Evolutionary Algorithms

Dr. Sascha Lange
AG Maschinelles Lernen und Natürlichsprachliche Systeme
Albert-Ludwigs-Universität Freiburg

slange@informatik.uni-freiburg.de

Acknowlegements and Further Reading

These slides are mainly based on the following three sources:

- A. E. Eiben, J. E. Smith, *Introduction to Evolutionary Computing*, corrected reprint, Springer, 2007 — recommendable, easy to read but somewhat lengthy
- B. Hammer, *Softcomputing*, Lecture Notes, University of Osnabrück, 2003 — shorter, more research oriented overview
- T. Mitchell, *Machine Learning*, McGraw Hill, 1997 — very condensed introduction with only a few selected topics

Further sources include several research papers (a few important and / or interesting are explicitly cited in the slides) and own experiences with the methods described in these slides.

'Evolutionary Algorithms' (EA) constitute a collection of methods that originally have been developed to solve combinatorial optimization problems. They adapt Darwinian principles to automated problem solving. Nowadays, Evolutionary Algorithms is a subset of Evolutionary Computation that itself is a subfield of Artificial Intelligence / Computational Intelligence.

Evolutionary Algorithms are those metaheuristic optimization algorithms from Evolutionary Computation that are population-based and are inspired by natural evolution. Typical ingredients are:

- A population (set) of individuals (the candidate solutions)
- A problem-specific fitness (objective function to be optimized)
- Mechanisms for selection, recombination and mutation (search strategy)

There is an ongoing controversy whether or not EA can be considered a machine learning technique. They have been deemed as 'uninformed search' and failing in the sense of learning from experience ('never make an error twice'). However, they have been applied successfully to problems that are at the very heart of machine learning.

History of Evolutionary Algorithms

Around ever since the early days of computing: Box 1957, Bledsoe 1961

Pioneered in the 1960s and early 70s as

- Genetic Algorithms (GA) by Holland and Goldberg (US):
  optimization of *bit strings* in analogy to discrete-valued DNA-sequences
- Evolution Strategies (ES) by Rechenberg and Schwefel (Europe):
  similar techniques, but using *real-valued numbers* and only *mutation*

Have been developed in parallel for about two decades.

Nowadays considered as two different flavours of the same thing (EA).

More recent developments include:

- Neuroevolution: evolution of (recurrent) neural networks in control tasks
- Evolutionary Image Processing: analyzing and understanding images with help of evolutionary programming

Outline

1 Motivation

2 Framework

3 Representations

4 Applications

5 Discussion

Section 1: Motivation

- Natural Evolution
- Surface Metaphor
- Convergence

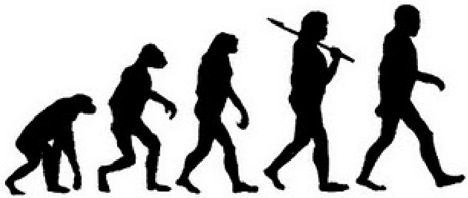
Blue-print Natural Evolution

Why might Evolution be an interesting model for computer algorithms?

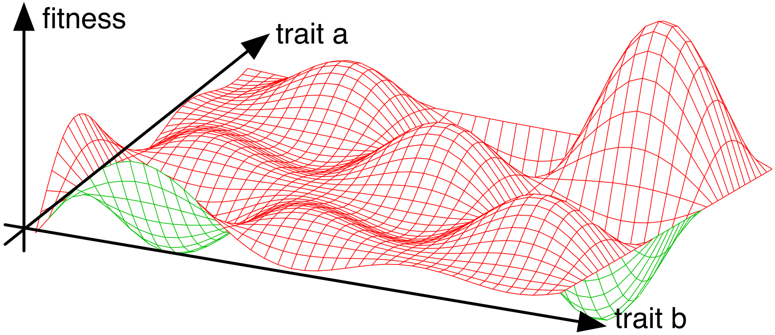
- Evolution has proven a powerful mechanism in ‘improving’ life-forms and forming ever more complex species.
- Driven by suprisingly simple mechanisms, nevertheless produced astonishing results.

Evolution is basically a random process, driven by evolutionary pressure:

1. Tinkering with genes (Genotype)
   - Mating: recombination of genes in descendants
   - Mutation: random changes (external influences, reproduction errors)
2. Testing (Phenotype), Competition (‘Survival of the fittest’)

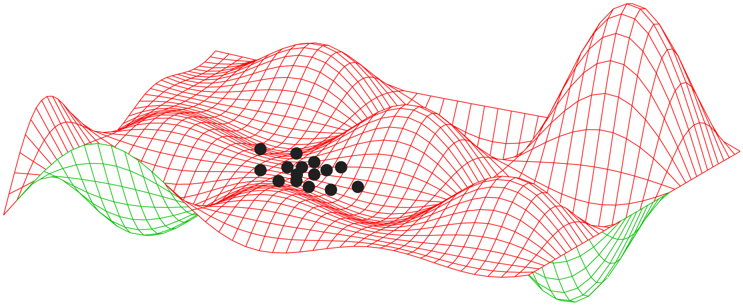


Surface metaphor



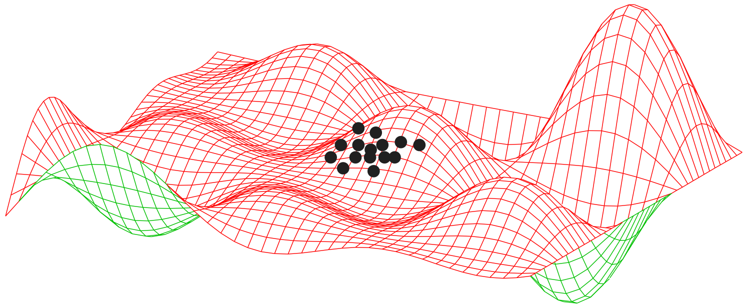
- Traits and fitness form a surface with hills and valleys.

Surface metaphor



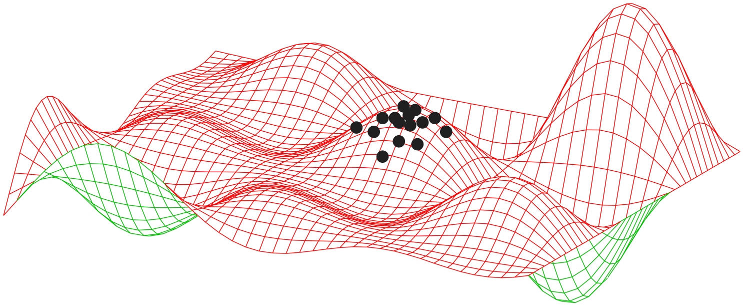
- Traits and fitness form a surface with hills and valleys.
- Population 'travels' this surface and slowly climbs the hills.

Surface metaphor



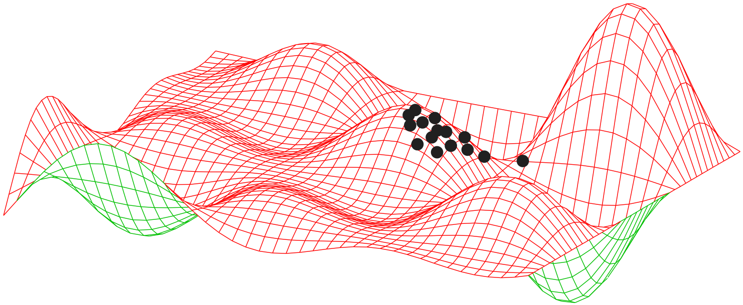
- Traits and fitness form a surface with hills and valleys.
- Population ‘travels’ this surface and slowly climbs the hills.

Surface metaphor



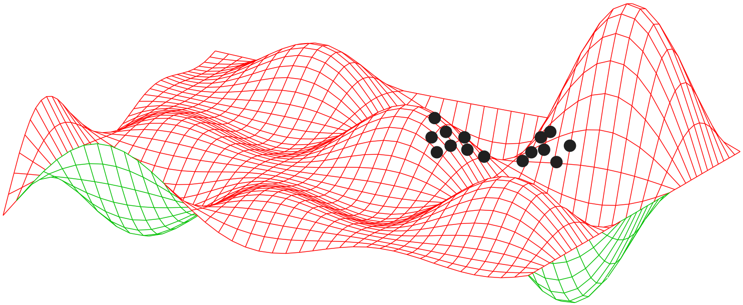
- Traits and fitness form a surface with hills and valleys.
- Population ‘travels’ this surface and slowly climbs the hills.

Surface metaphor



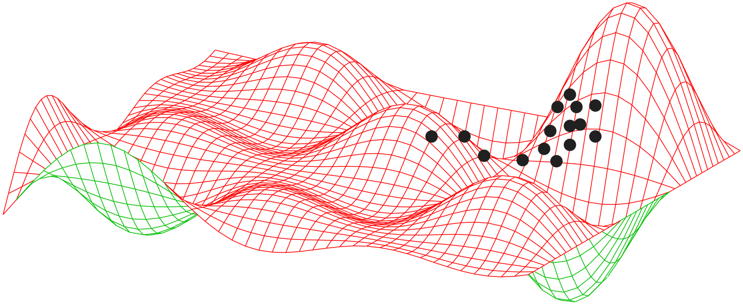
- Traits and fitness form a surface with hills and valleys.
- Population 'travels' this surface and slowly climbs the hills.
- Due to genetic drift it's possible to also travel through valleys and reach another (higher) hill.

Surface metaphor



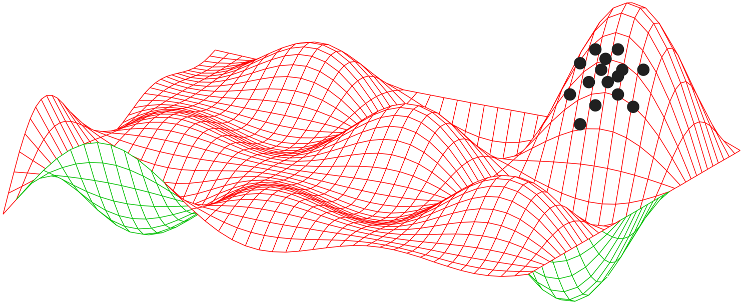
- Traits and fitness form a surface with hills and valleys.
- Population ‘travels’ this surface and slowly climbs the hills.
- Due to genetic drift it’s possible to also travel through valleys and reach another (higher) hill.

Surface metaphor



- Traits and fitness form a surface with hills and valleys.
- Population ‘travels’ this surface and slowly climbs the hills.
- Due to genetic drift it’s possible to also travel through valleys and reach another (higher) hill.

Surface metaphor



- Traits and fitness form a surface with hills and valleys.
- Population ‘travels’ this surface and slowly climbs the hills.
- Due to genetic drift it’s possible to also travel through valleys and reach another (higher) hill.

Convergence of Natural Evolution

One could be uncomfortable with such a random process (no driving force):

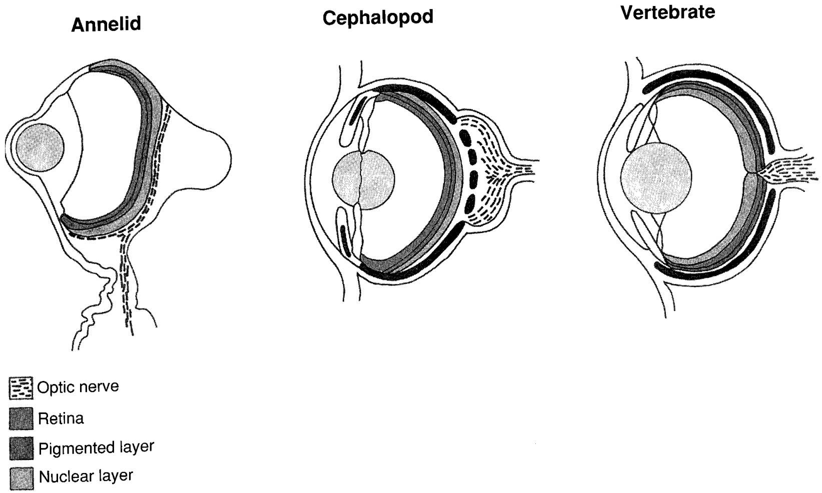
- Does it find solutions just by (unlikely) coincidence?
- How random are the solutions it finds? Is it repeatable?
- We're looking for 'specific' or even the 'optimal' solutions.

But, there is promising empirical evidence for evolution to work in a desired way. Example from natural evolution: hypothesis of 'convergence'.

- The argument is that results and 'solutions' found by evolution are not purely random but to a certain degree are repeatable and 'reasonable'.
- The details are random, but the principles are heavily constrained by environmental and physical necessities up to being 'inevitable'.
- Moreover, if evolution would be restarted on earth, the outcome might not be exactly the same but neither completely different.

Advocates of this argument try to justify it by looking at separated 'niches' of our eco-system (we have only one evolution at hand), identifying similar solutions found by independent processes; thus a 'convergence' of solutions.

