Genetic Programming and Evolvable Machines (2020) 21:251–262
https://doi.org/10.1007/s10710-019-09364-2

Automatic programming: The open issue?

Michael O'Neill[1] · **Lee Spector**[2]

Received: 17 October 2018 / Revised: 21 April 2019 / Published online: 11 September 2019

Abstract
Automatic programming, the automatic generation of a computer program given a high-level statement of the program's desired behaviour, is a stated objective of the field of genetic programming. As the general solution to a computational problem is to write a computer program, and given that genetic programming can automatically generate a computer program, researchers in the field of genetic programming refer to its ability to automatically solve problems. Genetic programming has also been described as an "invention machine" that is capable of generating human-competitive solutions. We argue that the majority of success and focus of our field has not actually been as a result of automatic programming. We set out to challenge the genetic programming community to refocus our research towards the objective of automatic programming, and to do so in a manner that embraces a wider perspective encompassing the related fields of, for example, artificial intelligence, machine learning, analytics, optimisation and software engineering.

Keywords Automatic programming · Genetic programming · Open issue

1 Introduction

In the 10th Anniversary issue of the Genetic Programming and Evolvable Machines journal many of the perceived open issues in the field of genetic programming at that time were identified [48]. One additional open issue, perhaps absent through its glaringly obvious nature, but arguably the most significant, is that of achieving

Handled by Dr. W. B. Langdon and Dr. Nicholas Freitag McPhee.

✉ Michael O'Neill
m.oneill@ucd.ie

Lee Spector
lspector@hampshire.edu

[1] Natural Computing Research and Applications Group, UCD School of Business, University College Dublin, Dublin, Ireland

[2] Amherst College, Hampshire College, University of Massachusetts, Amherst, MA, USA

automatic programming. This was later highlighted by the authors of the open issue paper in a tutorial presented at the GECCO conference held in Amsterdam [49] and more recently by O'Neill and Fagan at GPTP 2018 as being the "elephant in the room" [60] of our research community.

While there have been many significant advances in the field of Genetic Programming, including the successful application of Genetic Programming to numerous challenging real-world problems where it has produced solutions better than the state-of-the-art [1], we have not solved the problem of automatic programming in any meaningfully scalable manner. Much success lies in the generation of single line functions through Genetic Programming's application to Symbolic Regression (for example see [58]) and design, for example, in the generation of analogue circuits [33–35], robot morphology [25] and antennae [40]. Moreover the annual HUMIES competition [1] illustrates, and celebrates the wealth of successes of Genetic Programming and related methods in challenging, real-world problem domains.

In this article we set out to highlight, what is in our opinion, the open issue for our field, that of achieving automatic programming, and to challenge our community to focus on this task while embracing a wider perspective in order to achieve scalable automatic programming.

2 What is automatic programming?

What do we actually mean by automatic programming? Automatic programming is a concept that has witnessed different interpretations over time. Early computer scientists referred to the ability to automatically generate machine code from an assembly language as automatic programming, and similarly with respect to the creation of compilers [46]. Our more recent, machine intelligence-inspired definition is perhaps best captured by Arthur Samuel when he stated "tell the machine what to do, not how to do it" [57], encapsulating the definitive high-level language if we interpret to "tell the machine what to do" as being through a natural language, or perhaps even a brain–computer interface. As others have highlighted in the past [55], while this description may be an aspirational goal for automatic programming it is perhaps unrealistic, at least in the foreseeable future. In the following section we highlight some of the challenges we face to achieve this aspirational goal. Effectively though, automatic programming can be considered to have evolved over time towards increasingly higher level programming languages. And, as such, as researchers employing artificial evolution we are optimistic that incremental progress can be made towards automatic programming's aspirational goal.

From his earliest work in Genetic Programming Koza has raised the potential for Genetic Programming to be used for Automatic Programming amongst other Artificial Intelligence problems such as sequence induction, pattern recognition, planning and machine learning [32]. Additionally, Koza proposed a set of sixteen properties that an Automatic Programming system must possess [34], such as, having to start with a high-level statement of the problem requirements, with the system generating output in the form of a computer program detailing a sequence of steps as to how

to solve the problem. Other properties include the ability to automatically organise useful groups of steps so that they can be reused, that this reuse has the ability to be parameterised, and that a hierarchy of reuse can be constructed automatically. More generally, the system should have the property of problem independence, it should have wide and scalable applicability, and it should be capable of achieving human-competitive performance.

