Evolutionary Computation

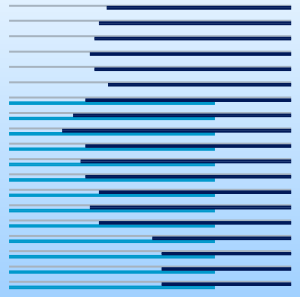
Machine Learning 2019

Part 2

Michael Wand, Jürgen Schmidhuber, Cesare Alippi
TAs: Robert Csordas, Krsto Prorokovic, Xingdong Zou, Francesco Faccio, Louis Kirsch

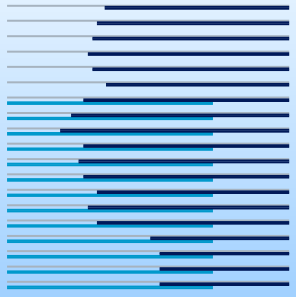
Slides credit: Faustino Gomez, Raoul Malm

Recap so far



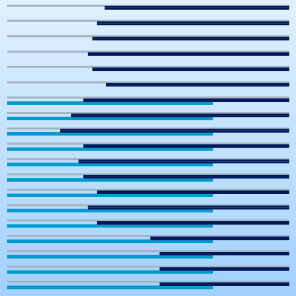
- we introduced evolutionary computing (EC)
  - multiple candidate solutions
  - *evolve* in a way that mimics natural evolution and selection
- we introduced genetic algorithms, one variant of EC
  - *encoding* of candidates in “chromosomes” which get mutated
- powerful, but a bit heuristic??

Recap so far



- we introduced evolutionary computing (EC)
  - multiple candidate solutions
  - *evolve* in a way that mimics natural evolution and selection
- we introduced genetic algorithms, one variant of EC
  - *encoding* of candidates in “chromosomes” which get mutated
- powerful, but a bit heuristic??

- We now consider one famous theorem which may answer this question to some extent: The *Schema* Theorem

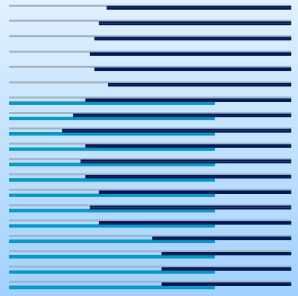


Theory of Genetic Algorithms

The Schema Theorem

Some details taken from
https://www.slideshare.net/AndresMendezVazquez/072-hollands-genetic-algorithmsschematheorem

Preliminaries



Consider a generic GA:

- binary alphabet, fixed length individuals of length *l*
- Fitness proportional selection
- Recombination by single point crossover
- Gene wise mutation

A Schema

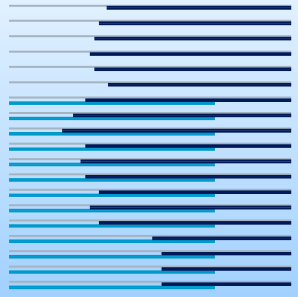
A *schema* is a template that identifies a subset of strings with similarities at certain positions.

Open positions are denoted by *.

Example (length 5): the schema [0 1 * 1 *] generates the individuals

0	1	0	1	0
0	1	0	1	1
0	1	1	1	0
0	1	1	1	1

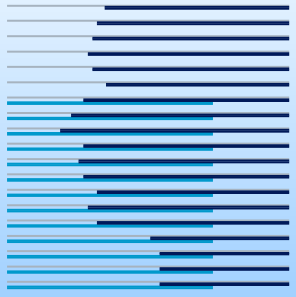
A schema



Schemas have quite different properties:

- A schema like [* 0 * * * * *] has much less information than [0 1 * * 0 0 1].
- A schema can span the entire length of an individual (like [0 1 * * 0 0 1]) or only a part (like [1 0 1 * * * *]).

Schema definitions



The *schema order* is the number of non * genes in the schema.

- Example: o([* * 1 * * 0 1]) = 3

The schema-defining *length* is the distance between the first and the last determined (not *) gene in the schema.

- Example: δ([* * 1 * * 0 1]) = 4

Also remember that the *length l* is the number of positions in the schema.

- The schemas above have length 7.

Probabilities

How likely is it that an individual of schema *H* creates offspring of schema *H*?

- Crossover (1X):

$$P_{\text{disruption}}(H, 1\text{X}) \approx \frac{\delta(H)}{(l-1)}$$

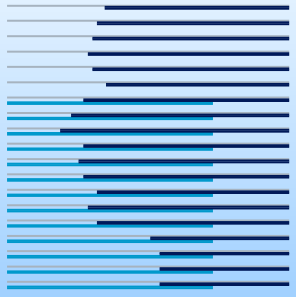
(note that crossover need not necessarily destroy the schema)

- Mutation (P_m = probability of mutation for a single gene)

$$P_{\text{disruption}}(H, \text{mutation}) = 1 - (1 - P_m)^{o(H)} \approx o(H) \cdot P_m$$

the last approximation comes from ignoring higher order terms

Probabilities



How likely is it that an individual *h* selected for mating (proportional fitness) is of schema *H?*

- Depends on the number of parents of schema *H* and of the *average fitness of schema H* relative to the population.

$$P(h \in H) = \frac{\text{Number of individuals matching } H}{\text{Population size}} \cdot \frac{\text{Mean fitness of schema H}}{\text{Mean fitness of population}}$$

$$= \frac{m(H,t) \cdot f(H,t)}{M \cdot \bar{f}(t)}$$

with

$$f(H,t) = \text{average fitness of schema } H \text{ at time } t$$

$$\bar{f}(t) = \text{mean fitness of population at time } t$$

$$m(H,t) = \text{number of individuals of schema } H \text{ at time } t$$

$$M = \text{population size}$$

Schema Theorem

Finally, we get the theorem:

Under fitness proportional selection, the expected number of instances of schema *H* at time *t* is

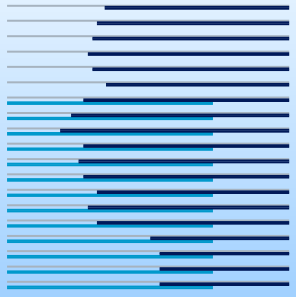
$$E(m(H,t+1)) \geq M \cdot P(h \in H) = \frac{m(H,t) \cdot f(H,t)}{\bar{f}(t)} \cdot (1-p)$$

where

$$p = \frac{\delta(h)}{l-1} P_c + o(H) P_m$$

p is the approximate probability that the schema gets disrupted by mutation or crossover.

Schema theorem

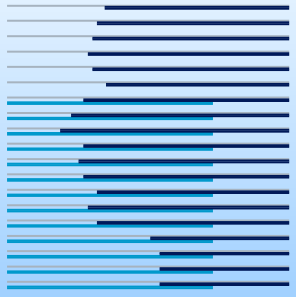


- From the schema theorem we see that the representation of a schema *H* in the population increases when

$$\frac{f(H,t)}{\bar{f}(t)} > \frac{1}{1-p} \approx 1+p$$

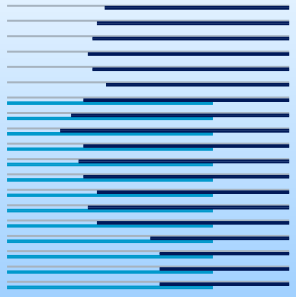
- This means that short-order schemas have a good chance to survive if their fitness is slightly above average
- Longer schemas are more sensitive to disruption and require higher fitness ratios
- The *building block* hypothesis: The GA creates complex solutions with excellent fitness from partial solutions whose fitness just slightly exceeds average

Schema Theorem



- If the interpretation of the Schema theorem holds, it is a powerful theoretical foundation for genetic algorithms
  - "building block hypothesis" - GA construct high-quality solutions out of small components
  - but the interpretation has been criticized (e.g. Altenberg, *The Schema Theorem and Price's Theorem*, 1995)
- It is important to consider the ordering of a Chromosome
  - values which are in close distance should be able to encode some relevant property of the solution

