Lineage Selection in Mixed Populations for Genetic Improvement

Penny Faulkner Rainford and Barry Porter
School of Computing and Communications, Lancaster University
faulknerrainford@gmail.com; b.f.porter@lancaster.ac.uk

Abstract

Emergent Software Systems take a large pool of potential building blocks, for a given system such as a web server, and learn at runtime how best to compose selected blocks from that pool in order to maximise some utility function in each set of deployment conditions that is encountered. To support this approach, at least some building blocks in the available pool must have *implementation variants* – alternatives which have the same functionality but achieve it using a different approach (such as different sorting algorithms or different cache eviction policies). We can automatically derive new building block variants for our pool of potential behaviour by using genetic improvement (GI), which has long proven effective for optimisation and repair of source code. When a novel deployment environment is detected, however, it is unclear which existing building block variant(s) should be used as starting points for new a GI process to tailor a new block for that environment; in this situation it would be necessary to try one GI process from every possible existing building block variant as a starting point, a process which could be extremely expensive. In this paper we present a mixed-population approach to examine whether GI can simultaneously offer both *lineage selection* and *optimisation* to find the ideal source code for a new building block variant tailored to a given environment. Using a lowest-common-ancestor approach to producing evolvable individuals, our results demonstrate strong evidence that combined lineage selection and optimisation is viable in multiple scenarios, offering far reduced compute time to locate a good individual for a novel environment.

Introduction

Genetic algorithms have long been used across many different computing applications, from finding the ideal parameters of systems with large parameter search-spaces (Bäck and Schwefel, 1993), to a useful meta-heuristic approach to fixing bugs in code (Haraldsson et al., 2017; Forrest et al., 2009). In this paper we focus on genetic code improvement (GI), which aims to derive new versions of existing computation logic that are optimised towards a utility function such as calls-per-second. GI has been approached in a wide variety of ways, from modifying bytecode to working on syntax trees or with grammar models of the target programming language (Petke et al., 2018). We use an approach based on the compiler-derived syntax tree of a piece of source code, augmented with grammar rules to guide valid mutations. We particularly target our GI approach towards emergent software systems, which are composed of many small pieces of interchangeable building blocks such as a sorting algorithm or a hash table (Porter et al., 2016). At runtime, emergent software systems monitor their environment and learn which variants of each building block are best suited to each set of deployment conditions encountered. Because each building block in these systems is relatively small (100-200 lines of code), our particular GI approach mixes new code synthesis with more traditional mutation types so that sufficient new genetic material is available (Rainford and Porter, 2021).

When an emergent software system detects a novel environment, it will learn the best composition of building blocks from among those which currently exist; it is also useful, however, to seek to generate new variants of building blocks that are tailored to that novel environment using GI. Using a traditional GI approach, when we have several existing variants of the same building block (such as a cache eviction policy), it is unclear which variant we might use as the starting point for a new GI run to derive a tailored variant for the novel environment. A naive approach may use a set of distinct GI runs, each starting from one existing variant and running for e.g. 200 generations, to find the best tailored variant considering each of the existing variants. This approach requires significant computation power, however, which scales poorly as more variants are added to the pool.

In this paper we examine whether a GI process, operating on source code, can perform both *lineage selection* and *optimisation* at the same time in a single run for a novel environment: i.e., can we add each existing variant to a common starting population, and run a single GI process from that population for 200 generations, to see an equivalent fitness individual emerging for our novel environment – when compared to a set of per-variant GI processes each running for an equal number of generations. Demonstrating that this is possible offers a far reduced volume of GI computation for each novel environment that is encountered.

Our use of lineage selection takes inspiration from the biological analogue: when different genes provide for the

same functionality, selection can be seen at both an individual and a lineage level. Selection at a lineage level means that, because of inheritance, a favoured individual's offspring will share the same advantages and so will their children, so the whole lineage (family tree) will be selected for (Akçay and Van Cleve, 2016). In GI this would mean that if we create a population with members from different algorithms (lineages), and evolve over the whole population, the lineage with the better algorithm for the given environment will be favoured by selection and dominate the population.

We specifically propose the use of *lowest common ancestors* (LCAs) of previous runs to support lineage selection. This takes an individual from a previous run that was the ancestor of all the individuals remaining in the final population. We take these LCAs from two different runs which were specialised for two different deployment environments A and B. We then test to see if a starting population containing copies of both LCA individuals, run against each deployment environment A and B, can correctly select the appropriate lineage for each problem and then optimise that individual to the same level as the best fit individual of the LCA's originating GI run. This approach has the potential to automatically select and optimize the ideal base method for a genetic improver's target function from a set of possibilities. We also examine the form of LCA that offers highest utility as a starting point for future GI runs: the 'full' LCA, as it appeared in its original GI run, or a 'reduced' LCA which has had all extraneous source code removed.