3 Challenges and solutions towards automatic programming

There are many obstacles to navigate in order to successfully achieve automatic programming. We claim that automatic programming is an example of an AI-complete problem, in the sense that solving it fully "requires a full solution to the artificial intelligence problem" [41]. In other words, it falls under the grand challenge of Artificial Intelligence to achieve artificial *general* intelligence. This leads to a series of challenges we have identified by way of example to illustrate some of the key issues which will need to be addressed.

If indeed automatic programming is an AI-complete problem, by extension it is unrealistic to expect that genetic programming alone will be sufficient to achieve automatic programming, it is but one tool in our potential toolkit. A clear example of where this is likely to be the case is in the first process, or subsystem, of an idealised automatic programming system, that of capturing user intent through a user-system interface, where we, in the words of Samuel [57], "tell the machine what to do".

If user intent can be captured sufficiently the subsequent subsystem is likely to focus on code synthesis. In addition to the broad and unanswered question of what is the best representation to tackle code synthesis, amongst many other related open questions, attempts at automatic programming to date have largely been directed at, and resulted in, relatively small bodies of code [21]. In essence approaches to automatic programming to date suffer a scalability challenge.

The majority of existing approaches to automatic programming systems also represent examples of narrow AI, with application to very constrained and small problem instances (e.g., list processing [2, 31], string manipulation [16], constraint generation [53]). This is understandable, given the scale of the challenge to tackle an AI-complete problem such as automatic programming, we first have to crawl before we can walk. To put it another way, there are aspects of the automatic programming problem which are hierarchically decomposable, and require different foci in terms of, for example, domain knowledge, constraints, representations and benchmark problems.

We discuss these and some related issues in more detail below.

3.1 Telling the machine what to do

A significant barrier to the idealised automatic programming scenario is the degree of ambiguity that increasingly higher-level descriptions of "what to do" will

embody. Think of a web search engine receiving a query for the term "jaguar". Did the searcher mean a big cat or a car? The scale of ambiguities in requirements which might arise in the much larger search space of automatically synthesising programs is daunting. Understandably this has led to the majority of approaches to automatic programming tackling highly narrow, constrained problem instances. Success on these is then often achieved, for example, through the adoption of a very small set of primitives, minimal control flow structures (if any), and tight specifications and constraints on the forms of solutions such that the language is often domain specific and may not be Turing complete. Other ways in which the problem is narrowed might be through limited data structures or types (e.g., string manipulation by FlashFill), and through small datasets where generalisation is a challenge.

Even when we quantify what we are searching for in the form of a fitness function in modern day genetic programming, we don't always get what we think we asked for. We will need to move beyond a scalar fitness function to achieve automatic programming, and are likely to face problems with multiple and sometimes competing objectives. Of course, more generally the related credit assignment problem is a well known challenge for machine intelligence (c.f., Bucket Brigade for Classifier Systems [24]). Stepping stones in the right direction are likely to include software tests of various kinds and at various levels of abstraction, ideas of complexification [61] and developmental evaluation [23] with the gradual increase in the complexity of the solution and in the difficulty of the problem domain to which candidate solutions are exposed, and smarter selection strategies such as behaviour-based lexicase selection [20].

3.2 Scalability

There are many dimensions to achieve scalability of automatic programming. Issues to consider include using various kinds of modularity, and multiple and possibly complex data types and control structures, and the potential to generate new instances of modularity, data and control structures, and to modify these on the fly [34, 38]. Where hierarchical decomposition of a problem is achievable, the appropriate mechanisms will need to be incorporated into the program synthesis representations to facilitate the automatic identification and reuse of modules. The incorporation of automated abstraction coupled to semantic modelling are likely to reduce the barrier here.

An integral part of achieving scalability is the suitability of the representation adopted. This means we need to consider issues such as effective encodings coupled to the development and use of effective search operators employed by the search and learning algorithms. More generally an open issue for machine learning and artificial intelligence, representation has long been a subject of research in genetic programming, with work exploring genetic encodings such as binary, integer, tree, linear-tree, graphs, and grammars (e.g., [7, 8, 10–13, 27, 32, 42, 43, 67]), along with work on searching the representation space itself [29, 47, 59]. More generally, as identified in the field of evolutionary computation, important features in the design of representation include locality, redundancy and scaling of alleles [56].

