Bio-Inspired Artificial Intelligence

Giovanni Iacca

a.y. 2024-25

Lorenzo Orsingher
LaTeX source

Contents

Chapter 1

Introduction to Bio-Inspired Artificial Intelligence

1.1 Introduction

What is Intelligence?

The definition of intelligence is not straightforward and its meaning is not entirely clear. Etymologically, "intelligence" comes from the Latin "intus legere", meaning "to read inside things". Therefore, intelligence can be seen as the capability to**understand**. However, there's no objective way to check for understanding, as it's subjective. Instead, we assess intelligence by observing behavior. Intuitively, we can differentiate between intelligent and non-intelligent behaviours.

What About Artificial (Machine) Intelligence (AI)?

A fundamental question is whether a machine can think. As with humans, we should look at the machine's behavior to assess its intelligence. A pragmatic definition of artificial intelligence is the ability to make a choice from a set of options to achieve a specific objective. For example, finding the shortest path or designing an optimal engineering object.

What is Artificial Intelligence?

A tentative definition of AI is: the ability to perform a choice (from a finite or infinite set of options) to achieve a certain objective.

1.2 The Classic Paradigm of AI

The classic paradigm of AI, from the 1950s to the 1980s, aimed to create intelligent machines. The goal was to replicate human cognition, such as reasoning, planning, and learning. However, this approach gradually moved away from biology, focusing instead on logic, optimal signal processing, data mining, and optimal control. This led to the neglect of the robustness of biological intelligence, such as autonomy, noise tolerance, embodiment, failure proofing, and adaptation.

1.3 The "Renaissance" of AI

Starting in the mid-1980s, AI experienced a "renaissance" where the objective was not just to mimic human capabilities but to surpass them. This led to the development of computational intelligence, bio-inspired algorithms, neural networks, and machine/deep learning.

1.4 Computational Intelligence

Computational intelligence is the study of adaptive mechanisms that enable intelligent behavior in complex and changing environments. It is also known as soft computing or natural computing. These

mechanisms exhibit the ability to learn, adapt to new situations, generalize, abstract, discover, and associate.

1.5 Taxonomy of AI Fields

The field of Artificial Intelligence encompasses machine learning. Machine learning includes neural networks, which lead to deep learning. Another overlapping area is computational intelligence and a subfield of this is**bio-inspired artificial intelligence**, which overlaps with biomimetics, bio-inspired engineering, bio-inspired robotics, and bio-inspired computing.

1.6 Examples of Biomimetics

Biomimetics involves creating devices and tools inspired by nature. Examples include:

- Velcro, inspired by pollen.
- Leonardo da Vinci's research on bird wings.
- Bio-robotics inspired by lizards, geckos, and insects.

These examples show that solutions found in nature can inspire artificial solutions.

1.7 Bio-Inspired Artificial Intelligence

The main focus of this course will be on the computational aspects of bio-inspired AI, rather than the engineering aspects. The main areas in bio-inspired AI are evolutionary computation and swarm intelligence. There are also other important paradigms:

- Neural Networks
- Cellular Automata
- Artificial Immune Systems
- Pattern Formation Techniques
- Epidemic Protocols
- Artificial Life (ALife) studies

Why Study Evolutionary Computation and Swarm Intelligence?

These paradigms have the ability not only to learn, but also to innovate and create novel solutions. They are applicable to an infinitely vast range of problems. These methods are robust, general-purpose and have not been fully exploited yet. Evolutionary Computation, and to some extent, Swarm Intelligence, may be the next "big thing" in AI, similar to the rediscovery of neural networks. For example, OpenAI published a paper in 2017 suggesting evolution strategies are a scalable alternative to reinforcement learning and in 2018, it was reported that evolutionary algorithms outperformed deep learning in video games.

1.8 Optimization

What is Optimization?

Optimization involves maximizing or minimizing a given objective function. A general optimization problem involves finding the optimum (x^*) of an objective function ($f(x)$), subject to constraints. Key elements:

- Decision variables: $x = [x_1, x_2, ..., x_n]$
- Objective function(s): $f(x)$

- Decision/search space: D
- Constraints: $g(x)$ and $h(x)$
- Global optimum: x^{**}

Note that minimizing $f(x)$ is the same as maximizing $-f(x)$. Optimization is a fundamental concept and is present in many problems.

Optimization is Everywhere

Optimization is applicable to diverse fields including:

- Car design
- Worker scheduling
- Industrial planning
- The Traveling Salesman Problem
- Robot trajectory optimization
- Portfolio optimization
- A/B testing
- Bug finding

Different Kinds of Optimization

Optimization problems can be classified into various types depending on the problem's characteristics:

- Continuous vs Combinatorial (Discrete) Optimization
- Linear vs Nonlinear Optimization
- Single vs Multi-Objective Optimization
- Constrained vs Unconstrained Optimization
- Stochastic/Dynamic vs Noiseless/Stationary Optimization

Classic Optimization Approaches

- Analytical Approach: The function has an explicit analytical expression that is derivable over all variables. Calculus is used to find optima.
- Exact Methods: The function respects specific hypotheses (e.g., linear or quadratic) and converges to the exact solution after a finite number of steps.
- Approximate Iterative Methods: The function respects some hypotheses but requires an infinite number of steps. These methods can provide approximations of the optimum after a finite number of steps.

Local Optimization

The goal of local optimization is to find a local optimum, starting from an initial guess.

- Gradient-based methods use the gradient or higher-level derivatives of the objective function. Examples include gradient descent, Newton methods, and quasi-Newton methods.
- Gradient-free methods do not use derivatives; they use a heuristic method. Examples include Rosenbrock, Nelder-Mead, and Hooke-Jeeves. Heuristics, from the Greek "heuriskō" meaning "I find", involve generating and testing solutions to find an approximate solution.

These methods are used when classic methods fail or are too slow.

Global Optimization

The goal of global optimization is to find the global optimum, which is a more complex task than local optimization.

- Deterministic approaches include brute force (discretizing the search space and evaluating all points).
- Stochastic approaches include random searches, starting from an initial point and perturbing it randomly.

Challenges in Optimization

Optimization problems can be easy to formulate but difficult to solve, especially in complex applications. This can be due to

- High nonlinearities
- High multimodality (many local optima)
- Noisy objective functions
- Approximated objective functions
- Computationally expensive problems
- Large-scale problems
- Limited hardware

These challenges make it necessary to use more robust problem-solving techniques.

1.9 Metaheuristics

What are Metaheuristics?

Metaheuristics, from the Greek "meta heuriskō", meaning "I find beyond", are algorithms that do not require specific assumptions about the objective function. They are often considered as a "black-box" optimization. They are especially useful when the analytical expression of the objective function is not available or is very complex, e.g. multivariate, noisy, non-differentiable, non-continuous, or nonlinear.

Computational Intelligence Optimization (CIO)

CIO is a subfield of Computational Intelligence that studies mathematical procedures to solve optimization problems, particularly where metaheuristics are the only option. Metaheuristics are stochastic optimization algorithms that are used as a last resort before using random or brute-force search. They are applied to problems where finding a good solution is difficult, but a grade can be given to any candidate solution.

A (Possible) Taxonomy of Metaheuristics

Metaheuristics can be categorized based on:

- Population vs trajectory-based
- Nature-inspired methods or not
- Dynamic vs static objective functions
- Memory-based vs memory-less algorithms
- Implicit, explicit, or direct metaheuristics

This course will focus on nature-inspired methods such as evolutionary algorithms and swarm intelligence techniques.

Examples of Metaheuristics

Examples include

- Simulated Annealing (SA)
- Evolutionary Algorithms (EAs) (including Genetic Algorithms (GAs), Evolutionary Programming (EP), Evolution Strategies (ES))
- Particle Swarm Optimization (PSO)
- Ant Colony Optimization (ACO)
- Bacterial Foraging Optimization (BFO)
- Differential Evolution (DE)
- Memetic Algorithms (MA)
- Hybrid Methods

Criticism of Metaheuristics

There has been criticism regarding the excessive use of metaphors in metaheuristics, where new algorithms are based on metaphors without significant novelty. Many algorithms have been inspired by various animals and natural phenomena, leading to many similar techniques. However, the evolutionary metaphor and swarm intelligence are very useful and have led to new families of algorithms.

No Free Lunch Theorems (NFLT)

NFLT states that for any pair of algorithms A and B, their performance over all possible problems is the same. This means that no single metaheuristic can solve all problems. It implies that every problem should be addressed with a proper algorithm that is tailored to the problem's features. There is no single "best" optimizer.

1.10 Natural Evolution

What is Evolution?

Biological systems result from an evolutionary process, where species originate from pre-existing types, with differences resulting from accumulated modifications over generations. The resulting biological systems are robust, complex, and adaptive.

The "Four Pillars" of Evolution

The four pillars are:

- Population: A group of several individuals
- Diversity: Individuals have different characteristics due to mutations
- Heredity: Characteristics are transmitted over generations
- Selection: Individuals produce more offspring than the environment can support, and fitter individuals reproduce more

All species derive from a common ancestor.

Evolution is NOT a Random Process

Randomness provides the material on which evolution acts. It is the result of two effects: variations (mostly random) and selection (mostly deterministic).

Evolution is NOT a Directional Process

Humans or any other species are not the top of the evolutionary ladder. Evolution proceeds without a predetermined direction or goal. If there is no competition, there is no selection of the fittest. There can be an accumulation of changes with no cost or benefit, known as neutral evolution.

Genotype vs. Phenotype

- Genotype: The genetic material of an organism, transmitted during reproduction, is affected by mutations and crossover, and selection does not operate directly on it.
- Phenotype: The manifestation of an organism (appearance, behavior, etc.), affected by the environment, development, and learning, and selection operates on it.

DNA (Deoxyribonucleic Acid)

DNA is a long, twisted molecule composed of two complementary sequences of four nucleotides (A, T, C, G). A gene is a sequence of nucleotides that codes for a protein. Humans have 23 pairs of DNA molecules (chromosomes).

Genes and Genome

The complete genetic material of an individual is called the genome. Within a species, most of the genetic material is the same. The genome contains genic DNA (codes for proteins) and non-genic DNA (still under study). Genome size is constant within a species but varies across species and it is not related to phenotype complexity.

From Genes to Proteins (Gene Expression)

Proteins define the type and function of cells. The sequence of nucleotides in DNA defines the type of protein. The expression of a gene into a protein is mediated by messenger RNA (mRNA). There is a one-way information flow from genotype to phenotype. Phenotypic traits (behavior/physical differences) affect responses to the environment, partly due to inheritance and partly due to development.

Genotype-Phenotype Mapping

The genotype-phenotype mapping is complex:

- One gene may affect many phenotypic traits (pleiotropy).
- Many genes may affect one phenotypic trait (polygeny).

Darwin vs. Lamarck

- Darwinian Evolution: If phenotypic traits lead to higher reproduction and can be inherited, they will increase in subsequent generations.
- Lamarckian Evolution: Acquired features can be inherited (partially incorrect, but has been revisited in light of phenotypic plasticity and epigenetics).

Genetic Mutations

Genetic mutations occur during cell replication and can be:

- Catastrophic: Offspring is not viable (most likely).
- Neutral: New feature does not influence fitness.
- Advantageous: New feature leads to higher fitness.

Redundancy in the genetic code helps with error checking. Recombination (crossover) is a type of mutation that affects two homologous chromosomes.

Adaptive Landscape Metaphor

A population with *n* phenotypic traits can be represented as existing in a *(n+1)* dimensional space, where the height corresponds to fitness. Each individual is a point in this landscape. A population can be seen as a "cloud" of points moving on the landscape as it evolves, with selection "pushing" it upwards.

Genetic Drift

Genetic drift is the random variation in feature distribution arising from sampling errors. It can cause the population to slide over hills and leave local optima.

1.11 Evolutionary Computation

What is Evolutionary Computation?

Evolutionary Computation is a set of techniques that copy the process of natural evolution, also known as Evolutionary Algorithms or Artificial Evolution algorithms.

Why Copying Natural Evolution?

- Nature has always been a source of inspiration for engineers and scientists.
- Evolution is the best problem solver known in nature, creating the human brain.

What Can Evolutionary Computation Be Used For?

- Understand "real" evolution
- Build evolutionary models (Artificial Life, or ALife)
- Solve optimization problems

Time for thorough problem analysis decreases, while complexity of the problem increases, therefore robust problem-solving technology is needed.

Similarities Between Natural and Artificial Evolution

- Individual: Encodes a potential solution to a problem, with a phenotype (computer program, object, etc.) and genotype (genetic representation of the phenotype).
- Population: A set of individuals.
- Diversity: A measure of how individuals differ in the population.
- Selection: A mechanism to select individuals that "survive" and "reproduce".
- Inheritance: A mechanism to transmit the properties of a solution.

Differences Between Natural and Artificial Evolution

- Fitness: In artificial evolution, fitness is a measure of how good a solution is, while in natural evolution, it measures reproductive success.
- Selection: In artificial evolution, selection is based on fitness, while in nature it is based on competition and interactions with the environment.
- Generations are typically non-overlapping in artificial evolution, while they overlap in natural evolution.
- Artificial evolution has expected improvement between the initial and final solutions, whereas natural evolution is not explicitly an optimization process.

However, when variations accumulate in a specific direction, natural evolution resembles an optimization process.

A Bit of History

- 1948: Turing mentions "genetical or evolutionary search".
- 1954: Barricelli publishes his paper "Esempi Numerici di processi di evoluzione".
- 1960s: Real development of the field starts.

One Framework, Different Paradigms

- Evolutionary Programming (EP)
- Evolution Strategies (ES)
- Genetic Algorithm (GA)
- Steady-State Evolution
- Cultural evolution, or Memetic Algorithms (MA)
- Genetic Programming (GP)
- Differential Evolution (DE)
- Estimation of Distribution Algorithm (EDA)
- Island Models
- Covariance Matrix Adaptation Evolution Strategy (CMA-ES)
- Non-Dominated Sorting Genetic Algorithm: (NSGA-2)

Evolutionary Algorithm

An evolutionary algorithm is a stochastic population-based metaheuristic that uses mechanisms inspired by biological evolution:

- Reproduction
- Inheritance
- Mutation
- Selection

Evolutionary algorithms require many simulations and inexpensive computer technology to run them.

The Key Elements: Individual and Fitness

An individual encodes a potential solution (e.g., a list of numbers, cities, bit strings). Each individual has a fitness, measuring how good the solution is.

Why Evolutionary Algorithms Work

Classic optimization algorithms (local search methods) work on a single solution, perturbed by some logic. They are good at exploitation but may miss the global optimum and are not parallelizable. Evolutionary algorithms, on the other hand, are population-based and inherently parallelizable. They are good at both exploitation and exploration and each solution can be perturbed differently, allowing for different search perspectives. Interactions among solutions can be beneficial.

Applicability

Evolutionary algorithms are used in a huge number of problems. Biological inspiration is essential, but often distorted. Different problems require different algorithms, and knowledge of the problem domain can help choose or assemble the best algorithm. While EAs are general-purpose, their performance depends heavily on the problem, and different EAs may perform differently on the same problem.

Some Applications

- Parameters optimization
- Finance/portfolio optimization
- Model and neural network training
- Forecast
- Scheduling
- Telecom/Networking
- CAD/CAE problems
- Database/Data mining
- Bioinformatics
- Bug identification
- Art
- ALife studies

CAD/CAE problems are computationally intensive with unknown fitness landscapes, vast infeasible regions, nonlinear constraints, and practical limitations.

1.12 Swarm Intelligence

Imagine a Treasure Hunt Game

Swarm intelligence can be understood through a treasure hunt analogy: a group of friends searching for treasure, using metal detectors, communicating with neighbors, and sharing rewards.

What is Swarm Intelligence?

Swarm intelligence is inspired by the collective behavior of social animals, where they perform tasks as a group that they cannot accomplish alone. A swarm is a group of simple agents that communicate, directly or indirectly, by changing their state or acting on their environment. Swarm intelligence is the property where local interactions of simple agents cause coherent global patterns to emerge. Main principles are that agents act on local information and cooperate using local information. Information propagates through the swarm, leading to distributed collective problem-solving.

Emergent Behavior (Without Leader)

Examples in nature include:

- Termites building large nests
- Ants dynamically allocating tasks and foraging via trail-following
- Bees communicating with the "waggle dance" for optimal foraging
- Birds in a flock and fishes in a school self-organizing
- Lions exhibiting hunting strategies
- Bacteria communicating using molecules
- Slime molds aggregating to form a mobile slug

Emergent Behavior (With Leader)

In some cases, there is a leader and more restrictive rules on relative motion, but individuals still use local information to decide how to move. Examples include herding, V formation, and processions.

Emergent Behavior (What About Humans?)

The "wisdom of the crowd" is an example of emergent behavior, as studied by J. Surowiecki.

1.13 Computational Swarm Intelligence

What is (Computational) Swarm Intelligence?

Computational swarm intelligence refers to algorithmic models of swarm intelligence behavior in nature and groups of agents.

Main Principles

- Unity is strength: The swarm performs tasks that a single individual cannot.
- Multiplicity/resilience: Swarms are composed of many individuals, which means it is resilient to individual loss or mistakes.
- Locality/simplicity: Individuals use only local information and perform simple actions, with complexity emerging from interactions.

Challenges

- Finding individual behavioral rules that result in desired swarm behavior.
- Ensuring the emergent behavior is stable.

Computational Swarm Intelligence Models

- Virtual models: Usually optimization algorithms, like Particle Swarm Optimization (PSO) and Ant Colony Optimization (ACO), also used for multi-agent simulations.
- Embodied models: Implemented in hardware, such as robots or network nodes.

Main features include population-based, where each individual changes its state according to strategies that use information from neighboring individuals.

