**Genetic Improvement: A Key Challenge for Evolutionary Computation**

William B. Langdon

Gabriela Ochoa

**Abstract**—Automatic Programming has long been a sub-goal of Artificial Intelligence (AI). It is feasible in limited domains. Genetic Improvement (GI) has expanded these dramatically to more than 100,000 lines of code by building on human written applications. Further scaling may need key advances in both Search Based Software Engineering (SBSE) and Evolutionary Computation (EC) research, particularly on representations, genetic operations, fitness landscapes, fitness surrogates, multi objective search and co-evolution.

I. INTRODUCTION

Genetic Algorithms were invented right at the start of Artificial Intelligence research in the hope they could harness the power of Darwin’s natural evolution [Darwin, 1859] so that we can get “computers to do what is needed to be done, without being told exactly how to do it” [Koza, 1992, page 1]. Natural selection and inheritable genetic variation acting over billions of years has seen life diversify and new species emerge and colonise every conceivable niche on the planet. Indeed geographic [Owen et al., 1990] or even man made environmental changes [Lack, 1968] have seen the evolution or diversification of species within a few hundred years. The hope that such rapid evolution might also take place within computer populations has in many cases been vindicated. Today Evolutionary Computing has been successfully applied to finding acceptable solutions (even optimal solutions) to a wide range of numerical problems within the computer and to the evolution of engineering structures and even art and music in the physical world. However, now more than twenty years after [Koza, 1992], Evolutionary Computing has had less success at evolving the programs which control our computers.

The success of Genetic Programming [Koza, 1992] is well known. It has evolved predictive models and classifiers [Pappa and Freitas, 2004; Bhowan et al., 2013] over a huge range of applications, from predicting human endeavours (finance [Neely and Weller, 1999]; Dempster and Jones, 2000; Tsang and Li, 2002); insurance [Langdon, 1999]) to the outcome of breast cancer [Langdon and Buxton, 2010] and drug research [Langdon and Barrett, 2004]. In engineering it has been used to save energy in steel manufacture [Kovacic and Sarler, 2014] and modelling microscopic particulate pollution [Kovacic et al., 2013]. It has been used to design electronic circuits [Koza et al., 1999]; Koza and Bennett III, 1999] radio antennas [Hornby et al., 2011]; Baker et al., 2010], to aid building architects [O’Reilly and Hemberg, 2007]; and in civil engineering [Baykasoglu et al., 2008; Najafzadeh and Barani, 2011; Mirzahosseini et al., 2011] and the environment [Savic et al., 1999; Brumby et al., 2001].

The next section introduces Genetic Improvement (GI) whilst Section III gives more details on how it uses Evolutionary Computing. Section IV asks how EC might further improve GI and we challenge EC to lead further down the road to Artificial Intelligence in general and to Automatic Programming in particular. Section V discusses the existing strengths and weakness of GI, whilst Section VI suggests ways that EC might improve it. Finally Section VII mentions some ways that GI researchers might help EC before we conclude (Section VIII).

II. PERSPECTIVE:

WHAT GENETIC IMPROVEMENT HAS DONE SO FAR

Although Evolutionary Computing has had many successes in software engineering [Harman, 2007], it has had less success generating new software. However it has recently started to be used for improving existing programs written by people. Genetic Programming has been used to automatically fix bugs [Arcuri, 2011]; Weimer et al., 2010]. Although still controversial in some software engineering circles, Weimer and Forrest [Forrest et al., 2009] kicked down the door and showed automatic bugfixing could be possible, whereas the software engineering community appear to have ignored the possibility until it was shown Evolutionary Computing could fix real bugs in real programs. (Albeit some bugs, not necessarily all of them.) Given this impetuous by EC,
automatic bug repair [Kessentini et al., 2011] is now a very active research topic in software engineering. Le Goues’ [Le Goues et al., 2012] prize winning work on bug fixing is far from the only prize winning work using genetic programming to improve existing code. Last year saw the first international workshop on genetic improvement and this year will see the second and also genetic improvement represented at this conference as well as various local GI events. As well as fixing errors, GI has been demonstrated speeding up code [Langdon and Harman, 2015b], generating [Langdon and Harman, 2010] and improving parallel code [Langdon and Harman, 2014] [Langdon et al., 2014] [Langdon et al., 2015] [Langdon and Harman, 2015a], automatically specialising programs [Petke et al., 2014], reducing energy [Schulte et al., 2014a] Bruce, 2015] and memory consumption [Wu et al., 2015]. Indeed there is great interested in multi-objective approaches where evolution might generate a Pareto front [Harman et al., 2012] allowing the designer to trade off different objectives (e.g. quality v. memory) indeed we might see the user being given the option of trading objectives according to circumstances. E.g. someone might want a mobile application’s full functionality whilst their phone was plugged in at home but be prepared to accept lower response in return for longer battery life when disconnected from a power supply during the day.

