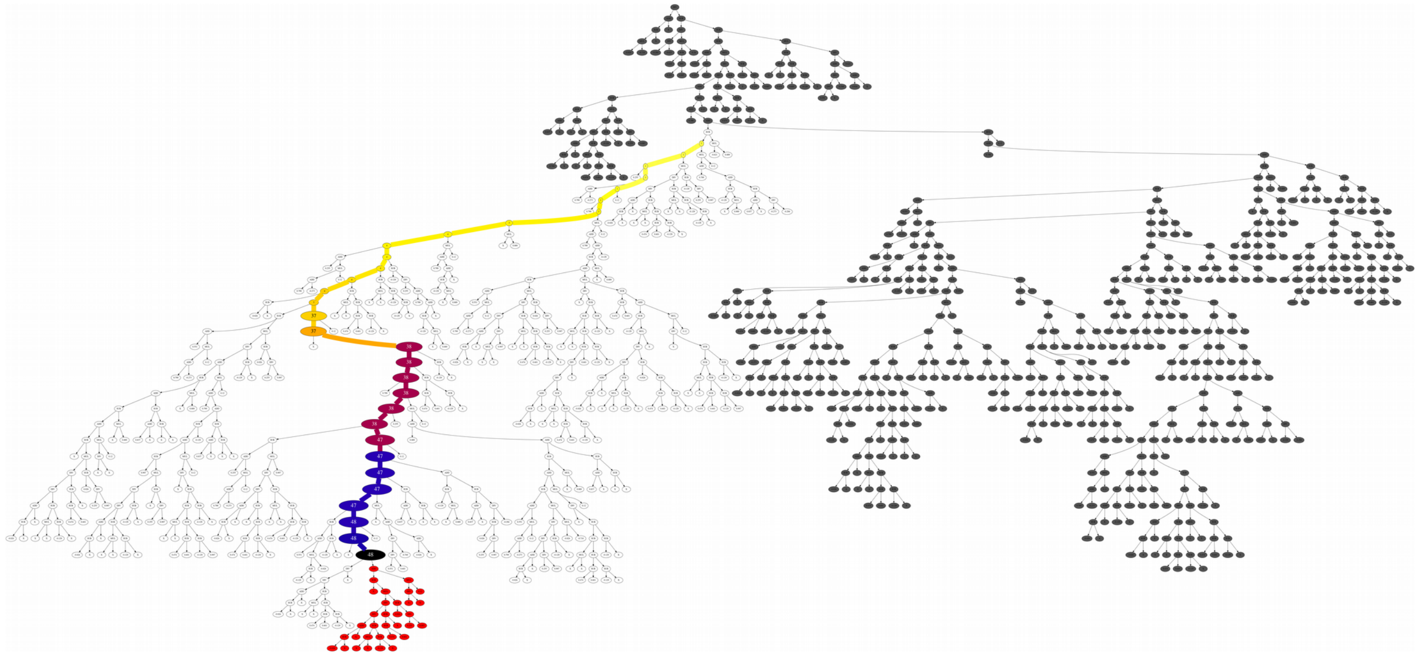
A Trillion Genetic Programming Instructions per Second

http://arxiv.org/abs/2205.0325

Goal build a genetic programming system which can run unconstrained for 100,000 generations, even a million generations, in a reasonable time on available hardware.

GP trees bloat, these are huge.

W. B. Langdon, UCL

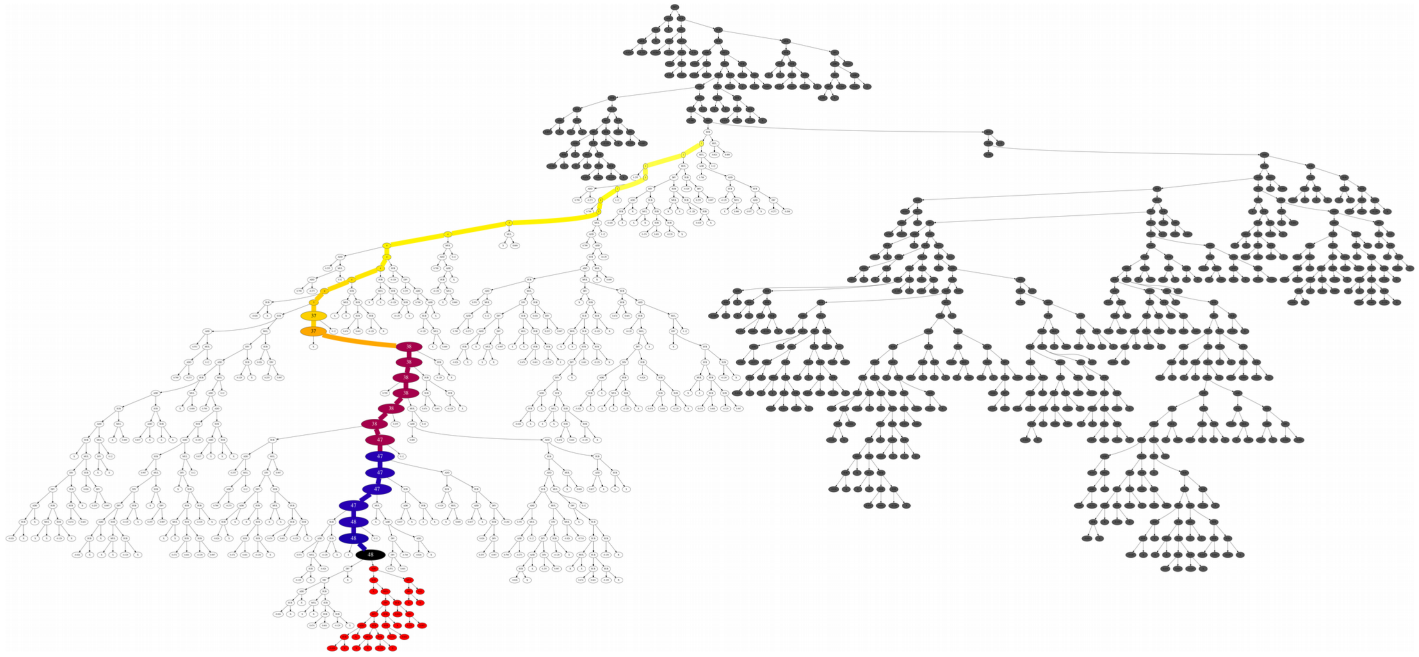


4.7.2022

Cheats and how you can used them

http://arxiv.org/abs/2205.0325

W. B. Langdon, UCL



To come Clean

- *Up to* 1.1 10^{12} GP operation per second is for fastest run.
    - <4 days, 73,000 generations, max GP tree 2,000,000,000
    - Turning off statistics saves 70%.
- Average across whole run.
- Used 16 cores via Posix threads (C++ code allows up to 96)
- AVX-512 Intel vector instructions doing 16 test case in parallel
- *Equivalent to* 10^{12} because optimised code avoids doing work.