Applications

- Optimization
- Multi-agent or crowd simulations
- Coordinated robots/drones
- Distributed control systems
- Distribution systems
- Telecommunication networks
- Mobile and pervasive computing
- Social networks

BOIDS

BOIDS (Bird-oid objects) is an example of computational swarm intelligence that simulates flocking behavior. Each boid perceives the angle and distance of its neighbors, and follows three simple rules:

1. Separation: Maintain a given distance from other boids
2. Cohesion: Move towards the center of mass of neighboring boids
3. Alignment: Align its angle with those of neighboring boids

Particle Swarm Optimization (PSO)

PSO mimics a group of birds searching for food, where each place in the environment has an associated reward. Each particle remembers its most successful place visited and gets information from its neighbors. In standard PSO, a swarm of particles is initialized with random positions and velocities. At each step, particles update their velocities (v') using the formula:

$$v' = \omega v + \phi_1 U_1 (y - x) + \phi_2 U_2 (z - x)$$

where:

- x and v are the particle's current position and velocity, respectively
- z and y are the neighborhood and focal particle's best position, respectively
- ω is the inertia (weighs the current velocity)
- ϕ_1 is the learning rate for the personal influence
- ϕ_2 is the learning rate for the social influence
- U_1 and U_2 are uniform random numbers in

Then, each particle updates its position: $x' = x + v'$. The particle's best position (y) and neighborhood's best position (z) are updated if there is improvement, and this process is repeated until a given stopping condition is met.

Ant Colony Optimization (ACO)

ACO mimics the navigation strategy used by foraging ants. Initially, ants move at random (exploration), then deposit pheromones to reinforce good paths. Pheromones along a trail evaporate if not followed, and ants communicate indirectly with each other via the environment (stigmergy). ACO is typically applied to the Traveling Salesman Problem (TSP) and other combinatorial problems. ACO can find the best solution on small problems and good solutions on large ones. It can also operate on dynamic problems that require fast rerouting.

Other Paradigms/Metaphors

Other paradigms inspired by nature include:

- Harmony Search
- Artificial Bee Colony Algorithm
- Bees Algorithm
- Glowworm Swarm Optimization
- Shuffled Frog Leaping Algorithm
- Imperialist Competitive Algorithm
- River Formation Dynamics
- Intelligent Water Drops Algorithm

- Gravitational Search Algorithm
- Cuckoo Search
- Bat Algorithm
- Spiral Optimization (SPO) Algorithm
- Flower Pollination Algorithm
- Cuttlefish Optimization Algorithm
- Duelist Algorithm
- Killer Whale Algorithm
- Rain Water Algorithm
- Mass and Energy Balances Algorithm

It is important to remember the criticism of metaphors in metaheuristics.

Chapter 2

Evolutionary Algorithms I

Evolutionary algorithms are a key area in bio-inspired artificial intelligence. They are a type of evolutionary computation and are implemented using a specific set of steps. Genetic algorithms are the first evolutionary algorithms to have been designed. These algorithms are applicable to any problem domain, provided a suitable genetic representation, fitness function, and genetic operators are chosen.

2.1 Genetic Algorithms: A Checklist

When implementing a genetic algorithm, several key steps must be followed:

1. **Devise a Genetic Representation**: A method to translate a problem's solution into a numerical representation, mapping phenotypes to genotypes and vice-versa. This is typically problem-specific and must match the chosen genetic operators.
    - The set of possible genotypes should include the optimal solution to the problem.
    - Choice of representation benefits from domain knowledge including the encoding of relevant parameters and appropriate accuracy.
    - It must be complete, representing every feasible solution in the genotype space.
    - A great simplification of genetics including single-stranded sequence of symbols, often fixed length, and often one chromosome with a haploid structure.
2. **Build an Initial Population**: Create a set of candidate solutions, often through random initialisation. The population size should be large enough to cover the search space, but small enough to be computationally feasible. Typical sizes range from tens to thousands of individuals.
    - For binary strings, 0 or 1 are assigned with equal probability.
    - For real-valued representations, a uniform distribution is used within given bounds, if bounded. Alternatively, a "best guess" based on domain knowledge or design of experiments can be used.
    - For trees, they are built recursively from a root chosen from the function set. Each branch is then randomly chosen from either the function or terminal set.
3. **Design a Fitness Function**: This evaluates the performance of a phenotype with numerical scores.
    - It involves choosing components and combining these in a way that can discriminate between better and worse solutions.
    - The fitness function can be objective, subjective through human inspection (Interactive Evolutionary Computation), or a combination of both.
    - Extensive testing of each phenotype is often required, and repeated evaluations may be needed.
    - The fitness function should be able to discriminate (return different values) so that the evolutionary process can be guided towards the optimum.
    - It is important to remember: "You Get What You Evaluate".

4. **Choose a Selection Method**: Determine which solutions should be selected for reproduction, biasing towards better individuals. The selection pressure is inversely proportional to the number of selected individuals.
    - It is important to ensure that less well performing individuals can also reproduce to some extent.
5. **Choose a Replacement Method**: Determine how new offspring replace members of the current population.
6. **Choose Crossover and Mutation**: Select genetic operators for creating new solutions from existing ones. Crossover combines genetic material from two parents, while mutation introduces variations to a single parent.
7. **Choose a Data Analysis Method**: Decide how to monitor the progress of the algorithm, tracking fitness, diversity, and convergence.

The generation cycle is repeated until a termination criterion is met, such as:

- Maximum fitness value is found.
- A solution found is "good enough".
- Maximum wall-clock time is reached.
- A certain convergence condition is met.

2.2 Genetic Representation

Genetic representation describes the elements of the genotype and its mapping to the phenotype. It is crucial for the success of a genetic algorithm.

Discrete Representations

The genotype is a sequence of discrete symbols from an alphabet of cardinality k.

- Binary strings are a common example, where $k = 2$.
- A binary string can be mapped to different phenotypes including integer values, real values in a given range, or binary assignments for problems like job scheduling.
- The conversion from real values to a bit string has a maximum accuracy of $\frac{max-min}{2^n-1}$.
- More bits increases accuracy but slows evolution.
- Binary coding can introduce "Hamming cliffs" where numerically adjacent values have very different bit representations.
- Gray coding can be used to minimize Hamming distance between representations of successive numerical values. Given a bitstring $B = [b_1 b_2 \dots b_n]$, the corresponding Gray bitstring $G = [g_1 g_2 \dots g_n]$ can be obtained as follows:
    - $g_1 = b_1$
    - $g_k = (b_{k-1}$ AND (NOT $b_1))$ OR $((\text{NOT } b_{k-1})$ AND $b_1)$

Sequence Representations

A particular case of discrete representation used for problems like the Traveling Salesman Problem (TSP). The individual is a permutation of n different symbols, each occurring exactly once. For example, planning ski holidays with the lowest transportation costs.

Real-Valued Representations

The genotype is a sequence of real values that directly represent problem parameters. Used when high-precision parameter optimization is required, such as the genetic encoding of a wing profile using pressure values.

Tree Representations

The genotype is a tree with branching points and terminals. Suitable for encoding hierarchical structures like computer programs (Genetic Programming, GP).

- Trees are made of operators (function set: multiplication, If-Then, Log, etc.) and operands (terminal set: constants, variables, sensor readings, etc.).
- Requirements for GP include Closure: functions accept all terminals and function outputs, and Sufficiency: function and terminal sets can generate the solution.

2.3 Initial Population

The initial population should be large enough to cover the search space but small enough in terms of evaluation. A uniform sample of search space is important.

- Binary strings: 0 or 1 with probability 0.5.
- Real-valued representations: uniform on a given interval.
- Trees: built recursively starting from the root, selecting from the function set or terminal set at random.

Hand-designed genotypes may cause a loss of genetic diversity or bias the evolutionary process.

2.4 Fitness Function

The fitness function evaluates the performance of a phenotype with one or more numerical scores.

- It involves the choice and combination of components, with extensive testing of each phenotype.
- The fitness function should be able to discriminate between better and worse solutions.
- Subjective fitness, based on human inspection, can be used when objective quantification is not possible. This is the basis of Interactive Evolution Computation.
- A key concept to remember is "You Get What You Evaluate".

2.5 Parent Selection

Parent selection ensures that better individuals produce more offspring. Selection pressure is inversely proportional to the number of selected individuals.

Random Selection

Each individual has a probability of $\frac{1}{N}$ to be selected, where N is the population size. This method does not use fitness information and is rarely used in EAs. It has the lowest selection pressure.

Fitness-Proportionate Selection (Roulette-Wheel Selection)

The probability that an individual reproduces is proportional to its fitness relative to the population's fitness: $p(i) = \frac{f(i)}{\sum f(i)}$. It biases selection towards the most fit individuals.

- Fitness values must be non-negative.
- Uniform fitness values result in random selection.
- Few high-fitness individuals can create excessively high selection pressure.

Rank-Based Selection

Individuals are sorted by fitness from best to worst. The rank is used to select individuals. The probability of selection is based on rank, offering lower selection pressure than fitness-proportionate selection: $p(i) = 1 - \frac{r(i)}{\sum r(i)}$.

Truncated Rank-Based Selection

Only the best x individuals can reproduce, each making the same number of offspring: $\frac{N}{x}$.

Tournament Selection

For every offspring, k individuals are randomly picked from the population, where k is the tournament size. The individual with the highest fitness is selected. Larger k results in larger selection pressure.

"Hall of Fame" Selection

The best individual from each generation is added to a Hall of Fame. This archive can be used as a parent pool for crossover. This is used in co-evolutionary algorithms.

(μ,λ) or (μ+λ) Selection

Deterministic, rank-based methods mainly used in Evolution Strategies. μ indicates the number of parents, and λ indicates the number of offspring produced.

- (μ, λ) selection selects the best μ individuals from λ offspring.
- $(\mu + \lambda)$ selection selects the best μ individuals from both the μ parents and the λ offspring. This is considered an elitist strategy.

2.6 Replacement (or "Survivor Selection")

This determines how the new offspring replace members of the current population.

Generational Replacement

The old population is entirely replaced by the offspring. This is the most frequent method. Also called "aged-based" or "non-overlapping" replacement.

Generational Rollover

Offspring are inserted "in place," replacing the worst individuals in the current generation.

- A special case is the Steady-State scheme, where one offspring is generated per generation and replaces the worst member of the population.

Elitism

Maintains the n best individuals from the previous generation to prevent the loss of the best individuals due to mutations or sub-optimal fitness evaluation. Higher n means less diversity.

2.7 Recombination (Crossover)

Emulates the recombination of genetic material from two parents during meiosis. It exploits the synergy of sub-solutions from parents. Applied to randomly paired offspring with a given probability P_c. Common crossover types include:

- One-point crossover
- Uniform crossover

- Arithmetic crossover

Crossover techniques also exist for sequences and trees.

2.8 Mutation

Emulates genetic mutations, exploring the variation of existing solutions. Applied to each gene in the genotype with a probability P_m.

- For binary genotypes, the common method is a bit flip.
- For real-valued genotypes, a delta value is added or subtracted to the existing value.
- Mutation operators also exist for sequence and tree genotypes.

2.9 Crossover or Mutation?

Both are generally good to have. Mutation provides exploration while crossover provides exploitation capabilities. Mutation-only EAs are possible, but crossover-only EAs would not work because they cannot introduce new genetic material into the population.

2.10 Data Analysis: Assessing Fitness Landscape

The fitness landscape is a plot of fitness values associated with all genotypes. The real landscape is unknown, but estimation helps to assess evolvability. The goal of evolution is to find the genotype with the best fitness. Estimating "ruggedness" of a fitness landscape is important.

1. Sample random genotypes: if flat, use large populations.
2. Explore surroundings of individuals by applying genetic operators in sequence: larger fitness improvement suggests it's "easier" to evolve.

2.11 Data Analysis: Monitoring Performance

Track best/worst and/or population average fitness (+/- standard deviation) of each generation. Multiple runs are necessary, and average data and standard error are plotted. Fitness graphs are meaningful if the problem is stationary. These plots can be used to detect if the algorithm has stagnated or prematurely converged.

- Stagnation: no further evolution possible even if the population remains diverse.
- Premature convergence: no further evolution possible because the population lost diversity.

Typical Behavior of an EA

Evolutionary runs typically have three phases:

- Early Phase: quasi-random population distribution.
- Mid-Phase: population arranged around/on hills.
- Late Phase: population concentrated on high hills.

Data Analysis: Measuring Diversity

Diversity indicates whether the population has potential for further evolution. Measures of diversity depend on the genetic representation, for example, the sum of Hamming or Euclidean distances for discrete and real values, respectively.

2.12 Some Extra Considerations

- Is it worth putting effort on smart initialisation? Yes, if good solutions/methods exist, but this can lead to a loss of diversity and/or bias.
- Are "long" runs beneficial? It depends on how much progress is desired. Multiple shorter runs might be better.

2.13 Generalities

The original GA is now known as the Simple Genetic Algorithm (SGA) (Holland, 1975). Main features include binary encoding (used for discrete optimization but also for real-valued problems) and fitness-proportionate selection, uniform mutation, and one-point crossover. The SGA is generally considered inefficient, especially for continuous optimisation, and should not be applied unless the problem is naturally binary. Many variants of GAs exist today with different mutation/crossover operators and selection models.

2.14 Alternative Types of Genetic Algorithms

Genetic algorithms can use either both recombination and mutation or use them as mutually exclusive operations.

2.15 Example: One-Max

A simple binary problem: maximize $f(x) = x_1 + x_2 + ... + x_n$ where $x_i \in \{0, 1\}$. The optimum $x^* = [1\ 1\ 1\ ...\ 1]$ gives $f(x^*) = n$.

- Individual representation is an n-dimensional binary string.
- Fitness is the sum of ones.
- GA configuration is generational, with random initialization, roulette wheel selection, one-point crossover, and single-bit flip mutation with $P_m = \frac{1}{n}$. The stopping condition is reaching the optimum.

2.16 Example: Traveling Salesman Problem (TSP)

Given a set of cities with distances, find the shortest route that visits each city once and returns to the origin.

- Individual representation is a permutation of symbols (each symbol = a city).
- Fitness is the length of the route.
- GA configuration is generational with random initialization, roulette wheel selection and the crossover operator is "Inver-Over," and a permutation mutation is used. The stopping condition is reaching the optimum.

2.17 Why/How Genetic Algorithms Work: Schemata Theory

A schema is a set of individuals with some common genes. The order of a schema is the number of defined positions and the defining length is the distance between the first and last defined position. The Building Block Hypothesis suggests that GAs seek near-optimal performance through the combination of short, low-order, high-performance schemata, called "building blocks". Short, low-order schemata receive exponentially increasing trials in subsequent generations. This theory has less credit today than in previous decades.

Chapter 3

Evolutionary Algorithms II

This chapter explores evolutionary algorithms beyond classical genetic algorithms, focusing on modern variants. These include Evolution Strategies, Evolutionary Programming, Differential Evolution, and Estimation of Distribution Algorithms. A key concept is the co-evolution of decision variables and algorithm parameters.

3.1 Evolution Strategies

Generalities

Evolution Strategies (ES) were developed by Rechenberg and Schwefel in 1973. They are real-valued algorithms primarily used for numerical optimization. Unlike genetic algorithms, each individual in ES consists of a vector x of decision variables and a set s of strategy parameters, such as mutation step-sizes. The algorithm aims to model the evolution of evolution by optimizing both the parameters of the problem and the algorithm itself. ES is considered solid, fast and reliable, and has a strong theoretical foundation.

Original vs. Modern ES

The original ES, known as (1+1)-ES, involved one parent and one offspring, using only uncorrelated mutations without recombination. Modern ES are multi-membered, employing multiple parents, correlated mutations, and recombination. These changes have led to significant performance improvements.

General Framework

The general framework involves initializing both decision variables and strategy parameters randomly. The algorithm then proceeds through typical evolutionary steps: selection of parents, crossover (if applicable) on both decision variables and strategy parameters, mutation of both, evaluation of fitness, and iterative repetition of these steps. The self-adaptation of mutation step sizes is a key difference from classical genetic algorithms, enabling the algorithm to determine the best search directions.

Mutation Strategies

Static Mutation

In static mutation, each individual generates an offspring using the formula $x_i' = x_i + \sigma \times N(0,1)$, where σ is a fixed mutation step-size.

Adaptive Mutation

Adaptive mutation uses the "1/5 rule" to adjust σ every k generations:

- $\sigma = \sigma/c$ if $p_s > 1/5$
- $\sigma = \sigma \times c$ if $p_s < 1/5$

- $\sigma = \sigma$ if $p_s = 1/5$

Here, p_s is the measured success probability of mutations, and c is usually between 0.8 and 1.0. Cumulative Step-size Adaptation (CSA) adjusts σ based on a cumulative path, combining previous step-sizes with weights that decrease with time.

Self-Adaptive Mutations

Self-adaptive mutations encode mutation parameters into the genotype.

Uncorrelated mutations with one σ Each individual is represented as $(x_1, x_2, ..., x_n, \sigma)$, where σ is a global step-size. Offspring are generated by:

- $\sigma' = \sigma \times \exp(\tau \times N(0,1))$
- $x_i' = x_i + \sigma' \times N(0,1)$

where $\tau \propto 1.0/\sqrt{n}$ is the learning rate.

Uncorrelated mutations with multiple σ Each individual is represented as $(x_1, x_2, ..., x_n, \sigma_1, \sigma_2, ..., \sigma_n)$, with each x_i having an independent mutation step-size σ_i. The offspring are generated by:

- $\sigma_i' = \sigma_i \times \exp(\tau' \times N(0,1) + \tau \times N(0,1))$
- $x_i' = x_i + \sigma_i' \times N(0,1)$

Here, $\tau' \propto 1.0/\sqrt{2n}$ is the global learning rate, and $\tau \propto 1.0/\sqrt{2\sqrt{n}}$ is the coordinate-wise learning rate.

Correlated mutations with multiple σ Each individual is represented as $(x_1, ..., x_n, \sigma_1, ..., \sigma_n, \alpha_1, ..., \alpha_k)$ where $k = n(n-1)/2$. This includes n independent mutation step-sizes and k rotation angles that represent interactions between step-sizes for each pair of variables. An $(n \times n)$-dimensional covariance matrix C is defined, with diagonal elements representing the variance σ_i^2 and off-diagonal elements representing interactions between variables. Offspring are generated by:

- $\sigma_i' = \sigma_i \times \exp(\tau' \times N(0,1) + \tau \times N(0,1))$
- $\alpha_j' = \alpha_j + \beta \times N(0,1)$
- $x' = x + N(0, C')$

where C' is the covariance matrix after mutation of σ and α values. The covariance matrix can be obtained by $C(\sigma, \alpha) = (S \times T)^T \times (S \times T)$, where S is a diagonal matrix of standard deviations, and T is the product of rotation matrices $R(\alpha_{ij})$.

