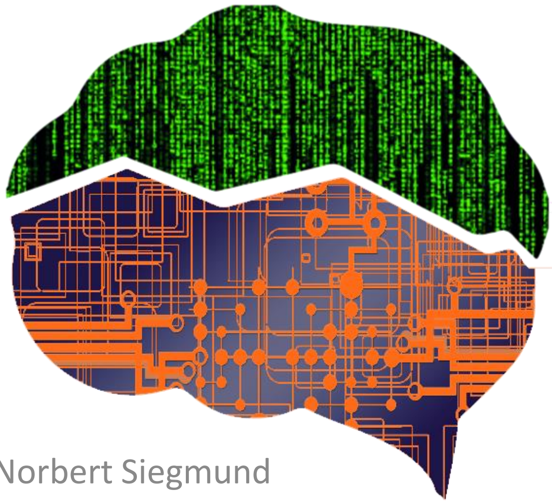
Machine Learning for Software Engineering

Multi-State Meta-Heuristics



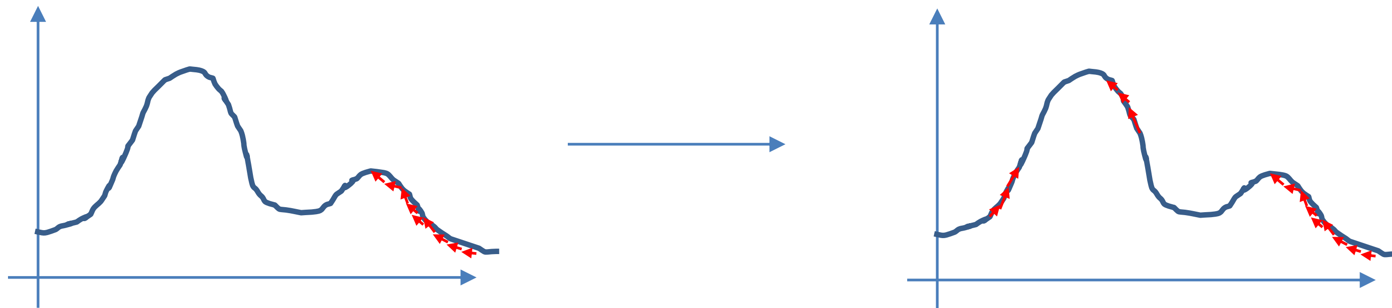
Prof. Dr.-Ing. Norbert Siegmund
Intelligent Software Systems

Bauhaus-Universität Weimar

Multi-State Optimization (Population Methods)

What is new?

- Instead of saving the globally best solution or the currently best solution, we keep a *sample* of candidate solutions



- Difference to parallel hill climbing:
  - Candidate solutions affect how other candidates will climb the hill
  - For ex. good solutions will replace bad ones by new solutions
  - For ex. bad solutions will be tweaked in the direction of good ones

Evolutionary Algorithms (EAs)

- Idea: borrow concepts from biology regarding genetics, evolution, and reproduction to resample the current candidate solutions
  - New candidate solutions are created or existing ones revised based on the results of older solutions

- General process:
  - Construct initial population
  - Iterate over:
    - **Assess fitness** of all individuals in the population
    - Use fitness information to **breed** a new population
    - **Join** the parents and children in some fashion to form the next-generation population

Terms & Notation

Term	Description
Individual	Candidate solution
Child and parent	Child is tweaked copy of a candidate solution (parent)
Population	Set of candidate solutions
Fitness	Quality
Fitness landscape	Quality function (usually unknown)
Selection	Select an individual based on the fitness value
Mutation	Tweak operation
Recombination & Crossover	Tweak operation with two parents as input and doing some operations over the elements of the parents to produce two children
Genotype or genome	Data structure of an individual
Chromosome	Genotype of a fixed-length vector
Gene	A specific slot or element in a chromosome
Allele	Particular setting of a gene
Phenotype	How the individual operates during fitness assessment
Generation	One cycle of fitness assessment, breeding, and population generation; or the population produced at each cycle

