CREST

CREST Open Workshop COW62, 21st January 2020

Genetic Improvement of Genetic Programming

W. B. Langdon

Computer Science, University College London

Humies

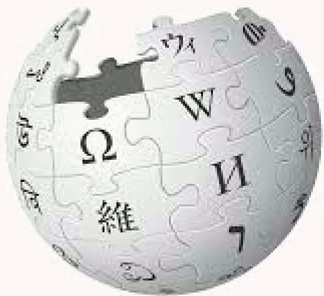
http://www.human-competitive.org/

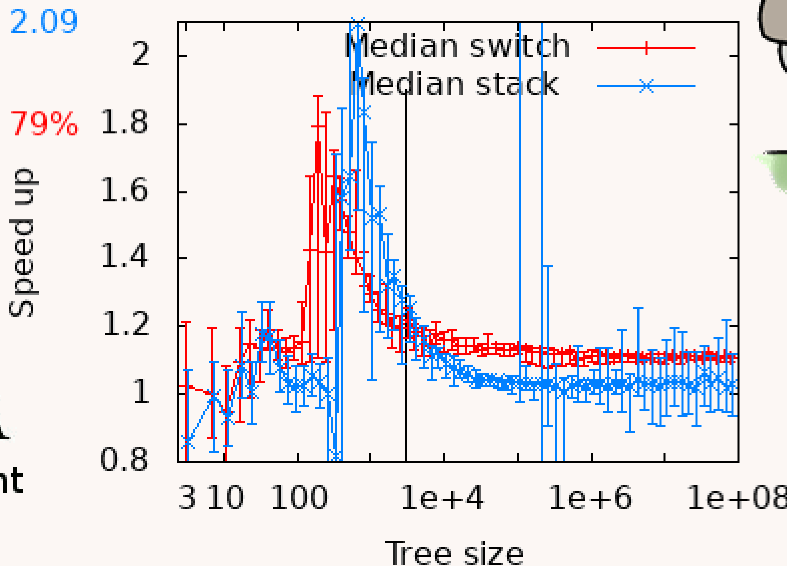
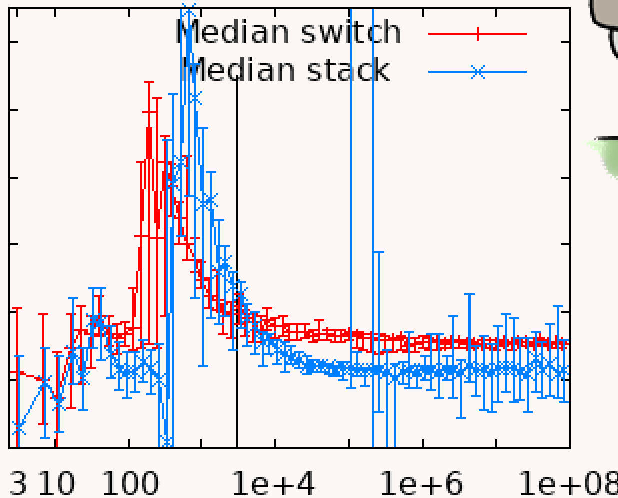
Human-Competitive results

$10,000 prizes

GECCO-2020 in Cancun, Mexico

Email 29 May goodman@msu.edu



Genetic Improvement of Genetic Programming

- Applying Genetic Improvement to own parallel C++ Genetic Programming system
- Intel AVX-512 parallel vector instructions
  - AVX does 16 float operations in parallel
- GPavx[1] written as part of existing C++ GP system, Singleton's GPquick [1993].
  - 6900 lines of code.
- GPavx can evolve trees of 100 million[2]
- Takes weeks. Overhead is AVX interpreter
- Can GI on interpreter do better?

Alternative Fast GPs

- GPavx fastest tree interpreter
- Avoid trees: Linear GP, Cartesian GP
- Avoid interpreting
  - Compile tree to machine code[1]
  - Evolve machine code: Discipulus
- Avoid interpreting whole tree, changes only
  - evolving population of trees as evolving directed acyclic graph holding partial fitness. Eval O(depth) not O(size) (no side effects)[2,3]
- Avoid interpreting dead code (introns)

CREST

Typical Genetic Programming

- Random initial population of trees

1. Test each tree, give it fitness score
2. Select better trees to be parents
3. Create next population from parents

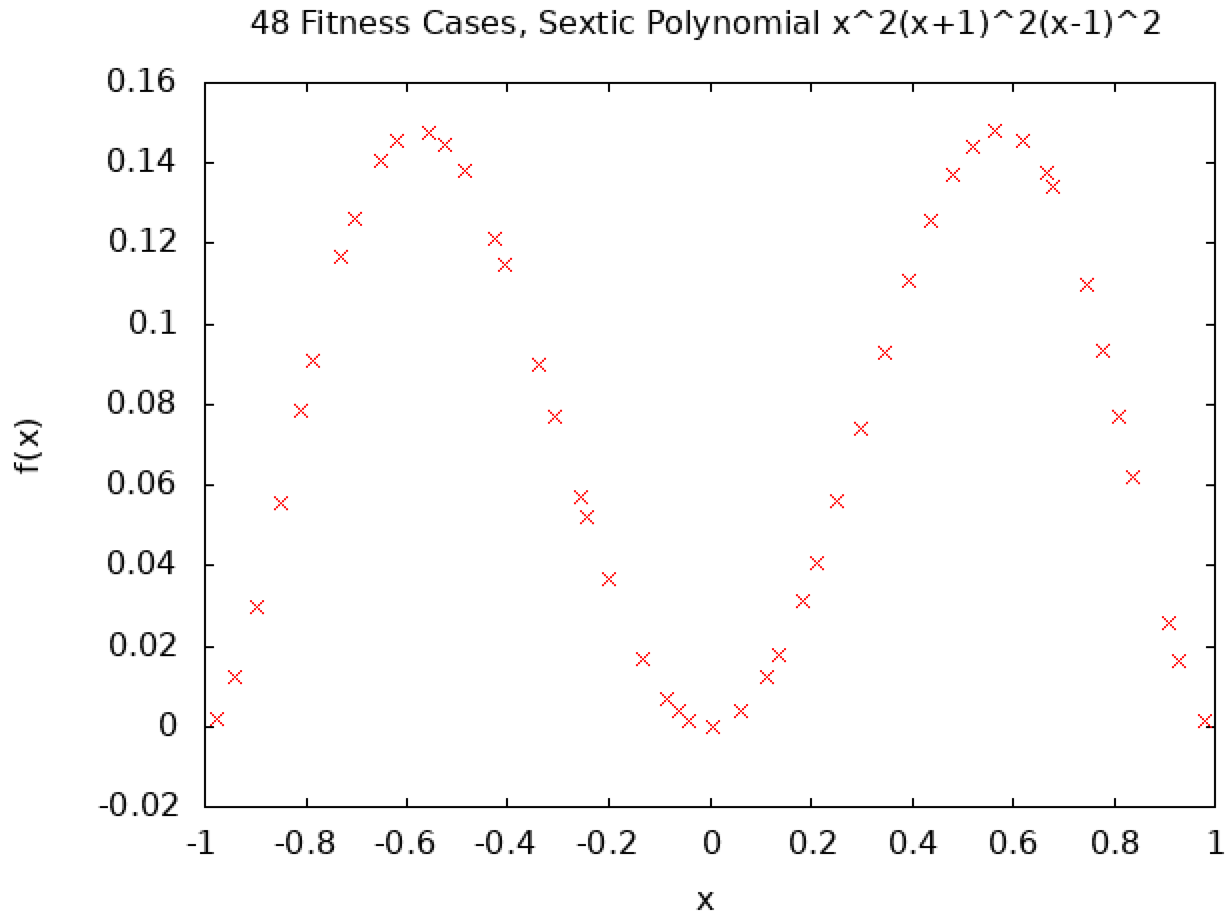
- Loop (1.) until done
- Most time is taken by fitness evaluation
  Often use multiple fitness cases.
- Parallel: multiple trees, multiple(48) fitness cases

