code and resources on CA as model of fundamental physics

Artificial immune system

In computer science, Artificial immune systems (AIS) are a class of computationally intelligent systems inspired by the principles and processes of the vertebrate immune system. The algorithms typically exploit the immune system's characteristics of learning and memory to solve a problem.

Definition

The field of Artificial Immune Systems (AIS) is concerned with abstracting the structure and function of the immune system to computational systems, and investigating the application of these systems towards solving computational problems from mathematics, engineering, and information technology. AIS is a sub-field of Biologically-inspired computing, and Natural computation, with interests in Machine Learning and belonging to the broader field of Artificial Intelligence.

AIS are adaptive systems, inspired by theoretical immunology and observed immune functions, principles and models, which are applied to problem solving.\[1\]

AIS is distinct from computational immunology and theoretical biology that are concerned with simulating immunology using computational and mathematical models towards better understanding the immune system, although such models initiated the field of AIS and continue to provide a fertile ground for inspiration. Finally, the field of AIS is not concerned with the investigation of the immune system as a substrate computation, such as DNA computing.

History

AIS began in the mid 1980s with Farmer, Packard and Perelson's (1986) and Bersini and Varela's papers on immune networks (1990). However, it was only in the mid 90s that AIS became a subject area in its own right. Forrest et al. (on negative selection) and Kephart et al.\[2\] published their first papers on AIS in 1994, and Dasgupta conducted extensive studies on Negative Selection Algorithms. Hunt and Cooke started the works on Immune Network models in 1995; Timmis and Neal continued this work and made some improvements. De Castro & Von Zuben's and Nicosia & Cutello's work (on clonal selection) became notable in 2002. The first book on Artificial Immune Systems was edited by Dasgupta in 1999.

New ideas, such as danger theory and algorithms inspired by the innate immune system, are also now being explored. Although some doubt that they are yet offering anything over and above existing AIS algorithms, this is
hotly debated, and the debate is providing one the main driving forces for AIS development at the moment. Other recent developments involve the exploration of degeneracy in AIS models,\[3\][4] which is motivated by its hypothesized role in open ended learning and evolution.\[5][6]

Originally AIS set out to find efficient abstractions of processes found in the immune system but, more recently, it is becoming interested in modelling the biological processes and in applying immune algorithms to bioinformatics problems.

In 2008, Dasgupta and Nino \[7\] published a textbook on Immunological Computation which presents a compendium of up-to-date work related to immunity-based techniques and describes a wide variety of applications.

### Techniques

The common techniques are inspired by specific immunological theories that explain the function and behavior of the mammalian adaptive immune system.

- **Clonal Selection Algorithm**: A class of algorithms inspired by the clonal selection theory of acquired immunity that explains how B and T lymphocytes improve their response to antigens over time called affinity maturation. These algorithms focus on the Darwinian attributes of the theory where selection is inspired by the affinity of antigen-antibody interactions, reproduction is inspired by cell division, and variation is inspired by somatic hypermutation. Clonal selection algorithms are most commonly applied to optimization and pattern recognition domains, some of which resemble parallel hill climbing and the genetic algorithm without the recombination operator.\[8\]

- **Negative Selection Algorithm**: Inspired by the positive and negative selection processes that occur during the maturation of T cells in the thymus called T cell tolerance. Negative selection refers to the identification and deletion (apoptosis) of self-reacting cells, that is T cells that may select for and attack self tissues. This class of algorithms are typically used for classification and pattern recognition problem domains where the problem space is modeled in the complement of available knowledge. For example in the case of an anomaly detection domain the algorithm prepares a set of exemplar pattern detectors trained on normal (non-anomalous) patterns that model and detect unseen or anomalous patterns.\[9\]

- **Immune Network Algorithms**: Algorithms inspired by the idiotypic network theory proposed by Niels Kaj Jerne that describes the regulation of the immune system by anti-idiotypic antibodies (antibodies that select for other antibodies). This class of algorithms focus on the network graph structures involved where antibodies (or antibody producing cells) represent the nodes and the training algorithm involves growing or pruning edges between the nodes based on affinity (similarity in the problems representation space). Immune network algorithms have been used in clustering, data visualization, control, and optimization domains, and share properties with artificial neural networks.\[10\]

- **Dendritic Cell Algorithms**: The Dendritic Cell Algorithm (DCA) is an example of an immune inspired algorithm developed using a multi-scale approach. This algorithm is based on an abstract model of dendritic cells (DCs). The DCA is abstracted and implemented through a process of examining and modeling various aspects of DC function, from the molecular networks present within the cell to the behaviour exhibited by a population of cells as a whole. Within the DCA information is granulated at different layers, achieved through multi-scale processing.\[11\]
Notes


References

People

- Uwe Aickelin (http://aickelin.com)
- Leandro de Castro (http://www.dca.fee.unicamp.br/~lnunes/)
- Fernando José Von Zuben (http://www.dca.fee.unicamp.br/~vonzuben/)
- Dipankar Dasgupta (http://www.msci.memphis.edu/~dasgupta/)
- Jon Timmis (http://www-users.cs.york.ac.uk/jtimmis/)
- Giuseppe Nicosia (http://www.dmi.unict.it/nicosia/)
- Stephanie Forrest (http://www.cs.unm.edu/~forrest/)
- Pablo Dalbem de Castro (http://www.dca.fee.unicamp.br/~pablo)
- Julie Greensmith (http://www.cs.nott.ac.uk/~jqg/)

External links

- AISWeb: The Online Home of Artificial Immune Systems (http://www.artificial-immune-systems.org)
  Information about AIS in general and links to a variety of resources including ICARIS conference series, code, teaching material and algorithm descriptions.
- ARTIST: Network for Artificial Immune Systems (http://www.elec.york.ac.uk/ARTIST) Provides information about the UK AIS network, ARTIST. It provides technical and financial support for AIS in the UK and beyond, and aims to promote AIS projects.
- Computer Immune Systems (http://www.cs.unm.edu/~immsec/) Group at the University of New Mexico led by Stephanie Forrest.
- The ISYS Project (http://www.aber.ac.uk/~dcswww/ISYS) A now out of date project at the University of Wales, Aberystwyth interested in data analysis with AIS.
- AIS on Facebook (http://www.facebook.com/group.php?gid=12481710452) Group for people interested in the scientific field of artificial immune systems.
- The Center for Modeling Immunity to Enteric Pathogens (MIEP) (http://www.modelingimmunity.org)
Evolutionary multi-modal optimization

In applied mathematics, **multimodal optimization** deals with Optimization (mathematics) tasks that involve finding all or most of the multiple solutions (as opposed to a single best solution).