We show that the reduced LCA has increased evolvabilty, versus the full LCA, as a start point for genetic improvement, due to the increased chance of mutations editing code which effects the fitness; this gives more variation in end-results of a new GI run than the full LCAs, which remain mostly homogeneous. We show that reduced LCAs are not more evolvable than final individuals due to early specialisation. Using these reduced LCAs we then show that a mixed population of these LCAs can successfully reach the same level of training as the original best-fit individual from the full GI run of the corresponding LCA – and that this mixed population does indeed perform both lineage selection and optimisation within 200 generations. We provide a replication package, with detailed instructions, with which all of our results can be repeated (Rainford and Porter, 2022).

Related Work

Non-homogeneous starting populations are not new in genetic algorithms. It is a common for genetic algorithms used for parameter optimization to be seeded with individuals from across the possible parameter settings to better cover the fitness landscape (Bäck and Schwefel, 1993). This allows the system to use selection to focus its evolution on areas of the landscape with high potential fitness. GI for code optimization has, by nature of the wish to optimize a particular function, started from a single start point (Forrest et al., 2009; Haraldsson et al., 2017). This is either the non-functional code to be repaired or a functional but non-optimal code to be improved. In the case of repair it is sensible to start with the code to be repaired rather than anything else. However in the case of optimization, multi-start points become an advantage in possibly providing a broader start to searching the fitness landscape allowing the system to focus in on the area of the landscape with higher potential.

In biology this selection of a particular area over another is called *lineage selection*. It has been explored in a digital context previously in relation to different artificial organisms and evolutionary systems (Dolson et al., 2020; Virgo et al., 2017). It is considered as part of analysis for evolvability (Kirschner and Gerhart, 1998). This is a natural connection. The existence of multiple lineages in an evolutionary system should produce higher robustness and variety in the system than a single lineage with a single start point. In most GI systems however, the focus is less on variety and robustness, and instead on finding the best optimisation to the immediate environment. In this work we test if multiple lineages detract from finding the best optimisation, or if they provide the same optimisation with greater evolutionary potential.

Algorithm

Our overall GI framework is illustrated in Fig. 1. An emergent software system is assembled from a large collection of small building blocks, such as stream processors, memory cache implementations, hash tables, and so on, and is deployed into a real environment. Once that system has learned the best composition of blocks for a given environment, it will select one of those building blocks and capture a short trace of the method calls that are issued to that block within the present environment. This trace is then sent to a GI system, along with the source code of the building block from which it was captured, so that the GI system can attempt to generate an improved variation of that building block which has higher performance for the given input data sample. If the GI system is successful, the improved building block is pushed back to the emergent software system which uses real-time learning to determine if the proposed improvement really does yield higher performance for the intended environment conditions in deployment.

For the purposes of this study we examine only the GI element of this overall concept. We focus on one particular building block throughout our experiments (a hash table), and we assume that short traces of function calls to this block have already been captured by the emergent system. We also assume that the GI process is able to identify the *hash function* of the hash table implementation as the specific area in which to focus when generating improved variations; in practice this focus area could be determined using function call frequency or CPU intensity analysis.

The overall core of our system is then based on a typical genetic improvement process, Algorithm 1, using mutation,

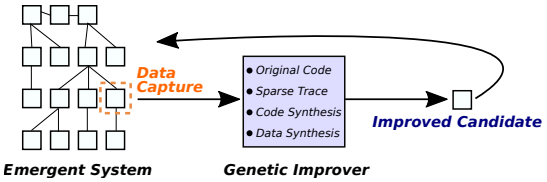

Figure 1: Our approach, which captures input data from a running emergent software system, and requests improved variants of particular building blocks from a GI system by replaying that captured data. The GI process uses mixed synthesis of code with traditional GI operators, to counter the small code sizes of each building block.

Algorithm 1 Genetic Improvement Algorithm

```
for i = 0 to generations do
  if i == 0 then
    create initial population of clones
  end if
  mutate a % of the population
  check fitness of all population members
  if i%5 == 0 then
    check performance on unseen data of all population
    members
  end if
  select new population (roulette wheel)
  crossover a % of the population
end for
```

fitness, selection, and crossover. We include crossover in this work to make use of existing code in expanding the code base of members of our population. Because our volume of starting genetic material is very small, we also include *new code synthesis* in our set of available mutations, combined with more typical mutation types.

