Search-Based Software Engineering

Genetic Programming
Part 1

Gordon Fraser
Lehrstuhl für Software Engineering II

John Koza: Genetic Programming: On the Programming of Computers by Means of Natural Selection, 1992

Genetic Programming

1: Randomly create an *initial population* of programs from the available primitives (more on this in Section 2.2).
2: **repeat**
3: *Execute* each program and ascertain its fitness.
4: *Select* one or two program(s) from the population with a probability based on fitness to participate in genetic operations (Section 2.3).
5: Create new individual program(s) by applying *genetic operations* with specified probabilities (Section 2.4).
6: **until** an acceptable solution is found or some other stopping condition is met (e.g., a maximum number of generations is reached).
7: **return** the best-so-far individual.

Algorithm 1.1: Genetic Programming

GA vs. GP

- GA operates with a population of potential solutions which are iteratively improved using the mechanisms of selection, crossover, mutation, and replacement
- GP operates with a population of potential solutions which are iteratively improved using the mechanisms of selection, crossover, mutation, and replacement
- There are two main distinctions between GP and GA:
  1. The form of representation used:
     1. GA operates on genotypes; GP operates on phenotypes
     2. GP Representation = trees
  2. The more open-ended nature of the evolutionary process in GP:
     1. Size is variable not fixed length
     2. Number of elements used in the final solution as well as their interconnections must be open to evolution

Programs, not numbers

- Programs are highly structured.

- GP mostly (but not exclusively) uses syntax tree to represent solutions.

  - max(x + x, x + 3 * y)
    = (max (+ x x) (+ x (* 3 y)))

- Obviously, it is easier to implement GP with some languages: high-level ADT, garbage collection, etc

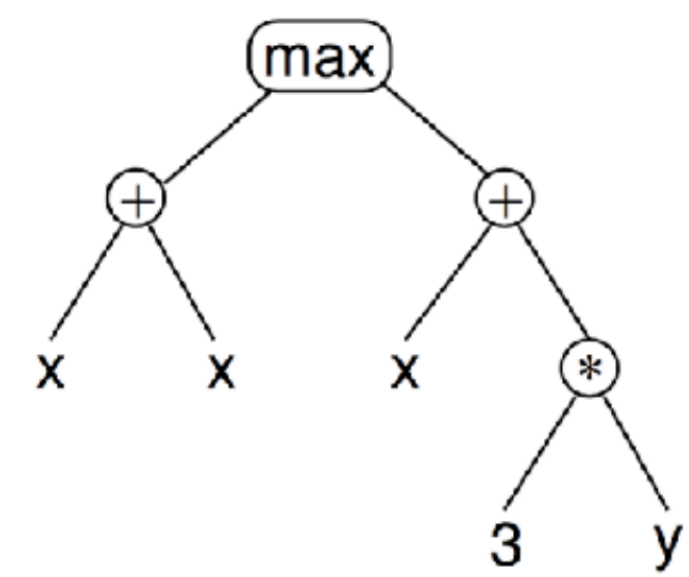
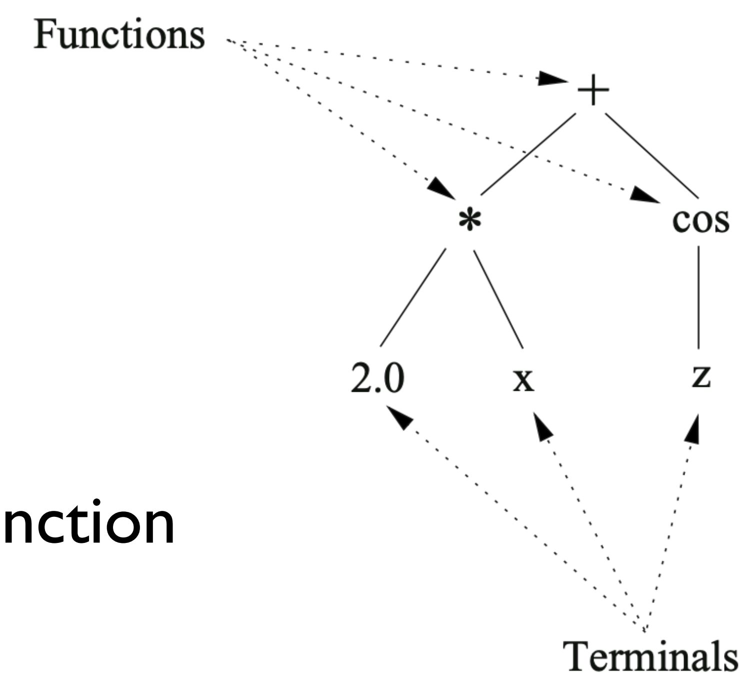


Figure 2.1: GP syntax tree representing max(x+x,x+3*y).

Classical GP



- Programs generated using two sets:
  - Terminal set: Items of arity 0
  - Function set: Items of arity > 0
- The input to a GP function can be the result of another function
- Required property: Type Closure
  - Each function should be able to handle gracefully all values it might ever receive as input
  - All terminals must be allowable inputs for all functions
  - The output from any function must be a permitted input to any other function
  - Closure ensures generated programs are syntactically correct