**Motivation**

Knowledge of multiple solutions to an optimization task is especially helpful in engineering, when due to physical (and/or cost) constraints, the best results may not always be realizable. In such a scenario, if multiple solutions (local and global) are known, the implementation can be quickly switched to another solution and still obtain a optimal system performance. Multiple solutions could also be analyzed to discover hidden properties (or relationships), which makes them high-performing. In addition, the algorithms for multimodal optimization usually not only locate multiple optima in a single run, but also preserve their population diversity, resulting in their global optimization ability on multimodal functions. Moreover, the techniques for multimodal optimization are usually borrowed as diversity maintenance techniques to other problems [1].

**Background**

Classical techniques of optimization would need multiple restart points and multiple runs in the hope that a different solution may be discovered every run, with no guarantee however. Evolutionary algorithms (EAs) due to their population based approach, provide a natural advantage over classical optimization techniques. They maintain a population of possible solutions, which are processed every generation, and if the multiple solutions can be preserved over all these generations, then at termination of the algorithm we will have multiple good solutions, rather than only the best solution. Note that, this is against the natural tendency of EAs, which will always converge to the best solution, or a sub-optimal solution (in a rugged, “badly behaving” function). Finding and Maintenance of multiple solutions is wherein lies the challenge of using EAs for multi-modal optimization. Niching [2] is a generic term referred to as the technique of finding and preserving multiple stable niches, or favorable parts of the solution space possibly around multiple solutions, so as to prevent convergence to a single solution.

The field of EAs today encompass Genetic Algorithms (GAs), Differential evolution (DE), Particle Swarm Optimization (PSO), Evolution strategy (ES) among others. Attempts have been made to solve multi-modal optimization in all these realms and most, if not all the various methods implement niching in some form or the other.

**Multimodal optimization using GAs**

Petrowski’s clearing method, Goldberg’s sharing function approach, restricted mating, maintaining multiple subpopulations are some of the popular approaches that have been proposed by the GA Community [3]. The first two methods are very well studied and respected in the GA community.

Recently, a Evolutionary Multiobjective optimization (EMO) approach was proposed [4], in which a suitable second objective is added to the originally single objective multimodal optimization problem, so that the multiple solutions form a weak pareto-optimal front. Hence, the multimodal optimization problem can be solved for its multiple solutions using a EMO algorithm. Improving upon their work [5], the same authors have made their algorithm self-adaptive, thus eliminating the need for pre-specifying the parameters.

An approach that does not use any radius for separating the population into subpopulations (or species) but employs the space topology instead is proposed in [6].
Multimodal optimization using DE

The niching methods used in GAs have also been explored with success in the DE community. DE based local selection and global selection approaches have also been attempted for solving multi-modal problems. DE’s coupled with local search algorithms (Memetic DE) have been explored as an approach to solve multi-modal problems.


References


Bibliography

• D. Goldberg and J. Richardson. (1987) "$\text{G}e\text{n}\text{e}\text{t}\text{i}\text{i}\text{c} \text{a}l\text{r}\text{g}\text{o}\text{r}\text{h}\text{i}t\text{s} \text{w}i\text{t}h\text{h} \text{sh}a\text{r}i\text{n}g\text{f}\text{o}\text{r} \text{m}u\text{lt}\text{i}\text{-}\text{m}o\text{d}a\text{l} \text{f}\text{u}\text{n}\text{c}\text{t}\text{i}\text{i}\text{o}\text{n} \text{o}\text{pt}\text{i}\text{m}\text{i}\text{z}\text{i}\text{a}\text{t}\text{i}\text{o}\text{n}$". In Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application table of contents, pages 41–49. L.Erlbaum Associates Inc. Hillsdale, NJ, USA, 1987.
• A. Petrowski. (1996) "$\text{A} \text{c}l\text{e}\text{a}\text{r}\text{i}\text{n}\text{g}\text{p}\text{r}\text{o}\text{c}\text{e}\text{d}\text{u}\text{r}\text{e}\text{s} \text{a} \text{s} \text{a} \text{n} \text{a} \text{m}\text{i}\text{n} \text{g} \text{m}\text{e}\text{t} \text{f}\text{o}\text{r} \text{g}\text{e}\text{n}\text{e}\text{t}\text{i}\text{c} \text{a}l\text{r}\text{g}\text{o}\text{r}\text{h}\text{i}t\text{s}\$". In Proceedings of the 1996 IEEE International Conference on Evolutionary Computation, pages 798–803. Citeseer, 1996.
• Deb,K., (2001) "$\text{M}u\text{lt}\text{i}\text{-}\text{o}\text{b}\text{j}\text{e}\text{c}\text{t}\text{i}\text{v}\text{e} \text{O}p\text{t}\text{i}\text{m}\text{i}\text{z}\text{i}\text{a}\text{t}\text{i}\text{o}\text{n} \text{u}\text{s}\text{i}\text{n}\text{g} \text{E}\text{v}\text{o}\text{l}\text{u}\text{t}\text{i}\text{o}\text{n}\text{a}\text{r}\text{i}\text{y} \text{A}\text{l}\text{g}\text{r}\text{i}\text{h}\text{m}\text{s}$", Wiley (Google Books) (http://books.google.com/books?id=0STnp4GSy2uQC&printsec=frontcover&dq=multi-objective+optimization&source=bl&ots=stCmpqNyNij0&sig=r00IY1DnjaRVU94DvotX-I5mVCI&hl=en&ei=fHnNS4Kl51MuLkAW8OGsS&sa=X&oi=book_result&ct=result&resnum=8&ved=0CD0Q6AewBwk#v=onepage&q&f=false)
• F. Streichert, G. Stein, H. Ulmer, and A. Zell. (2004) "$\text{A} \text{c}l\text{u}\text{s}\text{t}\text{e}\text{r}\text{i}\text{n}\text{g} \text{b}\text{a}\text{s}\text{e}\text{d} \text{n}\text{i}\text{c}\text{h}\text{i}\text{n} \text{E}\text{A} \text{f}\text{o}\text{r} \text{m}\text{u}\text{l}\text{t}\text{i}\text{-}\text{m}\text{o}\text{d}a\text{l} \text{s}\text{e}\text{a}\text{r}\text{c}\text{h}\text{s}\$". Lecture notes in computer science, pages 293–304, 2004.


