Code and Data Synthesis for Genetic Improvement in Emergent Software Systems

PENNY FAULKNER RAINFORD, School of Computing and Communications, Lancaster University, UK
BARRY PORTER, School of Computing and Communications, Lancaster University, UK

Emergent software systems are assembled from a collection of small code blocks, where some of those blocks have alternative implementation variants; they optimise at run-time by learning which compositions of alternative blocks best suit each deployment environment encountered.

In this paper we study the *automated* synthesis of new implementation variants for a running system using genetic improvement (GI). Typical GI approaches, however, rely on large amounts of data for accurate training and large code bases from which to source genetic material. In emergent systems we have neither asset, with sparsely sampled runtime data and small code volumes in each building block.

We therefore examine two approaches to more effective GI under these constraints: the synthesis of *data* from sparse samples to construct statistically representative larger training corpora; and the synthesis of *code* to counter the relative lack of genetic material in our starting population members.

Our results demonstrate that a mixture of synthesised and existing code is a viable optimisation strategy, and that phases of increased synthesis can make GI more robust to deleterious mutations. On synthesised data, we find that we can produce equivalent optimisation compared to GI methods using larger data sets, and that this optimisation can produce both useful specialists and generalists.

CCS Concepts: • **Computing methodologies** → **Genetic algorithms**; Artificial life.

Additional Key Words and Phrases: genetic improvement, optimization, emergent systems, data synthesis, data sampling, fitness function, language

1 INTRODUCTION

Emergent software systems are assembled from a large collection of small code blocks, where some of those blocks have alternative implementation variants available [Porter 2014; Porter et al. 2016]. Emergent systems optimise at run-time by learning which compositions of alternative building blocks are best suited to each set of deployment conditions encountered. In order to do this, at least some building blocks must have available implementation variants that are likely to be useful in different environments – such as different cache replacement algorithms or buffer management strategies. The more useful variation points there are available, the more likely it is that a system can closely optimise to its current deployment conditions.

At present the creation of alternative building blocks for emergent software systems – or any self-adaptive system based on component hot-swapping – is manual, requiring non-trivial programmer effort to design and develop variations that are likely to be useful in the different deployment conditions a system may encounter. This process is challenging firstly because it takes effort to create multiple correct versions of the same high-level functionality, but also because creating useful variants for a given deployment entails significant uncertainty for the developer – it is difficult to guess at design-time variations for each building block are likely to be of high utility. In this paper we study how the creation of such high-utility variations may be *automated*, based on the real-time observed experiences of a deployed system. In an emergent web server, for example, the input sequences issued to a memory cache component may be captured over a short time window for offline replay – to then derive a more effective cache eviction policy for a particular request sequence that typifies a given deployment environment experienced by the web server.

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

The most common approach to automated enhancement of source code is *genetic improvement*, which uses an evolutionary algorithm inspired by biological evolution – as Darwin described it: "descent with modification" [Darwin 1859]. Evolutionary algorithms have been studied for decades, and used for a wide range of purposes from data mining to parameter optimization [Bäck and Schwefel 1993; Freitas 2009]; in many cases they have proven able to find more efficient and sometimes unexpected solutions to problems such as antenna design [Torresen 2002].

In existing research, genetic improvement (GI) for source code is most widely applied for bug-fixing, in which a bug is discovered in a large code base and a GI process is used to mutate that code base to a state in which the bug is corrected [Arcuri and Xin Yao 2008; Forrest et al. 2009; Sidiroglou-Douskos et al. 2015]. GI for bug-fixing relies on the well-established hypothesis that in a large code base the fix for a given bug is likely to exist elsewhere in the same code base [Barr et al. 2014]. In this context mutations – and particularly crossover – draw on existing genetic material elsewhere in the code base being improved. GI has been applied to a lesser extent for performance optimisation of source code, which we expand on in Sec. 3.

In the context of emergent software systems we have two specific challenges that go beyond common GI solutions. The first is that genetic material is limited: building blocks in emergent systems are typically 100-200 lines of code, offering little to draw on for a GI process. In order to improve the chance of finding improved versions of a given building block we therefore need a way to *synthesise* new genetic material in addition to more traditional mutations. Adding enough genetic material to the system without preventing the system from optimising is a careful balance. In this work we will show that this is possible, that increased use of synthesised code may be beneficial and that it is vital that we consider length to maximise optimisation by preventing the code from becoming bloated.

The second is that we are attempting to optimise to a set of environment conditions detected by a running system in real-time; the function calls on a given building block in such an environment can be captured by the live system, but only for short periods of time to avoid unreasonable disruption to the running system. The data to which a building block is being optimised is therefore *sparse* and may provoke over-specialisation to a particular input sequence which is of limited use to the wider environment class from which that sequence was sampled. We therefore need to synthesise additional input data from our sparse sample to counter such tendencies for over-specialisation. Here we show that we can use synthesised data both partially and on its own in place of using larger amounts of data to optimize a function to the same or better level than the larger data methods.

We distill these challenges into two specific reseach questions:

(1) Can synthesised genetic material improve the ability of a GI process to find high-utility implementations from limited source material starting points?
(2) Can synthetic training data, based on sparse sampled data, reduce over-specialisation and improve a generated implementation's performance in the general environment class from which a sparse sample was drawn?

Our studies specifically examine hash table implementations, which use a hash function to take a text-based key, and map that key to an integer value within a fixed range. A hash table then uses a fixed-size number of buckets, and the key is used to calculate which bucket a given key/value pair is stored in. This allows for quick retrieval as only that bucket needs to be searched for the key/value pair (rather than the entire contents of the hash table). An ideal hash function will disperse the given keys equally between the available buckets, ensuring the fastest average retrieval time. Hash tables are very common to a wide range of systems, representing a useful way to study our research questions for wide applicability.

Our study of research question (1) is partly based on a prototype investigation into blended synthesis [McGowan et al. 2018]. We partly replicate these results, with an entirely novel GI framework implementation, but also examine multiple different synthesis methods, multiple different fitness functions, and we more deeply study the potential side-effects of synthesis phases. Our results demonstrate that (i) mixed synthesis mutations do contribute positively to enhanced individuals (a 2% improvement over 200 generations); (ii) small samples of source code subjected to GI can rapidly experience 'burn out' across a generation as useful elements of that code are removed; and (iii) dedicated expansion/synthesis phases interleaved with regular GI phases do not appear to contribute to long-term improvement, but do prevent burn-out effects.

Our study of research question (2) examines multiple different data sampling and synthesis approaches to help understand how data synthesis from sparse sample data impacts a GI process. Using the same GI framework as above, our results show that synthesised data can indeed produce statistically equivalent results to larger data sets training and performance on unseen data of the same class, gaining up to 40% performance gains on improved candidates – but that over-fitting to training data is still possible using our current approach. We also find that GI processes tended to generalise or specialise based on the *class* of data used (e.g., English words, or Polish words) in addition to whether it is using sampled or synthesised data. Some classes of data prove to be 'easier' to train for, and are more likely to lead to generalist individuals, while 'harder' classes to train for are more likely to produce specialists which are statistically significantly better than generalist results. The source code used in all of our experiments is made available online for repeatability, including detailed instructions and the data analysis scripts used [Rainford 2022].

In the remainder of this paper we first briefly introduce background on emergent software systems, then present closely related work in Sec. 3. In Sec. 4 we present our study methodology for both research questions, and in Sec. 5 and Sec. 6 we present our results for research questions (1) and (2) respectively.

2 EMERGENT SOFTWARE SYSTEMS

The concept of self-assembling systems has been examined in various forms, from hardware systems such as space telescopes and satellites [Rognant et al. 2019; Underwood et al. 2015], and swarm robotics [O'Hara et al. 2014].

Emergent software systems aim to bring these ideas to everyday software, such as web servers, caches and databases, big data processing frameworks, and microservices [Dean and Porter 2021; Filho et al. 2018; Rodrigues-Filho and Porter 2022]. Emergent software combines the general ability to wire units of logic together to form a system, and rewire those units safely while a system is running, with real-time machine learning which quantifies the current operating environment and learns online which combination of building blocks has the highest utility in each environment. Emergent software systems therefore automatically discover all possible ways to construct a target system, assuming various building blocks have implementation variations, and then use a process of continuous learning and re-assembly to optimise to the current environment. Emergent software systems generalise long-running trends towards autonomous and self-adaptive systems [Kephart and Chess 2003; Salehie and Tahvildari 2009], placing self-adaptive behaviours directly into the real-time logic composition of running systems.

At present, emergent software requires developers to manually design multiple variations of each building block – such as different sorting algorithms, cache eviction policies, or scheduling algorithms. In this paper we proposed and study a fully automated approach to variant generation using GI, able to generate variations that are of high utility to the deployment conditions actually experienced by a running system.

Our GI framework targets a novel programming language called *Dana* that underpins a large range of emergent software research. This is a full-stack object-oriented systems language which supports ubiquitous hot-swapping of components with soundness guarantees for the running system [Porter and Filho 2021]; the language is predominantly used in research, particularly in the self-adaptive systems community. Our research work is the first major example of genetic improvement which targets Dana. We note that while our results are likely to generalise to other programming languages, using Dana allows us to complement and integrate with the existing tool support and experimental testbeds commonly used in emergent software systems.

3 RELATED WORK

To date, GI for source code has been deployed predominantly for automated bug-fixing, with a smaller volume of research on automated optimisation. In wider research, GI-based optimisation to an environment has more typically been applied to parameter tuning or non-code domains, though work has also been done on using GI to update hard coded parameters and data [Langdon and Krauss 2021]. In this section we cover relevant research in GI for both areas.

In source-code bug-fixing and optimisation research [Langdon and Harman 2014; Petke et al. 2018, 2013], GI methods have long relied heavily on the use of an existing code base of a large project to source solutions to problems, where insertion of genetic source code material draws on existing code elsewhere. This technique is formalized as the 'plastic surgery hypothesis', with the finding that in large bodies of Java code 43% of human commits could be sourced from the existing code base [Barr et al. 2014]. This is a challenge for our emergent systems application domain of GI, in which there is no 'large body' of code because each building block is developed in isolation; within the scope of a single building block there are very few lines of code (around 100-200) in total, giving us a far smaller chance of finding useful existing code as a source for improvement.

Our approach of starting from such a small code base in some ways has closer kinship to the field of genetic programming for program synthesis, which start from no existing source code. This generally includes an element of code reuse from already-synthesised populations members, partly similar to our methods here. A particular example of this is n-version program synthesis [Chen and Avizienis 1978] which produces multiple versions of the same software to provide redundancy for fault tolerance. However, this kind of research differs in that it expects to start from multiple separate populations and will attempt to yield distinct software variations which will likely have different faults so that in combination they provide fault tolerance. Our work starts from a single population member and attempts to create new variations which are all equally correct but have different performance characteristics in different deployment conditions.

Beyond source-code, GI-based optimization to a deployment environment has often been considered with a focus on optimization to hardware such as battery usage and screen resolution [Li et al. 2014; Linares-Vásquez et al. 2015] rather than to data input. The focus of the challenge here is on multi-objective approaches that balance issues like battery usage and aesthetics to gain the best user experience, rather than on synthesising genetic material for starting individuals with a low amount of initial material.

In this research we draw from these areas to focus on blending traditional genetic improvement with synthesis of new genetic material, enabling us to generate new variants from a single population member to optimize towards the deployment conditions in which that code is operating. In creating these new variants, and pushing them as alternatives to the deployed system, this should provide optimized code blocks for an emergent software system to dynamically learn and utilize for the specific deployment conditions that it experiences.

Outside of the code re-use vs. synthesis question, the above GI methods generate improved candidates either by testing against a series of established test cases or by relying on a large data

corpus to train. This is a practical approach as most of these systems are designed to just train once they can take time to collect data and develop test cases. Our emergent software systems domain differs, however, in that these systems experience changing deployment environments (some of which are likely to recur) and try to optimise towards each one; because each environment may be new, we must sample input data from the running system to replay offline as an optimisation target. Because these systems are operating live, such samples must be short so that there is a low impact on the performance of the system's primary operations. For GI, this prompts the question of how we go from a small data sample to one that is large enough to train on without over-specialising. In existing work, data corpora have been expanded using various methods for data synthesis, particularly in image categorization [Alhammady and Ramamohanarao 2005]. Genetic methods, for example, are used to mutate and crossover existing data to create new data to expand categorisation data and data to help with the recognition of sign language [Wang et al. 2006]. Other systems have used neural networks to partly synthesise data by adding synthesised elements to images in order to make early training easier [Stergiou et al. 2018]. Very few systems generate entirely new data, though there are some examples of generating meta-data as a new data set to work with instead of the original visual data set [Long et al. 2018].

Finally, while we focus in this work on the optimisation of a hash function in the Dana programming language, previous attempts have been made to repair and optimize hash functions in other languages such as Java; in particular the work on Kocsis et al [Kocsis et al. 2014] relies on inheritance and the need of the hash function to meet contractual requirements and produce a uniform distribution. In this work we focus on the requirement for speed from our hash function; in many cases this should equate to uniformity of distribution, but in edge cases may also offer advantages of ignoring uniformity – such as when dealing with natural language on a small hash table, where an optimal solution would place more common words in less-used hash codes and rarer words in more full ones, thereby allowing for quick retrieval on more common words. As far as we are aware, our work is the first example of automated variant generation at a source-code level for self-adaptive systems and yields a number of questions that are specific to this domain.