3.3 Representation and approaches to code synthesis

Adopting evolutionary computation as a tool to synthesise code has resulted in the field of Genetic Programming. Using evolutionary computation in this inductive manner is, however, not the only approach to code synthesis. A number of approaches, which predate Genetic Programming, are captured by Biermann et al. [5] and include approaches which are logic-based (deductive and inductive), use formal specifications, or are based on production rules and the design of efficient data representations. Johnson [26] proposes that we consider links between genetic programming and the use of formal methods and program analysis.

Other approaches from which we might learn and adapt include inductive logic programming [44] and the related ADATE [45], and more recent incarnations such as DeepCoder [4], IGOR2 [31] and MagicHaskeller [28], Flashfill [15] and TerpreT [14]. It is heartening to see the awakening to this wider perspective in approaches such as by Bladek et al. [6] that combine formal specifications and genetic programming. The recent emergence of genetic improvement programming [54], which takes existing code and uses evolutionary heuristics to improve upon it in its functional and/or non-functional attributes, builds on the field of search-based software engineering [17]. Such approaches might be used to leverage existing code (e.g., the idea of leveraging "big code" in repositories such as github [64]) or propose methods which might be adopted as search operators to transform code which is being generated.

Given the diversity and numbers of approaches which have and could be adopted for code synthesis it would be desireable to compare and contrast performance. Pantridge et al. [51] recently attempted such a comparison across PushGP, a form of grammar-based GP, FlashFill, TerpreT and MagicHaskeller with the main observations that each of these approaches to program synthesis are designed to tackle different incarnations of program synthesis problems making comparison a challenge. This reminds us of the narrow versus general perspectives on AI we mentioned earlier and how the majority of automatic programming to date has arguably been by necessity narrow. To move towards the grand challenge of automatic programming we need to move towards more general approaches to program synthesis. Arguably GP has the best track record to date with respect to this aspect of the problem, as exemplified by work such as [3, 50, 59, 62, 69].

3.4 Benchmarks and baselines

To facilitate the development of automatic programming systems to move from narrow to more general classes of problems we require appropriate benchmark and grand challenge problems, and to adopt a scientifically rigorous approach with controls and baseline systems to compare to. In the recent Pantridge et al. study comparing multiple approaches to program synthesis [51], the General Program Synthesis Benchmark Suite [19] and a set of Basic Execution Model problems were employed. The latter represent problems which can be tackled at various levels of abstraction

while the former is a suite for general program synthesis requiring a range of data types, outputs and control structures. Another potential set of problems to allow comparison to, and perhaps hybridisation with, a wider set of machine learning algorithms is the Arcade Learning Environment (ALE) https://github.com/mgbellemare/Arcade-Learning-Environment, which was recently tackled using Cartesian GP [68] and Tangled Program Graph GP [30]. The adoption of the above problems represent the current best practice in selection of benchmarks. We are in no doubt of the need for the further principled development of additional benchmarks that can be used in a targeted manner to push the boundaries along different dimensions such as scalability, generalisation, and adaptation, and to facilitate comparison across a range of very different approaches to automatic programming. Also the development of Grand Challenge problems is useful to help us push towards more general Automatic Programming, and to raise awareness of the potential of this developing area to a wider audience with the openly defined HUMIES perhaps being our field's leading example.

3.5 GP as a tool

If Automatic Programming is the application domain, Genetic Programming is a method in our toolkit. Is Genetic Programming the best tool to achieve automatic programming, an AI-complete problem? On the one hand, it is not unusual for Genetic Programming to produce novel solutions that differ in a variety of ways from the solutions that humans would produce. This is a source of genetic programming's unusual power, but it may also present new challenges or exacerbate the challenges posed by all artificial intelligence technologies. On the other hand, solutions produced by genetic programming are expressed in the form of code, often in high-level languages, that humans can analyze and understand more easily than the products of many other artificial intelligence and machine learning technologies. With the aid of techniques for evolving more concise and interpretable solutions [18, 36, 37], genetic programming may offer benefits with respect to many of these issues. We would argue that we do not expect Genetic Programming to be sufficient, or at least on its own the most efficient method, to fully realise automatic programming, or at least the most appropriate or efficient method to achieve all the necessary functions that an automatic programming system requires.