Our selection process, where we choose which individuals to select from one generation for inclusion in the next generation (with associated crossover/mutation), uses a rank-weighted roulette wheel approach. Each population is ordered by fitness and then ranked. This rank is then used such that the fittest individuals have the highest probability of selection for the next generation. Selection is done with replacement, so that some individuals will appear multiple times in the following generation, and some will be completely absent, with a (small) possibility of even the worst individual being selected for the next generation.

A hash function (e.g. Listing 1) takes a set of key/value pairs, where a key is a string of characters, and uses a mathematical transformation of the key string to yield an integer result (such as adding together the binary representations of each string characeter). The integer result is used as an index value to place the key into a particular hash bucket, where the list of hash buckets is usually represented as a fixed-

```
1 int hash(char key[]) {
2 int result = 1
3 for (int i = 0; i < key.arrayLength; i++)
4 {
5 result = result * key[i]
6 }
7 return result % HT_LEN
8 }
```

Listing 1: Original Hash Function

length array. Each array cell has a linked list of all keys that have been placed into that array cell / bucket. When retrieving a key from a hash table, the hash function is applied to the key to derive the correct bucket, and the linked list of keys in that bucket is scanned iteratively to locate the matching key. The general objective of a hash function is to divide the set of keys it is given evenly between each hash bucket, so that the linked list within each hash bucket is the same length, thereby minimising average lookup time for a given key (compared to a situation, for example, in which every key mapped to hash bucket index 0, potentially necessitating a scan over every single key in the hash table).

Throughout this study we use two different environments for our hash function: one set of keys derived from English words, and one set of keys derived from Polish words. These two key sets are sufficiently different that they suggest very different hash functions to gain an even distribution of keys across buckets. In the final part of our study we use a third, novel environment, with keys based on French words.

Lowest Common Ancestor

When considering the results of an existing GI run, and the choice of an individual from that run to inject into a new GI run training to a novel environment, we hypothesise that using the best-fit individual from the final generation of an existing run is likely to be a poor choice. This individual is likely to be highly specialised to its own environment (e.g. the set of English keys) and may take longer to evolve to a novel environment. Instead we look further up the phylogenetic tree, shown in Figure 2, for an earlier less-specialised but still somewhat optimised individual: the lowest common ancestor (LCA) of a genetic improver's final population. The lower trees in Figure 2, for example, show the LCAs up to the final population for GI runs on English and Polish environments. The LCA is the most recent ancestor which is common to every member of the final population. The LCA may not have been the fittest individual of its generation, and indeed may not be a particularly fit individual at all, but evidently was an individual with high evolutionary potential shown by its ability to produce the offspring that lead to the final (and fittest) population. We assume that an LCA has higher evolvability to novel environments, when placed into a new GI process, than a final best-fit individual.

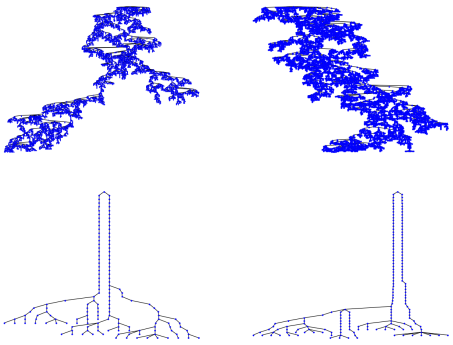

Figure 2: Phylogenetic trees for two runs of the genetic improver trained on different data: English (left) and Polish (right). The full tree (top) is busy so we also show the tree below the lower common ancestor (bottom).

While this assumption seems reasonable, there are two remaining sources of uncertainty in this approach. First, if we start a new GI process from a population of LCA copies, do we observe within that process a return to equivalent fitness compared to the best-fit final-generation individual from the LCA's original GI run. In other words, having removed all the additional genetic material that was available from the other members of the LCA's original population, is there sufficient material left to yield an equivalent fitness result.

Second, the precise form of LCA to use is uncertain. Here we have two options: we can use the 'full' LCA source code, exactly as it appeared in its original GI run. This has a significant amount of 'redundant' genetic material which does not contribute to the functional logic of the LCA. Alternatively, we can use a 'reduced' LCA which has all of its redundant genetic material removed. The trade off we expect here is that the full LCA has more genetic material to offer and potentially more diversity, but that mutations are less likely to affect the fitness of an individual (since each mutation has a higher chance of affecting redundant code); while the reduced LCA has less diversity of genetic material to offer, but mutations are more likely to affect the fitness of individuals since all source code contributes to functionality.

Experiments

We begin with two control runs of our GI system trained on each key set: the set of English keys, and the set of Polish keys. Both control runs start with a population made entirely of a single piece of source code (a hand-crafted hash function that was optimised for general speed of execution rather than key-distribution effectiveness, shown in Listing 1).

