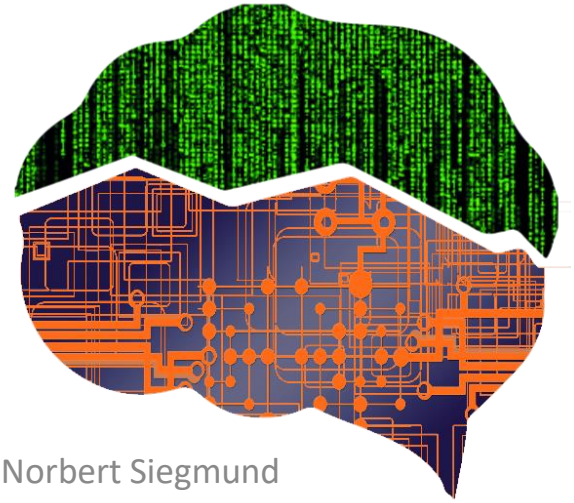
Machine Learning for Software Engineering

Exercise: Representation



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

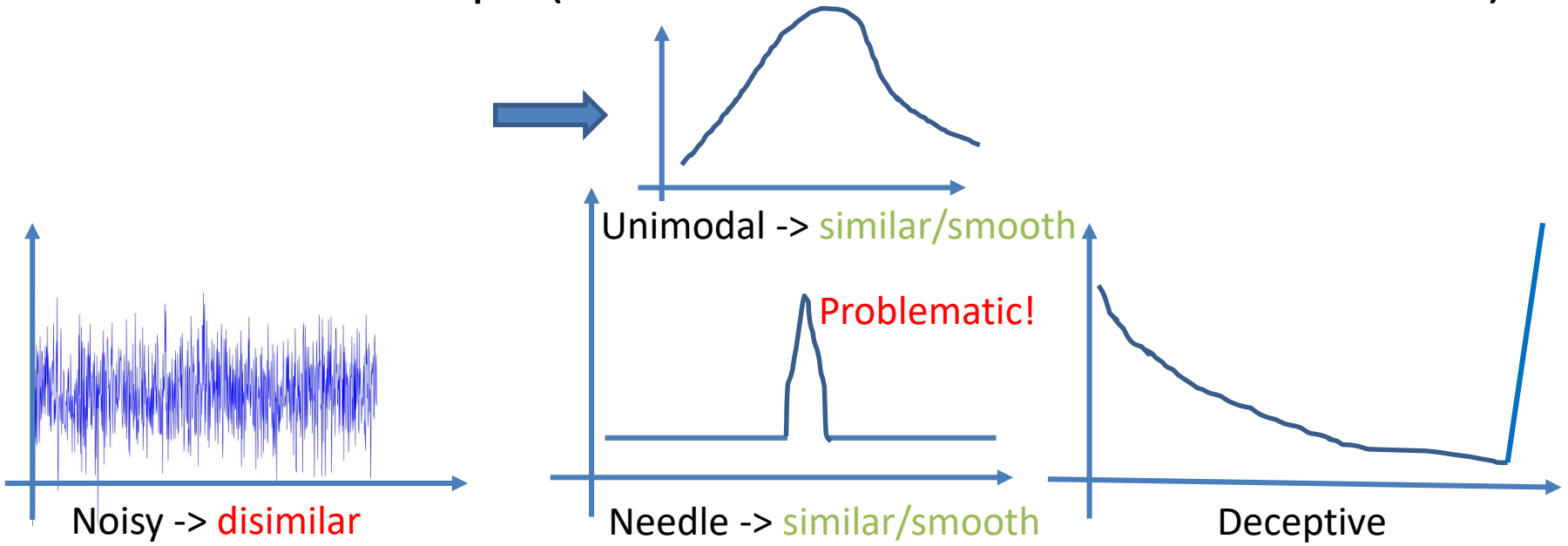
Bauhaus-Universität Weimar

General Considerations

- How to realize constructing, tweaking, and presenting an individual for fitness assessment?
  - Usually, considered as data structure
  - Now, consider it as two functions:
    - **Initialization** function for generating a random individual
    - **Tweak** function for realizing modifications
  - We might also need
    - **Fitness assessment** function
    - **Copy** function
- These are the only places where most algorithms need to know about the internals of an individual

Success of Optimization

- Depends on how we realize/implement the function and so how to represent an individual
  - Tweak is important as it explores the optimization space
- Goal: Find a representation, which allows for a smooth fitness landscape (similar individuals have similar fitness)



Similarity

- Being similar: Genotypes are similar
  - Genotype A is similar to genotype B if the probability is high that one can be tweaked to another
  - Close due to the choice of the tweak operation
- Behaving similar: Phenotypes are similar
  - Phenotype A is similar to phenotype B if they act/operate similar (nothing to do with fitness value)

- Which similarity do we need?
  - We need phenotype similarity as this is the ground truth representation of an individual (two individuals are similar, because they behave similar no matter how they actually look like)

So, How to Represent?

