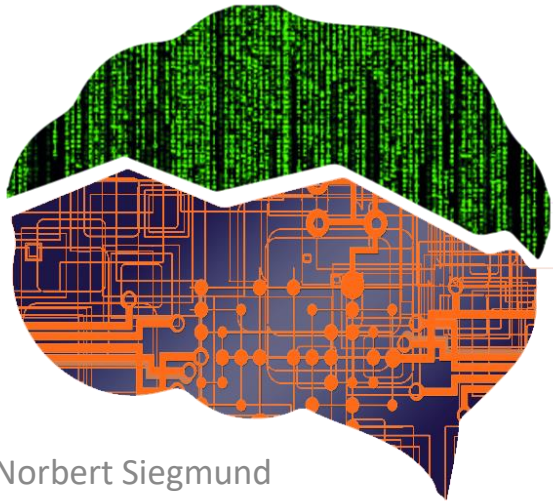


# Search-Based Software Engineering

## Representation



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

Bauhaus-Universität  
Weimar

# Recap

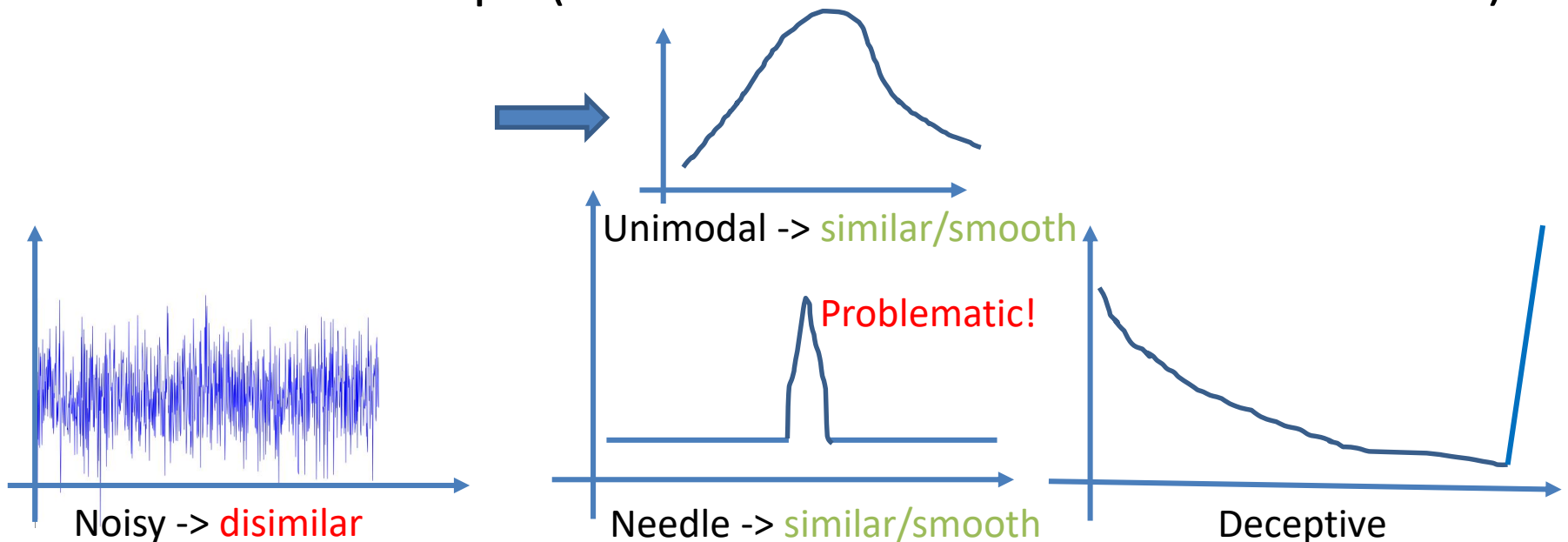
- What does  $(\mu, \lambda)$  stand for?
- Difference between  $(\mu, \lambda)$  and  $(\mu + \lambda)$  ?  
$$P \leftarrow \{\} \quad P \leftarrow \{Q\}$$
- Relation to Steepest Ascent Hill Climbing (with Replacement)?
- Basic operations of evolutionary algorithms?
  - Breed (how to select parents and how to tweak them to make children)
  - Join (replacing parents with children? How?)
  - Initialization (random? With bias?)

# 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

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

Enode →

Hamming Cliff

Small change in phenotype or fitness requires large change in genotype

Hard to find the optimum!

