Genetic Programming

School of Computing, KAIST
Shin Yoo

Contains materials from “A Field Guide to Genetic Programming” by R. Poli, W. Langdon, and N. McPhee

“In genetic programming we evolve a population of *computer programs*. That is, generation by generation, GP *stochastically* transforms populations of programs into new, hopefully better, populations of programs…”

- A Field Guide to 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

The End.

Or is it?

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

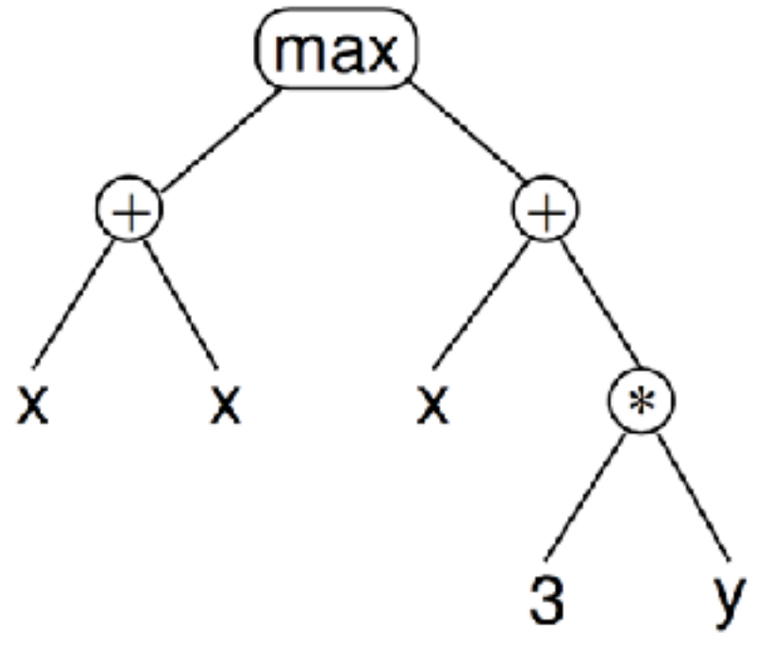
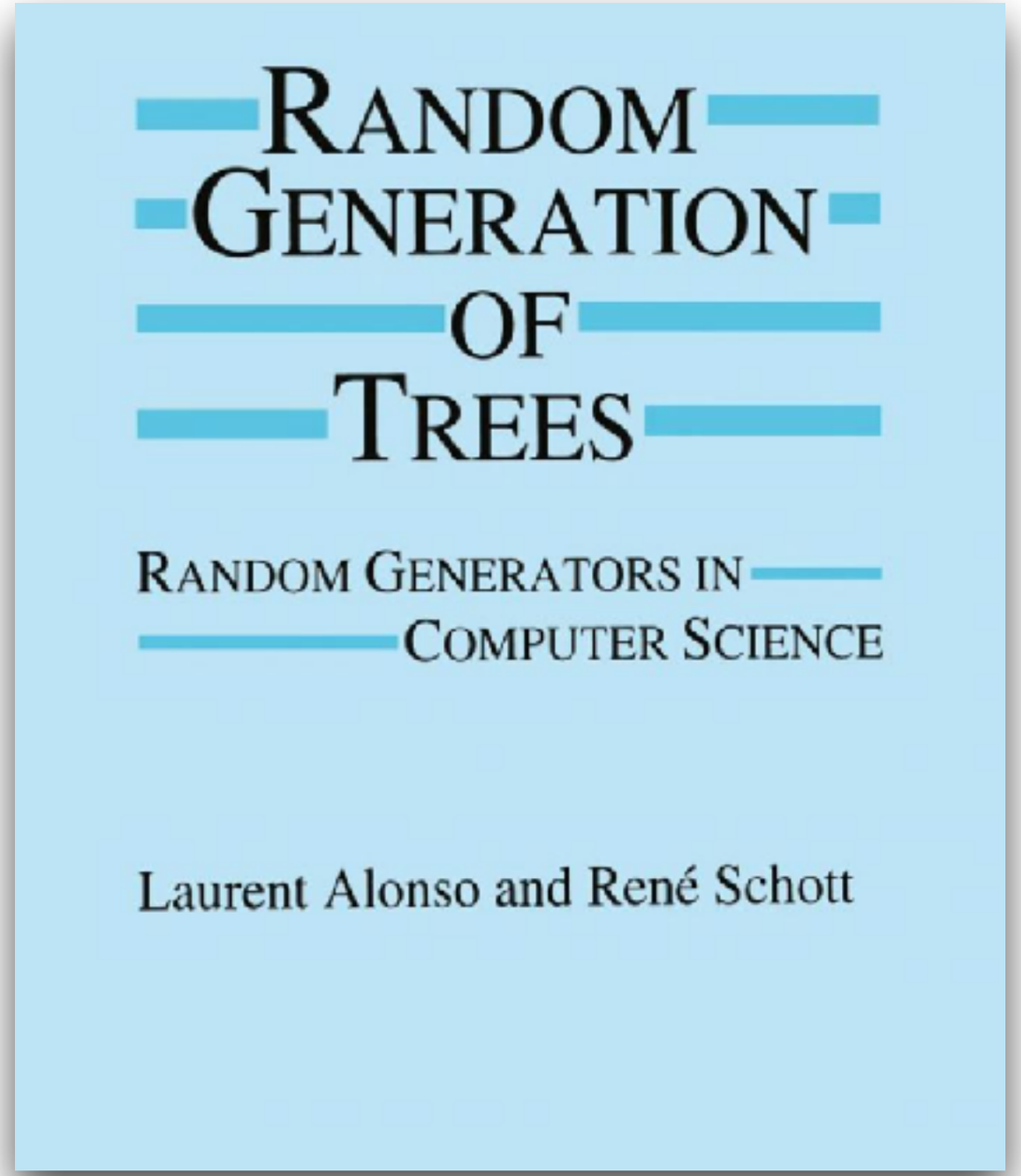


Figure 2.1: GP syntax tree representing `max(x+x,x+3*y)`.

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.



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.