Sufficiency

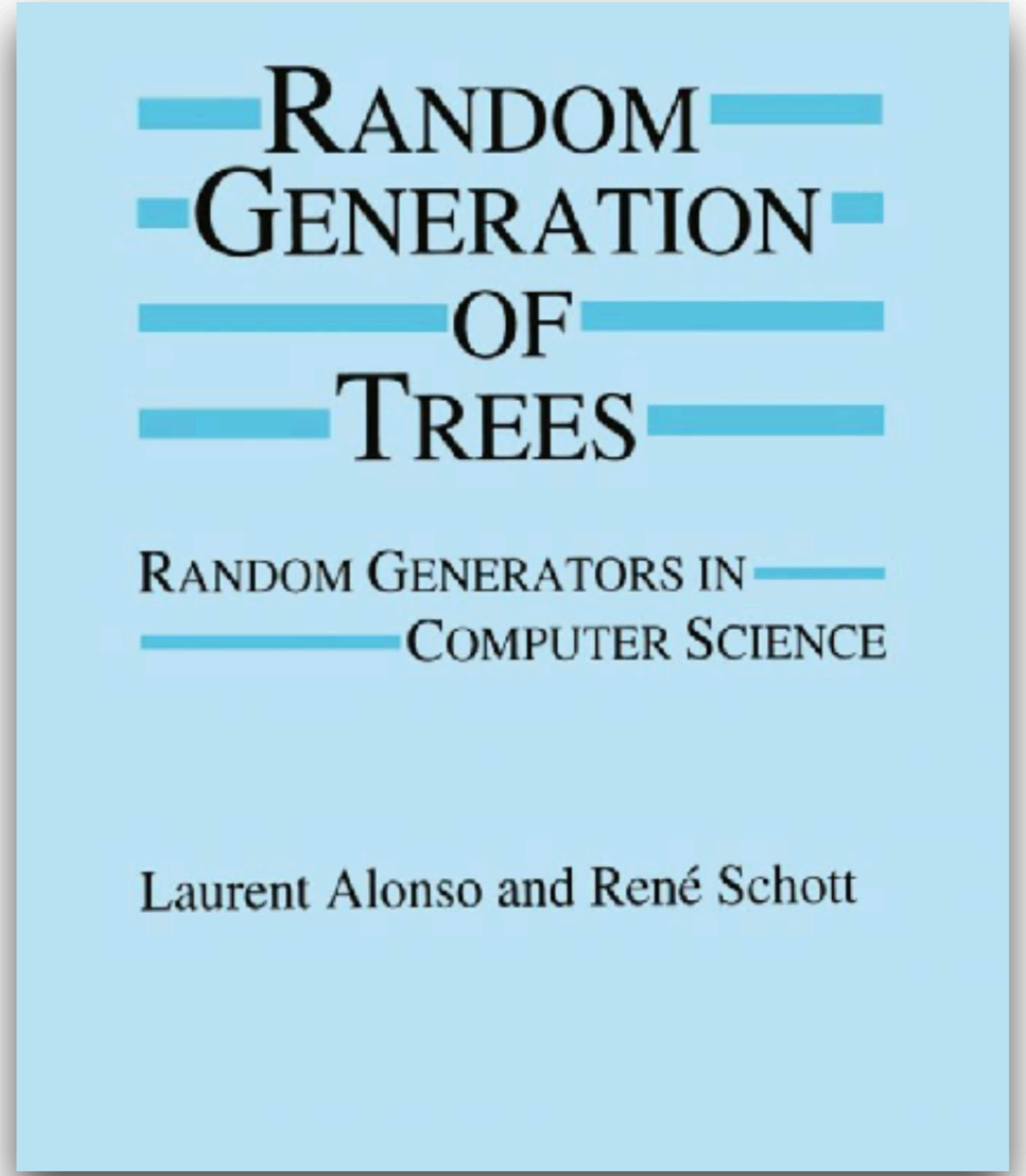
- Is the given set of terminals and non-terminals sufficient to express the solution to the problem?
- Unless there is a theoretical guarantee that comes WITH the problem, this is hard to answer.
- We can always approximate.

Ephemeral Random Constants

- It is impossible to include all real numbers in the terminal set
- Ephemeral random constants (ERCs):
  - Every time this terminal is chosen in the construction of an initial tree (or a new subtree to use in an operation like mutation), a different random value is generated which is then used for
    that particular terminal, and which will remain fixed for the rest of the run.
  - After the initial generation, new constants are created through the recombination of existing ERCs through arithmetic expressions
  - The value of the constant is constant for a tree, but may differ from one tree to another

Initialisation

- What is a random tree?
- We need to limit the size of the tree (i.e. depth): we do not want arbitrarily large trees as solutions.
- Many initialisation methods: full, grow, ramped half-and-half, and others.



https://coinse.kaist.ac.kr/assets/files/teaching/cs454/cs454-slide09.pdf

Full Initialisation

- Grow full trees
  - Add non-terminal nodes only until the depth limit is reached.
  - Then only add terminals as leaves.
  - All trees are fully grown.

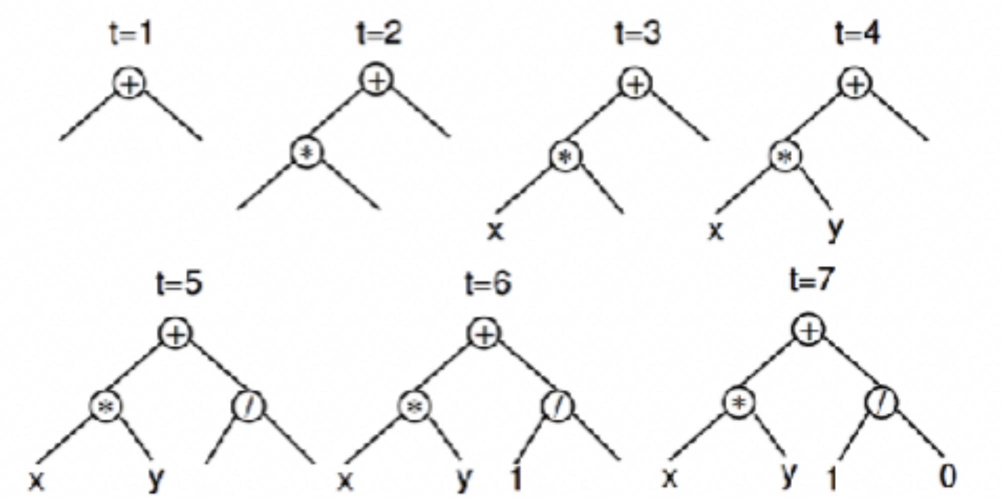


Figure 2.3: Creation of a full tree having maximum depth 2 using the `full` initialisation method (t = time).

https://coinse.kaist.ac.kr/assets/files/teaching/cs454/cs454-slide09.pdf

Grow Initialisation

- Grow various trees
- Add any node while there are empty slots and the depth limit is not reached.
- Results in trees of various sizes, but the ratio between terminals and non-terminals will bias the average size.

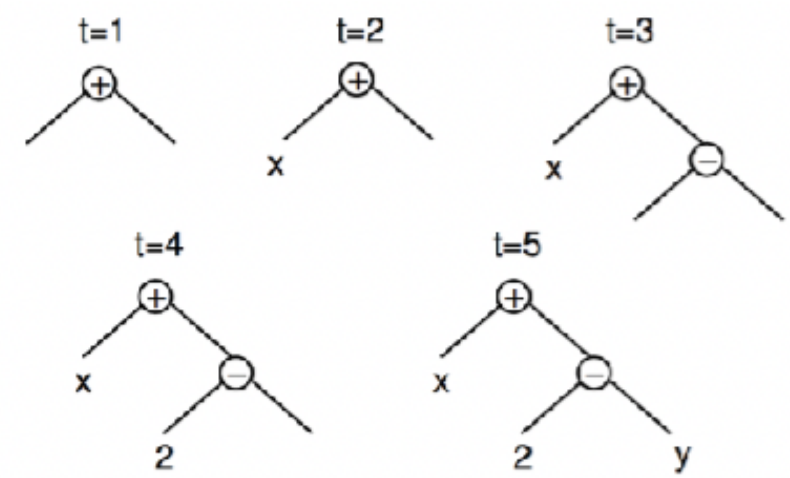
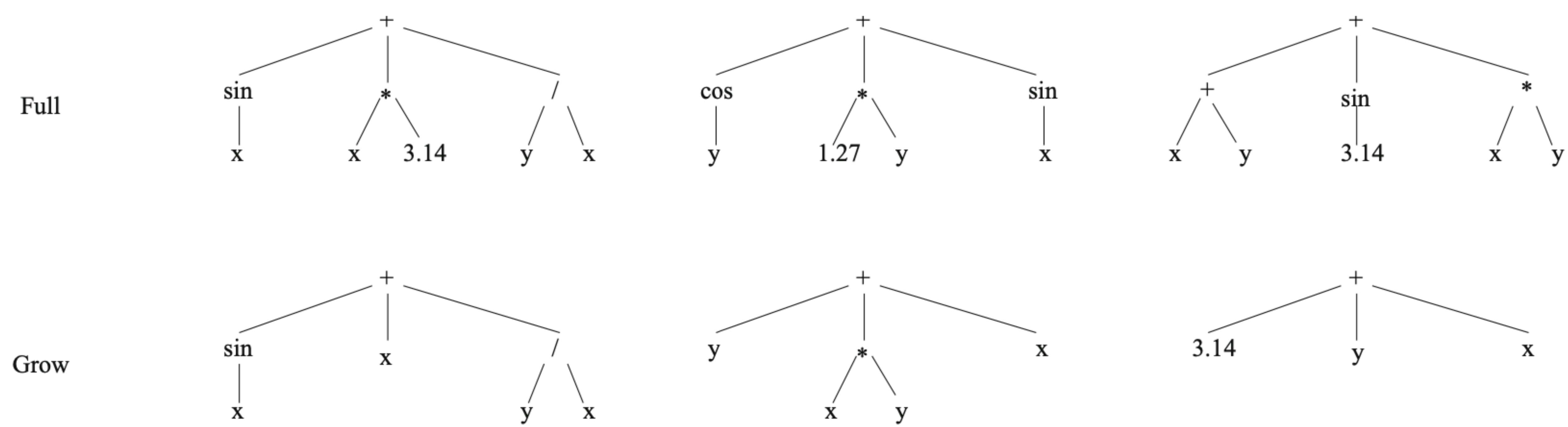


