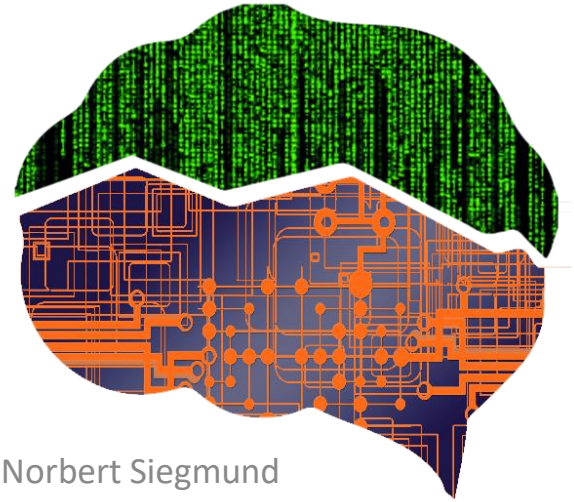
Machine Learning for Software Engineering

Exercise: Representation



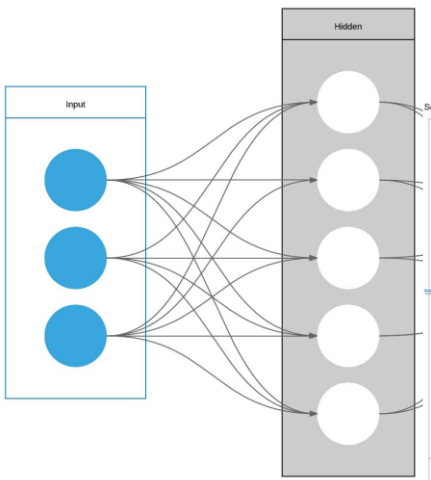
Prof. Dr.-Ing. Norbert Siegmund
Intelligent Software Systems