4 METHODS

Our overall framework is illustrated in Fig. 1. An emergent software system is assembled from a large collection of small building blocks, 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 core of our system is then based on a typical genetic improvement process, Algorithm 1, using mutation (line 5), fitness (line 6), selection (line 10), and crossover (line 11). We include crossover in this work to make use of existing code in expanding the code base of members of our

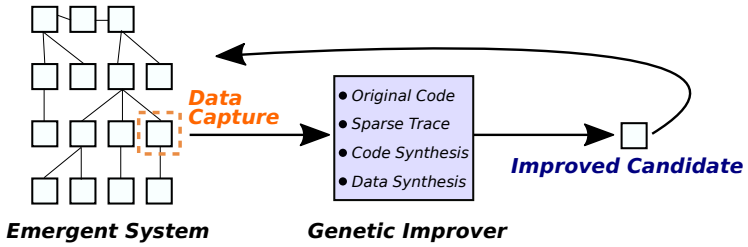

Fig. 1. Our overall approach, with data capture from a running emergent software system, requests for improved variations of particular building blocks to a GI system, and a GI process which uses mixed code synthesis to counter a low volume of available genetic material, along with input data synthesis to counter a sparsely sampled input trace towards which we are optimising.

Algorithm 1 Genetic Improvement Algorithm

```
1:  for i = 0 to generations do
2:      if i == 0 then
3:          create initial population of clones
4:      end if
5:      mutate a % of the population
6:      check fitness of all population members
7:      if i%5 == 0 then
8:          check performance on unseen data of all population members
9:      end if
10:     select new population (roulette wheel)
11:     crossover a % of the population
12: end for
```

population. We also include new code synthesis in our mutations, in combination with more typical mutation types, and a performance check (lines 7-9) which does not effect training but is used to examine whether the algorithm is training to the data class (and performs similarly on unseen data of that class) or to the specific data (and performs poorly on unseen data of the same class). In the remainder of this section we describe how each element of the algorithm works in detail.

Initial Population. The initial population of hash functions in the system are clones of an initial small, simple function, which is shown in Listing 1 .

There are 30 copies of this code in the initial population which will be mutated before assessment. This initial hash works by treating the input 'key' string as a character array and computing the sum of the numerical (binary) value of each cell of that array. The result is then restricted to be within the range of the hash bucket count by using a modulo operator.

Mutation. In this stage we apply a single mutation to a percentage of the population – where a single mutation is a change to one line of code. This change may be either adding a new line of code (insertion), removing a line of code (deletion) or changing variables or operations in a line of code (modification). Our mutation logic is aware of the grammar of the language and is designed to yield code which is both syntactically correct and also type-compatible for operations such as variable assignments. More detail on mutation, and code synthesis, is given in Section 4.1.

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

Listing 1. Original Hash Function

Fitness. The fitness function takes a genome, in this case the code of a hash function, and evaluates it to provide a fitness score. Here we test the phenotype of the code by loading it into a testing environment. We then time how long it takes the code to input 1,000 words as a key/value pair to the `put()` function of the has table, with the key and value being two copies of the same word. We then retrieve the same set of 1,000 words from the hash table using the `get()` function, checking to make sure the returned value still matches the string (and that the hash function has remained deterministic). Each incorrect return in the `get()` stage incurs a time penalty of 10 milliseconds for fitness evaluation, such that an implementation which returns an incorrect value for all 1,000 words will gain a total time penalty of 10,000ms. This value was selected to be significant when it produces errors above the noise introduced by our unreliable timing mechanism (native machine clock time comparisons), while also being proportional to the faster fitness times so as not to reward fast but incorrect solutions – where faster fitness times are in the region of 150 milliseconds in total. During fitness testing we also set a maximum time limit for the above tests to run, with a forced stop of the test if the running time exceeds this. This defends against the potential for infinite loops that are created through mutations. The same maximum fitness time is also assigned to population members that fail to compile – though this is rare as our mutation approach is grammar-aware. The above fitness testing results in a speed score of s_n, which we combine with the length of the code measured in lines l_n to give a fitness score, f_n:

$$f_n = s_n + 5l_n \tag{1}$$

In the above equation we multiply l_n by 5 to bring it into a similar range to average measurements for s_n in our starting individual, such that both parts of the equation have similar weight. Because we base our fitness score on speed, we are seeking a minimum value, so lower scores are better; we consequently approximate that lower scores for length (fewer lines of code) are also better, which helps to avoid uncontrolled code bloat through insertion mutations that could slow down the GI process to an ineffective level. In our experimental results we list the “relative" fitness scores of our code, using Equation 2. This is the fitness score of each hash function relative to the fitness score of the original code as assessed with the same fitness function. We use this relative approach to allow us to compare values across variations in the fitness functions.

$$r_n = f_n / f_0 \tag{2}$$

Performance. This stage works in the same way as the fitness function, but uses different input data – with the results of this test not used in the selection process. We call this *performance* as it is tested on *unseen* data which mimics the performance of the solutions in deployment after training. This allows us to measure over-specialisation to training data as the GI process progresses.

Selection. This phase copies selected individuals from the current population into the next population. There are three common methods for doing this: tournament, universal random, and roulette wheel. We use roulette wheel selection, proportional to the normalised inverse value of the fitness of a population member, so that their chance of selection is based on their score relative to the rest of the population. If the individual has the best score, under this approach, it will have the best chance of selection. We supplement this selection method with elitism, in which the best n members of the population are copied to the next generation unchanged. We use elitism with $n = 2$, with the rest of a generation formed by a weighted random sampling with replacement of the previous generation, using the roulette wheel selection method.

Crossover. This phase is designed to mimic biologic reproduction which causes gene transfer between two individuals. We use one-way crossover, which selects code from one individual and copies it and inserts it in another piece of code; we specifically take a single line from one piece of code and insert it in another. More detail on the process of crossover can be found in Section 4.2. Crossover is applied to a percentage of the population, such that for each population member (M_o) we select a second population member (M_s), then select a line in M_o, T_o, and a line from M_s, T_s; we then check scope issues and insert T_s at T_o.

4.1 Genetic Mutations

In this section we present the detail of how mutations work, including how we mix code synthesis with more traditional mutation types. We first describe our approach to sythesising new genetic material for insertion, then present our general mutation approach.

4.1.1 Synthesis. Whenever an insert mutation is selected, we synthesize basic code of three types: variable declarations (Listing 2), operations (Listing 3), and control structures (Listing 4). All three are restricted to the use of primitive types only (integers (int) range: 0 - 1000, characters (char) range: a - z, decimal (dec) range: 0 - 1 and booleans (bool) 0 or 1), and are inserted either before or after a code token in the existing code (including potentially inside existing control structures).

Example variable declarations are shown in Listing 2, and involve a type declaration and a variable name, using a lowercase letter string (generated in alphabetical order, e.g. x, y, z, aa, ab, ac), with an equals operator and a randomly generated constant which matches the declared type.

```
1  int a = 26
2  dec b = 0.029817845987
3  bool c = false
4  char d = "g"
```

Listing 2. Synthesized declaration examples

Example synthesised operations are shown in Listing 3. These are synthesised using pre-existing variables that are in-scope at the intended insertion point. Depending on the types of in-scope variables an operation is chosen at random (to make sure we have variables of the type needed). Once an operation is chosen, variables are chosen at random to serve as operands from among those that are in-scope. Operations use between one and three variables, e.g. $a = b * c$ or $a++$.

```
1  result = a + result
2  a += result
3  result ++
4  result = result % a
```

Listing 3. Synthesized operation examples

An example control structure is shown in Listing 4; we support for-loops, while-loops, and if-statements. In the case of for-loops the variable name for the loop is chosen from the same list as other variable declarations, and the limiter on the loop is set using either a variable that is in-scope or from a randomly generated integer. In the case of if and while structure the Boolean condition is generated using existing variables. All control structures are generated with an empty body (which can be filled in with future mutations).

```
for(e = 0; e < result; e++) { }
```

Listing 4. Synthesized control structure example

4.1.2 Mutations. Overall we use three types of mutation: insertion, deletion, and modification, each on which has different conditions placed on them. We select which type of mutation to apply randomly based on a configurable parameterised weighting (the specific values we choose are presented in our experiment setup in Sec. 5.1).

Insert mutations are always possible, but the inserted code must not cause scope problems in terms of variable usage. The checks to prevent this slow down the insertion process to a degree, but are assumed to speed up the overall genetic improvement process by ensuring that a larger proportion of the population remains viable code. An example of an insertion mutation applied to the original code in Listing 1 would be Listing 5, where the variable *a* has been added with the integer type and a random integer as its initial value.

Modifications can be performed on operations or conditions in control constructs; in both cases they can modify either the operator or the operands. In the case of operands we replace an operand with a variable of the same type. In the case of operators, we select an operator that works on the quantity and types of operands. In the example Listing 5 we have mutated the operator and one of the variables to use the inserted *a* variable. Thus in the operation within the for-loop, the operation has changed from $*$ to + and *result* is now *a*.

```
int hash(char key[])
{
int result = 1
int a = 36
for (int i = 0; i < key.arrayLength; i++) {
result = a + key[i]
}
return result % HT_LEN
}
```

Listing 5. One inserting and two modification mutations example

Deletions have conditions on their use to ensure that the resulting code is still compilable. Simple operations can always be deleted, but control structures and declarations have restrictions. Specifically, we can only delete a variable declaration if that variable is not used later in the code; and we can be only delete control structures if there is no code in the body of the control structure. This check is very simple due to our code representation in which we simply check for a lack of child nodes within the structure's scope. Two separate deletions are therefore needed to remove the control structure in Listing 5: the first to delete the content, Listing 6, and the second to delete the structure itself.

```
int hash(char key[])
{
int result = 1
```

```
int hash (char key[])
{
...
result = 1
result = result * result
result = result * result
result = result * result
i = 0 i = 0 i = 0 i = 0
e = 843 bool a = 1 i = 0
h = 823 i = 0 i = 0
result = result * key[i]
result = result * result
result = result * result
result = result + result
for (i=0; i<key.arrayLength; i++)
{
result = result * key[i]
result = result * key[i]
}
return result % HT_LEN
}
```

Listing 7. Actual optimized population member at 200 generation, truncated to active part of code

```
int a = 36
for (int i = 0; i < key.arrayLength; i++)
{
}
return result % HT_LEN
}
```

Listing 6. First deletion example

This particular example series of deletions does not create a good hash function (it always returns 1 and so places everything in a single bucket); by comparison, one of the best population members from our experiments after 200 generations is given in Listing 7.

By this stage the hash function has gone from the simple original hash function:

$$r_n = \prod_i k_i \tag{3}$$

To become a more complex hash function which projects key values into a larger space before the modulus is taken:

$$r_n = 2k_0^3 + \prod_i k_i^2 \tag{4}$$

4.2 Genetic Crossover

Genetic crossover works in much the same way as insertion, except that the code inserted is taken from another member of the population. This creates additional issues surrounding scope. Since all population members use the same set of names for synthesised variables, and have the same original source code, there is a very high likelihood of variable name conflict in the inserted code.

If the code is inserted without consideration then variables could be: undeclared at the point of insertion, declared twice in the same scope or have different types at different points in the code.

To resolve issues of this sort we will often rename the variable in the code being inserted as well as adding additional declarations with the inserted code for it's renamed variables. Renaming of variables uses the same name list as synthesis of variable declarations.

An example of crossover is visible in Listing 7 where a second copy of the $result = results * key[i]$ line has been inserted. In this case all the variables used were declared in the insert location scope with the same type so there was no need to rename. If this was inserted before the declaration of *result* then we would copy the declaration for the insert code and insert it with the line. This would mean there were two declarations for the variable so the second one would be deleted.

5 GI WITH MIXED SYNTHESIS STUDY (RQ1)

Our first study examines the problem of limited genetic material in an initial candidate, and the use of mixed code synthesis with more traditional GI mutations. We use four different fitness approaches to examine the efficacy of this approach, and understand two sub-topics: (i) the extent to which genetic improvement with synthesised genetic material for insert/modify mutations can improve source code; and (ii) how specific 'expansion' phases, where high synthesis rates of new code are used periodically to add larger amounts of new genetic material, affect the improver. We also compare our approach against a simple hill-climbing implementation to verify the utility of the overall GI method.

In detail, the four fitness variants used in this study are referred to as *Speed Static*, *Length Static*, *Length Dynamic*, and *Speed Dynamic*. The first three of these variants are different versions of the fitness function alone, while the fourth (Speed Dynamic) also changes mutation probabilities.

Speed Static Keeps a fixed fitness function and mutation/crossover probabilities over time, using a fitness function which focuses only on performance s_n in clock time of a population member n, where a lower time to complete the fitness test is better. We can express this fitness function f_n as simply:

$$f_n = s_n$$

Length Static This fitness function uses speed measurements, s_n, and combines this with a measurement of the total length of code l_n measured in abstract syntax tree nodes; we also weight the total code length to be of a similar range to typical speed measurements. Experiments following this fitness function reward code which is both fast *and* short:

$$f_n = s_n + (l_n * 5)$$

Length Dynamic This is a dynamic fitness approach which starts using the fitness function from *Length Static* but switches to an alternate fitness function every 5 generations, so that half the time it encourages expansion by rewarding longer code. The length-static fitness function is exactly as defined above, and the alternative expansion-oriented fitness function is given below.

$$f_n = (s_n - (s_n \% 10)) + 5(200 - l_n)$$

Our expansion-oriented fitness function, which is used alternately with the length-static fitness function, does two things: it *reduces* the importance of speed without removing it entirely, and it *inverts* the code-length part of the fitness equation. The importance of speed is reduced partly by smoothing out minor variations to round speed to the nearest 10ms, and partly by multiplying the code-length component by a factor of 5. The code-length element is inverted by assuming that 200 lines of code is the largest hash function that might be used, and subtracting the actual code-length of the individual from 200. A hash function with a length of 10 lines therefore yields a code-length measure of 190, while a hash function with a length of 50 lines yields a code-length measure of

150. Because this fitness function overall still favours lower values, the *longer* hash function would therefore be considered better in this case.