- Slides for possible discussion areas

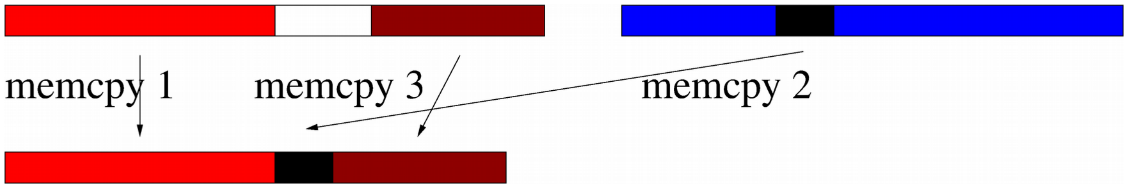
1. Converting Singleton's GPquick for SIMD interpreter
2. Importance of no side effects
3. Multi threaded crossover and fitness evaluation
4. Incremental fitness evaluation in deep trees
5. Evaluating fitness before crossover
6. Fatherless crossover

Single Instruction Multiple Data (SIMD) GPquick using AVX-512

- Since about 2005 CPU clock speeds not doubling, but Moore's law still giving extra transistors. These are used to give cheap parallel computing
- Intel AVX introduced in Xeon Phi accelerator cards (much like GPU [1])
- SSE 128, 256 or AVX 512 now included in many servers and desktops. (ARM $\leq$ 2048 bits)
- SIMD vector instructions do one operation on whole vector
  - Eg 512 bits = 16 floats
  - multiply 16 pairs of numbers
  - giving 16 answers, in one clock tick
- Given other non-parallel code, can in practice get speed up overall of > 3 (rather than 16)

GPquick

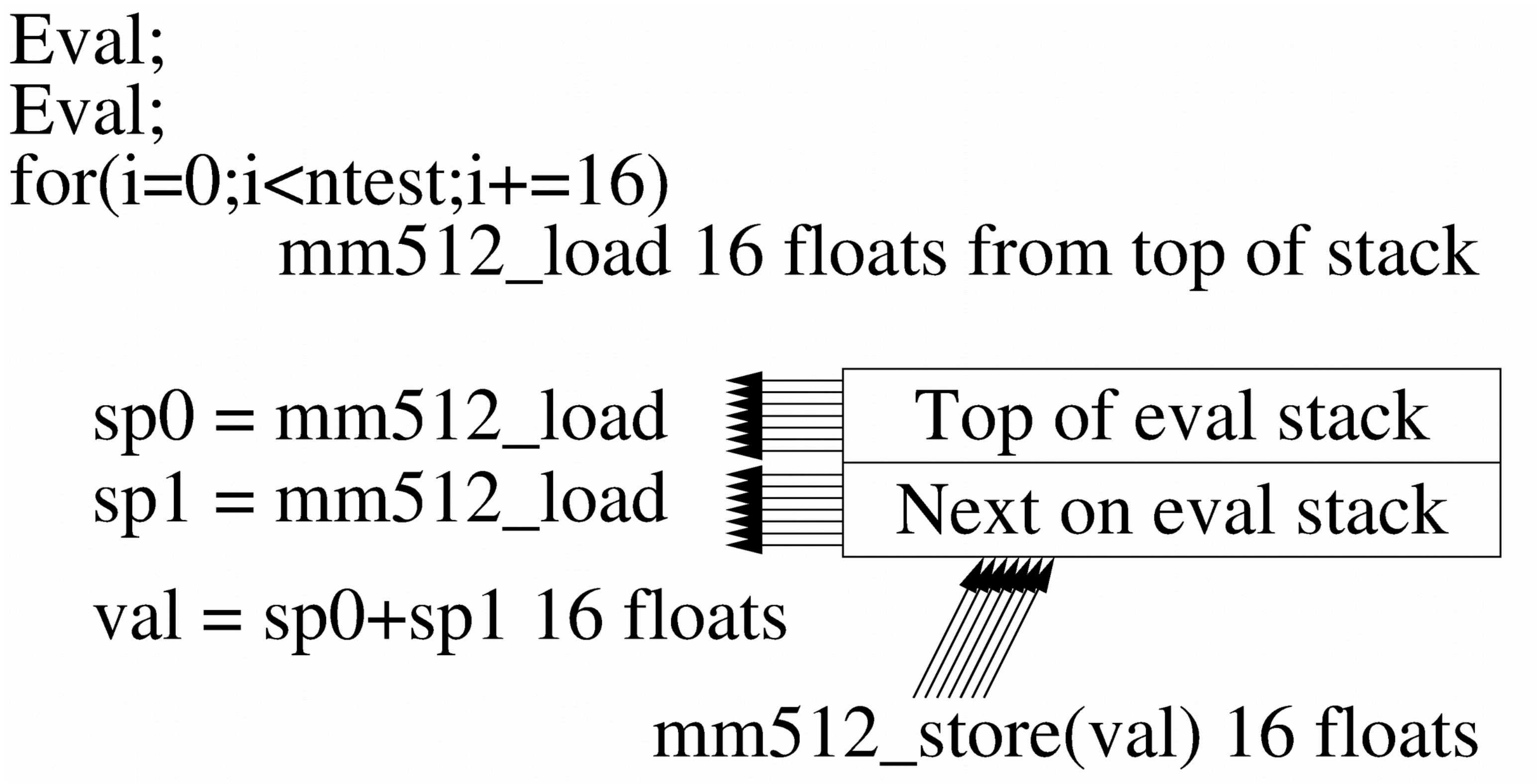
- C++ code written by Andy Singleton [1]
- Implements Keith & Martin’s Advances in GP (1994) [2]
- Using known branching factor of each internal node, GP trees are flatten into byte array of opcodes
    - 8 bit opcode limits size of function set + terminal set ≤ 255
- Compact linear representation one byte per tree node (allows trees of 2 billion nodes)
- Top down recursive interpreter:
    - traverse tree in byte order once per test case
- Originally steady state, now separate generations are used [3].