• Wong, K.C., (2010). Effect of Spatial Locality on an Evolutionary Algorithm for Multimodal Optimization. EvoApplications (1) 2010: 481-490 (http://www.springerlink.com/content/jn23t1036778017/)


**External links**

• Multi-modal optimization using Particle Swarm Optimization (PSO) (http://tracer.uc3m.es/tws/pso/multimodal.html)

• Niching in Evolution Strategy (ES) (http://www.princeton.edu/~oshir/NichingES/index.htm)
Evolutionary music

Evolutionary music is the audio counterpart to Evolutionary art, whereby algorithmic music is created using an evolutionary algorithm. The process begins with a population of individuals which by some means or other produce audio (e.g. a piece, melody, or loop), which is either initialized randomly or based on human-generated music. Then through the repeated application of computational steps analogous to biological selection, recombination and mutation the aim is for the produced audio to become more musical. Evolutionary sound synthesis is a related technique for generating sounds or synthesizer instruments. Evolutionary music is typically generated using an interactive evolutionary algorithm where the fitness function is the user or audience, as it is difficult to capture the aesthetic qualities of music computationally. However, research into automated measures of musical quality is also active. Evolutionary computation techniques have also been applied to harmonization and accompaniment tasks. The most commonly used evolutionary computation techniques are genetic algorithms and genetic programming.

History

NEUROGEN (Gibson & Byrne, 1991 [1]) employed a genetic algorithm to produce and combine musical fragments and a neural network (trained on examples of "real" music) to evaluate their fitness. A genetic algorithm is also a key part of the improvisation and accompaniment system GenJam [10] which has been developed since 1993 by Al Biles. Al and GenJam are together known as the Al Biles Virtual Quintet and have performed many times to human audiences. Since 1996 Rodney Waschka II has been using genetic algorithms for music composition including works such as Saint Ambrose [2] and his string quartets. [3] In 1997 Brad Johanson and Riccardo Poli developed the GP-Music System [4] which, as the name implies, used genetic programming to breed melodies according to both human and automated ratings. Several systems for drum loop evolution have been produced (including one commercial program called MuSing [5]).

Conferences

The EvoMUSART Conference [6] from 2012 (previously a workshop) was part of the Evo* [7] event annually from 2003. This event on evolutionary music and art is one of the main outlets for work on evolutionary music.

A annual Workshop in Evolutionary Music [8] has been held at GECCO (Genetic and Evolutionary Computation Conference [9]) since 2011.

Recent work

The EuroGP Song Contest [10] (a pun on Eurovision Song Contest) was held at EuroGP 2004 [11]. In this experiment several tens of users were first tested for their ability to recognise musical differences, and then a short piano-based melody was evolved.


Evolutune [14] is a small Windows application from 2005 for evolving simple loops of "beeps and boops". It has a graphical interface where the user can select parents manually.

The GeneticDrummer [15] is a Genetic Algorithm based system for generating human-competitive rhythm accompaniment.

The easy Song Builder [16] is a evolutionary composition program. The user decides which version of the song will be the germ for the next generation.
Books

- *Creative Evolutionary Systems* by David W. Corne, Peter J. Bentley [19]

References

[3] SpringerLink - Book Chapter (http://www.springerlink.com/content/j1up38mn7205g552/?p=e54526113482447681a3114bed6f5eef&pi=5)

Links

- Al Biles' Evolutionary Music Bibliography (http://www.it.rit.edu/~jab/EvoMusic/EvoMusBib.html) - also includes pointers to work on evolutionary sound synthesis.
- Evolectronica (http://evolectronica.com) interactive evolving streaming electronic music
Coevolution

In biology, **coevolution** is "the change of a biological object triggered by the change of a related object."[1] Coevolution can occur at many biological levels: it can be as microscopic as correlated mutations between amino acids in a protein, or as macroscopic as covarying traits between different species in an environment. Each party in a coevolutionary relationship exerts selective pressures on the other, thereby affecting each other's evolution. Coevolution of different species includes the evolution of a host species and its parasites (host–parasite coevolution), and examples of mutualism evolving through time. Evolution in response to abiotic factors, such as climate change, is not coevolution (since climate is not alive and does not undergo biological evolution). Coevolution between pairs of entities exists, such as that between predator and prey, host and symbiont or host and parasite, but many cases are less clearcut: a species may evolve in response to a number of other species, each of which is also evolving in response to a set of species. This situation has been referred to as "diffuse coevolution."

There is little evidence of coevolution driving large-scale changes in Earth's history, since abiotic factors such as mass extinction and expansion into ecospace seem to guide the shifts in the abundance of major groups.[2] However, there is evidence for coevolution at the level of populations and species. For example, the concept of coevolution was briefly described by Charles Darwin in *On the Origin of Species*, and developed in detail in *Fertilisation of Orchids*.[3][4][5] It is likely that viruses and their hosts may have coevolved in various scenarios.[6]

Coevolution is primarily a biological concept, but has been applied by analogy to fields such as computer science and astronomy.

**Models**

One model of coevolution was Leigh Van Valen's Red Queen's Hypothesis, which states that "for an evolutionary system, continuing development is needed just in order to maintain its fitness relative to the systems it is co-evolving with."[7]

Emphasizing the importance of sexual conflict, Thierry Lodé described the role of *antagonist interactions* in evolution, giving rise to a concept of *antagonist coevolution*.[8]

Coevolution branching strategies for asexual population dynamics in limited resource environments have been modeled using the generalized Lotka–Volterra equations.[9]
Specific examples

Hummingbirds and ornithophilous flowers

Hummingbirds and ornithophilous (bird-pollinated) flowers have evolved a mutualistic relationship. The flowers have nectar suited to the birds' diet, their color suits the birds' vision and their shape fits that of the birds' bills. The blooming times of the flowers have also been found to coincide with hummingbirds' breeding seasons.

Flowers have converged to take advantage of similar birds.\[10\] Flowers compete for pollinators, and adaptations reduce unfavourable effects of this competition.\[10\] Bird-pollinated flowers usually have higher volumes of nectar and higher sugar production than those pollinated by insects.\[11\] This meets the birds' high energy requirements, which are the most important determinants of their flower choice.\[11\] Following their respective breeding seasons, several species of hummingbirds occur at the same locations in North America, and several hummingbird flowers bloom simultaneously in these habitats. These flowers seem to have converged to a common morphology and color.\[11\] Different lengths and curvatures of the corolla tubes can affect the efficiency of extraction in hummingbird species in relation to differences in bill morphology.\[11\] Tubular flowers force a bird to orient its bill in a particular way when probing the flower, especially when the bill and corolla are both curved; this also allows the plant to place pollen on a certain part of the bird's body.\[11\] This opens the door for a variety of morphological co-adaptations.