We can describe an Automatic Programming system in such a manner as to include a number of distinct processes. Think of the complex life-cycle of software artefacts, designed, built and maintained by humans. The first process is likely the capture and translation of user-intent into an objective function, which is then passed on to the second process to guide code synthesis, before third and perhaps additional processes which might include code optimisation, localisation, maintenance and adaptation. This automatic programming system will also be a learning agent, embodied in and interacting with some environment. It is likely to require the ability to build models of its environment and the other agents (e.g., its users) with which it interacts, which for example, might facilitate more "intelligent" responses to the translation of user-intent into an objective function.

It is likely that the best tool(s) for the translation of user-intent into an objective function does not include Genetic Programming. For example, natural language processing technologies such as neural networks are more likely to dominate here. It is more likely that Genetic Programming will play a significant role in code synthesis and perhaps its adaptation (think genetic improvement programming), and that other heuristics and exact methods might be employed during optimisation (if not also during code synthesis). The learning process could employ various approaches to Machine Learning, likely hybridised to include neural networks, perhaps evolutionary computation, logic and reasoning.

One way to facilitate combining approaches from potentially useful fields such as Machine Learning, GP, Analytics, Optimisation, Search-based Software Engineering [17], program repair and improvement [39, 54, 65] etc., might include the development of appropriate software. For example, the python implementation of PushGP [52] opens this algorithm to a wider community beyond the field of GP, in particular those working in analytics and big data.

3.6 Trust, transparency, competence, reliability and ethics

Achieving the idealised form of automatic programming would be to solve an AI-complete problem and to facilitate the development of a form of general artificial, or machine, intelligence. We therefore have, as a community of researchers, a responsibility to consider the wider implications of such a technology for society. In addition to transparency of decision making, trust, competence and reliability of automatic programming systems, this work raises legal and ethical questions such as "Who is the author?," "Who is liable when the automatic programming system produces undesirable results?," and "How do we avoid bias in the algorithms [66]?" These are just a sample of issues that need serious attention. One approach is to shift responsibility towards the designer of the intelligent algorithm to ensure that it is well behaved [63].

Recently the European Commission's High-level expert group on Artificial Intelligence published "Ethics Guidelines for Trustworthy AI" [22], which builds upon principles identified by the European Group on Ethics in Science and New Technologies [9]. The key ideas behind trustworthy AI are that AI systems are lawful, ethical and robust. Being lawful refers to the systems' adherence to all the relevant regulations and laws. Robustness requires us to have confidence that the AI system will perform in a safe, secure and reliable manner and not unintentionally cause harm. Being ethical requires that ethical principles and values are respected. In realising AI, seven requirements are identified:

1. The need to respect human agency and oversight.
2. Technical robustness and safety.
3. Privacy and data governance.
4. Transparency.
5. Diversity, non-discrimination and fairness.
6. Environmental and societal well-being.
7. Accountability.

As responsible researchers there are clearly many issues which need to be taken into consideration when embarking upon research in this domain, and there is a clear need for further research and dialogue to be undertaken in how to approach the creation of trustworthy AI. Bringing this back to Automatic Programming, one advantage of increasingly higher-level code is its transparency. Higher-level languages are more open to human readability. This increasing transparency is desirable for multiple reasons, including, trust in the generated software, the ability to understand the generated software models leading to the potential to uncover new scientific knowledge, and the ability to modify and maintain the resulting software perhaps improving its competence and reliability. These properties would bring us closer to a form of Artificial Intelligence where we achieve augmented human performance by being able to work seamlessly in collaboration with AI technology such as automatic programming systems.

3.7 Other issues

Many of the open issues identified in the tenth anniversary issue article [48] also continue to present challenges to achieving automatic programming, such as the halting problem, achieving generalisation, the development of strong theory, the pros and cons of domain knowledge and the "A to I ratio," the large number of parameters, and dependence on syntax rather than behaviour and semantics.