GPquick using AVX-512

- Instead of processing tree once per test case, process each opcode once doing all test cases in parallel.
- Tree processed only once (gives savings if tree too big for cache).
- For simplicity reduce number of test cases from 50 to 48 (ie 3 times 16).
- Top down recursive evaluation uses implicit stack (32 bits) replaced by (cache aligned) explicit stack 48 floats wide.
- Explicit stack (one per thread) allocated at start up.
  - Stack depth sets maximum tree depth
    - either user defined pMaxDepth or 100+10(pMaxExpr)$^{1/2}$
      - average height of binary tree = $2\sqrt{2\pi n} + O(n^{1/4})$
- Rewrote interpreter per opcode functions to use AVX and external stack

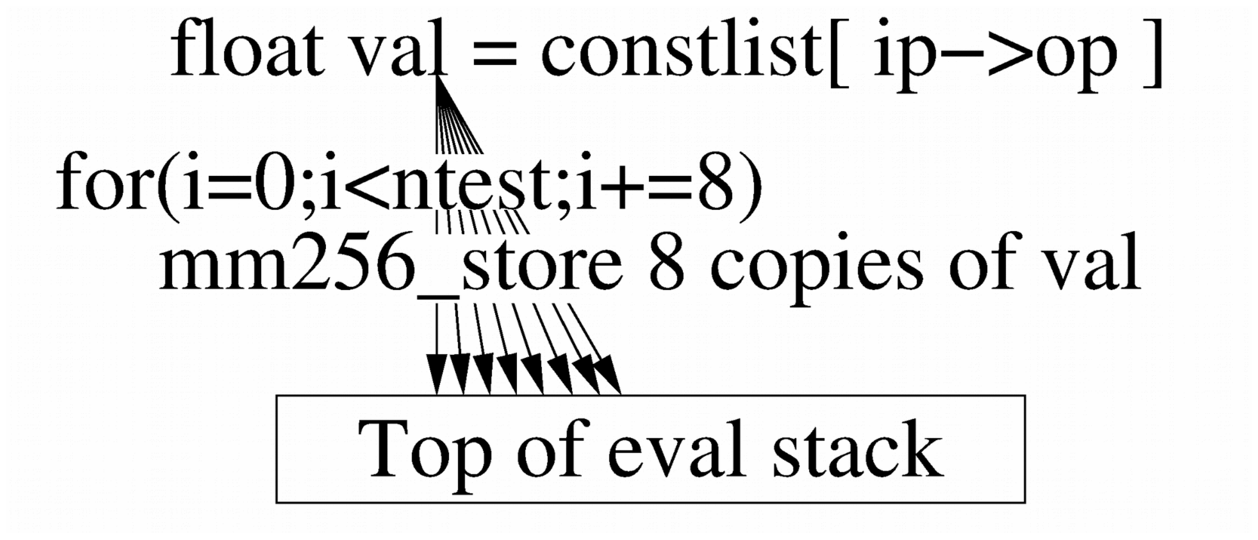
AVX-512 AddEval   _mm512_add_ps()



- Eval recursive call for each of function's arguments
- For loop i+=16 to process whole test set, leave answer on stack

CREST

SSE-256 ConstEval _mm256_store_ps()



- ≤ 250 constants available.
- Copy from constlist [index by opcode] to stack

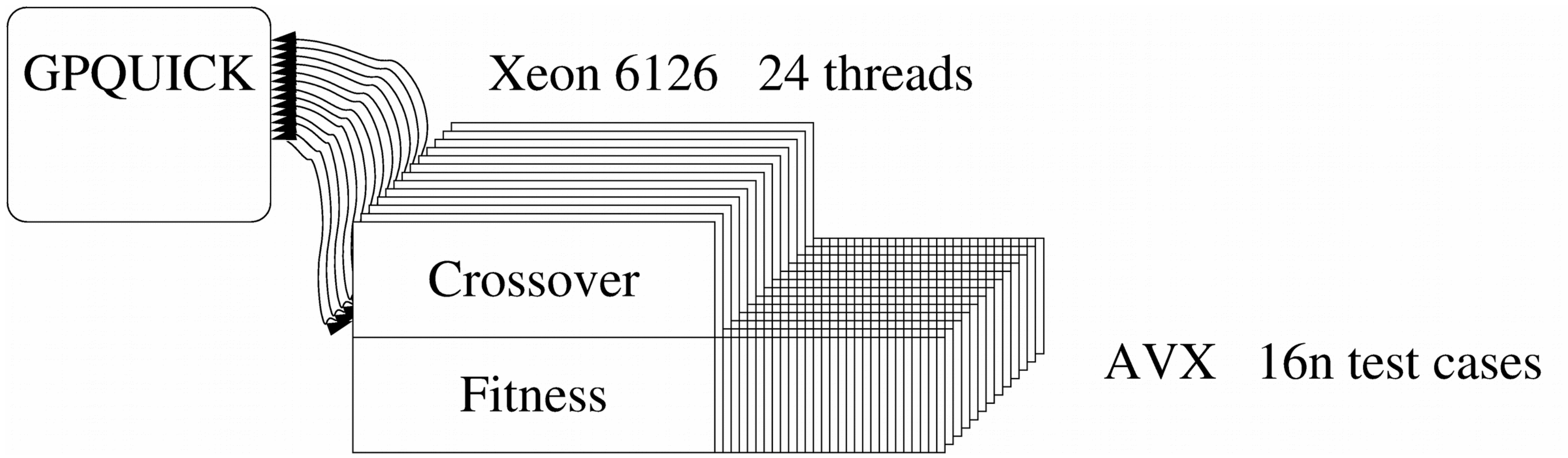
AVX-512 XEval _mm512_store_ps()

```
OPDEF(XEval){
#ifdef AVX
  for(int i=0;i<MAXTESTCASES;i+=16) {
    _mm512_store_ps(&EvalSP[i],_mm512_load_ps(&dataX[i]));
  }
#else
  memcpy(EvalSP,dataX,MAXTESTCASES*sizeof(retval));
#endif /*AVX*/
  inc_EvalSP; //++EvalSP;
}
```

- OPDEF GPquick macro used to define parts of its interpreter
- Copy all X values for all test cases on to stack
- _mm512_store_ps or memcpy
- Increment explicit stack pointer and return

Multi-core GPquick using POSIX threads

- 24 cores, each with AVX (24 x 16 = 384 flops per tick)
- Parallel both interpreter and crossover
- Ensure repeatability (and hence test ability)
  - All pseudo random number operations done in sequential code
  - Tournament selections made and crossover points chosen and stored before starting next generation