Now, easy mutation here

# 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

# What We Already Know

---

- 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  $\langle v_1, v_2, \dots, v_l \rangle$

$p \leftarrow$  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  $\langle v_1, v_2, \dots, v_l \rangle$   
 $p \leftarrow$  probability of randomizing an integer  
 $c \leftarrow$  probability 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

## Line Recombination Algorithm

$\vec{x} \leftarrow$  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) } Input

$\alpha \leftarrow$  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  $\vec{v}$

Example for  $p = 0.25$ : range:  $[-0.25; 1.25]$

E.g. with random:  $\alpha = 0.37$  and  $\beta = 0.11$

$x_i = 3.5; v_i = 1.0$

$t = 0.37 * 3.5 + (1 - 0.37) * 1.0 = 1.925$

$s = 0.11 * 1.0 + (1 - 0.11) * 3.5 = 3.21$

# Recombination of Integer Vectors

$\vec{x} \leftarrow$  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)

$\alpha \leftarrow$  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  $\vec{v}$

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



# Intermediate Recombination for Int. Vec.

## 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  $\alpha$  and  $\beta$  values for each element

# Intermediate Recombination for Int. Vec.

$\vec{x} \leftarrow$  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**  $\lfloor t + 1/2 \rfloor$  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  $\vec{v}$

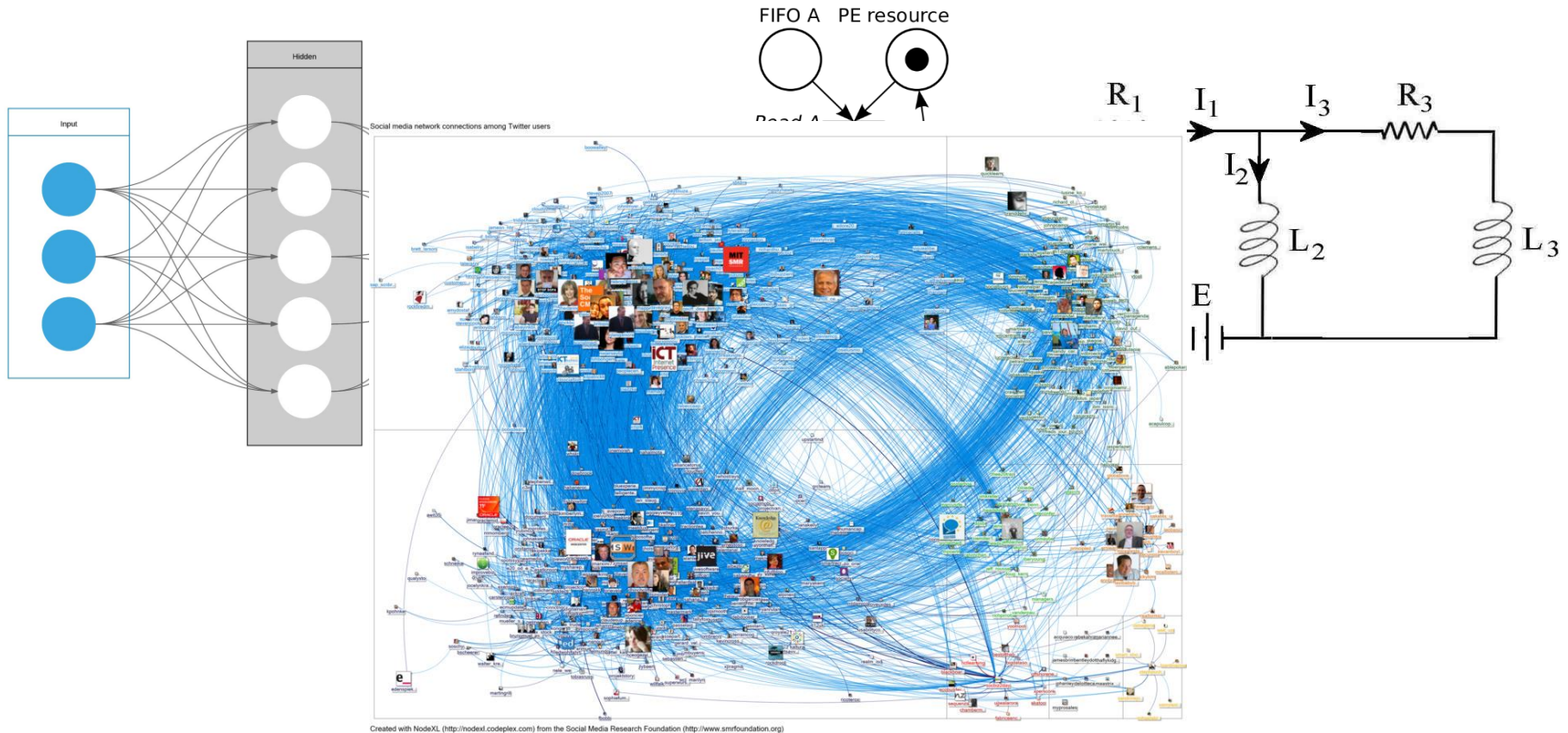
# What About Mixtures of Ints, Floats, etc?