Breed

Basic Algorithm

$P \leftarrow$ build initial population
$Best \leftarrow empty$
repeat
$AssessFitness(P)$
for each individual $P_i \in P$ **do**
if $Best == empty$ or $Fitness(P_i) > Fitness(Best)$ **then**
$Best \leftarrow P_i$
$P \leftarrow Join(P, Breed(P))$
until $Best$ is optimum or out of time
return $Best$

First, we need to construct a set of (random) candidate solutions

Second, compute the quality of each candidate solution and store it

Third, breed new solutions based on the quality of each candidate solution

Forth, join the newly bred candidate solutions with the solutions of the current population

Difference to single-state algorithms: We need to assess the fitness (quality) of all candidate solutions before we can decide which one survives/to breed

From Basic to Concrete Algorithms

- Breed operation:
  - How to select parents from the old population and how to tweak them to make children?
- Join operation:
  - Shall we replace the parent population completely or keep some of them?
- Initialization operation:
  - If you don't know anything about the "good-solution-area" -> random
  - If you have knowledge, bias the random generation toward the "good-solution-area" (e.g., include / seed user-defined solutions in the initial population)
  - Make sure that you use only unique individuals

Evolution Strategies (ES)

- Invented in mid 1960s by

Ingo Rechenberg

Hans-Paul Schwefel

- Characteristics:
  - Selecting individuals using Truncate Selection
  - Only use mutation as tweak realization
- Simplest algorithm is (μ,λ)
  - λ is the number of individuals, randomly generated
  - Delete from the population all, but μ fittest individuals
  - Each of the fittest individuals produce λ/μ children (mutation)
  - Join operation replaces the parents by the children

(μ,λ) Algorithm

$\mu \leftarrow$ number of parents that are used to breed children
$\lambda \leftarrow$ number of children to be generated by the parents
$P \leftarrow \{\ \}$
for λ times **do**
  $P \leftarrow P \cup$ {random individual}
$Best \leftarrow empty$
repeat
  for each individual $P_i \in P$ **do**
    $AssessFitness(P_i)$
    if $Best == empty$ or $Fitness(P_i) > Fitness(Best)$ **then**
      $Best \leftarrow P_i$
  $Q \leftarrow$ the μ individuals in P whose $Fitness$ are greatest
  $P \leftarrow \{\}$
  for each individual $Q_i \in Q$ **do**
    for λ / μ times **do**
      $P \leftarrow P \cup \{Mutate(Copy(Q_i))\}$
until $Best$ is optimum or out of time
return $Best$

Exploration vs. Exploitation in (μ,λ)

- λ controls sample size for each population
  - Equal to *n* in Steepest-Ascent Hill Climbing with Replacement
  - If it goes to infinity, it is random search
- μ controls the selectivity of the algorithm
  - If it is low, it maximizes exploitation
- The degree of mutation
  - Amount of noise that is used to mutate an individual to produce a new child
  - High noise means explorative and low means exploitative

(μ+λ) Algorithm

- The only difference is the join operation

(μ,λ)	(μ+λ)
$P \leftarrow \{\}$	$P \leftarrow \{Q\}$

- The fittest parents survive and compete with their children in the next generation

- Can cause premature convergence as the parents restrict exploration
- Compare: Steepest Ascent Hill Climbing (1+λ) with Steepest Ascent Hill Climbing with Replacement (1, λ)
  - So, (μ+λ) ES is the more general algorithm

Realizing Mutations for ES

- Usually, the individual is represented as fixed-length vector of real numbers

[2.4][1.2][-12.5][0.1][3]

- Numbers are generated and mutated with Gaussian Convolution (see last lecture)
  - Normal distribution with a given variance: σ^2 = mutation rate
  - Variance controls exploration vs. exploitation
- How to set σ^2?
  - Static: with our without domain knowledge
  - Adaptive: changing σ^2 over time = adaptive mutation rate

Adaptive Mutation Rate: One-Fifth Rule

- If more than $\frac{1}{5}$ children are fitter than their parents, too much exploitation -> increase σ^2
- If less than $\frac{1}{5}$ children are fitter than their parents, too much exploration -> decrease σ^2
- If exactly $\frac{1}{5}$ children are fitter than their parents, keep σ^2

Evolutionary Programming (EP) is very similar to ES, but often broader than ES with respect to the representation of an individual (and so the mutation operation is different)