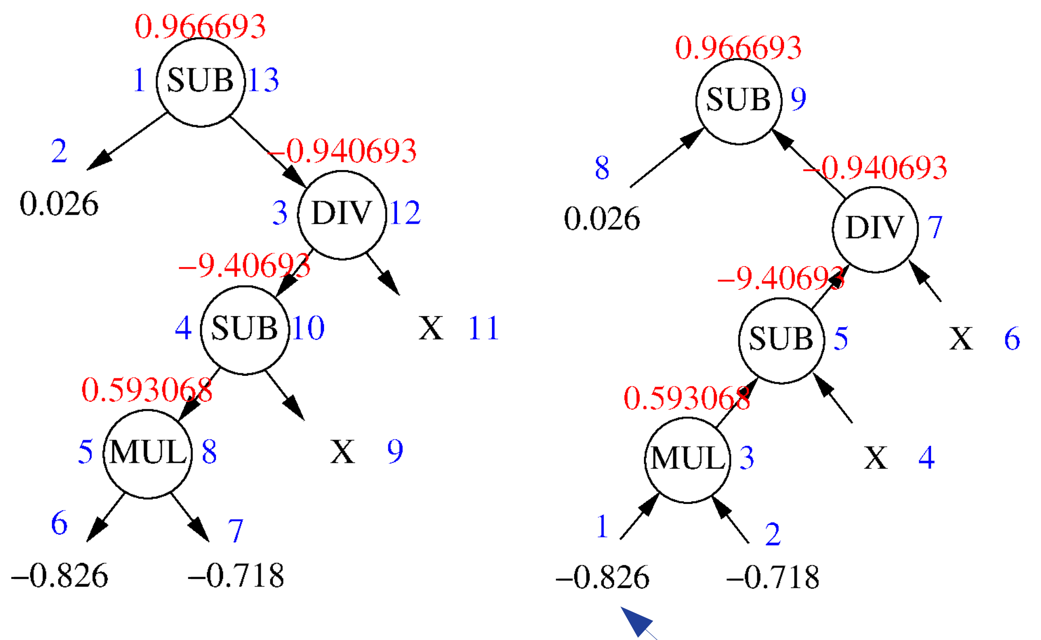


Absence of Side Effects

- Most tree genetic programming evolves expressions without side effects.
  - Exceptions include:
    - genetic programming with memory [1, 2]
    - evolving agents, eg
      - Sante Fe Ant [3] and robot soccer [4]

- Without side effects expression can be evaluated in any order
  - View expression as tree and evaluating the expression as processing each node in the tree. As long as we deal with the whole tree and ensure we pass intermediate values correctly, we are free to process the nodes in any order.
- Use flatten representation to navigate up tree as well as down

No side effects so Top Down = Bottom Up



Left: Conventional top-down recursive evaluation of (SUB 0.026 (DIV(SUB (MUL -0.826 -0.718) X) X)). X=10.

Blue integers indicate evaluation order, red floats are node return values.

Right: an alternative ordering, starting with leaf -0.826 and working to root node.

Both return exactly the same answer.

Trees are huge, Still not fast enough: Avoid work

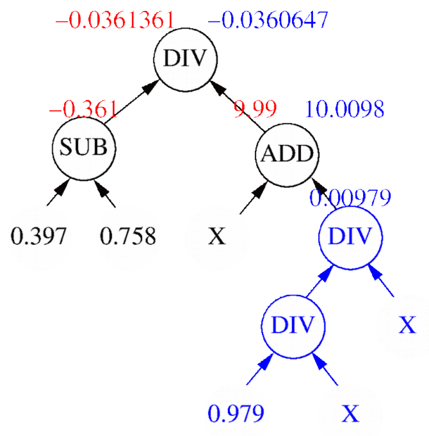
- Earlier work, eg
  - Simon Handley stored whole evolved population as single graph (DAG) [1].
  - Possible to cache subtree evaluations
  - Needs huge memory, forever chasing pointers?
  - Did not exploit population convergence
- 99% are the same, why do we recalculate the same fitness value over and over again?
  - Regression tests: compare with original run
  - Note where failed disruption propagation occurs [2], ie where disrupted and original evaluations becomes identical.
- Incremental Fitness Evaluation, EuroGP 2021 [3]

Genetic difference ≠> phenotypic difference

- Mum and child are identical except for inserted & removed subtrees.
- If by chance inserted & removed subtrees are identical:
  - mum and child are identical and so have the same fitness
- If inserted subtree evaluates to same value as removed subtree on every test case:
  - mum and child (at root node) evaluate to same value on every test case
  - genetic difference but mum and child have identical fitness
- What if the inserted subtree evaluates to different values to that given by remove subtree?
  - If we evaluate both child and mum starting at the change, there is a progressive fall in the number of test cases where the change is visible as we move towards the root node.

Evaluate both trees from change up

- Mum and child are identical above change.
- Fitness evaluation is identical except on route from change to root node.
- Evaluate both mum and child up this path.
- If they evaluate to identical values at any point then they evaluate to same value on the rest of the tree, including the root node:
  - semantic difference => identical fitness.