Figure 2.4: Creation of a five node tree using the **grow initialisation** method with a maximum depth of 2 (t = time). A terminal is chosen at $t = 2$, causing the left branch of the root to be closed at that point even though the maximum depth had not been reached.

https://coinse.kaist.ac.kr/assets/files/teaching/cs454/cs454-slide09.pdf

Full vs. Grow


Full
Grow
+
sin
*
/
x
x
3.14
y
x
+
cos
*
sin
y
1.27
y
x
+
+
sin
*
x
y
3.14
x
y
+
sin
x
/
x
y
x
+
y
*
x
x
y
+
3.14
y
x

Ramped Half and Half

- Half of population is initialised with Full method
- Half of population is initialised with Grow method
- A better diversity in terms of shapes and size
- Ramped method tends to generate *bushy* trees. Some programs have highly asymmetric shape, which is hard to achieve with ramped method.
- Various methods have been developed to sample trees with sizes that are more uniformly distributed (highly sophisticated combinatorics).

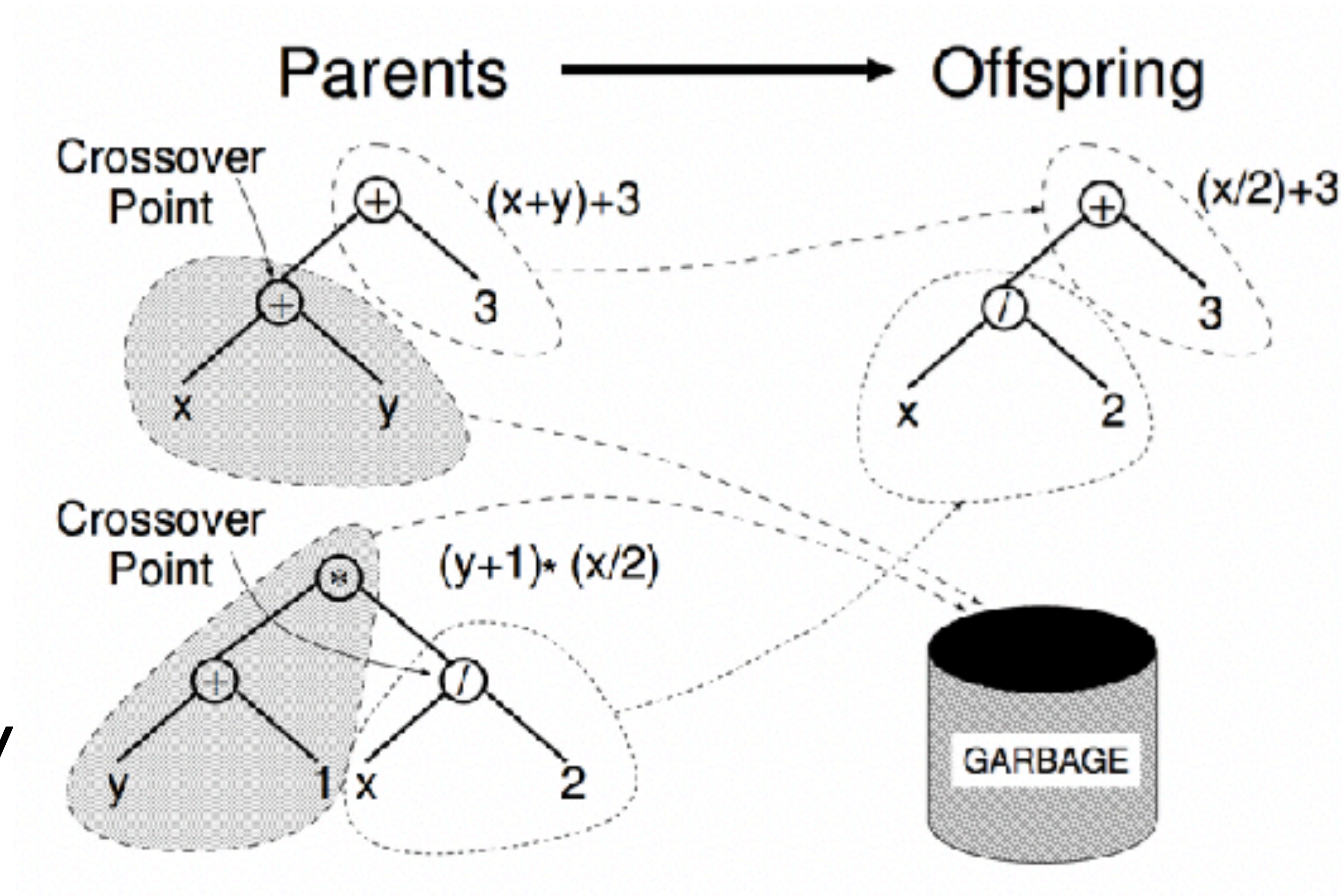
https://coinse.kaist.ac.kr/assets/files/teaching/cs454/cs454-slide09.pdf

Selection

- Nothing different really, except:
  - GP evolves programs;
  - The fitness of the program is usually measured by executing the candidate program;
  - This can be time consuming, despite the evaluation essentially being inherently parallel.