Example: Camera-Eye

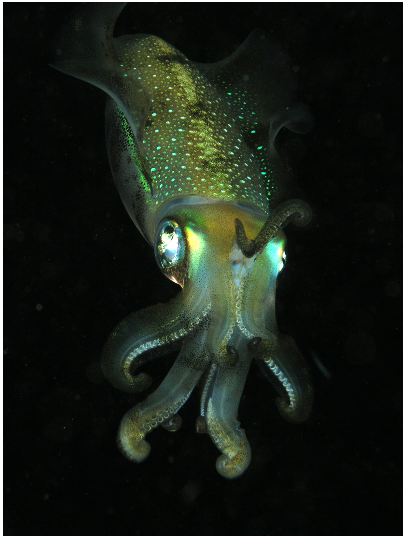


from: Simon Conway Morris, *Life's solution*, 2003

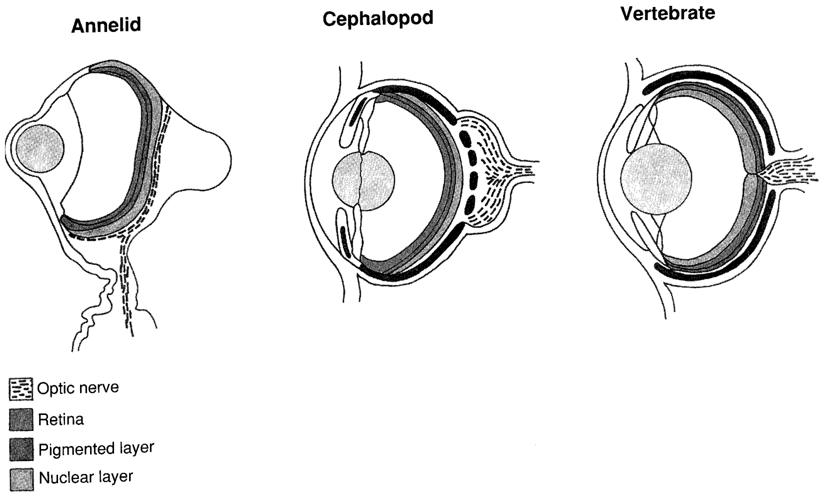
Example: Camera-Eye

Annelid (Ringelwurm). Image from: NOAA National Estuarine Research Reserve

Example: Camera-Eye

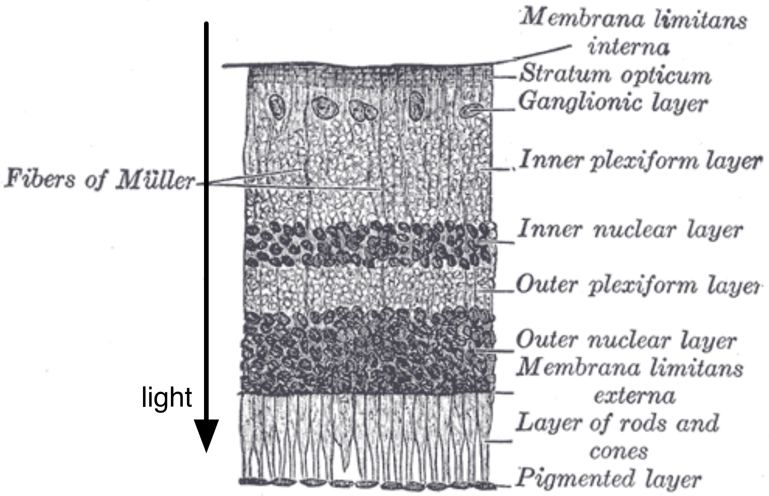

Cephalopod (Kopffüssler). Image from: Nhobgood, 2006, CCA-SA 3.0

Example: Camera-Eye



from: Simon Conway Morris, *Life's solution*, 2003

Example: Camera-Eye (continued)



from: *Gray's Anatomy*, 1918

Noteworthy Principles

Basics:

- Evolution is a random process of selection, reproduction and mutation
- It's driven by evolutionary pressure ('survival of the fitesst')
- 'Search' is conducted with generations of populations, not by improving individuals

Details:

- Populations evolving in different niches can independently develop different (but similar) solutions to the same problem (EA: parallel evolution, island model)
- Solutions may reach a local optimum from which it's hard to achieve any significant improvements (example: human eye)
- Nevertheless, it's possible that populations leave a hill and 'wade through the water' to finally reach a better hill (surface metaphor, genetic drift)
- Fitness of individuals may depend on the other individuals in the population (example: predator — prey, EA: coevolution)
- 'Good' solutions are somehow constrained by the environment (convergence, inevitable?)

Section 2: Framework

- From Biology to Computer Science
- Basic Framework
- Example: evoVision
- Advanced Techniques

From Natural Evolution to Evolutionary Algorithms

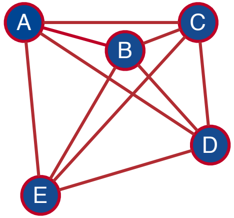
- Natural evolution has proven a powerful optimization process
- We have identified it's main ingreedients
- How can we use these principles for solving optimization problems?

Example: Traveling Salesman Problem (TSP)

Task: In a complete graph, given a list of pairwise distances between its nodes, find the shortest tour that visits every node exactly once. NP-hard optimization problem.

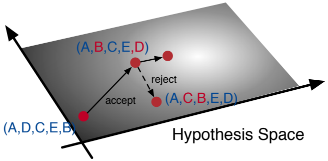
Naive search algorithm:

- Start with a random tour
- Loop:
  1. Alter tour randomly
  2. Keep new tour, iff shorter



Problems:

- Might get stuck in local optima
- Might be necessary to become worse in order to 'escape'
- Solution (local optimum) found heavily depends on starting point



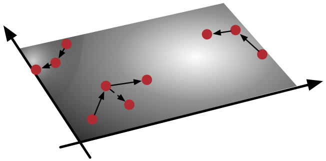
Example: Traveling Salesman Problem (TSP) (cont.)

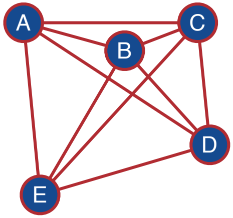
Idea: Search from different starting positions in parallel

- Explore different regions of hypothesis space
- Will end up in different local optima
- More likely to find global optimum (in the limit P towards 1)

Problem: How to distribute 'computing' power?

- Equal distribution: Same as doing naive search several times in a row
- Exploring always the best so far: equals naive search

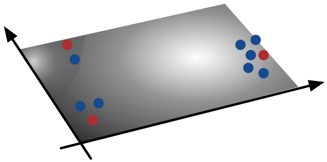




Example: Traveling Salesman Problem (TSP) (cont.)

Idea: Distribute according to quality

- Assign more computing power to exploring better regions
- Realized by exploring more descendents of already good tour
- Don't throw non-optimal tours away (immediately) but continue to explore their descendents

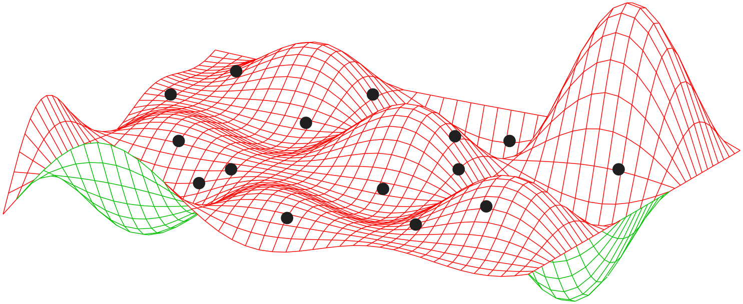


Idea for further improvement:

- Combine a tour that is good at the cities A, B, C with another tour that is good at D and E.

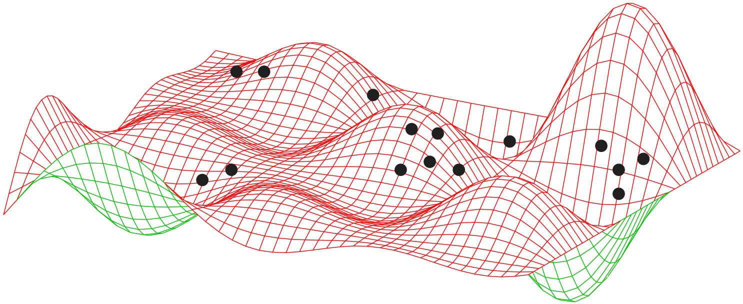
$$\begin{array}{r} (A,B,C,E,D) \\ +\ (B,C,A,D,E) \\ \hline (A,B,C,D,E) \\ \hline\hline \end{array}$$

Surface metaphor (EA)



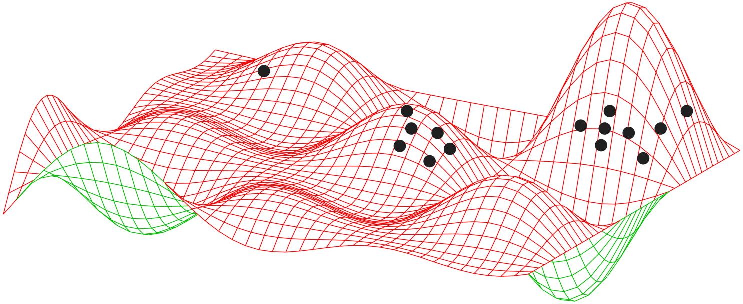
- Start with a random-initialized population of candidate solutions.
- Population ‘travels’ this surface and slowly climbs the hills.

Surface metaphor (EA)



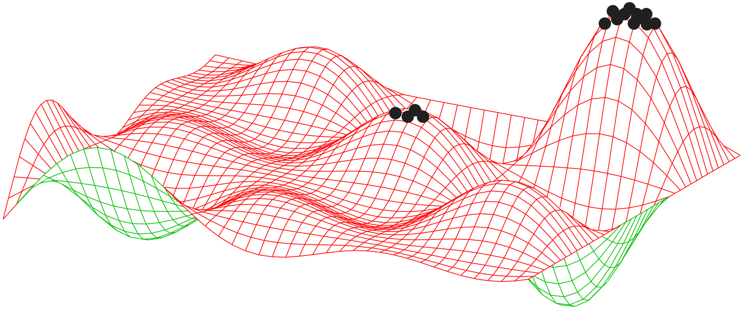
- Start with a random-initialized population of candidate solutions.
- Population ‘travels’ this surface and slowly climbs the hills.

Surface metaphor (EA)



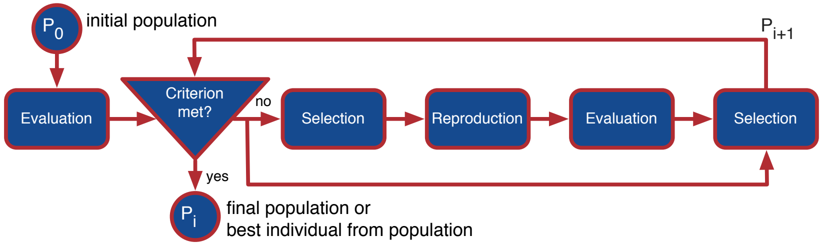
- Start with a random-initialized population of candidate solutions.
- Population ‘travels’ this surface and slowly climbs the hills.

Surface metaphor (EA)



- Start with a random-initialized population of candidate solutions.
- Population 'travels' this surface and slowly climbs the hills.
- Eventually, a (sub-)population will 'converge' on the global optimum.

General Framework of Evolutionary Algorithms



- Individuals: hypothesis x from a hypothesis space X
- Population: collection P of μ present hypotheses $P = \{x_i | i = 1, \ldots, \mu\}$
- Evaluation: apply a mapping $f : X \mapsto R$ (fitness function) to all individuals
- Selection mechanism: selects individuals $x \in P_i$ for reproduction (mating); selects individuals from offsprings and P_i to form the new population P_{i+1}
- Reproduction: combination of two or more individuals (Crossover) and random alteration (Mutation).

Individuals

Individuals are the chosen representation of the candidate hypotheses.
Within EA you have (nearly) free choice of the model!

Common representations within EA:

- Bit-strings: binary representations of logic formulae (e.g. rules), values of boolean variables, ... $\mapsto$ Genetic Algorithms
- Real-valued: parameter vectors of a polynom of 3rd degree, a control law, a neural network, a process, ... $\mapsto$ Evolutionary Strategies
- Structured: Decision trees, neural networks, programs, ... $\mapsto$ Genetic / Evolutionary Programming, Neuroevolution

Restriction: Definition of (at least) a meaningful mutation-operator for a given representation must be possible. (Crossover operator is no necessity.)

Example TSP: Sequence of nodes to visit, X the set of all permutations of (A, B, C, D, E).

Fitness Function

The fitness function ‘rates’ the quality of the candidate solutions and forms the basis for the selection procedure. Thus, it’s problem dependent! Usually:

- A function $f : X \mapsto R$ with $f(x) \geq 0, \forall x \in X$

The fitness may be either a direct function of the individual’s parameters, or it also may involve more complex computations or even a testing procedure that is performed in the real-world.

Examples:

- Calculating the generalization error of an evolved image-classifier on validation data
- Measuring the time an evolved control law manages to balance a pole
- Walking distance (within 30s) of a robot dog using an evolved gait pattern

As such, the fitness may have a non-deterministic component!
In nature: difference between testing the genotype or phenotype.