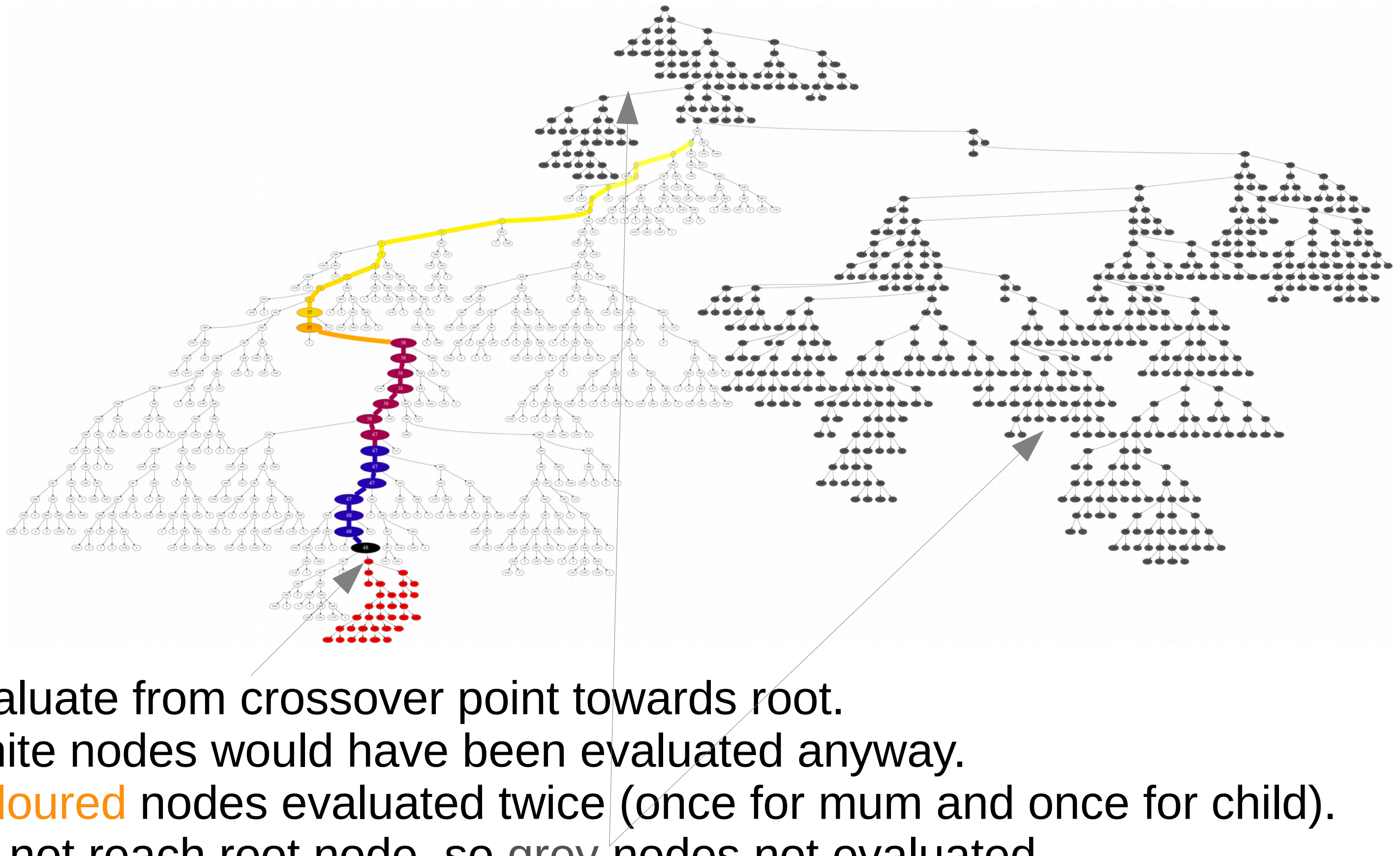


Evaluate mum in red. Evaluate child in blue. Inserted code (DIV (DIV 0.979 X) X) in blue. Here incremental evaluation proceeds 38 levels up the child tree before both mum and child evaluations are identical on all 48 test cases.

Functions lose information and so can give same output even with different inputs.

Evaluate both trees from change up

New code in red. Can stop fitness evaluation early as mum and child are phenotypically identical on all test cases.



- Evaluate from crossover point towards root.
- White nodes would have been evaluated anyway.
- Coloured nodes evaluated twice (once for mum and once for child).
- Do not reach root node, so grey nodes not evaluated.
- In deep trees incremental evaluation can be orders of magnitude faster.
  - Evaluation times very variable, giving thread imbalance

CREST

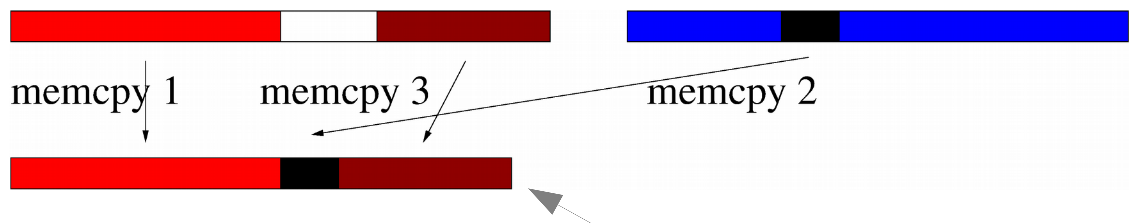
Fitness a thousand times faster: Bottleneck is memcpy in crossover, so do Fitness First

- Avoid useless crossovers and mutations
- If a child does not have children; do not create it [1].
- **Fitness** of child can be calculated exactly from parents' code
- Do this **before** creating child !
- Early in GP run with typical strong selection, 60% of population does not have any children (but trees may be small).
- Saving depends on degree of convergence and number of parents:
  - Two parents and 7-tournament, 60% have no children [GPTP 2021]
  - For 2 parents 100% convergence, e^{-2} 14% have no children.
  - 1 parent 100% convergence, e^{-1} 37% have no children.
- (No longer order of magnitude speed ups, worth having?)

Fatherless Crossover: Less memory + Faster crossover?

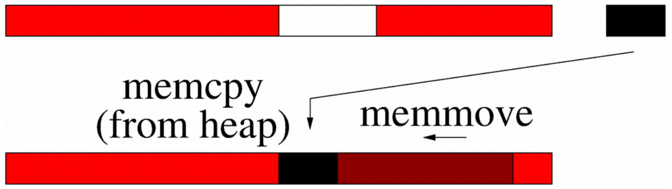
- Father donates small subtree
- Mother donates rest (ie most) of code.
- Where trees are huge, donated code is tiny in comparison and can be saved before crossover on heap (< 1 megabyte)
- Fathers no longer needed and can be deleted.
- Effectively gives 1 parent crossover.
- In converged population e^{-1} (37%) do not have children.
- Even without fitness first, some saving as more parents have only one child [next slide]
- With fitness first, memory usage can be below population size!