An important requirement for attraction is conspicuousness to birds, which reflects the properties of avian vision and habitat features.\[11\] Birds have their greatest spectral sensitivity and finest hue discrimination at the red end of the visual spectrum,\[11\] so red is particularly conspicuous to them. Hummingbirds may also be able to see ultraviolet "colors".\[11\] The prevalence of ultraviolet patterns and nectar guides in nectar-poor entomophilous (insect-pollinated) flowers warns the bird to avoid these flowers.\[11\]

Hummingbirds form the family Trochilidae, whose two subfamilies are the Phaethornithinae (hermits) and the Trochilinae. Each subfamily has evolved in conjunction with a particular set of flowers. Most Phaethornithinae species are associated with large monocotyledonous herbs, while the Trochilinae prefer dicotyledonous plant species.\[11\]

Angraecoid orchids and African moths

Angraecoid orchids and African moths coevolve because the moths are dependent on the flowers for nectar and the flowers are dependent on the moths to spread pollen so they can reproduce. Coevolution has led to deep flowers and moths with long probosci.

Old world swallowtail and fringed rue
An example of antagonistic coevolution is the old world swallowtail (*Papilio machaon*) caterpillar living on the fringed rue (*Ruta chalepensis*) plant. The rue produces etheric oils which repel plant-eating insects. The old world swallowtail caterpillar developed resistance to these poisonous substances, thus reducing competition with other plant-eating insects.

**Garter snake and rough-skinned newt**

Coevolution of predator and prey species is illustrated by the Rough-skinned newt (*Taricha granulosa*) and the common garter snake (*Thamnophis sirtalis*). The newts produce a potent neurotoxin that concentrates in their skin. Garter snakes have evolved resistance to this toxin through a series of genetic mutations, and prey upon the newts. The relationship between these animals has resulted in an evolutionary arms race that has driven toxin levels in the newt to extreme levels. This is an example of coevolution because differential survival caused each organism to change in response to changes in the other.

**California buckeye and pollinators**

When beehives are populated with bee species that have not coevolved with the California buckeye (*Aesculus californica*), sensitivity to aesculin, a neurotoxin present in its nectar, may be noticed; this sensitivity is only thought to be present in honeybees and other insects that did not coevolve with *A. californica*.\(^{12}\)

**Acacia ant and bullhorn acacia tree**

The acacia ant (*Pseudomyrmex ferruginea*) protects the bullhorn acacia (*Acacia cornigera*) from preying insects and from other plants competing for sunlight, and the tree provides nourishment and shelter for the ant and its larvae.\(^{13}\) Nevertheless, some ant species can exploit trees without reciprocating, and hence have been given various names such as 'cheaters', 'exploiters', 'robbers' and 'freeloaders'. Although cheater ants do important damage to the reproductive organs of trees, their net effect on host fitness is difficult to forecast and not necessarily negative.\(^{14}\)
Yucca Moth and the yucca plant

In this mutualistic symbiotic relationship, the yucca plant (*Yucca whipplei*) is pollinated exclusively by *Tegeticula maculata*, a species of yucca moth that in turn relies on the yucca for survival.*[15]* Yucca moths tend to visit the flowers of only one species of yucca plant. In the flowers, the moth eats the seeds of the plant, while at the same time gathering pollen on special mouth parts. The pollen is very sticky, and will easily remain on the mouth parts when the moth moves to the next flower. The yucca plant also provides a place for the moth to lay its eggs, deep within the flower where they are protected from any potential predators.*[16]* The adaptations that both species exhibit characterize coevolution because the species have evolved to become dependent on each other.

Coevolution outside biology

Coevolution is primarily a biological concept, but has been applied to other fields by analogy.

Technological coevolution

Computer software and hardware can be considered as two separate components but tied intrinsically by coevolution. Similarly, operating systems and computer applications, web browsers and web applications. All of these systems depend upon each other and advance step by step through a kind of evolutionary process. Changes in hardware, an operating system or web browser may introduce new features that are then incorporated into the corresponding applications running alongside.

Algorithms

Coevolutionary algorithms are a class of algorithms used for generating artificial life as well as for optimization, game learning and machine learning. Coevolutionary methods have been applied by Daniel Hillis, who coevolved sorting networks, and Karl Sims, who coevolved virtual creatures.

Cosmology and astronomy

In his book *The Self-organizing Universe*, Erich Jantsch attributed the entire evolution of the cosmos to coevolution. In astronomy, an emerging theory states that black holes and galaxies develop in an interdependent way analogous to biological coevolution.*[17]*
References


Further reading

- Dawkins, R. Unweaving the Rainbow.
Evolutionary art

Evolutionary art is created using a computer. The process starts by having a population of many randomly generated individual representations of artworks. Each representation is evaluated for its aesthetic value and given a fitness score. The individuals with the higher fitness scores have a higher chance of remaining in the population while individuals with lower fitness scores are more likely to be removed from the population. This is the evolutionary principle of *Survival of the fittest*. The survivors are randomly selected in pairs to mate with each other and have offspring. Each offspring will also be a representation of an artwork with some inherited properties from both of its parents. These offspring will then be added to the population and will also be evaluated and given a fitness score. This process of evaluation, selection and mating is repeated for many generations. Sometimes mutation is also applied to add new properties or change existing properties of a few randomly selected individuals. Over time the pressure from the fitness selection generally causes the evolution of more aesthetic combinations of properties that make up the representations of the artworks.

Evolutionary art is a branch of Generative art, which system is characterized by the use of evolutionary principles and natural selection as generative procedure. It distinguishes from BioArt by its medium dependency. If the latter adapts a similar project with carbon-based organisms, Evolutionary Art evolves silicon-based systems.

In common with natural selection and animal husbandry, the members of a population undergoing artificial evolution modify their form or behavior over many reproductive generations in response to a selective regime.

In interactive evolution the selective regime may be applied by the viewer explicitly by selecting individuals which are aesthetically pleasing. Alternatively a selection pressure can be generated implicitly, for example according to the length of time a viewer spends near a piece of evolving art.