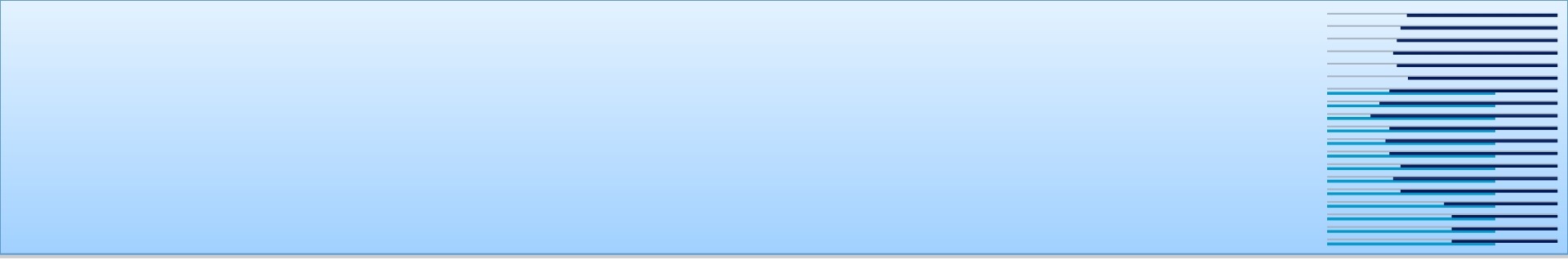
Schema Theorem



Further limitations of the Schema theorem:

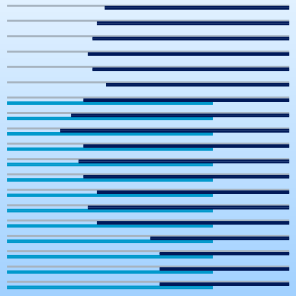
- Does not consider the positive effects of crossover and mutation
- Only considers a single evolution step, makes no statement about converging to a global optimum solution
- Relative fitness of a schema may vary over time
- Only mathematically valid if the population is infinite, or if infinitely many experiments are done
- Does not explain why GA occasionally performs poorly

For further reading see
https://www-cs.stanford.edu/people/nuwans/docs/GA.pdf



Evolution Strategies

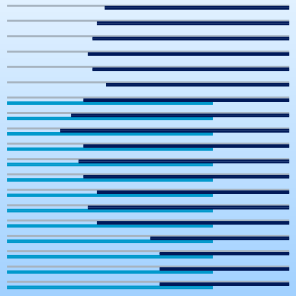
Evolution Strategies



Another variant of Evolutionary Computation

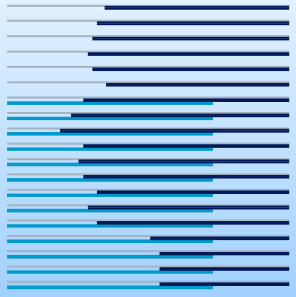
- Genes contain real values
  - Mutation can be done by adding normally distributed values
  - Crossover can be done by averaging of the genes of both parents
  - (but there are other variants)
- Focus on mutation!
- *The mutation parameters also evolve*

ES: basic procedure



1) **Initialize** parents and evaluate them
2) **Create** some offspring by perturbing parents with Gaussian noise according to parent's mutation parameters
3) **Evaluate** offspring
4) **Select** new parents from offspring and possibly old parents
5) If good solution not found Goto 2

ES: Genotype Encoding

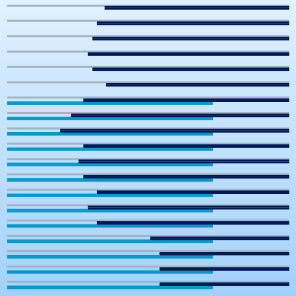


- Chromosomes contain *problem* parameters (as in genetic algorithms) and *strategy* parameters

$$\langle \underbrace{x_1, \ldots, x_n}_{\text{problem parameters}}, \underbrace{\sigma_1, \ldots, \sigma_n}_{\text{strategy parameters}} \rangle$$

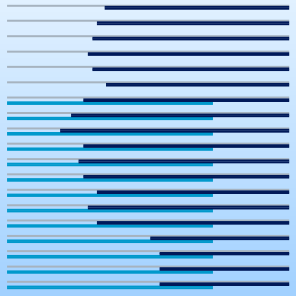
- The strategy parameters determine the amount of mutation (i.e. the standard deviation of the Gaussian from which the mutation is determined) which is applied to the corresponding problem parameter

ES Mutation



- Main mechanism: change value of a gene by adding Gaussian random noise
- $x'_i = x_i + N(0,\sigma)$
- Key idea: σ is part of the chromosome and is also mutated into σ' (see later how)
- thus the mutation step size coevolves with the solution x

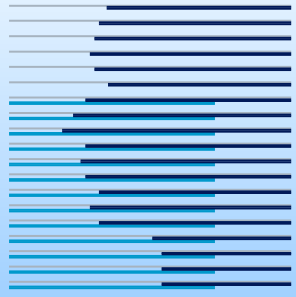
ES: Notation



We describe the metaparameters of ES by the ***(μ+,λ)*-Notation**

- μ is the number of parents (the size of each generation)
- λ is the number of offspring
- “,”: select new parents for next generation *only* from offspring
- “+”: select new parents for next generation from old parent and offspring

(μ+,λ)-Algorithm



1) generate *μ* parents

2) from (all) the parents, select *λ* individuals and mutate them

3) select *μ* new parents from the offspring only (if “,”) or from both old parents and offspring (if “+”)

4) if solution not good enough, go to 1

(if recombination from multiple parents is desired, introduce additional parameter *ρ* – number of parents for each offspring)

Examples:

(1+1)-ES: For each generation, create *one* offspring by mutation, take the better of the two for the next step

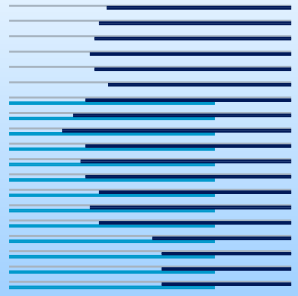
(1,λ)-ES: For each generation, create λ offspring and take the best individual of the offspring as (single) next generation parent

(μ+,λ)-Algorithm

Algorithm 1: $(\mu/\rho \overset{+}{,} \lambda)$ Self-Adaptation Evolution Strategy Algorithm

initialise the first generation $P_\mu^{g=1} = \{\mathbf{a}_1^{g=1}, \mathbf{a}_2^{g=1}, ..., \mathbf{a}_\mu^{g=1}\}$ and $P_\lambda = \{\}$
for *each generation* $g = 2, 3, ..., G$ **do**
  for *each offspring* $(1, 2, ..., \lambda)$ **do**
    select uniform randomly ρ parents from P_μ^{g-1}
    average the selected parents to form the candidate $\mathbf{a} = (\boldsymbol{\theta}, \boldsymbol{\sigma})$
    adapt the mutation parameter $\boldsymbol{\sigma}$ yielding the new value $\boldsymbol{\sigma}'$
    mutate the objective parameter $\boldsymbol{\theta}$ using $\boldsymbol{\sigma}'$ yielding the new value $\boldsymbol{\theta}'$
    add new offspring $(\boldsymbol{\theta}', \boldsymbol{\sigma}')$ to offspring population P_λ
  end
  select μ candidates for next generation P_μ^g via truncated selection from either
    - the offspring population P_λ ("$\mu/\rho, \lambda$" comma selection)
    - the offsprings P_λ and parents P_μ^{g-1} ("$\mu/\rho + \lambda$" plus selection)
  reset offspring population $P_\lambda = \{\}$
end

ES Mutation



- Mutation effect: $\langle\ x, \sigma\ \rangle \rightarrow \langle\ x', \sigma'\ \rangle$
- Order is important:
  - first $\sigma \rightarrow \sigma'$ (see later how)
  - then $x \rightarrow x' = x + N(0,\sigma')$
- Rationale: new $\langle\ x' ,\sigma'\ \rangle$ receives two valuations
  - Primary: x' is good if f(x') is good
  - Secondary: σ' is good if the x' it created is good
- With reversed mutation order this would not work