Regular GPquick Crossover



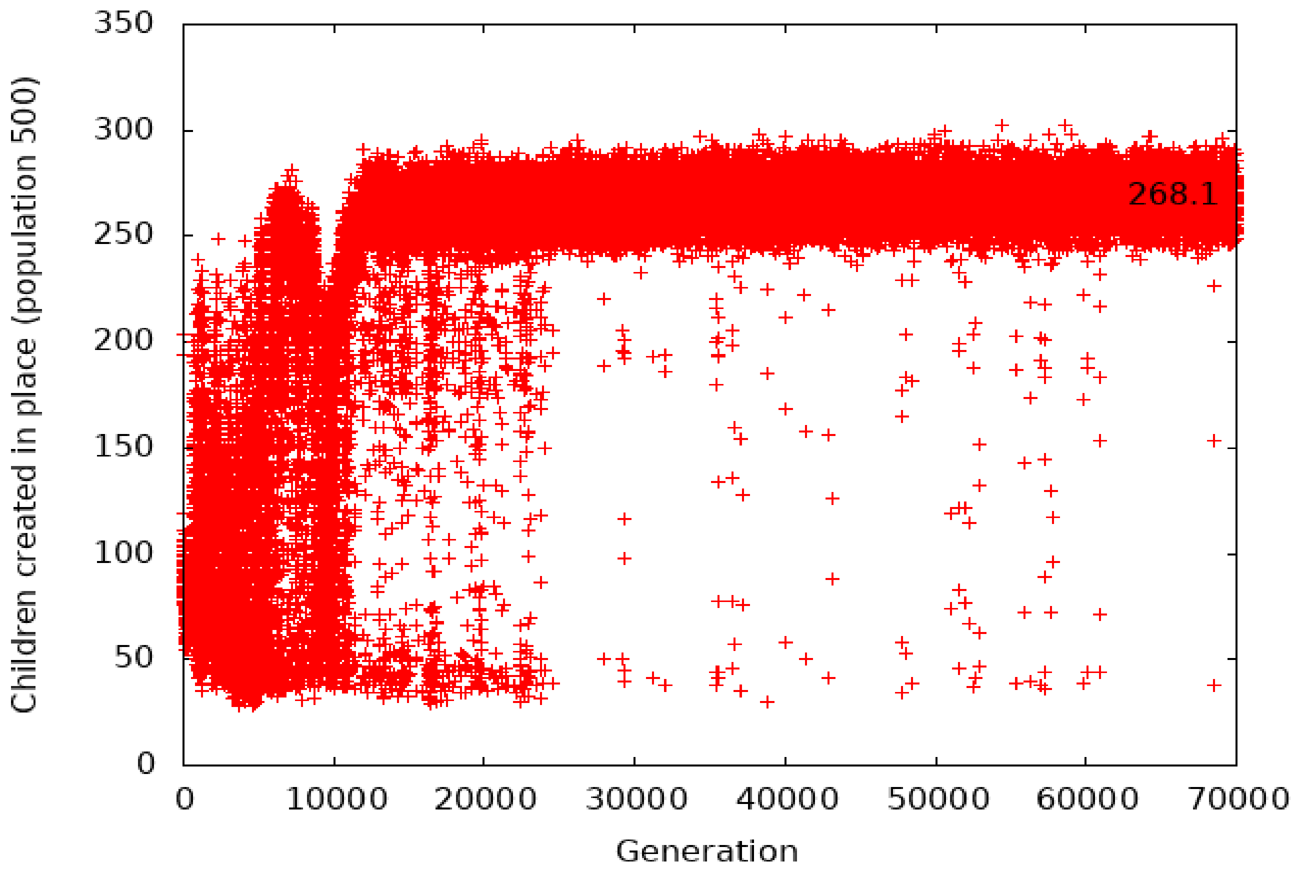
- Two parents (mum left, dad right). One child (lower).
- Subtrees chosen uniformly at random (mum white, dad black)
- One child created
  - Root of child copied from 1st parent (mum, red).
  - Subtree from 2nd parent copied (dad, black).
  - Remainder of 1st parent copied (brown).

Faster [in place] crossover: avoid copy



- Even in multi-core threaded code, all mothers eventually have one child left to create. (20 – 54% of population.)
- Where trees are huge, almost all the code for the child already exists in mum’s buffer.
- On average half of it is exactly in place.
- For mum’s last child, reuse her buffer.
- If subtrees are different sizes (73%), use memmove to shuffle on average half of the buffer up or down.
  - GCC memmove faster than memcpy.
- If inserted code is same size as removed code, need only overwrite removed code (memcpy from heap).

In place crossover avoids copy



On average 54% children created in parent's buffer (with fitness 1st and fatherless crossover)

Implementation

- Avoid “false sharing”.
  - pad `struct` to avoid threads using same cache line
  - cache align data shared by threads, eg `aligned_alloc()`
- Dont malloc/new/delete in threaded code (gcc high overhead)
- `pthread_mutex_t` faster than expected. Use on debug output
- Ensure evolution path does not depend on thread timing, eg by doing all stochastic choices in main thread
- Only crossover and fitness function optimised
- When bit shifting in C/C++
  - used `unsigned int`
  - force evaluation order by bracketing everything
- gcc 10.2.1 supports `__float128`
  - 128 bit float operations are really very slow
  - 10.5 times slower than double is considered good

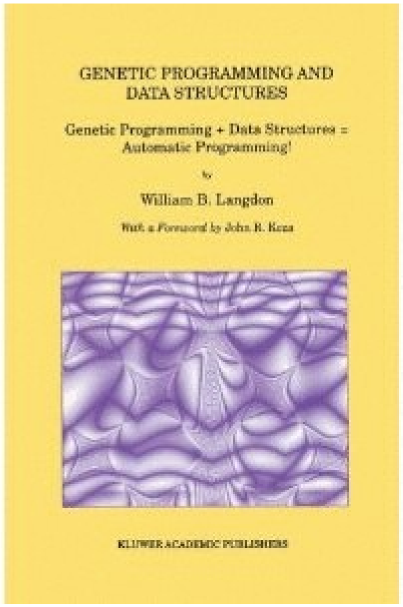
Conclusions

- Don’t collect unwanted statistics (70% speedup)
- Non-overlapping generations dont need more memory [1].
- In place crossover for 20% to 54% children (convergence etc)
- pthreads speedup typical 14x (16 threads)
- AVX typical speed up 3-4 fold (not 16 fold)
- With large trees incremental fitness more than 100x speed up possible by not doing work
- Not just genetic programming: eg Galaxy TSP needs huge chromosomes
- Study of GP convergence lead to *information theory* of widely applicable *failed disruption propagation* giving rise to software robustness and investigation of optimal test oracles [GECCO 2022 Monday GP1 11:10 and poster].