Speed Dynamic Lastly, to help understand whether there is a better way to encourage code expansion than through the fitness function, we also test a system which uses a common fitness function of speed-based measurements, but every five generations increases the likelihood of insertion mutations and the proportion of crossover.

Our evaluation examines the effects of our genetic improvement method on a real hash table implementation, using curated key/value input data as our improvement target as described next. All of our source code, with instructions on how to reproduce the results in these experiments, is available online [Rainford 2022].

5.1 Experimental Set Up

Throughout all of our experiments in this study, the specific parameter settings used for the genetic algorithm are listed below; the majority of these were based on a previous study [McGowan et al. 2018] which chose them after a limited manual search of the parameter space to determine a good configuration. Note that we have specifically changed the population size and generation count compared to this previous study in order to increase the search power of the GI process and better test the method. The 'speed dynamic' study variant switches between the standard and *alternate* settings to examine our second variant of mixed expansion phases.

- Population Size: 30
- Generation Count: 200
- Dynamic interval: 5 time steps
- Mutation proportion: 80%
- Crossover proportion (Normal): 20%
- *Alternate* crossover proportion (Speed Dynamic): 30%
- Mutation weights (Normal): insert - 30%, modify - 30%, delete - 30%
- *Alternate* mutation weights (Speed Dynamic): insert - 50%, modify - 25%, delete - 25%
- Number of Elite population members: 2
- Number of replicates: 30

The key/value improvement data that we use is designed to emulate a dataset that could be captured from a live deployed system. The characteristics of this data imply the direction that genetic improvement is likely to take the initial source code in deriving an improved version which is more specialised to this input sequence. In general, a hash function is likely to be influenced by (a) the average key length; and (b) the distribution of characters within keys. Our input data for the keys used here is based on a uniform random list of English words with a maximum length of 15 characters. The actual distribution of lengths and characters in the training and testing data (used for fitness calculations and performance testing respectively) are shown in Fig. 6. The distributions are broadly similar, but also have clear differences in average lengths and character distributions – indicating that we should be able to detect over-specialised output solutions of the genetic improver.

5.2 Results

We begin with our *Speed Static* study setting, in which our fitness function is fixed and only considers performance. This demonstrates the baseline effects of our synthesis elements in insert and modify mutations, without any specific expansion phases.

The results for all of our study settings are show in Fig. 3, Fig. 4, and Fig. 5, in which the x-axes represent generations over time, and the y-axes represent fitness or performance score for that generation. In all of our results, when calculating fitness or performance, we use the equation:

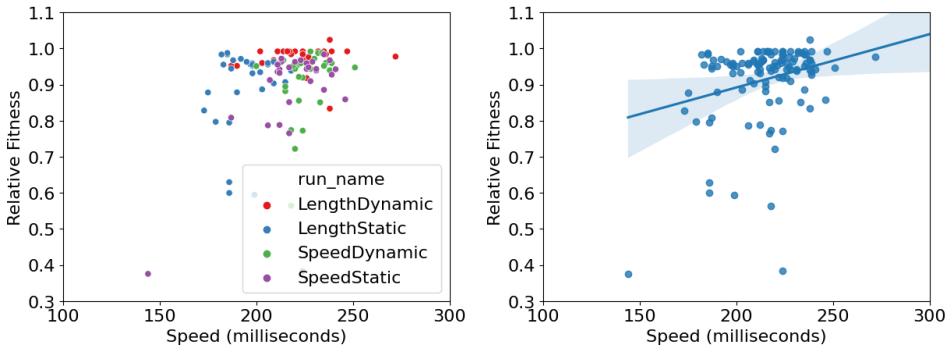

Fig. 2. Comparison of the raw millisecond speed of the final best individual from each GI variant, plotted against their relative fitness; each algorithm variant is highlighted on the left graph, with a line of best fit showing positive correlation (right).

$$r_n = f_n / f_0 \tag{5}$$

This is the fitness score of each hash function, relative to the fitness of the original code assessed with the same fitness function. This allows us to consider a single set of values over time when the fitness function used changes. In all of our graphs, a lower value on y-axes is therefore better.

To show that this correlates with improvements in the millisecond speed of the final best individual across our different fitness functions, in Figure 2 we plot relative fitness against raw millisecond speed (Speed) of the best individual from the final generation of each fitness function condition. This shows the different data points by algorithm variant (left) and the complete data set with a line of best fit (right). While there are clear outliers, the best fit line does show a clear positive correlation between the two metrics; to aid in comparisons across conditions we therefore use relative fitness as our main measurement throughout the remainder of our results.

Considering our Speed Static study setting (bottom right), we see that fitness of the best individual (Fig. 3) does improve across generations by 10%, and that the majority of this improvement occurs within the first 50 generations[1]. The average fitness of all population members shows a similar trend (Fig. 4), but also shows the effect of elitism and fragility: after 100 generations, the population overall begins to show markedly poorer fitness. This trend is only just shown in the previous graph of single-best-individual fitness because elitism carries over the best two population members un-modified between generations. To help understand why the general population incurs this more significant drop in fitness we examine the best single population member from this experiment group, shown in (Lst. 8).

Here we see that the mutations and crossover deployed into the population member change significantly how keys are derived. We can summarise this as moving from the original hash function which generates a hash index r_n using a formula:

$$r_n = \prod_i k_i \tag{6}$$

[1]Note that, on many of our elite-individual graphs, we see that fitness appears to get worse at some points during a run. This would not normally be possible, but is a result of slight changes in the environment: because we measure fitness as execution-time of a real running system, we see natural variation in these measurements caused by perturbations in OS scheduling policy or low-level CPU cache behaviour. This timing variation occasionally manifests as 'worse fitness' of an elite across subsequent generations.