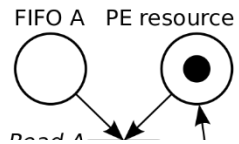
Bauhaus-Universität Weimar

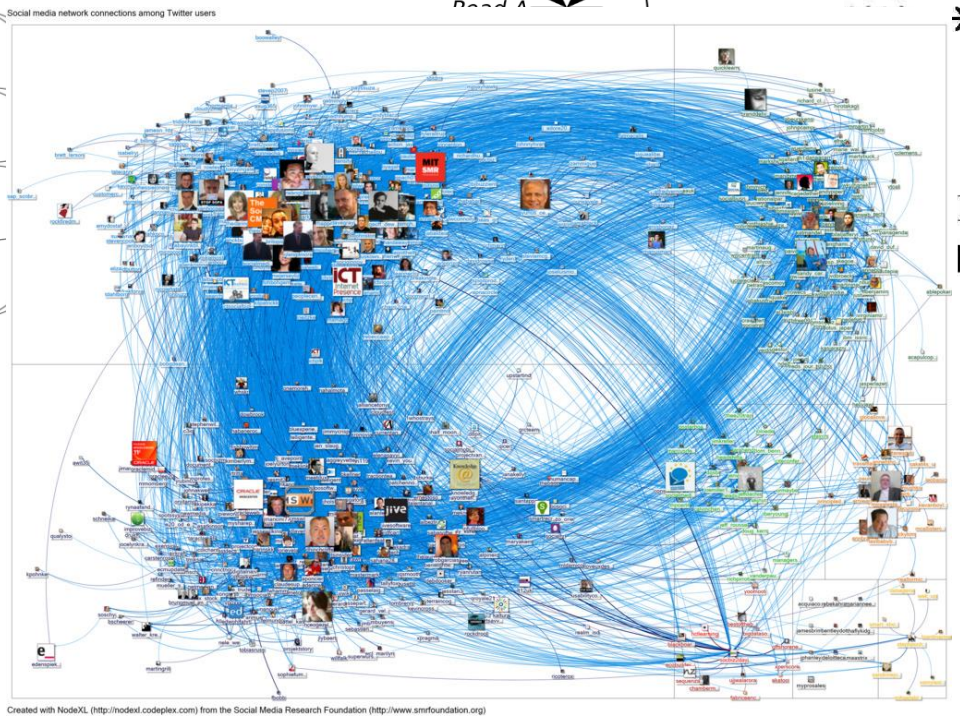
Direct Encoded Graphs

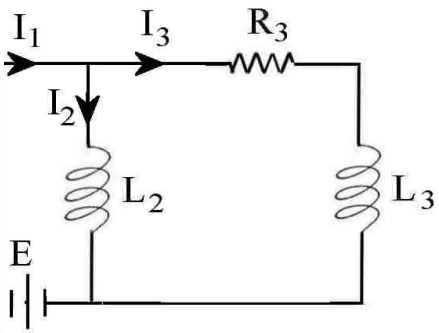
Graphs

- 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

$$\begin{bmatrix} 0.5 & 0.7 & -0.1 & 0.2 & No \\ No & -0.5 & -0.8 & 0.4 & No \\ 0.6 & 0.7 & 0.8 & No & -0.4 \\ -0.1 & No & No & 0.2 & No \\ 0.2 & No & -0.7 & No & No \end{bmatrix}$$

$\langle v_1, \dots, v_{25} \rangle$ where *No* = 0.0

$\langle v_1, \dots, v_{25} \rangle$ where each entry is the weight
$\langle b_1, \dots, b_{25} \rangle$ where each entry is a Boolean defining whether there is an edge

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

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

$n \leftarrow m - 1$
repeat
  $n \leftarrow n + 1$
until $p <$ random number chosen uniformly between 0.0 to 1.0 inclusive
return n

Graph Construction

$n \leftarrow$ computed number of nodes
$e \leftarrow$ computed number of edged
$f(j, k, Nodes, Edges) \leftarrow$ validity–check function returns $\boldsymbol{true}$ if an edge from j to k is valid

$N \leftarrow \{N_1, \ldots, N_n\}$ set of nodes
$E \leftarrow \{\quad\}$ set of edges
for each node $N_i \in N$ **do**
  $ProcessNode(N_i)$ ← Function to assign labels, weigths, 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 $\boldsymbol{true}$ ← Could take **very** long
  $g \leftarrow$ new edge from N_j to N_k
  $ProcessEdge(g)$ ← Function to assign labels, weigths, 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

$N \leftarrow \{N_1, \ldots, N_n\}$ set of nodes
$E \leftarrow \{\quad\}$ 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 ≥ 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 α_1
  - Add a random edge with probability α_2
  - Delete a node with all its edges with probability α_3
  - Add a node with probability α_4
  - Relabel a node with probability α_5
  - Relabel an edge with probability α_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**
  $S' \leftarrow S' \cup \{$random element from S 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)
$E' \leftarrow \{\quad\}$ 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 unifromly 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 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 limit the combination with the results or values of other operations