III. HOW GENETIC IMPROVEMENT WORKS

Although there are many approaches in detail, currently (see Figure 1) the general approach is to automatically pre-process the program source to give either a simple grammar or description at the AST level and use that description to constrain the genetic operations to ensure children in the next generation do not have syntax errors. The individuals in the evolving population are typically changes to the software to be evolved. This makes them more compact. This is a bit like seeding the population [Langdon and Nordin, 2000] with the human written code rather than starting with a totally random population. Although typically the syntax is correct, a sizeable fraction of the mutants may not compile. This is almost always due to variables being moved out of scope. In some cases the mutations may be restricted to respect variable scope limits, allowing 100% of children to compile. Typically how good a child is (i.e. its fitness) is found by compiling it, running it and then comparing its results with the original hand written code.

Great progress has been made with using Evolutionary Computation to generate test suite for programs [Fraser and Arcuri, 2011]. It is common to be able to generate test cases that will cause the software under test to execute most of the paths within it (NB. not all combinations). However in the classic software engineering case, the automatic tests say how to run the program but have no way of knowing if it produced the right answer. (In software engineering this is known as the “Oracle Problem”). But notice how Evolutionary Computing side-steps the Oracle Problem. We have an oracle. We have the original code. Admittedly in some cases (e.g. bug fixing) the original code’s answer may need some adjustment. But in many cases it is sufficient. We just want the improved code to give the same answer but be faster, take less resources, etc. As long as the fitness function can automatically qualitatively say if the code is better or not we can in principle automatically evolve code in the right direction. Even in the case of multi-objective evolution (MOEA), it may be possible to automatically score several objectives. E.g., as well as the program’s run time, it may be possible to calculate the quality of a mutant’s answer and compare it with the quality of the original code’s answer. Thus allowing a MOEA to automatically evolve mutants of the original hand written code.

Running the mutated code is like the well established software engineering tool of mutation testing [Langdon et al., 2010] (also called mutation analysis). As with mutation testing, typically you need to protect your system against badly behaved mutants, Therefore it is common to use either CPU or elapsed time limits to force termination. (Aborted programs seldom get high fitness). You may want to use some form of protection against mutants accessing data outside the bounds of arrays. Depending upon the range of allowed mutations, you may want to use virtual machines or some form of sandboxing to prevent damage by rogue mutants.

As with mutation testing, and indeed Evolutionary Algorithms (EAs), typically GI run time is dominated by fitness testing. Also sometimes the time taken to compile the mutants is also important. You may want to use Unix’ make to ensure only the modified code is recompiled, pre-compile libraries (e.g. C .h files) [Langdon and Harman, 2015b], compile the whole population of mutants in one operation [Langdon et al., 2015], or compile the population using multiple computers [Harding and Banzhaf, 2009].

IV. THE AUTOMATIC PROGRAMMING CHALLENGE TO EVOLUTIONARY COMPUTING

Artificial Intelligence has started to achieve impressive results. Twenty years ago IBM’s deep blue showed traditional AI can play chess better than every human on the planet. In the last few years neural network based deep learning has made great strides and we now have cars driving themselves. Although improving, these still suffer from the the AI being hand built for a task. Evolutionary Computing side-steps the Oracle Problem. We have the original code. Admittedly in some cases (e.g. bug fixing) the original code’s answer may need some adjustment. But in many cases it is sufficient. We just want the improved code to give the same answer but be faster, take less resources, etc. As long as the fitness function can automatically qualitatively say if the code is better or not we can in principle automatically evolve code in the right direction. Even in the case of multi-objective evolution (MOEA), it may be possible to automatically score several objectives. E.g., as well as the program’s run time, it may be possible to calculate the quality of a mutant’s answer and compare it with the quality of the original code’s answer. Thus allowing a MOEA to automatically evolve mutants of the original hand written code.

2Evolution at the level of Java byte code and Sipper, 2011] or indeed machine code [Lukschandl et al., 1998] [Orlov and Sipper, 2011] or indeed machine code [Schulte et al., 2014b] [Schulte et al., 2015] may also be possible.
improvements on those and so on, we might escape from the symbolic regression local attractor. However realistically what GI gives you is more like trying to evolve a human from an ape, rather than starting with a single celled amoeba. Genetic Improvement does not give human intelligence but it has been demonstrated on a few examples to be able to create better programs than people have. This is not to say that people could not have done better themselves but that the GI did better starting from where the people had got to and where they ran out of time/money/enthusiasm and stopped.