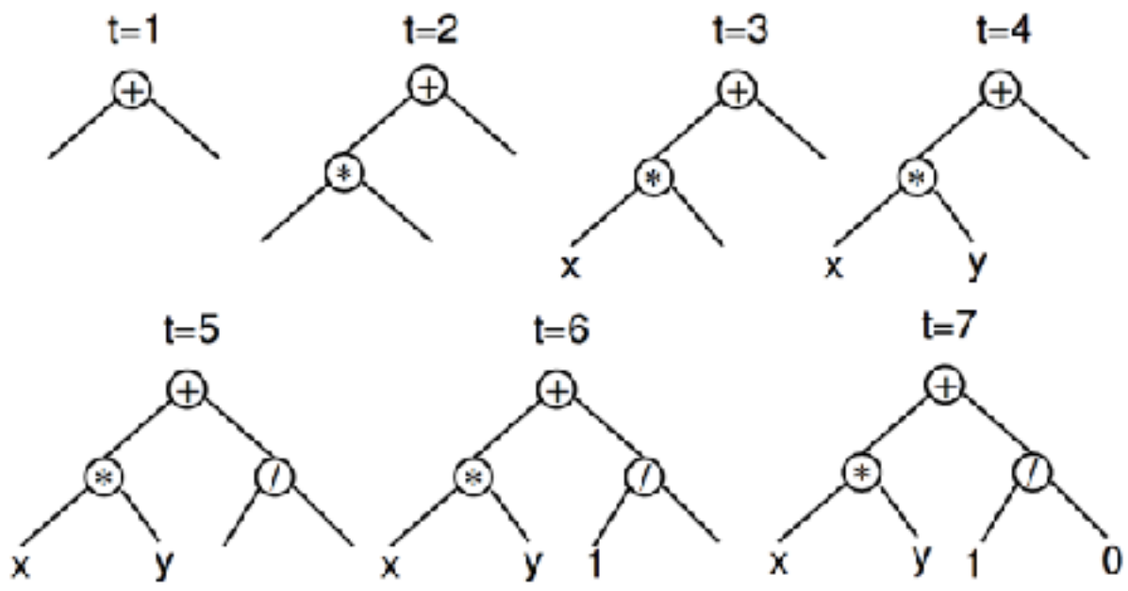


Figure 2.3: Creation of a full tree having maximum depth 2 using the `full` initialisation method ($t = \text{time}$).

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.

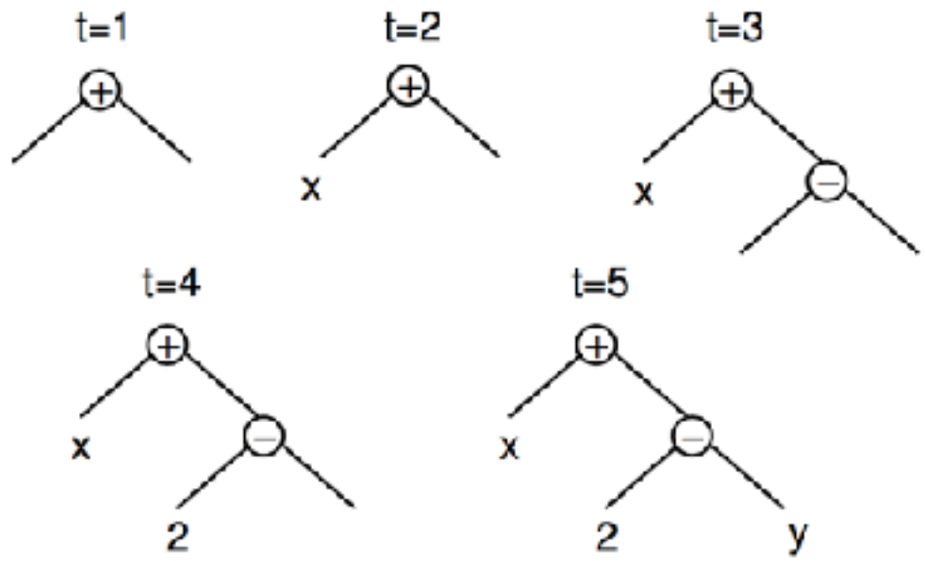


Figure 2.4: Creation of a five node tree using the **grow** initialisation method with a maximum depth of 2 (t = time). A terminal is chosen at $t = 2$, causing the left branch of the root to be closed at that point even though the maximum depth had not been reached.

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

Uniform Initialisation

- 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).

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.