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

```

Listing 8. best code after 200 generations from a single run

To the example best population member shown in Listing 8, which effectively uses a formula:

$$r_n = 2k_0^2 \prod_i k_i^2 \tag{7}$$

The effect of this is to project the original value generated by the original function into a much larger range within the 100 available hash buckets. This spreads the values out further before we take the modulus, giving a better distribution of values for this particular set of input keys. The majority of this effect comes from the duplicate line, created by crossover, within the for-loop – and so we can say that the optimisation of the best population member depends on a single line of code. This makes the optimisation very fragile to delete or modify mutations, which explains the way in which the overall population for Speed Static can easily drift into far worse territory from which it is difficult to recover.

We next examine the results from the second study group, Length Static (Top Right). In this experiment we add a reward to the system for shorter code. This tests if length is an important factor in our experiments and provides a contrast for our dynamic fitness function which rewards expansion and contraction in the genome alternately.

In fitness of the best individuals (Fig. 3), we see that once again most of the improvement is in the first 30 to 40 generation but after this the results vary less than in the Speed Static experiment, possibly due to the smaller amount of bloat in the code meaning less variance due to bloat. We see reduction in fitness at times despite the use of elites in these results due to the errors in our timing mechanism (comparison of two calls to the system clock, which has a small error), including the same reduction just after 100 generations which we see in Speed Static. The average fitness of all members of the population (Fig. 4) reflects this instability due to increased fragility when there is statistical difference between it and Speed Static both of which are very noisy.

We next examine the results from the third study group, Length Dynamic (Top left). Here we are looking to see if the introduction of an alternating fitness function, which switches between rewarding shorter and longer code, can help to better optimize our hash function by increasing the amount of genetic material available. Considering the fitness of the best population members (Fig. 3) we can immediately see the periodicity in the results caused by the changing fitness function as it re-optimises with each switch. Much like in the Static experiments, however, we see the main

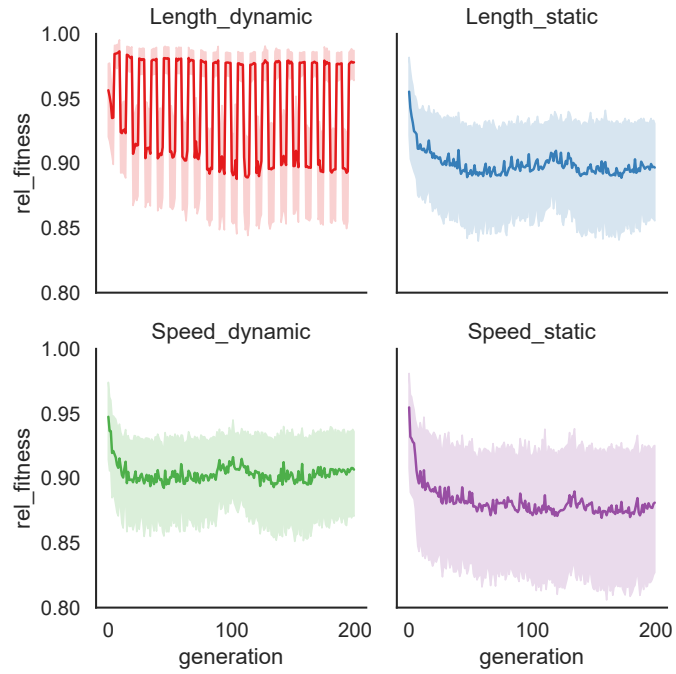

Fig. 3. Mean fitness of the best population member in each generation against training data, given as mean value of 30 repeats for each algorithm variant: Length Dynamic (Top left), Length Static (Top right), Speed Dynamic (Bottom left) and Speed Static (Bottom right).

improvements occur in the first 50 generations. Beyond this point, besides oscillations caused by our changing fitness function, the average fitness improves a little until just after generation 100 where it does not noticeably out-perform the Length Static experiment for fitness when on the same fitness metric.

Examining the average population fitness under these conditions (Fig. 4) we see that the Length Dynamic experiment does not experience any distinct periods of fitness degradation. We hypothesise here that the increased size of the code during expansion makes it less likely that the optimizing line of code is removed or modified, and the decreased size of code during contraction phases would make it easier for the population to recover within the fitness function cycle that already encourages continual re-optimization.

We next examine the results from the fourth study group, Speed Dynamic (Bottom left), which provided an alternative form of expansion phase by explicitly increasing the probability of insertion mutations and crossover (rather than rewarding longer program length). This expansion phase matches the frequency and length of the expansion phase in Length Dynamic. Examining the fitness of the best individuals here (Fig. 3) we do not see the same periodicity in the fitness scores as with Length Dynamic. This is because the increased length of the code will not always directly effect the fitness score and the best individuals will not necessarily expand. While the overall fitness of Speed Dynamic is the same as Length Static, it doesn't perform as strongly as Speed Static, which suggests that their contraction-based fitness functions may them more consistent but less effective.

Considering the whole population (Fig. 4), however, we see that Speed Dynamic does not have a noticeable fitness decline like the Speed Static and Length Static experiments. This is likely due

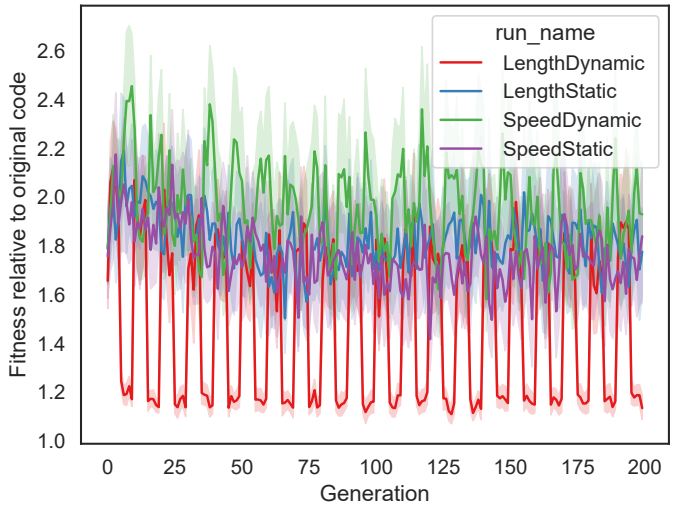

Fig. 4. Mean fitness of *all* members (30) of the population against training data for all repeats (30), for each of the four algorithm variants.

to the longer code making it less likely for an important line of code to be modified or deleted. Overall, the population here (despite being more robust) is periodic like Length Dynamic but does not perform as well as other algorithm variants at the end of the experiment – likely because the increased length of the code slows it down.

Finally, we consider how well the best populations members from all four experiments perform on unseen data generated in an identical manner to the training data. This allows us to assess whether evolution is specialising to the particular input data or the general class of training data. This is important in a real deployment environment as the actual data from the live system will vary over time and will include values not in training data.

Starting with Speed Static we see more clearly here that the decline in performance of the overall population, which was not very visible in the best members when tested on the training data, as a sudden increase in spiking in the performance. It is likely that the elite members of the population had remained better only due to features of the training data. There are small differences in the length and letter distributions (Fig. 6) between the two data sets. Given that the hash function is evolving only to distribute the keys into a larger space, to increase the chance of an even distribution, small variations may better suit one data set's particular words than the other.

The other three experiments: Length Static and Speed Dynamic perform very similarly on the unseen data suggesting that the evolved hash solution of squaring the values, which occurred in all experiments, is a good general solution to hashing English words. The Length Dynamic variant has more robust results and a clear periodicity caused by the changing fitness metric but the overall performance is statistically the same as the other variants. The experiments which incorporated length in their fitness functions were able to specialise to the particular word set better than the others but this did not produce better general performance than the general trend of evolution on unseen data.

Across all results we used a signed-rank Wilcoxon test to gauge statistical significance, which suits data that may not have a normal distribution. Using a significance threshold of 0.05, the results in this study indicate that all GI variants clearly produce a better hash table implementation

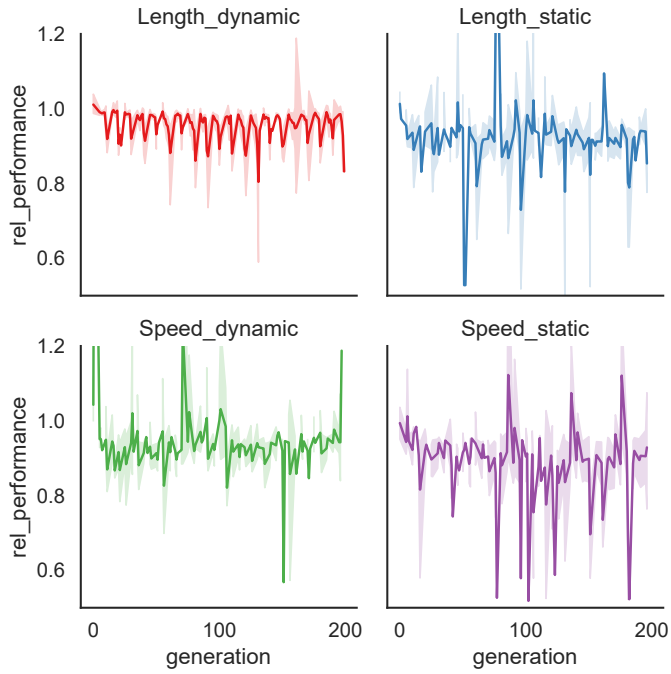

Fig. 5. Mean performance of best population members on unseen data. Performance is assessed every 5 generations. The x-axis on all graphs is the generation number, and the y-axis is the relative performance compared to the first generation (lower is better).

than that of the starting individual, but that the GI variants in comparison with each other are not differentiable in a statistically significant way. As an example, the largest statistical difference in this data is between speed-static and speed-dynamic, at a value of 0.125, which is clearly not below our significance threshold of 0.05. Full statistical results for all conditions are given in Appendix A.

5.3 Comparison with hill-climbing

To date there have not been any prior attempts to optimise source code in the way that we do here, leaving us without a viable direct comparator for evaluation. In order to confirm that the GI method in general is successful, compared to a simpler approach, we therefore implement a hill-climbing alternative. Our hill-climber uses the same set of mutations as our GI approach, but without crossover. At each time-step in our hill-climbing algorithm up to 100 attempts are made to find a successful step, using mutations, for a single population member which yields a better performance level (we note that the 100 attempts is more than 3x the size of our GI population). If no step is found the hill-climb run finishes there.

The results are shown in Fig. 7, using the same fitness function as in our length-static GI experiment, where each point represents the best individual at the end of one of the 30 runs. This demonstrate that the level of improvement seen here is far more limited than the GI method using the same fitness function, which achieved a best individual fitness of 0.57 compared to 0.75 will hill climbing (where lower is better). From a manual analysis of the best individuals in both cases we can infer that this poor performance in hill-climbing is essentially due to a lack of crossover: our GI runs benefit from duplication of an existing line of code during crossover which provides

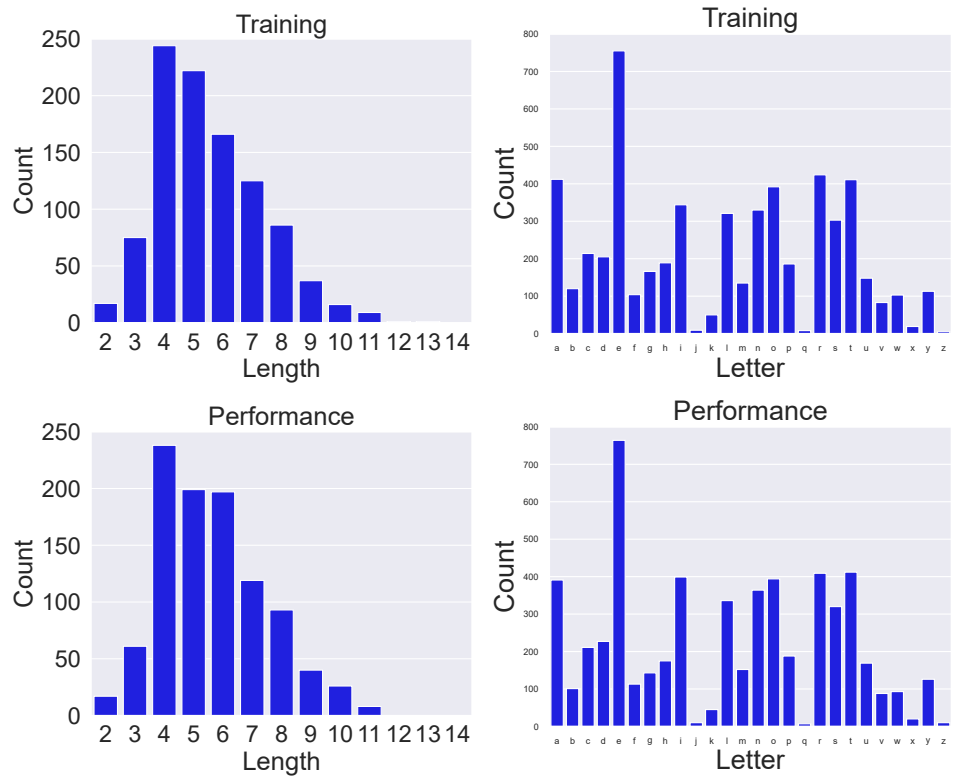

Fig. 6. Length (left) and Letter (right) distribution of the training data (top) and the unseen testing data (bottom)

significant performance increase in a child individual, and further crossover then spreads this code to other individuals in the population in successive generations.

5.4 Discussion