Possibly GI has been too ambitious so far, in that it has compared its artefacts with code generated by some of the brightest programmers on the planet on non-trivial tasks. A more mundane goal might be to compare with the bread-and-better tedious tasks which we now expect people to code. If the machines can do this, they will get cheaper and so free millions of human programmers for more interesting challenges. Should EC aim to evolve simple cheap boiler plate code better than the average jobbing programmer?

V. CURRENT WEAKNESSES OF GI AND WAYS FORWARD

A number of issues have already been raised

- Is the new code legible?
- Is it maintainable?

These are important points, which may make some reluctant to take up GI. [Fry et al., 2012] try to answer them to some extent. In their study, they showed that the provision of automatically generated comments to accompany the source code changes made automatically generated bug repairs more maintainable, rather than less.

As will be mentioned on at the end of Section [VII] it is common to minimise the size of the code change. For example, in bug fixing delta debugging [Zeller, 1999] is often used to remove unneeded changes. Reducing the volume of code is often assumed to make it more comprehensible.

Of course if GI is operating on machine code binaries (Section [III]) then we assume that we do not care about understanding the program (perhaps we do not even have the source code) even before it is debugged or its performance is automatically improved. Hence we need only try to understand the mutations from the academic point of view of understanding our evolutionary process.

- Does the evolved program work?
- Is it correct?

It is possible to run the full gamete of software validation tools post evolution. It seems reasonable that these tools will work on artificial code as well as on hand written code.

The original code remains available and the new code can be automatically compared with it. We had one case where the new and old code were run together and their answers automatically compared [Langdon and Harman, 2010]. We did this more than a million times. No difference was ever found.

Do you ever test your code a million times? And check it gives the right answer? Every time?

- Who is liable if (when?) something goes wrong?

Hmm its not clear that this is worse for GI than for any other part of the software tool chain. Compilers are not formally verified. They certainly can do unexpected things. However these days they are sufficiently good, that their behaviour has become the de facto definition of the language they compile.

- User acceptability.

It might be suggested that people will not want to trust code that has been automatically generated. However Microsoft claim their Flash Fill (end of Section [VII]) has more than a hundred million potential users. In fact this insert of practical AI into the user experience has been widely welcomed.

- Benchmarks

From a practical point of view, experts in Evolutionary Computing will want an easy route into genetic improvement. Part of this ought to be a set of simple to understand benchmark problems [Ang and Li, 2002] and their associated tools so that they can quickly get useful results without an unnecessarily steep learning curve. Claire Le Goues has made a start on this by making available her set of 105 bugs to be repaired as part of her GenProg tool. Something similar is needed for other forms of program improvement.

VI. STEPPING STONES ON THE ROUTE FORWARD

There are five steps we need to go through before running a genetic programming system [Poli et al., 2008, page 19]. A key challenge for EC is to decide if they are really suitable for genetic improvement. And if not to investigate alternatives.

A. Representation

Traditionally in tree based GP the first two steps (choice of terminals, leaf nodes, and functions, internal tree nodes) define the problem representation. However we should also include considering the genetic operations. In EC we have a wealth of experience in devising representations.

An obvious requirement is that the representation should include at least one acceptable solution. So far with the programs modified and their new requirements it has not been difficult to ensure the existence of solutions. However it may be that some bug repairs have been less successful because they have restricted the range of modification they allow too much. Unfortunately the existence of a solution is not sufficient. We need the representation, fitness function and genetic operations to conspire together to make a fitness landscape which makes it practical to find a solution.

Most GI work has represented members of the population as changes to the target program’s source code. This has the advantage that changes may be human readable but is it the best for evolution? Should we be looking at:

- Is the source the right target? Would Evolutionary Algorithms do better trying to modify the program trace or the sequence of instructions it executes?
• Much GI work has focused on industrial strength code written in C or C++. Would other languages better? Is the source code the right target? Would intermediate levels like Java or .net byte code be more evolvable?
• What of genotype-phenotype mappings? Should EC use an intermediate mapping, perhaps based on Gruau’s embryology [Gruau and Whitley, 1993; Gruau, 1996]? New mutation operators. GI has so far been restricted to operations like deleting a line of the target program and copying a line of code and and pasting it elsewhere.
• New crossover operators.
• Provably correct transformations. Right at the beginning [Ryan and Walsh, 1995], the then conservative nature of the parallel computing community effectively mandated only provably semantics preserving transformations be used to convert sequential to parallel code. The risk of mutated code doing something unwanted is still very much with us. Perhaps with now much faster ways to check for semantic equivalence GI should re-consider its fast and loose ways?
Although SAT technology has progressed in leaps and bounds in the last ten years, in practice Evolutionary Computing might want to consider hybrid approaches in which only the most likely mutants are validated. Or indeed, formal methods are only used after evolution has finished.