Equally, evolution may be employed as a mechanism for generating a dynamic world of adaptive individuals, in which the selection pressure is imposed by the program, and the viewer plays no role in selection, as in the Black Shoals project.
Further reading


Conferences

- "Evomusart. 1st International Conference and 10th European Event on Evolutionary and Biologically Inspired Music, Sound, Art and Design"[^3]

External links

- "Evolutionary Art Gallery"[^4], by Thomas Fernandez
- "Biomorphs"[^5], by Richard Dawkins
- EndlessForms.com[^3], Collaborative interactive evolution allowing you to evolve 3D objects and have them 3D printed.
- "Musigenesis"[^6], a program that evolves music on a PC
- "Evolve"[^7], a program by Josh Lee that evolves art through a voting process.
- "Living Image Project"[^8], a site where images are evolved based on votes of visitors.
- "An evolutionary art program using Cartesian Genetic Programming"[^9]
- "Darwinian Poetry"[^12]
- "One mans eyes?"[^10], Aesthetically evolved images by Ashley Mills.
- "E-volver"[^8], interactive breeding units.
- "Breed"[^11], evolved sculptures produced by rapid manufacturing techniques.
- "ImageBreeder"[^12], an online breeder and gallery for users to submit evolved images.
- "Picbreeder"[^15], Collaborative breeder allowing branching from other users' creations that produces pictures like faces and spaceships.
- "CFDG Mutate"[^13], a tool for image evolution based on Chris Coyne's Context Free Design Grammar.
- "xTNZ"[^14], a three-dimensional ecosystem, where creatures evolve shapes and sounds.
- Evolvotron[^16] - Evolutionary art software (example output[^17]).
Artificial life (often abbreviated ALife or A-Life\textsuperscript{1}) is a field of study and an associated art form which examine systems related to life, its processes, and its evolution through simulations using computer models, robotics, and biochemistry.\textsuperscript{2} The discipline was named by Christopher Langton, an American computer scientist, in 1986.\textsuperscript{3} There are three main kinds of alife,\textsuperscript{4} named for their approaches: soft,\textsuperscript{5} from software; hard,\textsuperscript{6} from hardware; and wet, from biochemistry. Artificial life imitates traditional biology by trying to recreate biological phenomena.\textsuperscript{7} The term "artificial life" is often used to specifically refer to soft alife.\textsuperscript{8}

Overview

Artificial life studies the logic of living systems in artificial environments. The goal is to study the phenomena of living systems in order to come to an understanding of the complex information processing that defines such systems.

Also sometimes included in the umbrella term Artificial Life are agent based systems which are used to study the emergent properties of societies of agents.

While life is, by definition, alive, artificial life is generally referred to as being confined to a digital environment and existence.

Philosophy

The modeling philosophy of alife strongly differs from traditional modeling, by studying not only "life-as-we-know-it", but also "life-as-it-might-be".\textsuperscript{9}

In the first approach, a traditional model of a biological system will focus on capturing its most important parameters. In contrast, an alife modeling approach will generally seek to decipher the most simple and general
principles underlying life and implement them in a simulation. The simulation then offers the possibility to analyse new, different life-like systems.

Red’ko proposed to generalize this distinction not just to the modeling of life, but to any process. This led to the more general distinction of "processes-as-we-know-them" and "processes-as-they-could-be"[10].

At present, the commonly accepted definition of life does not consider any current alife simulations or softwares to be alive, and they do not constitute part of the evolutionary process of any ecosystem. However, different opinions about artificial life's potential have arisen:

- The strong alife (cf. Strong AI) position states that "life is a process which can be abstracted away from any particular medium" (John von Neumann). Notably, Tom Ray declared that his program Tierra is not simulating life in a computer but synthesizing it.
- The weak alife position denies the possibility of generating a "living process" outside of a chemical solution. Its researchers try instead to simulate life processes to understand the underlying mechanics of biological phenomena.

Software-based - "soft"

Techniques

- Cellular automata were used in the early days of artificial life, and they are still often used for ease of scalability and parallelization. Alife and cellular automata share a closely tied history.
- Neural networks are sometimes used to model the brain of an agent. Although traditionally more of an artificial intelligence technique, neural nets can be important for simulating population dynamics of organisms that can learn. The symbiosis between learning and evolution is central to theories about the development of instincts in organisms with higher neurological complexity, as in, for instance, the Baldwin effect.

Notable simulators

This is a list of artificial life/digital organism simulators, organized by the method of creature definition.

<table>
<thead>
<tr>
<th>Name</th>
<th>Driven By</th>
<th>Started</th>
<th>Ended</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aevol</td>
<td>translatable dna</td>
<td>2003</td>
<td>NA</td>
</tr>
<tr>
<td>Avida</td>
<td>executable dna</td>
<td>1993</td>
<td>NA</td>
</tr>
<tr>
<td>breve</td>
<td>executable dna</td>
<td>2006</td>
<td>NA</td>
</tr>
<tr>
<td>Creatures</td>
<td>neural net</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Darwinbots</td>
<td>executable dna</td>
<td>2003</td>
<td></td>
</tr>
<tr>
<td>DigiHive</td>
<td>executable dna</td>
<td>2006</td>
<td>2009</td>
</tr>
<tr>
<td>Evolve 4.0</td>
<td>executable dna</td>
<td>1996</td>
<td>2007</td>
</tr>
<tr>
<td>Framsticks</td>
<td>executable dna</td>
<td>1996</td>
<td>NA</td>
</tr>
<tr>
<td>Primordial life</td>
<td>executable dna</td>
<td>1996</td>
<td>2003</td>
</tr>
<tr>
<td>TechnoSphere</td>
<td>modules</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tierra</td>
<td>executable dna</td>
<td>early 1990s</td>
<td>?</td>
</tr>
<tr>
<td>Noble Ape</td>
<td>neural net</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Polyworld</td>
<td>neural net</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3D Virtual Creature Evolution</td>
<td>neural net</td>
<td></td>
<td>NA</td>
</tr>
</tbody>
</table>
Program-based

Further information: programming game

These contain organisms with a complex DNA language, usually Turing complete. This language is more often in the form of a computer program than actual biological DNA. Assembly derivatives are the most common languages used. Use of cellular automata is common but not required.

Module-based

Individual modules are added to a creature. These modules modify the creature's behaviors and characteristics either directly, by hard coding into the simulation (leg type A increases speed and metabolism), or indirectly, through the emergent interactions between a creature's modules (leg type A moves up and down with a frequency of X, which interacts with other legs to create motion). Generally these are simulators which emphasize user creation and accessibility over mutation and evolution.