Our results demonstrate firstly that GI with mixed synthesis operators, for insertion and mutation, produces positive results with hash table implementations that are better tuned to a particular environment – and, as demonstrated by our example best population member, that these improvements can arise from synthesised code.

We also see that the use of specific phases of expansion, in which the GI process has a preference to insert more genetic material and/or reward longer programs before returning to its normal fitness function, does not affect overall performance – with no statistical differences in the final generations. It does, however, show a benefit of protecting more fragile optimisations and preventing burn out. This could also indicate that introducing a level of dynamism in the improvement algorithm (forcing re-optimization) may prevent detrimental mutations from staying in the population. Further investigation into this effect would be worthwhile and would require consideration to eliminate other variables that might effect “burn out" in our algorithms.

Phases to encourage expansion in our code (Length dynamic and speed dynamic) demonstrate almost equivalent optimization levels in terms of our final best individual fitness of the algorithms with our static experiments. The inclusion of a specific length measure in our fitness function, in

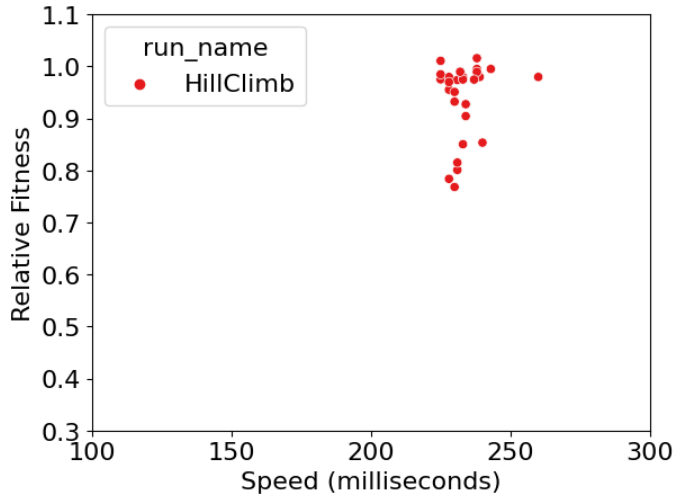

Fig. 7. Results of an alternative hill climb only algorithm, where each point is the final best individual of one of 30 repeats, where fitness is measured using our length-static fitness function. Results are significantly worse than GI using the length-static fitness function.

addition to simple performance testing, actually has a stronger effect on improving optimization for best individual fitness, though does appear to introduce a significant increase in noise to the best-fit individual.

On unseen data, outside of the training set, we see relatively minor improvements. While performance improves to an extent, indicating that the improvement process is yielding candidates which somewhat generalise to a class of data rather than over-specialising to their training data, the level of improvement here is significantly less than the improvement seen on training data. Our next study examines the question of specialisation vs. generalisation in detail.

6 DATA SYNTHESIS STUDY (RQ2)

Our first study demonstrates the potential to mitigate limited genetic material by blending code synthesis with traditional mutations. However, it also demonstrates a tendency for an improved candidate to over-specialise to the data that it is being trained on, rather than the broader class that this training data represents.

In our second study we examine whether GI using synthesised data can avoid over-specialisation while still yielding candidates that specifically fit a particular deployment environment (rather than being good at all environments). As hash functions are often applied to natural language words as keys, we draw sample data from the text of news articles from four different newspapers. Two are UK-based as English examples (The Economist and The Sun) and two are non-English examples (Le Monde, in French, and Rzeczpospolita and Wyborcza, in Polish). These use of both different writing styles and different languages so serves as a good target to study specialization and generalisation. We use our length-static fitness function for all experiments in this study and the default parameter settings reported in Section 5.1, therefore having a fixed parameterisation, and instead vary our training regimes. Again, all of our source code, with instructions on how to reproduce the results in these experiments, is available online [Rainford 2022].

6.1 Data Sources

We use the following articles from the online versions of each newspaper:

- Economist training (1754 words)
  - Covid-19 vaccines have alerted the world to the power of RNA therapies [eco 2021b]
- Economist test (1511 words)
  - Are vaccine passports a good idea? [eco 2021a]
- Sun training (1685 words)
  - Elon Musk changes his official job title to 'Tecknoking' of Tesla [Pettit 2021]
  - Official covid R rate creeps up again - but still hovers below crucial 1 [Williams 2021]
  - Monaco's Prince Albert slams Harry & Megan for 'inappropriate' oprah interview [Fuller 2021]
- Sun test (1213 words)
  - EU to agree covid passport scheme allowing vaccinated brits to travel [Aoraha 2021]
  - Quirky uk summer holiday homes from £20pp -including treehouses & lighthouses [Hopkins 2021]
- French training (1214 words)
  - Rencontres RH : l'agacement est plus difficile à résoudre que le conflit [Rodier 2021]
  - Covid-19 : Olivier Véran confirme la réouverture des terrasses des cafés et restaurants le 19 mai [Le Monde 2021a]
- French test (1274 words)
  - Féminicide de Mérignac : une mission d'inspection relève [Le Monde 2021c]
  - Covid-19 dans le monde : le Brésil suspend la vaccination avec AstraZeneca pour les femmes enceintes [Le Monde 2021b]
- Polish training (1073 words)
  - Studentce wstrzyknięto za dużo dawek szczepionki Pfizer [Hlebowicz 2021]
  - Niemedyczni pracownicy szpitali jednak z dodatkiem za pracę przy Covid-19 [Kowalska 2021c]
- Polish test (1459)
  - Aptekarze dostaną za szczepienia mniej niż lekarze [Kowalska 2021a]
  - Czy Unia Europejska poprze centralizację szpitali w Polsce [Kowalska 2021b]
  - Nowe obostrzenia covidowe - jest jednolity tekst rozporządzenia [Kuraś 2021]

As shown in the list, for each of the language classes: Economist, Sun, French, and Polish, we group the resultant word lists into two sets: *training* and *test*. The specific articles used to form word lists for each set are noted in the above list, alongside exactly how many words are in each set. In order to reach a target of 1,000 words for both training data and testing data we combined words drawn from multiple articles listed above. All together this yields eight data sets with words from four different sources.

Note that we use slightly different numbers of articles from different publications to gain an equivalent number of words from each one to reach our 1,000 word target (as each article length varies between different sources). In order to focus on the different tones of the publications we perform stop word removal on the text, leaving words that represent the content of each sentence. For English we remove stop words using the default Gensim stop word list in Python; we treat the French and Polish data in the same manner with language-appropriate stop word removal.

6.2 Experimental Set Up

Throughout this study we use our length-static fitness function using the default parameters noted in 5.1, and instead vary the way in which we train our GI system. Our data sets are used in different ways during GI to examine the effects of synthesis or sampling of training data.

In all cases our test data sets, used for performance testing without affecting fitness, are formed from the first 1,000 words of the respective data sources as shown in the above list; this selection does not change across the different studies.

For our training data sets, which inform fitness scores and so shape how the improved source code evolves across generations, we use four different approaches which we term Static, Sampling, Synthesis, and Mixed Sampling/Synthesis. In each generation of improvement, 1,000 keys are issued to the `put()` and `get()` functions of the hash table implementation for each population member, with the time taken to complete that of operations taken as the fitness score. The way in which these 1,000 training keys are derived for each generation works as follows:

Static. In this approach the set of keys used for training data is taken in the same way every generation, so there is no change in the data used for training over time. This data is formed from the first 1,000 words of the training data set we are attempting to specialize towards.

Sampling. Here the set of keys used for training data is randomly selected from the whole file of words, with a uniform weighting on all available words. This gives a different selection of words in each generation, but because we need 1,000 keys per generation this still entails significant overlap in keys used *between* each generation as there are less than 2,000 words in each word list. This sampling approach is a common solution used in machine learning to prevent over-fitting when there is plenty of data to sample from, and serves as a useful comparison to our synthesis approach.

Synthesis. Here the data is analysed for the distribution of word lengths, and frequency of letter usage in each word. These distributions are then used to randomly produce 1,000 strings with the same length and letter frequency distribution. This means the genetic improvement algorithm is no longer training on real words from a given language but on different random strings each generation. In emergent software systems, where we can only sample small amounts of real-world data, this synthesis method is a possible solution to creating a more robust training set for genetic improvement based on only a small amount of available “real" data.

Mixed Sampling and Synthesis. Here we mix together sampled with synthesized data to form our 1,000 keys per generation. We sample from the real word lists as above to provide 500 words per generation, which will have less overlap between generations than in the above sampling. This sample is then analysed for our length and letter distributions and the other 500 words are synthesized as above. This means our data set now includes both real words and entirely new string data at each generation. This serves as a comparison point to help understand how mixtures of sampled and synthesized training data may affect the outcome of genetic improvement, compared to pure sampled or synthesized training data.