Self-Adaptation Illustration

Self-adaptive ES can follow a moving optimum in a dynamic fitness landscape by adjusting the mutation step-size after every shift. In a static landscape, uncorrelated mutations with one σ lead to a circular distribution of offspring, while multiple σ result in an elliptical distribution. Correlated mutations allow the ellipsoid to rotate towards the optimal solution.

Selection and Recombination

Original ES versions use comma (,) and plus (+) notation for selection strategies, such as (1+1)-ES, (μ, λ)-ES, and $(\mu + \lambda)$-ES. Selection is based on the ranking of individuals' fitness, taking the μ best individuals (truncated selection). Modern ES use recombination, such as uniform crossover or arithmetic averaging of ρ solutions, denoted as $(\mu/\rho, \lambda)$-ES or $(\mu/\rho + \lambda)$-ES, where ρ is the mixing number. The value of ρ determines the type of crossover; $\rho = 2$ is local crossover, while $\rho > 2$ is global crossover. Some variants introduce a parameter κ for the maximum lifespan of an individual, denoted by (μ, κ, λ)-ES.

Comma vs Plus Strategies Comma strategies are more explorative, forgetting previous solutions, making them good for dynamic environments. Plus strategies are more exploitative and elitist, keeping the best solutions, making them good for large-scale problems or when marginal improvements are important.

Why Evolution Strategies Work

ES adapts mutation step-sizes based on time and space. In multi-membered ES, each individual has its own co-evolving mutation strategy. Different mutation step-sizes perform differently under different circumstances.

Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

CMA-ES is a state-of-the-art black-box optimization algorithm. It samples offspring using a multivariate Gaussian distribution. It learns from successful mutations to adapt both the step-size σ and the covariance matrix C, adjusting it to fit the fitness landscape. CMA-ES learns pairwise dependencies between variables, performs a Principle Component Analysis (PCA) on mutation step-sizes, and learns a new, rotated problem representation. It implicitly performs a PCA and learns dependencies between variables, adjusting the search to the fitness landscape.

3.2 Evolutionary Programming

Generalities

Evolutionary Programming (EP), developed by Fogel et al. in 1966, is one of the oldest EAs. Originally used for machine learning tasks with Finite State Machines (FSM), modern EP is applied to numerical optimization, often crossbred with ES. The main features of EP are deterministic reproduction, mutation only without recombination, and self-adaptation of mutation step-sizes.

General Framework

The framework is similar to ES. The key steps are: initialize the population, create offspring via mutation, evaluate fitness, and use a plus selection strategy. Each individual includes both problem variables $(x_1, x_2, ..., x_n)$ and mutation step-sizes $(\sigma_1, \sigma_2, ..., \sigma_n)$, similar to uncorrelated mutations with multiple σ in ES.

Algorithm Details

Representation Each individual is represented as $(x_1, x_2, ..., x_n, \sigma_1, \sigma_2, ..., \sigma_n)$, similar to ES.

Initialization A population of μ solutions is generated with x_i sampled uniformly within the decision space and σ_i sampled uniformly within.

Selection Each solution deterministically generates one offspring, independent of fitness, unlike other evolutionary algorithms.

Genetic Operators EP uses mutation only. Each offspring is generated using:

- $x_i' = x_i + \sigma_i \times N(0, 1)$
- $\sigma_i' = \sigma_i \times (1 + \alpha \times N(0, 1))$

where α is a parameter. The mutation step-size is updated *after* the variable, unlike in ES.

Survivor Selection Traditional EP uses $(\mu + \mu)$ selection, where μ parents and μ offspring are combined and sorted by fitness, selecting the best μ for the next generation (elitism). Stochastic variants evaluate each solution against q randomly chosen solutions. A "win" is assigned for each comparison where the solution is better, and the sum of wins (relative fitness) is used for selection.

Historical Applications

Evolving Finite State Machines (FSM) FSMs have states (S), inputs (I), and outputs (O), along with a transition function (δ: S x I $\rightarrow$ S x O). FSMs can be used as predictors, where the output should match the subsequent input. In the mid-1960s, Fogel used EP to evolve FSMs for tasks like predicting the next input in a sequence or predicting if a number is prime.

"Blondie 24" In 2002, Fogel evolved a checkers player using a neural network with 5046 weights and an additional weight for kings. The representation included weights and corresponding σ values. The program played against other programs without human trainers and achieved "expert class" ranking, outperforming 99.61% of rated players.

3.3 Differential Evolution

Generalities

Differential Evolution (DE), proposed by Storn and Price in 1995, is a milestone in continuous optimization. It combines aspects of evolutionary algorithms and swarm intelligence. DE uses three parameters: population size (N), scale factor (SF), and crossover probability (CR). It is known for its ease of implementation and power.

Algorithm Details

DE involves an initialization step, followed by an iterative process where for each solution in the population, mutation and crossover are applied. Selection is implemented by one-to-one spawning: the offspring replaces the parent if it is better.

Mutation

DE uses differential vectors for mutation. A differential vector is the difference between two vectors representing solutions in the population, which is used to perturb a current solution. Common mutation strategies include:

- rand/1: $x' = x + SF \times (a - b)$
- rand/2: $x' = x + SF \times (a - b) + SF \times (c - d)$
- best/1: $x' = best + SF \times (a - b)$
- best/2: $x' = best + SF \times (a - b) + SF \times (c - d)$
- cur-to-best/1: $x' = x + SF \times (best - x) + SF \times (a - b)$
- rand-to-best/2: $x' = a + SF \times (best - x) + SF \times (a - b) + SF \times (c - d)$

Here, a, b, c, and d are randomly chosen, mutually exclusive individuals from the population, and *best* is the best individual. The scale factor SF typically ranges from (0,2].

Crossover

DE uses two types of crossover: binomial ("bin") and exponential ("exp"). Binomial crossover swaps genes with a probability CR across all n variables and ensures that at least one gene is swapped. Exponential crossover swaps genes until a random number exceeds CR, using a geometric distribution with success probability equal to CR.

- Binomial crossover: Avg no. swapped genes $= 1 + CR \times n$
- Exponential crossover: Avg no. swapped genes $= (1 - CR^n)/(1 - CR)$

Why Differential Evolution Works

DE's effectiveness stems from the shrinking of randomized movements during optimization. Initially, differential vectors are large due to the wide spread of solutions, enabling large search movements. As solutions converge, differential vectors become smaller, allowing for search refinements. This constitutes an implicit form of adaptation. DE is explorative at the start of optimization and exploitative towards the end.

3.4 Estimation of Distribution Algorithms

Generalities

Estimation of Distribution Algorithms (EDAs) are a class of EAs that build and evolve an explicit probabilistic model of the population rather than using a set of solutions. They are also called Probabilistic Model-Building Genetic Algorithms (PMBGAs). Examples include PBIL, UMDA, EMNA, BMDA, and BOA.

Note CMA-ES is similar to an EDA, but in CMA-ES the covariance matrix models a distribution of mutation step-sizes rather than solutions.

Example: Population-Based Incremental Learning (PBIL)

PBIL is suitable for binary problems. It starts with a population vector where each bit has a 0.5 probability. Solutions are sampled from this distribution, and the best solutions are used to update the model by averaging. The probabilities in the population vector are biased towards the most promising solutions.

3.5 Conclusion

This lecture covered modern variants of evolutionary algorithms, including Evolution Strategies, Evolutionary Programming, Differential Evolution, and Estimation of Distribution Algorithms. Each algorithm offers unique approaches to optimization and adaptation, extending beyond the classical genetic algorithm paradigm.

Chapter 4

Multi-Objective Evolutionary Algorithms

Multi-objective optimisation deals with problems that involve multiple, often conflicting, objectives. Unlike single-objective optimization where the goal is to find one optimal solution, multi-objective problems require finding a set of solutions that represent the best possible trade-offs between different objectives. These problems are very common in the real-world.

4.1 Multi-Objective Optimization Problems (MOPs)

Definition

Many real-world problems are characterised by the presence of multiple, potentially conflicting objectives. Examples include:

- Buying a car: Balancing comfort against price.
- Buying a house: Considering location, size, quality and price.
- Choosing a job: Balancing location, salary, flexibility, and contract duration.
- Engineering design: Balancing lightness against strength.

When considering these objectives separately, different optimal solutions may be obtained. For instance, when buying a car, focusing solely on price might lead to a very cheap but uncomfortable car, while prioritizing comfort might result in a very expensive car. The goal of multi-objective optimization is to find solutions that represent different trade-offs between the objectives. An ideal solution would be one that is optimal for all objectives simultaneously, but this is rarely possible due to the conflicting nature of the objectives. For example, a car that costs the minimum (15k) and has the maximum comfort (9) would be ideal, but such a car is usually not available.

4.2 Conventional Approaches ("A Priori" Methods)

These methods require some decisions to be made before the optimization process begins.

Scalarization

Different objectives are combined into a single, linear or non-linear function. A common approach is the weighted sum:

$$f = w_1 \times f_1 + w_2 \times f_2 + ...$$

Where f_i represent the different objective functions, and w_i the associated weights. It's important to consider how each objective increases/decreases when f is minimized or maximized. Objectives often have different orders of magnitude, making the magnitude of weights crucial. Different weights correspond to different rankings (importance) of the objectives. This approach restricts the fitness space, requiring re-optimization with different weights to find different trade-off solutions.

Lexicographic Ordering

Objectives $(f_1, f_2, ..., f_k)$ are ranked in a user-defined order of importance. The problem is solved as a sequence of single-objective optimization problems. First, f_1 is optimized, then f_2 is optimized with a constraint on f_1, and so on. For example, in car selection, the user might decide that price is more important than comfort. The car with the minimum price is selected and then if there are ties, the car that maximizes comfort is selected. Conversely, if comfort is more important than price, then the car with the maximum comfort is selected, and if there are ties, the car with the lowest price is selected.

epsilon-Constraint Method

k single-objective optimization problems are solved separately. For each problem, $k-1$ constraints are imposed using user-defined thresholds (ϵ_j). For example, minimizing price subject to comfort being greater than or equal to 7, and then maximizing comfort subject to price less than 30k. These methods require some a priori information such as the ranking of objectives and user-defined thresholds. A key question is whether all objectives can be considered simultaneously, without resorting to a priori methods.

4.3 Pareto Dominance

Definition

A solution i is said to **Pareto-dominate** a solution j if:

- i is no worse than j on all objectives.
- i is better than j on at least one objective.

Given a solution, we can identify solutions that are better (dominating), worse (dominated), or incomparable. Incomparable solutions are better in one objective but worse in others. Pareto dominance is evaluated in the fitness space, not the design space. There is no single optimal solution, but there is a set of non-dominated solutions.

Pareto Front

The **Pareto front** is the set of non-dominated solutions. These solutions are not dominated by any other solutions. Solutions on the Pareto front are considered "good" solutions. The goal of multi-objective optimization is to find the Pareto front.

4.4 Multi-Objective Optimization (MOO) vs Multi-Criteria Decision Making (MCDM)

Multi-objective optimization problems are usually solved in two steps:

1. Find the Pareto front (optimization).
2. Select a solution based on particular interests (decision-making).

The main difference between a priori and a posteriori methods lies in when decisions are made:

- **A priori methods**: Decisions are made before optimization (e.g., ranking objectives, defining constraints).
- **A posteriori methods**: Decisions are made after optimization (based on the Pareto front).

A posteriori methods allow for learning about the problem, the trade-off surface, interactions among criteria, and structural information.

4.5 The Pareto Front: Shapes and Properties

The Pareto front can have different shapes and properties (convex/non-convex, continuous/discontinuous) depending on the problem and the minimization/maximization of each objective. Some solutions can be locally non-dominated, forming a local Pareto front. The goal is to find the global Pareto front. If the Pareto front is non-convex, some regions cannot be found by linear scalarization. Linear scalarization corresponds to intersecting the Pareto front with a plane, and some parts of a non-convex front may not be intersected by any plane. This is a key reason why Pareto-based approaches are preferred over scalarization. The Pareto front is usually a set of discrete points, although it is visually represented as a continuous line.

4.6 Problems with Linear Scalarization

Linear scalarization can miss some regions of a non-convex Pareto front. This is because linear scalarization essentially intersects the Pareto front with a plane, and certain areas may not be intersected with any combination of weights. To find all the solutions with a scalarization approach one would need to run the optimization with every possible combination of weights.

4.7 Crowding

Crowding refers to how close non-dominated solutions are to each other. Ideally, one wants to cover the entire Pareto front as evenly as possible, achieving a uniform density. Crowding can lead to a situation where some areas of the Pareto front are over-represented, while others are not covered at all. Algorithms should try to distribute solutions uniformly across the Pareto front.

4.8 Advantages of Evolutionary Algorithms

- Population-based search allows simultaneous search for points approximating the Pareto front.
- No need to make guesses about which combinations of weights are best.
- No need to make assumptions about the Pareto front shape, since they can be non-convex or discontinuous.

4.9 Important Aspects for Multi-Objective Optimization

- **Fitness Assignment**:
  - Weighted sum, with weights adapted during evolution.
  - Separate evaluation of each objective by a different subpart of the population (e.g., VEGA).
  - Pareto-dominance.
- **Diversity Preservation**:
  - Penalize "crowded" areas of the Pareto front.
  - Attempt to cover the front uniformly.
  - Fitness sharing.
  - Crowding distance sorting.
- **Memory of Non-Dominated Points**:
  - Archive of non-dominated points, which can undergo mutation and recombination.
  - Elitist selection (e.g., $(\mu + \lambda)$ strategy).

4.10 An Early Approach: Vector-Evaluated Genetic Algorithm (VEGA)

VEGA evaluates each solution for each objective function separately. With k objectives and N solutions, there are k populations of N individuals, resulting in $N \times k$ independent evaluations. For each objective, the best N/k individuals are selected, and then combined into a single population. However, fitness-proportionate selection corresponds to a linear combination of the objectives, where the weights depend on the population distribution. There is no explicit diversity preservation mechanism in VEGA.

4.11 Non-Dominated Sorting Genetic Algorithm-II (NSGA-II or NSGA-2)

Proposed by Deb et al. in 2002, NSGA-II is a milestone in multi-objective optimization. Typically used for binary or real-valued representations. Genetic operators (mutation and crossover) are the same as those used for single-objective optimization.

Selection Mechanism

The selection mechanism is composed of two parts:

1. **Pareto rank** based on non-dominated sorting, selecting solutions according to their dominance levels. Solutions are ranked based on how many other solutions they dominate, with non-dominated solutions assigned rank 1, next front rank 2, etc..
2. **Crowding-distance sorting** prefers "isolated" solutions, improving Pareto front representation. This also acts as a diversity preservation mechanism. Solutions are ranked based on their distance to other solutions in the fitness space. Solutions in less crowded areas are preferred.

Curse of Dimensionality

Performance breaks down as the number of objectives increases. As the number of objectives increases, it becomes harder to find non-dominated solutions, because the probability of generating a non-dominated solution becomes increasingly smaller. The region of the fitness space containing better solutions decreases exponentially with the number of objectives.

Pareto Front Levels

Solutions are ranked according to their level of dominance. Non-dominated solutions (Pareto front) have rank R_1, solutions dominated only by R_1 solutions have rank R_2, and so on. This process is recursive, identifying multiple fronts.

4.12 Algorithm Details: Non-Dominated Sorting

The algorithm assigns a rank to each solution based on how many other solutions dominate it. The fast non-dominated sorting algorithm identifies the different Pareto fronts.

4.13 Algorithm Details: Crowding-Distance Sorting

Crowding should be avoided, and the Pareto front should be covered as evenly as possible. The crowding distance is a measure of how isolated a solution is in the fitness space. Solutions with a higher crowding distance are preferred. Extreme solutions are assigned infinite crowding distance.

4.14 Algorithm Details: Putting it All Together

Parent and offspring populations are combined before sorting. This is an elitist strategy. First preference is given to Pareto rank, then to crowding-distance. The algorithm sorts all individuals (parents and offspring) based on Pareto rank and then uses crowding distance to select the best individuals for the next generation.

4.15 Conclusion

Multi-objective optimisation is essential for solving real-world problems, where the solutions have different trade-offs between conflicting objectives. Understanding concepts such as scalarization, lexicographic ordering, epsilon-constraint, and Pareto-dominance are important. Techniques such as NSGA-2 are based on Pareto dominance and crowding, and provide state-of-the-art techniques to obtain the Pareto Front.

Chapter 5

Constrained Evolutionary Algorithms

5.1 Constrained Optimization

Motivation

Evolutionary Algorithms (EAs) are inherently unconstrained search techniques, limited only by boundary constraints. However, practical problems often involve numerous equality and inequality constraints, both linear and nonlinear. This necessitates mechanisms for EAs to handle such constraints. The basic idea of constrained optimization is that we not only have a defined search space with upper and lower boundaries for the variables, but we also have additional constraints that further restrict the feasible search space. We need to find optimal solutions that satisfy not only the search space but also these additional constraints.

Constraint Handling Mechanisms

Several techniques have been developed to enable EAs to deal with equality and inequality constraints. These include:

- **Penalty functions**
- **Separation of constraints and objectives**
- **Special representations and operators**
- **Repair algorithms**

5.2 Penalty Functions

Core Idea

The main concept behind penalty functions is to transform a constrained optimization problem into an unconstrained one. This is achieved by adding (or subtracting) a value to the objective function based on the constraint violation of a given solution. This approach, initially proposed by Richard Courant in the 1940s for classical optimization, is now widely used in the EA community.

Types of Penalty Functions

In classical optimization, two types of penalty functions are considered:

- **Exterior Methods:** These start with an infeasible solution and move towards the feasible region as the penalty decreases. The penalty is high in unfeasible areas and decreases as the solution approaches feasibility. This method is most commonly used with EAs.

- **Interior Methods:** Here, the penalty is small for points far from constraint boundaries and increases to infinity as boundaries are approached. This method requires an initial feasible solution. Thus, this method is less commonly used with EAs.