Example TSP: $f : X \mapsto R : f(x) = f(x_1, \ldots, x_5) = 1 - \frac{d(x_1,x_2)+d(x_2,x_3)+\ldots+d(x_4,x_5)}{\max_{i,j} d(x_i,x_j) \cdot 4}$

Selection of Parents

Randomly selects individuals of a population that get the chance for generating offsprings. The sampling is usually done from a probability distribution somehow derived from the fitness of the individuals. Specifically, fitter individuals must be more likely to send offsprings to the next generation than less fit individuals.

- Number of individuals to select is a parameter; it's relation to the population size diffesr among EA variants (from smaller to larger than μ)
- Usually, it's allowed to select the same individual more than once (selection with or without replacement)

Selection mechanisms commonly found in EAs:

- Fitness proportional selection (roulette wheel selection)
- Ranking selection
- Tournament selection

- Uniform selection (pressure must come from survivor selection)

Selection of Parents: Fitness proportional selection

Probability $P(x_i)$ for selecting the individual x_i with fitness $f(x_i)$ is given by

$$P(x_i) = \frac{f(x_i)}{\sum_{j=1}^{\mu} f(x_j)}$$

Thus, the selection probability depends on the absolute fitness value of the individual compared to the absolute fitness values of the rest of the population.

Problems

- Premature Convergence: outstanding individuals dominate population too early
- Almost no selection pressure, when all individuals have similar fitness
- Different behavior on transposed versions of the fitness function

Improvements

- Windowing: Subtract a constant *beta*$_i$ from the fitness of each individual, e.g. $\beta_i = \min_{x \in P_i} f(x)$. Doesn't help with premature convergence.
- Sigma-Scaling: $f'(x) = \max\left(f(x) - (\bar{f} - c \cdot \sigma_f), 0\right)$ using the variance σ_f

Selection of Parents: Roulette Wheel Algorithm

Given the likelihood of each individual being selected for reproduction, how do we determine what individuals to select how often? Typically, the expected number of copies of an individual (likelihood multiplied by the total number λ of parents to select) is noninteger. Thus, we have to sample the parents.

```
$D \leftarrow$ empty collection (might contain multiple copies of same member)
WHILE $|D| < \lambda$
  $r \leftarrow$ uniformly picked random value from $[0, 1]$
  $i \leftarrow 0$
  DO
    $i \leftarrow i + 1$
    $r \leftarrow r - P(x_i)$ where $x_i$ $i$-th element of population $P$
  WHILE $r > 0$
  add $x_i$ to collection $D$
return D
```

Selection of Parents: Roulette Wheel Algorithm

Given the likelihood of each individual being selected for reproduction, how do we determine what individuals to select how often? Typically, the expected number of copies of an individual (likelihood multiplied by the total number λ of parents to select) is noninteger. Thus, we have to sample the parents.

```
$D \leftarrow$ empty collection (might contain multiple copies of same member)
WHILE $|D| < \lambda$
  $r \leftarrow$ uniformly picked random value from $[0, 1]$
  $i \leftarrow 0$
  DO
    $i \leftarrow i + 1$
    $r \leftarrow r - P(x_i)$ where $x_i$ $i$-th element of population $P$
  WHILE $r > 0$
  add $x_i$ to collection $D$
return D
```

Subtracting a probability $P(x_i)$ from the random value can be seen as letting a ball roll over a field (with its size proportional to $P(x_i)$) of a roulette wheel.

Selection of Parents: Roulette Wheel Algorithm

Given the likelihood of each individual being selected for reproduction, how do we determine what individuals to select how often? Typically, the expected number of copies of an individual (likelihood multiplied by the total number λ of parents to select) is noninteger. Thus, we have to sample the parents.

```
$D \leftarrow$ empty collection (might contain multiple copies of same member)
WHILE $|D| < \lambda$
  $r \leftarrow$ uniformly picked random value from $[0, 1]$
  $i \leftarrow 0$
  DO
    $i \leftarrow i + 1$
    $r \leftarrow r - P(x_i)$ where $x_i$ $i$-th element of population $P$
  WHILE $r > 0$
  add $x_i$ to collection $D$
return D
```

Subtracting a probability $P(x_i)$ from the random value can be seen as letting a ball roll over a field (with its size proportional to $P(x_i)$) of a roulette wheel.

Problem: The sample might quite largely deviate from ideal distribution.

Selection of Parents: Stochastic Universal Sampling

```
$D \leftarrow$ empty collection (might contain multiple copies of same member)
$i \leftarrow 1$
$r \leftarrow$ uniformly picked random value from $[0, 1/\lambda]$
WHILE $|D| < \lambda$
   WHILE $r \leq P(x_i)$ (where $x_i$ $i$-th element of population $P$)
      add $x_i$ to collection $D$
      $r \leftarrow r + 1/\lambda$
   $r \leftarrow r - P(x_i)$
   $i \leftarrow i + 1$
return D
```

Selection of Parents: Stochastic Universal Sampling

Idea: Just draw one random number to determine the whole sample. Spin only one time a roulette wheel with λ equally spaced arms, instead of spinning a one-armed wheel λ-times.

```
$D \leftarrow$ empty collection (might contain multiple copies of same member)
$i \leftarrow 1$
$r \leftarrow$ uniformly picked random value from $[0, 1/\lambda]$
WHILE $|D| < \lambda$
   WHILE $r \leq P(x_i)$ (where $x_i$ $i$-th element of population $P$)
      add $x_i$ to collection $D$
      $r \leftarrow r + 1/\lambda$
   $r \leftarrow r - P(x_i)$
   $i \leftarrow i + 1$
return D
```

The number of copies of each parent x_i is

- at least the integer part of $\lambda \cdot P(x_i)$
- no more than one greater,

because r initialized in $[0, 1/\lambda]$ and incremented by $1/\lambda$ with every selection.

Selection of Parents: Ranking Selection

Preserves constant selection pressure by sorting the population on the basis of fitness and then allocating selection probabilities to individuals according to their rank.

- the mapping from rank to selection probability can be done arbitrarily
- e.g. using a linearly or exponentially decreasing mapping
- as long as the probabilities add up to one

Ranking selection does not suffer from premature convergence and does not have the same problems as fitness-proportional selection with transposed versions of the fitness function.

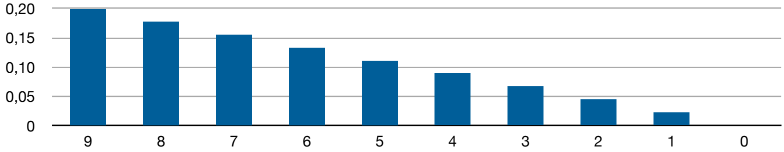
Selection of Parents: Linear Ranking Scheme

One particular ranking scheme that is often found in GA (with $i = 0$ the rank of the worst individual and $i = \mu - 1$ the rank of the best):

$$P_{lin\_rank}(x_i) = \frac{2-s}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)}$$

Here, we assume the size of the parent population μ equals the number of produced descendents λ. $s \in (1, 2]$ is a parameter controlling the expected number of copies of the highest-ranked individual.

Results for $s = 2$ and $\mu = 10$ individuals:



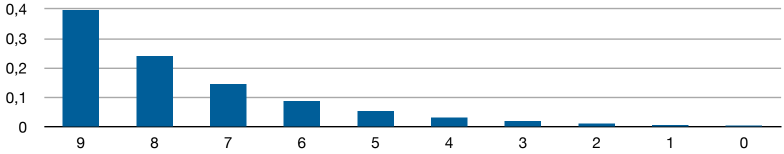
Selection of Parents: Exponential Ranking Scheme

If more selection pressure is needed and higher-ranked individuals should be more likely to being selected, one could use an exponential function for mapping ranks to selection probabilities, e.g. (with $i = 0$ the rank of the worst individual and $i = \mu - 1$ the rank of the best):

$$P_{exp\_rank}(x_i) = \frac{e^{-s(\mu-i)}}{\sum_{j=0}^{\mu-1} e^{-s(\mu-j)}}$$

Again, we assume the size of the parent population μ equals the number of produced descendents λ. $s \in (0, 1]$ is a parameter controlling the probability mass on the higher-ranked individuals.

Results for $s = 0.5$ and $\mu = 10$ individuals:



Selection of Parents: Tournament Selection

Tournament Selection (TS) is another selection mechanism that does look only at relative fitnesses and that has the same beneficial properties as the ranking schemes regarding translated and transposed versions of the fitness function.

```
$D \leftarrow$ empty collection (might contain multiple copies of same member)
WHILE $|D| < \lambda$
    select $k$ individuals randomly (with or without replacement)
    determine the best of these $k$ individuals comparing their fitness values
    add the best individual to $D$
return $D$
```

TS is widely used because the easy control of its selection pressure through

- the tournament size $k \in \{2, 3, 4, \ldots\}$ (larger $k \mapsto$ more pressure)
- the probability of selecting the winner (usually $p = 1$, highest pressure)
- replacement (without replacement: $k - 1$ worst cannot be chosen)

and because it doesn't have to know absolute fitness values of all individuals but only relative fitnesses of the tournament participants. Example: two candidate solutions compete against each other and the winner is selected.

Selection of Survivors

This processing step is responsible for producing the next generation P_{i+1} from the old population P_i and newly formed offsprings. It's mechanisms are closely coupled to the earlier parent selection.

In general, there are two principle population models to select from:

Generation-based

- given a population of size μ
- select a 'mating pool' of parents from the population
- produce λ offsprings (in GA often: $\lambda = \mu$)
- the whole population is replaced by $\mu \leq \lambda$ offsprings
- may loose the fittest individual and maximal fitness may decline again

Steady-State

- given a population of size μ
- produce a number of offsprings
- replace only part of the population by $\lambda < \mu$ offsprings

Selection of Survivors: Age-Based Replacement

Replacement of the individuals of a population independent of their fitness value but depending on their age (number of generations the individual survived). Keeping the individuals with highest fitness in this scheme depends on them being selected for reproduction.

Possible realizations:

- In the extreme case where $\lambda = \mu$ each individual just survives one generation, as the whole population is replaced in each step (generation-based)
- In the other extreme case of $\lambda = 1$ only the oldest individual is “killed” and each individual survives μ generations (realizing a FIFO) (steady-state)
- In the steady-state case of $\lambda < \mu$ the λ oldest are replaced

Note: Randomly selecting “dying” individuals is also considered age-based replacement. Although used quite often, using random selection is strongly discouraged as loosing the fittest individual is more likely (Smith and Vavak).

Selection of Survivors: Fitness-Based Replacement

Fitness-based replacement is a widely used technique with countless variants for selecting the μ individuals that form the new population from the $\mu + \lambda$ parents and offsprings. In principle, all techniques discussed for the selection of parents can be also used here, based on inverse fitness or ranks. Further techniques:

Replace Worst (GENITOR)

- Replace the λ worst individuals
- Rapidly improving mean fitness, but: danger of premature convergence
- Thus: use only with large populations and / or no-duplicates policy

Elitism

- Add-on to all age-based and stochastic fitness-based schemes
- Rule: always keep the fittest member in population
- If the fittest individual is selected for replacement and no offspring with better fitness is inserted, keep it and discard another individual
- Guaranteed monotonic improvement of fittest individual in population

Reproduction

The task of the reproduction step is to create new individuals from old ones. There are two principal types of variation operators: unary and binary operators.

Unary: Mutation

- Applied to one individual, delievers a "slightly" changed mutant (offspring)
- Mutation is almost always stochatic, causing a random, unbiased change

Binary: Recombination or "Cross-Over"

- Merges information from two parents into one or two offsprings
- As mutation, involves random choices of what and how to merge
- Often used option: a non-zero chance of this operator not being applied
- Operators involving more parents are possible, but seldom used

The details of the used operators depend on the particular representation. Thus, the operators will be discussed in more detail in the corresponding section.

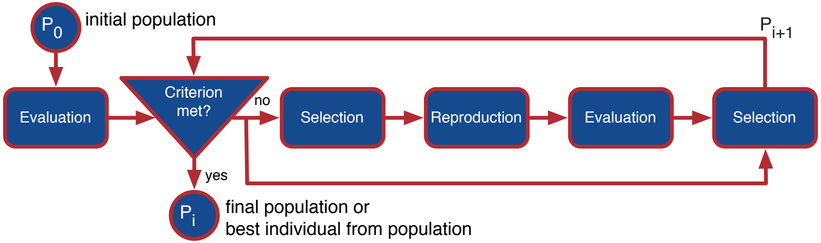
Termination Condition

Ideal world: if we know the optimum of the objective (fitness) function (and have unlimited time), we could stop the procedure when a candidate solution is found with a fitness that is within a given boundary $\epsilon > 0$ of the optimum.

