Fitness First and Fatherless Crossover

William B. Langdon
W.Langdon@cs.ucl.ac.uk
Department of Computer Science, University College London
London, UK

ABSTRACT

When programs are very large and fitness evaluation is fast (e.g. due to evaluating only a tiny fraction of the program) it can be efficient to evaluate fitness before creating the program, as only individuals with children need be created. The saving can be $> 50\%$ and even in highly converged populations, it saves $e^{-2} = 13.5\%$.

In GP crossover one child is created from two parents but the root donating parent (mum) contributes far more than the other (dad). The subtree from the father is usually small and can be extracted from each dad and saved before crossover. This gives single parent crossover, which when combined with fitness first, further reduces the number of crossovers. Even in the worst case the reduction is exp(-1) = 37 percent.

With large trees, even in populations of similar fitness, eliminating bachelors and spinsters is feasible and can reduce both runtime and memory consumption. Storage in a (N) multi-threaded implementation for a population M is about 0.63M + N, compared to the usual M+2N, in practice saving $\geq 17\%$. We achieve 692 billion GP operations per second, 692 giga GPops, on an Intel i7-9800X 16 thread 3.8GHz desktop (CPU bandwidth 85 GByte/second).

KEYWORDS

Genetic programming, parallel computing, MIMD and AVX-512 SIMD, fast tree evaluation, Speedup technique, runt free broods, memory efficient GA, generational EA, tournament selection, convergence, extended evolution, Long-Term Evolution Experiment, LTEE, Extended unbounded evolution, unlimited bloat

ACM Reference Format:
William B. Langdon. 2021. Fitness First and Fatherless Crossover. In *2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21 Companion), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3449726.3459437

1 FITNESS BEFORE CROSSOVER

The latest developments [2] mean in extended unconstrained GP runs (we run to 70 000 generations) the primary cost is creating and storing the next generation. The cost of subtree crossover can be reduced by 1) doing crossover after fitness and 2) separating the subtree donating parent (the dad). See Figures 1, 2 and 3.

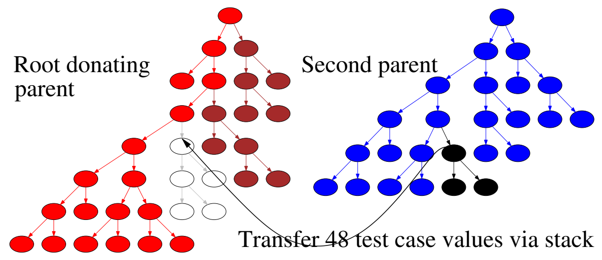

Figure 1: fitness is evaluated using only parents, i.e., before the child is created by crossover. Assuming no side effects, the subtree to be inserted (black) is evaluated on all test cases and values are transferred to evaluation of mum at the location of the subtree to be removed (white). We use incremental evaluation [2], so differences between original code (white subtree) and new are propagated up 1st parent (mum) until either all differences are zero or we reach the root node.

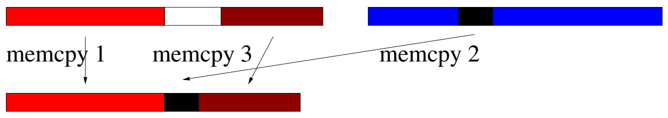

Figure 2: GPquick subtree crossover requires three memcpy buffer copies: 1) root segment of donating parent (mum, red/brown) is copied to offspring buffer. 2) subtree from second parent (dad, blue/black) is copied to offspring. 3) tail (brown) of 1st parent copied to child.

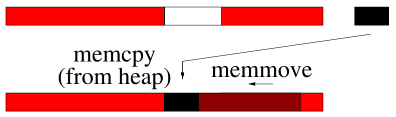

Figure 3: inplace subtree crossover. Offspring is last child of 1st parent and reuses its buffer. As with Fig. 2, only subtree to be inserted (black) of 2nd parent (dad) is kept. It is stored on the heap. 1) In 71% of children the subtree to be remove (white) and to be inserted (black) are different sizes, and so memmove is used to shuffle the second part of mum's buffer (brown) up or down. 2) Dad subtree overwrites the buffer.

Figure 4 shows an example of incremental fitness evaluation using only the child's parents. Since it does not use the child's code, it can determine exactly the child's fitness before the child is created. If it turns out the child is never used, e.g. because it is unlucky, it need never be created.

We assume the GP population is made of pure functions (i.e. there are no side effects) and the same test cases are used to assign fitness of the children as were used to find the fitness of their parents. Our EuroGP 2021 paper [2, Tab. 1] gives details of the GP. The C++ code ensures that it produces identical results with less overhead.

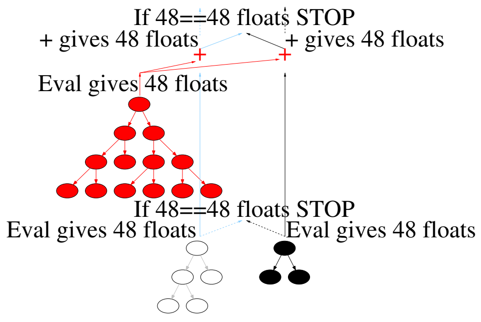

Figure 4: "Fitness first" begins by evaluating the subtree to be removed from the mum (white) and the subtree to be inserted (black). It proceeds up the mum's tree until either the evaluation in the mum and unborn child are the same or it reaches the root node. The red subtree is in the mum but it is identical to the code in its child and so need be evaluated only once per test case. Note the code from the parents is evaluated without creating the child.