Static Penalty

EAs typically adopt exterior penalty functions of the form:

$$\Phi(x) = f(x) + \sum_{i=1}^{jmax} r_i G_i(x) + \sum_{j=1}^{kmax} c_j L_j(x)$$

where $\Phi(x)$ is the new objective function, G_i and L_j are functions of the constraints $g_i(x)$ and $h_j(x)$, and r_i and c_j are positive constants called penalty factors. The most common forms for G_i and L_j are:

$$G_i(x) = max(0, g_i(x))^\beta$$

$$L_j(x) = |h_j(x)|^\gamma$$

where β and γ are typically 1 or 2. Equality constraints are often transformed into inequality constraints using a tolerance (ϵ): $|h_j(x)| - \epsilon \leq 0$.

Another static penalty approach adds a penalty factor proportional to the number of violated constraints:

$$\Phi(x) = f(x) + W \times penalty(x)$$

where W is a constant and $penalty(x)$ counts the number of violated constraints, a value of 1 is added if the constraint is violated, otherwise 0.

Dynamic Penalty

In dynamic penalty methods, the penalty function changes over time. A common approach increases the penalty over generations, such as the method proposed by Joines and Houck:

$$\Phi(x,t) = f(x) + C \times t^\alpha \times \left(\sum_i \max(0, g_i(x)) + \sum_j \max(0, |h_j(x)| - \epsilon) \right)^\beta$$

where C, α, and β are user-defined constants, and t is the generation number. Typically, the penalty increases as the evolutionary process proceeds, encouraging convergence to feasible areas.

Adaptive Penalty

Adaptive penalty methods adjust the penalty function based on feedback from the search process. Bean and Hadj-Alouane developed a method where each solution is evaluated by:

$$\Phi(x,t) = f(x) + \lambda(t) \times \left(\sum_i \max(0, g_i(x))^2 + \sum_j |h_j(x)|^2 \right)$$

The penalty component, $\lambda(t)$, is updated every generation:

$$\lambda(t+1) = \begin{cases} \lambda(t)/\beta_1 & \text{if all best solutions in last } k \text{ generations were feasible} \\ \lambda(t) \times \beta_2 & \text{if all best solutions in last } k \text{ generations were infeasible} \\ \lambda(t) & \text{otherwise} \end{cases}$$

where β_1, $\beta_2 > 1$ and $\beta_1 \neq \beta_2$, and k is a user-defined parameter representing the generation gap. This method decreases the penalty if recent best solutions were feasible and increases it if they were infeasible.

Main Issues of Penalty Functions

The ideal penalty factors are highly problem-dependent, even in dynamic or adaptive penalty methods. If the penalty is too high, the EA might quickly be pushed inside the feasible region, making it hard to explore the boundaries. If the penalty is too low, the EA may spend excessive time exploring the infeasible region.

Other Penalty-Based Approaches

ASCHEA

The Adaptive Segregational Constraint Handling Evolutionary Algorithm (ASCHEA) uses evolution strategies with an adaptive penalty, constraint-guided recombination, and segregational selection. The adaptive penalty function is given by:

$$\Phi(x) = \begin{cases} f(x) & \text{if } x \text{ is feasible} \\ f(x) - \alpha \sum_j g_j^+(x) + |h_j(x)| & \text{if } x \text{ is infeasible} \end{cases}$$

where $g_j^+(x)$ represents the positive part of $g_j(x)$. The penalty factor α is adapted based on the desired ratio of feasible solutions (τ_{target}) and the current ratio (τ_t). The recombination operator combines infeasible and feasible solutions when $\tau_t < \tau_{target}$, while the segregational selection operator selects a ratio (τ_{select}) of feasible solutions for the next generation.

Stochastic Ranking

This method uses multi-membered Evolution Strategies with a penalty function and a ranking process. It balances the influence of the objective and penalty functions using a user-defined parameter P_f, and does not require a penalty factor. The population is sorted using a stochastic bubble sort-like algorithm, where comparison is based on either the objective function or constraint violations with a probability of P_f or $1 - P_f$ respectively. Empirically, $0.4 < P_f < 0.5$ produces the best results. This method is easy to implement and requires fewer fitness evaluations than other methods.

5.3 Separation of Constraints and Objectives

Core Idea

Unlike penalty functions that combine the objective function and constraints, these approaches handle constraints and objectives separately.

Examples

- **Behavioral memory**
- **Coevolution** (using two populations)

Multi-objective Optimization Concepts

The main idea is to redefine the single-objective optimization of $f(x)$ as a multi-objective problem with $m + 1$ objectives, where m is the number of constraints. Then, any multi-objective EA (e.g., NSGA-II) can be used. The concept of constrained domination is used to extend the Pareto dominance to constrained optimization:

Single-objective Optimization

When comparing two solutions:

- Both feasible: Choose the solution with better objective function value.
- One feasible, one infeasible: Choose the feasible solution.
- Both infeasible: Choose the solution with smaller constraint violation.

Multi-objective Optimization

Solution i constrains dominates solution j if:

- Solution i is feasible and j is not.
- Both are infeasible, and i has a smaller constraint violation.

- Both are feasible, and i Pareto dominates j.

This means that feasible solutions have a better non-domination rank than infeasible ones. Feasible solutions are ranked according to their non-domination level while among infeasible solutions, the one with the smallest constraint violation has a better rank.

5.4 Special Representations and Operators

Special Representations

For difficult problems, a generic representation might not be suitable. Special representation schemes and genetic operators are designed for these cases, such as Random Keys or Homomorphous Maps. Homomorphous Maps transform the feasible region into an easier shape to explore, such as an n-dimensional cube.

Special Operators

Special genetic operators can be designed to produce offspring that lie on the boundary between feasible and infeasible regions, or that are feasible by construction.

5.5 Repair Algorithms

Core Idea

Repair algorithms transform an infeasible solution into a feasible one.

Repair Mechanisms

Possible repair mechanisms include:

- Greedy algorithms (optimization algorithms making the best decision at each step)
- Random algorithms (based on randomized perturbations of unfeasible solutions)
- Custom heuristics (problem-dependent)

Handling Repaired Solutions

Repaired solutions can be used either for fitness evaluation only ("never replacing") or to replace the original unfeasible solutions:

- Always replace ("always replacing")
- Replace with some probability (e.g., "5% rule" for combinatorial problems)

Replacement can be seen as a form of Lamarckian evolution.

Examples of EAs Based on Repair Mechanisms

- GENOCOP III (maintains separate populations of search and reference points)
- Problem-dependent repair algorithms (e.g., for robot path planning)

Repair algorithms are a good choice when infeasible solutions can be easily transformed into feasible ones, however, they may introduce search bias.

5.6 Hybrid Methods

These methods combine EAs with other search techniques.

Examples

- Genetic Algorithm + Artificial Immune System (AIS): Using feasible solutions as antigens and evolving antibodies (infeasible solutions) to be similar to antigens.
- ϵ-constrained method: Relaxing constraints using a satisfaction level (ϵ) with a lexicographic order, often used in Differential Evolution with gradient-based mutation and a modified Nelder-Mead method.

5.7 Recent Ideas

Ensemble of Constraint-Handling Techniques

Using several constraint-handling techniques, each with its own population and parameters. Offspring compete both within and across populations to automate the selection of the best technique for a problem.

Hybrid Methods with Gradient-Based Information

Incorporating gradient information from fitness or constraints, such as combining EAs with gradient-based local search, Sequential Quadratic Programming (SQP), or Support Vector Machines (SVM).

Modified CMA-ES

Applying adaptive penalty functions or modifying the Covariance Matrix update to sample only in the feasible region.

Viability Evolution

Eliminating unviable solutions based on shrinking feasibility boundaries that are adapted at runtime.

5.8 Recap of Constraint Handling Techniques (CHT)

Technique	Pros	Cons
Penalty-based		
Death	Simple implementation	No gradual "push" towards feasibilit
Static	Simple implementation	Needs extra parameters/weights
Dynamic	Adjusts penalty over time	Needs extra parameters/weights
Adaptive	Uses feedback from search	Extra parameters, hard to design ad
ASCHEA	Controls # feas. solutions	Extra parameters, needs large budge
SR	Needs (relatively) small budget	1 extra parameter (P_f)
Separation of constraints and objectives		
	No need for penalty, efficient handling also in MO cases	No control
Special representations & operators		
	Can be very efficient on specific problems	Complexity, problem-specificity
Repair algorithms	Efficient if repair is cheap	Complexity, problem-specificity, poss
Hybrid methods	Can be very efficient	Extra parameters, complexity
Recent ideas	Can be very efficient	Extra parameters, complexity, may r

5.9 Concluding Remarks

Various EAs have been used for constrained optimization with different constraint handling techniques. Evolution Strategies and Differential Evolution seem more suited than Genetic Algorithms. The (μ + λ) strategy tends to produce better performance, and elitism is needed. Diversity is important; keeping only feasible solutions is not always a good idea. Research in this field is ongoing.

Chapter 6

Swarm Intelligence I: Particle Swarm Optimization

Swarm intelligence is a field of study that draws inspiration from the collective behaviour of social animals such as bird flocks, fish schools, or ant colonies, to solve complex problems. This lecture focuses on Particle Swarm Optimization (PSO) as a primary example of swarm intelligence.

6.1 Particle Swarm Optimization

A Treasure Hunt Game

To introduce the concept of PSO, the lecture uses an analogy of a treasure hunt game.

- **Scenario:** Imagine a group of friends searching for a treasure on an island.
- **Tools:** Each friend has a metal detector and can access information about the signal strength and their neighbours' positions.
- **Goal:** The goal is to find the treasure, or at least get close to it.
- **Reward Sharing:** All participants who take part in the search will be rewarded, with the person who finds the treasure receiving a higher reward. The rest will be rewarded based on their distance from the treasure at the time when the treasure is found.

The key idea is that individuals (treasure hunters) share information and learn from each other to converge to the solution (treasure location) more effectively. This highlights the benefit of social information in problem-solving, especially for difficult problems.

BOIDS

As a first historical example of Computational Swarm Intelligence, the lecture introduces BOIDS. BOIDS simulates the flocking behavior of birds using simple agents called "boids". Each boid perceives the angle and distance of its neighbours.

Reynolds Flocking Rules: These rules model the complex behaviour of flocks based on three simple rules.

1. **Separation:** A boid maintains a certain distance from other boids.
2. **Cohesion:** A boid moves toward the centre of mass of its neighbouring boids.
3. **Alignment:** A boid aligns its direction with those of its neighbouring boids.

By combining these simple rules, complex and stable behaviours emerge, such as flocking and obstacle avoidance. It's important to note that BOIDS was not designed for optimization but for simulating swarm behaviour.

Generalities of Particle Swarm Optimization

Biological Inspiration: PSO is inspired by the foraging behaviour of bird flocks. Birds do not know the location of the food, but each bird can tell its neighbours how much food is in its current location, and each bird remembers its location where it found the highest concentration of food so far.

Strategies in PSO: Birds combine three strategies to find food efficiently:

1. **Brave:** Keep flying in the same direction.
2. **Conservative:** Fly back towards the best previous position.
3. **Swarm:** Move towards the best neighbour.

From Birds to Particles: In PSO, birds are abstracted as "particles," where the food concentration is analogous to the fitness landscape of the optimization problem. The position of the birds corresponds to solutions for that problem.

Particle Definition

A particle **p** is described by **¡x, v, f¿**, where:

- **x**: Represents the particle's position, which is a candidate solution, equivalent to the genotype in Evolutionary Algorithms.
- **v**: Represents the particle's velocity, used to move or perturb the solution.
- **f(x)**: Represents the particle's performance, which is the value of the objective function at that position (can be multi-objective).

Particles also have perception; they perceive the performance and positions of neighbouring particles and know the best particle among their neighbours (z). Additionally, particles remember their own best position where they obtained the best performance (y). Each particle maintains a memory of its own best position (y), where it found the highest fitness value, combining local information with that from its neighbours.

Neighborhood Definitions

Neighborhoods can be defined at different levels:

- **Global:** The best individual in the neighbourhood is the global best in the entire swarm. In this case, all particles consider the best of the entire swarm as its neighbour best and so the global best (z) is the same for all particles.
- **Distance-based:** Based on a proximity metric such as Euclidean distance between the particles' positions, using a given neighbourhood radius. Particles within this radius are considered neighbours.
- **List-based:** Based on a predetermined topology arranging the solution indexes according to some order/structure and a given neighbourhood size. For example, a ring structure where each particle only sees the particles with adjacent indexes or a star structure where every node is connected to all other nodes.

Different topological structures can be used to define social networks such as Ring, Star, Von Neumann, Pyramid, Clusters and Wheel.

Dynamic Neighbourhoods: Overlapping neighbourhoods are also possible, facilitating information exchange. Growing neighbourhoods start with a ring topology and grow toward a star topology, adding particles based on a scaled Euclidean distance. This allows for a transition from exploration to exploitation as the search progresses.

Hypercube Structure: This structure is used for binary-valued problems. Particles are defined as neighbours if the Hamming distance between the bit representation of their indices is one. The total number of particles must be a power of two, and the neighbourhood has **n** particles. Each neighborhood has exactly **n** particles, the maximum distance between any two particles is exactly **n**, and if particles **i** and **j** are neighbours, then **i** and **j** will have no other neighbours in common.

Main Parameters

Key parameters in PSO include:

- **Swarm Size:** The number of particles, typically 20 for problems with dimensionality 2-200.
- **Neighbourhood Size:** Typically 3 to 5, otherwise a global neighbourhood is used.
- **Velocity Update Factors**: Depend on the specific PSO implementation.

Standard PSO

The standard PSO algorithm is mostly used for continuous optimisation:

1. A swarm of particles is initialized with random positions and velocities.
2. At each step, every particle updates its velocity using the rule:

$$v' = w \cdot v + \Phi_1 U_1 \cdot (y - x) + \Phi_2 U_2 \cdot (z - x)$$

3. Then, each particle updates its position:

$$x' = x + v'$$

4. If the performance improves, the personal best position ($\mathbf{y}$) is updated, and, where appropriate, the neighbourhood best position ($\mathbf{z}$) is updated.

Where:

- $\mathbf{x}$ and $\mathbf{v}$: are the current position and velocity of the particle, respectively.
- $\mathbf{y}$ and $\mathbf{z}$: are the personal and social/global best positions, respectively.
- $\mathbf{w}$: is the inertia (weighs the current velocity).
- Φ_1, Φ_2: are acceleration coefficients/learning rates (cognitive and social, respectively).
- U_1 and U_2: are uniform random numbers in the range.

Velocity Components

The velocity update rule comprises three key components:

- **Inertia Velocity**: $w \cdot v$ represents the momentum or memory of the previous flight direction. It prevents the particle from drastically changing its direction.
- **Cognitive Velocity**: $\Phi_1 U_1 \cdot (y - x)$ quantifies performance relative to past performance, using the memory of the previous best position. It represents a form of "nostalgia" that pulls the particle back to it's best position.
- **Social Velocity**: $\Phi_2 U_2 \cdot (z - x)$ quantifies performance relative to neighbours. It represents a form of "envy" that pulls the particle toward the best neighbour's position.

Velocity Models

Two simplified velocity models can be considered:

- **Cognition-only model**: $v' = w \cdot v + \Phi_1 U_1 \cdot (y - x)$. In this model particles are independent hill-climbers, performing a local search.
- **Social-only model**: $v' = w \cdot v + \Phi_2 U_2 \cdot (z - x)$. In this model, the swarm acts as one stochastic hill-climber.

Geometric Interpretation of Velocity

The velocity update can be visualised as the summation of vectors representing the inertia, cognitive, and social components. The new position is obtained by adding the updated velocity to the current position.

Acceleration Coefficient Parameterisation

The choice of Φ_1 and Φ_2 affects the behaviour of the particles:

- $\Phi_1 = \Phi_2 = \mathbf{0}$: Particles move only based on inertia (no learning).
- Φ_1 **¿ 0,** $\Phi_2 = \mathbf{0}$: Cognition-only model.
- $\Phi_1 = \mathbf{0},$ Φ_2 **¿ 0**: Social-only model.
- $\Phi_1 = \Phi_2$ **¿ 0**: Particles are attracted towards the average of $\mathbf{y}$ and $\mathbf{z}$.

Additional considerations:

- $\Phi_1 < \Phi_2$: More beneficial for unimodal problems where the algorithm benefits from prioritising the social component.
- $\Phi_1 > \Phi_2$: More beneficial for multimodal problems where it is better to favour exploration using the cognitive component.
- **Low** Φ_1 **and** Φ_2: Leads to smooth particle trajectories.
- **High** Φ_1 **and** Φ_2: Results in higher acceleration and abrupt movements.

Convergence Conditions

The theoretical condition for convergence depends on $\mathbf{w}$, Φ_1 and Φ_2:

$$1 > w > \frac{1}{2}(\Phi_1 + \Phi_2) - 1 \geq 0$$

.

PSO in Action

The lecture demonstrates how PSO operates on a multimodal problem where particles converge to the area with the lowest objective function value. The algorithm is usually very efficient in finding the optimum, converging quickly.

Implementation Notes

The velocity update rule can be applied in two ways:

- **Synchronous**: Personal and social bests are updated after all particle positions are updated, this leads to slower feedback (better for global best).
- **Asynchronous**: Personal and social bests are updated after each particle position update, this provides immediate feedback about the best regions of the search space (better for distance/list-based best).

Similarities and Differences Between PSO and EAs

Similarities: Both PSO and Evolutionary Algorithms (EAs) are population-based algorithms. PSO uses two memory structures: one for personal best solutions and another for current positions, which is similar to having two populations in EAs. In PSO, particle positions "evolve" over time.

Differences: The main difference lies in the logic of selection. In EAs, selection is fitness-based and takes into account the entire population, while in PSO, selection is done by considering the solution before and after the perturbation (personal best is updated only in case of improvement). This is referred to as one-to-one spawning.

6.2 PSO Variants

Over the past 20 years, numerous PSO variants have been proposed. These variants generally focus on:

- Modifications to the velocity update rule.
- Modifications to the algorithm logic.

The main objectives of these variants are to improve convergence speed, quality of solutions, and the ability to converge.