Crossover

- Initial idea
- Randomly choose two crossover points in parent trees;
- Cut and swap subtrees below the crossover points.



https://coinse.kaist.ac.kr/assets/files/teaching/cs454/cs454-slide09.pdf

Crossover

- Often crossover points are NOT sampled with random distribution:
  - Average branching factor is 2 or more, which means the majority of the nodes are leaves, which means the majority of branches will simply cut a single leaf.
  - Type-aware crossover (Koza 1992):
    - 90% chance of choosing a non-terminal node
    - 10% chance of choosing a terminal node

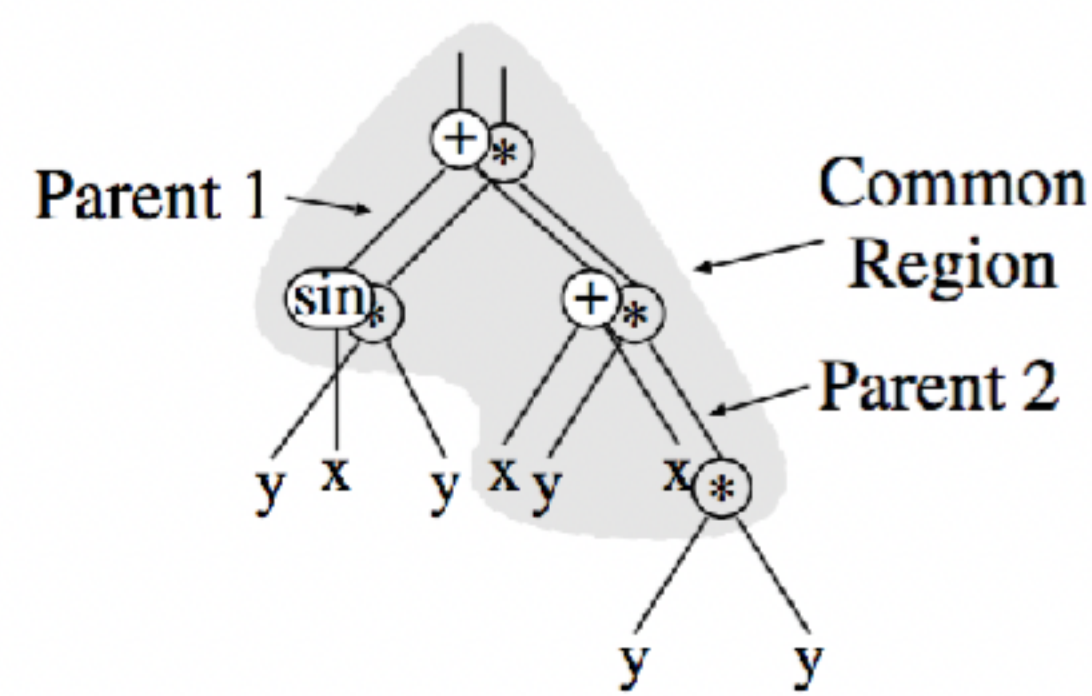
https://coinse.kaist.ac.kr/assets/files/teaching/cs454/cs454-slide09.pdf

Uniform Crossover

- Find the common region between two parents.

- For each node in the common region, flip a coin to decide whose node to take; when taking a non-terminal node, take its subtree.

- Mixes code nearer to the root more often, compared to other crossover operators.



https://coinse.kaist.ac.kr/assets/files/teaching/cs454/cs454-slide09.pdf

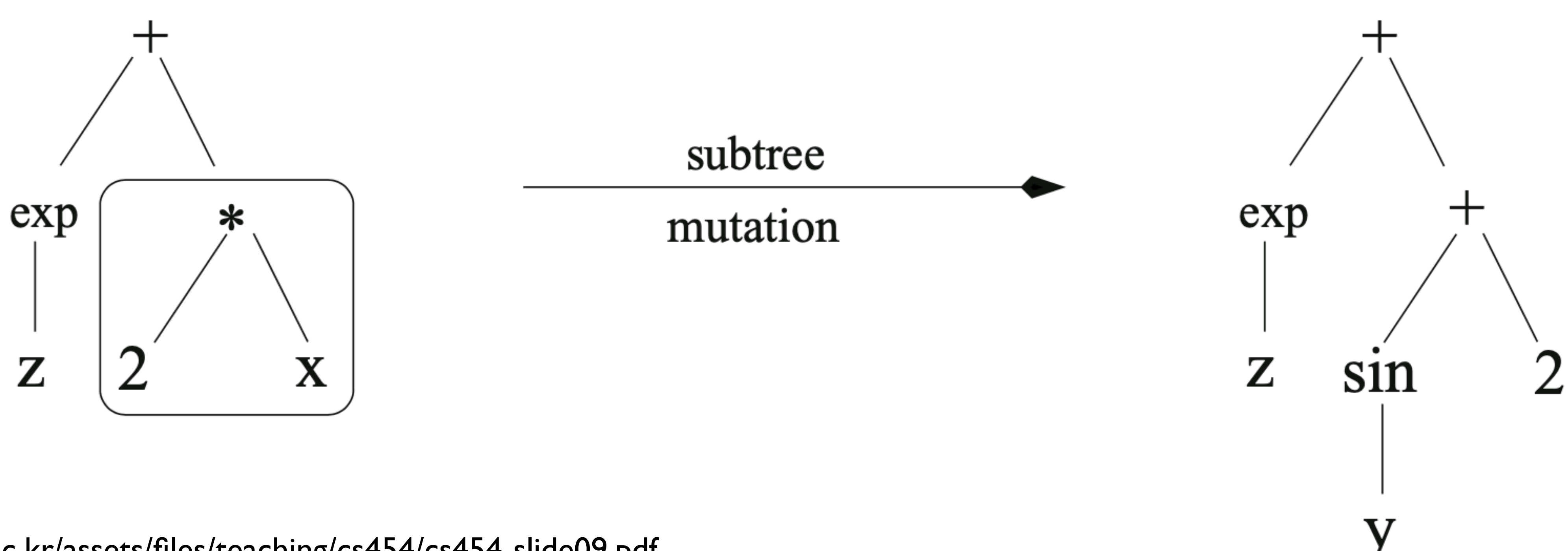
Size-fair Crossover

- First crossover point in one parent chosen randomly;
- Measure the subtree size;
- Constrain the size of the second subtree to be chosen from the other parent

https://coinse.kaist.ac.kr/assets/files/teaching/cs454/cs454-slide09.pdf

Subtree Mutation

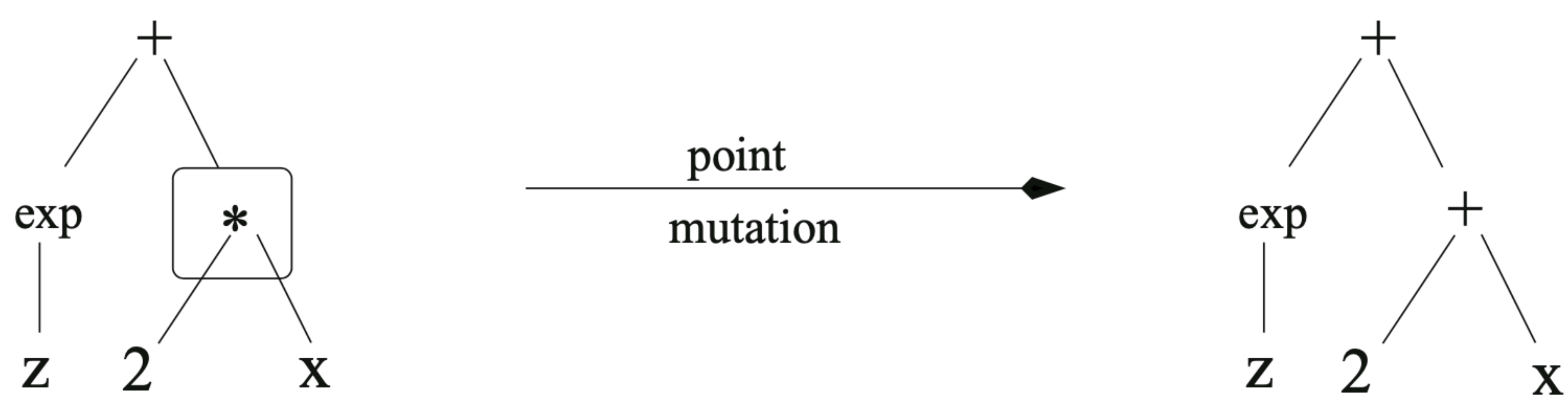
- Subtree mutation (a.k.a. headless chicken mutation):
    - Choose a subtree
    - Replace it with a randomly generated subtree