4 Conclusion

As a field Genetic Programming has enjoyed significant success perhaps best illustrated through its many successful applications captured in the annual HUMIES competition. Despite this success, Genetic Programming has not achieved its stated goal of realising automatic programming. We presented what we mean by automatic programming and discussed some of the obstacles to its realisation, and we challenge the community to refocus its efforts towards the goal of automatic programming. We are optimistic that significant gains will be made towards automatic programming over the coming years, and that these will represent some of the biggest successes and impact that this field can bring to computer science and machine intelligence.

Acknowledgements We would like to thank the reviewers for their constructive feedback. MO'N is supported by the Science Foundation Ireland under Grants 13/IA/1850 and 13/RC/2094. This material is based upon work supported by the National Science Foundation under Grant No. 1617087. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Annual "humies" awards for human-competitive results. http://www.human-competitive.org/. Accessed 21 Apr 2019
2. Z. Manna, R. Waldinger, A deductive approach to program synthesis, in *Automatic Program Construction Techniques*, ed. by A. Bierman, G. Guiho, Y. Kodratoff (Macmillan Publishing Company, 1984)
3. A. Arcuri, X. Yao, Co-evolutionary automatic programming for software development. Inf. Sci. **259**, 412–432 (2014)
4. M. Balog, A.L. Gaunt, M. Brockschmidt, S. Nowozin, D. Tarlow, Deepcoder: learning to write programs, in *Proceedings International Conference on Learning Representations 2017*. OpenReviews.net (2017). https://openreview.net/pdf?id=rkE3y85ee. Accessed 21 Apr 2019
5. A. Bierman, G. Guiho, Y. Kodratoff (eds.), *Automatic Program Construction Techniques*, (Macmillan Publishing Company, 1984)
6. I. Bladek, K. Krawiec, J. Swan, Counterexample-driven genetic programming: heuristic program synthesis from formal specifications. Evolut. Comput. **26**(3), 441–469 (2018). https://doi.org/10.1162/evco_a_00228
7. N.L. Cramer, A representation for the adaptive generation of simple sequential programs, in *Proceedings of an International Conference on Genetic Algorithms and the Applications*, ed. by J.J. Grefenstette (Carnegie-Mellon University, Pittsburgh, 1985), pp. 183–187
8. K.A. De Jong, On using genetic algorithms to search program spaces, in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application* (L. Erlbaum Associates Inc., Hillsdale, 1987), pp. 210–216. http://dl.acm.org/citation.cfm?id=42512.42540. Accessed 21 Apr 2019
9. L. Floridi et al., AI4people—an ethical framework for a good AI society: opportunities, risks, principles and recommendations. Minds Mach. **28**, 689–707 (2018)
10. R. Forsyth, BEAGLE a Darwinian approach to pattern recognition. Kybernetes **10**(3), 159–166 (1981). https://doi.org/10.1108/eb005587
11. R.M. Friedberg, A learning machine: part I. IBM J. Res. Dev. **2**(1), 2–13 (1958). https://doi.org/10.1147/rd.21.0002
12. R.M. Friedberg, B. Dunham, J.H. North, A learning machine: part II. IBM J. Res. Dev. **3**(3), 282–287 (1959). https://doi.org/10.1147/rd.33.0282
13. C. Fujiki, J. Dickinson, Using the genetic algorithm to generate LISP source code to solve the prisoner's dilemma, in *Proceedings of the 2nd International Conference on Genetic Algorithms*, Cambridge (1987), pp. 236–240
14. A.L. Gaunt, M. Brockschmidt, R. Singh, N. Kushman, P. Kohli, J. Taylor, D. Tarlow, Terpret: a probabilistic programming language for program induction (2016). CoRR arXiv:1608.04428
15. S. Gulwani, Automating string processing in spreadsheets using input–output examples. SIGPLAN Not. **46**(1), 317–330 (2011). https://doi.org/10.1145/1925844.1926423
16. S. Gulwani, W.R. Harris, R. Singh, Spreadsheet data manipulation using examples. Commun. ACM **55**(8), 97–105 (2012). https://doi.org/10.1145/2240236.2240260
17. M. Harman, S.A. Mansouri, Y. Zhang, Search-based software engineering: trends, techniques and applications. ACM Comput. Surv. **45**(1), 11:1–11:61 (2012). https://doi.org/10.1145/2379776.2379787
18. T. Helmuth, N.F. McPhee, E. Pantridge, L. Spector, Improving generalization of evolved programs through automatic simplification, in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '17* (ACM, Berlin, 2017), pp. 937–944. https://doi.org/10.1145/3071178.3071330
19. T. Helmuth, L. Spector, General program synthesis benchmark suite, in *GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, ed. by S. Silva, A.I. Esparcia-Alcazar, M. Lopez-Ibanez, S. Mostaghim, J. Timmis, C. Zarges, L. Correia, T. Soule, M. Giacobini, R. Urbanowicz, Y. Akimoto, T. Glasmachers, F.F. de Vega, A. Hoover, P. Larranaga, M. Soto, C. Cotta, F.B. Pereira, J. Handl, J. Koutnik, A. Gaspar-Cunha, H. Trautmann, J.B. Mouret, S. Risi, E. Costa, O. Schuetze, K. Krawiec, A. Moraglio, J.F. Miller, P. Widera, S. Cagnoni, J. Merelo, E. Hart, L. Trujillo, M. Kessentini, G. Ochoa, F. Chicano, C. Doerr (ACM, Madrid, 2015), pp. 1039–1046. https://doi.org/10.1145/2739480.2754769