Extreme GP

- Demo problem: match curve at 48 points
- Population up to 4000
- No tree size limit
- Run up to 1 million generations
- Trees evolve greater 100 million

  (GP continues to find improvements)
- Run on 46GB multiple core Intel server
- Parallel
  - multiple trees: one pthread per core
  - multiple fitness cases: 3×AVX = 48 eval in para

Extreme GP



Problem: match curve at 48 points

(48 chosen since multiple of 16)

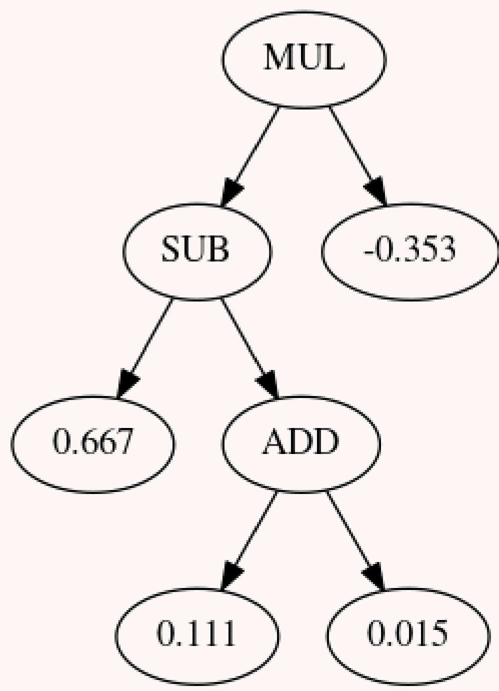
GI on avx.cc

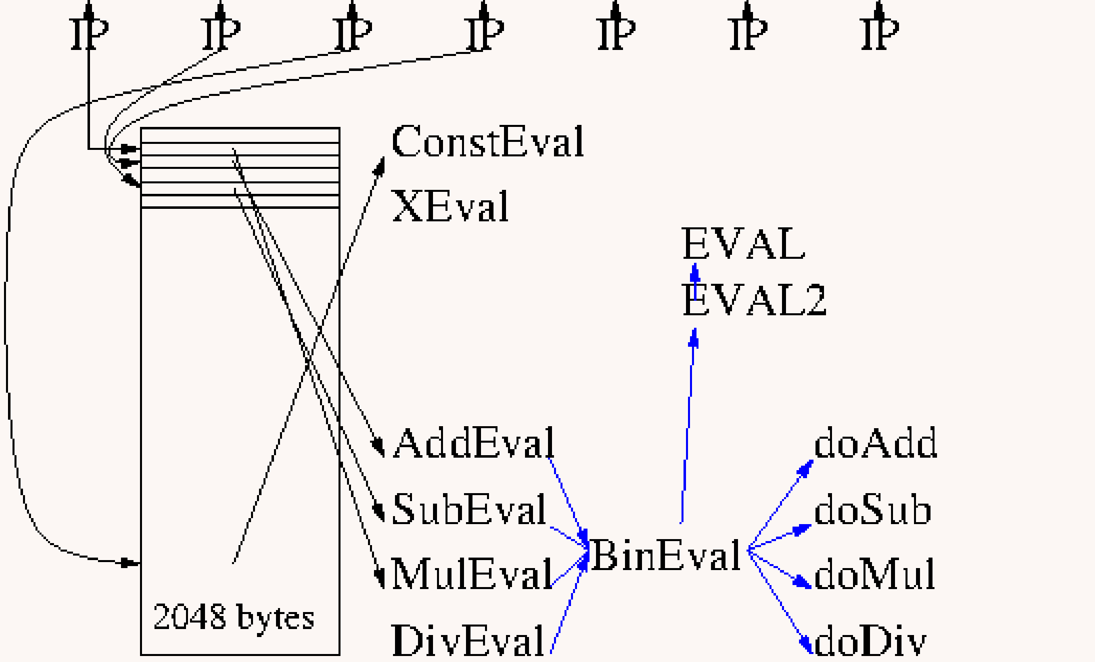
- avx.cc 448 lines of C++ code (eval 98 lines)
- Written in style of GPquick interpreters
  - OPDEF EVAL EVAL2 BINEVAL macros
- Supports + - × / (protected division) x -0.99
  - DivEval recursively EVAL both arguments
    - dodiv if(arg2==0) return 1.0f else return arg1/arg2
    - Use AVX to do 16 float operations in parallel
- Tight code (not good for evolution?)
- Six OPDEF(function)

EVAL

- EVAL (Evalfunc[(++IP)->op])(ip,sp)
- GPquick stores tree in linear array[IP]
- GPquick uses array indexed by op code to call interpreter code for op code.
  - Not switch.
- Tree 1 byte per node
- Up to 255 opcodes, Evalfunc[256] addresses = 256×8bytes = 2048 bytes = 32 cache lines

EVAL (Evalfunc[(++IP)->op])(ip,sp)





Evalfunc[256] array of function addresses

Re-entrant EVAL(ip,sp)

- To support reentrant multi-threading, replaced original global instruction pointer (i.e. point to active tree node) by passing IP as (hidden) argument to EVAL.
- Similarly pass stack pointer as EVAL arg sp
- Explicit stack
- Each thread has own IP and stack