Genetic Algorithms (GA)

Introduction to GA

- Invented by John Holland in 1970s
- Approach is similar to the (μ,λ) algorithm
- Difference in selection and breeding operation
  - ES selects parents before breeding children
  - GA selects little-by-little parents to breed new children
- Breeding:
  - Select two parents, copy them, crossover them, mutate results, and add the two children to the new population
  - Repeat until population is full

GA Algorithm

$size \leftarrow$ population size
$P \leftarrow \{\}$
for $size$ times **do**
  $P \leftarrow P \cup$ {random individual}
$Best \leftarrow empty$
repeat
  for each individual $P_i \in P$ **do**
    $AssessFitness(P_i)$
    if $Best == empty$ or $Fitness(P_i) > Fitness(Best)$ **then**
      $Best \leftarrow P_i$
  $Q \leftarrow \{\}$

From here it deviates from (μ,λ)

for $size/2$ times **do**
    $Parent\ P_a \leftarrow SelectWithReplacement(P)$
    $Parent\ P_b \leftarrow SelectWithReplacement(P)$
    $Children\ C_a, C_b \leftarrow Crossover\big(Copy(P_a), Copy(P_b)\big)$
    $Q \leftarrow \cup \{Mutate(C_a), Mutate(C_b)\}$
  $P \leftarrow Q$
until $Best$ is optimum or out of time
return $Best$

How to Do the Crossover?

- One-Point Crossover
  - Swap everything below a randomly chosen index

Index: 1 2 3 4 5

[1][1][0][0][1] Parent a

[0][1][1][1][0] Parent b

Swap=4 (randomly chosen)

[0][1][1][0][1] Child a

[1][1][0][1][0] Child b

A More Flexible Crossover

- Two-Point Crossover
  - Select two random indexes and switch all genes in between