- When new individuals should be created: translate phenotype to a genotype (encode), tweak this, translate back to phenotype (decode)
- Parent phenotype -> encode -> tweak -> decode -> child phenotype
- Lessons learned from past: do not encode everything as bit-vectors

So, be careful how to encode! Go for smoothness in encoding! Small changes in genotype should result in small changes in phenotype and fitness

Example

Phenotype	Genotype	Fitness	Gray Code
0	0000	0	0000
1	0001	1	0001
2	0010	2	0011
3	0011	3	0010
4	0100	4	0110
5	0101	5	0111
6	0110	6	0101
7	0111	7	0100
8	1000	8	1100
9	1001	0	1101
10	1010	0	1111
11	1011	0	1110
12	1100	0	1010
13	1101	0	1011
14	1110	0	1001
15	1111	0	1000

Enode

Hamming Cliff

Small change in phenotype or fitness requires large change in genotype

Hard to find the optimum!

Now, easy mutation here

Each successive number in phenotype differs only by one bit flip in genotype from its precessor

Best Practices

- Make genotype as similar as phenotype
  - If phenotype is a graph, model genotype as graph as well
- Keep the functions of initialization and tweak in your mind
- Use the following as suggestions, not ground truth

Vectors

Already Seen in Lecture

- Boolean vectors:
  - **Initialize:** random bit-vector with probabilities of 0.5 to be 0 or 1
  - **Mutate:** flip bits in the vector with a certain probability
- Floating-point vectors:
  - **Initialize:** Generate random real-valued vector using random values between min and max value
  - **Mutate**: Bounded uniform convolution or Gaussian convolution
- Cross-over:
  - One/two/uniform cross over, (Intermediate) Line Recombination

What about Integers?

- What do the integers represent?
  - Is it non-parametric? So, do you encode an enumeration?
  - Is it parametric? So, do you define distances/scores, etc.?
  - Whether it is a metric space matters for realizing mutation

- Next, let us focus on integer vectors for initialization, mutation, and recombination

Initializing an Integer Vector

- Approach: For each position in the vector, use a random (uniformly chosen) integer between min and max valid integers
- As always, knowledge helps to improve initial solutions
  - Bias the generation toward promising values and away from bad areas
  - Example: $v_1 = v_2 * v_3$ if this is a promising region, generate values for v_1 accordingly, based on random values for v_2, v_3
- **Seeds** are a common technique for initialization
  - (Manually) select solutions before optimization and insert them as initial candidates
- Keep in mind: bias and seeds are dangerous as our assumptions might be wrong

Mutating Integer Vectors I

- Recap:
  - Floating-Point vectors -> Gaussian convolution
  - Bit/Boolean vectors -> Bit-Flip mutation
- Integers… it depends on non-parametric or metric-based representation

- For non-parametric integer vectors:

$\vec{v} \leftarrow$ integer vector $\langle v_1, v_2, \ldots, v_l \rangle$
$p \leftarrow$ probability of randomizing an integer ← Eg., $1/l$
for i from 1 to l **do**
  if $p \geq$ random number chosen uniformly from 0.0 to 1.0 inclusive **then**
    $v_i \leftarrow new\ random\ legal\ integer$
return $\vec{v}$

Mutating Integer Vectors II

- Metric-space mutation
  - Idea: Do something similar to Gaussian convolution (mostly small changes, but occasionally large changes)
  - Flip a coin and count the trials you need to get heads
  - Use the count to do a random walk of that length
  - Noise centered around original value + global mutation

- Algorithm, see next

Random Walk Mutation

$\vec{v} \leftarrow$ integer vector $\langle v_1, v_2, \ldots, v_l \rangle$
$p \leftarrow$ probability of randomizing an integer
$c \leftarrow$ probality of a coin flip

For large integer regions or larger mutations, increase this value

for i from 1 to l **do**
  if $p \geq$ random number chosen uniformly from 0.0 to 1.0 inclusive **then**
    repeat
      $n \leftarrow$ either a 1 or -1, chosen randomly
      if $v_i + n$ is within bounds of valid integers **then**
        $v_i \leftarrow v_i + n$
      else if $v_i - n$ is within bounds of valid integers **then**
        $v_i \leftarrow v_i - n$
    until $b <$ random number chosen uniformly from 0.0 to 1.0 inclusive
return $\vec{v}$