Recursive Evaluation of 48 floats

- Each function recursively calls EVAL2.
- EVAL2 calls EVAL twice.
- Each EVAL leaves its answer on the (explicit) stack.
- Function, e.g. AddEval, pops twice, doAdd adds values, AddEval pushes result.
- XEval push value of x onto stack
- ConstEval pushes 48 copies of const
- Outermost Eval returns vector 48 floats

Multiple Test Cases Evaluate 48 in parallel

- Typically tree evaluated once per test case
- AVX do 16 test cases simultaneously.
- By making stack 48 floats wide, can eval whole tree in one pass by doing three AVX (sequentially).
- Eval returns vector 48 floats
- Fitness = for(i=0;i<48;i++) sum += $|error_i|$

  (not reduction, GPavx gives identical answers)

CREST

Applying Genetic Improvement to avx.cc

- Paper has lots of experiments, concentrate on last one.
- Want to evolve fast mutant for random tree of ≈twenty million
- Show it generalises to trees of size 3 to 100 million

Applying GI to avx.cc

- Automatically convert C++ to grammar
- Evolve grammar: mutation + crossover
- Fitness:
  - Does mutant compile, run, return right answers
  - Test on random tree (change each generation)
  - Compare with original code
    - All 48 answers the same?
    - Run time?
- Select better half of population as parents

Fitness Function, wall clock time

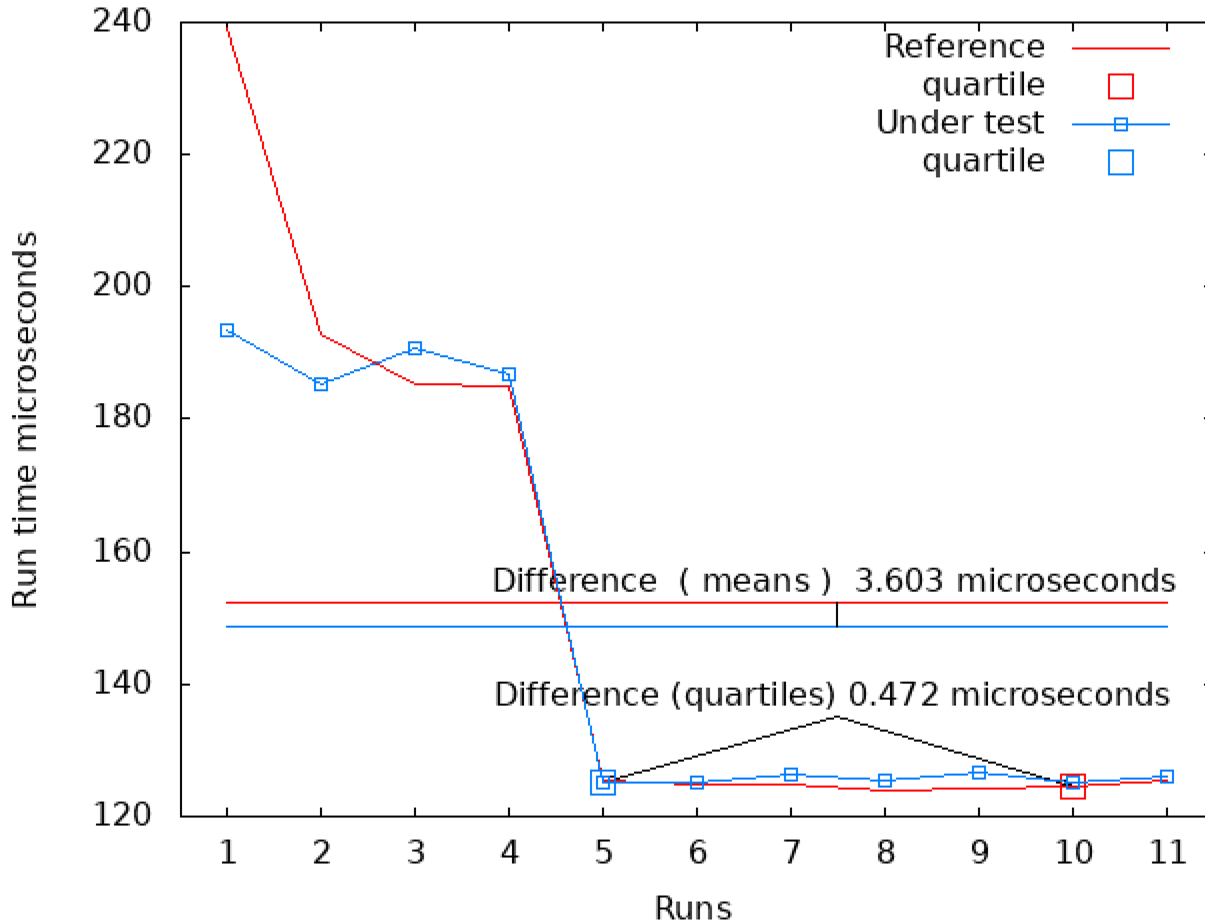
- Each mutation run independently (own exe)
- Run on multi-user AVX-512 server
  - Load varies with other users
  - Server dynamically changes each CPU core's clock frequency (1.00-3.00 GHz nominal 2.30)
  - OS sometimes moves process between cores
- Noise!

(less with unix perf stat instruction:u ?)

Combating Fitness Noise

- Use single core
- Performance relative to original code
  - small fast (≈0.1μS) trees usually on same core
  - usually same clock frequency
- 11 small runs. Difference in quartile time
- Only run fast mutants on big trees(≈1/2sec)
- Noise proportionately less, so run once

Use Quartile to combat runtime noise



Example of using quartile difference in run time. Reference code in red. Effectively only use faster half of runs, then take robust average (median). Fitness = 472

Mean not used as dominated by outliers.