Continued

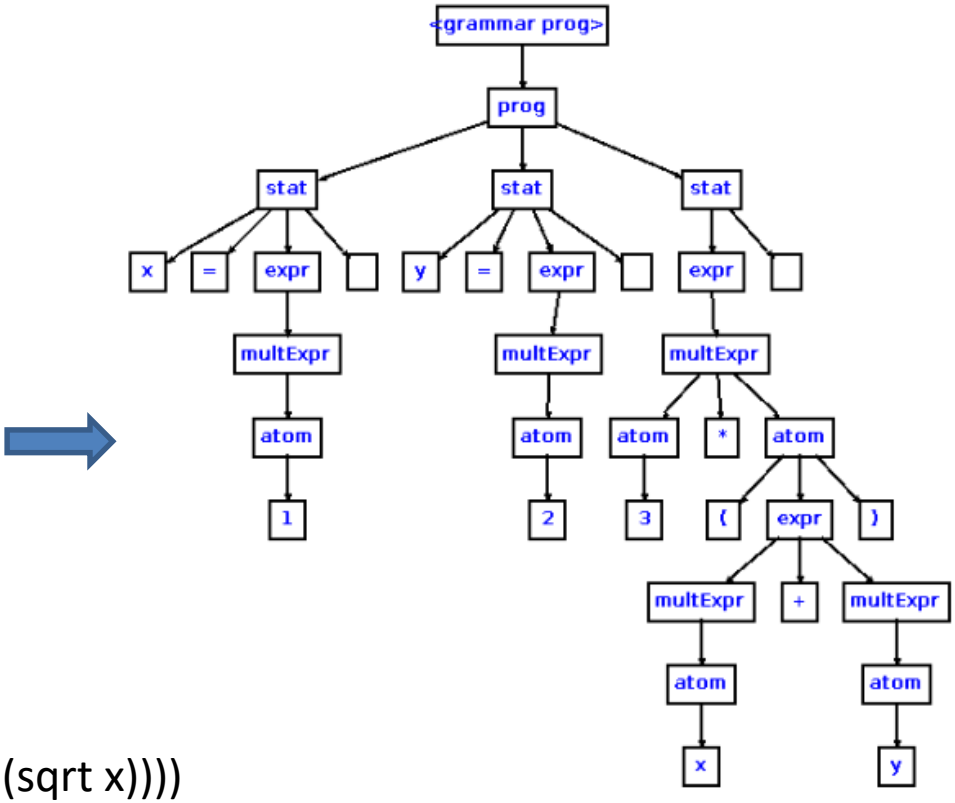
- Nodes in a tree may define certain number of children
  - Multiplication vs. increment
- So, initialization and mutation aims to maintain **closure**
  - Stay in the valid solution space
- Fitness assessment is usually done by executing the program
  - Data of genotype must somehow correspond to the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

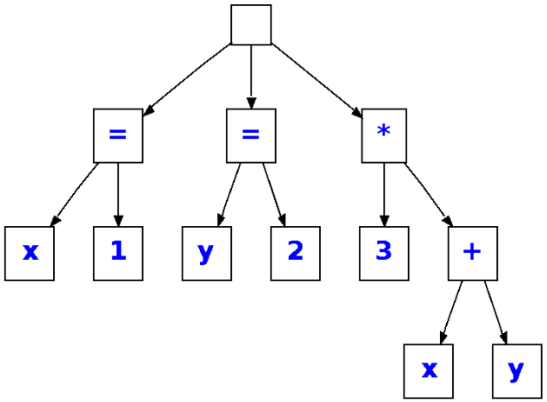


Program in Lisp: (sin (+ (cos (- x (sin x))) (* x (sqrt x))))