https://coinse.kaist.ac.kr/assets/files/teaching/cs454/cs454-slide09.pdf

Point Mutation

- For each node:
    - With a certain probability, replace the node with another node of same arity.
- Independently consider all nodes; may mutate more than one.



https://coinse.kaist.ac.kr/assets/files/teaching/cs454/cs454-slide09.pdf

... and many more

- Hoist mutation: create a new individual, which is a randomly chosen subtree of the parent.
- Shrink mutation: replace a randomly chosen subtree with a randomly chosen terminal node.
- Permutation mutation: change the order of function arguments in trees.
- Systematic constant mutation: use external optimisation to tune the constants in the expression tree.

https://coinse.kaist.ac.kr/assets/files/teaching/cs454/cs454-slide09.pdf

Bloat

- Average size of trees in the population remains relatively static for certain number of generations, then:
- It increases rapidly and significantly. This growth in size is not accompanied by improvement in fitness.
- Many attempt to explain why this happens; no unified theory yet.

Three Theoretical Attempts

- Replication accuracy theory (McPhee and Miller 1995): success of a GP individual depends on its ability to have offsprings that are functionally similar to itself, hence the tendency to repeat itself.
- Removal bias theory (Soule and Foster 1998): inactive (dead) code usually lies lower in the tree, and are smaller than average. When replaced (and removed), larger subtrees take their place, increasing the tree size.
- Program Search Space theory (Langdon and Poli 1997): above certain size, there is no correlation between size and fitness, but there are more longer programs, so they are just sampled more often.

Bloat Control

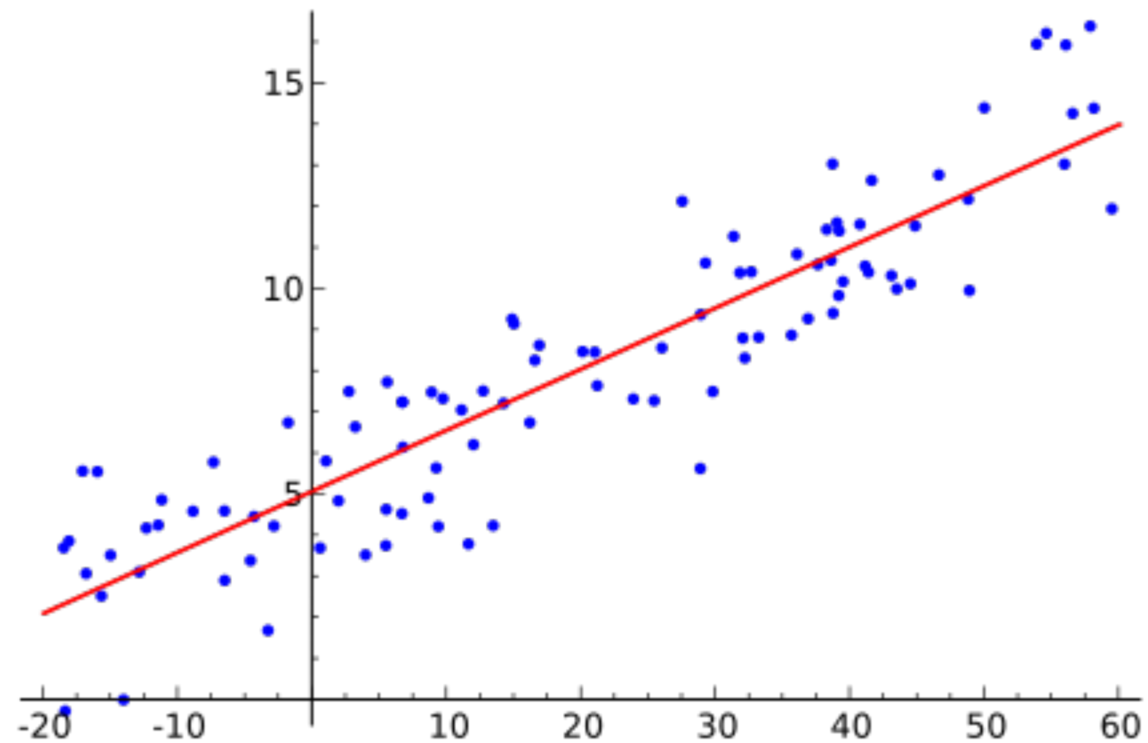
- Size and depth limit: do not accept too large individuals into population
- Bloat-aware genetic operators: do not generate too large individuals
- Bloat aware selection: consider program size as part of selection pressure

Symbolic Regression

- Regression analysis estimates the relationship between variables: for example, linear regression is to find the set of (a, b, c) such that y = ax + b fits the given data points with minimum error

- Symbolic regression searches for the model itself in the space of all possible equations

- Fitness: The usual choice for fitness is to minimise MSE (Mean Square Error): for each data point, measure the squared error, and get their average.



Example: Evolving Fault Localisation Formulas

- Shin Yoo. "Evolving human competitive spectra-based fault localisation techniques." International Symposium on Search Based Software Engineering. Springer, Berlin, Heidelberg, 2012.

Fault Localisation

```
int mid(int x, int y, int z) {
  int m;
  m = z;
  if (y < z) {
    if (x < y)
      m = y;
    else if (x < z)
      m = y;
  } else {
    if (x > y)
      m = y;
    else if (x > z)
      m = x;
  }
  return m;
}
```

Fault Localisation

mid(3,3,5) == 3	Pass
mid(1,2,3) == 2	Pass
mid(3,2,1) == 2	Pass
mid(5,5,5) == 5	Pass
mid(5,3,4) == 4	Pass
mid(2,1,3) == 2 1	Fail

Fault Localisation

Statement	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
int m;	✓	✓	✓	✓	✓	✓
m = z;	✓	✓	✓	✓	✓	✓
if(y < z) {	✓	✓	✓	✓	✓	✓
if(x < y)	✓	✓			✓	✓
m = y;		✓				
else if(x < z)	✓				✓	✓
m = y;	✓					✓
} else {			✓	✓		
if(x > y)			✓	✓		
m = y;			✓			
else if(x > z)				✓		
m = x;						
return m;	✓	✓	✓	✓	✓	✓
	Pass	Pass	Pass	Pass	Pass	Fail