Six EVAL environments

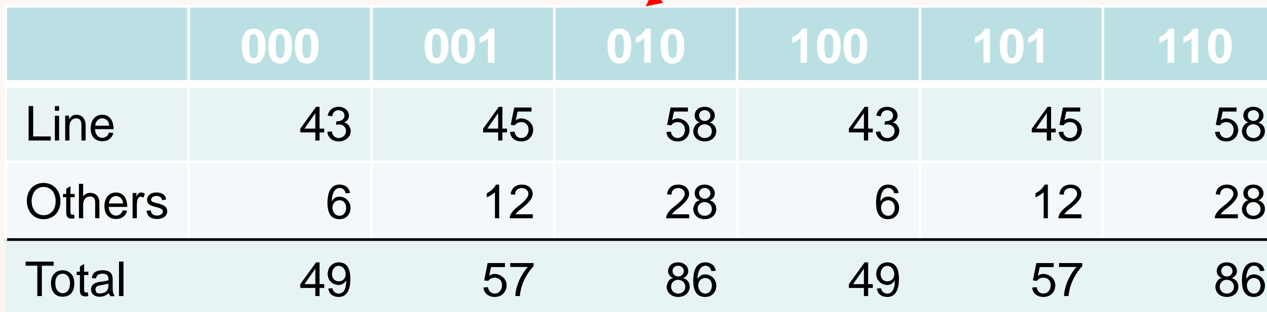
3 conditional compilation –D macros

- New switch code v. jump table
- Internal v. External stack
- (if internal) interpret tree 3 times or return 48 floats

Gives 6 options

Six avx.cc EVAL grammars

- Each option has own grammar
  - 5694 functions from Intel Intrinsics library
  - Most variable rules are type line
  - Few other types
- Concentrate upon option 010 (switch and explicit stack) as most successful.
- All six run ok



	000	001	010	100	101	110
Line	43	45	58	43	45	58
Others	6	12	28	6	12	28
Total	49	57	86	49	57	86

avx.cc EVAL with switch macro

```
#define EVAL() \
switch ((++IP)->op) {\
 case mul_op: MulEval_gp(ip,sp); break;        \
 case div_op: DivEval_gp(ip,sp); break;        \
 case sub_op: SubEval_gp(ip,sp); break;        \
 case x_op:   XEval_gp(ip,sp); break;          \
 case add_op: AddEval_gp(ip,sp); break;        \
 default:     ConstEval_gp(ip,sp); break;      \
}
```

```
29             ::= "#define EVAL() "
30             ::= "switch ((++IP)->op) {"
31             ::= <CASE_31>
<CASE_31>      ::= "case add_op: AddEval_gp(ip,sp); break;"
32             ::= <CASE_32>
<CASE_32>      ::= "case sub_op: SubEval_gp(ip,sp); break;"
33             ::= <CASE_33>
<CASE_33>      ::= "case mul_op: MulEval_gp(ip,sp); break;"
34             ::= <CASE_34>
<CASE_34>      ::= "case div_op: DivEval_gp(ip,sp); break;"
35             ::= <CASE_35>
<CASE_35>      ::= "case x_op:   XEval_gp(ip,sp); break;"
36             ::= <CASE_36>
<CASE_36>      ::= "default:     ConstEval_gp(ip,sp); break;"
37             ::= "}"
```

Swap mutation, eg <CASE_32>x<CASE_34>, allows easy re-ordering of case and default statements.

CREST

avx.cc Eval Leafs const, x

```
OPDEF(ConstEval_gp) {
  retval val;
  int i;
  val = GETVAL;
  for(i=0;i<MAXTESTCASES;i+=8) {
    _mm256_store_ps(&EvalSP[ i ],
                    _mm256_set1_ps(val));
  }
  inc_EvalSP;
}
OPDEF(XEval_gp){
  int i;
  for(i=0;i<MAXTESTCASES;i+=16)  {
    _mm512_store_ps(&EvalSP[ i ],
                    _mm512_load_ps(&dataX[ i ]));
  }
  inc_EvalSP;
}
```

Can not mutate: declarations, for and }

Comments removed.

CREST

avx.cc switch ConstEval grammar

```
68                  ::=     "OPDEF(ConstEval_gp) {"
77                  ::=     "retval val;"
78                  ::=     <line_78>
<line_78>           ::=     "{};"
79                  ::=     "int i;"
80                  ::=     <line_80>
<line_80>           ::=     "{};"
83                  ::=     <line_83>
<line_83>           ::=     "val = GETVAL;"
84                  ::=     <line_84>
<line_84>           ::=     "{};"
85                  ::=     "for(" "i=0" ";" "i<MAXTESTCASES" ";" "i+=" <vecsize> ")  {"
88                  ::=     <void(float*,veci)_1_88> "(&" <float*_2_88> "[ " <veci_3_88> " ],"
<void(float*,veci)_1_88>        ::=     "_mm256_store_ps"
<float*_2_88>                   ::=     "EvalSP"
<veci_3_88>                     ::=     "i"
89                              ::=     <veci(float)_1_89> "(" <float_2_89> "));"
<veci(float)_1_89>              ::=     "_mm256_set1_ps"
<float_2_89>                    ::=     "val"
90                              ::=     <line_90>
<line_90>                       ::=     "{};"
91                  ::=     "}"
92                  ::=     <line_92>
<line_92>           ::=     "inc_EvalSP;"
94                  ::=     "}"
```

Variable rules <type_etc>, e.g. type line, float*

Mutate rules of same type <line_92>x<line_84>

veci rules depend on GI vecsize (4,8 or 16)

CREST

avx.cc Eval functions

```
inline OPDEF(Eval2_gp) {
  retval* e0;
  EVAL;
  e0 = EvalSP;
  EVAL;
  dec2_EvalSP;
}
inline __m512 doadd_gp(const __m512 a, const __m512 b){return _mm512_add_ps(a,b);}
inline __m512 dosub_gp(const __m512 a, const __m512 b){return _mm512_sub_ps(a,b);}
inline __m512 domul_gp(const __m512 a, const __m512 b){return _mm512_mul_ps(a,b);}
inline __m512 dodiv_gp(const __m512 numerator, const __m512 denominator){
  __m512 zero;
  __mmask16 mask;
  __m512 val;
  __m512 one;
  __m512 ans;
  zero = _mm512_set1_ps(0.0f);
  memset(&zero,0,sizeof(__m512));
  zero = (__m512){0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f};
  mask = _mm512_cmpneq_epi32_mask((__m512i)denominator,
                                  (__m512i)zero);
  mask = _mm512_cmpneq_ps_mask(denominator,
                               zero);
  val = _mm512_maskz_div_ps(mask,
                            numerator,
                            denominator);
  one = _mm512_set1_ps(1.0f);
  one = (__m512){1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f};
  ans = _mm512_mask_blend_ps(mask,
                             one,
                             val);
  return ans;
}
inline OPDEF2(binEval_gp,__m512 f(const __m512 a, const __m512 b)) {
  int i;
  __m512 sp0;
  __m512 sp1;
  __m512 val;
  EVAL2;
  for(i=0;i<MAXTESTCASES;i+=16)  {
    sp0 = _mm512_load_ps(&EvalSP[ i ]);
    sp1 = _mm512_load_ps(&EvalSP[ MAXTESTCASES+i ]);
    val = f(sp0,
            sp1);
    _mm512_store_ps(&EvalSP[ i ],
                    val);
  }
  inc_EvalSP;
}
OPDEF(AddEval_gp) { return BINEVAL(doadd_gp);}
OPDEF(SubEval_gp) { return BINEVAL(dosub_gp);}
OPDEF(MulEval_gp) { return BINEVAL(domul_gp);}
OPDEF(DivEval_gp) { return BINEVAL(dodiv_gp);}
#if(0)
__asm__ __volatile__ ( "vzeroupper" : : : );
#endif
```

Multiple ways to set zero, mask, one, to allow –O2 compiler and evolution to choose best.