- 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

# Direct Encoded Graphs

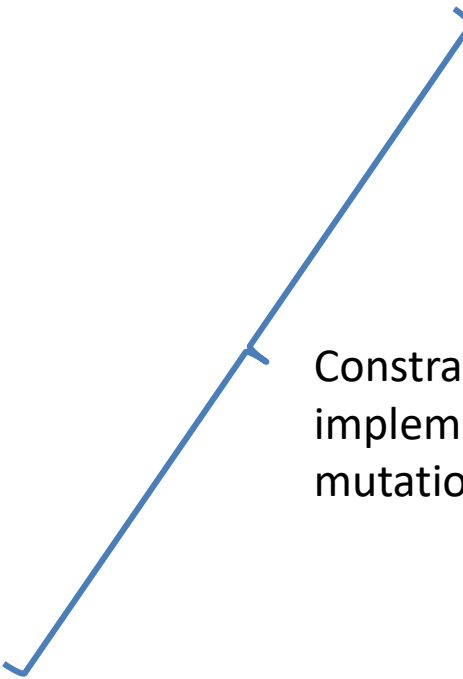
# Graphs

- Probably the most complex representation
- Application scenarios:



# Types of Graphs

- Weighted edges / no weights
- Weighted nodes / no weights
- Directed / undirected
- Labels on edges
- Labels on nodes
- Recurrent graphs
- Feed-forward graphs
- Sparse / dense
- Planar graphs
- ...



Constraints and properties define the implementation of initialization, mutation, etc.

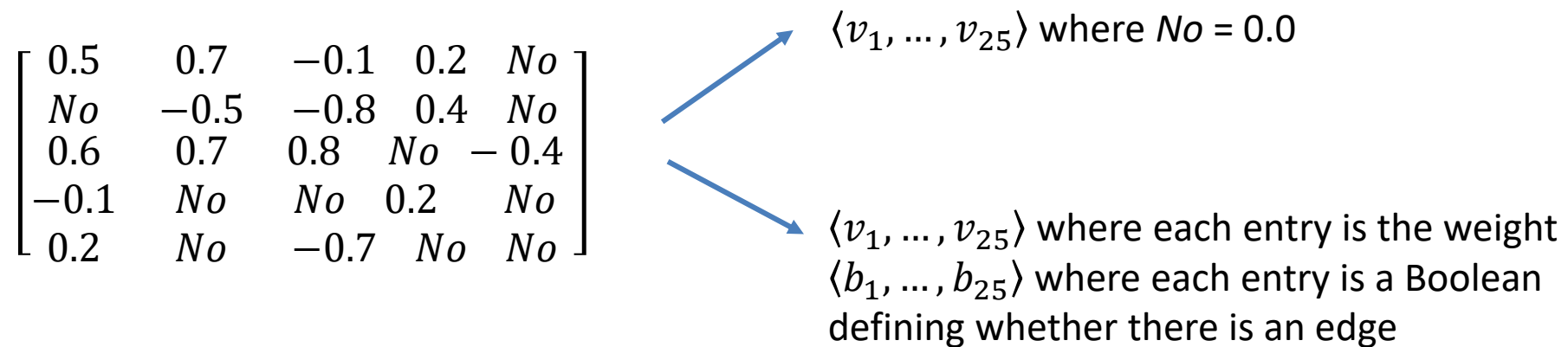


# Structure vs. Weights

- If structure / architecture is fixed (all nodes and edges are set and will not change), then finding weights is just a floating-point vector problem
- Here, we focus on arbitrarily structure graphs, for which we want to find the best structure
- Two main approaches:
  - Direct encoding
    - Stores exact edge-for-edge and node-for-node description of the graph
  - Indirect encoding
    - Stores a set of rules of a small program, which grows the graph when executed
    - Good, for recursive and repetitive graphs

# Full Adjacency Matrix

- Simplest direct encoding of a graph
- Requires to have an **absolute maximum graph size**
- Example: 5 nodes graph, recurrent, directed, no more than one edge between two nodes, self-edges are possible
  - Task: Find the optimal weights



Pro: Use the standard procedures with careful setting to No/0.0

Con: If already tuned weight is set to No, we lose the tuned weight

# Initializing Graphs

- Again, depends on what type of graph we have
- How many nodes and edges?
  - Uniform distribution from 1 to upper limit / large number
  - Geometric distribution favoring small numbers over large