Our evaluation examines how sampled and synthesised training data affects the outcomes of genetic improvement – and specifically whether synthesised data is a viable replacement to gain equivalent results. To do this we consider how well our GI process can *specialise* a hash table implementation towards either the language or class of data, and how the use of synthesised training data affects this specialisation capability. As an example, an ideal improved implementation for English words operates very well on both the training and test data for The Sun and Economist, but operates poorly on data from other languages. This would demonstrate that, despite limited

trace data from the emergent system, a candidate has specialised to the general case of English words, rather than specialising only to its limited training data.

6.3 Results

We report results in three categories: (i) effectiveness of training on different data choice methods, as measured by fitness to the training data; (ii) the performance of the trained systems of *unseen* data from the same source (testing generalisation to a class, rather than over-specialisation to training data); and (iii) the performance of different training regimes on different unseen data from each language (testing *over-generalisation* to all environments).

We start with the training performance shown in Figure 8. Training on real world data achieves higher optimisation of 10-25%. Here we see that for the Economist data the most effective training was using the fully synthesised data, with the partially synthesised data a close second. This is a positive result for our hypothesis that equivalent results can be achieved from synthesised data rather than a large data source or a sample of a larger data set. This trend is also seen with the French and Sun data. The Polish trained data also gets its best results from the synthesised data, while the mixed data training is slightly worse than the sampled data.

We also noted in this figure that the different language types do appear to have different properties, as the training runs level off at different fitness scores and do so at different generations. The Polish experiment in particular appears to be a difficult language to generate a good hash function for as it takes longer to reach a consistent fitness level and does not optimise as well as the other experiment data sets.

Conversely, the Economist data has the effect of being easier to optimize to, with on average 5% greater optimization than the others.

This training is only useful if the optimization to the *general class* of data is correct. To determine this, every 5 generations we test our population against unseen data from the same class. The results of this are shown in Figure 9.

We can confirm from these results that the Economist data class is easy to optimize to as the performance on unseen data is even better that the optimization on the training data, at 40% improvement. We see that the static trained populations slightly outperform the synthesized and mixed populations in performance on unseen data, but the differences are very small and still support both methods as good alternatives in case of small training data sets.

The French data shows the synthesised data trained populations performing best but again the performances on the unseen data are very similar across all four methods. The Polish data populations all perform poorly, supporting our speculation that Polish is particularly hard to train on with only a 10% optimization.

The Sun data shows notably higher difference between different training regimes, with the synthesis and mixed data yielding very poor results and the sample and static regimes performing better on unseen data. This is because the Sun class training data has an abnormal word length distribution, Figure 14. Data samples with abnormalities from the general data class are a common issue in sampled real world data which we discuss further in Section 6.4.3

Finally, in order to examine whether these systems are finding *over-general* solutions, rather than specialising to a data class, we consider their performance (tested on unseen data once every 5 generations) on the different language classes. For useful specialisation we would expect to see better performance on the class they were trained on, with worse performance on unfamiliar classes of data. This matches our motivation of synthesising good candidates for each environment encountered by an emergent software system. If, however, improved candidates are over-generalising they will uniformly perform well on easier data classes (Economist) and worse on harder classes (Polish).

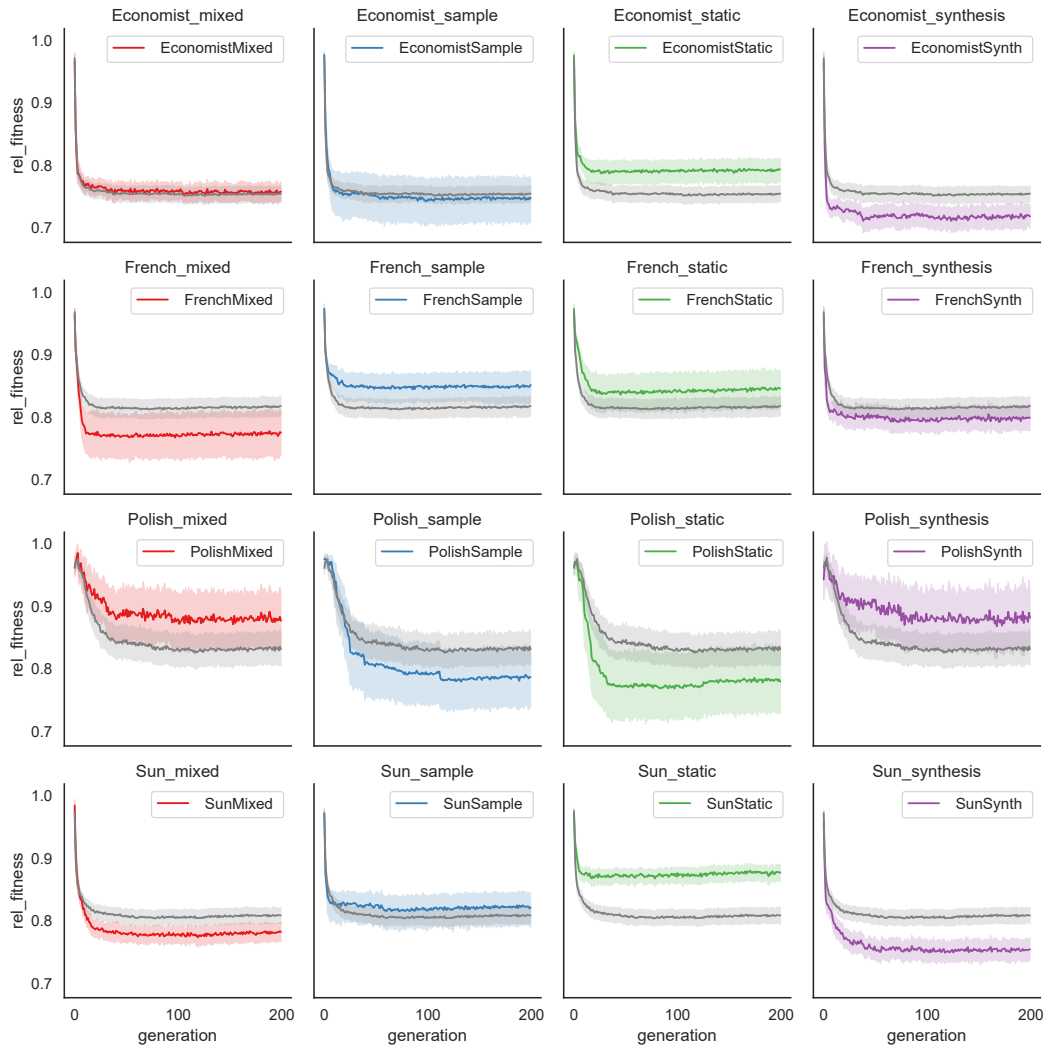

Fig. 8. Results of training for all algorithms with all languages as training data. Grey lines show average results of training for each language. The graph only shows the fitness of the best individual in a population, for each run and in each generation.

The results of testing against the Economist data are given in Figure 10. The gray lines on these plots show the average performance for all training set ups on the language.

We can see that all Economist trained candidates perform better than average on their general class, suggesting some level of useful specialization, with the same being true for the French trained populations. It could be possible then that both of these configurations simply produce better generalists than the other training configurations – though we can also see that the economist data is easy to hash with an average of 35% improvement by generation 100.

The Polish trained populations do better on the Economist class than they did on their own, supporting the idea of some generalisation and that the Economist class is easier to optimize to but

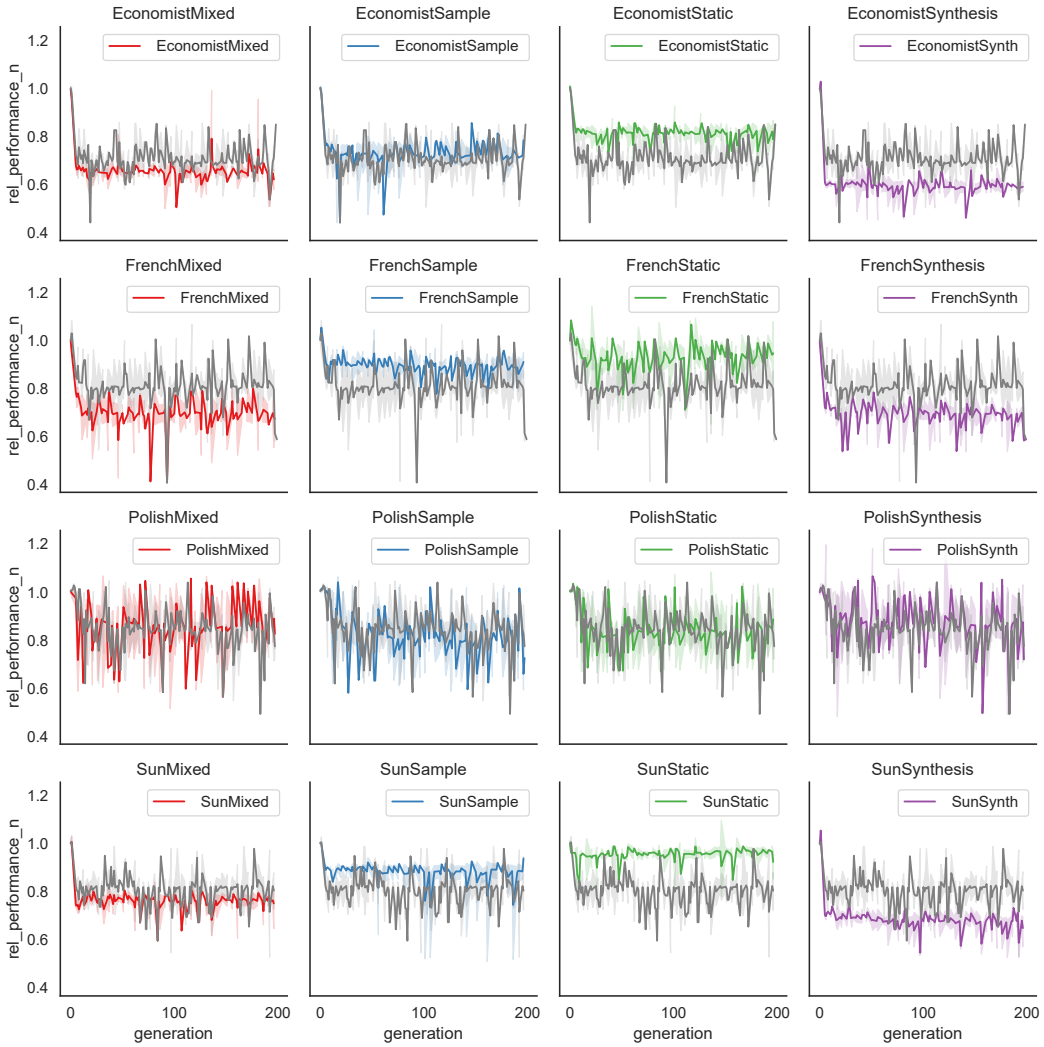

Fig. 9. Results for all training setups when tested on unseen data from the same class of data. The grey lines show the average performance for that class of data.

also supporting the idea that the Polish trained populations may be worse than the Economist and French outcomes on the Economist class due to some specialization.

The Sun populations trained on static or sampled data also do well, with the synthesised data doing worse, supporting the theory that they may have trained to an unusual data set and specialised to that rather than generalising to a normal language letter and length distribution.

The performance against French data is given in Figure 11 and is not as impressive with an average improvement of 15%. Again we see the Economist and French populations performing similarly. Their synthesis trained populations do better than the sample trained, suggesting agreeing that these methods are valid alternatives. It also suggests that both these systems have not specialised to their data classes but have both generalised.

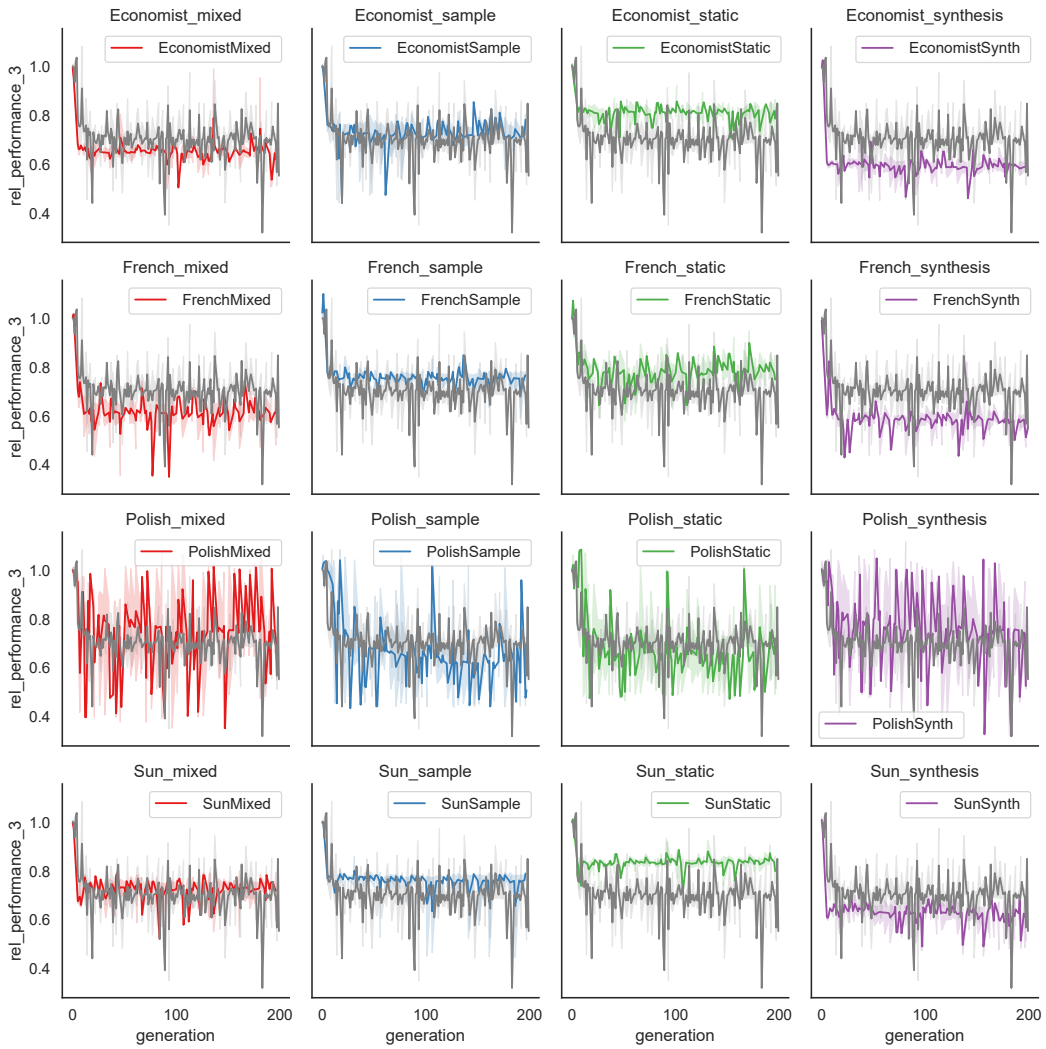

Fig. 10. Testing against the unseen Economist class data every 5 generations for each language and data preparation. The grey lines show the average performance on the Economist class data.

Interestingly the specialised Polish trained populations perform well on the french data class. Better that the French trained data. This suggests that it has specialised on features that are shared between French and Polish. Possibly due to both having an expanded alphabet.

The Sun populations perform similarly to the economist and french sets for the sample and static trained populations. This is not the case for the specialised synthesis and mixed systems.

The results of testing against Polish class data Figure 12 show that Polish is a harder class with the average improvement 1-2%. It confirms the theory that the Polish trained populations have specialised to work with the Polish class data. They far outperform all other populations. It also confirms the difficulty of the Polish data class. All the generalized optimisations of the Economist, French and Sun populations seem to perform worse on the Polish data. They all level out worse

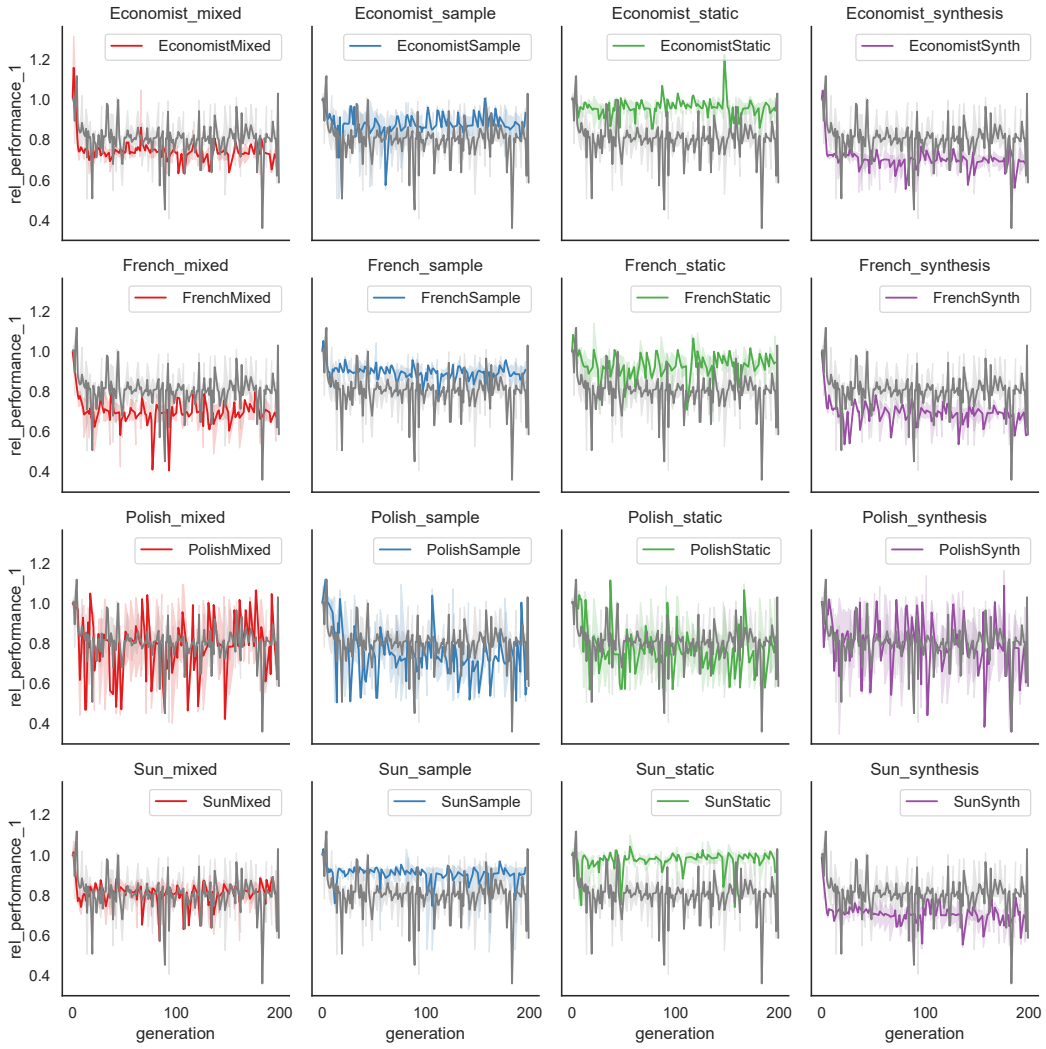

Fig. 11. Testing against the unseen French class data every 5 generations for each language and data preparation. The grey lines show the average performance on the French class data.

than the starting values after optimizing to different data classes. This is particularly true of our training data specialist the mixed and synthesis populations of the Sun class.

Lastly the results of testing against the Sun class data are given in Figure 13 with the second best average improvement at just over 20%. It suggests there is nothing strange about the unseen performance data for the Sun class as our generalists all perform well (economist, french and sample and static sun populations) while our specialists such as the mixed Polish trained population struggle more. This includes the Sun-trained populations which used synthesis and seem to specialised to the point of over fitting to the training data. This also indicates that the Sun class testing data is easy to optimize to as so many of the generalist perform so similarly on it.

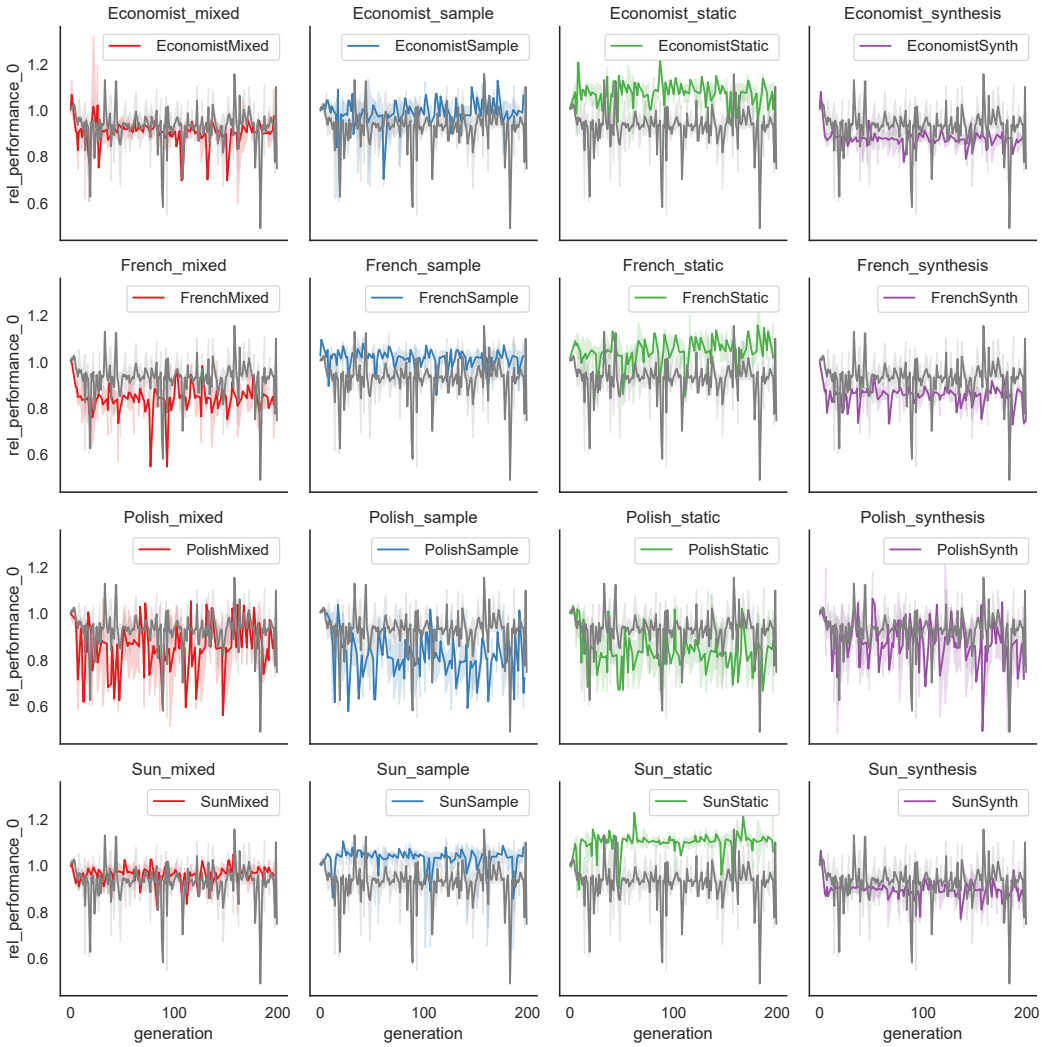

Fig. 12. Testing against the unseen Polish class data every 5 generations for each language and data preparation. The grey lines show the average performance on the Polish class data.

We test all results using a signed-rank Wilcoxon test to gauge statistical differences in the final trained individuals, again using values below 0.05 as indicators of statistical significance. These are presented in full in Appendix A. In terms of the data treatments training we see significant differences in almost all cases between the Mixed and Synthesis variants (e.g., French Mixed v Static p=0.00927, Synth v Sample p=0.000831) and between the Sample and Static variants. The data treatments with and without synthesized data are generally similar (French Mixed v Synth p=0.371, Sample v Static p=0.750) with small effect sizes when they are different (exception Static and Sample trained on Sun).

Looking at the final individuals, in terms of performance on *different* languages, we see a strong pattern of Polish finalists being different from the others, including when testing against Polish; the

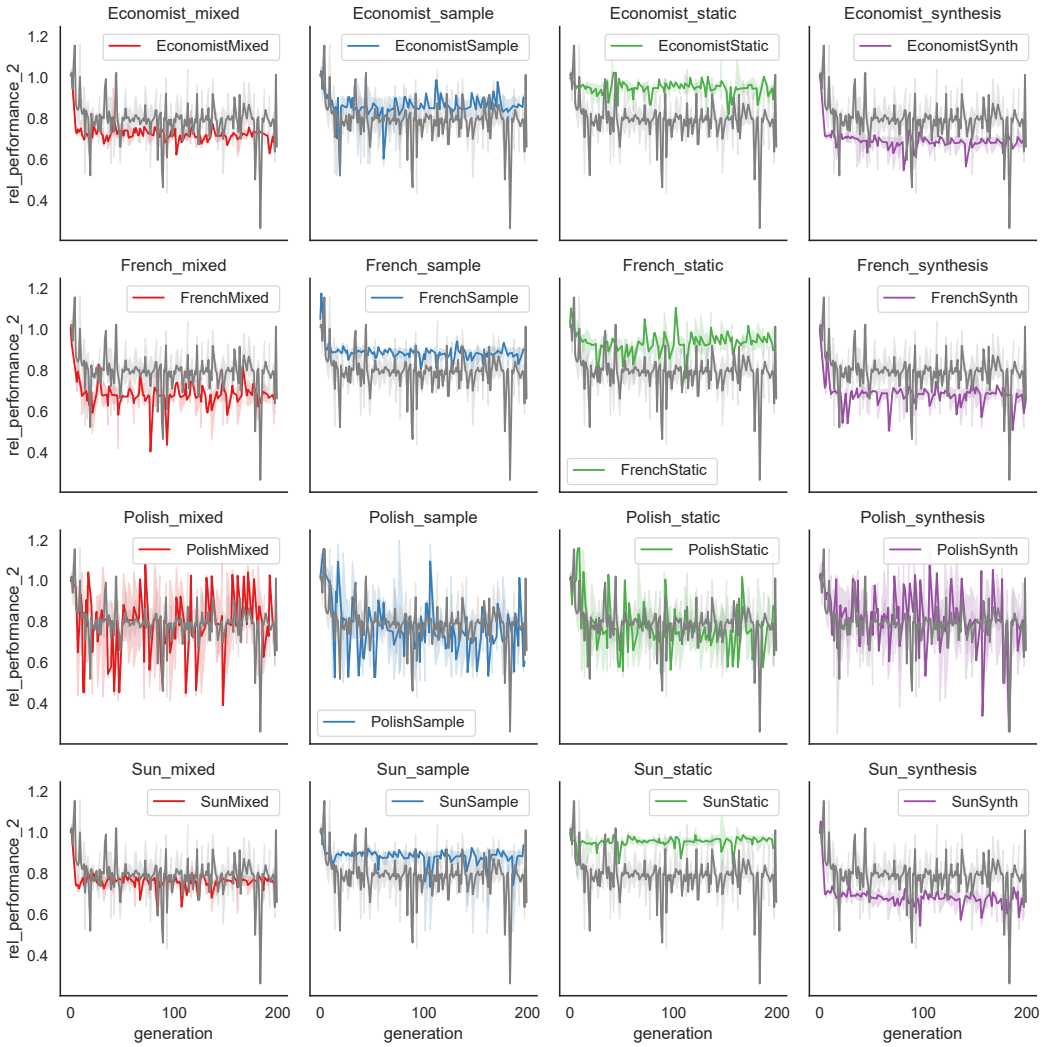

Fig. 13. Testing against the unseen Sun class data every 5 generations for each language and data preparation. The grey lines show the average performance on the Sun class data.