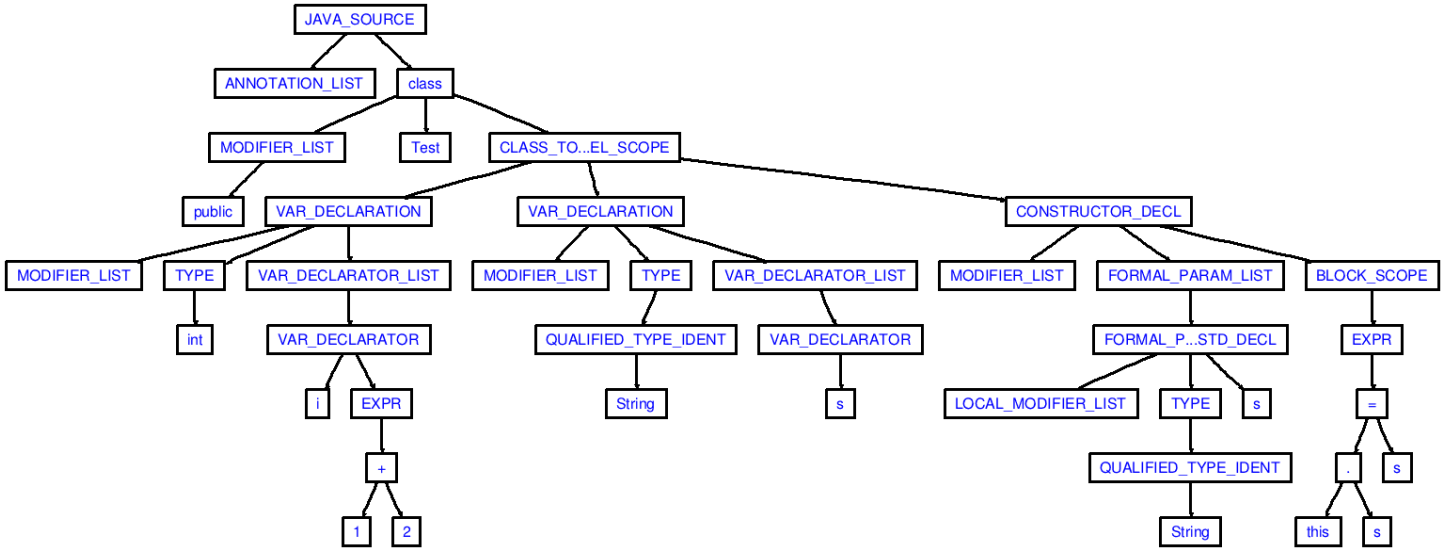
Abstract Syntax Tree

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

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

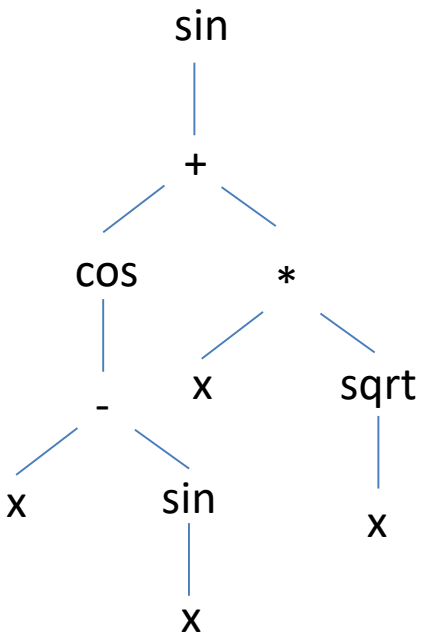


Java AST Example



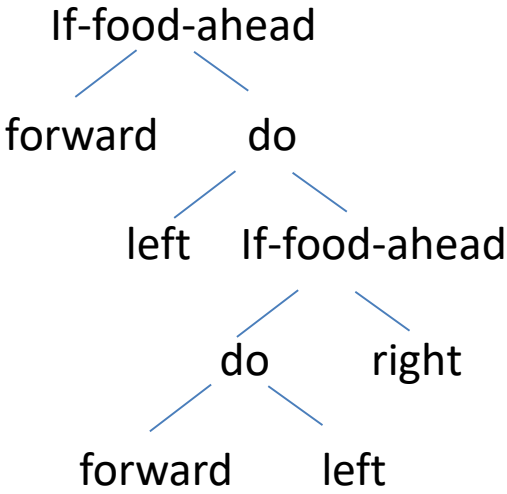
Example: Symbolic Regression

- 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: $\varepsilon_i = \left(r_i - f(x_i)\right)^2$



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

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

procedure $DoGrow(deph, max, FunctionSet)$
  if $depth \geq max$ **then** ← Maximum depth, we need a leaf node to abort recursion
    return $Copy$(a randomly chosen **leaf node** from $FunctionSet$)
  else
    $n \leftarrow Copy$(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**
    $Q \leftarrow Q \cup \{b\}$

Root node + argument slots added

  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

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 $c \leftarrow 0$
$CountNodes(r, f)$ ← How does this work?
if $c == 0$ **then**
  return $\boldsymbol{null}$ ← There is no node with the desired type
else
  $a \leftarrow$ random integer from 1 to c inclusive
  $c \leftarrow 0$
  return $PickNode(r, a, f)$ ← 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
  for each child i of r **do**
    $v \leftarrow PickNode(i, a, f)$
    if $v! = \boldsymbol{null}$ **then**
      return v
  return $\boldsymbol{null}$

Depth-first search

Reached our random number, so return current node

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