Uncorrelated mutation, case 1

Uncorrelated mutation with 1 mutation parameter

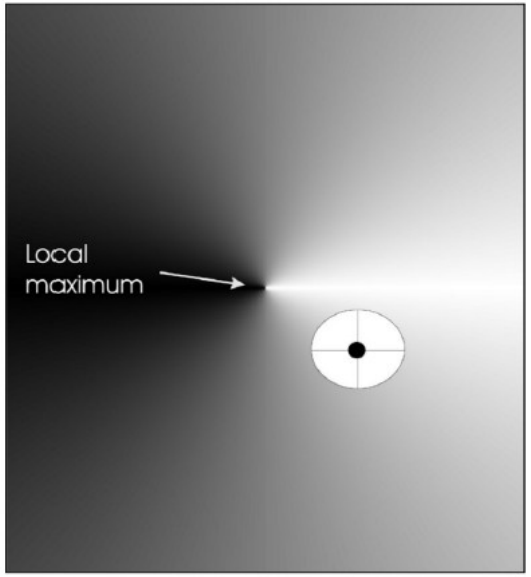
- look at one mutation step of the (parent average) candidate $(\boldsymbol{\theta}, \sigma)$

$$\sigma' = \sigma \exp(\epsilon), \qquad \epsilon \sim N(0, \tau)$$
$$\theta_i' = \theta_i + \epsilon_i, \qquad \epsilon_i \sim N(0, \sigma'),\ i = 1, ..., d$$

- τ is the learning rate, typically $\tau \sim 1/d^{1/2}$
- we impose the boundary rule that if $\sigma' < \epsilon$ then $\sigma' = \epsilon$, where ϵ is some fixed threshold
- changing the mutation strength σ allows for a self-tuning of the mutation strength (σ self-adaptation)

isotropically distributed mutations

- mutants on the circle have the same probability of being created from the parent in the centre



[Eiben, Smith: Introduction to Evolutionary Computation]

Uncorrelated mutation, case 2

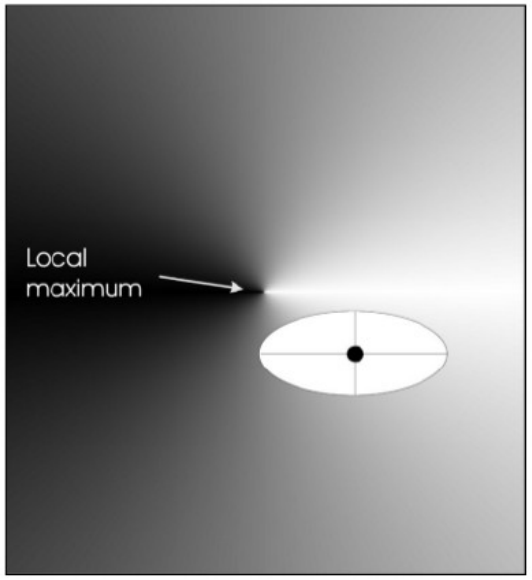
Uncorrelated mutation with d mutation parameters

- look at one mutation step of the (parent average) candidate $(\boldsymbol{\theta}, \boldsymbol{\sigma})$

$$\sigma_i' = \sigma_i \exp(\epsilon + \epsilon_i') \,, \qquad \epsilon \sim N(0, \tau) \,, \epsilon_i' \sim N(0, \tau'),$$
$$\theta_i' = \theta_i + \epsilon_i \,, \qquad \epsilon_i \sim N(0, \sigma_i'),\ i = 1, ..., d$$

- two learning rates
  - overall learning rate $\tau \sim 1/d^{1/2}$
  - coordinate-wise learning rate $\tau' \sim 1/d^{1/4}$
- we impose the boundary rule that if $\sigma' < \epsilon$ then $\sigma' = \epsilon$, where ϵ is some fixed threshold
- probability of mutation varies along the coordinates of the $\boldsymbol{\theta}$ space

mutants on the ellipse have the same probability of being created from the parent in the centre



[Eiben, Smith: Introduction to Evolutionary Computation]

Correlated mutation

Correlated mutation with a $d \times d$ covariance matrix $\mathbf{C}$

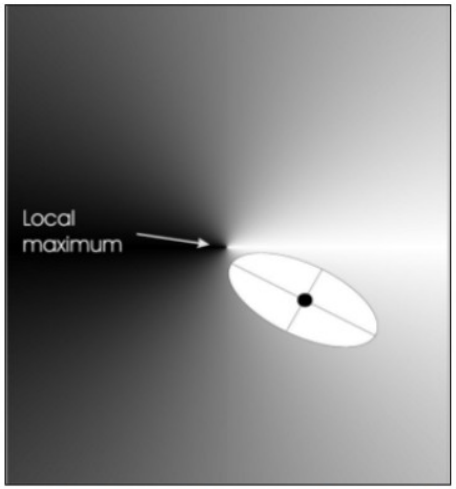
- $\mathbf{C}$ is a real, symmetric, positive semi-definite matrix: $\mathbf{C} = \mathbf{R}\,\mathbf{D}\,\mathbf{R}^T$ with diagonal positive semi-definite matrix $\mathbf{D}$ and orthogonal matrix $\mathbf{R}$
- $\mathbf{C}$ can be parametrised by d standard deviations $\boldsymbol{\sigma} = (\sigma_1, ..., \sigma_d)$ and $k = d(d-1)/2$ rotation angles $\boldsymbol{\alpha} = (\alpha_1, ..., \alpha_k)$
- look at one mutation step for the candidate $(\boldsymbol{\theta}, \boldsymbol{\sigma}, \boldsymbol{\alpha})$

$$\sigma_i' = \sigma_i \exp(\epsilon + \epsilon_i'), \qquad \epsilon \sim N(0, \tau),\ \epsilon_i' \sim N(0, \tau')$$
$$\alpha_i' = \alpha_i + \epsilon_i, \qquad \epsilon_i \sim N(0, \tau''),\ i = 1, ..., d$$

- we impose the same boundary rule as before
- two learning rates as before (τ and τ') and additionally τ''
- create $\mathbf{C}'$ from $\boldsymbol{\sigma}', \boldsymbol{\alpha}'$ and mutate objective parameters

$$\boldsymbol{\theta}' = \boldsymbol{\theta} + \boldsymbol{\epsilon}, \qquad \boldsymbol{\epsilon} \sim N(0, \mathbf{C}')$$

- probability of mutation (variance) can vary along any direction in the $\boldsymbol{\theta}$ space



[Eiben, Smith: Introduction to Evolutionary Computation]

Jet Nozzle Experiment

Task: to optimize the shape of a jet nozzle
Approach: random mutations to shape + selection

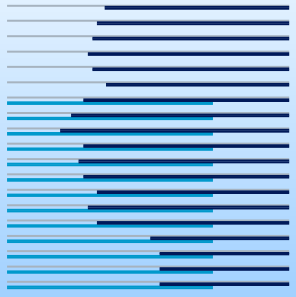
Initial shape

Final shape

Used (1+1)-ES

H.-P. Schwefel

Covariance Matrix Adaptation



- one of the most popular gradient-free optimisation algorithms
- useful in particular on non-convex, non-separable, ill-conditioned, multi-modal or noisy objective functions
- considers the **mean** and **covariance** of the current candidates in the population and adapts them in direction towards the fittest candidates for the next generation
- belongs to the class of ***(μ/μw, λ) ES*** algorithms where all parents are averaged (weighted) before computing offsprings by mutation
- heuristically can be used when the search space dimensionality is less than ~1000

Details and Tutorial: [https://arxiv.org/abs/1604.00772]

Covariance Matrix Adaptation

Algorithm 1: Basic idea behind CMA-ES, see [arXiv:1604.00772] for details

1 initialise mean $\mathbf{m}^{(0)}$ and covariance matrix $\mathbf{C}^{(0)} = \sigma\,\mathbf{1}$

2 **for** *each generation* $g = 1, 2, ..., G$ **do**

3 sample λ offsprings

$$\boldsymbol{\theta}_i^{(g)} \sim \mathbf{m}^{(g-1)} + N(0, \mathbf{C}^{(g-1)}) \quad \text{for} \quad i = 1, 2, ..., \lambda$$

4 select parents via truncated selection and average them

$$\mathbf{m}^{(g)} = \frac{1}{\mu} \sum_{i=1}^{\mu} \boldsymbol{\theta}_{i:\lambda}^{(g)} ,$$

where $\boldsymbol{\theta}_{i:\lambda}^{(g)}$ denotes the i^{th}-fittest offspring of generation g.