Using these two control runs, we extract their LCAs, and derive both the full LCA and reduced LCA (English reduced LCA, Listing 2, and Polish reduced LCA, Listing 3). We then perform new GI runs that start from populations comprised entirely of copies of the full or reduced LCA, yielding

```
int hash(char key[]) {
int result = 1
for (int i = 0; i < key.arrayLength; i++)
{
result = result * key[i]
result = result - key.arrayLength
result = result * key[i]
}
return result % HT_LEN
}
```

Listing 2: Reduced English LCA

```
int hash(char key[]) {
int result = 1
dec a = 0.8653064475373070635
for (int i = 0; i < key.arrayLength; i++)
{
result = result * key[i]
a = 0.8653064475373070635
result = key.arrayLength + key.arrayLength
}
return result % HT_LEN
}
```

Listing 3: Reduced Polish LCA

4 GI runs in total – two for English LCAs training on the English key set, and two for Polish LCAs on the Polish key set.

We perform 30 GI runs for each case, and compare with the original control runs using the original hand-crafted start code. These runs are analysed in terms of relative improvement when compared with the control runs. The relative improvement is important to understanding the ability of the system to still train and specialise from the LCAs.

Results

In general, when using the LCAs, we do not expect much training to occur as the LCAs are relatively well optimised. In Figure 3 we see the fitness relative to the original source code for both English language data (left column) and Polish language data (right column). All of our GI graphs show the average fitness of the best individual of each generation, and the standard deviation, for 30 repeated runs.

For the English environment we can see there is no training in the full LCA run, but the reduced LCA run does become varied including up to 5% improvement after 100 generations. The reduced LCA might be more likely to make changes to the critical path than the full LCA. In the context of the original code, these results show that the LCA selected was from a successful run, as both LCA experiments start with better fitness than the average of the control run, and that the reduced LCA run actually attains slightly better fitness than the control run. This offers confirmation that using reduced LCAs is the better choice, and that using these LCAs can yield equivalent fitness compared to the original

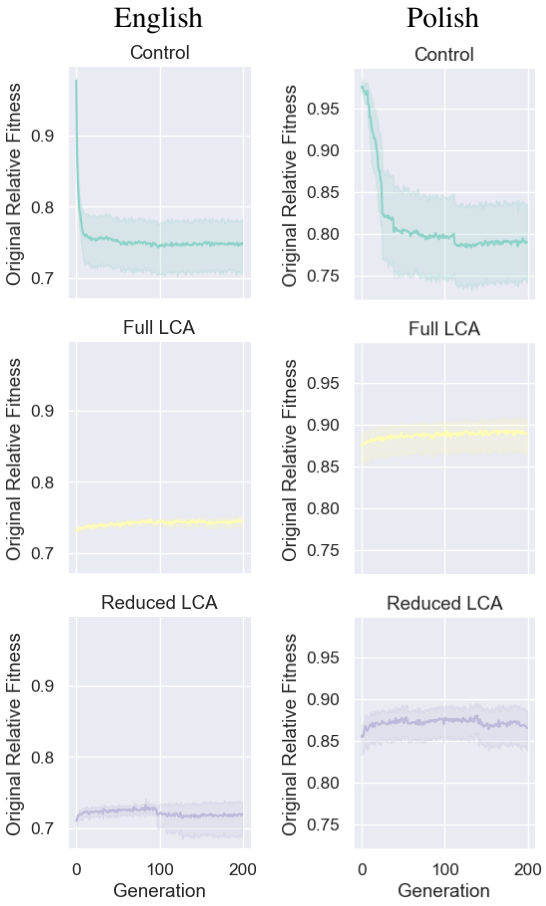

Figure 3: Relative fitness of experiments with English (left column) and Polish (right) data, showing: Control (Top), Full LCA (Middle) and Reduced LCA (Bottom).

run despite having far less genetic material overall.

Examining the Polish data runs, shown in the right column of Figure 3, we see similar trends. There is very little variation or training in the full LCA experiment, but clear evidence of training in the reduced LCA run towards generation 150. The LCA-derived training is generally much weaker in the Polish environment than the English; this is possibly explained when we consider that the LCA run started with relatively poor fitness, suggesting that the LCA in the control run was more reliant on other genetic material in its population compared to the English environment.

Overall the data here clearly indicates that reduced LCAs have higher utility, and the use of LCAs in general is promising as start points for new GI runs for novel environments.

Meta-populations

Having established the utility of LCAs as evolvable starting points, in this section we examine the ability of a GI process to simultaneously perform *lineage selection* and *optimisation* within 200 generations. This demonstrates the major saving in computation cost when deriving a new variant for a novel environment, versus running a set of individual GI processes on each starting point.