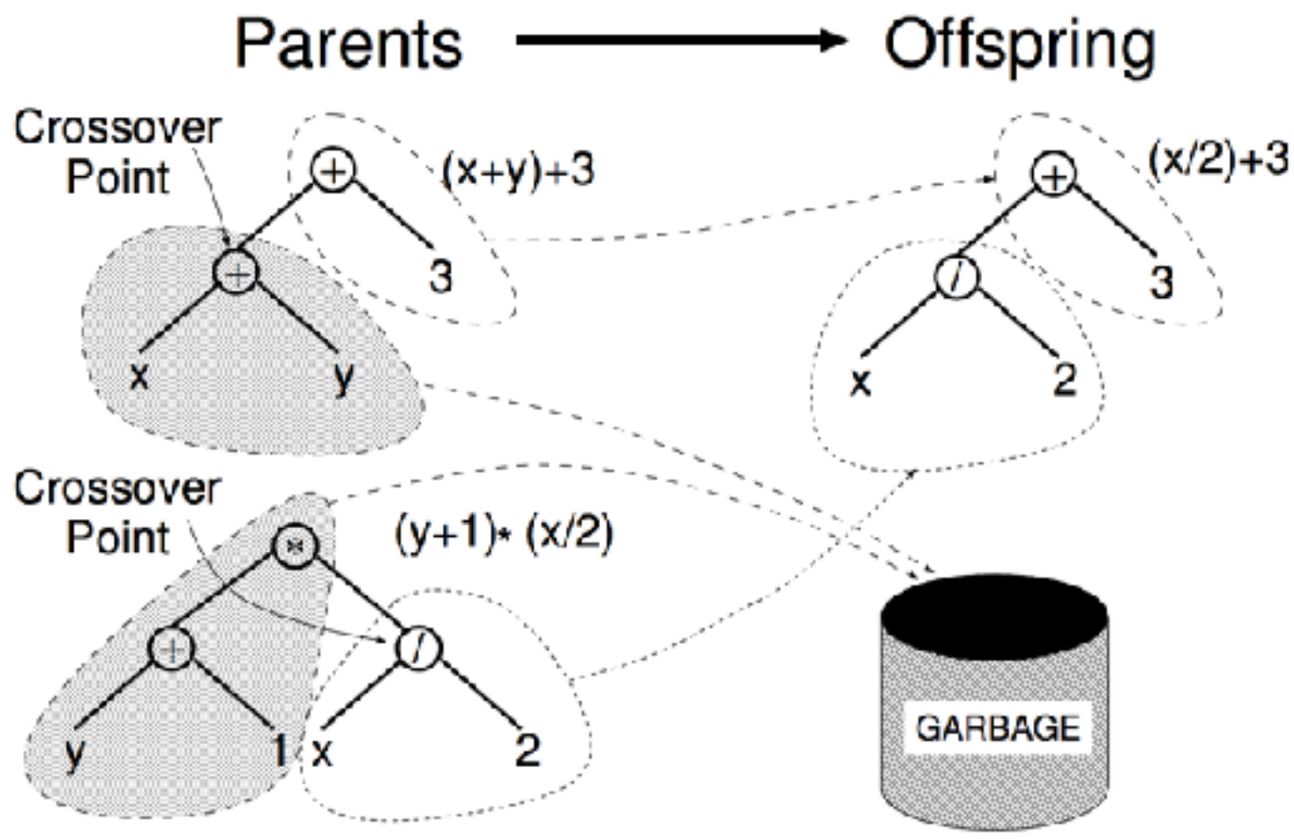


Figure 2.5: Example of subtree crossover. Note that the trees on the left are actually *copies* of the parents. So, their genetic material can freely be used without altering the original individuals.

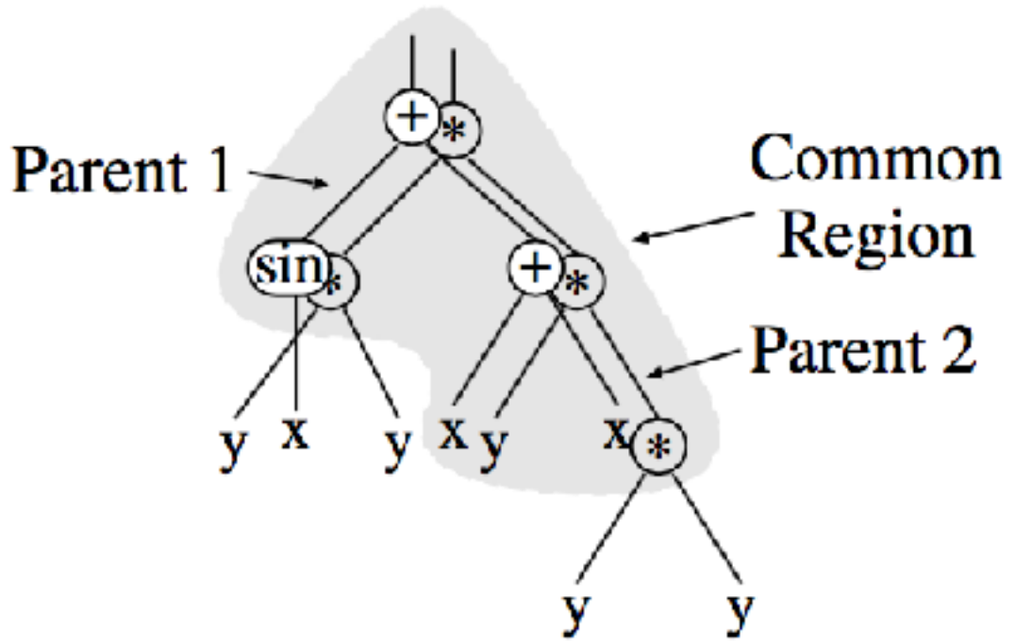
Crossover

- Often crossover points are NOT sampled with uniform 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

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.



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.

Subtree Mutation

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

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.

… 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.

Type Closure

- Non-terminals need to be type consistent: it is necessary that any subtree can be used in any argument position of any function in the set.

- Rewriting may do. For example, **`if(boolean, expr1, expr2)`** can be rewritten as **`if(expr1, expr2, expr3, expr4)`** where the predicate is **`expr1 < expr2`**.

Strongly-typed GP

- An alternative to basic type consistency: all node have types, and operators should follow the type rules.

- Has been extended to generics, polymorphism, and higher-order functions

Target Language Can Help

- The target language, with which GP is trying to evolve a program, may also help the type problem.
- You can even design your own target language! :)
  - For example, imagine a stack based language where each type uses separate stack.
  - GP for Ahui(아희), anyone?

Evaluation Safety

- Some combinations of nodes will fail at runtime: division by zero!
- Rewrite functions to be fail-proof:
  - `1 / x → 1 if x == 0 else 1 / x`
  - Overflows are trickier to deal with.

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.

Interpretation

- Again, fitness evaluation equals execution of the program (or evaluation of the expression)

- Typically GP systems will implement a small interpreter for the tree representations.

- Language features can help too.

Bloats

- 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

Other Forms of GP

- **Linear GP**: programs are, eventually, a sequence of instructions, so why not use linear list of instructions? X86 code has been evolved.

- **Parallel and Distributed GP**: uses graphs, not trees, to reuse partial evaluations. Execution is bottom-up propagation of input value, rather than top-down evaluation (visit) of trees.