Modifications of Velocity Update Rule

- **Dynamic Inertia**: The inertia weight **w** is updated dynamically to allow transition from exploration to exploitation. **w** >1, will lead to higher velocities, which can be used for exploration. When 0 <**w** <1 particles decelerate leading to more exploitation. Various methods for dynamic inertia include, random sampling from a Gaussian distribution, decaying w linearly at each iteration or multiplying w by a constant.
- **Velocity Clamping**: The velocity **v** is limited to control exploration. Velocity clamping involves saturating the velocity to a given threshold **Vmax** to control global exploration.
- **Adaptive Acceleration Coefficients**: Φ_1 and Φ_2 are updated dynamically to allow transition from exploration to exploitation. Typically Φ_1 is decreased over time while Φ_2 is increased to promote exploration at the beginning of the search and exploitation towards the end.
- **Fully Informed PSO**: The velocity equation is modified such that each particle is influenced by the successes of all its neighbours, using a weighted sum of their best positions. This allows for more information to be used, and influence is proportional to the particle's fitness.
- **Prey-Predator PSO**: Introduces "virtual" particles (predators) that force exploration. Prey particles are repelled by predators which is achieved by modifying the velocity rule to increase the velocity exponentially as the distance to a predator decreases.
- **Comprehensive Learning PSO (CLPSO)**: For each variable, it replaces probabilistically the personal best with a "functional local best" using a tournament selection between two random particles.

Modifications of Algorithm Logics

- **Guaranteed Convergence PSO (GPSO)**: Forces global best update in case of null velocities. It addresses stagnation by perturbing the global best position when **x=y=z** to ensure the particles keep moving and searching.
- **Multi-start PSO**: Injects randomness into the particles' positions to increase diversity. This can help the algorithm escape local optima, but it needs a careful balance of randomness and convergence.
- **Cooperatively Coevolving Particle Swarms (CCPSO2)**: Randomly divides the problem into **n** separate sub-problems and does not use velocity update, but samples solutions from a Gaussian or Cauchy distribution. This approach is often used for large scale problems, and each sub-swarm acts on a random subset of variables. The context vector is created by concatenating the global best of each subswarm to evaluate the solution and each sub-PSO cooperates with the best from the other swarms.

6.3 Recap of PSO Variants

A table is included in the slides summarising the PSO variants and their descriptions. The lecture highlights that this area of research is still active and new variants are devised each year.

Chapter 7

Swarm Intelligence II: Ant Colony Optimization

Ant Colony Optimization (ACO) is a swarm intelligence algorithm inspired by the foraging behavior of ants. It is particularly effective for solving combinatorial problems. Unlike Particle Swarm Optimization where each particle represents a solution, in ACO, each ant is an agent that constructs a solution iteratively.

7.1 Emergent Problem Solving in Ants

Ant colonies exhibit remarkable collective problem-solving abilities with no central coordination. The tasks they perform often require cooperation and complex interaction. These tasks include:

- Regulation of nest temperature: Some ant species can regulate nest temperature within approximately 1 degree Celsius of a desired temperature by closing gaps and vibrating their bodies.
- Forming bridges: Ants use their bodies to create bridges over gaps, allowing other ants to cross.
- Building and protecting the nest: Ants have different roles, with some specialised for protection and others for construction.
- Sorting brood, corpses, garbage and food items: Ants keep their nests tidy and clean by sorting different items.
- Cooperating in carrying large items: Ants can work together to move objects much larger than themselves.
- Emigration of a colony: Ants can decide to move their nest based on conditions like temperature or food scarcity.
- Finding the shortest route from nest to food: Ants are able to find the shortest path between their nest and food sources through a process called stigmergy.

These are examples of emergent swarm behavior where the collective action of simple agents leads to complex problem solving.

Stigmergy: Communication Through the Environment

Stigmergy is a form of communication where individuals interact by modifying their environment. These modifications are then perceived by other individuals. In the case of ants, this communication occurs primarily through pheromone trails.

Types of Stigmergy:

- *Sematectonic*: Agents modify the environment, and other agents use these changes to exchange information.
- *Sign-based*: Agents use signals (e.g., chemical or auditory) to communicate through the environment.

In ant colonies, stigmergy manifests when ants leave pheromone trails. These trails allow other ants to find paths between food and nest. The more ants follow a path, the more pheromone is deposited, reinforcing that path and attracting even more ants.

Pheromone Trails and Path Optimization

When ants explore, they leave pheromone trails, and other ants tend to follow the paths with higher pheromone concentrations.

- Initially, without pheromones, ants have an equal probability of choosing a long or short path.
- Shorter paths get more traffic, and thus more pheromones, and less decay because of the increased traffic.
- The result is a positive feedback loop where the shortest paths are reinforced through this process.
- This mechanism allows ants to solve graph based problems (like finding the shortest path) in a fully decentralized manner.

7.2 Ant Colony Optimization Algorithm

Generalities of ACO

ACO is an algorithm developed by Dorigo et al. in 1992, inspired by stigmergic communication to find the shortest path in a network. The original implementation was called the Ant System (AS). **Typically, ACO assumes an optimization problem represented as a graph problem**.

Common Applications:

- Combinatorial problems over networks (e.g., routing)
- Travel Salesman Problem (TSP)
- Vehicle Routing Problem (VRP), including variants with time windows (VRPTW).
- These problems are commonly used in real-world logistics and scheduling by companies such as British Telecom, MCI Worldcom, Barilla, Migros.

Modern ACO algorithms often include hybridizations with other metaheuristics and local search. Extensions also include self-adaptation and solutions to continuous and multi-objective optimization problems.

Advantages of ACO

- **Dynamic Rerouting**: ACO can dynamically reroute through the shortest path if a node is damaged or broken.
- **Rapid Discovery of Solutions**: Positive feedback leads to fast identification of good solutions.
- **Distributed Computation**: Inherent parallelism with stochasticity avoids premature convergence.
- **Greedy Heuristic**: Helps find acceptable solutions early in the search.
- **Collective Interaction**: The interaction of many agents helps solve complex problems.

Disadvantages of ACO

- **Slower Convergence**: Sometimes slower than other metaheuristics, especially on large problems.
- **Lack of Central Coordination**: The absence of central coordination can sometimes lead to inefficiencies.

How ACO Works

- **Constructive Nature**: Ants are stimulus-response agents that stochastically and iteratively construct a solution to a graph problem.
- **Ant Movement**: Each ant moves on a graph, choosing where to go based on pheromone strength and a problem-dependent heuristic (typically, distance).
- **Probabilistic Transition Rule**: Ants use a probabilistic transition rule to favour edges with high pheromone and short distances.
- **Path Memory**: Each ant keeps a "path memory" (tabu moves) to prevent revisiting the same node twice.
- **Candidate Solutions**: The total path of an ant represents a specific candidate solution.
- **Pheromone Deposition**: When an ant completes a solution, it lays pheromone on its path, proportional to the quality of the solution. Shorter paths receive more pheromone.
- **Pheromone Evaporation**: Pheromone levels decrease over time.

Example: 4-City TSP

This example demonstrates a simplified version of ACO applied to the Traveling Salesman Problem.

- Initialisation:
  - Random pheromone levels are scattered on edges.
  - An ant is placed at a random node.
- Ant Movement:
  - Ant decides where to go from that node based on probabilities calculated from pheromone strengths and next-hop distances.
  - The ant records its path memory to avoid backtracking.
- Completion and Update:
  - The ant completes its path, forming a tour.
  - Pheromone on the path is increased based on the fitness (total length) of that path.
  - Pheromone on all edges is decreased a little to model decay of trail strength over time.
- The process repeats with another ant in a random position.

Algorithmic Details: Transition Rule

At each iteration, each ant incrementally constructs a path (a solution).

- At each step t, each ant k samples a random number q in.
- If q is smaller than a threshold q_0, the ant greedily chooses the edge with the largest amount of pheromone τ or the shortest length η.
- Otherwise, it uses a probabilistic transition rule:

$$p_{ij}^k(t) = \frac{[\tau_{ij}(t)]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in N_i^k} [\tau_{il}(t)]^\alpha [\eta_{il}]^\beta}$$

  where:
  - $\tau_{ij}(t)$ is the pheromone level on edge i-j at time t.
  - η_{ij} is the heuristic information of the distance of the edge i-j.
  - N_i^k is the set of nodes that can be visited from node i.

  - α and β influence the algorithm's behavior; α controls the importance of the pheromone and β controls the importance of the heuristic information. If $\alpha = 0$, the decision is based solely on the edge length. If $\beta = 0$, the decision is based solely on pheromone.

- Loops are removed once the destination node has been reached (alternatively, ants use path memory).

Algorithmic Details: Pheromone Update

Original Ant System (AS) Pheromone Update:

$$\tau_{ij}(t+1) = (1-\rho) \cdot \tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t)$$

where:

- $\tau_{ij}(t)$ is the pheromone level of each edge, which decreases linearly with time depending on ρ.
- ρ is the pheromone update rate.
- $\Delta\tau_{ij}^{k}(t) = \frac{1}{L_k(t)}$ if ant k uses edge i-j at time t, and 0 otherwise; where $L_k(t)$ is the total length of ant k's path.

Modern ACO Pheromone Update:

- *Local Pheromone Update*: The pheromone level of each edge visited by an ant is decreased by a fraction (1-ρ) of its current level and increased by a fraction ρ of the initial level $\tau(0)$:

$$\tau_{ij}(t+1) = (1-\rho) \cdot \tau_{ij}(t) + \rho \cdot \tau_{ij}(0)$$

  .

- *Global Pheromone Update*: After all ants complete their paths, the length L of the shortest path is found, and only the pheromone levels of the edges belonging to the shortest path are updated:

$$\tau_{ij}(t+1) = (1-\rho) \cdot \tau_{ij}(t) + \rho/L$$

  .

7.3 Applications of Ant Colony Optimization

General Applicability

ACO is naturally applicable to any sequencing problem, or any problem where solutions can be represented as paths in a graph. This requires some way to represent the solutions as paths on a graph.

Example: Shortest Path

Consider finding the shortest path between two nodes:

- Algorithm Parameterization:
  - m ants (e.g. m = 100) are generated and distributed on random nodes.
  - The initial pheromone level is the same for all edges and is inversely proportional to the number of nodes in the graph (n) times the estimated length (L*) of the optimal path: $\tau(0) = \frac{1}{n \cdot L^*}$.
- Typical parameter settings include:
  - Relative importance of pheromone and edge length: $\alpha = 1$ and $\beta = 2$.
  - Exploration threshold: $q_0 = 0.9$.
  - Pheromone update rate: $\rho = 0.1$.

Example: Single Machine Scheduling with Due Dates

A number of jobs must be completed, each with a specific due date and processing time. The goal is to minimize some measure of lateness.

- Each job is considered as a node in a graph.
- Ants find paths representing sequences of jobs.
- No need to return to a starting node; the path is complete when every node is visited.
- An ant starts at a "virtual" start node. The first edge chosen defines the first task to schedule.
- Ants use a transition rule biased by pheromone levels and a heuristic score, each time choosing the next job to schedule.
- Duplicates are avoided using tabu moves.
- The heuristic score for choosing the next job could be the minimization of lateness.

Other Applications

- Drilling wooden boards/iron sheets, where the aim is to minimise time spent drilling holes.
- Capacitated Vehicle Routing Problem with Time Windows and Multiple Trips (CVRPTWMT).
- Neural Architecture Search (using the DEEP-SWARM toolbox).

7.4 Variants of Ant Colony Optimization

Ant Colony Systems (ACS)

ACS differs from the original Ant System in several aspects:

- **Transition Rule**: Biases search towards nodes connected by short links and with a large amount of pheromone, exploiting greedy search.
- **Pheromone Update**: Only the best ants are allowed to update pheromone concentrations on links of the global-best path.
- **Local Pheromone Updates**: Uses adaptive pheromone evaporation (adaptive ρ).
- **Candidate Node Lists**: Lists of candidate nodes, sorted by distance, are used to favour specific nodes during solution construction.

Max-Min AS (MMAS)

MMAS also differs from AS in these aspects:

- **Pheromone Update**: Only the best ants update pheromone concentrations on the global-best path.
- **Pheromone Restriction**: Pheromone intensities are restricted within the interval $[\tau_{min}, \tau_{max}]$.
- **Initial Pheromone Values**: Initially, pheromones are set to the max allowed value τ_{max} to promote exploration.
- **Stagnation Avoidance**: If stagnation is detected, pheromones are reset to τ_{max}.
- **Pheromone Smoothing**: A pheromone smoothing mechanism is used to reduce differences between high and low pheromone concentrations, promoting exploration.

Other Variants

- **Elitist Pheromone Updates**: The best ants add pheromone proportional to their paths' quality to guide search towards good routes.
- **Fast Ant System (FANT)**: Uses only one ant, with the same transition rule as AS (with $\beta = 0$). It uses a different pheromone update rule that does not make use of any evaporation.
- **Antabu**: Adapts AS to include a local search based on Tabu Search, uses a modified update rule.
- **AS-Rank**: Differs from AS in pheromone update rules. Only best ants update pheromone on the best path and it uses elitist updates based on a ranking of the ants.
- **Continuous Ant Colony Optimization (CACO)**: Maps a continuous space problem into a graph search problem.
  - Performs a bi-level search:
    - Local search component to exploit "good" regions.
    - Global search component to explore "bad" regions.
  - Includes local and global search pseudocodes.
  - The transition update rule is preserved.
- **Multi-Objective Ant Colony Optimization**: Has two main methods:
  - Multi-pheromone approach: One pheromone for each objective.
  - Multi-colony approach: One colony for each objective.

7.5 Concluding Remarks

- **ACO is well-suited for solving hard combinatorial (graph-based) optimization problems**.
- Artificial ants implement a randomized construction heuristic with probabilistic decisions.
- Accumulated search experience is incorporated by adapting a pheromone trail.
- **ACO shows strong performance on dynamic problems, like network routing**.
- Modern ACO versions include local search (between local and global update).
- State-of-the-art results are found on very large (¿100k cities) instances of TSP.
- **ACO is able to reroute and regenerate dynamically new solutions by using the ants that eventually will reconstruct new solutions that are adapted to new conditions of the environment**

Chapter 8

Neuro-evolution

8.1 Introduction to Neural Systems

Biological Neural Networks - Generalities

The nervous system is a network of interconnected, specialised cells called **neurons**. These cells process and transmit signals, enabling complex functions. It is important to note that not all organisms possess a nervous system. Simpler life forms, such as paramecia and sponges, can still perform actions like moving, eating, and escaping using chemical reactions in response to environmental stimuli.

Advantages of Nervous Systems

Nervous systems provide significant evolutionary advantages:

- **Selective Transmission of Signals**: They allow the transmission of signals across distant areas, facilitating the evolution of more complex bodies. There is a co-evolution between the body and the brain, as more complex bodies require more computational power from the brain.
- **Complex Adaptation**: Nervous systems enable better adaptation to changing environments by processing information from sensors and allowing for more flexible responses.

It has been observed that many elements of nervous systems and neuron behaviour are similar across animal species. This suggests that a common ancestor developed a nervous system from which several species then evolved, maintaining core functional elements. The complexity of an organism is often associated with the number of interconnections rather than simply the number of neurons. More connections facilitate more efficient information exchange.

Artificial Neural Networks - Models

Two Main Neuron Communication Models

Two primary computational models are used to represent neural networks:

1. **Spiking Neurons**: These models, rooted in computational neuroscience, consider the timing of neuron firing. Each neuron is modelled as a dynamical system with time-variant input/output signals. Activation depends on the time at which a signal is sent or received. These models are more biologically plausible, but computationally expensive.
2. **McCulloch-Pitts**: This model forms the basis of connectionism, focusing on the rate at which neurons fire. Signal strength is key, without time-variant components. If the sum of input signals surpasses an activation threshold, the neuron fires. The time aspect is removed and the output depends on the signal strength. This model is computationally less expensive, forming the basis of most modern AI applications.

Most modern AI is based on the McCulloch-Pitts model, using signal strength without time-variant components.

General Structure of ANNs

Artificial Neural Networks (ANNs) are composed of interconnected units, often arranged in layers. These networks communicate with an external environment through input and output units. Any other units are called internal or hidden units. These units are connected by connections, called **synapses**, which have associated weights that multiply the incoming signals. Each node computes the weighted sum of its inputs and applies an activation function Φ to generate an output signal.

Bias Node

A bias node provides a fixed output, such as -1, which is modulated by a weight. This adaptable threshold allows the network to be a universal approximator.

Activation Functions

Activation functions can be linear or non-linear and determine the output of a neuron given a weighted sum of inputs. Common examples include:

- Identity: A simple linear function.
- Step: Introduces a discontinuity at zero.
- Sigmoid: Continuous, non-linear, monotonic, bounded, and asymptotic. Sigmoid functions are smooth and their derivatives are simplified.
- ReLU (Rectified Linear Unit): Often used in deep neural networks.

Sigmoid functions are commonly used due to their properties.

Different Architectures

Different ANN architectures exist based on how information flows through the network:

1. **Feed-Forward Neural Networks (FFNN)**: Information flows in one direction, from inputs to outputs.
    - **Perceptron**: The simplest model with only input and output nodes, and no hidden layers.
    - **Multi-Layer Perceptron (MLP)**: Similar to a perceptron but with one or more hidden layers.
2. **Recurrent Neural Networks (RNN)**: Information flows both ways, creating a form of memory. Recurrent connections go from forward to backward layers (outputs to inputs). Different types include auto-associative, Hopfield, Elman and self-organizing maps.

How ANNs Work

Each node divides the input space into two regions, one where the weighted sum of the inputs (A) is greater than or equal to zero (A¿=0) and one where A ¡ 0. The separation line is defined by the synaptic weights: $w_1x_1 + w_2x_2 - \vartheta = 0 \rightarrow x_2 = \vartheta/w_2 - (w_1/w_2)x_1$. By combining neurons, we create intersections that separate regions in the input space, allowing for classification.

Single vs Multi-Layer Perceptron

- **Single-Layer Perceptron**: Can only solve linearly separable problems.
- **Multi-Layer Perceptron (MLP)**: Can solve non-linearly separable problems through hidden nodes that remap the input space into a linearly separable space. For example, the XOR problem, a non-linear problem, requires hidden layers to be solved.

Hidden layers are crucial for solving non-linear real-world problems.