Tightly written code only allows mutations in Eval2, dodiv, binEval. Also zeroupper can be copied into them

Cf slide 9.

avx.cc switch Eval2 grammar

```
inline OPDEF(Eval2_gp) {
  retval* e0;
  EVAL;
  e0 = EvalSP;
  EVAL;
  dec2_EvalSP;
}
```

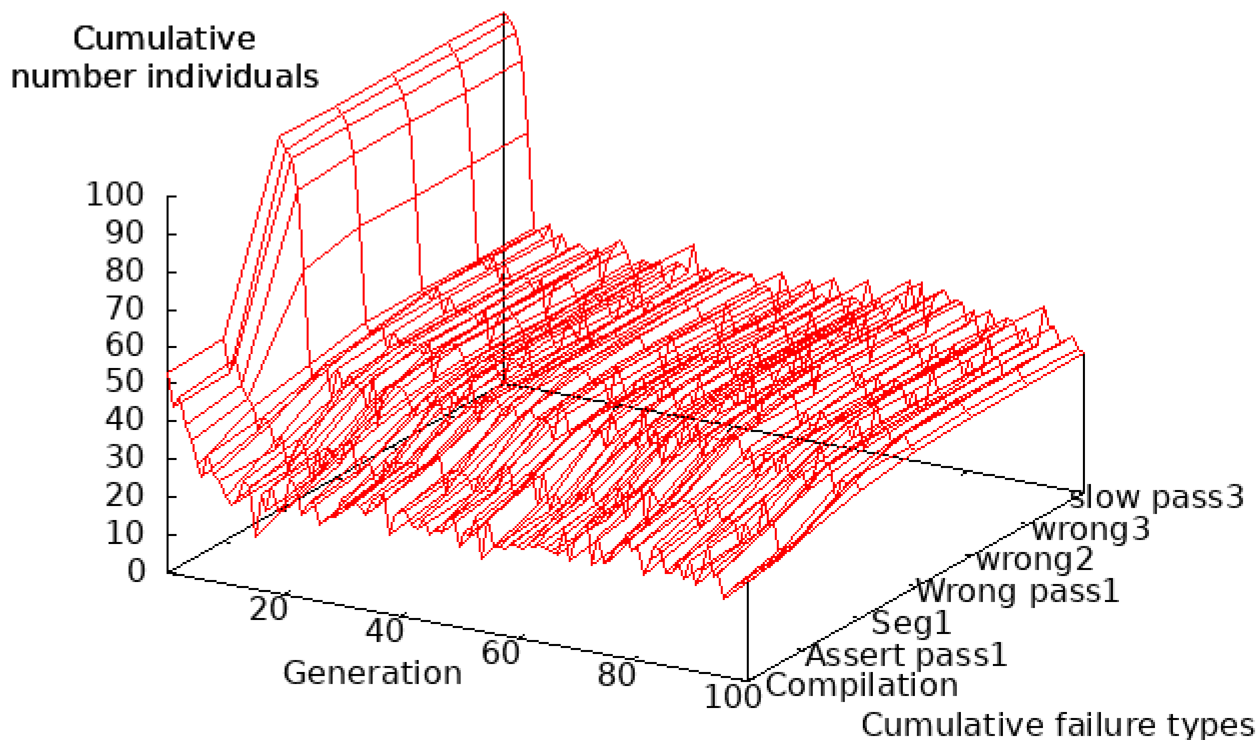
```
131         ::=     "inline"
132         ::=     "OPDEF(Eval2_gp) {"
134         ::=     "retval* e0;"
135         ::=     <line_135>
<line_135>  ::=     "{};"
137         ::=     <line_137>
<line_137>  ::=     "e0 = EvalSP;"
138         ::=     <line_138>
<line_138>  ::=     "{};"
139         ::=     <line_139>
<line_139>  ::=     "EVAL;"
140         ::=     <line_140>
<line_140>  ::=     "{};"
142         ::=     <line_142>
<line_142>  ::=     "EVAL;"
143         ::=     <line_143>
<line_143>  ::=     "{};"
146         ::=     <line_146>
<line_146>  ::=     "dec2_EvalSP;"
148         ::=     "}"
```

- All variable rules for Eval2 are of type line
  - Can be deleted, inserted, replaced or swapped with any other grammar rule of type line
- Variable e0 left over from earlier debug version

6 runs

- Population 100, up to generation 100.
- Takes ≈11 hours (3.8 seconds per mutant)

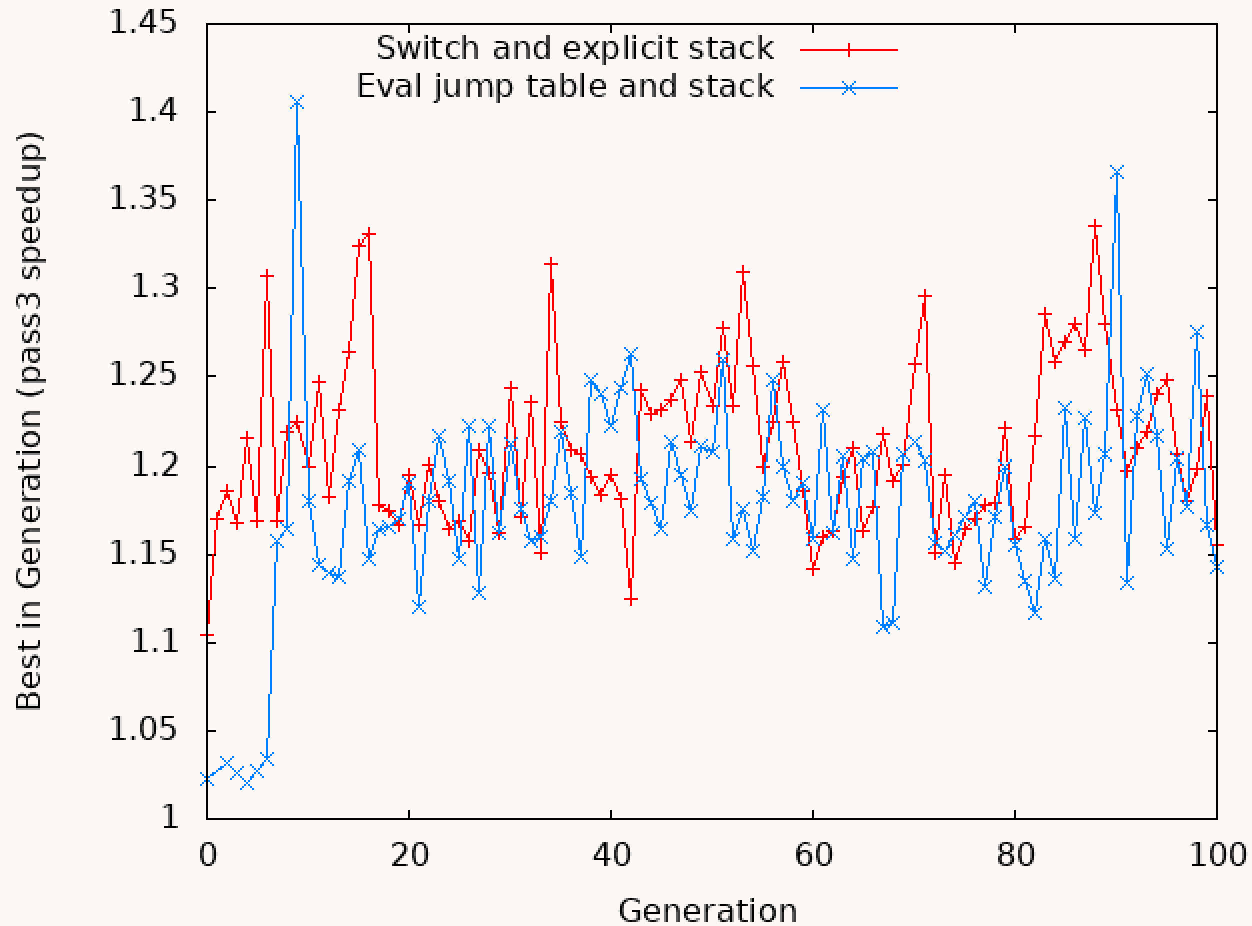
Number of mutants which fail at each generation