Practice: in most cases one of the following stopping criteria is used:

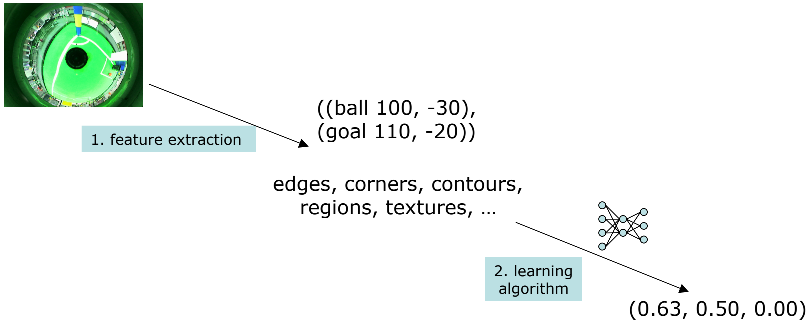
- The maximally allowed time elapses (CPU time, real time)
- The total number of fitness evaluations reaches a given limit
- The fitness improvement remains under a given threshold for a given period of time
- The diversity of the population remains under a given threshold
- A given number of generations has been evolved

Example: evoVision — Exemplary Realization of the Modules



Example: evoVision — Supervised Learning with Feature Extraction

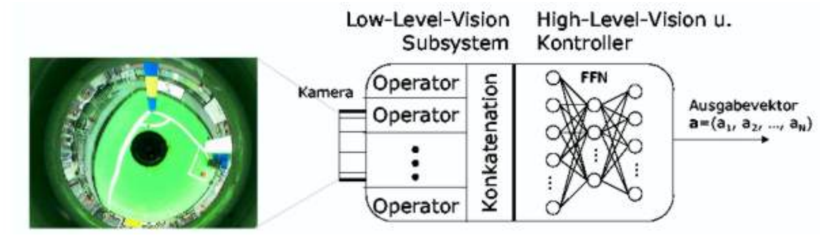
Common approach to learning with images (classification or regression): two layered architecture.



1. feature extraction
((ball 100, -30), (goal 110, -20))
edges, corners, contours, regions, textures, ...
2. learning algorithm
(0.63, 0.50, 0.00)

Open problem: how can the feature extraction subsytem also be learned?

Example: evoVision — Representation


Low-Level-Vision Subsystem
High-Level-Vision u. Kontroller
Kamera
Operator
Operator
Operator
Konkatenation
FFN
Ausgabevektor
$a=(a_1, a_2, \ldots, a_N)$

Phenotype: image processing algorihtms (operators) & trained neural network
Genotype: boolean, ordinal and real-valued parameters of image processing algorithms, for controlling the neural net's topology, selecting the error function

Exemplary Operators:

- Histograms. Parameters: number of windows, number of buckets
- Colored-Blobs-Encoder. Parameters: Prototype colors for color segmentation of the whole image, properties of blobs to pass on (e.g. position, size, bounding box, roundness, etc.)

Example: evoVision — Reproduction

Cross-Over

- The i-th parent is recombined with both the $(i - i)$-th and the $(i + 1)$-th parent (produces as many offsprings as selected parents)
- The offspring inherits a randomly selected subset of the joint set of its parents' operators as well as the neural net topology of its first parent.
- Parameters of operators and neural net remain unchanged.
- There is a chance of $p = 1 - r_{recombination}$ for each parent remaining unchanged and being copied directly to the collection of offsprings.

Mutation

- Each offspring is mutated with a chance of $r_{mutation}$
- Mutation can delete operators or add random-initialized new operators
- Each parameter is changed with a 10%-chance
- Real-valued parameters get added normal distributed noise
- Ordinal numbers are drawn from a uniform distribution
- Ranges (variances derived) are specified for each parameter individually

Example: evoVision — Parameter Abstraction

```
class ParameterMetaInfo {
public:
  enum parameterType {
    BOOL=0,    //!< interpret as vector of boolean values
    INT,       //!< interpret as vector of integer values
    DOUBLE     //!< interpret as vector of double values
  };
  ParameterMetaInfo(enum parameterType type,
                    unsigned int dimension,
                    const vector<double>& minValues,
                    const vector<double>& maxValues);

  Parameter create() const;
};

class Parameter {
public:
  const ParameterMetaInfo * const getMetaInfo() const;

  vector<double> getValue() const;
  void setValue(const vector<double>& v);
};
```

Example: evoVision — Parameter Abstraction

```
class Individual {
public:
  virtual Individual* clone() const;

  const vector<ParameterMetaInfo>& getParameterDescriptions() const;

  void addParameter( const Parameter& );
  Parameter* getParameter(unsigned int i) const;
  virtual bool isParameterValid(int i);

  virtual void initialize();
};
```

Example: evoVision — Population Model

- Random initialization of candidates
- Constant population size of (typically) 60 individuals (quite small)
- Selection: Fitness proportional selection of 20 parents for mating, implemented with roulette wheel algorithm (not optimal!)
- Replacement: The $\lambda = 15$ best offsprings replace the 15 worst individuals (GENITOR)
- Thus: steady-state population model

Problem in experiments: Some operators quickly dominated the whole population. That was due to some operators already producing ok-results with initial parameters (e.g. histogram), whereas other operators needed much more time to optimize parameters to just reach average results.

Implemented solution: Start with 30 epochs of parallel evolution of several sub-populations that each only contain one particular operator. After 30 epochs merge the candidates with already pre-optimized parameters into one population (fitness-based). Crossover slowly mixes operators into offsprings.

Example: evoVision — Fitness

The fitness of the individuals is determined testing the fully trained neural network on the training and testing data sets.

The fitness includes the following components:

- training error (directing early evolution)
- generalization error (directing evolution in later phases)
- size of description generated by the operators (smaller is better)
- size of neural network (smaller is better)
- computing time (real-time capabilities wanted!)

The computing time turned out to be a very useful component in replacing both size components and in speeding-up the whole evolutionary process.

All components were transformed (non-negative, larger = better) and scaled appropriately. Determining good weighting factors for the components wasn't easy and had large influence on results in particular tasks.

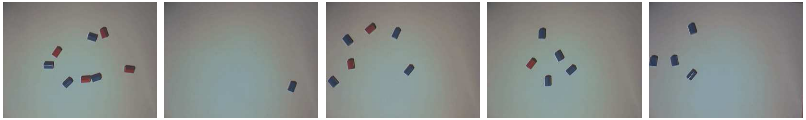
Example: evoVision — Termination

Very basic, non-optimized termination criterion:

- Fixed number of generations
- Number of generations in the range from 200 to 1000 (problem dependent)
- Tendency: longer than necessary

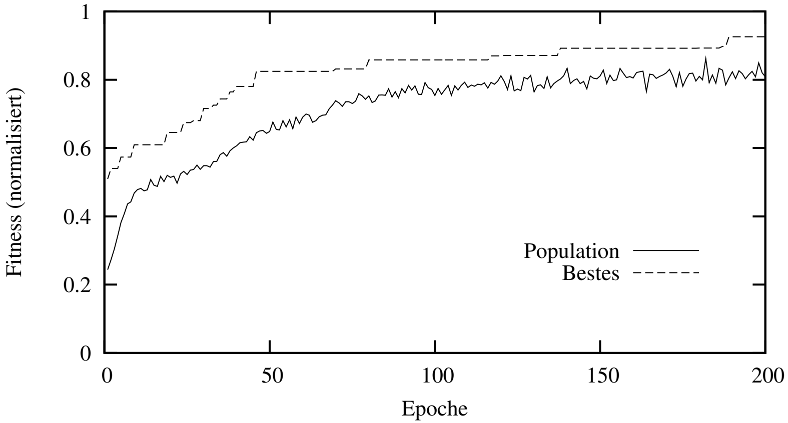
Note: Overfitting here was only a minor problem because the fitness function contained a component reflecting the generalization error / generalization performance on a validation set.

Example: evoVision — Classification Tasks

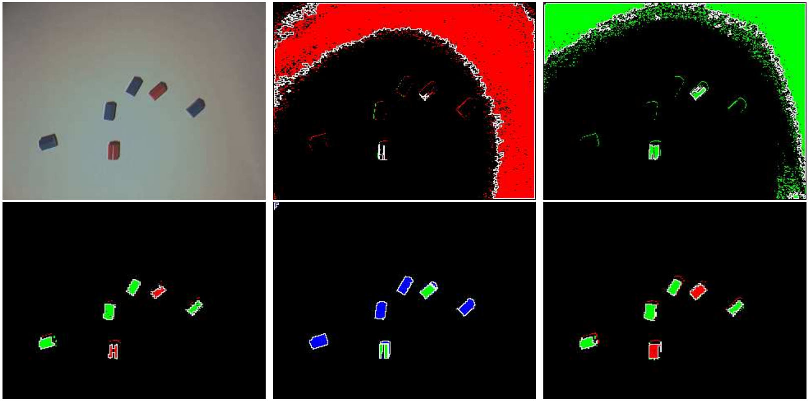


	Subtraktion	
	Testmenge	Testmenge 2
keine Isolation	96%	44%
10 Epochen isoliert	96%	56%
30 Epochen isoliert	100%	100%

Example: evoVision — Exemplary Learning Curve



Example: evoVision — Exemplary Improvement of Fittest Individual



Example: evoVision — Movie

Driving to a ball: training data — learned behavior 1, learned behavior 2
Turning towards a balll: learned behavior
Driving to the goal: learned behavior

Theoretical Properties of the EA Framework

It is often noted that Evolutionary Algorithms do a global search and are not affected by local optima as other, local search and learning methods are.

In fact, in many cases it can be proven that EAs are guaranteed to find the optimal solution with probability 1 if we let them 'search long enough'.

This is often done by proving the probability $P(x_{optimal} \in P_i)$ of having an optimal candidate solution $x_{optimal}$ with the best possible fitness value in the population P_i going to 1 as the number i of generations goes to ∞:

$$\lim_{i \to \infty} P(x_{optimal} \in P_i) = 1.$$

Necessary condition: hypothesis space is connected; mutation and cross-over operator can reach every possible hypothesis

Can be easily achieved by using a mutation operator *mutate* with a small but positive probability of mutating any given individual directly into any other possible individual: $\forall x_1, x_2 \in X \quad P(mutate(x_1) = x_2) > 0$.

Criticism: no convergence rate, no error-bounds if searching only for finite time

Advanced Techniques

In the following, we will briefly discuss several advanced techniques. These can be used as generic (problem independent) extensions of the basic framework.

- Coevolution
- Parallel Distributed Evolution (Island Model EA)
- Self Adaptation

Advanced Techniques: Coevolution

Two (or more) separate populations are evolved at the same time with the intention of stimulating the evolutionary improvement of both.

Cooperative Coevolution

A taks is split into two or more sub-task and several populations each solving one of the sub-tasks are evolved in parallel. The fitness is tested on the whole task, combining sub-solutions from the populations.

One major problem is how to select the individuals that are combined for fitness-testing. For example, 1) test each individual of population A with best individual of population B or 2) test each individual of population A with n randomly selected individuals (encounters) of population B.

Competitive Coevolution

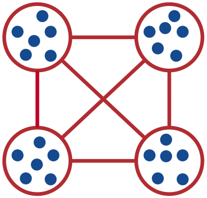
In this scheme individuals compete with each other and gain fitness at each other's expense. Competing individuals can be members of the same population or of different, separately evolved populations.

Classic examples: board games, prisoners dilemma.

Advanced Techniques: Parallel Distributed Evolution

Individuals are spread among several sub-populations that are evolved in parallel and (most of the time) independently from each other. Benefits:

- Simple but effective scheme for utilizing parallel processors (speed-up)
- Helps fighting premature convergence (can be used as initial phase)



Sub-population "communicate" with each other from time to time, by exchanging individuals. Questions regarding this "migration" process involve:

- When to migrate?
- Which individuals to migrate (e.g. random or fitness-based selelction)?
- Which populations communicate with each other? $\mapsto$ Which topology?

Also known as 'coarse-grain' parallel EA or 'Island Model' EA.

Advanced Techniques: Self Adaptation

Finding good parameters for the population size μ, the mutation rate p_m, the recombination rate p_r and the parameters controlling the selective pressure (e.g. number of replaced individuals λ) often is a hard task in itself.

Moreover, the optimum of the combination of these parameters may vary over the course of the evolutionary process.

Idea of self-adaptation: some of the parameters of the evolutionary process are made subject to optimization themselves; the evolutionary process evolves its own parameters!

- Self-adaptation is a standard method in Evolution Strategies
- Parameters are included in the representation of the individuals:

$$x = (\underbrace{x_1, x_2, \dots, x_n}_{\text{solution}},\ \underbrace{p_1, p_2, \dots, p_k}_{\text{parameters}})$$

- Most often: parameters regarding the mutation (Evolution Strategies)

Section 3: Representations

- Genetic Algorithms (bit strings)
- Evolution Strategies (real-valued vectors)
- Genetic Programming (trees, programms)
- Neuroevolution (neural networks)

Genetic Algorithms

Pioneered by John Holland and David E. Goldberg in the 1970s. Classic book:
J. Holland, *Adaptation in Natural and Artificial Systems*,
The MIT Press; Reprint edition 1992 (originally published in 1975).

We will discuss A) 'simple' Genetic Algorithms that exclusively use bit-strings and B) a variant for evolving permutations (like in the TSP discussed earlier).