B. Improving in Multiple Ways: EMOs
Traditionally people have been able to optimise code for one objective (typically speed). It appears they are less able to optimise programs for non-traditional objectives like extending battery life. However machines may be able to automatically optimise non-traditional objectives provided suitable measurements (e.g. energy consumption) can be incorporated into the fitness function.

It is typical in engineering to seek a good trade-off between multiple conflicting objectives. Evolutionary Multiobjective Optimization (EMO), e.g. [Deb et al., 2002], has been widely used (e.g. Coello Coello and Cruz Cortes, 2005 [Xue et al., 2013]) and are increasingly being used in Search Based Software Engineering (e.g. Langdon et al., 2009). Although very asymmetric objectives may be problematic [Langdon and Harman, 2014], EMOs offer the prospect of automatically optimising code for several objectives [Harman et al., 2012], which may be difficult for manual coders.

C. Improving Code and its Validation: Coevolution
Coevolution [Darwen and Yao, 2001] of code to pass the current test suite and simultaneous evolution of the tests to stretch the code [Hillis, 1992] has been considered but with little progress. However successful applications in financial modelling, presented at the recent UCL workshop on Genetic Improvement [Hemberg et al., 2015], may encourage more research into using coevolution within GI.

D. Fitness Measure. Can GI use Surrogates?
As mentioned in Section III existing GI work performs selection by creating and running the mutant code and comparing its performance with that of the original code. This is computationally demanding and typically the end result of all this work is condensed into a single bit: does this mutant get children or not.

To reduce computational overhead and so allow bigger populations, usually GI uses random subsets selected from the complete test suite every generation. Assessing fitness on dynamic randomised subsets goes back to Gathercole [Gathercole and Ross, 1994]. Langdon [2008] pointed out that probing a complete test suite every generation has boundaries and discontinuities. Which gives hope that they may be applied to program spaces. As will be seen in Section VI-H program spaces may be better behave than common prejudices suggest.

E. Setting key parameters values
In EC it is well known that parameters like population size and mutation and crossover rates can make a huge difference to how successful a run will be. There has been no published studies of how parameters affect GI. Surely there is EC theory [Back, 1996] or experience [Ribeiro Filho et al., 1994] which could be applied to the problem of evolving better programs?

F. Termination. Who is the result
As with many non-trivial EC problems, the choice of when to terminate evolution is often dominated by the available compute resources. But again, perhaps there is EC theory and practise to be applied here. Should we be looking at restart strategies when the population (genotype or phenotype) appears to have converged? How can we reliably recognise premature convergence? There has been only a little work in GI on preventing re-exploration of the same solutions (via some form of tabu list [Langdon et al., 2013]).

In EC (including GI) it is common to simply use the best individual in the last generation as the result of evolution. However, it is entirely feasible to store every mutant program and its associated fitness. Perhaps an earlier mutant might be chosen. We might want to use other criteria as well as fitness to choose the final mutant. For example we might opt for the
mutant which makes the fewest changes to the original code. In GI it is common to chose the mutant with the best fitness and then minimise it after evolution by removing changes one at a time and retaining only those essential for its improved performance.

G. The Search Space

The global structure of program search spaces is little understood, partly due to the lack of tools for analysing their complex structure. A recent model, local optima networks [Ochoa et al., 2014; Verel et al., 2011], helps to fill this gap by providing a way of expressing search spaces as graphs where nodes are local optima under a given mutation operator; and edges represent probabilistic transitions with an explorative operator, such as a stronger perturbation or crossover (Ochoa et al., 2015a). Modelling landscapes as networks brings a new set of tools and metrics for analysing search spaces and the possibility of visualising them (see Figure 2). The global structure of several combinatorial spaces, such as the travelling salesman problem, has been thought to contain a big-valley or central-massif where many local optima exist. That is, the local optima are not randomly distributed, instead good solutions tend to cluster around the global optimum. However, recent studies have observed that, for solutions close to the global optimum, this structure breaks down into multiple valleys (Hains et al., 2011; Ochoa et al., 2015b; Ochoa and Veerapan, 2016) (see Figure 2). In the study of energy surfaces in theoretical chemistry these have been called multiple funnels (Doye et al., 1999). Multiple funnels implies that local optima are organised into clusters. We suggest that local optima networks can be used to analyse the global structure of program spaces. Important aspects to study are the distribution of local optima and their connectivity pattern. Do program spaces conform a big-valley? Do they divide into multiple valleys or funnels? Answering these questions will help to design more effective algorithms for traversing program search spaces.