Parallel and Distributed GP

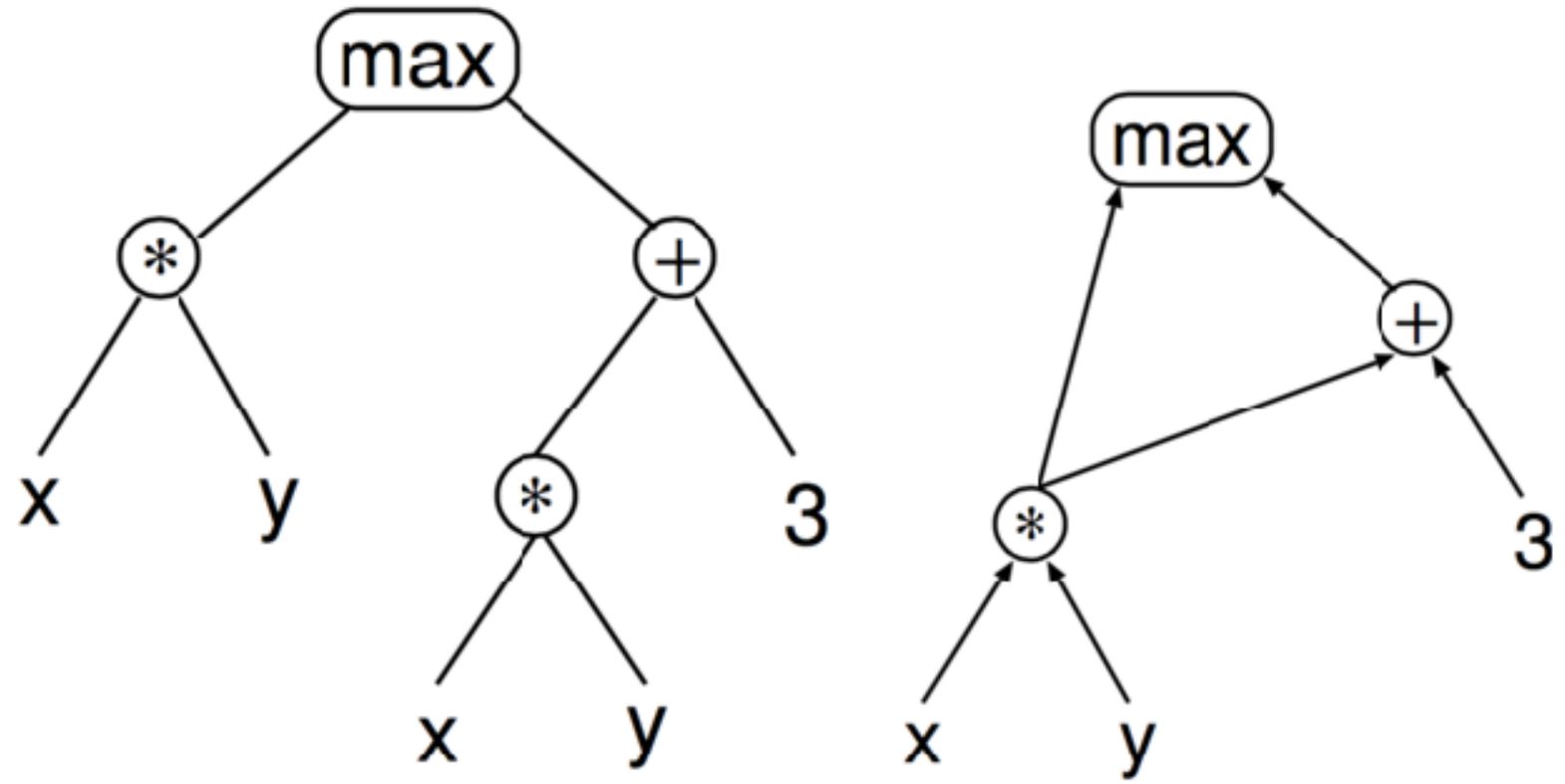
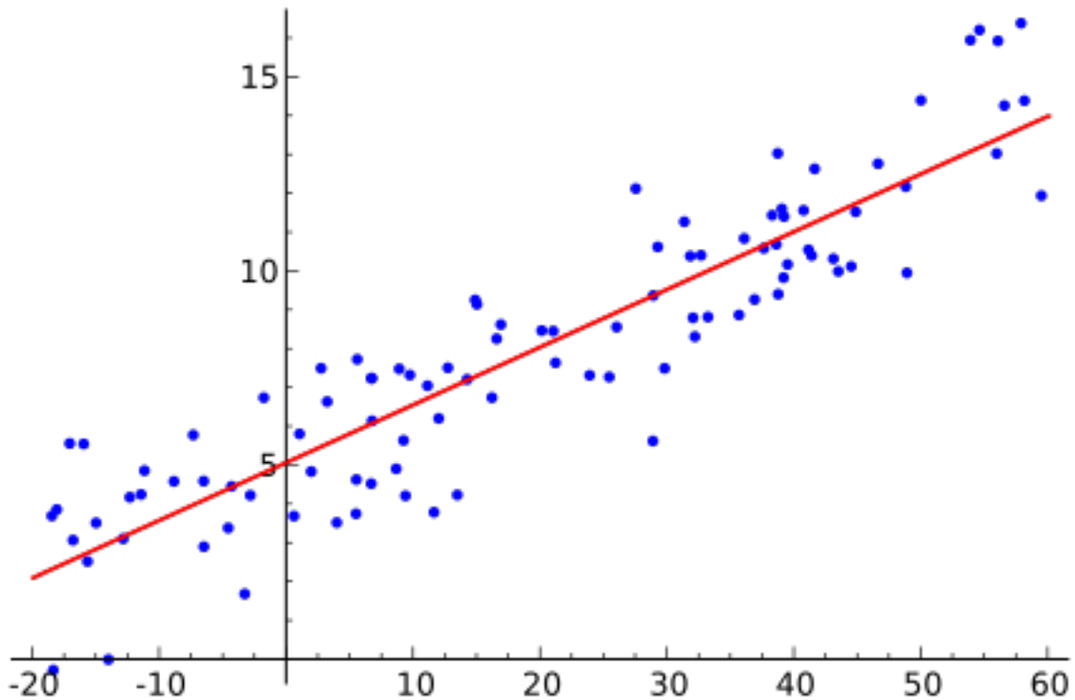


Figure 7.5: A sample tree where the same subtree is used twice (a) and the corresponding graph-based representation of the same program (b). The graph representation may be more efficient since it makes it possible to avoid the repeated evaluation of the same subtree.