20. T. Helmuth, L. Spector, J. Matheson, Solving uncompromising problems with lexicase selection. IEEE Trans. Evolut. Comput. **19**(5), 630–643 (2015). https://doi.org/10.1109/TEVC.2014.2362729
21. T.M. Helmuth, General program synthesis from examples using genetic programming with parent selection based on random lexicographic orderings of test cases. Ph.D. thesis, College of Information and Computer Sciences, University of Massachusetts Amherst, USA (2015). https://web.cs.umass.edu/publication/details.php?id=2398. Accessed 21 Apr 2019
22. High Level Expert Group on Artificial Intelligence, Ethics guidelines for trustworthy AI. Technical report, European Commission (2019)
23. T.H. Hoang, D. Essam, R.I.B. McKay, N.X. Hoai, Developmental evaluation in genetic programming: the TAG-based frame work. Int. J. Knowl. Based Intell. Eng. Syst. **12**(1), 69–82 (2008). https://doi.org/10.3233/KES-2008-12106
24. J.H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence* (The University of Michigan, Ann Arbor, 1975)
25. G.S. Hornby, H. Lipson, J.B. Pollack, Generative representations for the automated design of modular physical robots. IEEE Trans. Robot. Autom. **19**(4), 709–713 (2003). https://doi.org/10.1109/TRA.2003.814502
26. C.G. Johnson, What can automatic programming learn from theoretical computer science?, in *The 2002 U.K. Workshop on Computational Intelligence (UKCI'02)* ed. by X. Yao (Birmingham, 2002). http://kar.kent.ac.uk/id/eprint/13729. Accessed 21 Apr 2019
27. W. Kantschik, W. Banzhaf, Linear-graph GP: a new GP structure, in *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, vol. 2278, Lecture Notes in Computer Science, ed. by J.A. Foster, E. Lutton, J. Miller, C. Ryan, A.G.B. Tettamanzi (Springer, Kinsale, 2002), pp. 83–92. https://doi.org/10.1007/3-540-45984-7_8
28. S. Katayama, Recent improvements of magichaskeller, in *Approaches and Applications of Inductive Programming*, ed. by U. Schmid, E. Kitzelmann, R. Plasmeijer (Springer, Berlin, 2010), pp. 174–193
29. R.E. Keller, W. Banzhaf, The evolution of genetic code in genetic programming, in *Proceedings of the Genetic and Evolutionary Computation Conference*, vol. 2, ed. by W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, R.E. Smith (Morgan Kaufmann, Orlando, 1999), pp. 1077–1082
30. S. Kelly, M.I. Heywood, Emergent tangled graph representations for Atari game playing agents, in *EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming*, vol. 10196, Lecture Notes in Computer Science, ed. by M. Castelli, J. McDermott, L. Sekanina (Springer, Amsterdam, 2017), pp. 64–79. https://doi.org/10.1007/978-3-319-55696-3_5
31. E. Kitzelmann, Data-driven induction of recursive functions from input/output-examples, in *Proceedings of the ECML/PKDD 2007 Workshop on Approaches and Applications of Inductive Programming (AAIP 2007)* (2007), pp. 15–26
32. J.R. Koza, Hierarchical genetic algorithms operating on populations of computer programs, in *Proceedings of the 11th International Joint Conference on Artificial Intelligence IJCAI-89*, vol. 1, ed. by N.S. Sridharan (Morgan Kaufmann, Detroit, 1989), pp. 768–774
33. J.R. Koza, Human-competitive results produced by genetic programming. Genet. Program. Evol. Mach. **11**(3/4), 251–284 (2010). https://doi.org/10.1007/s10710-010-9112-3
34. J.R. Koza, D. Andre, F.H. Bennett III, M. Keane, *Genetic Programming III: Darwinian Invention and Problem Solving* (Morgan Kaufman, Burlington, 1999)
35. J.R. Koza, M.A. Keane, M.J. Streeter, W. Mydlowec, J. Yu, G. Lanza, *Genetic Programming IV: Routine Human-Competitive Machine Intelligence* (Kluwer Academic Publishers, Dordrecht, 2003)
36. W. La Cava, K. Danai, L. Spector, Inference of compact nonlinear dynamic models by epigenetic local search. Eng. Appl. Artif. Intell. **55**, 292–306 (2016). https://doi.org/10.1016/j.engappai.2016.07.004
37. W.G. La Cava, Automatic development and adaptation of concise nonlinear models for system identification. Doctoral dissertations May 2014-current, vol. 731 (2016). http://scholarworks.umass.edu/dissertations_2/731/. Accessed 21 Apr 2019
38. W.B. Langdon, *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, vol. 1, Genetic Programming (Kluwer, Boston, 1998). https://doi.org/10.1007/978-1-4615-5731-9
39. C. Le Goues, S. Forrest, W. Weimer, Current challenges in automatic software repair. Softw. Qual. J. **21**, 421–443 (2013). https://doi.org/10.1007/s11219-013-9208-0