H. Neutral Networks

Traditionally the space of program mutations is regarded as very disjointed with few good programs. However actual experience with sizeable real-world programs (Schulte et al., 2014b) (it may be small toy program are less robust) in equivalent mutants in mutation testing (Yao et al., 2014) automatic bug repair and genetic improvement (Langdon and Petke, 2015) suggests that many changes do not affect programs at all. Indeed it may be that program spaces may not be as hard to search as expected. Neutral Networks have been studied in GP (Langdon and Poli, 1998; Banzhaf and Leier, 2005), Evolvable Hardware (Vassilev et al., 2000), GI (Schulte, 2014), Artificial Life (Standish, 2003) as well as in Nature (Babajide et al., 1997; van Nimwegen et al., 1999).

A key challenge to EC is to consolidate prior theory on landscapes riddled with fitness neutral pathways and usefully apply it to real world programs.

---

Fig. 2. Local optima network of a travelling salesman (TSP) instance with 666 cities. Nodes are local optima according to Lin-Kernighan, and edges represent probabilistic transitions with the L-K double-bridge perturbation operator. Colours identify the four largest connected components, which are related to the dominant funnels in the landscape. The red component (8 O’Clock) contains the global optima.

I. Semantic Search of Open Source code (GitHub etc.)

Software engineers are now used to vast repositories of less than perfect but freely available program source code. At present this is only exploited manually. However we are starting to see the use of Evolutionary Computing both to automatically fix bugs by reusing free code (Ke et al., 2015) and to transplant new functionality into existing code (Barr et al., 2015; Marginean et al., 2015). Indeed EC can evolve new functionality for sizeable applications (Harman et al., 2014; Jia et al., 2015).

VII. BENEFITS TO EVOLUTIONARY COMPUTING

As mentioned in the previous section (VI), there are several key challenges which are already core to the on going success of Evolutionary Computing. GI research contains many important practical real world problems which EC has already made substantial progress on and also Automatic Programming lies at the roots of Evolutionary Computing as a practical Artificial Intelligence technique. Whilst evolutionary routes to true AI may be probably some way yet, genetic improvement of existing code is here and now. Evolutionary Algorithms techniques offer the prospect of substantial progress both in the short term and towards more distance goals.

By sidestepping the Software Engineering Oracle Problem (mentioned in Section III and using existing automatic test case generation tools, EC is close to having automated fitness functions. By substantially automating program modification and by re-using open source code, it may be that EC can
lift millions of programmers from their current error prone grind of mundane programming to a higher level, which is more like them saying what needs to be done without having to tell the computer how to do it [Langdon and Poli, 2002]. When the computer gets it wrong, the future response might be to update the test suite, rather than the code. Indeed we are already seeing user level programming based solely on examples [Gulwani et al., 2012]. For example three years ago, Microsoft released “Flash Fill” within their excel 2013 spreadsheet. Flash Fill allows people to program excel purely from examples within their spreadsheet.

Surely Evolutionary Computing can do more!

VIII. CONCLUSIONS

Today Automatic Programming, in restricted domains, is a reality for millions of users (see previous section). Genetic Improvement [Langdon, 2015] is firmly rooted in Evolutionary Computing and already offers a general way of extending sizeable existing programs by using genetic programming [Poli et al., 2008] to evolve not complete programs but patches to them. In Section VI we have listed many deficiencies of current GI and hopes that the Evolutionary Computing experts may help. Perhaps the most urgent are the related problems of representation and the fitness landscape and also GA expertise in fitness surrogates may help radically reduce fitness evaluation effort. Being the second best way of solving any problem [Eiben and Smith, 2015] makes Evolutionary Computing very general but it is always at risk of being usurped in any domain by algorithms developed exclusively for that domain. To survive EC must keep conquering new challenges. Solving problems no one else can, or simply no one has been brave enough to try. The principle payment to EC (Section VII) may simply be the opportunity to work on truly challenging problems, relating back to the AI roots of EC and moving Automatic Programming towards the sort of programs that are well within the scope of manual methods.

REFERENCES