Index: 1 2 3 4 5

[1] [1] [0] [0] [1] Parent a

[0] [1] [1] [1] [0] Parent b

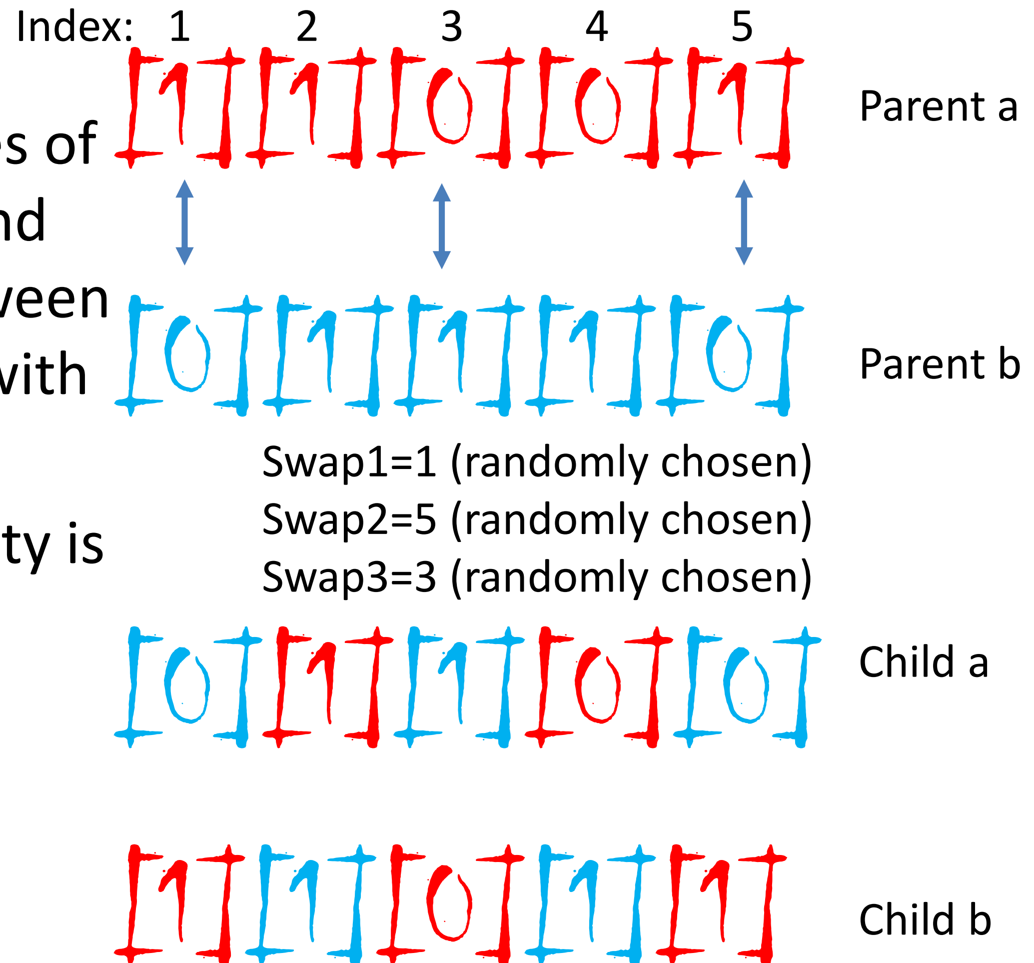
Swap1=2 (randomly chosen)
Swap2=5

[1] [1] [1] [1] [1] Child a

[0] [1] [0] [0] [0] Child b

Crossover Arbitrary Genes

- Uniform Crossover
  - Go through the genes of the first individual and swap the genes between the two individuals with a certain probability
  - Usually the probability is just 0.5



Algorithms Overview

One-Point Crossover

$\vec{x} \leftarrow$ first parent: $\langle x_1, \dots, x_l \rangle$
$\vec{v} \leftarrow$ second parent: $\langle v_1, \dots, v_l \rangle$
$c \leftarrow$ random int chosen uniformly from 1 to l
if $c \neq 1$ **then**
  for i from 1 to $c - 1$ **do**
    swap the values of x_i and v_i
return $\vec{x}$ and $\vec{v}$

Two-Point Crossover

$\vec{x} \leftarrow$ first parent: $\langle x_1, \dots, x_l \rangle$
$\vec{v} \leftarrow$ second parent: $\langle v_1, \dots, v_l \rangle$
$c \leftarrow$ random int chosen uniformly from 1 to l
$d \leftarrow$ random int chosen uniformly from 1 to l
if $c > d$ **then**
  swap c with d
if $c \neq d$ **then**
  for i from c to $d - 1$ **do**
    swap the values of x_i and v_i
return $\vec{x}$ and $\vec{v}$

Uniform Crossover

$p \leftarrow$ probability of swapping a gene
$\vec{x} \leftarrow$ first parent: $\langle x_1, \dots, x_l \rangle$
$\vec{v} \leftarrow$ second parent: $\langle v_1, \dots, v_l \rangle$
for i from 1 to l **do**
  if $p \geq$ uniform random nb (0 to 1) **then**
    swap the values of x_i and v_i
return $\vec{x}$ and $\vec{v}$

Why is Crossover Alone not Sufficient?

- Children will be constrained to the hyper space that the parents span
- Hyper space might be significantly smaller than the overall search space
- Best solutions might lie outside the hyper space
  - We won't find the global optimum

- So, we need an operation to break out of the hyper space
- Still, crossover has its benefits to share high-performing building blocks of individuals
  - Building blocks are combinations of genes that are linked (i.e., interact positively wrt. the objective function)
  - One- and two-point crossover assumes that the linked genes are encoded as neighbors in the vector representing the individual (often unlikely, though)

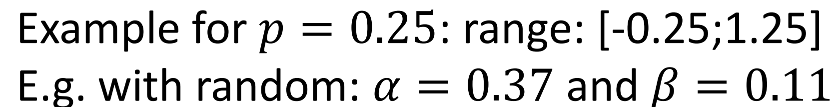
Going Beyond Binary for Crossover

- Swapping the exact floating-point number makes not so much sense
- What can we do?
  - Use the average between two floating-point values
  - Use a random number between two floating-point values

- Can we generate also new values to break out of the hyper cube?
  - Idea: **Line Recombination**