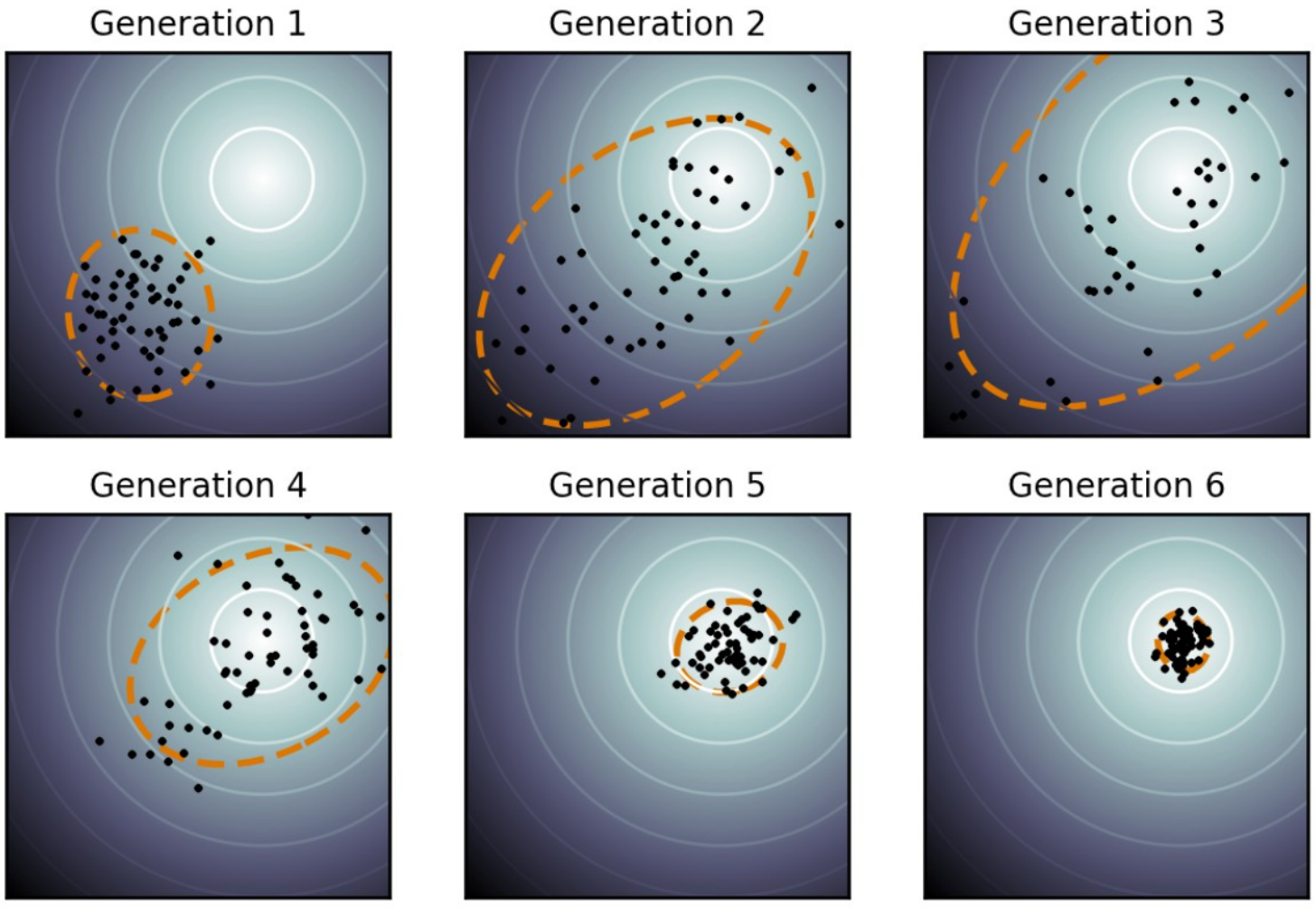
5 adapt the covariance matrix

$$\mathbf{C}^{(g)} = \frac{1}{\mu} \sum_{i=1}^{\mu} (\boldsymbol{\theta}_{i:\lambda}^{(g)} - \mathbf{m}^{(g-1)}) (\boldsymbol{\theta}_{i:\lambda}^{(g)} - \mathbf{m}^{(g-1)})^T$$

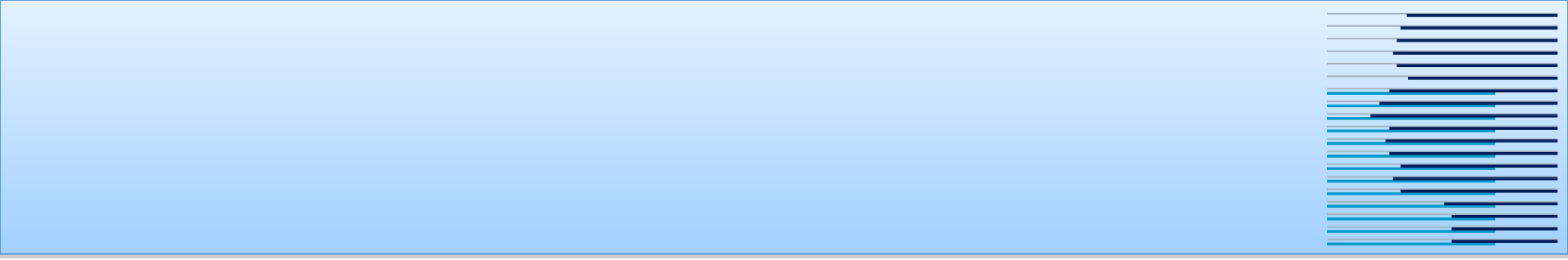
use true mean !!!

6 **end**

Covariance Matrix Adaptation



[https://en.wikipedia.org/wiki/CMA-ES]



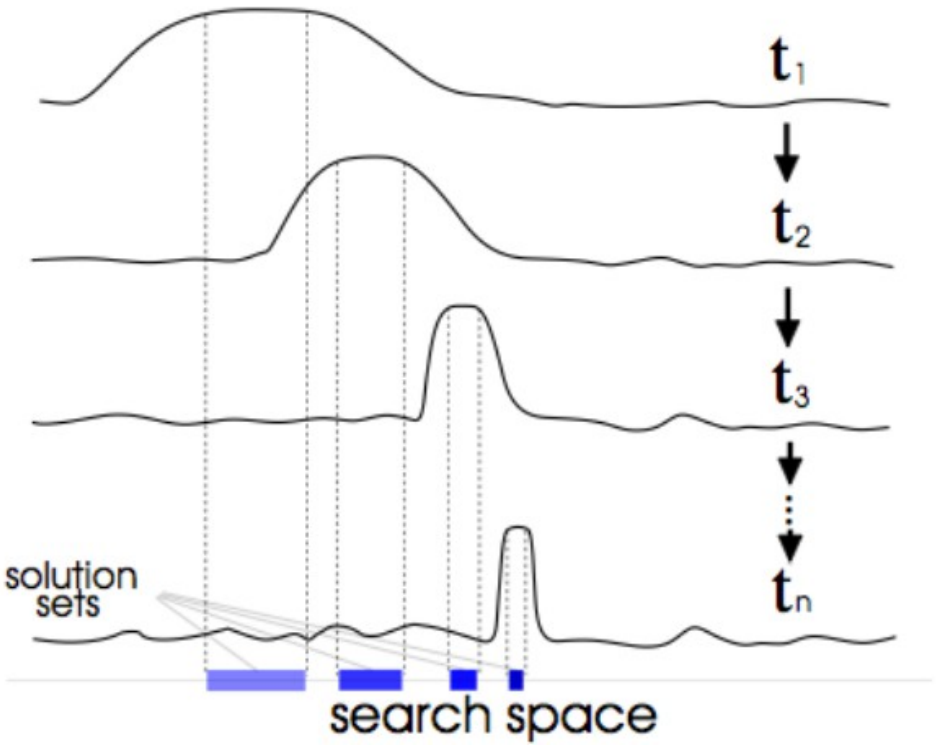
Advanced examples and methods

Incremental Evolution

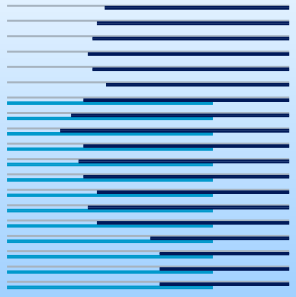
- Decompose target task into a sequence of increasingly difficult tasks

$$t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_{target}$$

- Start evolving on the easiest task
- Once solved, move to next task, continue until main task solved