In these experiments, instead of cloning a single piece of code to form our initial population, we divide our population equally into clones of each of our LCA candidates (in this case 15 clones of our English LCA, and 15 of our Polish LCA). These are assigned to lineages based on which piece of code their first ancestor was cloned from. We do not otherwise separate the populations or make them known to the GI process. The biological analogy here would be if one took equal numbers of individuals from meta-populations and put them together to form a single new population.

To explore this we use the end-result of our two existing single-lineage GI runs – on English and Polish key sets, which start from their respective reduced LCAs – as our ground truth. When our mixed-lineage LCA-based starting point run is provided with our English key training set, it should ideally be able to both select the English lineage to become dominant in the population, and simultaneously optimise that population from its LCA starting point to a similar level to that seen in the final generation of our single-lineage English key training set. Demonstrating this would potentially allow us to train over many LCA-based start points with very little increase in resources used, versus training independently from a set of single start points.

Validation Experiment

We execute our mixed-population LCA experiment, in which half of the starting population is from our English LCA and half is from our Polish LCA, separately against the English training data and the Polish training data. Each run has 200 generations, with a population size of 30, and we repeat each run 30 times to record an average.

We compare the results against the single-lineage LCA runs for both English and Polish training data, which again used 200 generations and a population size of 30. We examine the results of the first two training set-ups first to ensure they match the better of the two single-lineage runs trained of the same data, and second to study the distribution of lineages in the population over time to see if one lineage dominates the other and if so how long it takes for this to occur.

Validation Results

Figure 4 shows the results when training against our English key set. The left column of graphs shows fitness against the training data, while the right column of graphs shows performance against unseen data (which is drawn from the same distribution as the training data). Fitness indicates how well the algorithm is specialising, while performance shows how well it is generalising to the *class* of data on which it is being trained. The top row of graphs in Figure 4 are when we start with the single-lineage English LCA, the bottom two graphs show a start point of the single-lineage Polish LCA, and the middle two show our mixed-lineage experiment.

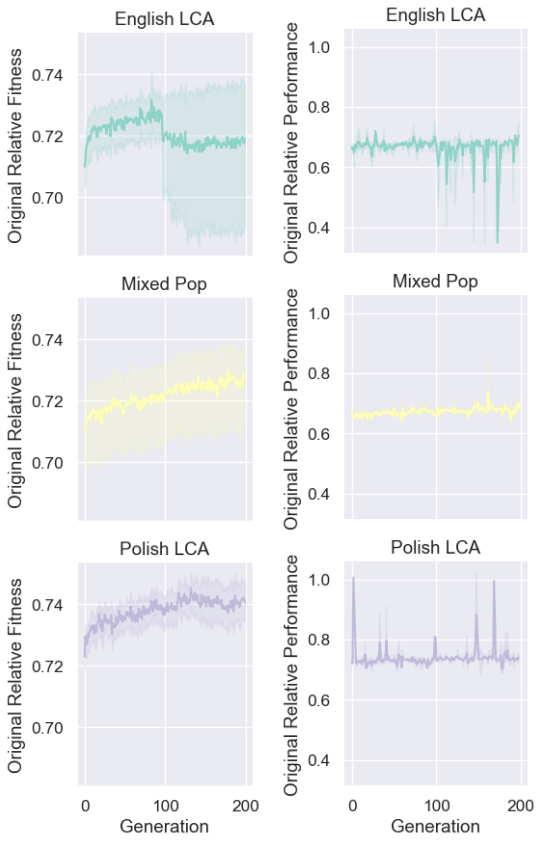

Figure 4: Fitness (left) and performance (right) for systems trained on English data: Control (top) using English-trained reduced-LCA, Mixed population (mid) using both reduced LCAs, and the Polish-trained reduced LCA (bottom).

Starting with fitness, the single-lineage English-trained LCA (top) shows the expected improvement as discussed earlier in the paper. The single-lineage Polish-trained LCA (bottom), when trained against English data, likewise shows an expected poor fitness training towards the alternative data set. When we examine our mixed-lineage experiment (middle), we observe a fitness score in the final generation which comes very close to the final fitness of the single-lineage English LCA comparison. This suggests that the mixed-lineage GI run experiences both lineage selection and training within the same number of generations. When we examine relative performance against unseen data of the best individuals, in the right column of graphs, we see that our mixed-lineage experiment performs as well as than the single-lineage English comparison with improvements of 30.2% and 32.6% respectively and no statistically significant difference in the final populations (comparison of best individuals with signed-rand Wilcoxon test, 5% significance); this may be as the benefit of the diversity in genetic material at the beginning of the run outweighs the smaller number of individuals of the correct lineage in the starting population.