Line Recombination Algorithm

$\alpha \leftarrow$ random value from $-p$ to $1 + p$ inclusive
$\beta \leftarrow$ random value from $-p$ to $1 + p$ inclusive
for i from 1 *to* l **do**
  $t \leftarrow \alpha x_i + (1 - \alpha) v_i$
  $s \leftarrow \beta v_i + (1 - \beta) x_i$
    if t and s are within bounds **then**
      $x_i \leftarrow t$
      $v_i \leftarrow s$
return $\vec{x}$ and $\vec{v}$



$x_i = 3.5; v_i = 1.0$
$t = 0.37 * 3.5 + (1 - 0.37) * 1.0 = 1.925$
$s = 0.11 * 1.0 + (1 - 0.11) * 3.5 = 3.21$

Extension: Intermediate Recombination

- Just shifting two lines allows us to generate children not only on the line vector between two parents, but in the whole hyper cube

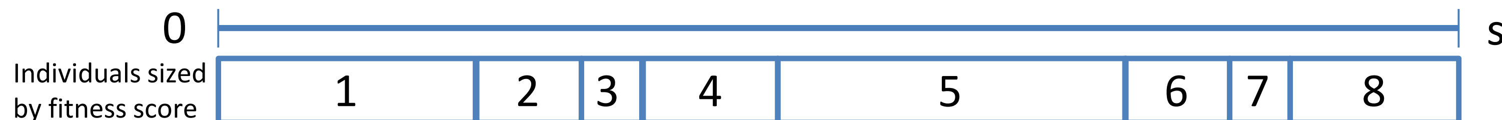
$\vec{x} \leftarrow$ first parent: $\langle x_1, \dots, x_l \rangle$
$\vec{v} \leftarrow$ second parent: $\langle v_1, \dots, v_l \rangle$
$p \leftarrow$ positive value defining how far we outrach the hyper cube (e.g., 0.25)

for i from 1 to l **do**
  repeat
    $\alpha \leftarrow$ random value from $-p$ to $1 + p$ inclusive
    $\beta \leftarrow$ random value from $-p$ to $1 + p$ inclusive
    $t \leftarrow \alpha x_i + (1 - \alpha) v_i$
    $s \leftarrow \beta v_i + (1 - \beta) x_i$
  until t and s are within bounds
  $x_i \leftarrow t$
  $v_i \leftarrow s$
return $\vec{x}$ and $\vec{v}$

Moved lines mean that we use different α and β values for each element

A Better Selection Operation

- So far: SelectWithReplacement
  - Can lead to selecting the same individual multiple times
  - Can select some low-fitness individuals
- Better: Select with a higher probability an individual with a high fitness score: Fitness Proportionate Selection (or Roulette Selection)
  - Idea:
    - Span a value range that is proportional to an individual's score
    - Concatenate all value ranges
    - Compute a random number in the all-value range and look up the corresponding individual



Fitness-Proportionate Selection (FPS)

$\vec{p} \leftarrow$ population consisting of a vector of individuals: $\langle \vec{p_1}, \ldots, \vec{p_l} \rangle$
$\vec{f} \leftarrow$ fitness score of each individual (same order as $in\ \vec{p}$): $\langle f_1, \ldots, f_l \rangle$

forall $f\ in\ \vec{f}$ **do**
  if $f == 0$ **then**
    $f \leftarrow 1.0$
for i from 2 to l ***do***
  $f_i \leftarrow f_i + f_{i-1}$

Deal with 0 fitness score to have at least a tiny chance to be accepted
Build the value range of all fitness scores as a cumulative density function (CDF)

$n \leftarrow$ random number from 0 to f_l inclusive
for i from 2 to l **do**
  if $f_{i-1} < n \leq f_i$ **then**
    return p_i
return p_1

Repeat this for each parent to be selected for crossover
Select the parent individual based on a random number falling into its corresponding interval

Note: That this is always an 1-based index value (not zero-based)