40. J.D. Lohn, G. Hornby, D.S. Linden, Human-competitive evolved antennas. Artif. Intell. Eng. Des. Anal. Manuf. **22**(3), 235–247 (2008). https://doi.org/10.1017/S0890060408000164
41. J.C. Mallery, Thinking about foreign policy: finding an appropriate role for artificially intelligent computers, in *The 1988 Annual Meeting of the International Studies Association* (1988)
42. R.I. McKay, N.X. Hoai, P.A. Whigham, Y. Shan, M. O'Neill, Grammar-based genetic programming: a survey. Genet. Program. Evol. Mach. **11**(3/4), 365–396 (2010). https://doi.org/10.1007/s10710-010-9109-y
43. J.F. Miller (ed.), *Cartesian Genetic Programming*, Natural Computing Series (Springer, Berlin, 2011). https://doi.org/10.1007/978-3-642-17310-3
44. S. Muggleton, Inductive logic programming: issues, results and the challenge of learning language in logic. Artif. Intell. **114**, 283–296 (1999)
45. R. Olsson, Inductive functional programming using incremental program transformation. Artif. Intell. **74**(1), 55–81 (1995)
46. M. O'Neill, Automatic programming in an arbitrary language: evolving programs with grammatical evolution. Ph.D. thesis, University of Limerick, Ireland (2001). http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/oneill/MichaelONeillThesis.ps.gz. Accessed 21 Apr 2019
47. M. O'Neill, C. Ryan, Grammatical evolution by grammatical evolution: the evolution of grammar and genetic code, in *Genetic Programming 7th European Conference, EuroGP 2004. Proceedings*, vol. 3003, Lecture Notes in Computer Science, ed. by M. Keijzer, U.M. O'Reilly, S.M. Lucas, E. Costa, T. Soule (Springer, Coimbra, 2004), pp. 138–149. https://doi.org/10.1007/978-3-540-24650-3_13
48. M. O'Neill, L. Vanneschi, S. Gustafson, W. Banzhaf, Open issues in genetic programming. Genet. Program. Evol. Mach. **11**(3/4), 339–363 (2010). https://doi.org/10.1007/s10710-010-9113-2
49. M. O'Neill, L. Vanneschi, S. Gustafson, W. Banzhaf, Open issues in genetic programming, in *Tutorial on Open Issues in Genetic Programming at GECCO 2013* (The Netherlands, Amsterdam, 2013)
50. M. Orlov, M. Sipper, FINCH: a system for evolving Java (bytecode), in *Genetic Programming Theory and Practice VIII, Genetic and Evolutionary Computation, chap. 1*, vol. 8, ed. by R. Riolo, T. McConaghy, E. Vladislavleva (Springer, Ann Arbor, 2010), pp. 1–16
51. E. Pantridge, T. Helmuth, N.F. McPhee, L. Spector, On the difficulty of benchmarking inductive program synthesis methods, in *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '17* (ACM, Berlin, 2017), pp. 1589–1596. https://doi.org/10.1145/3067695.3082533
52. E. Pantridge, L. Spector, PyshGP: PushGP in python, in *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '17* (ACM, Berlin, 2017), pp. 1255–1262. https://doi.org/10.1145/3067695.3082468
53. T.P. Pawlak, K. Krawiec, Synthesis of constraints for mathematical programming with one-class genetic programming. IEEE Trans. Evolut. Comput. (2018). https://doi.org/10.1109/TEVC.2018.2835565
54. J. Petke, S.O. Haraldsson, M. Harman, W.B. Langdon, D.R. White, J.R. Woodward, Genetic improvement of software: a comprehensive survey. IEEE Trans. Evolut. Comput. **22**(3), 415–432 (2018). https://doi.org/10.1109/TEVC.2017.2693219
55. C. Rich, R.C. Waters, Automatic programming: myths and prospects. Computer **21**(8), 40–51 (1988). https://doi.org/10.1109/2.75
56. F. Rothlauf, *Representations for Genetic and Evolutionary Algorithms* (Springer, Berlin, 2006)
57. A.L. Samuel, Some studies in machine learning using the game of checkers. IBM J. Res. Dev. **3**(3), 210–229 (1959). https://doi.org/10.1147/rd.33.0210
58. M. Schmidt, H. Lipson, Distilling free-form natural laws from experimental data. Science **324**(5923), 81–85 (2009). https://doi.org/10.1126/science.1165893
59. L. Spector, A. Robinson, Genetic programming and autoconstructive evolution with the push programming language. Genet. Program. Evol. Mach. **3**(1), 7–40 (2002). https://doi.org/10.1023/A:1014538503543
60. M. O'Neill, D. Fagan, The Elephant in the room: Towards the application of genetic programming to automatic programming. in *Genetic Programming Theory and Practice XVI* (Springer, 2019), pp. 179–192.
61. K.O. Stanley, R. Miikkulainen, Evolving neural networks through augmenting topologies. Evolut. Comput. **10**(2), 99–127 (2002). https://doi.org/10.1162/106365602320169811