Applications of ANNs

ANNs can be used for a wide range of applications:

- **Pattern Recognition**: Such as handwritten text recognition (OCR), voice, and anomaly detection.
- **Content Generation**: For instance, creating synthetic reproductions of artistic images or voice.
- **Regression**: Used for time-series modelling and forecasting, and function approximation.
- **Classification**: Including biomedical data analysis and image recognition.
- **Control**: Used as a black-box control system for robots and industrial plants.
- **Self-driving cars**: For image detection, recognition, and driving.
- **Network efficiency**: For adaptive protocols based on traffic modelling and forecasting.
- **Cybersecurity**: Used for malware and intrusion detection.
- **Marketing**: For customer profiling and advertising.
- **Fin-tech**: Used for trading, stock trend forecasting.

8.2 ANN Training Algorithms

Types of Learning

Training algorithms can be categorized into three main types:

1. **Supervised Learning**: Correct outputs are known, used in regression and classification. The goal is to minimize the error, such as least squares error (LSE). The main algorithm is **backpropagation** (backprop). Backpropagation adjusts weights based on the error between the target output and the actual output of the network.
2. **Unsupervised Learning**: Correct outputs are not known. The goal is to discover correlations in data, perform dimensionality reduction/compression (e.g., auto-encoders, PCA), and feature extraction. The main algorithm is **Hebbian learning**, which adjusts weights based on the correlation of node outputs.
3. **Sequential Decision Tasks**: Correct outputs are not known, and the goal is to find a mapping from states to actions. The main algorithm is **reinforcement learning**, where an agent learns optimal actions based on rewards.

Preventing Overfitting

Overfitting occurs when a network performs well on training data but poorly on new data. Too many weights can cause the network to overfit the training data. To avoid this, the available data is typically divided into:

- **Training Set**: Used for weight updates.
- **Validation Set**: Used for error monitoring. Training should be stopped when error on validation set begins to grow.

Other techniques like cross-validation are used to further avoid overfitting.

Training by Back-Propagation

Backpropagation is a supervised learning algorithm used to adjust weights in a neural network based on errors observed from a known set of labelled inputs. The process involves:

1. **Inject an Entry**: Input is fed into the system.
2. **Compute the intermediate h**: Calculate the output of each hidden node, denoted as h, given by h = f(W*x) where W is the matrix of weights and x is the vector of the inputs.
3. **Compute the output o**: Calculate the output of each output node, denoted as o, given by o = f(Z*h) where Z is the matrix of weights between hidden and output nodes.
4. **Compute the error output**: This is done by using calculus, denoted as δ_{output} = f'(Zh) * (t-o), where t is the target output.
5. **Adjust Z on the basis of the error**: The weights between hidden and output layers are updated using Z(t+1) = Z(t) + $\eta\delta_{output}$ h, where η is the learning rate.
6. **Compute the error on the hidden layer**: This is done by using calculus, denoted as δ_{hidden} = f'(Wx) * (Zδ_{output}).
7. **Adjust W on the basis of this error**: The weights between input and hidden layers are updated using W(t+1) = W(t) + $\eta\delta_{hidden}$ x.

Backpropagation works recursively from output to input layers to adjust weights until a minimum error is achieved. Modern optimizers are typically based on this concept.

8.3 Neuro-evolution

Training by Neuro-evolution

Neuroevolution uses evolutionary computation to construct neural networks, offering a way to find non-linear mappings from inputs to outputs. It is often used in evolutionary robotics, where sensor inputs are directly mapped to actions. Neuroevolution can handle large or continuous state and output spaces, which is often difficult for backpropagation. It is usually applied to supervised learning (error function is used as fitness), but is also used in agent-based simulations and artificial life (Alife) studies.

Basic Neuro-evolution

The genotype in neuroevolution can encode several aspects of the neural network:

1. **Weights**: With a predefined network topology, each weight is encoded in a separate gene. The genotype is typically of fixed length.
2. **Topology**: A variable-length genotype encodes the presence and type of neurons and their connectivity.
3. **Topology and Weights**: A combination of the above two cases.
4. **Learning rules**: Fixed or variable length genotype encodes learning rules rather than weights.

Evolution of Weights - Generalities

A population of ANNs with random weights is evolved with an evolutionary algorithm. The ANN topology is decided in advance and fixed. Each weight is represented by one or more genes. The fitness function can be error, as in back-propagation, or a higher-level consequence of the network output, such as the behaviour of a robot. The initial weights are genetically encoded and learning can happen at each generation, with fitness measured after the training or lifetime learning. Usually, the trained weights are not written back into the genome to avoid Lamarckian learning.

Evolution of Weights - Challenges

- **Premature Convergence**: Loss of diversity and stagnation in progress.
- **Large Networks**: Too many parameters to optimize simultaneously.
- **"Competing Conventions"**: Different network representations can have the same functionality, which makes crossover problematic. Crossover of weights between the same links can result in crossing of weights that are not related to the same part of the network.

Evolution of Weights - Cooperative Synapse Neuroevolution (CoSyNE)

CoSyNE extends the concept of ESP to synapses rather than neurons. Each subpopulation keeps candidate values for a specific weight and, at each step, the algorithm shuffles the elements to create candidate networks, using one weight from each subpopulation. This approach allows the optimization of weights for specific roles in the network. Selection and reproduction occur within each subpopulation. CoSyNE is a form of cooperative co-evolution where each neuron is optimized to have a compatible role.

Evolution of Topology - Analog Genetic Encoding (AGE)

AGE allows the representation and evolution of arbitrary topologies of analog networks, including neural networks. This involves a variable-length genotype with terminal tokens and non-coding genomes interspersed throughout.

Evolution of Topology - Weight Agnostic Neural Networks

A weight agnostic approach to neuro-evolution proposes that certain tasks can be solved by neural networks evolved only with respect to their topology, regardless of the weights used. The networks are tested with a fixed range of weights applied randomly, and networks are ranked based on the average performance. This approach focuses on the effect of topology, similar to some emerging ideas in neuroscience.

Evolution of Topology and Weights

Algorithms in this class are called TWEANNs, topology and weight evolving artificial neural networks. This approach faces several challenges:

- **Initial population**: Random initialisation can result in many inefficient networks.
- **High-Dimensional Search Space**: Allowing for topological evolution can lead to unnecessary complexity.
- **Competing conventions**: More complex as the topology itself also evolves.
- **Loss of innovative structures**: More complex networks may not perform well initially compared to simpler networks and there is a risk that they are lost from the population.

Evolution of Learning Rules

Learning rules describe how weights are updated at runtime, and can be represented by polynomial expressions. Instead of optimizing weights, the constants in the learning rule expression are encoded and optimized. This is used in studying synaptic plasticity, allowing analysis of the evolution of learning mechanisms in living organisms.

8.4 Advanced Neuro-evolution

Neural Evolution of Augmenting Topologies (NEAT)

NEAT is a successful TWEANN algorithm, which addresses initialization problems by starting with minimal structures. The search starts in a space associated with minimal topologies and transitions into high-dimensional space as complexity increases. It protects complex networks from competing with simpler networks.

Advanced Genetic Encoding

NEAT utilizes an advanced genetic encoding where each gene encodes a connection, including input and output node IDs, a weight, an enabled flag and a historical marking or innovation marking.

Historical Markings

Historical markings track when a topological feature, such as a connection, appeared during the evolutionary process. This number allows matching of networks with different topologies and allows for homologous recombination.

Crossover

Crossover is performed by aligning genotypes based on historical markings and then randomly selecting genes from either parent to form offspring, which might change the weights, but keeps the semantics of the network intact.

Speciation

NEAT divides the population into species based on similarity, grouping organisms by network architecture and historical markings. Mating happens within species, allowing promising topologies the chance to be optimized. Fitness sharing prevents a single species from dominating the population, maintaining diversity. Speciation is based on clustering methods that consider excess genes, disjoint genes and average weight differences in shared genes.

Example Applications of NEAT

NEAT has been used to create AI that plays games and is also applicable to complex real world problems:

- **AI playing car racing games**.
- **AI playing Flappy Bird**.

Compositional Pattern Producing Networks (CPPNs)

CPPNs act as indirect encoding for other objects. They have different activation functions which possess desirable properties. They generate properties observed in biological systems such as symmetry, repetition and variation. CPPNs can encode 2D images and 3D robot morphologies. CPPNs act as a function of geometry and compose mathematical functions to generate properties.

HyperNEAT

HyperNEAT uses CPPNs to encode Neural Network connection strengths. By compactly encoding connectivity patterns, HyperNEAT has been shown to evolve networks with several million connections. It applies NEAT to the CPPN rather than to the original networks. HyperNEAT has been used successfully to evolve neural networks for various tasks, including robot locomotion. It optimises a smaller CPPN to provide weights to a larger neural network.

Compressed Network Search

Compressed Network Search evolves networks with over 1,000,000 weights by transforming genotypes using a Fourier transformation. This compression technique allows search to be performed in the Fourier space rather than the original space.

Deep Learning and Network Architecture Search (NAS)

Network Architecture Search (NAS) is an active research area in Deep Learning with many papers based on Evolutionary Computation. Approaches based on regularised evolution are used to minimize complexity while optimising performance. Techniques, such as NEAT and HyperNEAT, that were developed before the deep learning era, are now being used for optimizing deep networks.

Chapter 9

Swarm and Evolutionary Robotics

Swarm robotics and **evolutionary robotics** are areas of research that apply principles observed in nature to robotics. The goal is to create systems that display desirable properties found in natural swarms. This lecture will cover some interesting applications in these areas.

9.1 Swarm Robotics

Generalities

Swarm Robotics is the robotic (embodied) implementation of emergent swarm behaviours observed in nature. The aim is to design multi-agent robotic systems characterized by:

- **Robustness**: Ability to adapt to environmental changes
- **Scalability**: System capabilities increase with the number of robots.
- **Versatility/Flexibility**: Ability to adapt to different tasks.
- **Low Cost**: Simpler individual units reduce overall system cost.

The main idea is to decentralize complexity by using simple robots.

Sources of Inspiration

Swarm intelligence is inspired by behaviours seen in nature. Examples include:

- Flocking and coordinated movement
- Foraging via stigmergy, like ant colony optimization
- Creation of bridges and complex structures

Ants and other social insects provide valuable examples of physical cooperation and task solving. Examples of tasks solved by ants include:

- Passing a gap
- Nest building
- Grouped falling
- Plugging potholes in a trail

These behaviours are desirable to replicate in robotics to create systems that are spontaneous and emergent.

Example 1: Coordinated Exploration

Pheromone Robotics: One approach uses artificial pheromones.

- Main features: Gradient via hop counts, shortest path, pheromone diffusion/evaporation
- Example: Payton et al. (2005) used infrared signals to simulate pheromone trails

Light-Based Repellent Pheromone:

- Hunt et al. (2019) used a light-based "repellent" pheromone with 400 Kilobots.
- Surprising result: At high robot densities, stigmergy performed slightly worse than random walk.

This result indicates that random movement can be more efficient for exploration in certain scenarios.

Example 2: Chaining

Chains in prey retrieval/division of labour:

- Mondada et al. (2005), Nouyan et al. (2009)
- Main features: Limited sensing range, signaling of colours (directional chains)
- Robots create chain-like structures to perform complex tasks.

Example 3: Coordinated Box Pushing

- A group of robots push objects together. (Kube and Zhang, 1993; Kube and Bonabeau, 2000)
- Main features: Task requires cooperation, no explicit communication, behaviour-based approach, ant-inspired stagnation recovery mechanism.
- Cooperation emerges from observing other robots' behaviours.
- Robots work together to push objects towards a desired area.

Example 4: Cooperative Manipulation

- Mobile units assemble into connected entities that are larger and stronger than any individual unit. (e.g., Mondada et al., 2005; Gross et al., 2006)
- Robots create chains or other formations to overcome obstacles or perform tasks too difficult for a single robot.

Example 5: Artificial Stigmergy for Autonomous Building Operations

- Deterministic (add cell to corner area if 2 or 3 adjacent walls are present) vs probabilistic rules (add cell with a given prob. depending on the no. of adjacent walls). (Camazine et al., 2001)
- Coordination is based on the stimulus of occupied cells nearby, and response of adding a wall when two or three adjacent walls are present.
- This system uses simple rules to build complex structures.

9.2 Reconfigurable Robotics

Generalities

A modular robot is composed of several (usually identical) components that can be re-organized to create morphologies suitable for different tasks. **Biological Inspiration**:

- Groups of cells (cellular automata)
- Groups of individuals (swarm intelligence)

Modular robots can change their morphology to adapt to different tasks.

Types of Reconfigurable Robots

- **Chain-type**: Modules form chain-like structures.
- **Lattice-type**: Modules form a 2D or 3D matrix.
- **Hybrid type**: Combination of chain and lattice structures.
- **Other types**: Robots that do not fit into the above categories.

Chain Type

Example: CONRO (Castano et al., 2000)

- Main features: Fully self-contained, pin-hole connector (+latch), infrared-based guidance, docking relatively complex, good mobility.
- Control can cope with sudden changes in the robot's morphology.

Example: PolyBot (Yim et al., 2002)

- Main features: 1-DOF module; MPC555 processor; externally powered.
- Able to create a closed chain and change the robot's architecture.

Lattice Type

Example: A-TRON (Maersk McKinney Moller Institute, University of Southern Denmark)

- Main features: Two half-spheres; 4 male and 4 female connectors; self-docking relatively simple; self-reconfiguration can require many steps.
- These 3D structures can create complex morphological shapes.

Hybrid Type (Chain+Lattice)

Example: M-TRON (Murata et al., 2002)

- Main features: Magnets or actuated mechanical hooks, driven by cellular automata rules.
- Modules use magnetic hooks to attach to one another, with magnetic actuators for movement.

Other Types

- **Claytronics** (Goldstein et al., 2005): Relative displacement by means of electro-magnet rings.
- **PPT** (Klavins et al., 2005): Stochastic reconfiguration of passively moving parts.
- **Anatomy-based** (Christensen et al., 2008): Anatomic structures with differentiated elements.

Modular Soft Robots

- **Simulated voxels** (Cheney et al., 2013): Evolution of locomotion.
  - Uses simulation to evolve different kinds of tissues such as bone, muscles and tendons
  - Different morphologies emerge using evolutionary computation.
- **Air-filled silicon membranes** (Kriegman et al., 2019): Automatic shapeshifting in damaged robots.
  - Soft robots can automatically reconfigure their shape.
  - Soft tissues allow for flexibility and compliance.
- **Tensegrity modules** (Zardini et al. 2021): Diversity-based evolution of locomotion.
  - Modules are composed of elastomers and rigid sticks with linear actuators.
  - Both morphology and controller can be co-evolved.
- Hardware validation (sim-to-real analysis).

9.3 Evolutionary Robotics

Generalities

Evolutionary Robotics (ER) is the automatic generation of control systems and/or morphologies of autonomous robots. It is based on a process of Artificial Evolution without human intervention. **Motivations**:

- Difficult to design autonomous systems using a purely top-down engineering process, due to the complex and hard to predict interactions between robots and their environment.
- ER can be used as a synthetic (as opposed to analytic) approach to study the mechanisms of adaptive behaviour in machines and animals.

The engineer defines the control components and selection criteria and artificial evolution discovers the most suitable combinations while robots interact with the environment.

Representing Behaviour in ER

Behaviour in ER is a mapping between stimuli from the environment and the response of the agent. Examples of stimuli include:

- smell
- vision
- touch

Examples of behaviour include:

- actions such as moving to food
- following a buddy

Reminder - Evolution of Neural Networks

The genotype in ER can encode:

- **Weights**: Pre-defined network topology, each weight encoded in separate genes, fixed-length genotype.
- **Topology**: Variable-length genotype encodes presence/type of neurons and their connectivity.
- **Topology and Weights**: Combination of the first two cases.
- **Learning rules**: Fixed or variable-length genotype encodes learning rules.

Neural networks can be used as a primitive approximation of the brain.

How to Evolve Behaviour in ER

The process of evolving behaviour includes:

1. **Population Initialization**: Initializing the population of robots and controllers.
2. **Robot Task Execution**: Robots perform a task in the environment.
3. **Fitness Evaluation**: Measuring the performance of each robot.
4. **Fitness-based Ranking**: Ranking robots based on fitness.
5. **Selection**: Selecting the fittest individuals to reproduce.
6. **Reproduction (mutation)**: Creating new individuals through mutation.

This process is repeated for a certain number of generations until a satisfactory solution is reached.

Example 1: Collision-Free Navigation

- Experimental setup: A single 2-wheeled robot is placed in an arena with obstacles, and it has to explore it as much as possible (keeping on moving) without any collision.
- Population of neural networks is evolved on a computer and transferred into robots at each generation for evaluation.
- Experimental results: Evolved robots tend to have a preferential direction/speed.

Example 2: Homing for Battery Charge

- Experimental setup: The robot has a battery that lasts 20 seconds, and there is a battery charger in the arena. The robot has to reach it before its battery dies out.
- Experimental results: After 240 generations, the EA finds a robot that is capable of moving around and going to recharge 2 seconds before its battery is completely discharged.

Example 3: Car Driving

- A car can be trained on a set of circuits by means of neural networks.
- The total travel distance for each race is used to calculate the fitness.
- By doing this in simulation we can optimise neural networks to perform self-driving

Example 4: The "Golem" Project

- Lipson and Pollack (2000) added the physical construction of the creatures by using a 3D thermoplastic printer and extensible bars.
- Evolution takes place in simulation.
- Fitness = distance covered by the robot.
- Selected individuals are built by:
  1. Printing the bars
  2. Fitting joints and motors
  3. Downloading the ANN into an embedded controller

A (Non-Comprehensive) List of Projects on Swarm/Evolutionary Robotics

A lot of research is ongoing in this field. Examples include:

- **ARE** (Autonomous Robot Evolution): Investigates the physical evolution of artifacts and recycling material.
- **PHOENIX**: Developed methods for evolving small sensor agents for exploring inaccessible environments.
- **DEMIURGE**: Developing methods to design all aspects of a robot swarm (sw/hw).
- **EVOBODY**: Explored the potential of physically embodied evolutionary systems.
- **SYMBRION**: Investigated modular robotic organisms that emerge by self-aggregating independent robotic units.
- **NEW TIES**: Investigated artificial societies subject to adaptation through evolution, individual learning, and social learning.
- **REPLICATOR**: Focused on the development of an advanced robotic system consisting of a super-large-scale swarm of small autonomous mobile micro-robots capable of self-assembling into large artificial organisms.