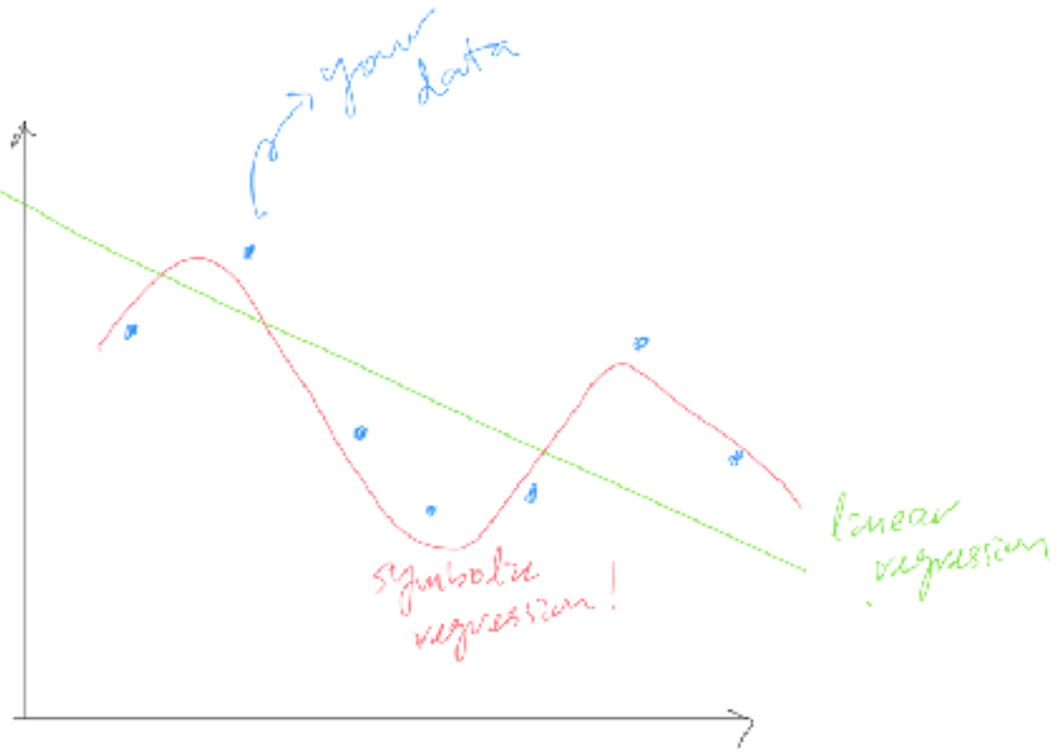
Symbolic Regression

- 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



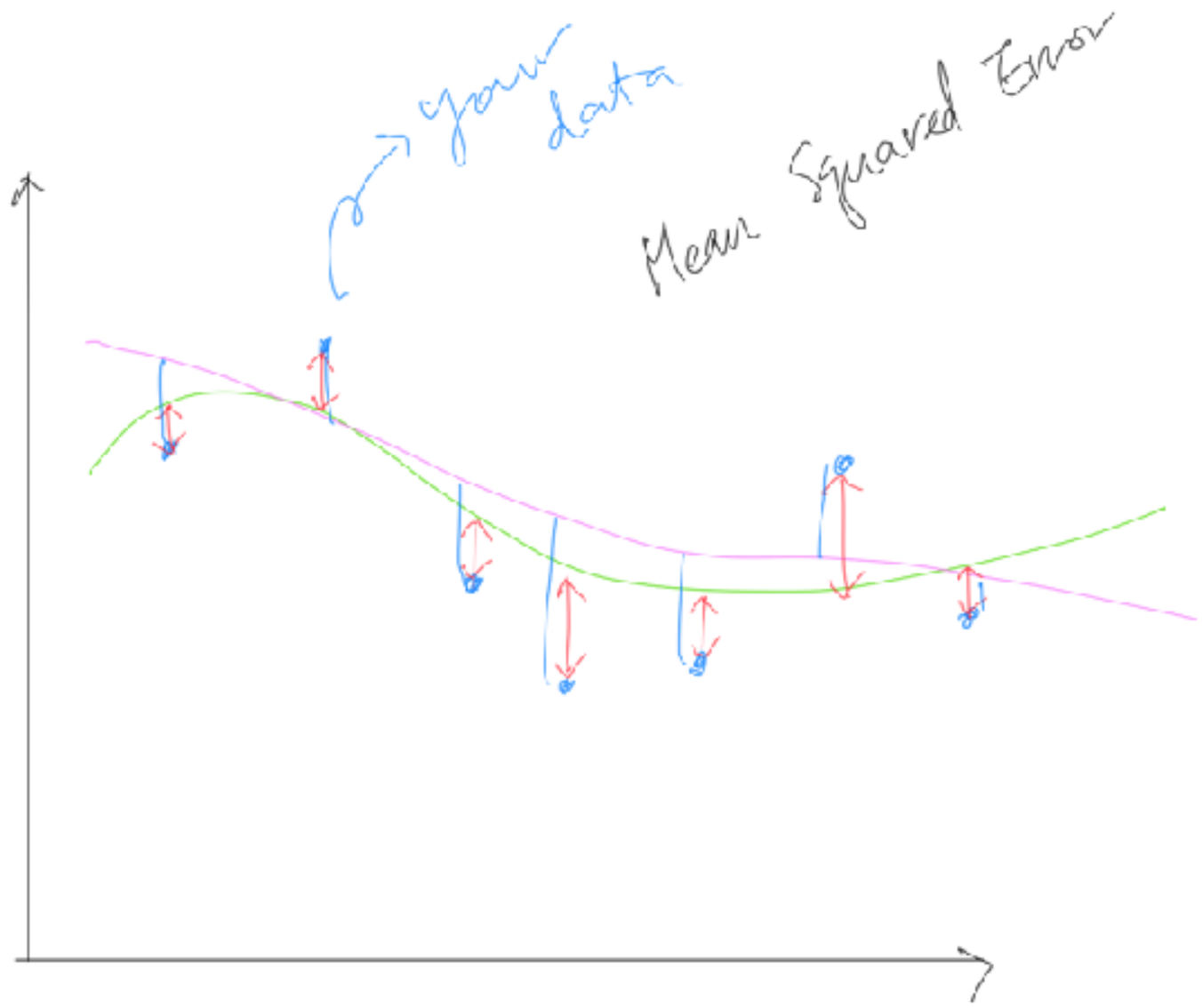
Symbolic Regression

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



Symbolic Regression: 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.



Symbolic Regression: Constants

- When it is relatively obvious which set of constants will be “helpful”, you can provide them.

- You can provide building blocks of them (1, 10, 100, etc)

- You can use Ephemeral Random Constant (ERC): a constant whose value is randomly determined when it is first created.