$p \leftarrow$  probability of selecting a larger number

$m \leftarrow$  minimum valid number

$n \leftarrow m - 1$

**repeat**

$n \leftarrow n + 1$

**until**  $p <$  random number chosen uniformly between 0.0 to 1.0 inclusive

**return**  $n$

Larger  $p$  values result in larger  $n$  values  
 $E(n) = m + p/(1 - p)$

# Graph Construction

$n \leftarrow$  computed number of nodes

$e \leftarrow$  computed number of edges

$f(j, k, Nodes, Edges) \leftarrow$  validity-check function returns **true** if an edge from  $j$  to  $k$  is valid

$N \leftarrow \{N_1, \dots, N_n\}$  set of nodes

$E \leftarrow \{ \}$  set of edges

**for** each node  $N_i \in N$  **do**

$ProcessNode(N_i)$   $\leftarrow$  Function to assign labels, weights, etc.

**for**  $i$  from 1 to  $e$  **do**

**repeat**

$j \leftarrow$  random number chosen uniformly from 1 to  $n$  inclusive

$k \leftarrow$  random number chosen uniformly from 1 to  $n$  inclusive

**until**  $f(j, k, Nodes, Edges)$  returns **true**  $\leftarrow$  Could take **very** long

$g \leftarrow$  new edge from  $N_j$  to  $N_k$

$ProcessEdge(g)$   $\leftarrow$  Function to assign labels, weights, etc.

$E \leftarrow E \cup \{g\}$

**return**  $N, E$

Problem here: Disjoint graphs are possible

# Construct Directed Acyclic Graph

$n \leftarrow$  computed number of nodes

$D(m) \leftarrow$  probability distribution of the number of edges out of a node,  
given number of in-nodes  $m$

$f(j, k, Nodes, Edges) \leftarrow$  validity-check function

$N \leftarrow \{N_1, \dots, N_n\}$  set of nodes

$E \leftarrow \{ \}$  set of edges

**for** each node  $N_i \in N$  **do**

$ProcessNode(N_i)$

**for**  $i$  from 2 to  $n$  **do**

$p \leftarrow$  random integer  $\geq 1$  chosen using  $D(i - 1)$

**for**  $j$  from 1 to  $p$  **do**

**repeat**

$k \leftarrow$  random number chosen uniformly from 1 to  $i - 1$  inclusive

**until**  $f(i, k, Nodes, Edges)$  returns **true**

$g \leftarrow$  new edge from  $N_i$  to  $N_k$

$ProcessEdge(g)$

$E \leftarrow E \cup \{g\}$

**return**  $N, E$

# Mutating Graphs

- Pick a random number  $n$  and do  $n$  times any of this:
  - Delete a random edge with probability  $\alpha_1$
  - Add a random edge with probability  $\alpha_2$
  - Delete a node with all its edges with probability  $\alpha_3$
  - Add a node with probability  $\alpha_4$
  - Relabel a node with probability  $\alpha_5$
  - Relabel an edge with probability  $\alpha_6$
  - With  $\sum_i \alpha_i = 1$
- $n$  might come from a Geometric distribution, some probabilities should be lower than other; keep smoothness in mind!

# Recombination of Graphs

- Often too hard to be applied
- If applied, a subset of nodes and edges must be found

$S \leftarrow$  original set from which we want to draw a subset

$p \leftarrow$  probability of being a member of the subset

$S' \leftarrow \{ \quad \}$  subset

**for** each element  $S_i \in S$  **do**

**if**  $p \geq$  random number chosen uniformly from 0.0 to 1.0 inclusive **then**

$S' \leftarrow S' \cup \{S_i\}$

**return**  $S'$

$S \leftarrow$  original set from which we want to draw a subset

$n \leftarrow$  intended size of the subset

$S' \leftarrow \{ \quad \}$  subset

**for**  $i$  from 1 to  $n$  **do**

$S' \leftarrow S' \cup \{\text{random element from } S \text{ chosen without replacement}\}$

**return**  $S'$

# Crossover of Graphs

- Subsets have constraints and it is hard to exchange them (e.g., might end in disjoint graphs)
- Better, pick a whole subgraph and swap this

$N \leftarrow$  nodes in the original graph

$E \leftarrow$  edges in the original graph

$N' \subseteq N \leftarrow$  nodes in the subgraph (chosen to be a subset operation as before)

$E' \leftarrow \{ \}$  subset of edges

**for** each edge  $E_i \in E$  **do**

$j, k \leftarrow$  nodes connected by  $E_i$

**if**  $j \in N'$  and  $k \in N'$  **then**

$E' \leftarrow E' \cup \{E_i\}$

**return**  $N', E'$

- Still, subgraph is disjoint, so we need to merge next



# Merging of Graphs

$N \leftarrow$  nodes in the first graph

$E \leftarrow$  edges in the first graph


$N' \leftarrow$  nodes in the second graph

$E' \leftarrow$  edges in the second graph

$p \leftarrow$  probability of merging a given node from  $N$  into a node from  $N'$