Parameter-based

Organisms are generally constructed with pre-defined and fixed behaviors that are controlled by various parameters that mutate. That is, each organism contains a collection of numbers or other finite parameters. Each parameter controls one or several aspects of an organism in a well-defined way.

Neural net-based

These simulations have creatures that learn and grow using neural nets or a close derivative. Emphasis is often, although not always, more on learning than on natural selection.

Hardware-based - "hard"

Further information: Robot

Hardware-based artificial life mainly consist of robots, that is, automatically guided machines, able to do tasks on their own.

Biochemical-based - "wet"

Further information: Synthetic life and Synthetic biology

Biochemical-based life is studied in the field of synthetic biology. It involves e.g. the creation of synthetic DNA. The term "wet" is an extension of the term "wetware".

Related subjects

1. Artificial intelligence has traditionally used a top down approach, while alife generally works from the bottom up.\[^{11}\]
2. Artificial chemistry started as a method within the alife community to abstract the processes of chemical reactions.
3. Evolutionary algorithms are a practical application of the weak alife principle applied to optimization problems. Many optimization algorithms have been crafted which borrow from or closely mirror alife techniques. The primary difference lies in explicitly defining the fitness of an agent by its ability to solve a problem, instead of its ability to find food, reproduce, or avoid death. The following is a list of evolutionary algorithms closely related to and used in alife:
   - Ant colony optimization
   - Evolutionary algorithm
   - Genetic algorithm
• Genetic programming
• Swarm intelligence
4. Evolutionary art uses techniques and methods from artificial life to create new forms of art.
5. Evolutionary music uses similar techniques, but applied to music instead of visual art.
6. Abiogenesis and the origin of life sometimes employ alife methodologies as well.

Criticism
Alife has had a controversial history. John Maynard Smith criticized certain artificial life work in 1994 as "fact-free science". However, the recent publication of artificial life articles in widely read journals such as Science and Nature is evidence that artificial life techniques are becoming more accepted in the mainstream, at least as a method of studying evolution.

References
[1] Molecules and Thoughts Y Tarnopolsky - 2003 "Artificial Life (often abbreviated as Alife or A-life) is a small universe existing parallel to the much larger Artificial Intelligence. The origins of both areas were different.

External links
• Computers: Artificial Life (http://www.dmoz.org/Computers/Artificial_Life/) at the open directory project
• Computers: Artificial Life Framework (http://www.artificiallife.org/)
• International Society of Artificial Life (http://ailife.org/)
• Artificial Life (http://www.mitpressjournals.org/loi/artl) MIT Press Journal
• The Artificial Life Lab (http://www.envirtech.com/artificial-life-lab.html) Envirtech Island, Second Life
Machine learning

**Machine learning**, a branch of artificial intelligence, is a scientific discipline concerned with the design and development of algorithms that allow computers to evolve behaviors based on empirical data, such as from sensor data or databases. A learner can take advantage of examples (data) to capture characteristics of interest of their unknown underlying probability distribution. Data can be seen as examples that illustrate relations between observed variables. A major focus of machine learning research is to automatically learn to recognize complex patterns and make intelligent decisions based on data; the difficulty lies in the fact that the set of all possible behaviors given all possible inputs is too large to be covered by the set of observed examples (training data). Hence the learner must generalize from the given examples, so as to be able to produce a useful output in new cases.

**Definition**

Tom M. Mitchell provided a widely quoted definition: A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E. \[1\]

**Generalization**

Generalization is the ability of a machine learning algorithm to perform accurately on new, unseen examples after training on a finite data set. The core objective of a learner is to generalize from its experience. \[2\] The training examples from its experience come from some generally unknown probability distribution and the learner has to extract from them something more general, something about that distribution, that allows it to produce useful answers in new cases.

**Machine learning, knowledge discovery in databases (KDD) and data mining**

These three terms are commonly confused, as they often employ the same methods and overlap strongly. They can be roughly separated as follows:

- Machine learning focuses on the prediction, based on *known* properties learned from the training data
- Data mining (which is the analysis step of Knowledge Discovery in Databases) focuses on the discovery of (previously) *unknown* properties on the data

However, these two areas overlap in many ways: data mining uses many machine learning methods, but often with a slightly different goal in mind. On the other hand, machine learning also employs data mining methods as "unsupervised learning" or as a preprocessing step to improve learner accuracy. Much of the confusion between these two research communities (which do often have separate conferences and separate journals, ECML PKDD being a major exception) comes from the basic assumptions they work with: in machine learning, the performance is usually evaluated with respect to the ability to *reproduce known* knowledge, while in KDD the key task is the discovery of previously *unknown* knowledge. Evaluated with respect to known knowledge, an uninformed (unsupervised) method will easily be outperformed by supervised methods, while in a typical KDD task, supervised methods cannot be used due to the unavailability of training data.
**Human interaction**

Some machine learning systems attempt to eliminate the need for human intuition in data analysis, while others adopt a collaborative approach between human and machine. Human intuition cannot, however, be entirely eliminated, since the system's designer must specify how the data is to be represented and what mechanisms will be used to search for a characterization of the data.

**Algorithm types**

Machine learning algorithms can be organized into a taxonomy based on the desired outcome of the algorithm.

- **Supervised learning** generates a function that maps inputs to desired outputs (also called labels, because they are often provided by human experts labeling the training examples). For example, in a classification problem, the learner approximates a function mapping a vector into classes by looking at input-output examples of the function.
- **Unsupervised learning** models a set of inputs, like clustering. See also data mining and knowledge discovery.
- **Semi-supervised learning** combines both labeled and unlabeled examples to generate an appropriate function or classifier.
- **Reinforcement learning** learns how to act given an observation of the world. Every action has some impact in the environment, and the environment provides feedback in the form of rewards that guides the learning algorithm.
- **Transduction**, or transductive inference, tries to predict new outputs on specific and fixed (test) cases from observed, specific (training) cases.
- **Learning to learn** learns its own inductive bias based on previous experience.

**Theory**

The computational analysis of machine learning algorithms and their performance is a branch of theoretical computer science known as computational learning theory. Because training sets are finite and the future is uncertain, learning theory usually does not yield guarantees of the performance of algorithms. Instead, probabilistic bounds on the performance are quite common.