- **BROS**: Combines blockchain technology and swarm robotics systems to generate robotic models where robot interactions are encapsulated in transactions on the blockchain.
- **SWARM-BOTS**: Designed and implemented self-organizing and self-assembling artifacts called swarm-bots.
- **COCORO**: Developed autonomous robots that interact and exchange information to become aware of their environment.
- **I-SWARM**: Aimed to develop mass-production technology of micro-robots, for swarms of up to 1000 robots.
- **ROBOSWARM**: Aimed to develop an open knowledge environment for self-configurable, low-cost, and robust robot swarms.
- **E-SWARM**: Aimed to develop a rigorous engineering methodology for the design and implementation of artificial swarm intelligence systems.
- **CHOBOTIX**: Aimed to develop µm-sized chemical swarm robots ("chobots") that can move in their environment, and selectively exchange molecules in response to changes in temperature or concentration.

9.4 Conclusion

Swarm and evolutionary robotics are active areas of research with promising results. They offer unique approaches to designing intelligent systems by drawing inspiration from nature. The combination of both swarm and evolutionary principles is likely to lead to future practical applications.

Chapter 10

Competitive and Cooperative Co-Evolution

This chapter explores co-evolution, a bio-inspired paradigm in artificial intelligence, focusing on two main types: competitive and cooperative co-evolution.

10.1 Competitive Co-evolution

Generalities

Competitive co-evolution occurs when two or more different species co-evolve against each other. Classic examples of this include**prey-predator** and**host-parasite** relationships. The key aspect is that the fitness of each species is dependent on the fitness of the opposing species.

Potential Advantages

- **Increased Adaptivity**: Competitive co-evolution can lead to an "evolutionary arms race", driving species to become increasingly better at overcoming their opponents, thus increasing their adaptivity.
- **Incremental Improvement**: Better solutions can emerge incrementally as each species strives to outcompete the other, leading to increasingly complex and effective behaviours.
- **Reduced Human Bias**: The need for a human-designed fitness function is reduced, allowing for more autonomous systems where better solutions emerge spontaneously. The system itself determines what constitutes a 'good' solution by how well it overcomes the other competitor.
- **Dynamic Fitness Landscape**: The continuously changing fitness landscape can help prevent stagnation in local minima, making it harder for the evolutionary process to get stuck.

Formal Models

Formal models of competitive co-evolution often use the**Lotka-Volterra equations**, which are a set of differential equations that describe the variation in population size of two competing species, particularly in a prey-predator scenario. The equations are as follows:

$$\frac{dN_1}{dt} = N_1(r_1 - b_1 N_2)$$
$$\frac{dN_2}{dt} = N_2(-r_2 + b_2 N_1)$$

Where:

- N_1 and N_2 are the two populations (e.g., prey and predators)
- r_1 is the increment rate of prey without predators

- r_2 is the death rate of predators without prey
- b_1 is the death rate of prey caused by predators
- b_2 is the ability of predators to catch prey

It's important to note that in biology, fitness is related to the variation in population size, not the behavioural performance, which is difficult to define and measure.

Computational Models

Formal models often assume constant behavioural performance across generations, which is not suitable for predicting how competitive co-evolution generates better individuals. Computational models aim to improve this and generate increasingly better solutions.

Hillis's Sorting Algorithm Example

Hillis (1990) demonstrated that co-evolution could produce more efficient sorting programs than evolution alone, surpassing hand-designed algorithms. This involved co-evolving sorting programs against testing programs designed to produce difficult-to-sort lists. The sorting program's fitness is the quality of sorting, and the testing program's fitness is one minus the quality.

Strategy Recycling

A common issue in competitive co-evolution is**strategy recycling**, where the same set of solutions is discovered repeatedly across generations. This can lead to stagnation in relatively simple solutions. Possible causes of strategy recycling are:

- **Lack of "generational memory"**: Individuals only compete against those of the current generation, ignoring past, potentially better, solutions.
- **Restricted possibility for variation**: The need to overcome the other species can limit the variety of behaviours and solutions.
- **Small genetic diversity**: Individuals become too similar because they are evolving to overcome a similar challenge.

Dynamic Fitness Landscape

Competitive co-evolution involves a dynamic fitness landscape, where the fitness of one species depends on the other, causing the landscape to change over time. In contrast to single-species evolution, where the fitness landscape is static, this makes evolutionary progress harder to track because fitness trends can oscillate. This phenomenon is known as the "**Red Queen hypothesis**," from *Through the Looking Glass*, where the Queen says, "it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that". This means that the fitness landscape itself is constantly changing.

Measuring Evolutionary Progress

Due to the dynamic fitness landscape, evolutionary progress in competitive co-evolution cannot be measured solely by fitness trends of individual species. Alternative measures include:

- **CIAO Graphs**: These graphs (Current Individual vs. Ancestral Opponent) represent the outcome of tournaments between current individuals and best opponents from previous generations. Ideal continuous progress would show a black lower diagonal and white upper diagonal.
- **Master Tournaments**: These graphs plot the average outcome of tournaments between the current individual and all previous best opponents. Ideal continuous progress is indicated by continuous growth.

Examples in Evolutionary Robotics

Robotic Prey-Predator

An example of competitive co-evolution is a robotic prey-predator scenario. Two robots, one as predator and one as prey, compete in an arena.

- **Goal**: The predator must catch the prey, and the prey must avoid the predator.
- **Setup**: The prey has proximity sensors and is faster, while the predator has proximity and vision sensors but is slower.
- **Fitness**: Prey fitness is the time to contact (maximized), and predator fitness is one minus the time to contact (maximized).
- **Results**: Average and best fitness graphs display oscillations. Master Tournaments show progress only in the initial generations, followed by flat or decreasing graphs, indicating recycling dynamics. CIAO graphs were also not very effective in revealing progress. However, co-evolved individuals often display highly adapted strategies.

Fitness Function Components

When designing fitness functions for evolving autonomous systems, it is important to consider:

- **Internal vs. External**: Whether the fitness is based on information directly available to the agent's sensors or not.
- **Functional vs. Behavioral**: Whether the fitness defines specific functional requirements or behavioral outcomes.
- **Implicit vs. Explicit**: Depending on how many variables and constraints are included in the fitness function.

Experiments show that co-evolution might work better with internal, implicit, and behavioral fitness functions.

Hall of Fame

To avoid recycling, Rosin and Belew (1997) proposed the "Hall of Fame," where new individuals are tested against all best individuals found so far. It can be sufficient to test new individuals against a limited sample (e.g. 10) randomly selected from the Hall of Fame, which produces continuous incremental progress, as shown by CIAO and Master Tournaments. However, in the long run, the Hall of Fame can become similar to single-agent evolution due to a static pool of opponents, reducing creativity of new solutions.

Adaptation

Adding Hebbian learning to the neural networks in a prey-predator system can drastically change co-evolutionary dynamics. In one experiment, predators consistently won after 20 generations and always chose adaptation, while prey often chose random synapses due to poor sensors, showing how adaptation doesn't always help in co-evolutionary scenarios.

Man-Machine Co-evolution

In man-machine co-evolution, computer programs are co-evolved against human players, as seen in a simplified version of the game Tron, where computer programs are represented as trees and evolved using genetic programming. Computer programs become increasingly better, while humans learn across trials, rather than evolving in a biological sense.

Agents vs Environments

The Paired Open-Ended Trailblazer (POET) system co-evolves a bipedal walker against an environment with obstacles to create a learning curriculum. Environments are filtered out if they are too simple or too hard, and agents survive if they achieve a minimum performance. This approach allows for the gradual complexification of the environment, which helps to create more robust agents.

10.2 Cooperative Co-evolution

Generalities

Cooperative co-evolution involves two or more species that co-evolve, where one species benefits another, but not necessarily vice-versa (interspecies cooperation), or individuals within a species co-evolve with some benefiting others in the same species (intraspecies cooperation).

Types of Cooperation

- **Simple/Reciprocal Cooperation**: Helping individuals or species receive some advantage in return, bigger than or equal to the cost of helping.
- **Altruistic Cooperation**: Helping individuals or species incur a cost greater than any advantage they gain (if any), such as warrior ants that die to save a colony. This is harder to explain with a pure "gene-centric" approach.

Formal Models of Altruism

Kin Selection/Inclusive Fitness

Hamilton's rule (1964) states that altruism can evolve when the genetic relatedness between individuals is high, formalized as $r > \frac{c}{b}$, where r is the genetic relatedness, c is the cost of altruism, and b is the benefit.

Multi-level (Group) Selection

This theory suggests that selection acts at the group level as well as the individual level; this approach does not require genetic relatedness. However, it has received criticism due to slower/less likely mutations at the group level.

Computational Models

In artificial evolution, there is no need to explicitly compute individual fitness or genetic relatedness; instead, one can focus on group fitness, if appropriate. Various algorithms/strategies can be used:

- **Homogeneous Groups**: All individuals are the same
- **Heterogeneous Groups**: Individuals are different
- **Team Selection**: Selection occurs at the team or group level
- **Individual Selection**: Selection occurs at the individual level

Applications

Robotic Foraging Task

A robotic foraging task demonstrates cooperative behavior. In this experiment, mobile robots were evolved to collect objects in different scenarios:

- **Individual**: One fitness point per object to the foraging robot.
- **Cooperative**: One fitness point to all robots for each object (two robots needed to push an object).
- **Altruistic**: One fitness point to all robots for each large object, and one point to the individual robot for each small object.

Results showed that cooperative individuals were genetically related and that altruism can lead to slightly lower foraging efficiency because some agents become free riders.

Robotic Foraging Task with Uncertainty

In another foraging task, robots had to distinguish between food and poison sources. Genetically related individuals obtained the highest performance. Robots communicated with each other by emitting light when they found the food.

Co-evolution of Dispersal and Altruism

This experiment studied the relationship between dispersal (movement) and altruism among robots in a structured environment. Dispersal can decrease local competition and relatedness, affecting altruism. Models of selfish and altruistic behaviors were tested, with different selection levels (global vs. local), finding that altruism emerges when the benefits are high enough.

Summary

- Competitive co-evolution can create more efficient and novel systems but can be hard to direct.
- Generational memory is useful for preventing recycling.
- Altruistic cooperation evolves if individuals are genetically related or there is group-level selection.
- In robotics, altruism can emerge naturally through artificial evolution, with individuals sharing resources, performing cooperative tasks, and communicating information.
- These findings can be used to understand biological systems and to implement swarms of autonomously cooperating robots.

Chapter 11

Genetic Programming

11.1 Genetic Programming

Genetic programming is a type of evolutionary algorithm designed to evolve computer programs automatically. This is achieved by representing programs as tree structures and using principles of natural selection to find optimal solutions.

The Challenge

- Virtually all problems in computer science can be framed as a search for a computer program that solves a specific problem.
- Alan Turing first proposed the idea of using evolutionary search in the program space in 1948.
- Arthur Samuel, an AI pioneer, asked: "How can computers learn to solve problems without being explicitly programmed?" This question underlies the field of Genetic Programming.
- Genetic programming provides a method for automatically creating a working computer program from a high-level problem statement, also known as automatic programming, program synthesis, or program induction.

11.2 Generalities

- Genetic programming (GP) is a specialization of genetic algorithms (GAs).
- The primary difference between GAs and GP is that GP uses tree-based representations for individuals, whereas GAs use linear structures.
- The original goal of GP was to develop a general framework to evolve computer programs.
- Currently, GP is mostly used for black-box (data-driven) modelling, forecasting, and classification tasks. It serves as an alternative to neural networks.
- Early attempts to evolve computer programs were made by Smith (1980), and the first GP implementations were by Camamer (1985) and Koza (1989) in LISP.
- **John Koza is considered the "founding father" of genetic programming**.
- GP produces human-readable models/programs, unlike neural networks, enabling inspection, modification, and debugging.
- GP allows the reuse of existing subprograms, facilitating scalability and hierarchical expansion.
- GP's weaknesses include its need for large populations and slow convergence, although research is making progress to improve this.

11.3 Why Trees?

- Trees are a universal form capable of representing various structures:
  - Arithmetic expressions (e.g., $y = x * ln(a) + sin(z)/exp(-x) - 3.4$).
  - Boolean expressions (e.g., (x_1 AND NOT x_2) OR (NOT x_1 AND x_2)).
  - Computer programs (including loops, if-then-else structures).

Implications

- Adaptive individuals: the size of individuals is not fixed; it depends on the depth of the tree and its branching factor.
- Domain-specific grammar is needed to accurately reflect the problem and should represent any possible expression.

11.4 Grammar Definition

- A grammar is defined using two sets: the terminal set and the function set.
- **Terminal set (T)**: specifies all variables and constants, which are the leaves of the trees.
- **Function set (F)**: contains all functions that can be applied to the elements of the terminal set, such as arithmetic, Boolean, decision structures like if-then-else, and loops.
- Each function f in F has its own arity (number of arguments).
- **Closure property:** Generally, each function f in F can take any other function g in F as an argument (GP expressions are usually not typed). There is also a variant called strongly typed genetic programming which does not have this closure property.
- Correct expressions generated by the grammar can be defined recursively:
  - Every t in T is a correct expression.
  - $f(e_1, e_2, ..., e_n)$ is a correct expression if and only if:
    1. f is in F
    2. $arity(f) = n$ is correct
    3. $e_1, e_2, ..., e_n$ are correct expressions
- Semantic rules can be added to characterize valid expressions.

11.5 Examples

Boolean Expression

- Goal: Evolve an expression such as (x_1 AND NOT x_2) OR (NOT x_1 AND x_2).
- Function set: {AND, OR, NOT}.
- Terminal set: $\{x_1, x_2\}$, where $x_1, x_2 \in \{0, 1\}$.
- A truth table with target outputs for each combination of x_1 and x_2 is used to train the model.

x_1	x_2	Target Output
0	0	0
0	1	1
1	0	1
1	1	0

Symbolic Regression

- Goal: Evolve an arithmetic expression such as $y = x * ln(a) + \frac{sin(z)}{exp(-x)} - 3.4$.
- Given a set of training samples in $\mathbb{R}^2$: $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$.
- Function set: $\{-, +, *, /, \text{sin, exp, ln}\}$.
- Terminal set: $\{a, x, z, 3.4\}$, with $a, x, z \in \mathbb{R}$.
- A GP solution will find a function $f(x)$ such that $f(x_i) = y_i$ for each i.

Symbolic Regression (Black-Box)

- Goal: Find a function $f(x)$ such that for every i in $\{1, 2, ..., n\}$, $f(x_i) = y_i$, given points in $\mathbb{R}^2$: $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$.
- Function set: $\{-, +, *, /, \text{sin, exp, ln}\}$.
- Terminal set: $\mathbb{R} + \{x\}$.
- Fitness is the sum of squared errors: $\sum_{i=1}^{n}(f(x_i) - y_i)^2$, or a similar metric such as mean squared error (MSE) or root mean squared error (RMSE).

Computer Program

- Goal: Evolve a loop cycle (e.g. an example is provided that increments i from 1 to 20).
- Given an input and a desired program output.
- Example input: $i = 1$; Example output: $i = 20$.
- Function set: {while, ¡, ¿, =}.
- Terminal set: $\{i, 20\}$, with $i \in \mathbb{Z}^+$.

```
i = 1;
while (i < 20) {
  i = i + 1;
}
```

Computer Program (Controller)

- Goal: Evolve a controller for guiding an agent (e.g., a mouse) towards a goal (cheese) in a maze.
- Function set: {If-Way-Ahead-Blocked, While-Not-At-Cheese (root node)}.
- Terminal set: {Move-Forward, Turn-Left, Turn-Right}.
- Example of a possible evolved controller program:

```
While not at the cheese
    If the way ahead is blocked
        Turn right 90 degrees
        Move forward one space
        Move forward one space
        Move forward one space
    Otherwise
        Move forward one space
        Turn right 90 degrees
        Move forward one space
        Move forward one space
        Turn left 90 degrees
```

```
If the way ahead is blocked
    Turn left 90 degrees
Otherwise
     Move forward one space
```

Classification (Bank Credit Scoring)

- Goal: A bank wants to distinguish "good" from "bad" loan applicants.
- A model is needed that matches historical data:

ID	No of Children	Salary	Marital status	Class
ID-1	2	45000	Married	bad
ID-2	0	30000	Single	good
ID-3	1	40000	Divorced	good
...	...	...	...	...

- A possible model: IF (NOC = 2) AND (S > 80000) THEN good ELSE bad.
- Function set: {AND, OR, NOT, ¡, ¿, =}.
- Terminal set: {NOC, S, Married/Single/Divorced, C}, with $C \in \mathbb{Z}^+$.
- This approach allows for interpretable models, unlike black-box methods.
- In general, the goal in classification problems is to identify an if-then rule of the kind: IF [expression] THEN [target output].
- The search space is a set of expressions (phenotypes).
- The fitness of an expression is the percentage of well-classified cases (supervised learning).
- The representation (genotypes) of expressions are parse trees, also called Symbolic-Expressions (S-expressions).

11.6 Fitness Functions

- Fitness functions are usually problem-dependent.
- Fitness calculation requires evaluating the expression/program against a number of test cases to determine if it matches the desired outcome.
  - For Boolean expressions: fitness is the number of correctly predicted target outputs.
  - For arithmetic expressions: fitness is the sum of squared errors with respect to supplied target outputs.
  - For computer programs: fitness is the number of correctly generated target outputs.
- Fitness functions may include a penalty to penalize individuals with undesirable structural properties such as invalid expressions or big tree sizes.

11.7 Differences Between GP and GA

- **Representation:** In GA, chromosomes are linear structures (bit strings, integer strings, real-valued vectors, permutations). In GP, chromosomes are tree-shaped, non-linear structures.
- **Chromosome size:** In GA, the size of the chromosomes is fixed. In GP, trees may vary in depth and width.
- **Use of operators:** In GA, crossover AND mutation are used. In GP, crossover OR mutation is used. Both crossover and mutation depend on user-defined probabilities.
- In GP, crossover is usually highly probable ($p_c > 0.8$), while mutation is quite rare ($p_m < 0.05$).