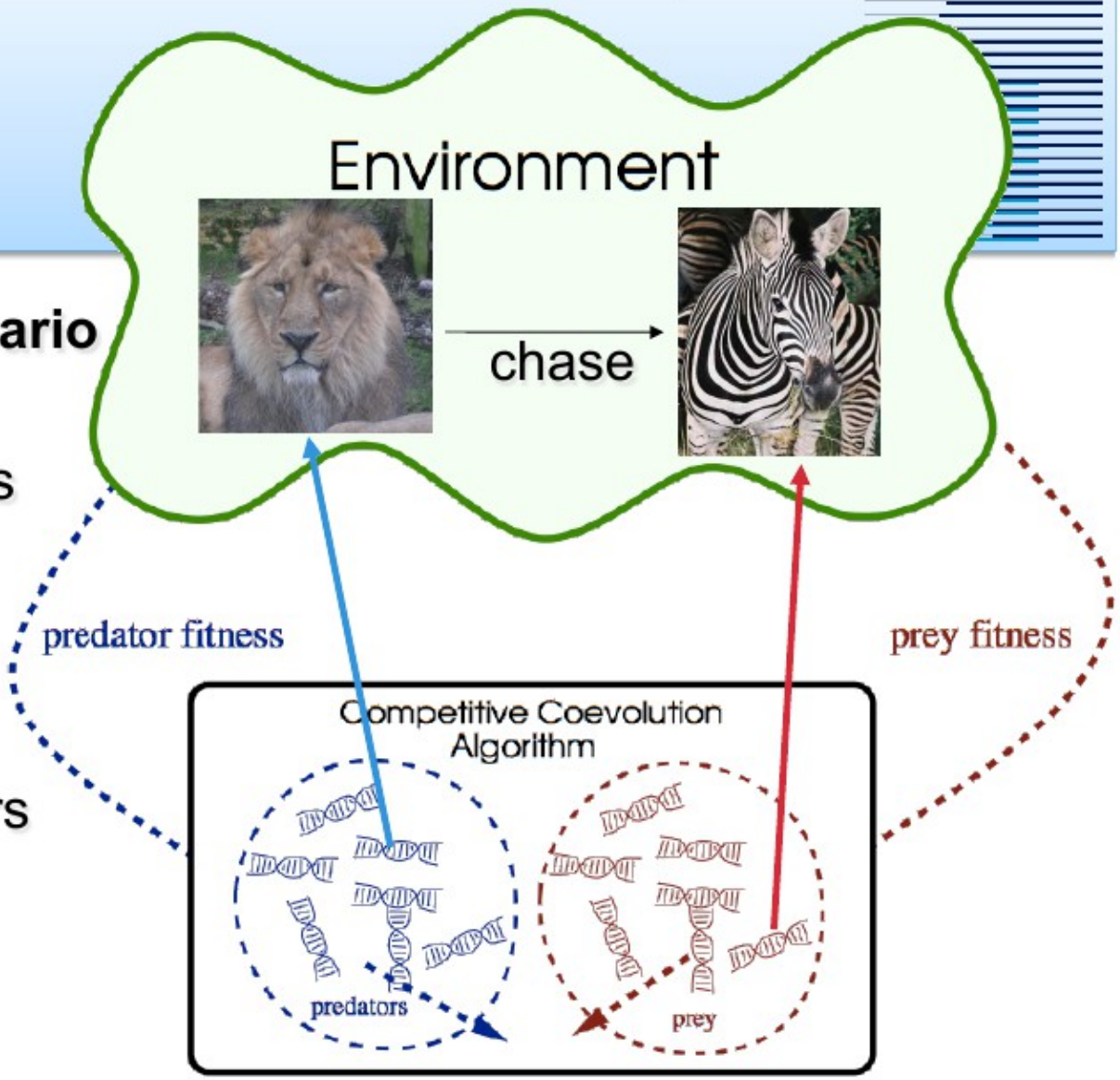
Coevolution



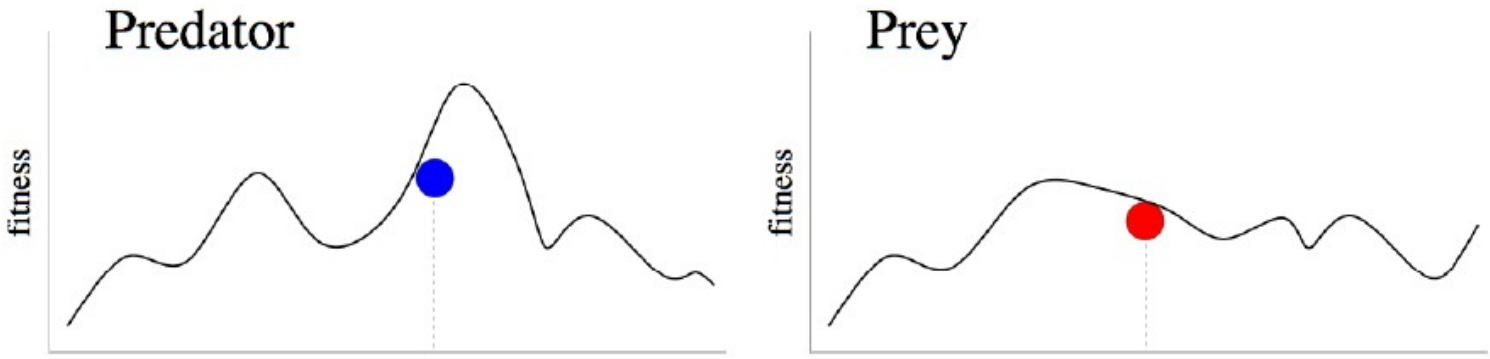
- In nature, different species compete for resources and cooperate to survive
- Standard EC algorithms can be thought of as involving only one species
- Coevolutionary Algorithms have multiple species, usually contained in separate populations
  - species *compete* against each other
- Objective is to trigger progressive improvement of each species against the other
- Useful in tasks where there is no good environment to evolve against (e.g. game playing)

Predator-Prey Scenario

- fitness of predators based on how well they capture prey
- fitness of prey based on how well they evade predators