Examples of Symbolic Regression

Name	Variables	Equation	Training Set Testing Set
Keijzer-6 [25] [46]	1	$\sum_i^x \frac{1}{i}$	E[1, 50, 1] E[1, 120, 1]
Korns-12 [27]	5	$2 - 2.1\cos(9.8\,x)\,\sin(1.3\,w)$	U[-50, 50, 10000] U[-50, 50, 10000]
Vladislavleva-4 [50]	5	$\frac{10}{5 + \sum_{i=1}^{5}(x_i - 3)^2}$	U[0.05, 6.05, 1024] U[-0.25, 6.35, 5000]
Nguyen-7 [33]	1	$\ln(x+1) + \ln(x^2+1)$	U[0, 2, 20] None
Pagie-1 [36]	2	$\frac{1}{1+x^{-4}} + \frac{1}{1+y^{-4}}$	E[-5, 5, 0.4] None
Dow Chemical *(see Section 6.1)*	57	chemical process data[6]	747 points 319 points
GP Challenge [56] *(see Section 6.1)*	8	protein energy data	1250–2000 per protein None

Table 5 Proposed symbolic regression benchmarks. In the training and testing sets, U[a,b,c] is c uniform random samples drawn from a to b, inclusive. E[a,b,c] is a grid of points evenly spaced with an interval of c, from a to b inclusive.

Better GP Benchmarks: Community Survey Results and Proposals: David R. White, James McDermott, Mauro Castelli, Luca Manzoni, Brian W. Goldman, Gabriel Kronberger, Wojciech Jaskowski, Una-May O'Reilly, and Sean Luke. Genetic Programming and Evolvable Machines 14:1 (3-29), 2013.

GP as Modification not Evolution

- Borrow GP to slightly modify a large, existing, code base, rather than to evolve something from the scratch.

- Grammar based modification (Langdon and Harman 2015):

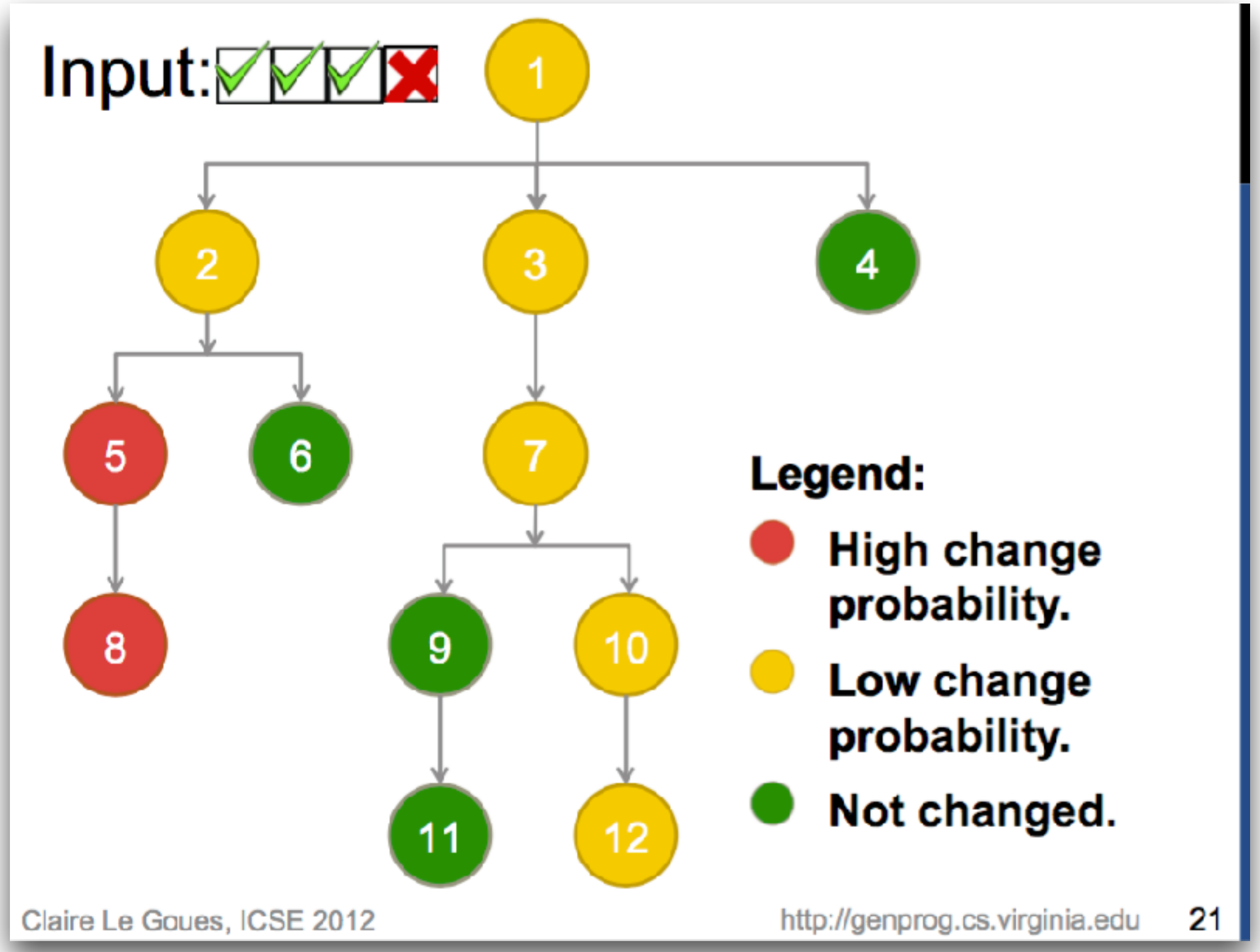
  - First, represent the existing code as a typed BNF grammar rule.

  - Second, modify the grammar, and expand the modified grammar.