highest p-value for Polish v Economist, or French systems on Economist data, was p=0.0175. The mixed data treatment is the main difference between the Sun and other systems, with a significant difference between Sun and the Economist and French systems on all data sets. Finally the French and Economist systems perform the same on almost all systems, with only two significant results and the lowest p-value otherwise p=0.136.

6.4 Discussion

This study has revealed three different results of training on a single language: generalists, high-utility specialists, and over-fitting to the training data. Each of these will effect the use of the

resultant function in a dynamic deployment environment where it may see unexpected data such as from a different language, or class within a language.

6.4.1 Generalists. The development of generalists indicates training data that provokes optimisation towards most data, rather than data from a particular environment. We view this as a mostly positive result; while a generalist might not be as optimal as a specialist, it will better survive environment change and be of some utility until retaining has created a new specialist.

In detail, almost all evolutionary optimization is likely to produce *some* general optimization, as well as specialization. This is often a positive feature that could be used to advantage in an emergent systems context: if, over the course of multiple retrainings, we can identify generalist optimizations we can then potentially improve the base (pre evolution) population to include those enhancements. This offers good general-purpose behaviour for unseen environments as and when they arise, but does not prevent those optimizations being removed in future to allow for specialization.

6.4.2 Specialists. The specialists shown here, in particular those trained on the Polish data, emerge from exposure to a hard class on which the generalist optimisations are less effective. While still performing worse on unseen Polish data than most other systems on familiar class data, it is the only improved version that achieves better than starting results on Polish.

This kind of specialisation is one of the core reasons to use evolutionary algorithms and retraining in an emergent software system. We cannot easily predict the existence of these classes of data prior to deployment a system, and encountering them unexpectedly can strongly effect the performance of a system if there is no mechanism to create improvements. By retraining using an evolutionary technique we can develop a solution from sparse sample data captured at runtime. The interaction between this capability, and the ideal method of improving the base population as a curated genetic pool over time, is a topic of future work.

6.4.3 Over fitting. Our Sun-trained populations did not behave as expected on the unseen data. Based on the results in Figure 8, the populations have all successfully trained to the data with similar curves to those trained by other language classes.

However, the populations which have trained on synthesised data perform as well on the unseen data. This is consistent and statistically significant across all unseen data (again using a signed-rank Wilcoxon test). This suggests that the training over-fitted to the synthesised data, also suggesting that some aspect of the synthesised data was different from the raw data.

On further investigation we find that the training data for the Sun has a particularly high number of single letter words (6% in the training data vs 3.1% in the testing data). This is also true in the synthesised data, and the sampled and static data sets drawn from it. The functions produced by training on this data have a strong preference for additional increments in the hash function's for-loop. These operations may assist in ensuring that single-letter keys are better distributed across hash buckets. The challenges appears to arise in that the single letter words in the random strings of our synthesis model come from the entire character set of English, while in the sampled or static data single letters in English generally fall into the set: a, i, r, s, t (where r, s, and t are a result of our stop-word removal approach from words such as "don't"). Because this bias is not present in the synthetic data, the improved versions are good at distributing single-letters uniformly, but may cluster the letters {a, i, r, s, t} in similar buckets (r is included in this list as one of the articles used was about covid infections and referenced r values heavily). This will be a fragile solution when the single letter words are from a different distribution, or are not as common.

To confirm this, we training with the performance data set and check again the original training set; when doing this, we find that the full synthesis system still specialises but does not over fit based on its relative performance to the other systems (Figure 15).

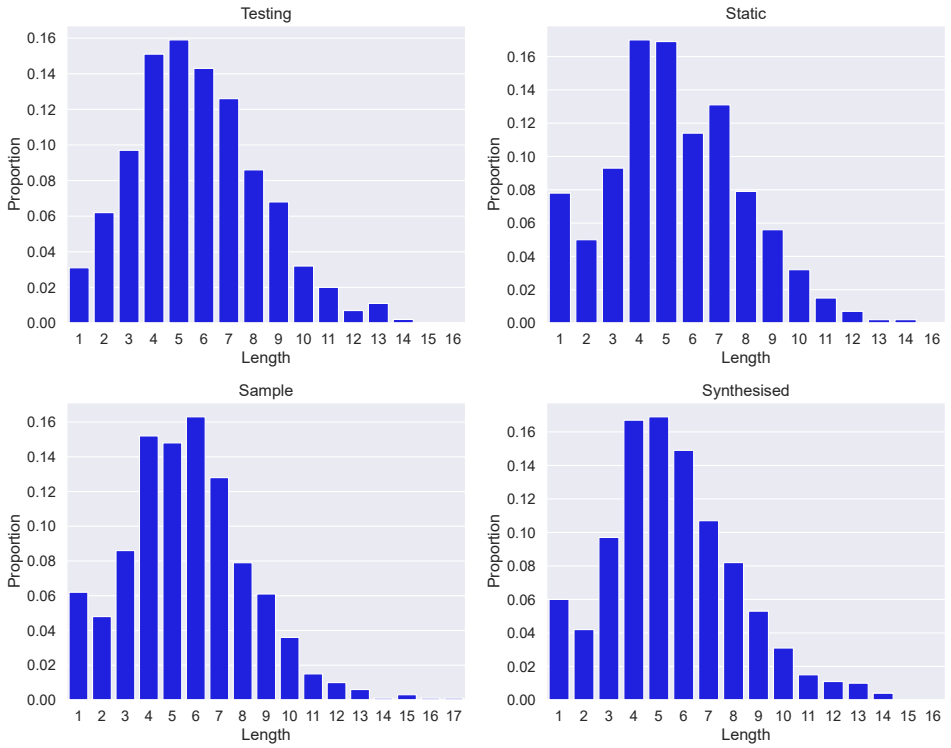

Fig. 14. Histograms of length distributions for (from top to bottom) the testing data, static training data, sampled training data, and synthesised training data for the Sun class.

This identifies a very specific issue with our synthesis method, and this data set. Correcting it requires a more tailored synthesis model which better captures the underlying trends in the data, such as by considering frequency of letters on a per-word-length basis, or treating short words as coming from a preset dictionary of words for synthesis.

In general, to avoid problems like this we need to check if a sample of data is representative of ongoing reality. This is a particularly challenging issue as it requires us to differentiate between a bad sample and a real change in the data to a different environment or class. This is an interesting direction for future investigation.

7 CONCLUSIONS

Synthesis is a powerful tool for augmenting GI systems with small amounts of code and data. Insertion of synthesised code allows us to optimise initial candidates, by as much as 40%, that have very little genetic material, and also creates more robust evolutionary systems that work on small code fragments by maintaining a stable level of genetic material which can guard against the probability of “burn out" of a population. Our goal in future is to improve the sophistication of our synthesised insert code to increase its utility to the function and find parameter setting based on initial code and data which will help to achieve more consistently strong optimisation.

Our overall approach demonstrates that we can generate high-utility generalists and specialists, offering good indicators that the system can adapt to change and remain useful during retraining, as well as adapt to new challenges in the deployment environment. This makes them perfect as

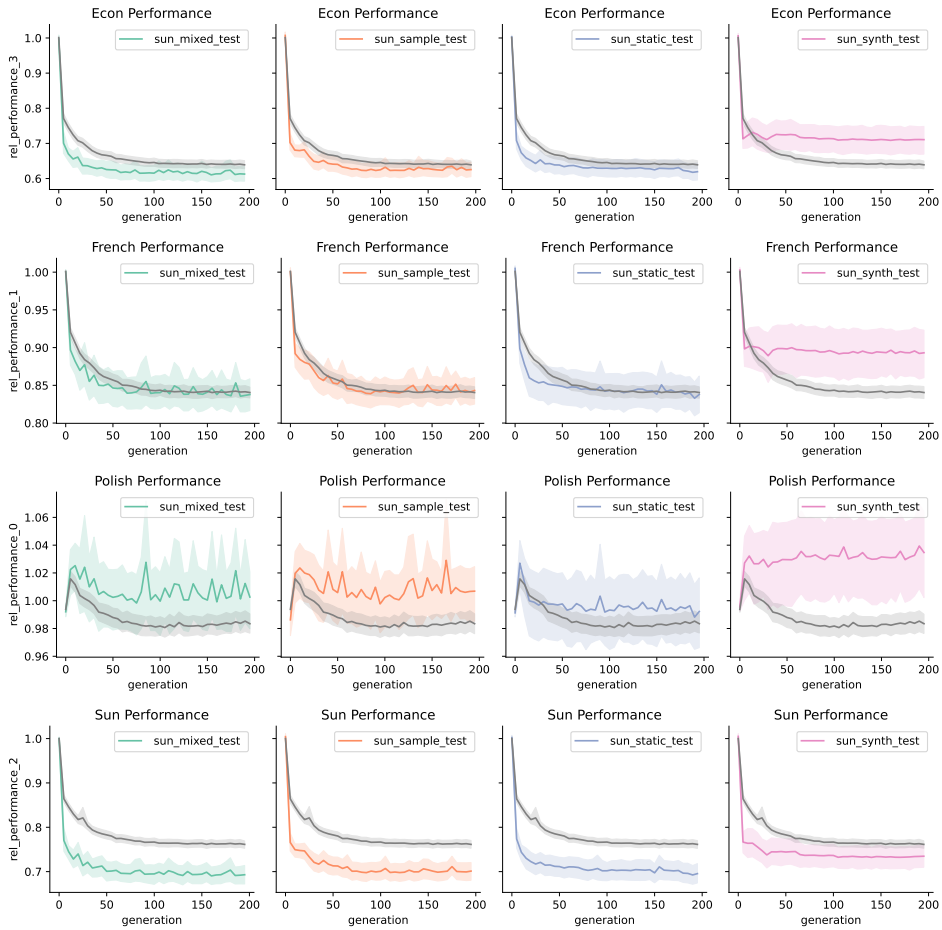

Fig. 15. Training with the Sun test data with the normal length distribution tested against original test sets for the Economist, French and Polish data, and the original training data for the Sun.

variants to be changed in and out of an emergent system in response to changes in the deployment environment.

The use of synthesised training data for GI systems can add to or replace sparsely sampled data for training in low data environments to give results equivalent to those for methods with more data available. Our results there demonstrate that, if the sampled data used to synthesise the new data is representative of the class we wish to fit to, synthesis can help to prevent over-fitting to the sample. There is, however, still work to be done here on how to shape a system that is robust to bad samples without being prevented from identifying change in the system.

8 ACKNOWLEDGEMENTS

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

REFERENCES

2021a. Are vaccine passports a good idea? https://www.economist.com/science-and-technology/2021/03/13/are-vaccine-passports-a-good-idea

2021b. Covid-19 vaccines have alerted the world to the power of RNA therapies. https://www.economist.com/briefing/2021/03/27/covid-19-vaccines-have-alerted-the-world-to-the-power-of-rna-therapies

Hamad Alhammady and Kotagiri Ramamohanarao. 2005. Expanding the Training Data Space Using Emerging Patterns and Genetic Methods. In *Proceedings of the 2005 SIAM International Conference on Data Mining (SDM)*. Society for Industrial and Applied Mathematics, 481–485.

Claudia Aoraha. 2021. EU to agree Covid passport scheme allowing vaccinated Brits to travel. https://www.thesun.co.uk/travel/14363494/covid-passport-eu-holiday-summer-britain/

A Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. ieeexplore.ieee.org, 162–168.

Thomas Bäck and Hans-Paul Schwefel. 1993. An overview of evolutionary algorithms for parameter optimization. *Evol. Comput.* 1, 1 (March 1993), 1–23.

Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. (Nov. 2014), 306–317.

Liming Chen and Algirdas Avizienis. 1978. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, Vol. 1. 3–9.

Charles Darwin. 1859. *The Origin of Species* (penguin classics, 1985 ed.). John Murray.

Paul Dean and Barry Porter. 2021. The Design Space of Emergent Scheduling for Distributed Execution Frameworks. In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 186–195. https://doi.org/10.1109/SEAMS51251.2021.00032

Roberto Rodrigues Filho, Marcio Pereira de Sá, Barry Porter, and Fábio M. Costa. 2018. Towards Emergent Microservices for Client-Tailored Design. In *Proceedings of the 19th Workshop on Adaptive and Reflexive Middleware* (Rennes, France) *(ARM '18)*. Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3289175.3289177

Stephanie Forrest, Thanhvu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation* (Montreal, Québec, Canada) *(GECCO '09)*. Association for Computing Machinery, New York, NY, USA, 947–954.

A A Freitas. 2009. A review of evolutionary algorithms for data mining. *Data Mining and Knowledge Discovery Handbook* (2009).