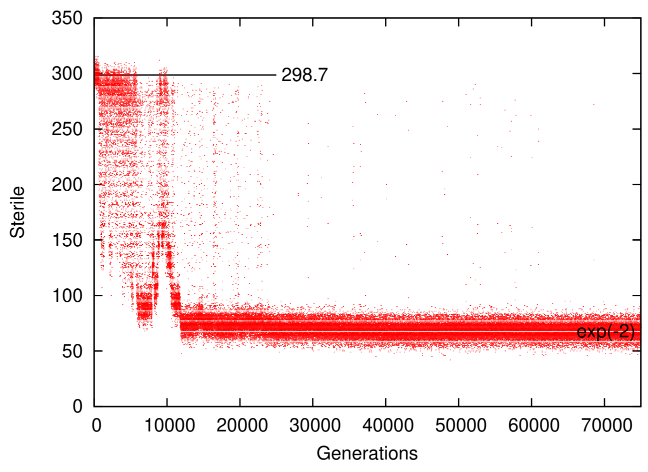

Figure 5: Evolution of number in population without children in next generation. 100% two parent crossover, 7-tournament, pop=500.

2 AVOIDING RUNTS WASTING EFFORT

Early in GP runs at each generation there are many poor individuals which need not be created (see Figure 5).

In the limit of large converged populations (containing M individuals) on average there will be $e^{-2}M$ individuals which are never selected to have children (see right hand side of Figure 5). If we consider just the first parent in crossover, then this rises to $e^{-1}M$.

As Figure 5 shows, delaying crossover until after fitness selection can save creating more than half the population during the early part of a run. Even later, when convergence ensures almost the whole population has the same fitness, 14% (e^{-2}) of the population need not be created.

3 LAST CHILD & FATHERLESS CROSSOVER

Initially the populations are very variable and, with strong selection, breeding is concentrated in a few fit parents. As the populations starts to converge, there are more parents (with fewer children each). In each generation, as each child is created, eventually for each parent, there is only one child left to be created. On reaching the last child for a root donating parent, instead of copying the code into the child, the buffer holding the parent's genome is unhooked from the parent and passed to the child (see Figure 2). This saves copying the first part of the child.

In bloated populations, the second parent (dad) donates only a tiny fraction of the opcodes in the child. Therefore we extract and save on the C++ heap all the subtrees which will be inserted later. This is relatively cheap and is done before the bulk of the crossover operations are done using the root donating parents (mums). This allows the mum's last child crossover short cut to be used about twice as often.

Notice whilst fitness convergence reduces the number of childless members of the population, here it helps: as spreading the breeding effort, means there are more parents in general, and thus more cases where a mum has only one child left to be created. That is, convergence increases the number of times the inplace crossover optimisation can be applied. As the population converges and there are more parents with children, the number of inplace crossovers rises, so that on average 268.1 ($\lesssim M(1-e^{-2})(1-e^{-1})$) crossovers are done inplace per generation.

In about one third (28.9%) of cases, the removed and inserted subtrees are the same size. If so, the mum's buffer can be simply over written with the inserted code (from the dad). However most (71.1%) of the time they are not the same size and on average half the buffer must be shuffled either up or down to take account of the difference in the subtree sizes (see Figure 3). By excluding dads from crossover, we can use the inplace short cut >50% of the time.

After crossover has been optimised to: 1) ignore individuals which will not have children (saving about 13.5%) and 2) where possible, modifying chromosomes inplace, we reduce the opcodes copied by crossover by 48.1%. This leads to a reduction in the time taken by crossover by about a quarter (24.4%).

The fitness results are of course identical.

Acknowledgments

This work was inspired by conversations at Dagstuhl Seminar 18052 on Genetic Improvement of Software.

Funded by EPSRC grant EP/P005888/1.

The new GPQuick code is available in http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/GPinc.tar.gz

REFERENCES

[1] W. B. Langdon. 2020. Multi-threaded Memory Efficient Crossover in C++ for Generational Genetic Programming. *SIGEVOLution newsletter of the ACM Special Interest Group on Genetic and Evolutionary Computation* 13, 3 (Oct. 2020), 2–4. http://dx.doi.org/10.1145/3430913.3430914

[2] William B. Langdon. 2021. Incremental Evaluation in Genetic Programming. In *EuroGP 2021: Proceedings of the 24th European Conference on Genetic Programming (LNCS, Vol. 12691)*, Ting Hu et al. (Eds.). Springer Verlag, Virtual Event, 229–246. http://dx.doi.org/10.1007/978-3-030-72812-0_15

[3] William B. Langdon and Wolfgang Banzhaf. 2019. Continuous Long-Term Evolution of Genetic Programming. In *Conference on Artificial Life (ALIFE 2019)*, Rudolf Fuechslin (Ed.). MIT Press, Newcastle, 388–395. http://dx.doi.org/10.1162/isal_a_00191