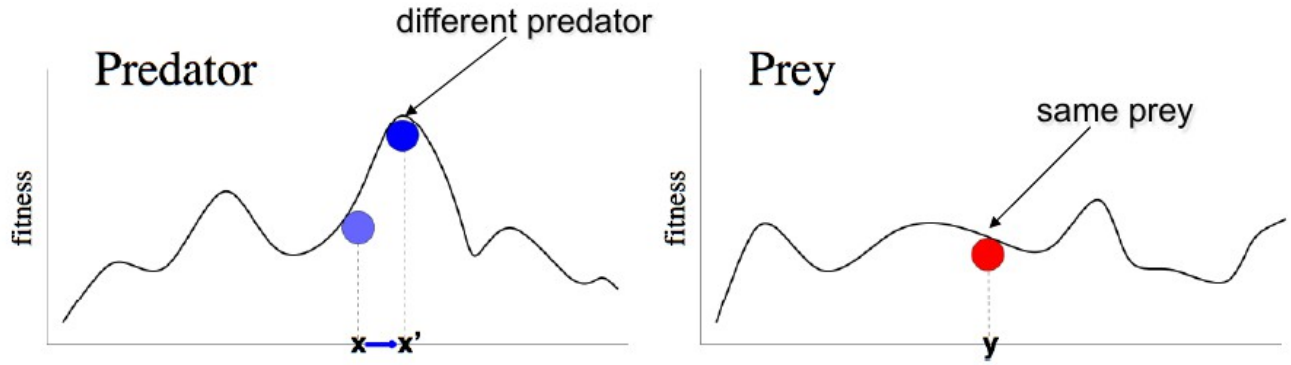


Coupled Fitness Landscapes



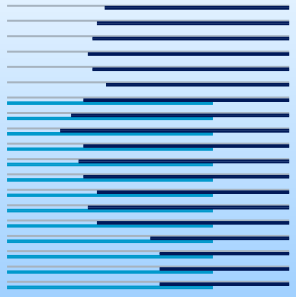
- The fitness of one species is dependent on that of the others
- Each strategy (genotype) in one population changes the fitness landscape for the other species
  - Finding a good strategy is a “moving target”

Coupled Fitness Landscapes



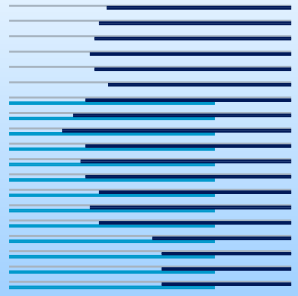
- If, for example, the predator changes, the prey fitness landscape also will change because now the fitness of the prey is measured against a different opponent
  - and vice versa: if the prey learns to adapt (and evade the predator), the predator needs to adapt its behavior

Cooperative Coevolution



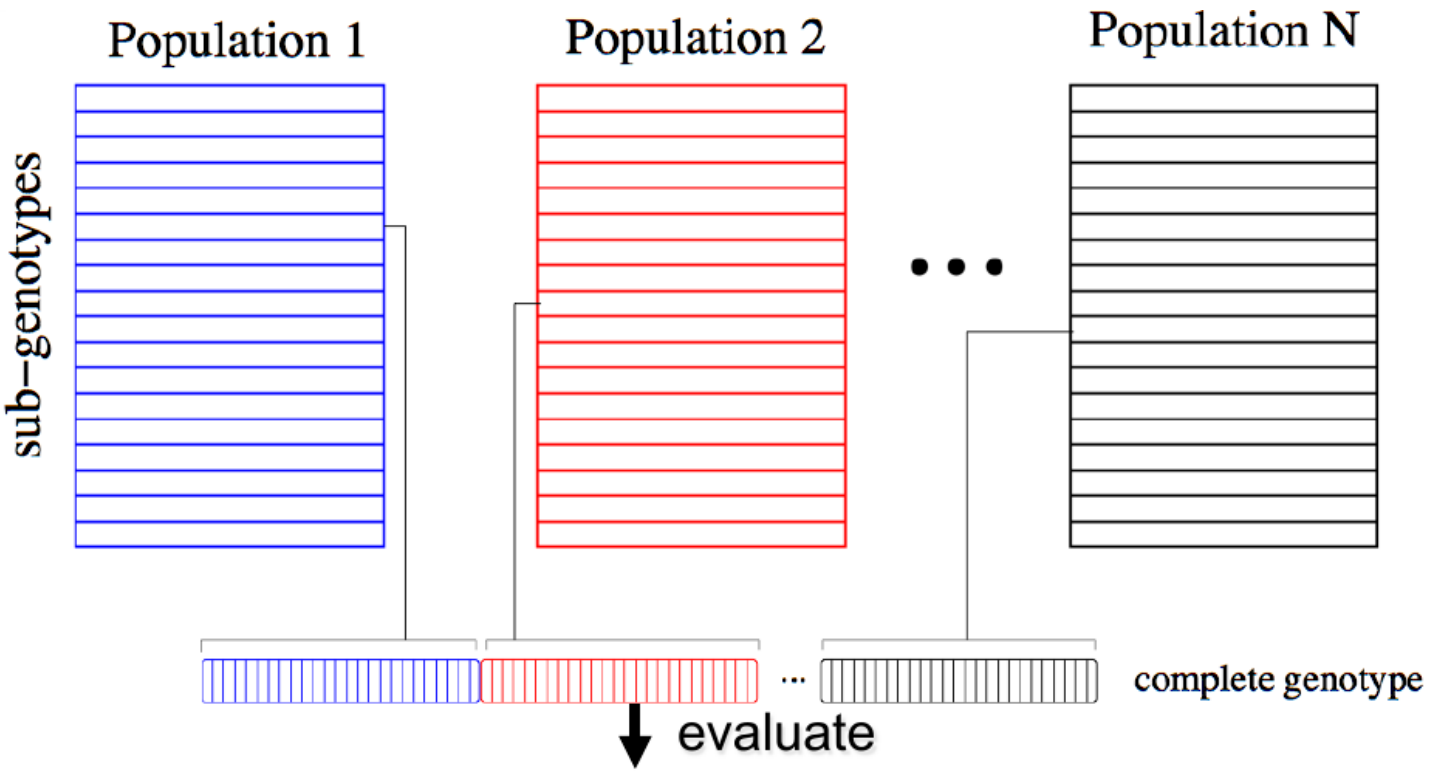
- Unlike most EC algorithms, in Coop Coev, each genotype is a component (sub-genotype) of a complete solution
- Sub-genotype fitness is based on the fitness of the complete solution in which it participates
- Problem is divided into smaller interacting subproblems that are searched semi-independently

Cooperative Coevolution: Issues



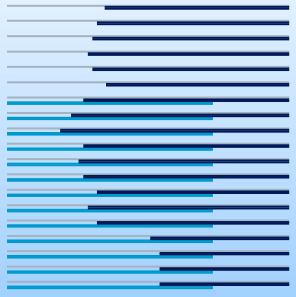
- Difficult to initiate and maintain arms race
- In naïve approach, each population converges to overspecialized solution
  - strategies are not so easy to beat with random mutation: arms race stalls
- Solution: need to maintain family of diverse strategies in each population that cannot all be beaten by some opposing strategy

Cooperative Coevolution



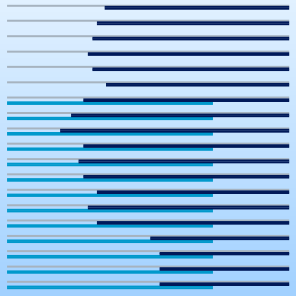
Complete solutions formed by selecting one sub-genotype from each population

Open Questions



- How many trials (“tournament”) should a subgenotype participate in?
  - If species have complex interactions, more trials might help obtain true fitness
- How should the competitors be selected?
  - Should they be selected at random or based on e.g. fitness
- What should be the fitness of the subgenotype given the chosen number of trials?
  - Take the fitness of the best (usually a good idea) or worst trial?
  - Or the average of the trial fitnesses?