62. A. Teller, Turing completeness in the language of genetic programming with indexed memory, in *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, vol. 1 (IEEE Press, Orlando, 1994), pp. 136–141. https://doi.org/10.1109/ICEC.1994.350027
63. P.S. Thomas, B.C. da Silva, A.G. Barto, E. Brunskill, On ensuring that intelligent machines are well-behaved. CoRR arXiv:1708.05448 (2017)
64. M. Vechev, E. Yahav, Programming with "big code". Found. Trends Program. Lang. **3**(4), 231–284 (2016). https://doi.org/10.1561/2500000028
65. W. Weimer, S. Forrest, C. Le Goues, T. Nguyen, Automatic program repair with evolutionary computation. Commun. ACM **53**(5), 109–116 (2010). https://doi.org/10.1145/1735223.1735249
66. S.M. West, M. Whittaker, K. Crawford, Discriminating systems: gender, race and power in AI. Technical report (2019)
67. P.A. Whigham, Grammatically-based genetic programming, in *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications* ed. by J.P. Rosca, Tahoe City, California, USA, pp. 33–41 (1995). http://divcom.otago.ac.nz/sirc/Peterw/Publications/ml95.zip. Accessed 21 Apr 2019
68. D.G. Wilson, S. Cussat-Blanc, H. Luga, J.F. Miller, Evolving simple programs for playing Atari games, in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '18* (ACM, New York, 2018), pp. 229–236. https://doi.org/10.1145/3205455.3205578
69. J. Woodward, Evolving turing complete representations, in *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, ed. by R. Sarker, R. Reynolds, H. Abbass, K.C. Tan, B. McKay, D. Essam, T. Gedeon (IEEE Press, Canberra, 2003), pp. 830–837. https://doi.org/10.1109/CEC.2003.1299753