CREST

Genetic Programming

W. B. Langdon



GENETIC PROGRAMMING AND DATA STRUCTURES

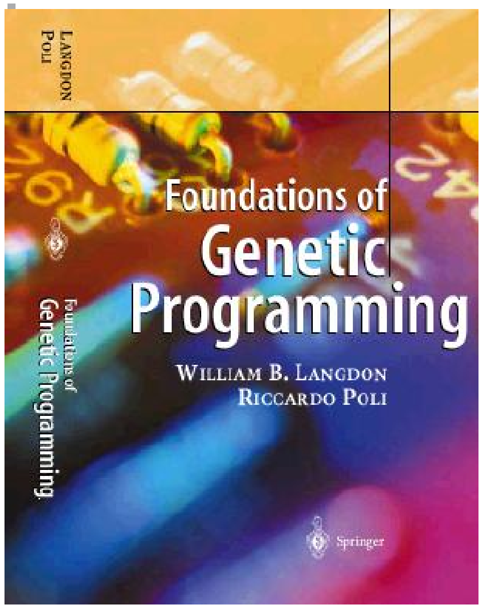
Genetic Programming + Data Structures = Automatic Programming!

by

William B. Langdon

With a Foreword by John R. Koza

KLUWER ACADEMIC PUBLISHERS

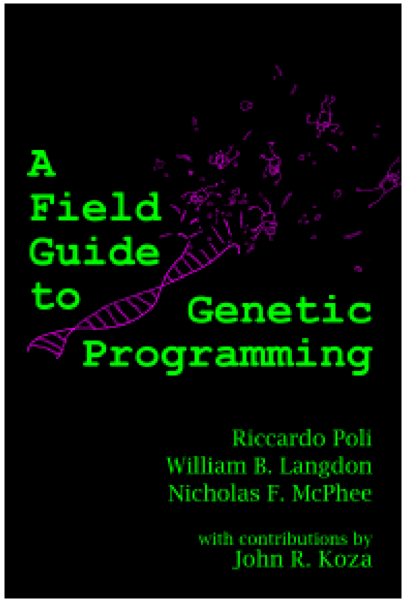


LANGDON POLI

Foundations of Genetic Programming

WILLIAM B. LANGDON
RICCARDO POLI

Springer



A Field Guide to Genetic Programming

Riccardo Poli
William B. Langdon
Nicholas F. McPhee

with contributions by
John R. Koza

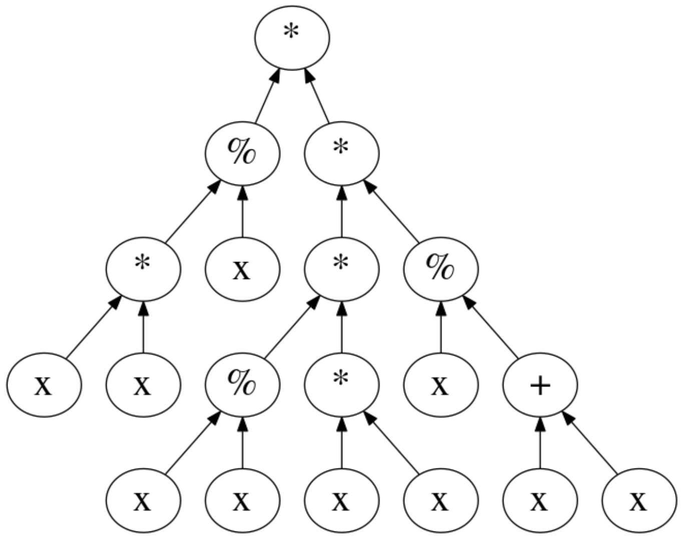
CREST

Table of GP parameters

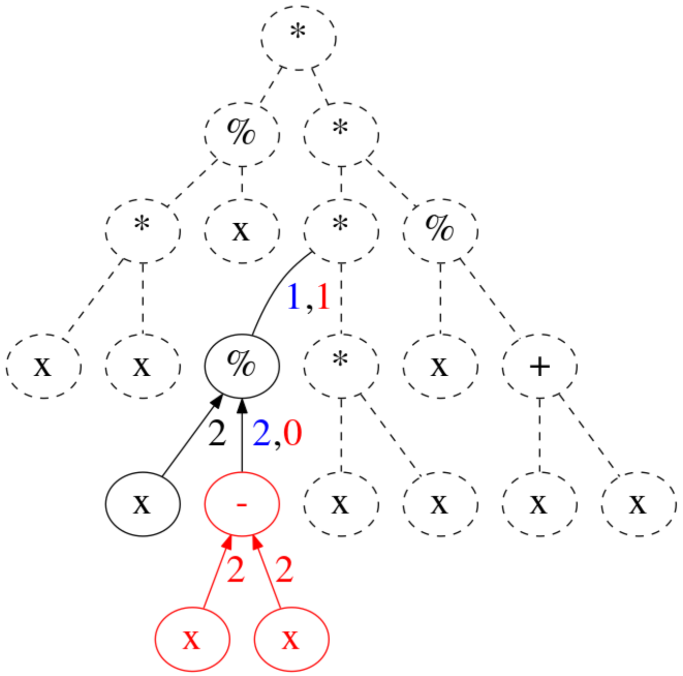
Terminal set:	X, 250 constants -0.995 to 0.997
Function set:	MUL ADD DIV SUB (Section 6.1)
	MUL ADD (Section 6.2)
Fitness cases:	48 fixed input -0.97789 to 0.979541 (randomly selected in -1.0 to +1.0). Target Sextic polynomial $y = xx(x-1)(x-1)(x+1)(x+1)$
Selection:	Fitness = $\frac{1}{48}\sum_{i=1}^{48} \lvert GP(x_i) - y_i \rvert$ tournament size 7
Population:	Panmictic, non-elitist, generational.
GP parameters:	Initial population 500 ramped half and half [14] depth between 2 and 6. 100% unbiased subtree crossover. 600 generations. No size or depth limit.

DIV and % denote protected division `(y!=0)? x/y : 1`

Crossover disruption fails to propagate to root node for test case x=2



Original code (best generation 5)