NEAT – Evolve neural network topologies



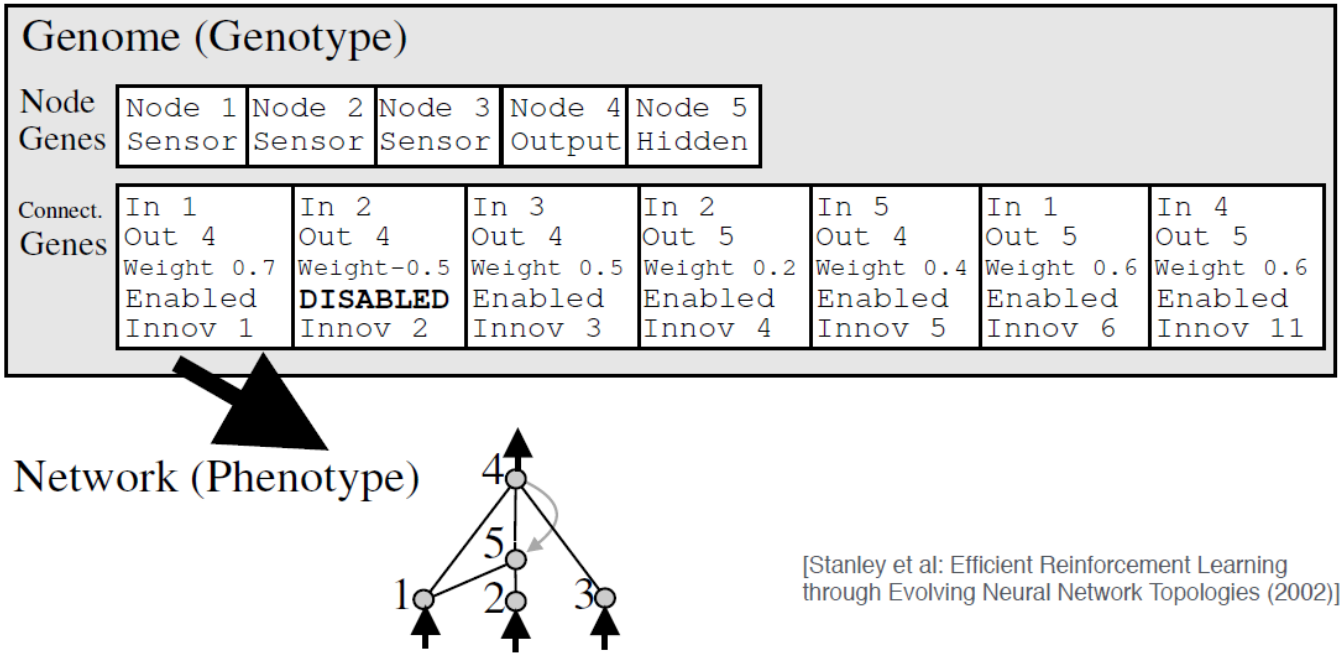
Idea

- evolve neural networks using GA (NeuroEvolution of Augmented Topologies)
- main idea is that it is most effective to start evolution with small, simple networks and allow them to become increasingly complex over generations
- genotype consists of list of nodes (neurons) and connection weights (synapses)
- key aspects
  - genetic encoding with historical markings
  - speciation
  - complexification

[Stanley et al: Efficient Reinforcement Learning through Evolving Neural Network Topologies (2002)]

NEAT – Evolve neural network topologies

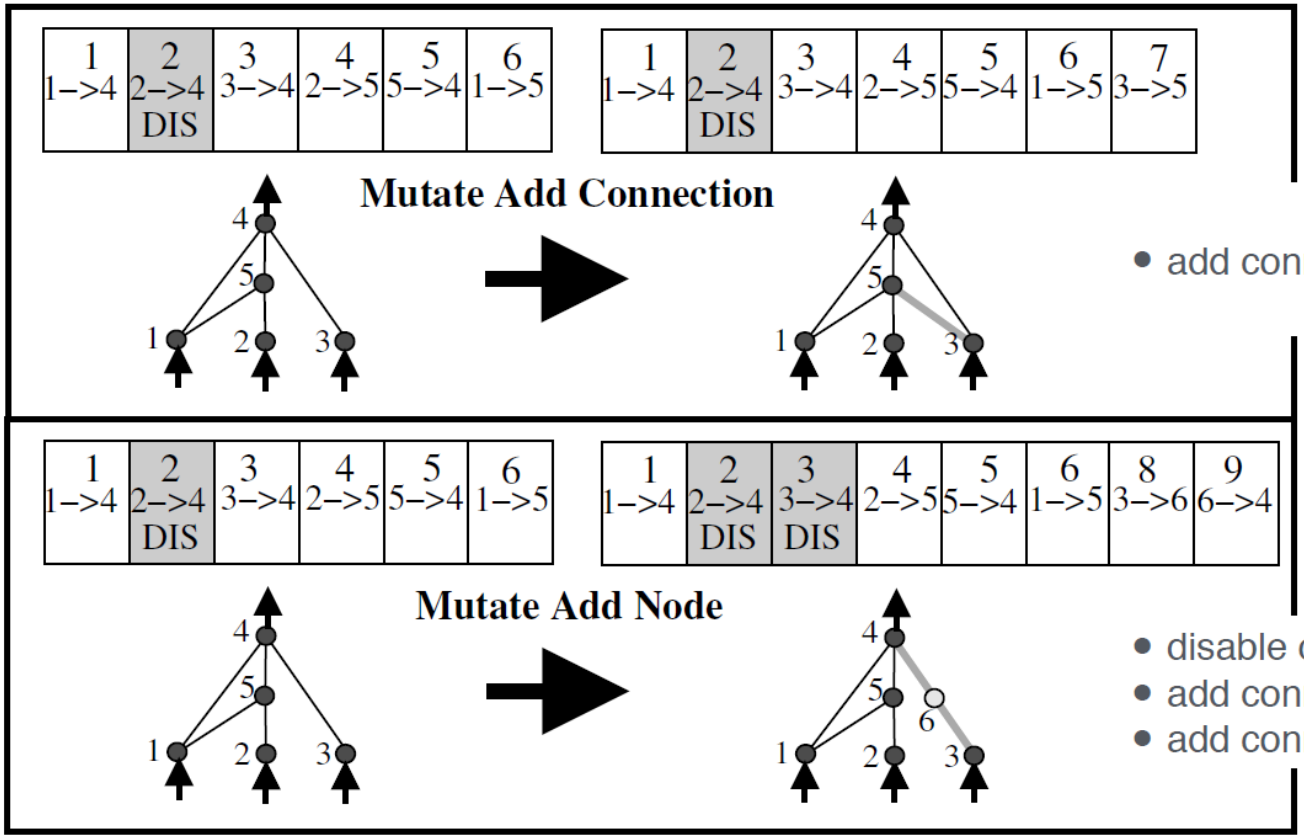
- variable-size chromosomes
- connection gene: in-node, out-node, weight and status
- node gene: input (sensor), output or hidden node
- each gene carries innovation number which allows chromosomes to be lined up
  - it is important to overlay two topologies for crossover
  - each new connection gene has its own innovation number



[Stanley et al: Efficient Reinforcement Learning through Evolving Neural Network Topologies (2002)]