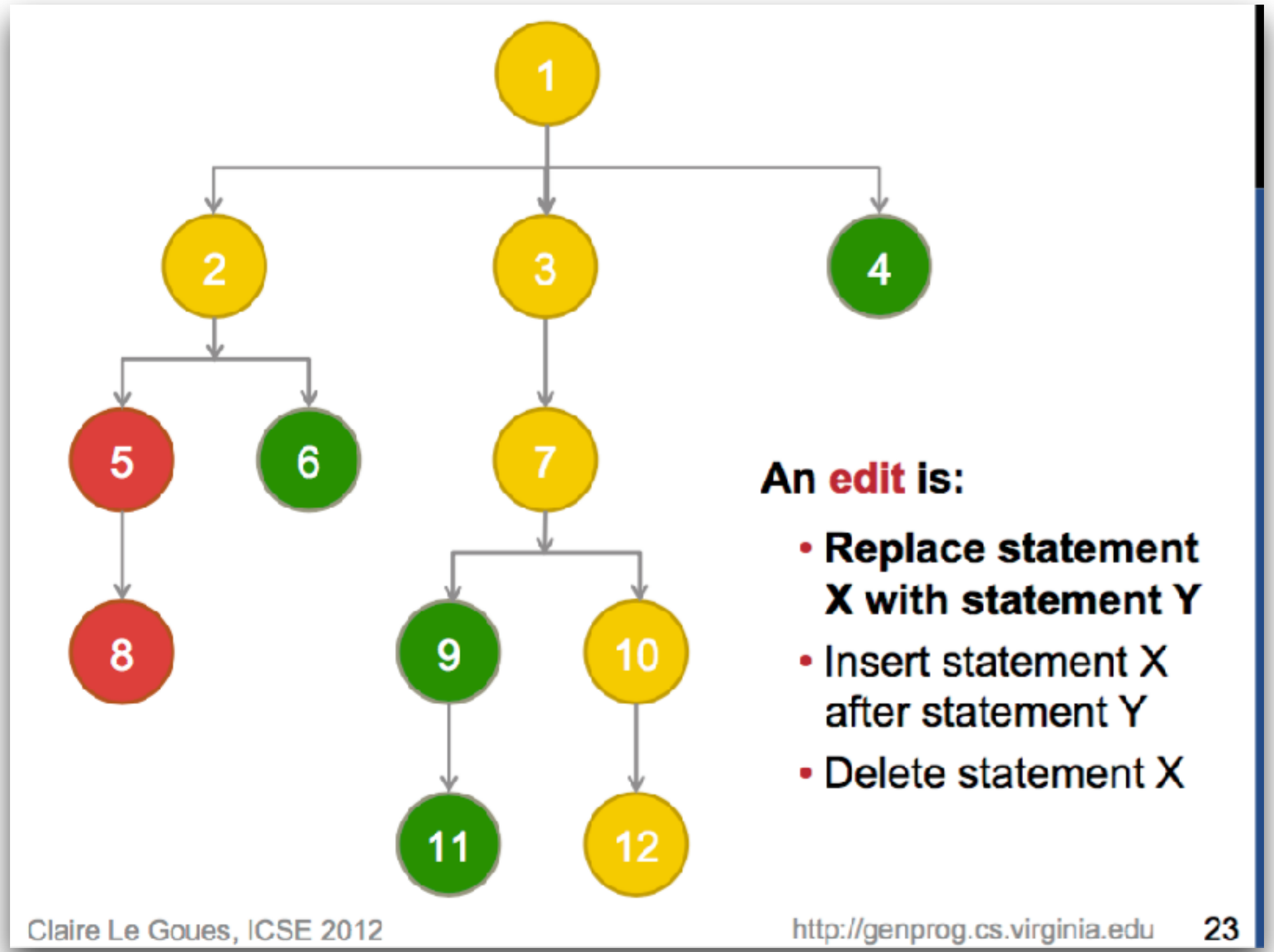
Intuition for Automated Program Repair (APR)

- Not all, but many patches are usually already in the same codebase, just somewhere else (think missing null-check, for example).

- Remember: code is not so unique, and it is natural.



C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In Proceedings of the 34th International Conference on Software Engineering, pages 3–13, 2012.



C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In Proceedings of the 34th International Conference on Software Engineering, pages 3–13, 2012.

105 real faults, 13 hours of Genetic Programming on Amazon EC2

SUCCESS/COST

Program	Defects Repaired	Cost per non-repair		Cost per repair	
		Hours	US$	Hours	US$
fbc	1/3	8.52	5.56	6.52	4.08
gmp	1/2	9.93	6.61	1.60	0.44
gzip	1/5	5.11	3.04	1.41	0.30
libtiff	17/24	7.81	5.04	1.05	0.04
lighttpd	5/9	10.79	7.25	1.34	0.25
php	28/44	13.00	8.80	1.84	0.62
python	1/11	13.00	8.80	1.22	0.16
wireshark	1/7	13.00	8.80	1.23	0.17
Total	**55/105**	**11.22h**		**1.60h**	

$403 for all 105 trials, leading to 55 repairs; $7.32 per bug repaired.

C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In Proceedings of the 34th International Conference on Software Engineering, pages 3–13, 2012.

Machines are dumb

- **`nullhttpd`**: POST test failed, so it removed the entire functionality.

- **`sort`**: output had to be sorted, so the fix was to always print an empty set (which is, by definition, sorted).

- Test oracle was to compare **`output.txt`** to pre-defined **`correct_output.txt`**: upon failure, it deleted **`correct_output.txt`** and printed nothing.

How about improvements that are not repairs?

BNF Grammar

```
vmax = vlo;
```

Line 365 of aligner_swsse_ee_u8.cpp

```
<aligner_swsse_ee_u8_365>  ::="" <_aligner_swsse_ee_u8_365>
.                                  "{Log_count64++;/*28577*/}\n"
<_aligner_swsse_ee_u8_365> ::="vmax = vlo;"
```

Fragment of Grammar (Total 28765 rules)

(Slides borrowed from Dr. Landgon's keynote at SSBSE 2015, "Genetic Improvement of Software for Multiple Objectives")

Example Mutating Grammar

```
<_aligner_swsse_ee_u8_707> ::=    "vh = _mm_max_epu8(vh, vf);"
<_aligner_swsse_ee_u8_365> ::=    "vmax = vlo;"
```

2 lines from grammar

```
<_aligner_swsse_ee_u8_707><_aligner_swsse_ee_u8_365>
```

Fragment of list of mutations

Says replace line 707 of file aligner_swsse_ee_u8.cpp by line 365

```
vh = _mm_max_epu8(vh, vf);{Log_count64++;/*28919*/}
```