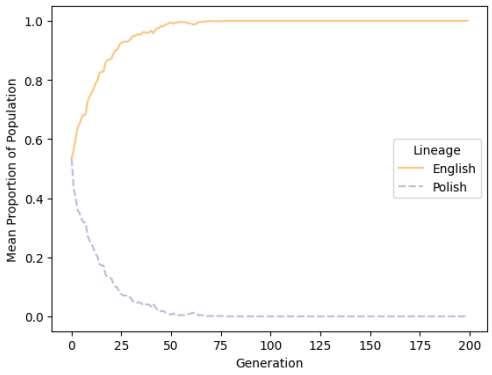

Figure 5: Proportion of population at each generation of each lineage for all runs of the Mixed Population against English training data.

We also examine lineage selection in isolation, with the results shown in Figure 5. This confirms our above data, showing that the English LCA lineage on average occupies more than 80% of the population after 14 generations, with the Polish LCA linage extinct after 77 generations. We can therefore say that the first 77 generations is spent doing a mixture of lineage selection and training (including potential exchange of genetic material between the lineage members), with the remaining generations exclusively doing training.

We next examine the same set of starting populations (English, Polish, and mixed-lineage) when trained on our Polish hash key data set. The Polish LCA was a weaker starting candidate as discussed in the previous section, however in Figure 6 we see that it still performs far better than the English LCA when trained on Polish data. On fitness we again see that our mixed-lineage experiment more closely tracks the single-lineage Polish comparison, though it diverges more than in our English target experiment. Examining performance, in the column of graphs on the right, the single-lineage Polish LCA has an average improvement in performance on unseen data of 13.4%, while the English LCA only has a mean improvement of 1.8%. Here our mixed-lineage experiment does closely match the control performance (Polish LCA) with an average performance of 12.2% (with our tests indicating no statistical difference in either performance or training fitness in the later generations between the Polish LCA and the mixed population).

Turning again to lineage selection, shown in Figure 7, here we see an even more pronounced effect; the Polish LCA lineage represents more than 80% the population after just 15 generations and drives the English LCA lineage into extinction after only 60 generations.

New Environment Experiment

Having confirmed that lineage selection is possible and correct when we know which lineage will perform better, we

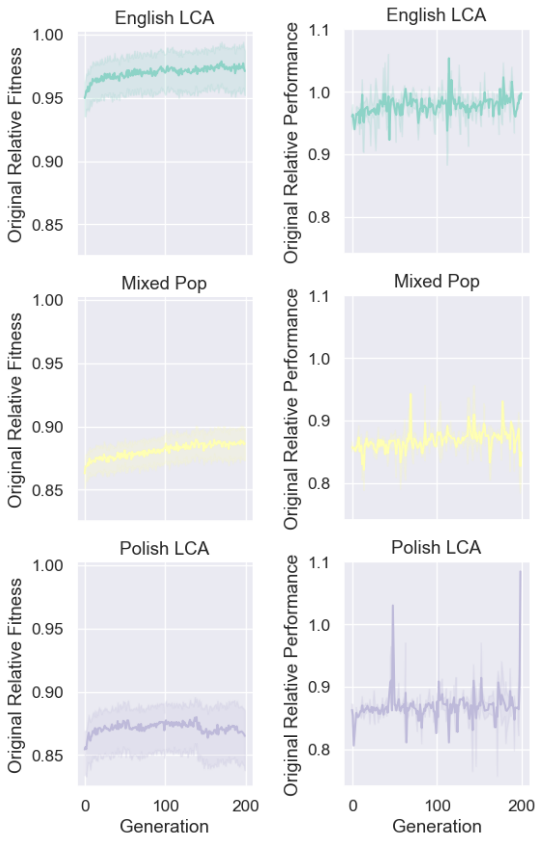

Figure 6: Fitness (left) and performance (right) for systems trained on Polish data: the English-trained reduced LCA (top), Mixed population (mid) using both reduced LCAs, and the Polish-trained reduced LCA (bottom).

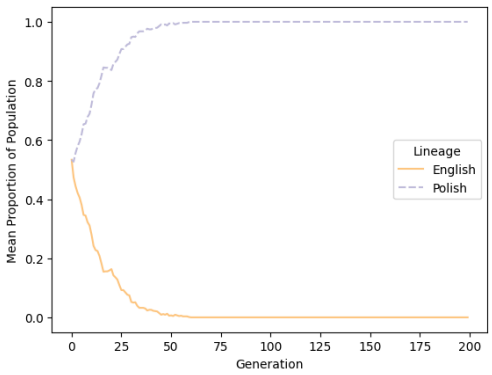

Figure 7: Proportion of population at each generation of each lineage for all runs of the Mixed Population against Polish training data.

now examine whether we see equivalent results in a novel environment. To do this we execute our mixed-population LCA experiment, in which half of the starting population is from our English LCA and half is from our Polish LCA, against a set of hash keys based on French words. Each run has 200 generations, with a population size of 30, and we repeat each run 30 times to record an average.