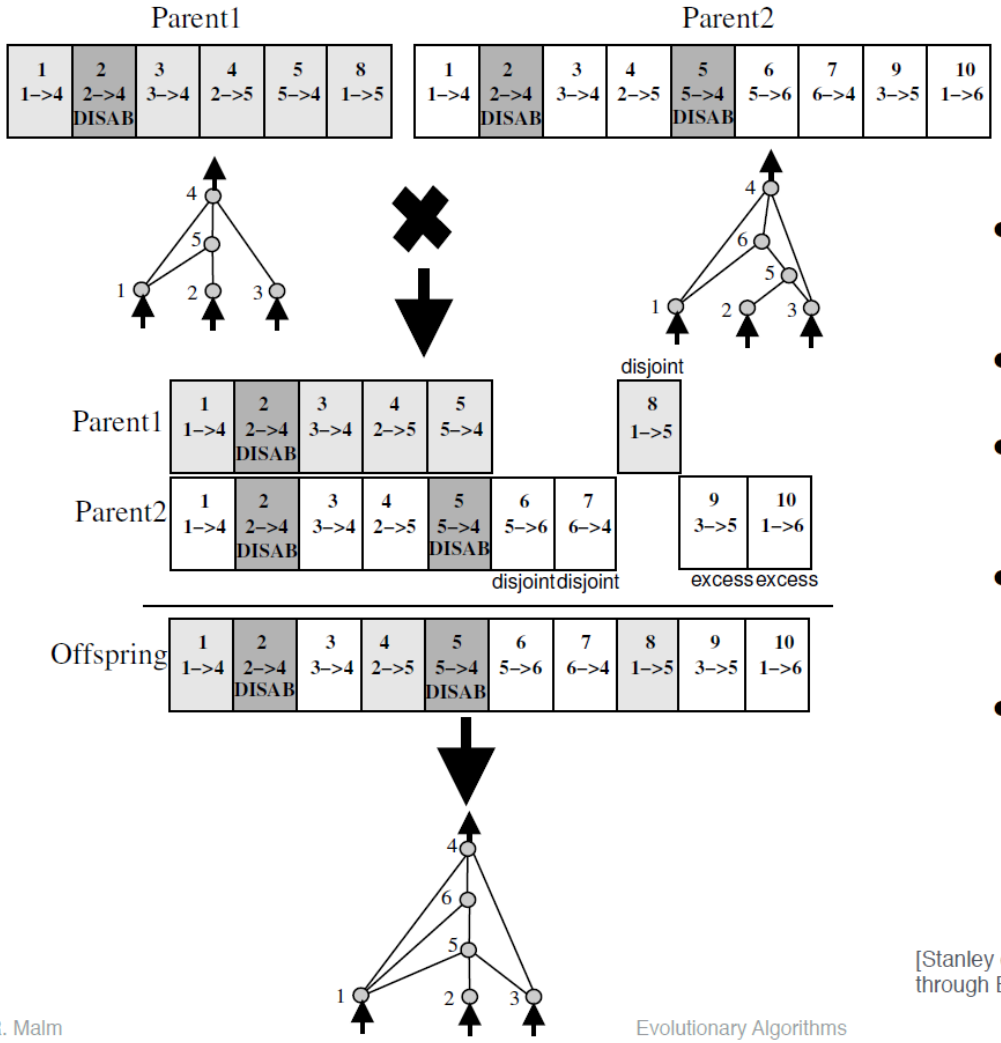
NEAT – Evolve neural network topologies

- mutation operators: add connection, add node, perturb connection weight
  - DIS = “disable”
- maintain mapping of mutations to innovation numbers



- add connection 3->5
- disable connection 3->4
- add connection 3->6
- add connection 6->4

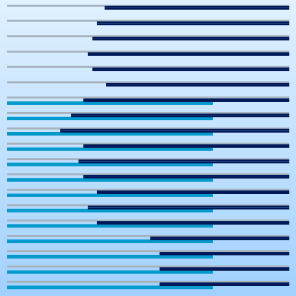
NEAT – Evolve neural network topologies



- genes in both genomes with the same innovation number are lined up
- these genes are called matching genes
- genes that do not match are either disjoint or excess
- matching genes are selected randomly for the offspring
- disjoint/excess genes are always included

[Stanley et al: Efficient Reinforcement Learning through Evolving Neural Network Topologies (2002)]

NEAT – Speciation



Problem

- smaller networks optimise faster than larger networks, i.e. adding nodes and connections initially decrease the fitness, compared to just perturbing weights
- this prevents diversity/innovation of network topologies

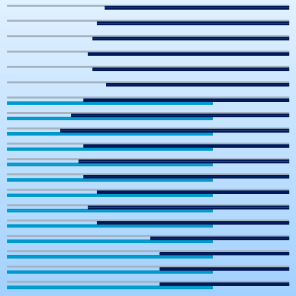
Solution: Speciation

- divide population into species such that similar topologies are in the same species, and then perform competition within each species
- measure “compatibility distance” between two chromosomes (topologies) as a linear combination of the number of excess (E), disjoint (D) and the average weight differences of matching genes (W)

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 W$$

- N = number of genes in the larger genome, c1, c2, c3 are adjustable hyperparameters
- form a species if the distance of two networks is below a compatibility threshold

NEAT – Complexification



Idea: start with simplest network and then incrementally grow structure

- first generation starts with population of network with identical topology
  - minimal structure
  - no hidden nodes
  - possibly only one connection
- each network starts with random weights
- complexification is the process of introducing structure through add connection and add node mutations in an incremental process

NEAT – Evolve neural network topologies

Double Pole Balancing

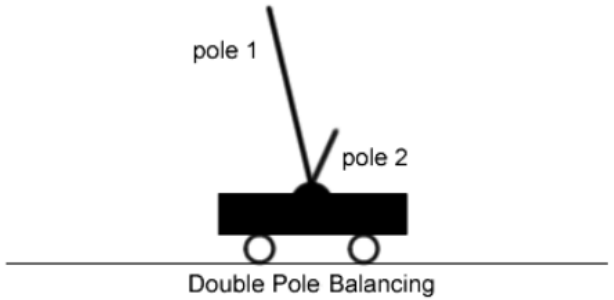
- with velocity information
- 150 NEAT networks

Method	Evaluations	Generations	No. Nets
Ev. Programming	**307,200**	150	2048
Conventional NE	**80,000**	800	100
SANE	**12,600**	63	200
ESP	**3,800**	19	200
NEAT	**3,578**	24	150

- without velocity information
- 1000 NEAT networks

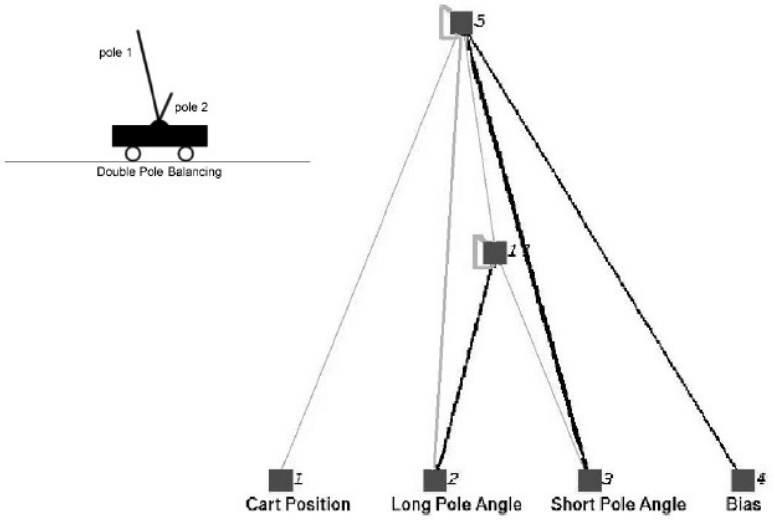
Method	Evaluations	Generalization	No. Nets
CE	**840,000**	300	16,384
ESP	**169,466**	289	1,000
NEAT	**33,184**	286	1,000

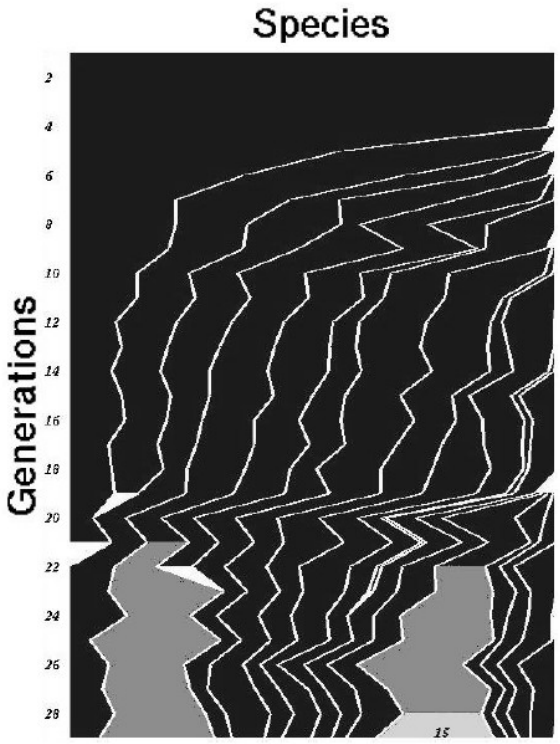
[Stanley et al: Efficient Reinforcement Learning through Evolving Neural Network Topologies (2002)]



Double Pole Balancing

NEAT – Evolve neural network topologies





- NEAT found a simple recurrent network
- using the recurrent connection to itself the single hidden node determines whether the poles are falling away or towards each other
- allows controlling the system without computing the velocities of each pole separately

[Stanley et al: Efficient Reinforcement Learning through Evolving Neural Network Topologies (2002)]