**for**  $l$  from 1 to  $||N||$  **do**

**if**  $l == 1$  or  $p \geq$  random number chosen uniformly from 0.0 to 1.0 inclusive **then**

$n' \leftarrow$  random node chosen uniformly from  $N'$   We will merge  $N_l$  with  $n'$

**for**  $i$  from 1 to  $||E||$  **do**

$j, k \leftarrow$  nodes conneted by  $E_i$

**if**  $j == N_l$  **then**


                Change  $j$  to  $n'$  in  $E_i$

**if**  $k == N_l$  **then**

                Change  $k$  to  $n'$  in  $E_i$

When merging nodes, we need to rename certain edges, as they point to nonexistent nodes

**else**

$N' \leftarrow N' \cup \{N_l\}$   We do not merge, but just add  $N_l$  directly

$E' \leftarrow E' \cup E$

**return**  $N', E'$

# Trees and Genetic Programming

# How to Generate a Computer Program?

---

- Represent a program as a tree
- Have a notion of what is a good or bad program rather than what is a correct or incorrect program to be optimizable
  - Nearly correct programs are better than totally wrong programs
  - Degree of correctness might be a good fitness function
- Variable-sized data structures required (lists and trees)
- Formed based on basic operations/functions
  - Addition, subtraction, move up, call database
  - Operations might have a context, which limits the combination with the results or values of other operations

# Continued

---

- Nodes in a tree may define certain number of children
  - Multiplication vs. increment
- So, initialization and mutation aims to maintain **closure**
  - Stay in the valid solution space
- Fitness assessment is usually done by executing the program
  - Data of genotype must somehow correspond to the code of the phenotype when executed

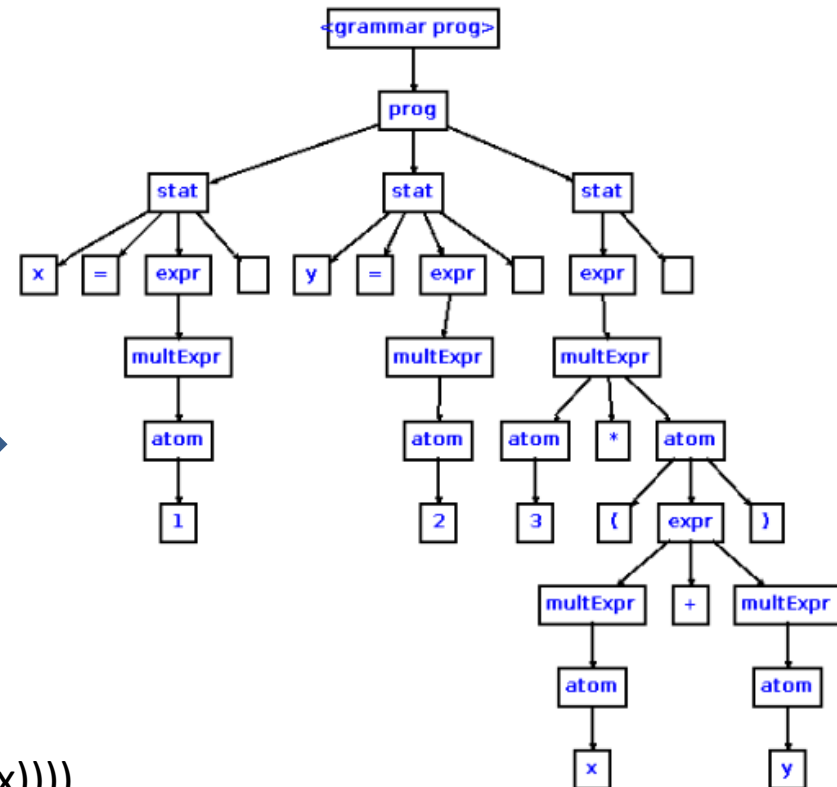
# Parse Tree

- Parse tree is the concrete representation of a parsed program with respect to a given grammar, retaining all information, such as white spaces, brackets, etc.

```

grammar Expr002;
options {
    output=AST;
    ASTLabelType=CommonTree; // type of $stat.tree ref etc...
}
prog : ( stat )+ ;
stat : expr NEWLINE -> expr
      | ID '=' expr NEWLINE -> ^('=' ID expr)
      | NEWLINE -> ;
expr : multExpr (( '+'^ | '-'^ ) multExpr)* ;
multExpr : atom ('*'^ atom)* ;
atom : INT
      | ID
      | '('! expr ')'! ;
ID : ('a'..'z' | 'A'..'Z' )+ ;
INT : '0'..'9'+ ;
NEWLINE : '\r'? '\n' ;
WS : ( ' ' | '\t' )+ { skip(); } ;

x=1
y=2
3*(x+y)
    
```