We also in this case test the use of the best final individual for each run that produced the reduced LCAs used in previous experiments; we do this to re-verify that the use of LCAs has value in yielding individuals with higher evolvability.

We compare the best individual, reduced LCA, and mixed-lineage runs on fitness and performance for unseen (French) data. To meet our requirements the mixed-lineage run should have equivalent final population performance and fitness to the best of the other runs; we will also examine whether, and how quickly, lineage selection occurred.

New Environment Results

Figure 8 shows the results of these experiments. Addressing trained fitness (on the left) first we can see that the English trained specialist (both finalist and LCA, top 2 graphs) have less than 20% improvement on the original code, while the mixed-lineage and Polish specialists all have greater than 25% improvement. The mean improvement in the final generations for the mixed-lineage (27.3%), Polish Finalist (27.3%) and Polish LCA (26.8%) are very similar and there is no statistical difference in their final generation.

Examining performance on unseen data, in the column of graphs on the right, shows closer results but still demonstrates that Polish and multi-lineage populations still perform statistically-significantly better than the English populations. The English Finalist and English LCA achieve performance improvements of 18.9% and 19.3% respectively, while the mixed-lineage, Polish Finalist and Polish LCA achieve 20.5%, 20.9%, and 20.5% improvements with no statistical difference again in their final generation.

The implication that lineage selection has successfully chosen and optimised to the Polish lineage is confirmed in Figure 9 which shows that the Polish lineage represents more than 80% of the population after just 12 generations and the English lineage has been driven to extinction after only 50 generations. Overall these results confirm that both lineage selection and optimisation is possible in a single GI run, and can save on significant computation time when presented with a novel environment for which a new source code variant is required.

Discussion

Our set of experiments demonstrate that the use of a lowest-common-ancestor has good utility as an evolvable starting point (and one which may not yet have been over-specialised). However, the lack of very significant improvements in training from the mixed-lineage runs suggests that

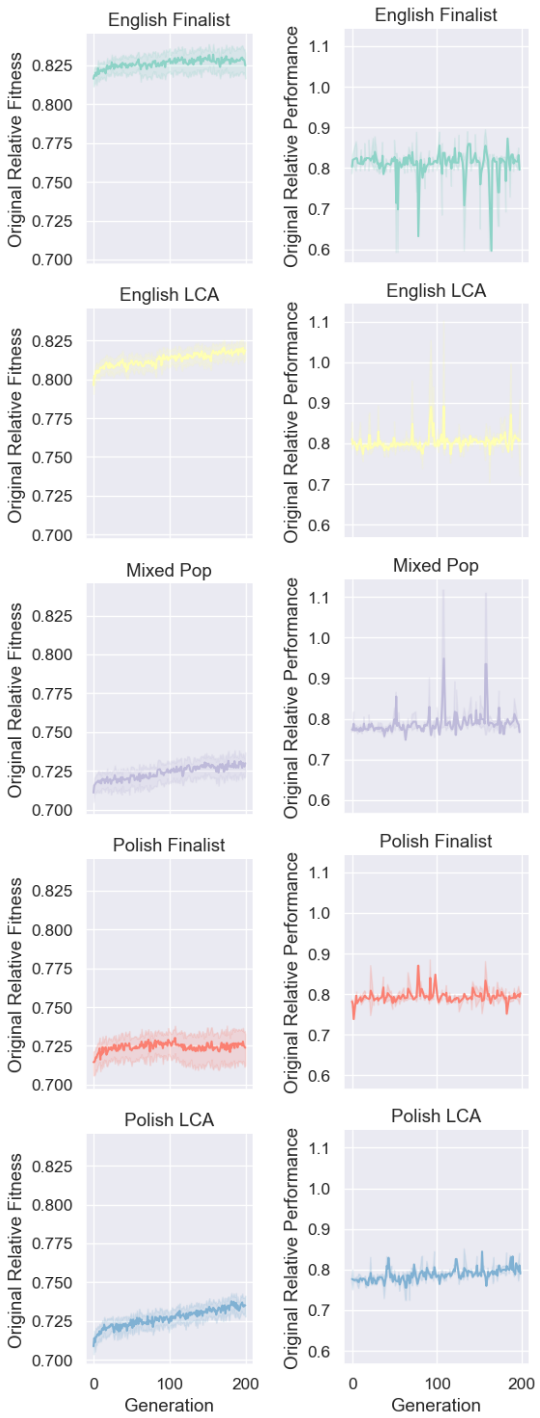

Figure 8: Fitness (left) and performance on unseen data (right) for systems trained on French data: the best English trained individual (top), the reduced English LCA (top mid) , Mixed population (mid) using both reduced LCAs, the best Polish trained individual (bottom mid) and the reduced Polish trained LCA (bottom) for comparison.