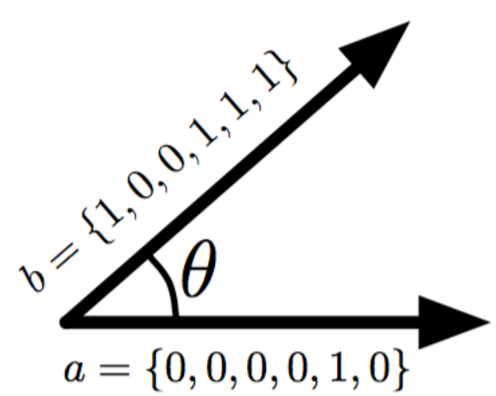
Error Vector

Suspiciousness score

- Insight: Entities in a program that are primarily executed by failed test cases are more likely to be faulty than those that are primarily executed by passed test cases

- Each component (row) is **ranked** according to their **similarity** to the **error vector**

  - Many similarity coefficients exist.



$$\mathit{Ochiai}(a, b) = \cos(\theta)$$

- **Ochiai** similarity is equivalent to the cosine of the angle between two vectors in an n-dimensional space

Fault Localisation

Tarantula $$S(s) = \frac{failed(s)/totalfailed}{failed(s)/totalfailed + passed(s)/totalpassed}$$

Ochiai $$S(s) = \frac{failed(s)}{\sqrt{totalfailed \cdot (failed(s) + passed(s))}}$$

Op2 $$S(s) = failed(s) - \frac{passed(s)}{totalpassed + 1}$$

Barinel $$S(s) = 1 - \frac{passed(s)}{passed(s) + failed(s)}$$

DStar $$S(s) = \frac{failed(s)^*}{passed(s) + (totalfailed - failed(s))}$$

Fault Localisation

Statement	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	Susp
`int m;`	✓	✓	✓	✓	✓	✓	
`m = z;`	✓	✓	✓	✓	✓	✓	
`if(y < z) {`	✓	✓	✓	✓	✓	✓	
`if(x < y)`	✓	✓			✓	✓	
`m = y;`		✓					
`else if(x < z)`	✓				✓	✓	
`m = y;`	✓					✓	
`} else {`			✓	✓			
`if(x > y)`			✓	✓			
`m = y;`			✓				
`else if(x > z)`				✓			
`m = x;`							
`return m;`	✓	✓	✓	✓	✓	✓	
	Pass	Pass	Pass	Pass	Pass	Fail	

$$\text{Suspicious}(s) = \frac{\text{fail}(s) / \text{totalfail}}{\text{fail}(s) / \text{totalfail} + \text{pass}(s) / \text{total pass}}$$

Fault Localisation

Statement	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	Susp
`int m;`	✓	✓	✓	✓	✓	✓	0.5
`m = z;`	✓	✓	✓	✓	✓	✓	
`if(y < z) {`	✓	✓	✓	✓	✓	✓	
`if(x < y)`	✓	✓			✓	✓	
`m = y;`		✓					
`else if(x < z)`	✓				✓	✓	
`m = y;`	✓					✓	
`} else {`			✓	✓			
`if(x > y)`			✓	✓			
`m = y;`			✓				
`else if(x > z)`				✓			
`m = x;`							
`return m;`	✓	✓	✓	✓	✓	✓	
	Pass	Pass	Pass	Pass	Pass	Fail	

$$\text{Suspicious}(s) = \frac{\text{fail}(s) / \text{totalfail}}{\text{fail}(s) / \text{totalfail} + \text{pass}(s) / \text{total pass}}$$

Fault Localisation

Statement	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	Susp
`int m;`	✓	✓	✓	✓	✓	✓	0.5
`m = z;`	✓	✓	✓	✓	✓	✓	0.5
`if(y < z) {`	✓	✓	✓	✓	✓	✓	
`if(x < y)`	✓	✓			✓	✓	
`m = y;`		✓					
`else if(x < z)`	✓				✓	✓	
`m = y;`	✓					✓	
`} else {`			✓	✓			
`if(x > y)`			✓	✓			
`m = y;`			✓				
`else if(x > z)`				✓			
`m = x;`							
`return m;`	✓	✓	✓	✓	✓	✓	
	Pass	Pass	Pass	Pass	Pass	Fail	

$$\text{Suspicious}(s) = \frac{\text{fail}(s) / \text{totalfail}}{\text{fail}(s) / \text{totalfail} + \text{pass}(s) / \text{total pass}}$$

Fault Localisation

Statement	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	Susp
`int m;`	✓	✓	✓	✓	✓	✓	0.5
`m = z;`	✓	✓	✓	✓	✓	✓	0.5
`if(y < z) {`	✓	✓	✓	✓	✓	✓	0.5
`if(x < y)`	✓	✓			✓	✓	
`m = y;`		✓					
`else if(x < z)`	✓				✓	✓	
`m = y;`	✓					✓	
`} else {`			✓	✓			
`if(x > y)`			✓	✓			
`m = y;`			✓				
`else if(x > z)`				✓			
`m = x;`							
`return m;`	✓	✓	✓	✓	✓	✓	
	Pass	Pass	Pass	Pass	Pass	Fail	

$$\text{Suspicious}(s) = \frac{\text{fail}(s) \,/\, \text{totalfail}}{\text{fail}(s) \,/\, \text{totalfail} + \text{pass}(s) \,/\, \text{total pass}}$$

Fault Localisation

Statement	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	Susp
`int m;`	✓	✓	✓	✓	✓	✓	0.5
`m = z;`	✓	✓	✓	✓	✓	✓	0.5
`if(y < z) {`	✓	✓	✓	✓	✓	✓	0.5
`if(x < y)`	✓	✓			✓	✓	0.63
`m = y;`		✓					
`else if(x < z)`	✓				✓	✓	
`m = y;`	✓					✓	
`} else {`			✓	✓			
`if(x > y)`			✓	✓			
`m = y;`			✓				
`else if(x > z)`				✓			
`m = x;`							
`return m;`	✓	✓	✓	✓	✓	✓	
	Pass	Pass	Pass	Pass	Pass	Fail	

$$\text{Suspicious}(s) = \frac{\text{fail}(s)\ /\ \text{totalfail}}{\text{fail}(s)\ /\ \text{totalfail} + \text{pass}(s)\ /\ \text{total pass}}$$

Fault Localisation

Statement	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	Susp
`int m;`	✓	✓	✓	✓	✓	✓	0.5
`m = z;`	✓	✓	✓	✓	✓	✓	0.5
`if(y < z) {`	✓	✓	✓	✓	✓	✓	0.5
`if(x < y)`	✓	✓			✓	✓	0.63
`m = y;`		✓					0
`else if(x < z)`	✓				✓	✓	
`m = y;`	✓					✓	
`} else {`			✓	✓			
`if(x > y)`			✓	✓			
`m = y;`			✓				
`else if(x > z)`				✓			
`m = x;`							
`return m;`	✓	✓	✓	✓	✓	✓	
	Pass	Pass	Pass	Pass	Pass	Fail	

$$\text{Suspicious}(s) = \frac{\text{fail}(s) \,/\, \text{totalfail}}{\text{fail}(s) \,/\, \text{totalfail} + \text{pass}(s) \,/\, \text{total pass}}$$