In addition to performance bounds, computational learning theorists study the time complexity and feasibility of learning. In computational learning theory, a computation is considered feasible if it can be done in polynomial time. There are two kinds of time complexity results. Positive results show that a certain class of functions can be learned in polynomial time. Negative results show that certain classes cannot be learned in polynomial time.

There are many similarities between machine learning theory and statistics, although they use different terms.

**Approaches**

**Decision tree learning**

Decision tree learning uses a decision tree as a predictive model which maps observations about an item to conclusions about the item's target value.

**Association rule learning**

Association rule learning is a method for discovering interesting relations between variables in large databases.

**Artificial neural networks**
An artificial neural network (ANN) learning algorithm, usually called "neural network" (NN), is a learning algorithm that is inspired by the structure and functional aspects of biological neural networks. Computations are structured in terms of an interconnected group of artificial neurons, processing information using a connectionist approach to computation. Modern neural networks are non-linear statistical data modeling tools. They are usually used to model complex relationships between inputs and outputs, to find patterns in data, or to capture the statistical structure in an unknown joint probability distribution between observed variables.

**Genetic programming**

Genetic programming (GP) is an evolutionary algorithm-based methodology inspired by biological evolution to find computer programs that perform a user-defined task. It is a specialization of genetic algorithms (GA) where each individual is a computer program. It is a machine learning technique used to optimize a population of computer programs according to a fitness landscape determined by a program's ability to perform a given computational task.

**Inductive logic programming**

Inductive logic programming (ILP) is an approach to rule learning using logic programming as a uniform representation for examples, background knowledge, and hypotheses. Given an encoding of the known background knowledge and a set of examples represented as a logical database of facts, an ILP system will derive a hypothesized logic program which entails all the positive and none of the negative examples.

**Support vector machines**

Support vector machines (SVMs) are a set of related supervised learning methods used for classification and regression. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that predicts whether a new example falls into one category or the other.

**Clustering**

Cluster analysis is the assignment of a set of observations into subsets (called clusters) so that observations in the same cluster are similar in some sense, while observations in different clusters are dissimilar. The variety of clustering techniques make different assumptions on the structure of the data, often defined by some similarity metric and evaluated for example by internal compactness (similarity between members of the same cluster) and separation between different clusters. Other methods are based on estimated density and graph connectivity. Clustering is a method of unsupervised learning, and a common technique for statistical data analysis.

**Bayesian networks**

A Bayesian network, belief network or directed acyclic graphical model is a probabilistic graphical model that represents a set of random variables and their conditional independencies via a directed acyclic graph (DAG). For example, a Bayesian network could represent the probabilistic relationships between diseases and symptoms. Given symptoms, the network can be used to compute the probabilities of the presence of various diseases. Efficient algorithms exist that perform inference and learning.
Reinforcement learning

Reinforcement learning is concerned with how an agent ought to take actions in an environment so as to maximize some notion of long-term reward. Reinforcement learning algorithms attempt to find a policy that maps states of the world to the actions the agent ought to take in those states. Reinforcement learning differs from the supervised learning problem in that correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected.

Representation learning

Several learning algorithms, mostly unsupervised learning algorithms, aim at discovering better representations of the inputs provided during training. Classical examples include principal components analysis and cluster analysis. Representation learning algorithms often attempt to preserve the information in their input but transform it in a way that makes it useful, often as a pre-processing step before performing classification or predictions, allowing to reconstruct the inputs coming from the unknown data generating distribution, while not being necessarily faithful for configurations that are implausible under that distribution. Manifold learning algorithms attempt to do so under the constraint that the learned representation is low-dimensional. Sparse coding algorithms attempt to do so under the constraint that the learned representation is sparse (has many zeros). Deep learning algorithms discover multiple levels of representation, or a hierarchy of features, with higher-level, more abstract features defined in terms of (or generating) lower-level features. It has been argued that an intelligent machine is one that learns a representation that disentangles the underlying factors of variation that explain the observed data.\cite{3}

Sparse Dictionary Learning

Sparse dictionary learning has been successfully used in a number of learning applications. In this method, a datum is represented as a linear combination of basis functions, and the coefficients are assumed to be sparse. Let $x$ be a $d$-dimensional datum, $D$ be a $d$ by $n$ matrix, where each column of $D$ represent a basis function. $r$ is the coefficient to represent $x$ using $D$. Mathematically, sparse dictionary learning means the following $x \approx D \times r$

where $r$ is sparse. Generally speaking, $n$ is assumed to be larger than $d$ to allow the freedom for a sparse representation.

Sparse dictionary learning has been applied in several contexts. In classification, the problem is to determine whether a new data belongs to which classes. Suppose we already build a dictionary for each class, then a new data is associate to the class such that it is best sparsely represented by the corresponding dictionary. People also applied sparse dictionary learning in image denoising. The key idea is that clean image path can be sparsely represented by a image dictionary, but the noise cannot. User can refer to \cite{4} if interested.

Applications

Applications for machine learning include:

- machine perception
- computer vision
- natural language processing
- syntactic pattern recognition
- search engines
- medical diagnosis
- bioinformatics
- brain-machine interfaces
- cheminformatics
- Detecting credit card fraud
- stock market analysis
• Classifying DNA sequences
• Sequence mining
• speech and handwriting recognition
• object recognition in computer vision
• game playing
• software engineering
• adaptive websites
• robot locomotion
• computational finance
• structural health monitoring.
• Sentiment Analysis (or Opinion Mining).
• Affective computing

In 2006, the on-line movie company Netflix held the first "Netfex Prize" competition to find a program to better predict user preferences and beat its existing Netflix movie recommendation system by at least 10%. The AT&T Research Team BellKor beat out several other teams with their machine learning program "Pragmatic Chaos". After winning several minor prizes, it won the grand prize competition in 2009 for $1 million. [5]

Software
RapidMiner, LIONsolver, KNIME, Weka, ODM, Shogun toolbox, Orange, Apache Mahout, scikit-learn, mlpy are software suites containing a variety of machine learning algorithms.