Classic Simple Genetic Algorithms:

Represtation:	Bit-strings
Recombination:	1-Point crossover
Mutation:	Bit flip
Parent selection:	Fitness proportional
Survival selection:	Generational

Representation

Genotype: Bit-string with $b_i \in \{0, 1\}$:

$$\boxed{b_1}\boxed{b_2}\boxed{b_3}\boxed{b_4}\cdots\boxed{b_n} \quad \text{e.g.} \quad \boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{1}$$

Length and interpretation as phenotype depend on application.

Representing numbers in binary is problematic. For example:

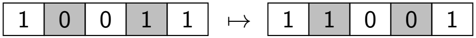
- Hamming distance between 7 (0111) and 6 (0110) is smaller than that from 7 to 8 (1000)
- Thus, changing a 7 to a 6 is more likely than changing it to an 8
- Usually, we would want similar chances for both 6 and 8

Gray coding ensures that consecutive integers have Hamming distance one:

- the 1st (most significant) bit of the gray code is the same as the binary
- the i-th ($i > 1$) bit g_i is the result of $XOR(b_{i-1}, b_i)$
- Examples: $3 : 011 \mapsto 010$, $4 : 100 \mapsto 110$, $5 : 101 \mapsto 111$

Mutation

Each bit gets flipped with a small probability p_m (mutation rate). For example:



The expected number of flipped bits for an encoding of length L is $L \cdot p_m$.

Good mutation rates depend on the application and the desired outcome (good population vs. one highly fit individual).

Rule of thumb: choose mutation rate such that in average one bit per generation to one bit per offspring is changed.

Crossover

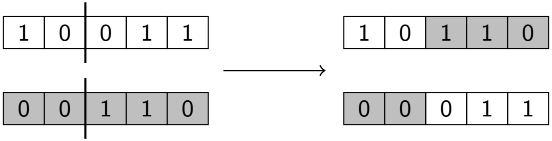
In GA, the recombination operator is considered the primary mechanism for creating diversity, with mutation being only a 'background' search operator.

This operator is the most distinguishing feature of GAs from other global optimization methods.

It's common to apply crossover operators probabilistically with a probability p_r. With probability $1 - p_r$ the parent at hand is copied directly to the offsprings.

One-Point Crossover

Chooses a random integer r from the range $[1, L-1]$, splits both parents in two parts after this position and joins these parts to form the two offsprings:

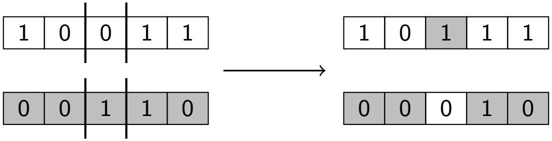


with the splitting position at $r = 2$.

Crossover (continued)

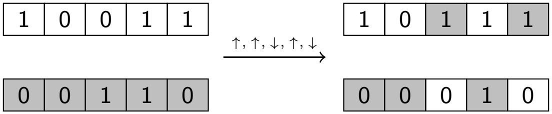
N-Point Crossover

Generalized version of the 1-Point Crossover that chooses N random crossover points from the range $[1, L-1]$, for example N=2:



Uniform Crossover

Treats every bit (gene) separately and decides with the help of a random experiment whether to choose the bit of parent 1 or 2:



Permutation Representations

Many problems naturally take the form of deciding on the order in which a sequence of given events should occur. Examples include job-shop scheduling tasks and the traveling-salesman problem.

Genotype: Sequence of integers (or letters) $e_i \in \{1, 2, 3, \dots, L\}$:

e_1	e_2	e_3	e_4	$\cdots$	e_L

e.g.

4	2	3	1	5

Phenotype: Sequence of nodes to visit / jobs to execute.
Length L equals the number of nodes to visit / events to schedule.

Some of these problems may be order-based—that is, in job-shop scheduling, the order of execution is important—and some may not—e.g. the routes $4, 3, 1, 2$ and $2, 1, 3, 4$ in a TSP are equivalent and have the same fitness.

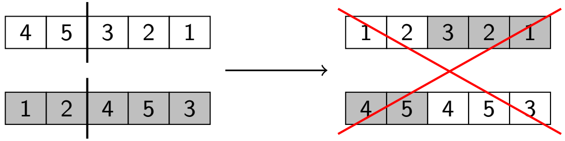
Mutation Operators

Legal mutations are limited to moving values around in the genome:

Swap Mutation:	Swap the values at two randomly selected positions
Insert Mutation:	Move a randomly selected value to a random position
Scramble Mutation:	Scramble the positions of a randomly selected sub-string

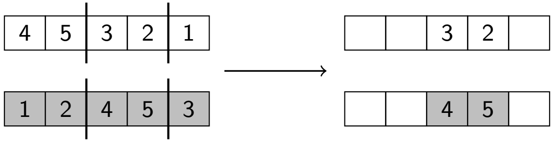
Permutation Representations (continued)

Problem of crossing over: (both) offsprings need to be valid permutations.



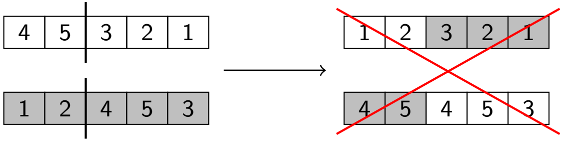
Order Crossover

Selects two crossover points, copies the values between these two points from one parent to the offspring



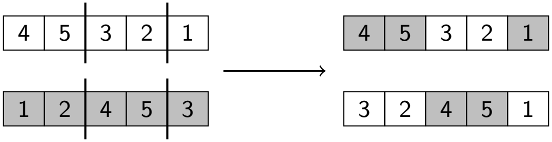
Permutation Representations (continued)

Problem of crossing over: (both) offsprings need to be valid permutations.



Order Crossover

Selects two crossover points, copies the values between these two points from one parent to the offspring and then copies the remaining values to the empty spots in the order they appear in the other parent (starting at the second crossover point and wrapping-around at the end of the string).

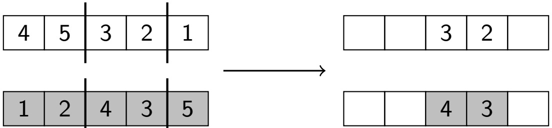


Permutation Representations (continued)

Partially Mapped Crossover (PMX)

Partially Mapped Crossover is the most widely used operator in adjacency-type problems (like the TSP). It works as follows (considering the first offspring):

1. Choose two crossover points at random and copy the values between them from the first parent (P1) to the offspring.

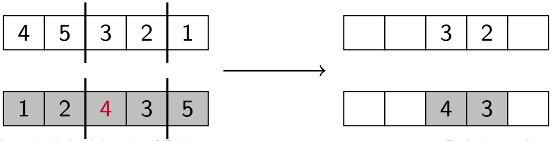


Permutation Representations (continued)

Partially Mapped Crossover (PMX)

Partially Mapped Crossover is the most widely used operator in adjacency-type problems (like the TSP). It works as follows (considering the first offspring):

1. Choose two crossover points at random and copy the values between them from the first parent (P1) to the offspring.
2. Starting from the first crossover point look for elements *i* in that segment of the other parent (P2) that have not been copied ('4' in P2).

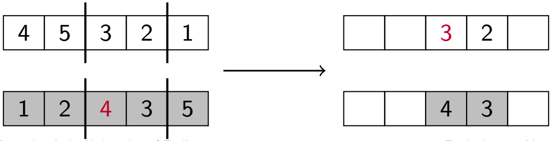


Permutation Representations (continued)

Partially Mapped Crossover (PMX)

Partially Mapped Crossover is the most widely used operator in adjacency-type problems (like the TSP). It works as follows (considering the first offspring):

1. Choose two crossover points at random and copy the values between them from the first parent (P1) to the offspring.
2. Starting from the first crossover point look for elements *i* in that segment of the other parent (P2) that have not been copied ('4' in P2).
3. For each of these elements *i*, look in the offspring to see what element *j* has been copied in its place from P1.

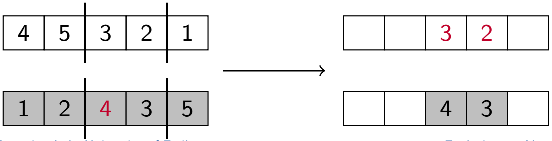


Permutation Representations (continued)

Partially Mapped Crossover (PMX)

Partially Mapped Crossover is the most widely used operator in adjacency-type problems (like the TSP). It works as follows (considering the first offspring):

1. Choose two crossover points at random and copy the values between them from the first parent (P1) to the offspring.
2. Starting from the first crossover point look for elements *i* in that segment of the other parent (P2) that have not been copied ('4' in P2).
3. For each of these elements *i*, look in the offspring to see what element *j* has been copied in its place from P1.
4. Place *i* into the position that was occupied by element *j* in P2.

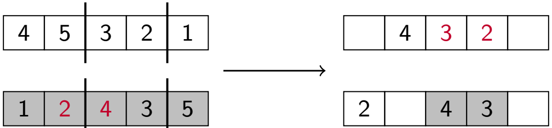


Permutation Representations (continued)

Partially Mapped Crossover (PMX)

Partially Mapped Crossover is the most widely used operator in adjacency-type problems (like the TSP). It works as follows (considering the first offspring):

1. Choose two crossover points at random and copy the values between them from the first parent (P1) to the offspring.
2. Starting from the first crossover point look for elements *i* in that segment of the other parent (P2) that have not been copied ('4' in P2).
3. For each of these elements *i*, look in the offspring to see what element *j* has been copied in its place from P1.
4. Place *i* into the position that was occupied by element *j* in P2.
5. If that place has already been filled by an element *k*, put *i* in the position occupied by *k* in P2.

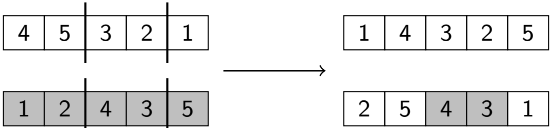


Permutation Representations (continued)

Partially Mapped Crossover (PMX)

Partially Mapped Crossover is the most widely used operator in adjacency-type problems (like the TSP). It works as follows (considering the first offspring):

1. Choose two crossover points at random and copy the values between them from the first parent (P1) to the offspring.
2. Starting from the first crossover point look for elements *i* in that segment of the other parent (P2) that have not been copied ('4' in P2).
3. For each of these elements *i*, look in the offspring to see what element *j* has been copied in its place from P1.
4. Place *i* into the position that was occupied by element *j* in P2.
5. If that place has already been filled by an element *k*, put *i* in the position occupied by *k* in P2.
6. The rest of the offspring can be filled from P2.



Permutation Representations (continued)

Edge Crossover

Edge Crossover is based on the idea of using only edges in the offspring that are present in at least one of the parents.

Edge-3 Crossover produces 1 offspring from 2 parents and works as follows:

1. Construct an edge table.
2. Copy a randomly chosen element to the offspring as 'current element'
3. Remove all references to this element from the table
4. Choose an edge from the list of edges for the current element:
   - If there is a common edge (+), pick that to be the next element
   - Otherwise pick the entry which itself has the shortest list
   - Ties are split at random
5. Continue with the new element with step 3 of the algorithm. In the case of reaching an empty list, the other end of the offspring is examined for possible extensions; otherwise continue with step 2.

Permutation Representations (continued)

Edge Table

For each possible value list the values of this value's neighbors in both parents (thus, up to four entries). If an edge is present in both neighbors, mark it with a '+' as a common edge (these should be preferred).

4	5	3	2	1

1	2	3	4	5

element	edge
1	4, 5, 2+
2	1+, 3+
3	2+, 4, 5
4	5+, 1, 3
5	4+, 1, 3

Schemata

The distinguishing property of GAs is the usage of a crossover operator.

Question: Is there some theoretical justification for using this type of operators? Is it better than just using mutation (random-search)? Can we decide, whether or not a given representation and operators do actually ‘work‘?

Answer: There’s some positive evidence (not a proof) with the Schema Theory.

Definition: Schema (for bit-strings)

A Schema H is simply a hyperplane in the search space. For binary alphabets, they’re usually defined with the help of a third literal: the '#' symbol (AE: pound / number sign, BE: hash) for denoting ‘don’t care’.

The schema $\begin{array}{|c|c|c|c|c|}\hline 1 & 1 & \# & \# & \# \\ \hline\end{array}$ (from here on we’ll simply write ‘11###’) stands for the hyperplane defined by having ones in the first two positions and either a one or a zero in each of the third to fifth position.

All strings x meeting this criterion (e.g. $x = 11101$) are called instances $x \in H$ of this schema H. The given schema, for example, has 2^3 different instances.

Schemata (continued)

Definition: Order and (Defining) Length of a Schema

The order $o(H)$ of a schema H is defined as the number of places where the schema does not have the '$\#$' symbol.
Examples: $o(1\#1\#0) = 3$, $o(0\#\#\#1) = 2$, $o(11\#\#\#) = 2$

The defining length $d(H)$ of a schema H is defined as the distance between the outermost defined positions (defined: $H_i \in \{0, 1\}$, not 'don't care').
Examples: $d(1\#1\#0) = 5 - 1 = 4$, $d(0\#\#\#1) = 5 - 1 = 4$,
$d(11\#\#\#) = 2 - 1 = 1$

Definition: Fitness of a Schema

The fitness $f(H)$ of a schema is defined as the mean fitness

$$f(H) = \frac{\sum_{x \in H} f(x)}{|H|}$$

of its instances $x \in H$. If there are a lot of different instances, the fitness $f(H)$ is often estimated from samples.