Fault Localisation

Statement	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	Susp
`int m;`	✓	✓	✓	✓	✓	✓	0.5
`m = z;`	✓	✓	✓	✓	✓	✓	0.5
`if(y < z) {`	✓	✓	✓	✓	✓	✓	0.5
`if(x < y)`	✓	✓			✓	✓	0.63
`m = y;`		✓					0
`else if(x < z)`	✓				✓	✓	0.71
`m = y;`	✓					✓	
`} else {`			✓	✓			
`if(x > y)`			✓	✓			
`m = y;`			✓				
`else if(x > z)`				✓			
`m = x;`							
`return m;`	✓	✓	✓	✓	✓	✓	
	Pass	Pass	Pass	Pass	Pass	Fail	

$$\text{Suspicious}(s) = \frac{\text{fail}(s) / \text{totalfail}}{\text{fail}(s) / \text{totalfail} + \text{pass}(s) / \text{total pass}}$$

Fault Localisation

Statement	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	Susp
`int m;`	✓	✓	✓	✓	✓	✓	0.5
`m = z;`	✓	✓	✓	✓	✓	✓	0.5
`if(y < z) {`	✓	✓	✓	✓	✓	✓	0.5
`if(x < y)`	✓	✓			✓	✓	0.63
`m = y;`		✓					0
`else if(x < z)`	✓				✓	✓	0.71
`m = y;`	✓					✓	0.83
`} else {`			✓	✓			
`if(x > y)`			✓	✓			
`m = y;`			✓				
`else if(x > z)`				✓			
`m = x;`							
`return m;`	✓	✓	✓	✓	✓	✓	
	Pass	Pass	Pass	Pass	Pass	Fail	

$$\text{Suspicious}(s) = \frac{\text{fail}(s) / \text{totalfail}}{\text{fail}(s) / \text{totalfail} + \text{pass}(s) / \text{total pass}}$$

Fault Localisation

Statement	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	Susp
`int m;`	✓	✓	✓	✓	✓	✓	0.5
`m = z;`	✓	✓	✓	✓	✓	✓	0.5
`if(y < z) {`	✓	✓	✓	✓	✓	✓	0.5
`if(x < y)`	✓	✓			✓	✓	0.63
`m = y;`		✓					0
`else if(x < z)`	✓				✓	✓	0.71
`m = y;`	✓					✓	0.83
`} else {`			✓	✓			0
`if(x > y)`			✓	✓			0
`m = y;`			✓				0
`else if(x > z)`				✓			0
`m = x;`							0
`return m;`	✓	✓	✓	✓	✓	✓	
	Pass	Pass	Pass	Pass	Pass	Fail	

$$\text{Suspicious}(s) = \frac{\text{fail}(s) \,/\, \text{totalfail}}{\text{fail}(s) \,/\, \text{totalfail} + \text{pass}(s) \,/\, \text{total pass}}$$

Fault Localisation

Statement	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	Susp
`int m;`	✓	✓	✓	✓	✓	✓	0.5
`m = z;`	✓	✓	✓	✓	✓	✓	0.5
`if(y < z) {`	✓	✓	✓	✓	✓	✓	0.5
`if(x < y)`	✓	✓			✓	✓	0.63
`m = y;`		✓					0
`else if(x < z)`	✓				✓	✓	0.71
`m = y;`	✓					✓	0.83
`} else {`			✓	✓			0
`if(x > y)`			✓	✓			0
`m = y;`			✓				0
`else if(x > z)`				✓			0
`m = x;`							0
`return m;`	✓	✓	✓	✓	✓	✓	0.5
	Pass	Pass	Pass	Pass	Pass	Fail	

$$\text{Suspicious}(s) = \frac{\text{fail}(s) / \text{totalfail}}{\text{fail}(s) / \text{totalfail} + \text{pass}(s) / \text{total pass}}$$

Fault Localisation

Statement	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	Susp
`m = y;`	✓					✓	0.83
`else if(x < z)`	✓				✓	✓	0.71
`if(x < y)`	✓	✓			✓	✓	0.63
`int m;`	✓	✓	✓	✓	✓	✓	0.5
`m = z;`	✓	✓	✓	✓	✓	✓	0.5
`if(y < z) {`	✓	✓	✓	✓	✓	✓	0.5
`return m;`	✓	✓	✓	✓	✓	✓	0.5
`m = y;`		✓					0
`} else {`			✓	✓			0
`if(x > y)`			✓	✓			0
`m = y;`			✓				0
`else if(x > z)`				✓			0
`m = x;`							0
	Pass	Pass	Pass	Pass	Pass	Fail	

Fault Localisation

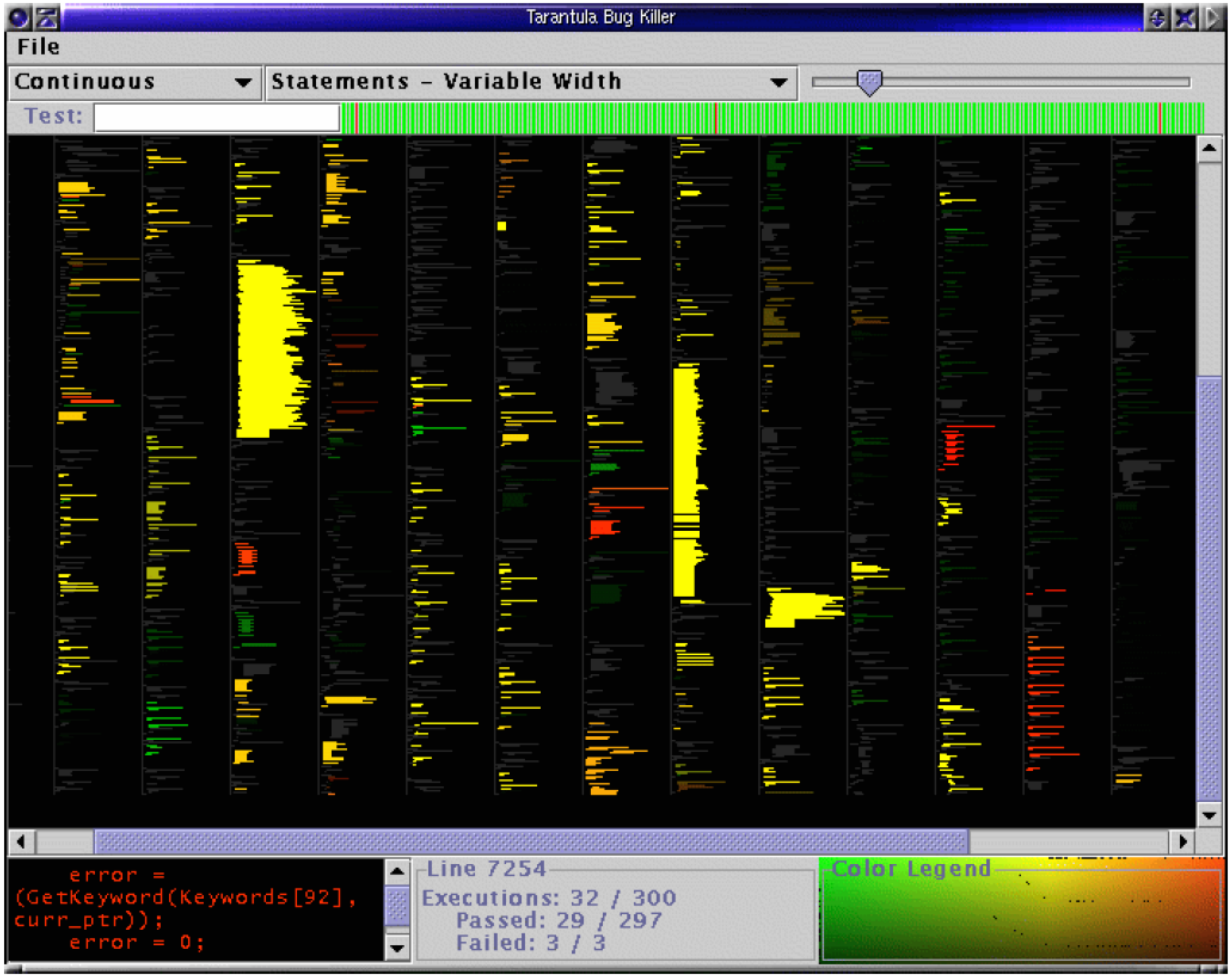
```
int mid(int x, int y, int z) {
  int m;
  m = z;
  if (y < z) {
    if (x < y)
      m = y;
    else if (x < z)
      m = y;
  } else {
    if (x > y)
      m = y;
    else if ( x > z)
      m = x;
  }
  return m;
}
```