Overall 57% give pass3 speedup, 27% fail to compile, 9% segfault in pass1.

Pass 2+3 errors < 1%

Evolution of Fitness



Mutant Clean Up

- Use best in last generation.
- Typically bloated (i.e. BNF changes with no external impact)
- Compile as far as assembler g++ -S
- Scan whole mutant.
  - Remove each BNF gene one at a time
  - Iff .s different restore else remove permanently
- Scan again, in case can now remove more

Mutant Clean Up

USING ASSEMBLER CODE AS A GUIDE ALLOWED NOISELESS REMOVAL OF INEFFECTIVE CODE FROM BEST OF GENERATION 100 GI INDIVIDUALS

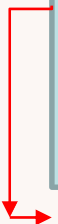
Environment		Number of genes evolved	final
010	switch and explicit stack	16	6
110	original GPavx	20	3
101	PJT jump table and eval in one pass (T48)	23	2

- Only 6 of 16 genes impact assembler code generated by g++ compiler
- No speed up in other three runs 000, 001, 100

Switch Mutant

Environment	size	Speedup p2	Speedup p3	Speedup 995	Speedup 20 056 365
010	6	40%±0.2%	16%	30%±4%	11%±0.7%

vecsize=16 <CASE_alleval.cc_34>x<CASE_alleval.cc_32> <CASE_alleval.cc_34>x<CASE_alleval.cc_33> <_alleval.cc_139>x<_alleval.cc_137> <CASE_alleval.cc_35>x<CASE_alleval.cc_34> <CASE_alleval.cc_35>x<CASE_alleval.cc_31>



Ensure AVX512 SIMD instructions are used through out

<CASE_ mutants reorder case: statements in eval dispatch switch moving XEval closer to switch statement and placing MUL and DIV before ADD and SUB.

Swapping lines 139 and 137 moves the assignment of e0 to later in Eval2. e0 is not used. Perhaps moving it makes it easier for the optimising g++ compiler to remove.

CREST

Switch Mutant code changes

```
32,33d31
< case add_op: EVAL_(x,AddEval_gp,T); break;
< case sub_op: EVAL_(x,SubEval_gp,T); break;
35a34
> case sub_op: EVAL_(x,SubEval_gp,T); break;
36a36
> case add_op: EVAL_(x,AddEval_gp,T); break;
60,62c60,62
< for(i=0;i<MAXTESTCASES;i+=8)  {
< _mm256_store_ps(&EvalSP[ i ],
< _mm256_set1_ps(val));
---
> for(i=0;i<MAXTESTCASES;i+=16)  {
> _mm512_store_ps(&EvalSP[ i ],
> _mm512_set1_ps(val));
81,82d80
< e0 = EvalSP;
< {};
83a82,83
> {};
> e0 = EvalSP;
```