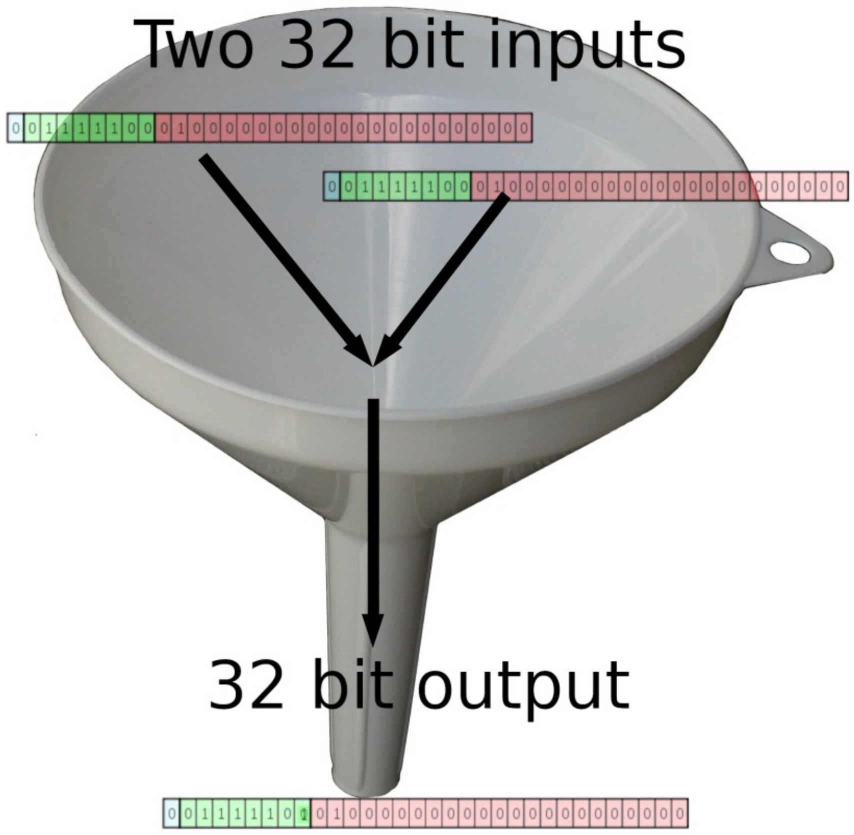
- Crossover removes "x" and inserts "(- x x)"
- On test case x=2, Eval 2 replaced by eval 0
- Protected division "(% x x)"
    - was eval 1
    - now "(% x 0)" eval 1
- Change has no impact at % or above %
- Disruption fails to propagate to root node
- No change in fitness

No side effects.
Static environment.

CREST

Information Funnel

Computer operators are irreversible. Meaning input state cannot be inferred from outputs. Information is lost

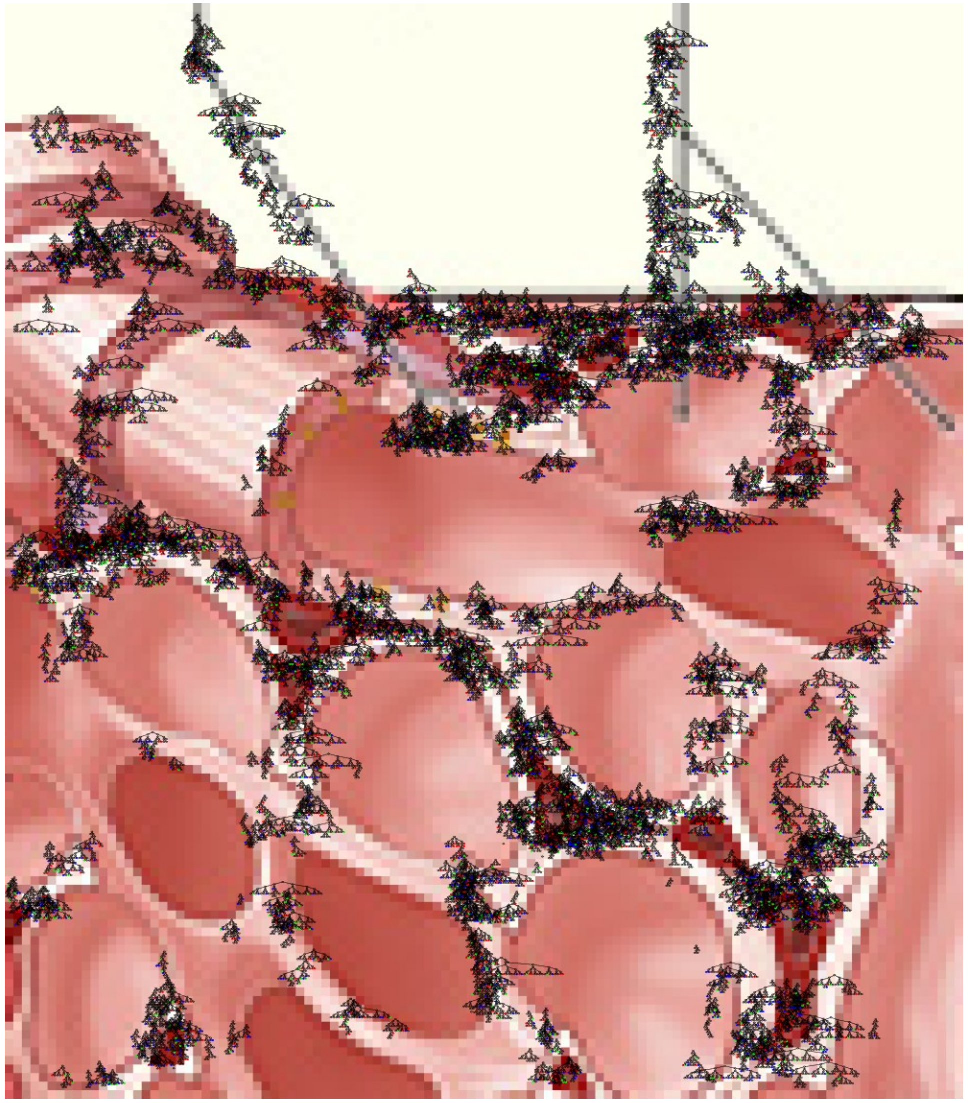


Information funnel

More information enters than leaves

Evolve Large Open, Lung Like, Open Architecture

- Make code is shallow.
- Shallow code does not suffer failed disruption propagation.
- Instead fitness disruption caused by mutations and crossover do have impact.
- Fitness can direct evolution.
- Suggest large porous code
- All code near organism’s environment.
- Communication between code internally & externally eased by globals, side effects, pipes, TCP/IP etc.

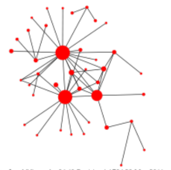


http://arxiv.org/abs/2112.00812

The Genetic Programming Bibliography

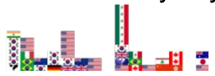
15589 references, 15000 authors

Make sure it has all of your papers!
E.g. email W.Langdon@cs.ucl.ac.uk or use | Add to It | web link



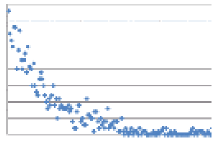
Co-authorship community.
Downloads

Downloads by day



A personalised list of every author's
GP publications.

blog



Your papers

Googling GP bibliography, eg:
Development and learning site:gpbib.cs.ucl.ac.uk