Fault Localisation

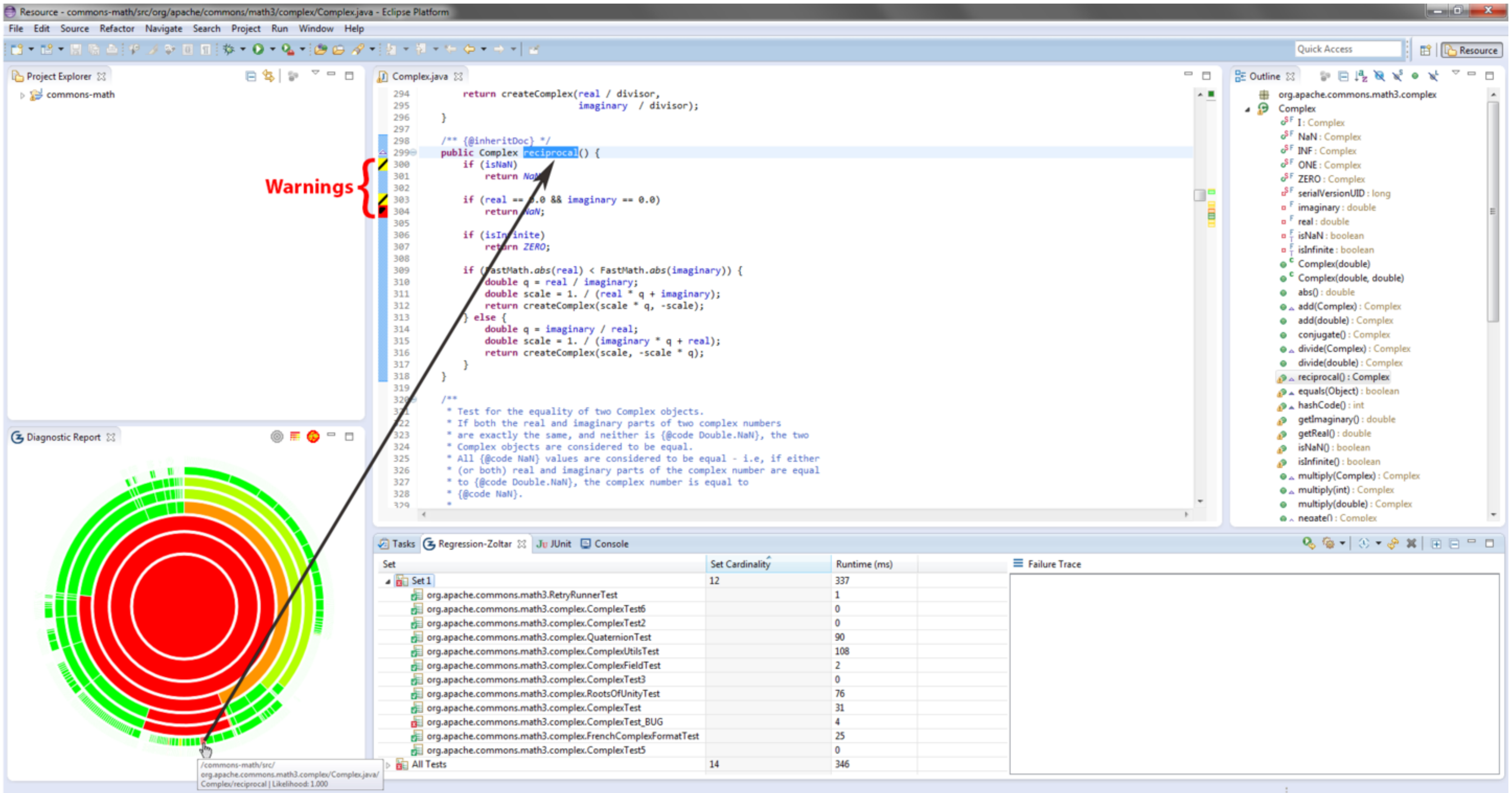
```
int mid(int x, int y, int z) {
  int m;
  m = z;
  if (y < z) {
    if (x < y)
      m = y;
    else if (x < z)
      m = y;                        // should be m = x;
  } else {
    if (x > y)
      m = y;
    else if ( x > z)
      m = x;
  }
  return m;
}
```

Hue = red + %pass/(%pass+%fail) * range

Tarantula


Tarantula Bug Killer
File
Continuous
Statements – Variable Width
Test:
error =
(GetKeyword(Keywords[92],
curr_ptr));
error = 0;
Line 7254
Executions: 32 / 300
Passed: 29 / 297
Failed: 3 / 3
Color Legend

GZoltar


Warnings
Resource - commons-math/src/org/apache/commons/math3/complex/Complex.java - Eclipse Platform
Project Explorer
Complex.java
Outline
Diagnostic Report
Tasks
Regression-Zoltar
JUnit
Console
Set
Set Cardinality
Runtime (ms)
Failure Trace
Set 1
All Tests


Diagnostic Report
1
2
3
4
5
6
7
8
workspace.project.packageroot.package.file.class.method.line

GP Implementation

- Functions: add, sub, mul, div, sqrt
- Terminals: ep, ef, np, nf, 1
- Fitness function:

$$E(\tau, p, b) = \frac{\text{Ranking of } b \text{ according to } \tau}{\text{Number of statements in } p} * 100$$

$$\text{fitness}(\tau, B, P) = \frac{1}{n}\sum_{i=1}^{n} E(\tau, p_i, b_i) \text{ (to be minimised)}$$