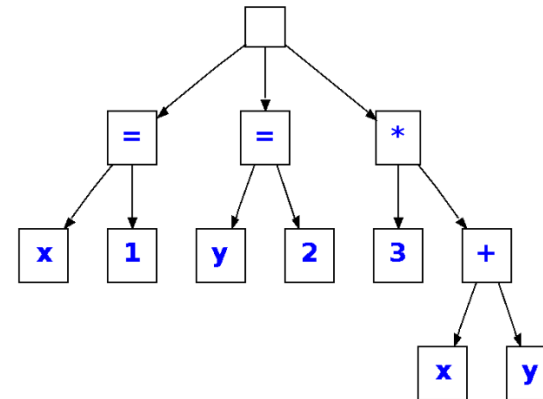


Program in Lisp: (sin (+ (cos (- x (sin x))) (\* x (sqrt x))))

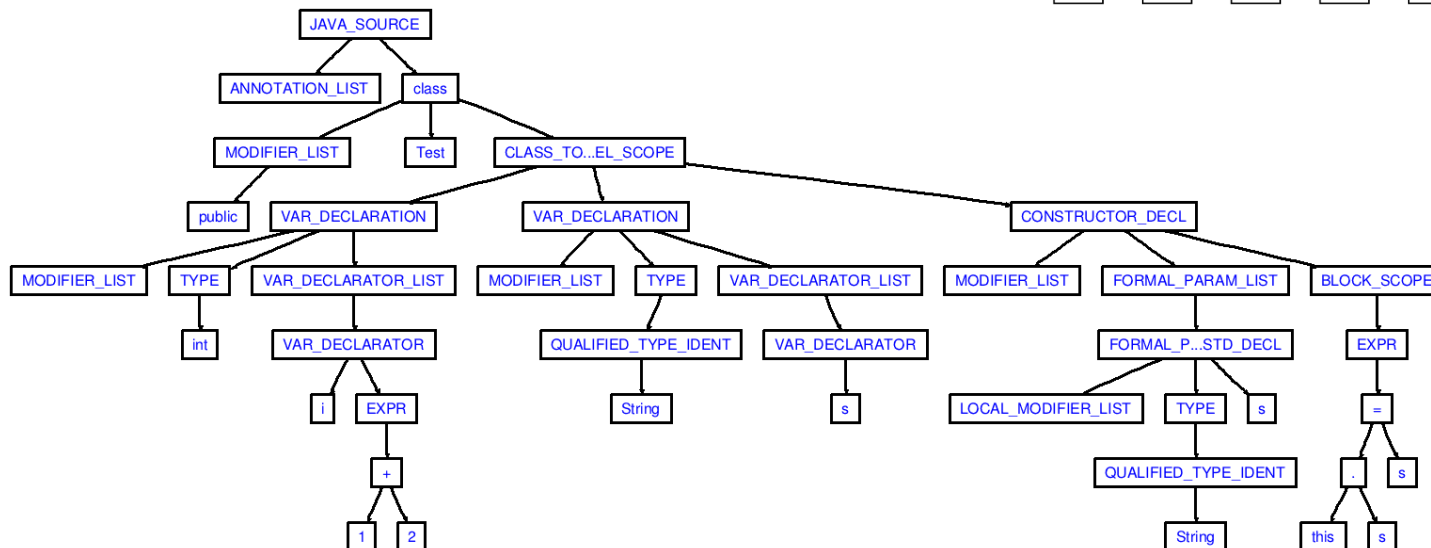
# Abstract Syntax Tree

- Abstracts from unnecessary syntax information as, for example, parentheses is not needed due to tree structure

```
x=1  
y=2  
3*(x+y)
```