Schemata (continued)

Definitions Regarding a Population P

- $m(H, P)$ is the percentage of individuals x in population P that are instances of schema H
- $\hat{f}(H, P)$ is the mean fitness of the individuals x in population P which are instances of schema H (this can be interpreted as an estimate of $f(H)$)
- $f(P) = \frac{\sum_{x \in P} f(x)}{|P|}$ is the mean fitness of all individuals in population P

Global optimization can be described as searching for the schema with zero ‘don’t care’ symbols that has the maximum possible fitness.

As a particular individual can be an instance of many schemata, GAs ‘explore’ many schemata (a multiple of the population size) at the same time.

A string of length L is an instance of 2^L schemata. According to a result of Holland, a population of size n will ‘usefully’ process $\mathcal{O}(n^3)$ schemata at the same time.

Analyzing Operators with Schemata

With the help of these definitions, we can try to analyze the presence and frequency of particular schemata in an evolving population.

Using fitness proportional selection, the probability $P_s(H)$ of selecting a schema H as a parent given a population P of size n is

$$P_s(H) = m(H,P) \cdot \frac{\hat{f}(H,P)}{f(P)} \, .$$

This, again, can be seen when considering the roulette wheel metaphor:

- m(H,P) is the fraction $\frac{n(H,P)}{n}$, with n the population size and $n(H,P)$ the number of individuals $x \in P$ that are instances of H
- hence, $\frac{\hat{f}(H,P)}{f(P) \cdot n}$ is the average size of a field of the roulette wheel that represents one instance $x \in H$ that is in the population P
- in total, there are $n(H,P)$ fields on the roulette wheel representing instances of the schema H, so the total size is $n(H,P) \cdot \frac{\hat{f}(H,P)}{f(P) \cdot n} = P_s(H)$

Analyzing Operators with Schemata (continued)

While the selection operators only change the relative frequencies of examples of pre-existing schemata in an evolving population, mutation and crossover operators can both create new examples and destroy current examples.

The probability $P_{1X}(H)$ of destroying an instance $x \in H$ of a schema H when applying the 1-Point Crossover operator (1X) is the probability of selecting a crossover point that falls between the 'ends' of the schema[1]:

$$P_{1X}(H) = \frac{d(H)}{L-1} \; .$$

The probability $P_m(H)$ of destroying an instance $x \in H$ of a schema H when applying the bitwise mutation operator with mutation rate p_m depends on the order $o(H)$ of the schema:

$$P_m(H) = 1 - (1 - p_m)^{o(H)} \; .$$

[1] Speaking precisely, this is an upper bound on the actual probability as a crossover operation, by coincidence, could create an offspring that is also an instance of the same schema.

Schema Theorem

Putting these results together, we obtain Holland's schema theorem:

$$E\left[m(H,P_{i+1})\right] \geq m(H,P_i)\cdot\frac{\hat{f}(H,P_i)}{f(P_i)}\cdot\left[1-\left(p_r\cdot\frac{d(H)}{L-1}\right)\right]\cdot\left[(1-p_m)^{o(H)}\right]$$

This theorem relates the frequency—more precisely: the frequency's expectation—of instances of a schema H in a population P_{i+1} to its frequency in previous generations:

Schema Theorem

Putting these results together, we obtain Holland's schema theorem:

$$E\left[m(H, P_{i+1})\right] \geq m(H, P_i) \cdot \frac{\hat{f}(H, P_i)}{f(P_i)} \cdot \left[1 - \left(p_r \cdot \frac{d(H)}{L-1}\right)\right] \cdot \left[(1 - p_m)^{o(H)}\right]$$

This theorem relates the frequency—more precisely: the frequency's expectation—of instances of a schema H in a population P_{i+1} to its frequency in previous generations:

- the first term is the probability of schema H getting selected (note that we use the observed fitness $\hat{f}(H, P_i)$ here)

Schema Theorem

Putting these results together, we obtain Holland's schema theorem:

$$E\left[m(H, P_{i+1})\right] \geq m(H, P_i) \cdot \frac{\hat{f}(H, P_i)}{f(P_i)} \cdot \left[1 - \left(p_r \cdot \frac{d(H)}{L-1}\right)\right] \cdot \left[(1 - p_m)^{o(H)}\right]$$

This theorem relates the frequency—more precisely: the frequency's expectation—of instances of a schema H in a population P_{i+1} to its frequency in previous generations:

- the first term is the probability of schema H getting selected (note that we use the observed fitness $\hat{f}(H, P_i)$ here)
- the second term is the probability of the schema not being destroyed by recombination $(1 - p_r P_{1X}(H)$ with p_r the probability of applying $X1)$

Schema Theorem

Putting these results together, we obtain Holland's schema theorem:

$$E\left[m(H, P_{i+1})\right] \geq m(H, P_i) \cdot \frac{\hat{f}(H, P_i)}{f(P_i)} \cdot \left[1 - \left(p_r \cdot \frac{d(H)}{L-1}\right)\right] \cdot \left[(1 - p_m)^{o(H)}\right]$$

This theorem relates the frequency—more precisely: the frequency's expectation—of instances of a schema H in a population P_{i+1} to its frequency in previous generations:

- the first term is the probability of schema H getting selected (note that we use the observed fitness $\hat{f}(H, P_i)$ here)
- the second term is the probability of the schema not being destroyed by recombination $(1 - p_r P_{1X}(H)$ with p_r the probability of applying $X1$)
- the third term is the probability of the schema not being destroyed by mutation (because $1 - (1 - (1 - p_m)^{o(H)}) = (1 - p_m)^{o(H)}$)

Schema Theorem

Putting these results together, we obtain Holland's schema theorem:

$$E\left[m(H, P_{i+1})\right] \geq m(H, P_i) \cdot \frac{\hat{f}(H, P_i)}{f(P_i)} \cdot \left[1 - \left(p_r \cdot \frac{d(H)}{L-1}\right)\right] \cdot \left[(1 - p_m)^{o(H)}\right]$$

This theorem relates the frequency—more precisely: the frequency's expectation—of instances of a schema H in a population P_{i+1} to its frequency in previous generations:

- the first term is the probability of schema H getting selected (note that we use the observed fitness $\hat{f}(H, P_i)$ here)
- the second term is the probability of the schema not being destroyed by recombination $(1 - p_r P_{1X}(H)$ with p_r the probability of applying $X1$)
- the third term is the probability of the schema not being destroyed by mutation (because $1 - (1 - (1 - p_m)^{o(H)}) = (1 - p_m)^{o(H)}$)
- the inequality is a result of the fact that the theorem's right hand side does not consider the probability of a new instance of the schema H being created by mutation or crossover

Schema Theorem (continued)

- The original understanding was that a schema H of above-average fitness would increase its number of instances from generation to generation

Schema Theorem (continued)

- The original understanding was that a schema H of above-average fitness would increase its number of instances from generation to generation
- We can derive the conditions for a schema H to increase it's frequency directly from the theorem:

$$\frac{\hat{f}(H, P_i)}{f(P_i)} \cdot \left[1 - \left(p_r \cdot \frac{d(H)}{L-1}\right)\right] \cdot \left[(1-p_m)^{o(H)}\right] > 1$$

Schema Theorem (continued)

- The original understanding was that a schema H of above-average fitness would increase its number of instances from generation to generation
- We can derive the conditions for a schema H to increase it's frequency directly from the theorem:

$$\frac{\hat{f}(H, P_i)}{f(P_i)} \cdot \left[1 - \left(p_r \cdot \frac{d(H)}{L-1}\right)\right] \cdot \left[(1-p_m)^{o(H)}\right] > 1$$

Also considering effects of crossover and mutation, this tells us that schemata of above-average fitness which are short and have only few defined values, are likely to reproduce.

Schema Theorem (continued)

- The original understanding was that a schema H of above-average fitness would increase its number of instances from generation to generation
- We can derive the conditions for a schema H to increase it's frequency directly from the theorem:

$$\frac{\hat{f}(H,P_i)}{f(P_i)} \cdot \left[1 - \left(p_r \cdot \frac{d(H)}{L-1}\right)\right] \cdot \left[(1-p_m)^{o(H)}\right] > 1$$

Also considering effects of crossover and mutation, this tells us that schemata of above-average fitness which are short and have only few defined values, are likely to reproduce.

Thus, GAs seem well adapted for optimizing functions that are described by short building blocks, which independently of each other determine high fitness values. GAs would explore them in parallel and slowly increase their frequency.

However, the estimation only hints in this direction, but it's not a formal proof.

Evolution Strategies

Pioneered by Schwefel and Rechenberg in the late 1960s.

Evolution Strategies:

Represtation:	Real-valued vectors
Recombination:	none / discrete or intermediate recombination
Mutation:	Gaussian perturbation
Parent selection:	Uniform random
Survivor selection:	Generational (μ, λ) or steady-state $(\mu + \lambda)$

Representation

Genotype: Real-valued vector $x = (x_1, ..., x_n) \in R^n$ of n variables:

$$\boxed{x_1}\boxed{x_2}\boxed{x_3}\boxed{x_4}\cdots\boxed{x_n} \quad \text{e.g.} \quad \boxed{1.3}\boxed{-0.5}\boxed{0.32}\boxed{11.23}\boxed{6}$$

Typically, evolutionary strategies are used for continuous parameter optimization, meaning that the problem at hand can be given as an objective function $R^n \mapsto R$. In this case, the genotype space and phenotype space are identical.

Self-Adaptation: In ES, it's common practice to include parameters controlling the mutation directly into each individual.

$$x = (\underbrace{x_1, x_2, \ldots, x_n}_{\text{candidate}}, \underbrace{\sigma_1, \sigma_2, \ldots, \sigma_k}_{\text{step size}}, \underbrace{\alpha_1, \alpha_2, \ldots, \alpha_l}_{\text{`}\textit{interaction}\text{'}})$$

Common mutation parameters include

- either $k = 1$ (one value σ for all variables x_i) or $k = n$ (individual value σ_i for each variable x_i) parameters controlling the mutation step sizes
- (optionally) l parameters controlling the 'interaction' between the step sizes of different variables.

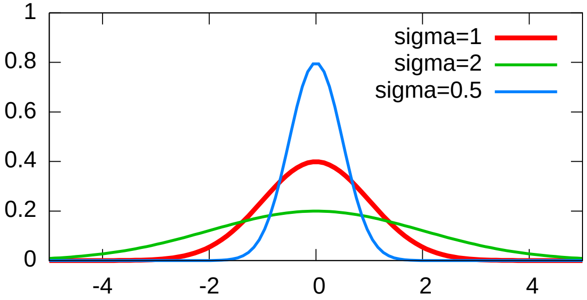
Mutation

Mutation is realized by adding a random number Δx_i to each of the values x_i:

$$x_i' = x_i + \Delta x_i.$$

The delta-terms are usually drawn from a zero-centered normal distribution $\mathcal{N}(0, \sigma)$ with mean $\mu = 0$ and standard deviation σ. In this case, the probability density function is

$$p(\Delta x_i) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(\Delta x_i - \mu)^2}{2\sigma^2}}$$



Mutation (continued)

Regarding the mutation operator, there are two choices to make:

A) whether to use a constant step size σ during the whole evolutionary process or to use a (self-adapted) variable step size

B) whether to use the same step size σ for all variables x_i or to use an individual σ_i for each variable x_i.