11.8 Algorithmic Details

Initial Population

- Initial trees are randomly generated within a maximum depth, usually small.
- A root is randomly selected from the function set F.
- Two methods for choosing nodes below the root:
  1. **Full method:** Each branch has depth exactly equal to D_{max}. Nodes at depth $d < D_{max}$ are chosen from the function set F and nodes at depth $d = D_{max}$ are chosen from the terminal set T.
  2. **Grow method:** Each branch has a depth at most equal to D_{max}. Nodes at depth $d < D_{max}$ are chosen from $F \cup T$ and nodes at depth $d = D_{max}$ are chosen from T.
- The full and grow methods are often combined into the "ramped half-and-half method", where each method generates 50% of the initial individuals.

Parent Selection

- Classic GP: fitness-proportionate selection.
- Modern GP implementations use "over-selection" to increase efficiency.
  1. Rank the population by fitness.
  2. Divide into two groups:
     - Group 1: best x% of the population.
     - Group 2: other (100-x)%.
  3. Select 80% of parents from group 1 and 20% from group 2.
  4. x% thresholds are usually set by rules of thumb. For populations of 1000, 2000, 4000, and 8000, x is 32%, 16%, 8% and 4% respectively.

Survivor Selection

- Classic GP uses very large populations (50k-600k individuals) with a purely generational scheme (no survival of parents).
- Modern GP uses smaller populations ($<< 10k$) with steady-state schemes and elitism to preserve the best individuals in the population.

Crossover

- Crossover involves selecting two parent trees, identifying two cut points (usually function nodes), and swapping the subtrees below these points to create two offspring.
- Crossover must check for validity in strongly typed GP.

Mutation

- The tree-based representation allows several possible mutations:
  - Function node mutation.
  - Terminal node mutation.
  - Growth (expansion) mutation.
  - Truncation mutation.
  - Swapping mutation.
  - Gaussian mutation.

  - **Function node mutation**: Replaces a function node with another function with the same arity.
  - **Terminal node mutation**: Replaces a terminal node with another terminal node.
  - **Growth mutation**: Expands the tree by adding a new randomly generated subtree at a chosen point in the existing tree.
  - **Truncation mutation**: Shrinks the tree by removing a subtree and replacing it with a leaf.
  - **Swapping mutation**: Swaps the order or position of subtrees within the same individual.
  - **Gaussian mutation**: Modifies constant values by adding a Gaussian random number.

Stop Conditions

- Similar to other evolutionary algorithms:
  - Number of fitness evaluations.
  - Number of generations.
  - Number of generations without improvement.
  - Elapsed time.
- Additional criterion for symbolic regression problems: number of "hits" reached (where "hit" is if $|f(x_i) - y_i| < \epsilon$, e.g., stop when 5000 hits with $\epsilon = 0.0001$).
- Multiple criteria can be combined (usually in OR).

11.9 Successful Applications

- **Robotics**: automatic generation of gaits for biped robots, hand gestures for humanoid robots, mouth motions for speaking robots, and state machines for behavioral robots.
- **Biology**: transmembrane segment identification for proteins.
- **Quantum Computing**: automatic design of algorithms and communication protocols.
- **Electronics and TLC**: design of analog circuits (topology and RLC parameters), controllers, and antennas.
- **System identification** and automatic discovery of physics laws (e.g., Eureka tool).

11.10 Issues in GP

Bloat

- Bloat is the tendency for GP trees to increase in size over time, which can make the models hard to handle and understand.
- Possible countermeasures:
  - Limit the size of trees in the initial population.
  - Prohibit variation operators (mutation and crossover) that generate "too big" trees.
  - Introduce checks/repair mechanisms in mutations and crossover.
  - Pruning (worst trees replaced by branches pruned from best trees).
  - Penalize the fitness of oversized trees (parsimony pressure).
  - Use ad-hoc operators.

Bloat ("Building Block" Approach)

- Start with simple trees (root node and first children) and let trees grow as needed during evolution.
- Expand trees when:
  - Simplicity no longer accounts for the complexity of the problem.
  - No improvement in fitness is observed.
- Expansion occurs by adding a randomly generated building block (new node) to individuals.
- Expansion occurs at a specified expansion probability (p_e).
- Helps reduce the computational complexity of the evolution process and produces smaller individuals.

Transfer to Reality

- There is a gap between trees for data fitting and trees (programs) that are actually executable.
- Execution can change the environment; therefore, fitness calculation should account for that (e.g., a robot controller).
- Fitness calculations are mostly by simulation, which can be expensive (time-consuming).
- Evolved controllers are often very good despite these issues.

Efficiency

- GP trees are usually very hard to evolve and require a population of thousands of individuals, with slow convergence and expensive fitness computations.
- Possible solutions:
  - Parallel GP (e.g., multi-threaded GP or GP on GPU), with parallelism at the tree or instruction level.
  - Compiled GP (trees are precompiled with compiling optimizations before execution).

11.11 Discussion

What is, in the end, GP?

- The art of evolving computer programs.
- A means to "automagically" program computers.
- A genetic algorithm with another representation.

Chapter 12

Applications and Recent Trends

12.1 Applications of Evolutionary Algorithms

Case Study 1: NASA ST5 Antenna Design by EA

The NASA ST5 mission aimed to measure the effect of solar activity on the Earth's magnetosphere, using three nanosatellites (50 cm each). The challenge was to design an optimal antenna for these satellites to send data to a ground station.

Problem

Designing the antenna manually was particularly difficult, hence an evolutionary algorithm was used.

Experimental/Algorithmic Setup

A tree-based encoding was used, allowing for the development of a 3D structure resembling a tree. The function set included a forward step and rotations along the x, y, and z axes. The terminal set defined the geometrical parameters of the antenna.

- Function Set: $\{f = \text{forward(length, radius)}, r_{x/y/z} = \text{rotate x/y/z}\}$
- Terminal Set: {length, radius, x, y, z}

Each function node had a constraint of a maximum of three branches.

Experimental Results

The fitness of the antenna designs were evaluated in simulation, and the best design was built and tested in an anechoic chamber. The evolutionary algorithm yielded an unconventional design that engineers had not conceived of, demonstrating how these algorithms can produce unexpected solutions.

Case Study 2: Automatic Design of Industrial Controllers by EA

Evolutionary algorithms are also used in the automatic design of industrial controllers, such as PID controllers.

Problem

Given a system, the goal is to design an optimal controller that satisfies user-defined requirements.

Experimental/Algorithmic Setup

Online optimisation is achieved using a hardware-in-the-loop with a steady-state EA. The integral absolute error (IAE) is used as a fitness function to minimize the error between the actual output and the reference point.

Key Takeaway

EAs can optimize controller parameters, even with hardware-in-the-loop, but system stability must be ensured.

Case Study 3: Job Shop Scheduling Problem by EA

The job shop scheduling problem (JSSP) is another area where evolutionary algorithms are applied.

Problem

The goal is to schedule jobs on machines to minimise the total duration, given constraints such as which machines can perform which operations and the order of operations.

- J is a set of jobs.
- O is a set of operations.
- M is a set of machines.
- $Able \subseteq O \times M$ defines which machines can perform which operations.
- $Pre \subseteq O \times O$ defines which operations should precede others.
- $Dur : O \times M \to R$ defines the duration of operation $o \in O$ on machine $m \in M$

Experimental/Algorithmic Setup

Individuals are represented as permutations of operations, decoded into schedules using a decoding procedure that takes the first operation, looks up its machine, and assigns the earliest possible starting time subject to machine occupation and precedence relations.

Fitness Evaluation

The fitness of a permutation is the duration of the corresponding schedule. Genetic algorithms are well-suited for this combinatorial problem, often combined with heuristic techniques.

Key Takeaway

EAs can efficiently find optimal or near-optimal solutions for complex scheduling problems.

Case Study 4: Drift Correction in Electronic Noses by ES

Electronic noses use arrays of gas/chemical sensors to mimic human olfaction.

Problem

Sensors are prone to drift due to changes in sensitivity, environmental factors, and degradation. This drift must be corrected for proper compound classification.

Experimental/Algorithmic Setup

Evolution strategies (ES) are used to tune the parameters of the Pattern Recognition module (PARC). This optimisation corrects drift and improves separation among odor classes.

Key Takeaway

EAs can be used to dynamically correct sensor drift, enhancing the accuracy of pattern recognition systems.

Case Study 5: Evolutionary Electronics

Evolutionary electronics (EE) uses evolutionary techniques to design and optimise electronic circuits.

Possibilities

- Evolution of parameters (sizing) with a fixed topology.
- Evolution of both parameters and circuit topology.
- Placement and routing of devices.

Challenges

Designed circuits must be robust and perform correctly under various conditions, which requires extensive verification. The evolutionary process can yield unconventional circuits that are difficult to understand and verify.

Analog vs Digital Design

- Analog: Functionality is determined by connectivity and circuit parameters, which are mostly continuous. Genetic representations include schematics, configuration bits, and trees.
- Digital: Functionality is determined by connectivity, with few circuit parameters. The search space is discontinuous. Genetic representations include schematics, configuration bits, and truth tables.

Extrinsic vs Intrinsic Design

- Extrinsic: Each circuit is simulated to assess performance. This method is safe but the simulation is an approximation.
- Intrinsic: Each circuit is physically implemented and tested. This method is accurate but has physical constraints and potential for damage.

Examples

Examples include the evolution of an inverter with bipolar transistors and the evolution of a robot controller.

Key Takeaway

EAs can automate and optimise electronic circuit design, but require careful validation of the results.

Case Study 6: Cell Phone Software Validation by EA

EAs can be used to find bugs in software.

Problem

The goal is to find a sequence of operations that causes incorrect behavior in cell phones, such as excessive power consumption in deep sleep. This is a "needle in the haystack" problem.

Motivation

Increased complexity of cell phones, unreliable integration, and fast time-to-market often lead to undiscovered bugs.

Experimental/Algorithmic Setup

The entire system is modeled as a Finite State Machine (FSM), reconstructed from debug log messages. The fitness is based on the number of state transitions, which correlates with power consumption.

Experimental Results

This approach identified two power-related bugs that had been missed by previous testing, as well as an interface bug.

Key Takeaway

EAs are useful in finding corner cases and software bugs that traditional methods may miss.

Case Study 7: Network Protocol Verification in WSN by EA

EAs can be used to verify network protocols in Wireless Sensor Networks (WSN).

Problem

The goal is to quantify the worst-case network behaviour at protocol design time, such as energy consumption and latency.

Limitations of Traditional Testing

Random testing, lab testing, and formal software verification may not always be suitable.

Experimental/Algorithmic Setup

Populations of candidate topologies are simulated and evaluated for network traffic. An evolutionary algorithm is used to iteratively generate network topologies where traffic is maximised.

Experimental Results

Novel topological features were discovered that correlate with high traffic and energy consumption. These features can be used as predictive metrics.

Key Takeaway

EAs can effectively stress-test network protocols and identify critical network topologies.

Algorithmic details

The evolutionary framework uses μGP coupled with the TOSSIM network simulator. Individual representation is an asymmetric matrix of link signal strengths. Individual fitness is determined by the max or sum of network packets generated by nodes. A multi-objective approach is also possible.

Case Studies 8-10

- **Influence Maximisation in Social Networks**: EAs are used to identify the most influential nodes in social networks. This is done by modelling the propagation of information from one node to another. EAs can be used to optimise marketing or political campaigns.
- **Security Assessment & Attack Simulations in Ad-Hoc Networks**: EAs are used to simulate attacks and assess security vulnerabilities in ad-hoc networks, by injecting malicious nodes into the network to determine how much they can decrease the data delivery rate.
- **Co-evolution of Sensor Systems**: EAs are used to develop swarms of passive sensor agents for exploring inaccessible environments, co-evolving the hardware and software and the model of the environment.

Use of EAs by Tech Giants

Major tech companies such as Facebook and Google use evolutionary algorithms for various purposes. For example, Facebook uses EAs to find bugs in Android devices. Google uses evolutionary computation in its AutoML tool, to automatically design machine learning algorithms.

Commercial and Open-Source Software

- Commercial software includes modeFRONTIER, Isight, Kimeme, MATLAB (Genetic Toolbox), and Nexus. These are used for CAD/CAE applications but often have closed-source algorithms that cannot be easily modified.
- Open-source software includes inspyred, deap, peas, jgap, jmetal, and moeaframework. These are used for rapid prototyping and research and can be modified but require algorithmic expertise.

12.2 Recent Trends in Evolutionary Computation

Parameter Control

Metaheuristics depend on parameters, and controlling these is crucial for performance.

Parameter Tuning

Parameter tuning involves testing different sets of parameters before the main run to find optimal values, but it is time-consuming and may not always be the best option. A hyper-heuristic is when an EA optimises the parameters of another EA, although this approach is computationally expensive.

Parameter Control Strategies

- Deterministic: Parameter changes are based on time or number of evaluations.
- Adaptive: Parameters are changed using feedback from the search process, such as the one-fifth rule in evolution strategies.
- Self-adaptive: Parameters are encoded in chromosomes and evolve through natural selection.

Trigger Conditions for Parameter Changes

These can be based on:

- Time or number of evaluations (deterministic).
- Population statistics (adaptive), such as diversity or gene distribution.
- Relative fitness of individuals created with given parameter values (adaptive or self-adaptive).

Key Takeaway

Finding optimal parameter control strategies can be as challenging as optimising parameters themselves.

Memetic Computing

Memetic computing combines evolutionary algorithms with local search or problem-specific operators to improve exploration/exploitation balance.

Motivation

EAs are good at solving a wide range of problems, but in some cases the exploration/exploitation balance may not be enough to obtain satisfactory results.

Memetic Algorithms

- Hybrid GAs, also referred to as Baldwinian/Lamarckian GAs.
- EAs + Local Search.
- EAs incorporating "intelligent" (ad-hoc) initialization, crossover/mutation, and/or problem-specific operators.

Concept of Memes

Memes, like genes, are units of cultural transmission. In MAs, memes are strategies for improving solutions.

Algorithmic Details

Local search can be deterministic or stochastic and gradient based or gradient free. A pool of multiple local search algorithms can be used (Multi-Meme Algorithms). Coordination between local search and EA is crucial.

Approaches to Coordination

- Adaptive MAs: Local search is controlled during evolution by feedback from the search process.
- Self-Adaptive MAs: Adaptive rules are encoded in the genotype of each individual.
- Co-evolutionary MAs: Two populations of solutions and operators co-evolve.

Intelligent Initialisation

Instead of random sampling, custom generators (e.g., orthogonal arrays) can be used, but this might decrease diversity.

Key Takeaway

Memetic computing enhances EA performance by incorporating problem-specific knowledge and local search techniques.

Island Models and Spatial Distribution

These methods aim to preserve diversity in the population for multimodal optimization problems.

Motivation

Most optimisation problems have more than one local optimum, but a finite population with global recombination and selection will eventually converge to one optimum.

Island Model

Multiple populations (islands) are run in parallel, with occasional migration between islands. Different operators can be used on each island.

Migration Frequency

It is important to choose how often migration should occur. Too frequent migrations will cause all islands to converge to the same solution, whereas too rare migrations mean there is little information exchange among the islands.

Diffusion Model

Individuals are placed on a spatial structure (usually a toroidal grid), and selection, recombination and replacement are based on the local neighbourhood.

Speciation

Individuals are recombined only with genotypically/phenotypically similar individuals. Extra genes are added to the problem representation to indicate the species. These species genes are subject to recombination and mutation and can be used as tags to perform fitness sharing.

Fitness Sharing

The number of individuals within a niche is restricted by sharing their fitness to allocate individuals to niches in proportion to niche fitness. The niche size can be set in genotype or phenotype space.

Crowding

Crowding attempts to distribute individuals evenly amongst niches by making similar individuals compete with each other for survival.

Key Takeaway

Spatial distribution and island models help maintain diversity and explore multiple peaks in multimodal problems.

Interactive Evolutionary Computation

Interactive EC is required when it is not possible to define an objective fitness function and subjective judgment is required.

Generalities

This involves a human evaluator in the selection and variation processes and in the fitness evaluation. The user selects the individuals that will take part in reproduction.

Key Takeaway

Interactive EC integrates human subjectivity, which allows for evolution in cases where objective fitness functions do not exist.

Fitness-Free Evolutionary Algorithms

These algorithms abandon the goal of improving performance and instead focus on finding behaviourally different solutions.

Motivation

Traditional methods are too focused on objectives, and greatness should not result from measuring improvement in the pursuit of a goal.

Novelty Search (NS)

The algorithm keeps an archive of the most novel solutions found. It maximises behavioural novelty rather than fitness.

MAP-Elites

This discretises an n-dimensional behavioural space and keeps the best individual in each bin of the grid. It is called an "illuminating algorithm" as it highlights the fitness potential of each region in the space.

Quality Diversity (QD)

QD maximises the total fitness across all filled grid bins within the behavioural space.

Key Takeaway

Fitness-free algorithms prioritize diversity and novelty, often outperforming traditional methods in complex tasks.

Combining Machine and Reinforcement Learning with Evolutionary Computation

This is an area of ongoing research, exploring ways to combine reinforcement learning (RL) with EAs.

RL-Powered EAs

RL can be used to learn a model of the problem, learn the best algorithm parameters, or learn the best algorithm operators to execute. This can be useful in problems where the function evaluations are computationally expensive.

RL+EAs for Agent-Based Tasks

Hybrid schemes where agent controllers are partially evolved and partially learned are used, especially in cases with uncertain or missing rewards.

The Triangle of Life

This concept, which is similar to what happens in nature, involves the phases of birth, infancy and maturity. In artificial agents, properties can be inherited from their parents (birth), they can learn during their lifetime (infancy), and eventually they can reproduce (maturity).

Key Takeaway

Combining RL and EAs can leverage the strengths of both to solve problems that are beyond the capabilities of each approach alone.

12.3 Conclusion

Evolutionary computation techniques are widely used across various industries and research fields. By adapting these algorithms and combining them with other tools such as machine learning, these methods will become even more important in solving complex problems in the future.

Special Thanks

We would like to extend our heartfelt gratitude to all the individuals and organizations who have contributed to the success of this project. In particular, we would like to thank OpenAI's Whisper and GPT-4o, Google's NotebookLM and Microsoft's Copilot for their invaluable assistance in the creation of this document.