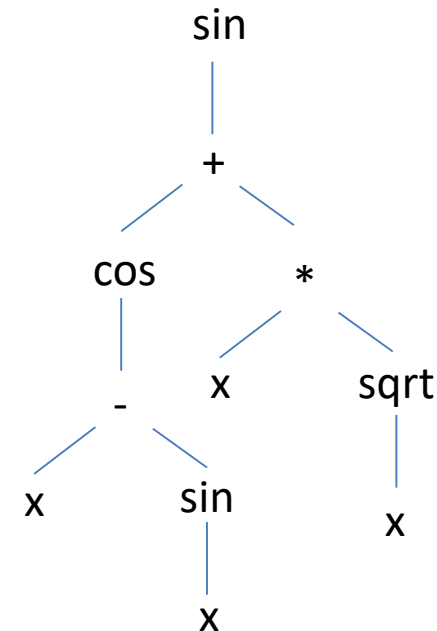


## Java AST Example



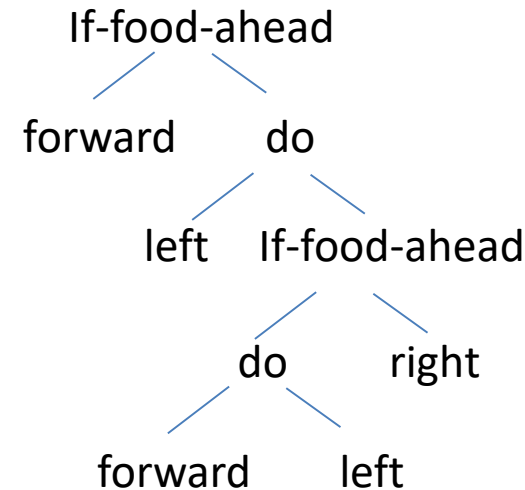
# Example: Symbolic Regression

- We aim at finding a function/program  $f(x)$  that fits best a given data set in the form:  $\langle x_i, f(x_i) \rangle$ 
  - How to find  $f(x)$  for an arbitrarily complex function? -> Symbolic regression
  - We generate many functions as on the right and evaluate their fitness
- Fitness evaluation: execute the program for all  $x_i$  and obtain the result  $r_i$  and compare it against the given  $f(x_i)$  using sum of squares:  $\varepsilon_i = (r_i - f(x_i))^2$



# Example: Artificial Ant

- Common test program: Generate a program for a maze/field, which consumes the most food when executed (multiple times)
- Simple form of problem solution, artificial intelligence
  - Imagine new sorting algorithms this way
  - Imagine database queries to be generated this way






# Initializing Trees

- Idea: Build a tree by consecutive selections from a set of functions (nodes in the tree) and connecting them
- Functions have an **arity**, defining the number of children:
  - X & forward have 0 children; do & + have two
  - 0-Child nodes are **leaf nodes**, so connecting nodes need to respect the arity of functions
- Approach: Grow a tree up to a desired depth

# Grow Algorithm

$max \leftarrow$  maximum valid depth

$FunctionSet \leftarrow$  set of functions to be used to construct the tree

**return**  $DoGrow(1, max, FunctionSet)$   Start recursion with root level 1

**procedure**  $DoGrow(depth, max, FunctionSet)$

**if**  $depth \geq max$  **then**  Maximum depth, we need a leaf node to abort recursion

**return**  $Copy(\text{a randomly chosen leaf node from } FunctionSet)$

**else**

$n \leftarrow Copy(\text{a randomly chosen node from } FunctionSet)$

$l \leftarrow$  number of child nodes expected for  $n$

**for**  $i$  from 1 to  $l$  **do**

$Child[i]$  of  $n \leftarrow DoGrow(depth + 1, max, FunctionSet)$

**return**  $n$

 Recursion step for each child node of the current node

If this is defined as choosing from nonleaf node: it is the **full algorithm**, forcing the grow till maximum size

# Initialization: Ramped Half-and-Half

$minMax \leftarrow$  minimum allowed maximum depth  
 $maxMax \leftarrow$  maximum allowed maximum depth  
 $FunctionSet \leftarrow$  function set

$d \leftarrow$  random integer chosen uniformly from  $minMax$  to  $maxMax$  inclusive  
**if**  $0.5 <$  random value chosen uniformly from 0.0 to 1.0 **then**  
    **return**  $DoGrow(1, d, FunctionSet)$   
**else**  
    **return**  $DoFull(1, d, FunctionSet)$

Problem: No control over size of the tree; unnatural forms are common

# PTC2 Algorithm

$s \leftarrow$  desired tree size

$FunctionSet \leftarrow$  function set

**if**  $s == 1$  **then**

**return**  $Copy$ (a randomly chosen **leaf node** from  $FunctionSet$ )

**else**

$Q \leftarrow \{ \}$

$r \leftarrow Copy$ (a randomly chosen **nonleaf node** from  $FunctionSet$ )

$c \leftarrow 1$

**for** each child argument slot  $b$  of  $r$  **do**

$Q \leftarrow Q \cup \{b\}$

**while**  $c + ||Q|| < s$  **do**  Grow tree as long as nodes + their arguments are below  $s$

$a \leftarrow$  an argument slot removed at random from  $Q$

$m \leftarrow Copy$ (a randomly chosen **nonleaf node** from  $FunctionSet$ )