3 CASE swap reorder switch

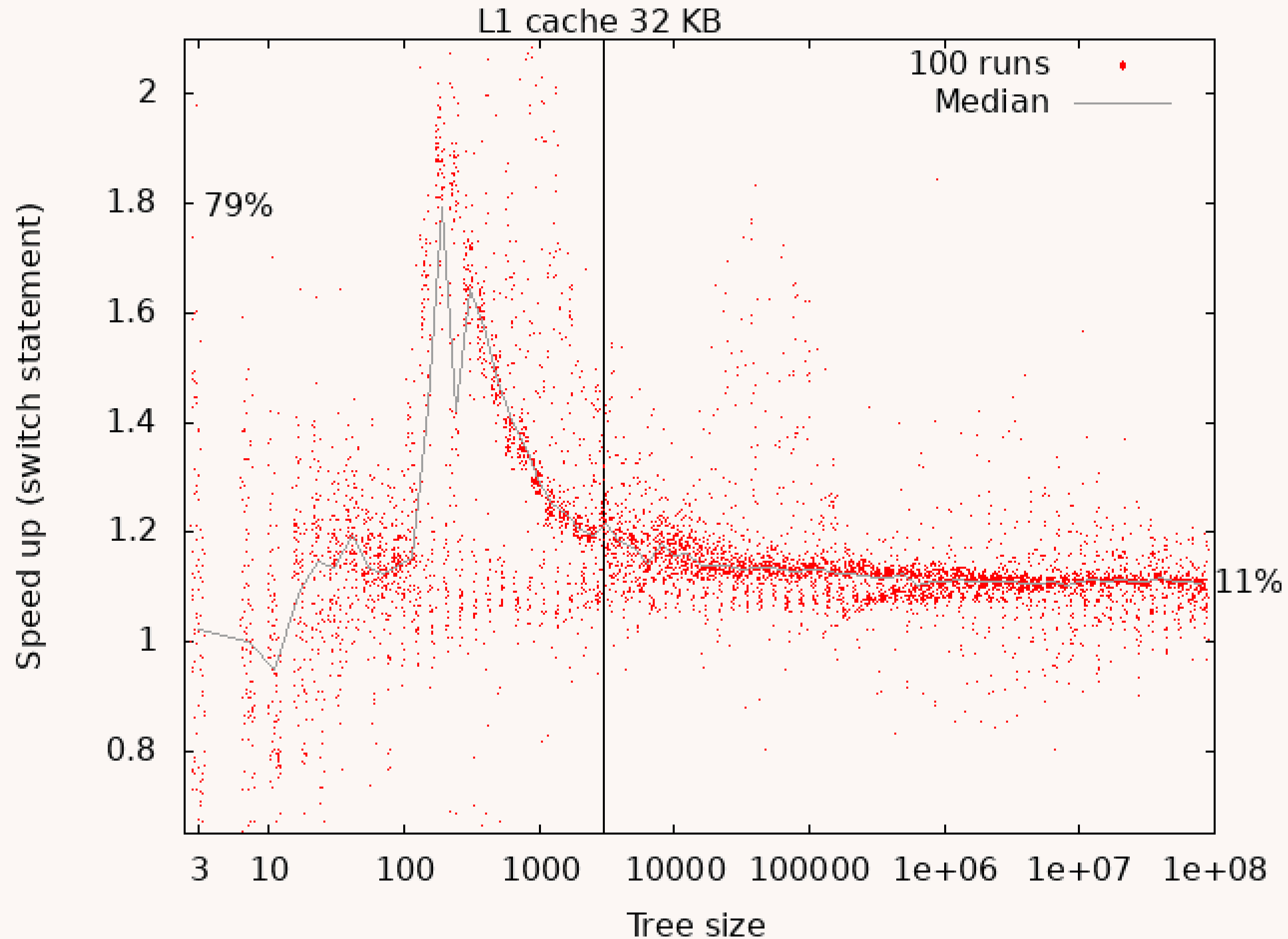
vecsize=16 forces use of AVX-512

Swapping lines 139 and 139 probably no effect

< Original

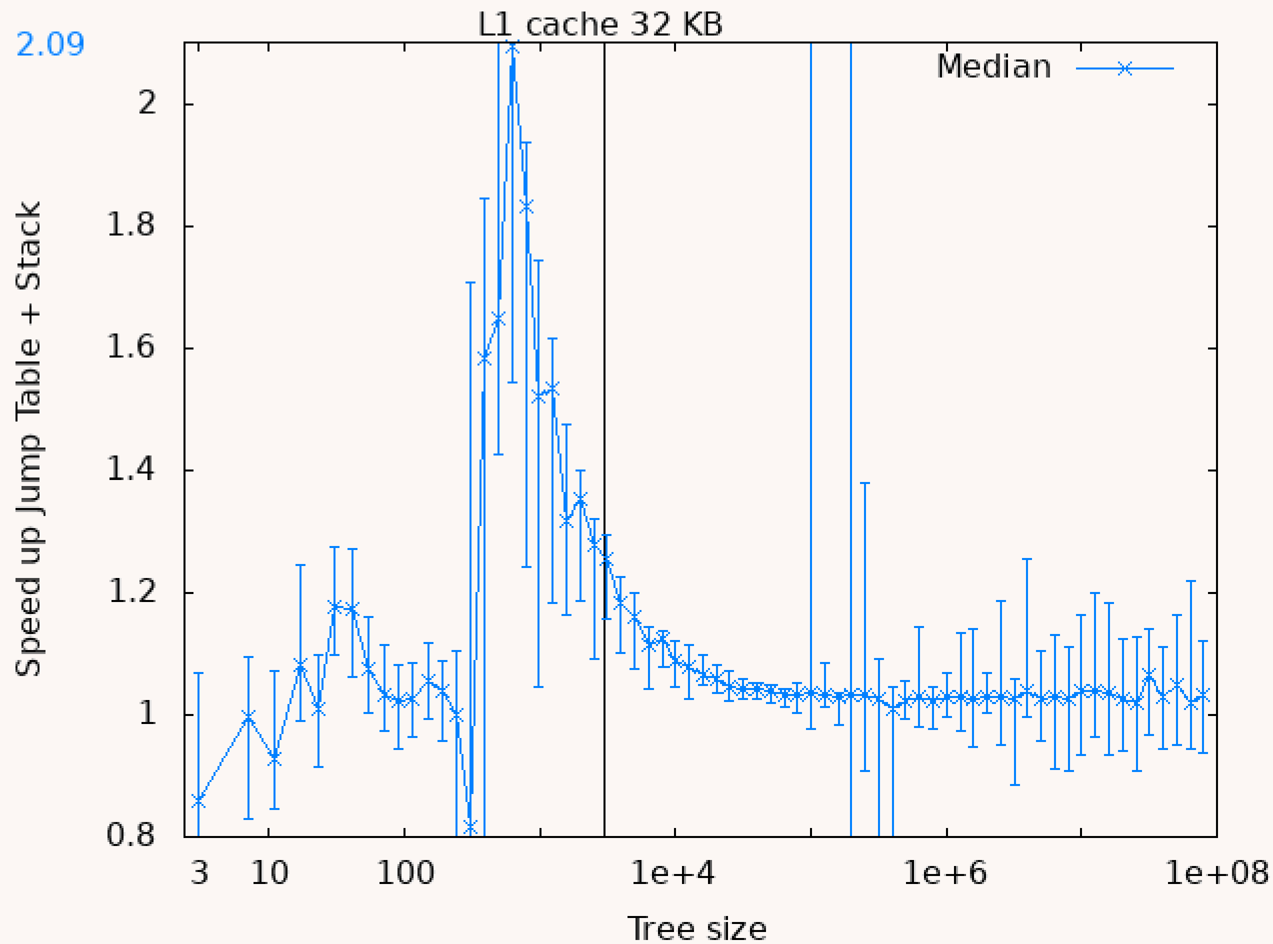
> Mutated code

After clean up: Out of Sample Generalisation



CREST

After clean up: Out of Sample Generalisation



Conclusions

- Have applied GI to my own code.
- GI has found mutants to fastest tree based genetic programming interpreter which speed it up (up to 2.1×)
- 20+ years established wisdom overturned
  - Jump table forced out of cache so switch faster
- Performance of large trees limited by cache
- Can use real runtime as fitness even on noisy time sharing cluster with dynamic power management.