Journals and conferences
• Machine Learning (journal)
• Journal of Machine Learning Research
• Neural Computation (journal)
• Journal of Intelligent Systems (journal) [6]
• International Conference on Machine Learning (ICML) (conference)
• Neural Information Processing Systems (NIPS) (conference)

References
[5] "BelKor Home Page" (http://www2.research.att.com/~volinsky/netfex/) research.att.com
Further reading

• Toby Segaran, Programming Collective Intelligence, O'Reilly ISBN 0-596-52932-5
• Ian H. Witten and Eibe Frank Data Mining: Practical machine learning tools and techniques Morgan Kaufmann ISBN 0-12-088407-0.
• Mierswa, Ingo and Wurst, Michael and Klinkenberg, Ralf and Scholz, Martin and Euler, Timm: YALE: Rapid Prototyping for Complex Data Mining Tasks, in Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-06), 2006.
External links

- International Machine Learning Society (http://machinelearning.org/)
- There is a popular online course by Andrew Ng, at ml-class.org (http://www.ml-class.org). It uses GNU Octave. The course is a free version of Stanford University's actual course, whose lectures are also available for free (http://see.stanford.edu/see/courseinfo.aspx?coll=348ca38a-3a6d-4052-937d-cb017338d7b1).
- Machine Learning Video Lectures (http://videolectures.net/Top/Computer_Science/Machine_Learning/)

Evolvable hardware

Evolvable hardware (EH) is a new field about the use of evolutionary algorithms (EA) to create specialized electronics without manual engineering. It brings together reconfigurable hardware, artificial intelligence, fault tolerance and autonomous systems. Evolvable hardware refers to hardware that can change its architecture and behavior dynamically and autonomously by interacting with its environment.

Introduction

In its most fundamental form an evolutionary algorithm manipulates a population of individuals where each individual describes how to construct a candidate circuit. Each circuit is assigned a fitness, which indicates how well a candidate circuit satisfies the design specification. The evolutionary algorithm uses stochastic operators to evolve new circuit configurations from existing ones. Done properly, over time the evolutionary algorithm will evolve a circuit configuration that exhibits desirable behavior.

Each candidate circuit can either be simulated or physically implemented in a reconfigurable device. Typical reconfigurable devices are field-programmable gate arrays (for digital designs) or field-programmable analog arrays (for analog designs). At a lower level of abstraction are the field-programmable transistor arrays that can implement either digital or analog designs.

The concept was pioneered by Adrian Thompson at the University of Sussex, England, who in 1996 evolved a tone discriminator using fewer than 40 programmable logic gates and no clock signal in a FPGA. This is a remarkably small design for such a device and relied on exploiting peculiarities of the hardware that engineers normally avoid. For example, one group of gates has no logical connection to the rest of the circuit, yet is crucial to its function.

Why evolve circuits?

In many cases, conventional design methods (formulas, etc.) can be used to design a circuit. But in other cases, the design specification doesn't provide sufficient information to permit using conventional design methods. For example, the specification may only state desired behavior of the target hardware.

In other cases, an existing circuit must adapt—i.e., modify its configuration—to compensate for faults or perhaps a changing operational environment. For instance, deep-space probes may encounter sudden high radiation environments, which alter a circuit's performance; the circuit must self-adapt to restore as much of the original behavior as possible.
Finding the fitness of an evolved circuit

The fitness of an evolved circuit is a measure of how well the circuit matches the design specification. Fitness in evolvable hardware problems is determined via two methods:

• extrinsic evolution: all circuits are simulated to see how they perform
• intrinsic evolution: physical tests are run on actual hardware.

In extrinsic evolution only the final best solution in the final population of the evolutionary algorithm is physically implemented, whereas with intrinsic evolution every individual in every generation of the EA's population is physically realized and tested.

Future research directions

Evolvable hardware problems fall into two categories: original design and adaptive systems. Original design uses evolutionary algorithms to design a system that meets a predefined specification. Adaptive systems reconfigure an existing design to counteract faults or a changed operational environment.

Original design of digital systems is not of much interest because industry already can synthesize enormously complex circuitry. For example, one can buy IP to synthesize USB port circuitry, ethernet microcontrollers and even entire RISC processors. Some research into original design still yields useful results, for example genetic algorithms have been used to design logic systems with integrated fault detection that outperform hand designed equivalents. Original design of analog circuitry is still a wide-open research area. Indeed, the analog design industry is nowhere near as mature as is the digital design industry. Adaptive systems has been and remains an area of intense interest.

Literature


External links

• NASA-DoD-sponsored conference 2004 [1]
• NASA-DoD-sponsored conference 2005 [2]
• NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2006) [3]
• Adrian Thompson's Research Page [7]
• Adrian Thompson's paper on the Discriminator [8]
• Evolutionary Electronics at the University of Sussex [9]
NEAT Particles

NEAT Particles is an Interactive evolutionary computation program that enables users to evolve particle systems intended for use as special effects in video games or movie graphics. Rather than being hand-coded like typical particle systems, the behaviors of NEAT Particle effects are evolved by user preference. Therefore non-programmer, non-artist users may evolve complex and unique special effects in real time. NEAT Particles is meant to augment and assist the time-consuming computer graphics content generation process.

Method

In NEAT Particles, each particle system is controlled by a Compositional pattern-producing network (CPPN), a type of artificial neural network, or ANN. In other words, the usually hand-coded 'rules' of a particle system are replaced by automatically generated CPPNs. The CPPNs are evolved and complexified by NeuroEvolution of Augmenting Topologies (NEAT). A simple, interactive evolutionary computation (IEC) interface enables user guided evolution. In this manner increasingly complex particle system effects are evolved by user preference.

Benefit

The main benefit of NEAT Particles is to decouple particle system creation from programming, allowing unique and interesting effects to be quickly evolved by users without programming or artistic skill. Additionally, it provides a way for content developers to explore the range of possible effects. And finally, it can act as a concept art tool or idea generator, in which novel and useful effects are easily discovered.
Implications
The methodology of NEAT Particles can be applied to generation of other types of content, such as 3D models or programmable shader effects. The most significant implication of NEAT Particles and other Interactive evolutionary computation applications, is the possibility of automated content generation within a game itself, while it is played.

Bibliography


External links

- "Evolutionary Complexity Research Group at UCF" [2] - home of NEAT Particles and other evolutionary complexity research projects
- "NEAT Particles" [3] - latest source code and executable

References

Evolvable hardware [Source: http://en.wikipedia.org/w/index.php?oldid=484709551] Contributors: Crispin Cooper, Foobar, Hooperbloob, Lordvolton, Luzian, Mdd, Michael Hardy, MikeCombrink, Nabarry, Nicklott, Raji, RI, Sejomagno, Thekingofspain, Wbm1058, 32 anonymous edits

License

Creative Commons Attribution-Share Alike 3.0 Unported
//creativecommons.org/licenses/by-sa/3.0/