Adapting a Step Size with the $\frac{1}{5}$-rule:

- Rechenberg: ideally, $1/5$ of the mutations should be 'successful' (fitter than parents)
- Adapt step size σ every k generations according to measured proportion p_s of successful mutations ($0.8 \leq c \leq 1$):

$$\sigma = \begin{cases} \sigma / c & \text{if } p_s > 1/5 \\ \sigma \cdot c & \text{if } p_s < 1/5 \\ \sigma & \text{if } p_s = 1/5. \end{cases}$$

Mutation (continued)

Adapting a Step Size using Self-Adaptation:

- In the most simple form include one step-size in the individual:

$$(x_1, \ldots, x_n, \sigma)$$

- σ undergoes same variation as the other values x_i
- Important: First mutate σ, then use the new σ' to produce offsprings

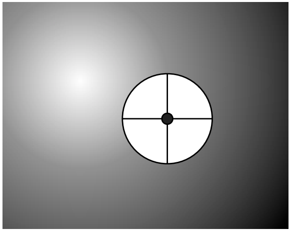
Uncorrelated Mutation with One Step Size:

- Individual $(x_1, \ldots, x_n, \sigma)$ is mutated according to

$$\sigma' = \sigma \cdot e^{\mathcal{N}(0,\tau)}$$

$$x_i' = x_i + \mathcal{N}_i(0, \sigma')$$

- Two parameters: 'learning rate' $\tau \propto 1/\sqrt{n}$ and 'boundary rule': $\sigma' < \epsilon_0 \Rightarrow \sigma' = \epsilon_0$



Mutation (continued)

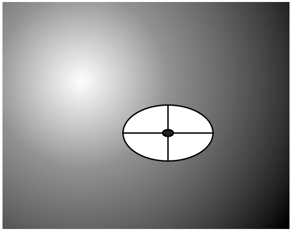
Uncorrelated Mutation with n Step Sizes:

- Observation: fitness landscape can have different slopes in different directions
- Idea: individual $(x_1, \ldots, x_n,\ \sigma_1, \ldots \sigma_n)$ with one individual step size σ_i for each dimension x_i
- Mutation mechanism:

$$\sigma_i' = \sigma_i \cdot e^{\mathcal{N}(0,\tau') + \mathcal{N}_i(0,\tau)}$$

$$x_i' = x_i + \mathcal{N}_i(0, \sigma_i')$$

- Three parameters:
  overall change of mutability:
  $\tau' \propto 1/\sqrt{2n}$
  coordinate-wise change: $\tau \propto 1/\sqrt{2\sqrt{n}}$
  and 'boundary rule': $\sigma' < \epsilon_0 \Rightarrow \sigma' = \epsilon_0$



Mutation (continued)

Correlated Mutation:

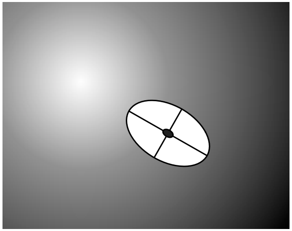
- Observation: ellipses in previous scheme were orthogonal to x-axis
- Idea: individual $(x_1, \ldots, x_n,\ \sigma_1, \ldots \sigma_n,\ \alpha_1, \ldots \alpha_l)$ $(l = n(n-1)/2)$ with correlated step-sizes using a 'covariance' matrix C with

$$\begin{aligned} c_{ii} &= \sigma_i^2 \\ c_{ij,i\neq j} &= 0 \text{ iff i and j are not correlated} \\ c_{ij,i\neq j} &= \frac{1}{2}(\sigma_i^2 - \sigma_j^2)\tan(2\alpha_{ij}) \text{ iff i and j are correlated} \end{aligned}$$

- Mutation mechanism:

$$\begin{aligned} \sigma_i' &= \sigma_i \cdot e^{\mathcal{N}(0,\tau')+\mathcal{N}_i(0,\tau)} \\ \alpha_j' &= \alpha_j + \mathcal{N}_j(0,\beta) \\ x' &= x + \mathcal{N}(0, C') \end{aligned}$$

where $\Delta x = \mathcal{N}(0, C')$ is a vector drawn from a n-dimensional normal distribution with covariance matrix C'.



Recombination

If recombination is used, the most common theme in ES is to combine two randomly selected parents that form one offspring. For the parent vectors x and y, one child z is created:

$$z_i = \begin{cases} (x_i + y_i)/2 & \text{intermediary recombination} \\ x_i \text{ or } y_i \text{ chosen randomly} & \text{discrete recombination} \end{cases}$$

Usually, discrete recombination is used for the variables representing the candidate solution and intermediary recombination is used for the search-strategy part.

The procedure is repeated λ times in order to produce λ offsprings.

Variant: selecting different parents x and y for each variable z_i individually (‘global recombination’).

Selection

Parent Selection

In ES not biased by fitness values, parents are drawn randomly with uniform distribution from the population of μ individuals.

Survivor Selection

After creating λ offsprings, μ survivors are selected:

- (μ, λ) selection: μ survivors from the λ offsprings only
- $(\mu + \lambda)$ selection: μ survivors from the union of parents and offsprings.

In both variants the selection is deterministic and rank based.

Practical Considerations:

(μ, λ) selection is often preferred for the following reasons:

1. replaces all parents, thus able to leave local optima,
2. its better suited to follow non-stationary (moving) optima,
3. in $(\mu + \lambda)$ better candidate solutions dominate better search strategies.

Usually, selection pressure in ES is very high. Common setting: $\lambda = 7 \cdot \mu$

Genetic Programming

Genetic Programming (GP) was developed in the 1990's. Early names: J. Koza. GA: combinatorial optimization. GP: typically machine learning tasks like classification, prediction, etc.

Genetic Programming:

Represtation:	Tree structures
Recombination:	Exchange of subtrees
Mutation:	Random change in tree
Parent selection:	Fitness proportional
Survivor selection:	Generational

Example: Evolve Decision Rule (Classifier) for the Credit Risk Problem

Task: given past experience (examples), learn a classifier

$$f : X_1 \times \ldots \times X_n \mapsto \{\text{good}, \ \text{bad}\}$$

that classifies (new) examples according to their attributes' values.

Idea: evolutionary search in space of all possible classifiers (hypothesis space).

Individuals: represent classifiers (candidate models).

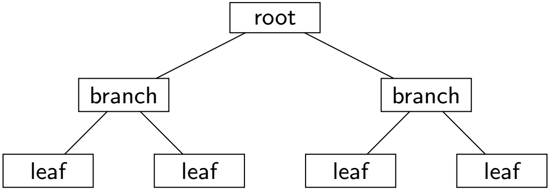
- IF formula THEN *good* ELSE *bad*

Fitness: classification accuracy (training data, validation set) of one particular individual (candidate classifier).

Representation

Genotype: (parse) trees (e.g. for representing the formula)

Three types of nodes: root node, branch node and leaf (node).

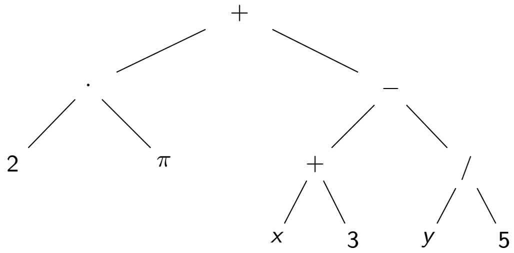


Note: no fixed ‘size’ in GP! Individuals with different size in same population.

Specification of how to represent the individuals by defining the syntax of the trees (or syntax of equivalent symbolic expression).

Syntax definition is usually done by defining a function set (used in branches) and a terminal set (used at leafs).

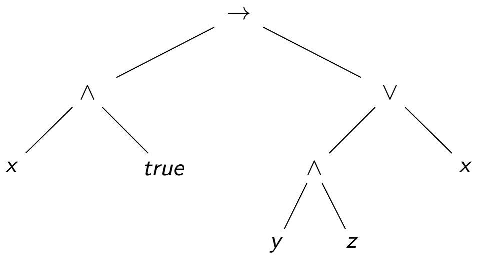
Examle: Representing an Arithmetic Formula



Function set: symbols or functions $(+, -, \cdot, /, =, \exp)$
Terminal set: numbers (e.g. R)

Inorder depth-first tree traversal (expand left child, parent, right child, versus postorder: l child, r child, parent and preorder: parent, l child, r child) yields the represented expression: $((2 \cdot \pi) + ((x + 3) - (y/5))$

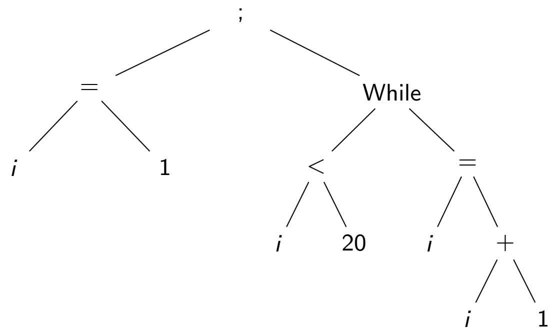
Examle: Representing a Logical Formula



Inorder depth-first tree traversal: $(x \wedge true) \rightarrow (y \wedge z) \vee x$
(obviously, this implication is not true!)

Examle: Representing an Executable Programm

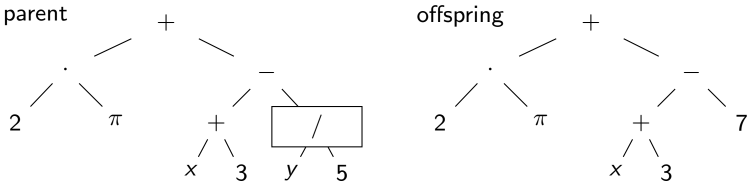
```
$i = 1$;
WHILE ($i < 20$) {
    i = i + 1
}
```



Mutation

Whereas GA and ES apply both, recombination and mutation in two consecutive steps to offsprings, GP usually applies either mutation OR recombination to form an offspring (with probabilistic operator selection).

Standard procedure: randomly select a node and replace the subtree starting at that node by a random generated subtree.



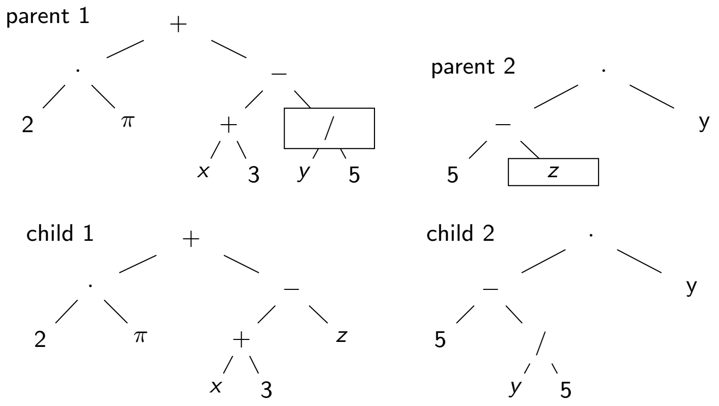
Thus, mutation in GP has two parameters:

- the probability of choosing mutation over recombination
- the probability of choosing the internal node where to place the random generated subtree

In GP, it's common practice to use a very low, but positive 'mutation rate'.

Recombination

In subtree crossover two parents exchange two subtrees starting at two randomly selected nodes.



When applying this binary operator, the depth of the trees may change.

Selection

GP often uses very large population sizes of several thousand individuals.

Parent Selection

Fitness proportional selection is most widely used.

A sometimes used variant is over-selection, where the very best individuals of a population (e.g. the 16% best individuals) have an even (unproportionally) larger probability of being selected.

Survivor Selection

Historically, Genetic Programming was proposed with a generational strategy with no elitism, where the number of produced offsprings matches the population size.

Initialization

Initialization is quite important in Genetic Programming and usually more complex than in ES and GA.

A common strategy is the 'ramped half-and-half' method: Given a maximum initial depth D_{max} create each individual according to one of the two following strategies (chosen with equal probability):

- Full method: Every branch of the tree has full depth. The contents of each node are chosen from the appropriate set, either the terminal set or the function set.
- Grow method: The construction of the tree is started at its root, in each node the contents are chosen stochastically from the union of the function set and terminal set (in depth $d = D_{max}$ from the terminal set only).

Bloat

A practical problem of GP comes with the varying size of the individuals' representation. In practice, without appropriate countermeasures, the average tree size usually grows during the evolutionary process ('survival of the fattest').

Possible countermeasures:

- Introduce a maximum size (depth) and immediately reject larger children.
- Parsimony Pressure: penalty term in fitness function reducing the fitness of larger individuals.

Neuroevolution

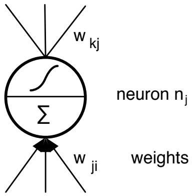
Started in the late 1990's and is still a very active research topic.
Typical area of application: control of dynamical systems, solving reinforcement learning tasks.

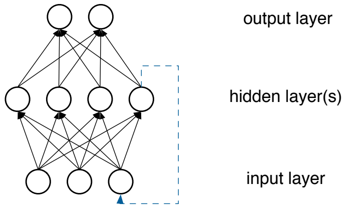
Neuroevolution:

Representation:	(Recurrent) Neural Networks
Recombination:	Exchange of weights or whole subnets
Mutation:	Random change in weights and / or structure of net
Parent selection:	No specific, often fitness proportional
Survivor selection:	No specific, often Steady-State

Representation

Genotype: Recurrent (optional) neural networks. Actual encoding (structured) differs among algorithms (we discuss NEAT).



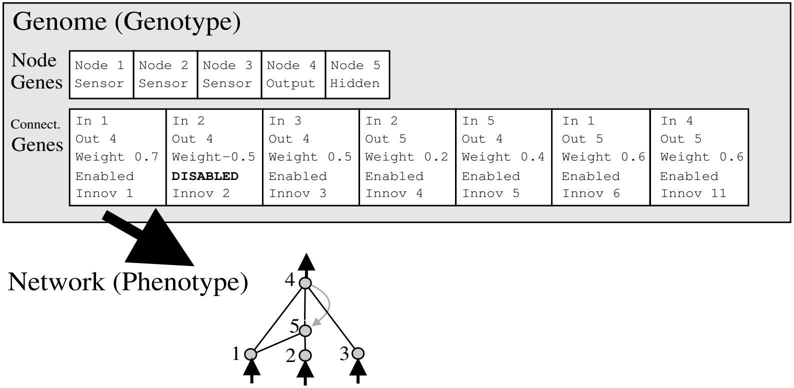


Take-away message NN: 1. Inputs are processed, an output is calculated. 2. Outputs may depend on previous outputs (recurrency). 3. Local processing unit is simple (weighted sum of inputs, non-linear 'activation' function).