$c \leftarrow c + 1$

        Fill slot  $a$  with  $m$   Fill a random argument slot with a random nonleaf node

**for** each child argument slot  $b$  of  $m$  **do**

$Q \leftarrow Q \cup \{b\}$   Add the arguments of the newly added node to the list

**for** each argument slot  $q \in Q$  **do**

$m \leftarrow Copy$ (a randomly chosen **leaf node** from  $FunctionSet$ )

        Fill slot  $q$  with  $m$   We are at our desired size, so fill all loose ends with leaf nodes

**return**  $r$

} Root node +  
argument slots  
added

# How to Handle Constants?

- We cannot include every possible constant in the *FunctionSet*
- Idea: Include a special placeholder, called **ephemeral random constant (ERC)**, which gets transformed during inclusion to a randomly generated constant
- This constant may be a good candidate for later mutation

# Recombining Trees

- Idea: Subtree crossover
  - Select a random subtree (root is possible as well) in each individual
  - Swap those two subtrees
  - Often, 10% leaf nodes and 90% nonleaf nodes

$r \leftarrow$  root node of the tree

$f(node) \leftarrow$  function return *true* if the node is of the desired type

**global**  $c \leftarrow 0$

$CountNodes(r, f) \leftarrow$  How does this work?

**if**  $c == 0$  **then**

**return** *null*  $\leftarrow$  There is no node with the desired type

**else**

$a \leftarrow$  random integer from 1 to  $c$  inclusive

$c \leftarrow 0$

**return**  $PickNode(r, a, f) \leftarrow$  How does this work?

# Helper Methods

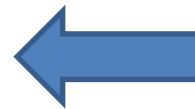
**procedure** *CountNodes*(*r*, *f*)

**if** *f*(*r*) **is true then**

$c \leftarrow c + 1$

**for each child** *i* **of** *r* **do**

*CountNodes*(*r*, *f*)



Depth-first search

**procedure** *PickNode*(*r*, *a*, *f*)

**if** *f*(*r*) **is true then**

$c \leftarrow c + 1$

**if**  $c \geq a$  **then**

**return** *r*

← Reached our random number, so return current node

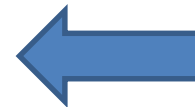
**for each child** *i* **of** *r* **do**

$v \leftarrow \text{PickNode}(i, a, f)$

**if**  $v \neq \text{null}$  **then**

**return** *v*

**return** *null*



Depth-first search

# Mutating Trees

- Often not necessary, because already crossover is highly mutative
- Subtree mutation: Replace a randomly chosen subtree with a randomly generated tree with a max-depth of 5 (pick leaf nodes 10% and inner nodes 90% of the time)
- Replace a non-leaf node with one of its subtrees
- Pick a random non-leaf node and swap its subtrees
- Mutate ephemeral random constants
- Select two independent subtrees and swap them



# Forests

- GP can maintain a vector of trees
  - So, subtasks might be divided into subtrees
  - Good, when typical functions are repeatedly used
- Idea: break a program into several functions, each is represented as a tree or even in more fine-grained trees
- Overarching tree is responsible for arranging the subtrees (e.g., execution order) and calling the methods
  - Requires an additional leaf-node per argument of the to-be-called function (subtree)
  - The arguments appear in the subtree as additional elements in the *FunctionSet*

# Strongly-Typed GP

- Variant of GP, in which we add nodes of a certain type in the tree, such that we do not have all nodes return the same type (e.g., floating point), but there can be other types as well (e.g., Boolean types for if-then-else constructs)
- Closure problem: Mutation and crossover becomes hard, as we have to consider the type of the nodes
- Solution: Add type constraints to each node to specify, which one can be joined / work with other ones
  - Atomic typing
  - Set typing
  - Polymorphic typing

# Cellular Encoding

- Idea: Generate a program that generates a data structure (e.g., a graph or a state machine)
- How would you do that?
  - Have a *FunctionSet* that consists of functions that generate edges and nodes or other elements of your data structure
  - Generate a tree that assembles these functions
  - Execute the tree means to start with an empty (or given) data structure and manipulate this data structure for each node in the tree
  - The quality of your tree is evaluated with the quality of the generated data structure (the data structure is your phenotype)
- Used for generating RNA sequences

# Take Home Message

---

- There is a gap between the real world and how we encode optimization
- Phenotype and genotype
  - Tweaking in real world and in our representation
  - Both should match
- Adapt tweaking, selection, and other operators to your representation
- Different encodings for different problems
  - Cellular vs. trees vs. graphs vs. ...
  - Encodings require special operators