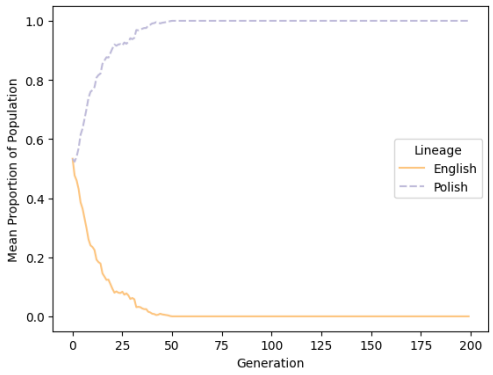

Figure 9: Proportion of population at each generation of each lineage for all runs of the Mixed Population against French training data.

our LCAs are still relatively well specialised – and that taking earlier ancestors may be beneficial.

Our lineage selection experiments show a very clear result, that a single GI run starting from a mixed population quickly selects the most ideal lineage for its training data, and that it maintains potential to perform further training towards new environments at the same time. While we have used two lineages in this study, an interesting question for future work is how many different lineages can be used in a starting population – without changing the population size – while maintaining clear lineage selection and training, and potentially gaining the benefit of higher genetic diversity.

Conclusions

We have shown here that lineage selection using lowest common ancestors can be used to select for the correct variant code for deployment conditions. The use of lowest common ancestors has successfully provided two specialist variants as starting points for training towards different deployment conditions. In future we will investigate the use of higher common ancestors for increased training potential and evolvability. We have shown that in the same population size and run length (and so the same computational resources) our genetic improvement algorithm can quickly select the correct lineage for the deployment conditions while also making use of the increased genetic material to improve the selected lineage resulting in the same overall improvements as the single source code alternatives. Further investigations into more divergent code in multiple populations and the use of Meta-populations with more separation in crossover and selection could yield further improvements.

Acknowledgements

This work was partly supported by the Leverhulme Trust Research Grant 'The Emergent Data Centre', RPG-2017-166.

References

Akçay, E. and Van Cleve, J. (2016). There is no fitness but fitness, and the lineage is its bearer. *Philos. Trans. R. Soc. Lond. B Biol. Sci.*, 371(1687):20150085.

Bäck, T. and Schwefel, H.-P. (1993). An overview of evolutionary algorithms for parameter optimization. *Evol. Comput.*, 1(1):1–23.

Dolson, E., Lalejini, A., Jorgensen, S., and Ofria, C. (2020). Interpreting the tape of life: Ancestry-based analyses provide insights and intuition about evolutionary dynamics. *Artif. Life*, 26(1):58–79.

Forrest, S., Nguyen, T., Weimer, W., and Le Goues, C. (2009). A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, pages 947–954, New York, NY, USA. Association for Computing Machinery.

Haraldsson, S. O., Woodward, J. R., Brownlee, A. E. I., and Siggeirsdottir, K. (2017). Fixing bugs in your sleep: how genetic improvement became an overnight success. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, pages 1513–1520, New York, NY, USA. Association for Computing Machinery.

Kirschner, M. and Gerhart, J. (1998). Evolvability. *Proc. Natl. Acad. Sci. U. S. A.*, 95(15):8420–8427.

Petke, J., Haraldsson, S. O., Harman, M., Langdon, W. B., White, D. R., and Woodward, J. R. (2018). Genetic improvement of software: A comprehensive survey. *IEEE Trans. Evol. Comput.*, 22(3):415–432.

Porter, B., Grieves, M., Rodrigues Filho, R., and Leslie, D. (2016). {REX}: A development platform and online learning approach for runtime emergent software systems. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 333–348. usenix.org.

Rainford, F. P. and Porter, B. (2022). ALife 2022 replication package: http://www.projectdana.com/research/alife2022rainford.

Rainford, P. F. and Porter, B. (2021). Open challenges in genetic improvement for emergent software systems. *geneticimprovementofsoftware.com*.

Virgo, N., Agmon, E., and Fernando, C. (2017). Lineage selection leads to evolvability at large population sizes. In *Artificial Life Conference Proceedings 14*, pages 420–427. MIT Press.