Two variants regarding what's subject to evolution:

- network structure is predefined, only weights are modified
- similar to GP, weights and structure are subject to evolution

Representation (continued)



Node Genes

Node 1 Sensor	Node 2 Sensor	Node 3 Sensor	Node 4 Output	Node 5 Hidden

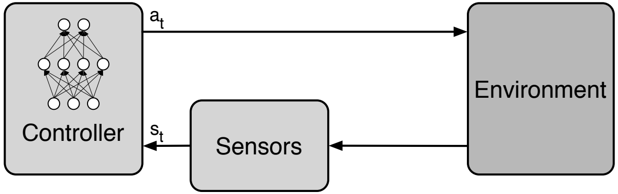
Connect. Genes

In 1	In 2	In 3	In 2	In 5	In 1	In 4
Out 4	Out 4	Out 4	Out 5	Out 4	Out 5	Out 5
Weight 0.7	Weight−0.5	Weight 0.5	Weight 0.2	Weight 0.4	Weight 0.6	Weight 0.6
Enabled	**DISABLED**	Enabled	Enabled	Enabled	Enabled	Enabled
Innov 1	Innov 2	Innov 3	Innov 4	Innov 5	Innov 6	Innov 11

From: Kenneth O. Stanley, *Efficient Evolution of Neural Networks through Complexification*, PhD Thesis, The University of Texas at Austin, 2004

Representation (continued)

Phenotype: how is the net actually used?



- measurement of the present state is applied to the neural net's input layer
- the net processes an output signal that is then used as the next action
- thus, the size of the input layer is predetermined by the measurement's (state signal's) dimension, the output layer matches the action's size

Note: nets are not adapted during testing / application

Representation (continued)

Neural networks are usually trained using a gradient-descent learning algorithm called ‘back-propagation’.

Why is evolving neural networks interesting?

- First, we can not use traditional training algorithms (backpropagation), since controlling a dynamical system is not a supervised learning but a reinforcement learning task
- Nevertheless, we would like to use neural networks, since they have nice properties; especially recurrent nets are interesting for controlling dynamical systems (memory, attracting states, hysteresis)
- But, unfortunately, combining classical training methods from reinforcement learning with traditional ‘training’ of neural networks is known to be problematic (e.g unstable, divergent)

Thus, using a ‘black-box’ search algorithm instead of an alternative ‘learning’ algorithm is worth trying.

Evaluation

The individuals are tested on (dynamical) systems.

Evaluation of whole ‘episodes’, the individual’s fitness reflects its performance.

- no credit assignment, no evaluation of particular ‘decisions’ (actions)

Problem with probabilistic (real-world) systems:

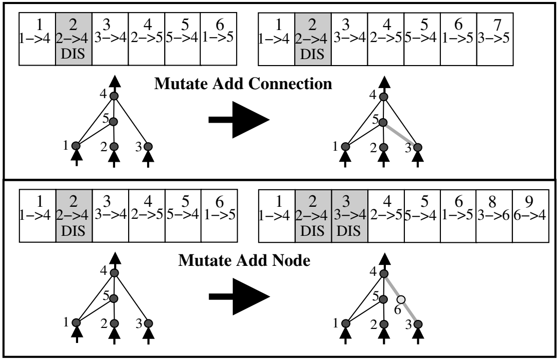
- either repeat the test several times and use the average of the results in order to get a good estimate of the fitness and minimize the variance
- or test only one time and treat fitness itself as a value generated by a random process (e.g. retest all individuals in the next generation)

Further problems arise when different starting states are possible and should be considered for the evaluation.

Mutation

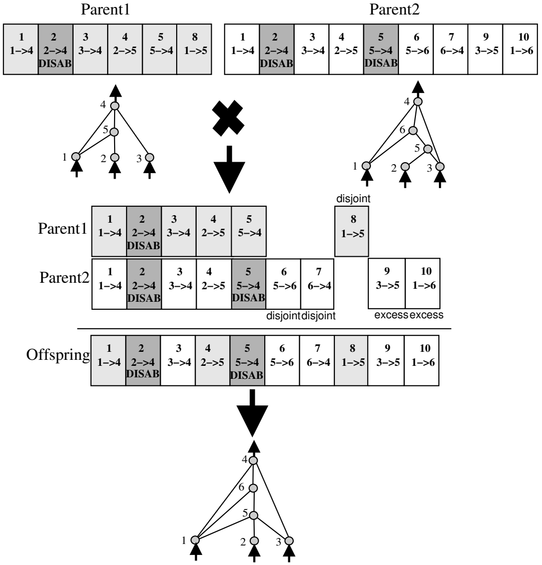
Three types of mutation that are chosen randomly:

1. Random change of weights
2. Addition of a connection between randomly selected neurons
3. Addition of a hidden neuron



From: Kenneth O. Stanley, *Efficient Evolution of Neural Networks through Complexification*, PhD Thesis, The University of Texas at Austin, 2004

Recombination


Parent1
1 1–>4 | 2 2–>4 DISAB | 3 3–>4 | 4 2–>5 | 5 5–>4 | 8 1–>5
Parent2
1 1–>4 | 2 2–>4 DISAB | 3 3–>4 | 4 2–>5 | 5 5–>4 DISAB | 6 5–>6 | 7 6–>4 | 9 3–>5 | 10 1–>6
disjoint
disjoint
disjoint
excess
excess
Offspring
1 1–>4 | 2 2–>4 DISAB | 3 3–>4 | 4 2–>5 | 5 5–>4 DISAB | 6 5–>6 | 7 6–>4 | 8 1–>5 | 9 3–>5 | 10 1–>6
1
2
3
4
5
6

Methods

NEAT - NeuroEvolution of Augmented Topologies

- evolves weights and structure
- grows nets: starts with fully-connected nets with just an input and an output layer (sizes predetermined by problem) and no hidden layers.
- uses 'innovation numbers' and protects new genes
- uses sub-species and fitness sharing
- open source libraries in C++, C#, Java, Python, Delphi, Matlab,...: http://www.cs.ucf.edu/~kstanley/neat.html

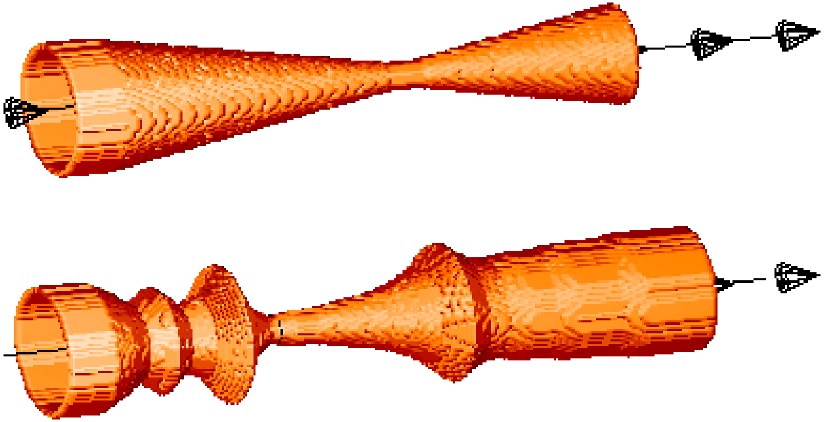
Other Approaches to Neuroevolution:

- SANE and ESP: (older) alternative methods from same group (Risto Miikkulainen)
- ENS^3: another method of about the same age from Frank Pasemann
- EANT(2): newer methods from Siebel & Sommer, outperforming NEAT (at least on some tasks)

Section 4: Applications

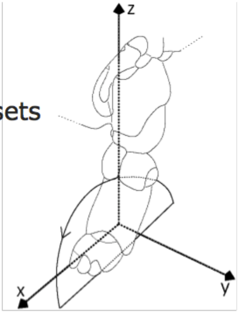
- Jetz Nozzle Optimization
- Gait Pattern Optimization
- Applications of Neuroevolution
- Artificial Life

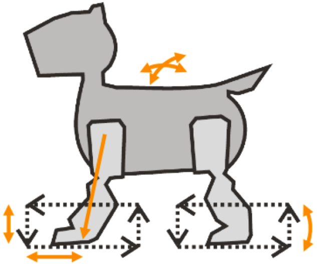
Evolution Strategies: Jet Nozzle Experiment (Schwefel)



Parameter Optimization: Gait-Optimization (Röfer)

- Front/Rear locus
  - x, y, z offsets
  - step height
  - tilt
  - ground, lift, air, and lowering phases
- Step
  - size
  - duration
- Rear to Front offsets
  - x speed ratio
  - phase shift
- Body shift
  - x and y ratios
  - phase offset

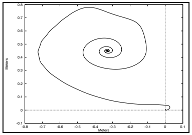




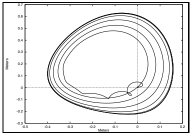
Thomas Röfer. *Evolutionary Gait-Optimization Using a Fitness Function Based on Proprioception.* LNCS, Springer, 2005

Neuroevolution: Pole Balancing (One Example)

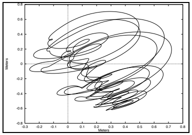
Gomez, Miikukulainen, 2-D Pole Balancing With Recurrent Evolutionary Networks, ICANN, 1998



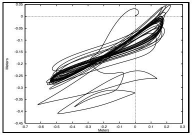
(a)



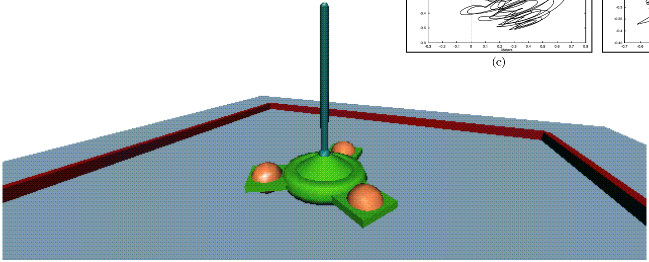
(b)



(c)

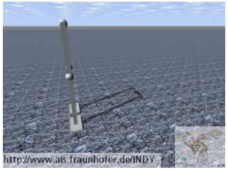


(d)



Neuroevolution: Lab of Frank Pasemann

Coevolved Leg Controllers



Evolved Morphology

Evolved Neural Controller

Neuroevolution: NERO

- Open Source Computer Game build on rtNEAT (by Ken Stanley)
- Started in 2003, first release 2005
- 'Train' (evolve) an army of AI-soldiers by setting objectves (modifying the fitness function) and creating training situations (modifying the environment)
- http://www.nerogame.org/ (Windows, Linux, Mac)

Other game based on NEAT: Galactic Arms Race

Artificial Life

Karl Sims, http://www.karlsims.com/

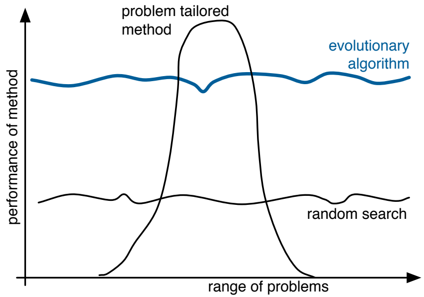
Section 5: Discussion

- Summary
- No-Free-Lunch Theorem
- Bottom Line

Summary

- Evolutionary Algorithms offers a whole tool box of different methods and representations
- We have discussed Genetic Algorithms (binary), Evolution Strategies (real-valued), Genetic Programming (Trees) and Neuroevolution (ANN)
- These methods have been applied very successfully to both combinatorial optimization and classical machine learning problems
- EA is a black box search method and can be tried when there is no better, domain specific method available

80's view of EA performance (after Goldberg)



No-Free-Lunch Theorem

Question: Is an evolutionary algorithm really better than random search (or any other comparable method)?

No-Free-Lunch Theorem by Wolpert and Macready:

- if we average over the space of all possible problems
- then all nonrevisiting[2] black box algorithms will exhibit the same performance.

So, no specific black box algorithm is better than any other black box algorithm 'in the general case'.

Thus, the 80's view (hope) of EA's performance just isn't right.

[2] Nonrevisiting means the algorithm does not visit and test the same point in the search space twice (can be realized in EAs using a memory of examined individuals).

Bottom Line

Where to use EAs

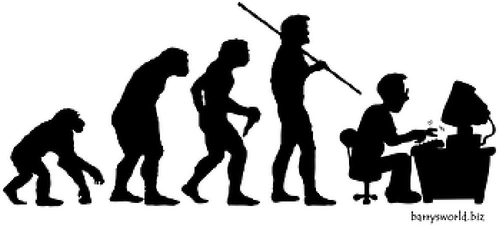
- EA's are a solid choice in situations where
  - EA's are just good enough to do the job — easy to set-up, good results
  - there's no domain specific better (e.g. gradient-based) method available.
- You could use an EA as a black box method for a first try on a new problem just to get a first impression before you start further analyzing it.
- EA's are often used as a Meta-Optimization of the parameters of an ('inner') machine learning algorithm.

Performance of EAs

- You can improve your EA's performance and circumvent the No-Free-Lunch Theorem by introducing domain specific knowledge
- This comes at the cost of a worsened performance on other problems.

Setting up EAs

- Good parameters depend on the problem and representation at hand.
- Finding good parameters for an EA is often more an art than a science.
- Build on experience and start with the rules of thumb given in the slides.


barrysworld.biz