Alice Fuller. 2021. Monaco's Prince Albert slams Harry & Meghan for 'inappropriate' Oprah interview. https://www.thesun.co.uk/news/14462468/prince-albert-monaco-slams-harry-meghan-oprah/

Bartoz Hlebowicz. 2021. Studentce wstrzyknięto za dużo dawek szczepionki Pfize. *wyborcza.pl* (May 2021). https://wyborcza.pl/7,75399,27069101,studentce-wstrzyknieto-przez-pomylke-szesc-dawek-szczepionki.html

Hannah Hopkins. 2021. Quirky UK summer holiday homes from £20pp - including treehouses& lighthouses. https://www.thesun.co.uk/travel/14431435/quirky-uk-properties-summer/

J.O. Kephart and D.M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50. https://doi.org/10.1109/MC.2003.1160055

Zoltan A Kocsis, Geoff Neumann, Jerry Swan, Michael G Epitropakis, Alexander E I Brownlee, Sami O Haraldsson, and Edward Bowles. 2014. Repairing and Optimizing Hadoop hashCode Implementations. In *Search-Based Software Engineering*. Springer International Publishing, 259–264.

Karolina Kowalska. 2021a. Aptekarze dostaną za szczepienia mniej niż lekarze. *Rzeczpospolita* (May 2021). https://www.rp.pl/prawo-dla-ciebie/art146801-aptekarze-dostana-za-szczepienia-mniej-niz-lekarze

Karolina Kowalska. 2021b. Czy Unia Europejska poprze centralizację szpitali w Polsce. *Rzeczpospolita* (May 2021). https://www.rp.pl/prawo-dla-ciebie/art141111-czy-unia-europejska-poprze-centralizacje-szpitali-w-polsce

Karolina Kowalska. 2021c. Niemedyczni pracownicy szpitali jednak z dodatkiem za pracę przy Covid-19. *Rzeczpospolita* (Apr 2021). https://www.rp.pl/prawo-dla-ciebie/art8602191-niemedyczni-pracownicy-szpitali-jednak-z-dodatkiem-za-prace-przy-covid-19

Jagoda Kuraś. 2021. Nowe obostrzenia covidowe - jest jednolity tekst rozporządzenia. *Rzeczpospolita* (Feb 2021). https://www.rp.pl/prawo-dla-ciebie/art8660791-nowe-obostrzenia-covidowe-jest-jednolity-tekst-rozporzadzenia

William B Langdon and Mark Harman. 2014. Genetically Improved CUDA C++ Software. In *Genetic Programming*. Springer Berlin Heidelberg, 87–99.

William B Langdon and Oliver Krauss. 2021. Genetic Improvement of Data for Maths Functions. *ACM Trans. Evol. Learn. Optim.* 1, 2 (July 2021), 1–30.

Le Monde. 2021a. Covid-19 : Olivier Véran confirme la réouverture des terrasses des cafés et restaurants le 19mai. *Le Monde* (May 2021). https://www.lemonde.fr/planete/article/2021/05/10/covid-19-olivier-veran-confirme-la-reouverture-des-

terrasses-des-cafes-et-restaurants-le-19-mai_6079722_3244.html

Le Monde. 2021b. Covid-19 dans le monde : le Brésil suspend la vaccination avec AstraZeneca pour les femmesenceintes. *Le Monde* (May 2021). https://www.lemonde.fr/planete/article/2021/05/11/astrazeneca-l-union-europeenne-reclame-en-justice-les-90-millions-de-doses-non-livrees-au-premier-trimestre_6079871_3244.html

Le Monde. 2021c. Féminicide de Mérignac : une mission d'inspection relève. *Le Monde* (May 2021). https://www.lemonde.fr/societe/article/2021/05/12/feminicide-de-merignac-une-mission-d-inspection-releve-une-serie-de-defaillances-dans-le-suivi-du-mari-violent_6079956_3224.html

Ding Li, Angelica Huyen Tran, and William G J Halfond. 2014. Making web applications more energy efficient for OLED smartphones. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 527–538.

Mario Linares-Vásquez, Gabriele Bavota, Carlos Eduardo Bernal Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2015. Optimizing energy consumption of GUIs in Android apps: a multi-objective approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 143–154.

Yang Long, Li Liu, Fumin Shen, Ling Shao, and Xuelong Li. 2018. Zero-Shot Learning Using Synthesised Unseen Visual Data with Diffusion Regularisation. *IEEE Trans. Pattern Anal. Mach. Intell.* 40, 10 (Oct. 2018), 2498–2512.

Christopher McGowan, Alexander Wild, and Barry Porter. 2018. Experiments in genetic divergence for emergent systems. In *Proceedings of the 4th International Workshop on Genetic Improvement Workshop* (Gothenburg, Sweden) *(GI '18)*. Association for Computing Machinery, New York, NY, USA, 9–16.

Ian O'Hara, James Paulos, Jay Davey, Nick Eckenstein, Neel Doshi, Tarik Tosun, Jonathan Greco, Jungwon Seo, Matt Turpin, Vijay Kumar, and Mark Yim. 2014. Self-assembly of a swarm of autonomous boats into floating structures. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*. ieeexplore.ieee.org, 1234–1240.

Justyna Petke, Saemundur O Haraldsson, Mark Harman, William B Langdon, David R White, and John R Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Trans. Evol. Comput.* 22, 3 (June 2018), 415–432.

Justyna Petke, William B Langdon, and Mark Harman. 2013. Applying Genetic Improvement to MiniSAT. In *Search Based Software Engineering*. Springer Berlin Heidelberg, 257–262.

Harry Pettit. 2021. Elon Musk changes his official job title to 'Technoking' of Tesla. https://www.thesun.co.uk/tech/14352831/elon-musk-tesla-job-title-technoking/

Barry Porter. 2014. Runtime modularity in complex structures: a component model for fine grained runtime adaptation. In *Proceedings of the 17th international ACM Sigsoft symposium on Component-based software engineering* (Marcq-en-Bareul, France) *(CBSE '14)*. Association for Computing Machinery, New York, NY, USA, 29–34.

Barry Porter and Roberto Rodrigues Filho. 2021. A Programming Language for Sound Self-Adaptive Systems. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. ieeexplore.ieee.org, 145–150.

Barry Porter, Matthew Grieves, Roberto Rodrigues Filho, and David Leslie. 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)*. usenix.org, 333–348.

Penelope Rainford. 2022. *Replication package for experiments in this paper.* http://www.projectdana.com/research/telo2022rainford

Anne Rodier. 2021. Rencontres RH : l'agacement est plus difficile à résoudre que le conflit. *Le Monde* (May 2021). https://www.lemonde.fr/emploi/article/2021/05/12/rencontres-rh-l-agacement-est-plus-difficile-a-resoudre-que-le-conflit_6079952_1698637.html

Roberto Rodrigues-Filho and Barry Porter. 2022. Hatch: Self-distributing systems for data centers. *Future Generation Computer Systems* 132 (2022), 80–92. https://doi.org/10.1016/j.future.2022.02.008

M Rognant, C Cumer, J M Biannic, M Roa, and others. 2019. Autonomous assembly of large structures in space: a technology review. *EUCASS* (2019).

Mazeiar Salehie and Ladan Tahvildari. 2009. Self-Adaptive Software: Landscape and Research Challenges. *ACM Trans. Auton. Adapt. Syst.* 4, 2, Article 14 (may 2009), 42 pages. https://doi.org/10.1145/1516533.1516538

Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 43–54.

Alexandros Stergiou, Grigorios Kalliatakis, and Christos Chrysoulas. 2018. Traffic Sign Recognition based on Synthesised Training Data. *Big Data and Cognitive Computing* 2, 3 (July 2018), 19.

Jim Torresen. 2002. A Dynamic Fitness Function Applied to Improve the Generalisation when Evolving a Signal Processing Hardware Architecture. In *Applications of Evolutionary Computing*. Springer Berlin Heidelberg, 267–279.

Craig Underwood, Sergio Pellegrino, Vaios J Lappas, Christopher P Bridges, and John Baker. 2015. Using CubeSat/micro-satellite technology to demonstrate the Autonomous Assembly of a Reconfigurable Space Telescope (AAReST). *Acta*

Astronaut. 114 (Sept. 2015), 112–122.

Chunli Wang, Xilin Chen, and Wen Gao. 2006. Expanding Training Set for Chinese Sign Language Recognition. In *7th International Conference on Automatic Face and Gesture Recognition (FGR06).* 323–328.

Terri-Ann Williams. 2021. Official Covid R rate creeps up AGAIN - but still hovers below crucial 1. https://www.thesun.co.uk/news/14465292/covid-r-rate-creeps-up-again-hovers-below-1/

A STATISTICAL TEST RESULTS

	Length Dynamic	Speed Static	Speed Dynamic
Length Static	0.192	0.199	0.0148
Length Dynamic	-	0.614	0.704
Speed Static	-	-	0.289

Table 1. RQ1 Generation 194 fitness comparison for best individuals (the last generation with the same fitness function for both length variants)

Economist	Synth	Static	Sample
Mixed	**0.0195**	**0.0230**	0.781
Synth	-	**0.000136**	**0.0316**
Static	-	-	0.0571
Sun	**Synth**	**Static**	**Sample**
Mixed	**0.0218**	**3.88e-06**	**0.00129**
Synth	-	**1.92e-06**	**0.000771**
Static	-	-	**0.000894**
French	**Synth**	**Static**	**Sample**
Mixed	0.371	**0.00927**	**0.00258**
Synth	-	**0.0243**	**0.000831**
Static	-	-	0.750
Polish	**Synth**	**Static**	**Sample**
Mixed	0.629	**0.0207**	**0.00727**
Synth	-	**0.0111**	**0.00411**
Static	-	-	0.750

Table 2. RQ2 Comparison of distributions of final generation fitness between systems trained on the same language, statistically significant results shown in bold.

Unseen Economist Data	Mixed	Sample	Static	Synthesis
Sun v Economist	**3.41e-05**	**0.0387**	0.271	0.0598
Sun v French	**7.51e-05**	0.465	**0.00411**	**0.0428**
Sun v Polish	0.465	**0.000332**	**0.000136**	**0.0218**
Polish v Economist	**0.0175**	**0.00822**	**0.000125**	**0.00567**
Polish v French	**0.0148**	**0.000716**	**0.00226**	**0.00604**
French v Economist	0.199	0.141	**0.00927**	0.894
Unseen French Data	Mixed	Sample	Static	Synthesis
Sun v Economist	**0.000831**	0.237	0.504	0.688
Sun v French	**0.000106**	0.530	0.141	0.943
Sun v Polish	0.318	**1.80e-05**	**1.80e-05**	0.0786
Polish v Economist	0.280	**0.000490**	**1.97e-05**	0.136
Polish v French	0.131	**0.000174**	**0.000174**	0.120
French v Economist	0.165	0.781	0.229	0.959
Unseen Polish Data	Mixed	Sample	Static	Synthesis
Sun v Economist	**0.00385**	0.0598	0.362	0.229
Sun v French	**2.60e-05**	0.178	**0.0350**	**0.0387**
Sun v Polish	**0.00411**	**1.92e-06**	**3.88e-06**	0.558
Polish v Economist	0.165	**6.89e-05**	**4.73e-06**	0.688
Polish v French	0.734	**9.32e-06**	**6.99e-06**	0.766
French v Economist	**0.0175**	0.572	0.165	0.629
Sun Performance	Mixed	Sample	Static	Synthesis
Sun v Economist	**0.000490**	0.289	0.318	0.943
Sun v French	**0.000205**	0.453	0.116	0.975
Sun v Polish	0.600	**0.000283**	**4.86e-05**	**0.0132**
Polish v Economist	0.0752	**0.00411**	**6.89e-05**	**0.0132**
Polish v French	**0.0428**	**0.000222**	**0.000222**	**0.0256**
French v Economist	0.136	0.572	0.309	0.6889

Table 3. RQ2 Performance on final individuals on different languages compared with like data treatments and unlike training languages. Statistically significant results are shown in bold.