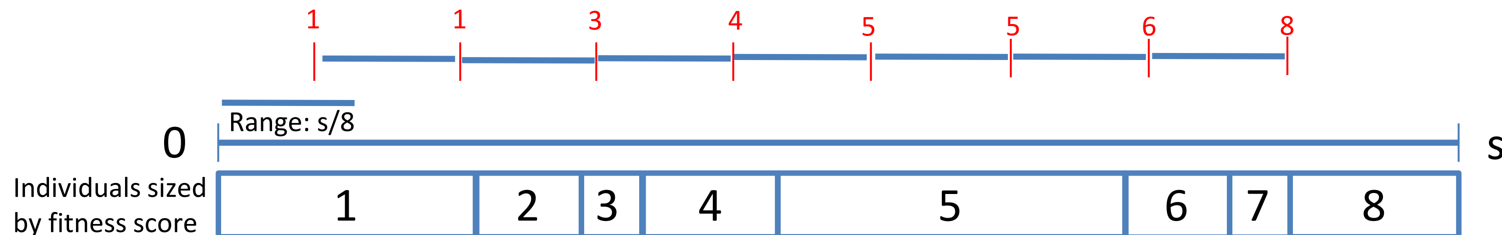
Problems of FPS

- Weak solutions can still be selected very often
- We might never the select the best solutions

- => Stochastic Universal Sampling (SUS)
  - Fit individuals get selected at least once
  - Also used in other areas (Particle Filters) under the term low variance resampling

Stochastic Universal Sampling (SUS) Algorithm

- Build fitness array as in FPS
- Draw a random number between 0 and s/n (here, s/8)
- Select individual at this position (here, 1)
- Increment current position by, s/n and repeat till n individuals have been selected
- Benefit: O(n) effort vs. O(n log n) for FPS
- Benefit: SUS guarantees that if an individual has a high score (>s/n), it will get chosen by the algorithm



In Code (for you to do at home)

$\vec{p} \leftarrow$ population consisting of a vector of individuals: $\langle \overrightarrow{p_1}, \dots, \overrightarrow{p_l} \rangle$

$\vec{f} \leftarrow$ fitness score of each individual (same order as $in\ \vec{p}$): $\langle f_1, \dots, f_l \rangle$

$index \leftarrow 0$

forall f in $\vec{f}$ **do**

  if $f == 0$ **then**

    $f \leftarrow 1.0$

for i from 2 to l ***do***

  $f_i \leftarrow f_i + f_{i-1}$

$offset \leftarrow$ random number from 0 to $\frac{f_l}{n}$ inclusive (where usually $n = l$)

for $f_{index} < offset$ **do**

  $index \leftarrow index + 1$

$offset \leftarrow offset + \frac{f_l}{n}$

return p_{index}

Repeat this for each parent to be selected for crossover

Nature of Fitness Value

- Assumption so for: Fitness value is on a metric scale
  - Distances between two fitness value has a meaning
  - Also called parametric function
- Often not the case: Consider the property reliability in software engineering
  - Systems that run reliably are up to 98.99, 99.97, 99.98, or 99.99 percent of a year (the peak is 99.99)
  - But using SUS all individuals have nearly the same probability to be selected
- What can we do?

Non-Parametric Selection Algorithm

- Non-parametric tests in statistics are based only on ranking
- There is no notion of distances
- Tournament Selection: Bigger is better

$P \leftarrow$ population of any representation
$t \leftarrow$ tournament size with $t \geq 1$
$Best \leftarrow$ individual picked at random from P with replacement
for i from 1 to t ***do***
  $Next \leftarrow$ individual picked at random from P with replacement
  if $Fitness(Next) > Fitness(Best)$ **then**
    $Best \leftarrow Next$
return $Best$

- Primary selection technique for a genetic algorithm!
  - Great tuning capability with tournament size (usually t=2)

Take Home Message:

- Evolutionary strategies use only mutation as tweak and select individuals using a truncate operation
- Genetic algorithms go a step further be recombining parents using a crossover operations
- Many variants to implement crossover, selection of individuals for the next generation, and mutation
  - Depends on the encoding of a solution (e.g., if nearby genes are correlated)
  - On the fitness function (e.g., if metric scale or ranking scale)
  - On exploration vs. exploitation

Next Lecture

- Exploitative algorithms of population based optimization techniques
  - Elitism
  - The Steady-State Genetic Algorithm
  - Tree-Style Genetic Programming Pipeline
  - Hybrid Optimization
  - Scatter Search
- Differential Evolution