Instrumented original code

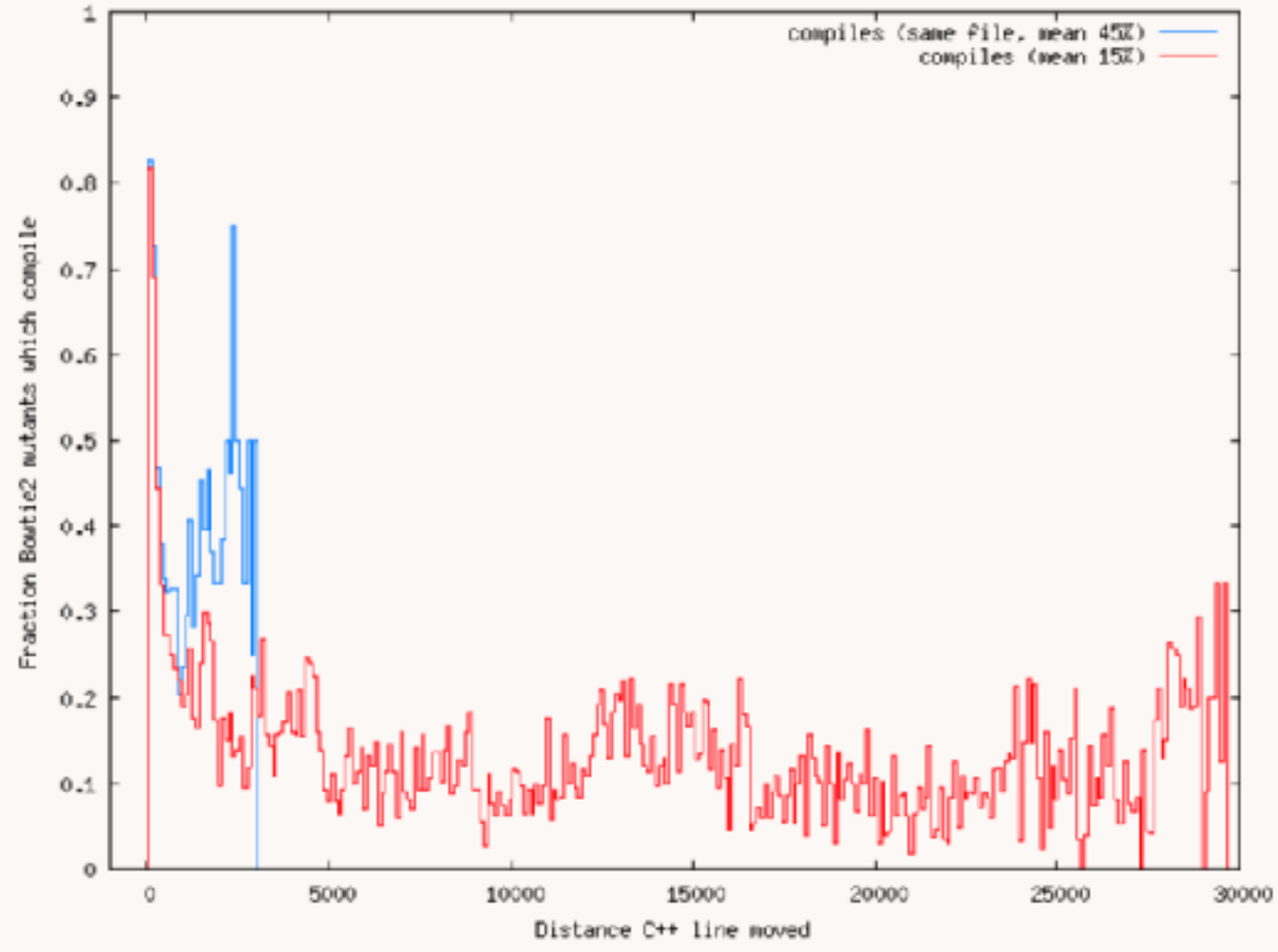
```
vmax = vlo;{Log_count64++;/*28919*/}
```

New code

(Slides borrowed from Dr. Landgon's keynote at SSBSE 2015, "Genetic Improvement of Software for Multiple Objectives")

Compilation Errors

- Use grammar to replace random line, only 15% compile. But if move <100 lines 82% compile.
- Restrict moves to same file, 45% compile



(Slides borrowed from Dr. Landgon's keynote at SSBSE 2015, "Genetic Improvement of Software for Multiple Objectives")

Representation

- GP evolves patches. Patches are lists of changes to the grammar.
- Append crossover adds one list to another
- Mutation adds one randomly chosen change
- 3 possible changes:
  - Delete line of source code (or replace by “”, 0)
  - Replace with line of Bowtie2 (same type)
  - Insert a copy of another Bowtie2 line

Run time errors

- During evolution 74% compile
- 6% fail at run time
  - 3% segfault
  - 2% cpulimit expired
  - 0.6% heap corruption, floating point (e.g. divide by zero) or Bowtie2 internal checks
- 68% run ok

Patch

Weight	Mutation	Source file	line	type	Original Code	New Code
999	replaced	bt2_io.cpp	622	for2	i < offsLenSampled	i < this->_nPat
1000	replaced	sa_rescomb.cpp	50	for2	i < satup_->offs.size()	0
1000	disabled		69	for2	j < satup_->offs.size()	
100	replaced		707		vh = _mm_max_epu8(vh, vf);	vmax = vlo;
1000	deleted		766		pvFStore += 4;	
1000	replaced	aligner_swsse_ee_u8.cpp	772		_mm_store_si128(pvHStore, vh);	vh = _mm_max_epu8(vh, vf);
1000	deleted		778		ve = _mm_max_epu8(ve, vh);	

- Evolved patch 39 changes in 6 .cpp files
- Cleaned up 7 changes in 3 .cpp files
- 70+ times faster

GP as a Complete Autonomous programmer

- Evolving a complete application is hard.
- PushCalc is a complicated system that uses Clojure (a LISP-variant on JVM) and Push GP system (stack-based GP): its aim is to evolve a complete calculator application.

7. CONCLUSION

We have described four experiments that were used to test the hypothesis that complete software applications can evolved by a genetic programming system that has the ability to automatically evolve and name its own modular functions.

No individuals in any of the populations were able to evolve a complete calculator within 1000 generations. Individuals that had access to the tagging mechanism present in PushCalc had smaller average initial error rates than individuals without tags and these individuals also succeeded the most at evolving the calculator by having the smallest errors. At the end of 1000 generations, individuals with access to the tagging mechanism had error rates that were over 400% smaller than individuals that had no access to the tagging mechanism. This supports the hypothesis that tags would be very useful in solving the problem.

K. Yeboah-Antwi. Evolving software applications using genetic programming – pushcalc: The evolved calculator. In Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation, GECCO '12, pages 569–572, New York, NY, USA, 2012. ACM.

Summary

- GP at its bare basic is simply GA with structured representations.

- Evolving complete applications from the scratch is still very hard; however, applying GP to existing code has produced some very interesting results.