CREST

GI 2020

Genetic Improvement of Software

geneticimprovementofsoftware.com

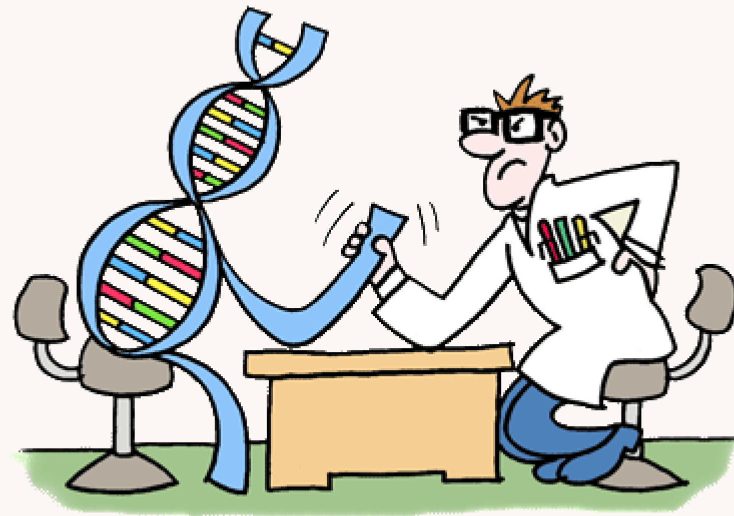
Workshop at ICSE 2020

- Position papers (1 or 2 pages)
- Research papers (up to 8 pages)

Submissions due **22 January**

https://icse20-gi8.hotcrp.com/

HotCRP



Humies

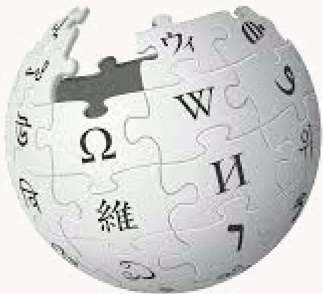
http://www.human-competitive.org/

Awards for Human-Competitive results

$10,000 prizes

Presentation at GECCO-2020 in Cancun, Mexico

send email before 29 May to goodman@msu.edu



WIKIPEDIA

Genetic Improvement

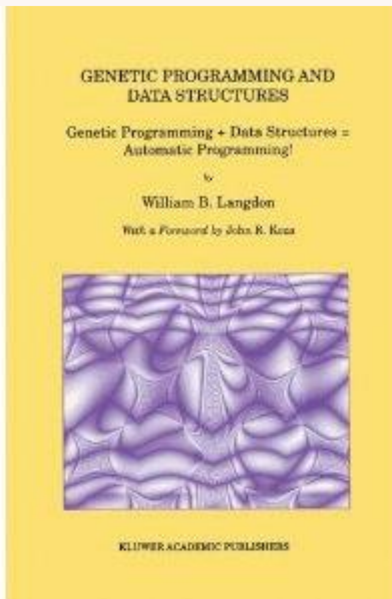
http://www.epsrc.ac.uk/ EPSRC

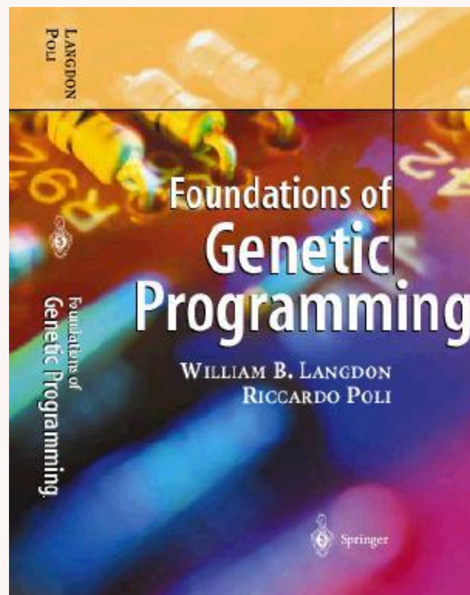
END

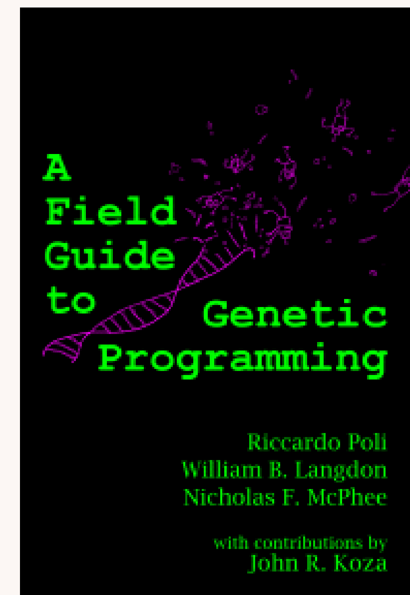
http://www.cs.ucl.ac.uk/staff/W.Langdon/

Genetic Programming

W. B. Langdon
CREST
Department of Computer Science







Six impossible things before breakfast

- To have impact do something considered impossible.
- If you believe software is fragile you will not only be wrong but shut out the possibility of mutating it into something better.
- Genetic Improvement has repeatedly shown mutation need not be disastrous and can lead to great things.

CREST

GI Parameters (6 experiments)

GI TO IMPROVE AVX CONSTEVAL, DIVEVAL AND ALLEVAL GPQUICK

Representation:	variable list of replacements, deletions, swaps and insertions into BNF grammar. Grammars variable rules: const 15, div 67, all 129, split (000 49, 001 57, 010 86, 100 49, 101 57, 110 86), plus 5694 for Intel Intrinsics library
Fitness:	compile with g++ 9.2.0 -O3 -DNDEBUG -pthread -march=skylake-avx512. (Although the pthread interface is retained, in these experiments all the code runs in a single thread.) Run on random test cases changed every generation. Sizes chosen with log distribution (so half lie below 1000 and half above). const: between 100 and 10 000 randomly chosen ERCs. div: (100/3–10000/3 random ($\div$ c_1 c_2) trees. all: 1 random tree between 100 and 10 000. Fitness is difference in first quartiles of elapsed time between original and evolved C++ code.
Population:	100 500), panmictic, no elite, generational.
Parameters:	Initial population of random single mutants. Best 51 fitness cases selected to be parents 50% two point crossover, 50% mutation. No size limit. Stop after 100 generations.

The Genetic Programming Bibliography

http://gpbib.cs.ucl.ac.uk/

13435 references, 12000 authors

Make sure it has all of your papers!

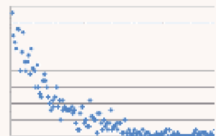
E.g. email W.Langdon@cs.ucl.ac.uk or use | Add to It | web link

RSS Support available through the Collection of CS Bibliographies.

A web form for adding your entries. Co-authorship community. Downloads

Downloads by day

A personalised list of every author's GP publications.



Your papers

blog

Search the GP Bibliography at

http://liinwww.ira.uka.de/bibliography/Ai/genetic.programming.html