So far, all genes have an independent and the same probability to be mutated

Point Mutations

- Take one or n genes and mutate only these
- Good when your problem requires to make progress only when one gene is changed
- Bad for several ways:
  - Point Mutation is not global
  - It cannot break out of local optima

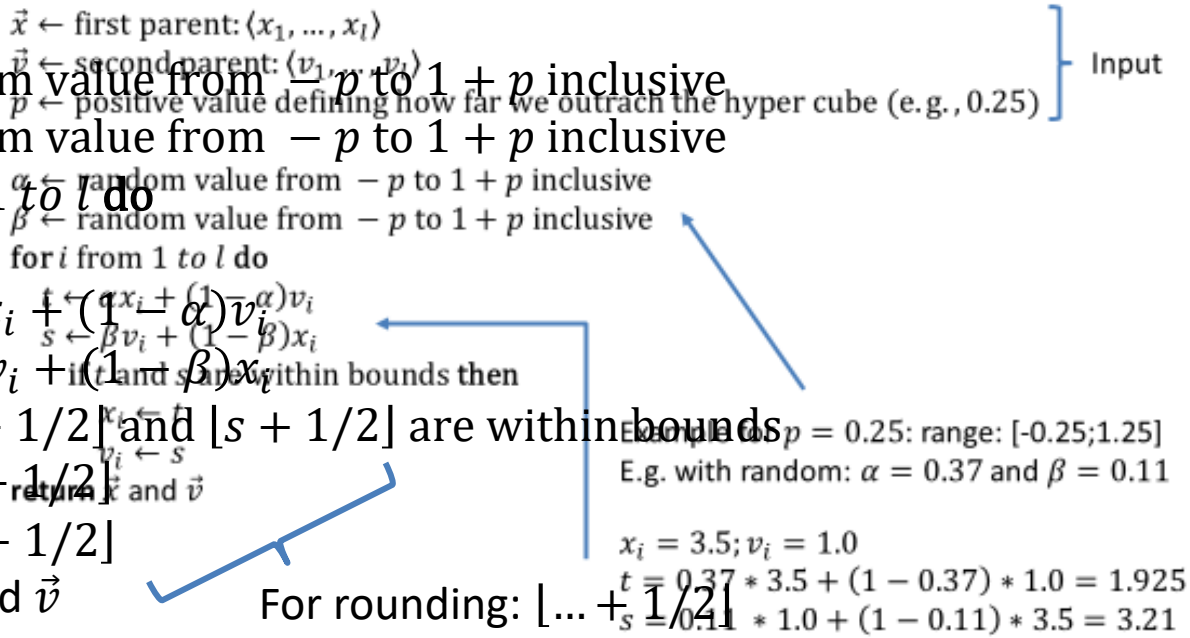
- So, be aware of this possibility

Recombination of Integer Vectors

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

$\alpha \leftarrow$ 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**
  repeat
    $t \leftarrow \alpha x_i + (1 - \alpha) v_i$
    $s \leftarrow \beta v_i + (1 - \beta) x_i$
  until $\lfloor t + 1/2 \rfloor$ and $\lfloor s + 1/2 \rfloor$ are within bounds
  $x_i \leftarrow \lfloor t + 1/2 \rfloor$
  $v_i \leftarrow \lfloor s + 1/2 \rfloor$
return $\vec{x}$ and $\vec{v}$

Line Recombination Algorithm



For rounding: $\lfloor \ldots + 1/2 \rfloor$

Intermediate Recombination for Int. Vec.

$\vec{x} \leftarrow$ first parent: $\langle x_1, \ldots, x_l \rangle$
$\vec{v} \leftarrow$ second parent: $\langle v_1, \ldots, v_l \rangle$
$p \leftarrow$ 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 $\lfloor t + 1/2 \rfloor$ and $\lfloor s + 1/2 \rfloor$ are within bounds
  $x_i \leftarrow \lfloor t + 1/2 \rfloor$
  $v_i \leftarrow \lfloor s + 1/2 \rfloor$
return $\vec{x}$ and $\vec{v}$

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: $\langle x_1, \ldots, x_l \rangle$
$\vec{v} \leftarrow$ second parent: $\langle v_1, \ldots, v_l \rangle$
$p \leftarrow$ 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

What About Mixtures of Ints, Floats, etc?

- Idea: Make all floating-point values
  - Bad: If one enumerates just colors (yellow=1, blue=2, etc.) what would Gaussian convolution give us? Nonsense!
- Better idea: Mutate, initialize, and crossover each gene according to its type
- Worst case: if you have also graphs, trees, etc. in genes, you need to use a representation of a vector of objects and develop an individual procedure for each object
- Phenotype of mutation and crossover
  - If phenotype is a matrix and genotype a vector, you might want to do the crossover in the phenotype to slice out a rectangular region of the matrix and not a slice in the vector