Connecting automatic parameter tuning, genetic programming as a hyper-heuristic and genetic improvement programming.

John R. Woodward
University of Stirling
Stirling
Scotland, United Kingdom
jrw@cs.stir.ac.uk

Colin G.Johnson
University of Kent
Kent
England, United Kingdom
C.G.Johnson@kent.ac.uk

Sandy Brownlee
University of Stirling
Stirling
Scotland, United Kingdom
sbr@cs.stir.ac.uk

ABSTRACT

Automatically designing algorithms has long been a dream of computer scientists. However, early attempts, including genetic programming (GP) which evolves computer programs *from scratch*, have fail to meet this lofty goal. However, in recent years there have been a number of different technologies with an alternative goal of taking *existing programs* and attempting to improvement them.

These methods form a continuum of methodologies, from the "limited" ability to change (for example only the parameters) to the ability to change the whole program. These include; automatic parameter tuning (APT), using GP as a hyper-heuristic (GPHH) to automatically design algorithms, and genetic improvement (GI), which we will now briefly review. Part of research is building links between existing work, and the aim of this paper is to bring together these currently separate approaches.

Keywords

Genetic Improvement, Genetic Programming

1. POSITION

This paper will first examine a number of approaches (APT, GPHH, and GI), and consider these as a spectrum. It will then point out that there are common issues with all of these methods, such as the pitfall of overtuning, and the ability to specialize to a probability distribtion of input data. These current methods typically only contribute a small change to the resulting program, but is a step toward automatic programming which *does not* start from scratch.

APT takes programs and automatically searches the space of parameters for that program [9]. Different parameter settings compete in a race towards the optimal parameter setting. Deep parameter tuning [14] distinguishes between *surface parameters*, which are intended for the user to change, and *deep parameters*, which sit within the bowels of the program, and are not typically exposed to the user. "Programming by Optimisation" [5], is the philosophy which encourages the programmer not to make premature design decisions and hard code them, but allow them to be delayed and exposed to user as parameters (which could automatically be tuned [9]). This moves the onus of design choices from the programmer to the APT tool, effectively making the program more flexible. This philosophy, of course, raises an interesting question about how much of the internal workings of a program should be exposed to a novice or expert user.

GPHH [12, 4] defines a fixed template which is considered to be hard coded, and variations are generated by GP and treated as a parameter [13]. GPHH takes an existing program, and evolves variations on a component of the program, given the type signature of the component. It does not alter other parts of the program (i.e. the template). GPHH treats a function as a parameter which can be passed in, as is normal in functional programming. GPHH makes use of the template method design pattern [13].

GI takes an existing program, and evolves edits to it, and can potentially change any part of the program. GI has had some notable successes [6]. Also notable is that GI can explore non-functional trade-offs [3]. An overarching view of these approaches, is not to think of a program as a single entity, but as a set of programs. This defines a parato-front of programs which APT can can move between, depending on the environment in which they want the program to operate in [4].

APT, GPHH and GI form a spectrum of technologies to improve an existing program. In the extreme case we can view APT as automatic programming by considering the following example. A vector of bit-strings is fed into a program. We can interpret these bit-strings as the "numerical" parameters of the program. However, a looking at it differently, we can think of the strings as programs being fed into a Universal Turing Machine (or as a text program being fed into a compiler). In the remainder of this paper, we look at APT as a machine learning problem, and then overfitting and specialization.

Optimisation is typically considered as a one stage process (i.e. optimisation), and machine learning as a two stage process (i.e. training and validation). However, automatically constructing an optimisation algorithm for a probability distribution of problem instances is a two stage process consisting of training and validation, and is therefore a ma-

WOODSTOCK '97 El Paso, Texas USA

DOI: 10.475/123_4

chine learning methodology. Approaching the construction of an optimization algorithm as a machine learning problem makes explicit the separation of a training and test set. To manually tune a metaheuristic on a problem instance, and then claim it has good performance on that instance is not the correct approach. [1].

In early GP papers, quality of a program was judged by how well it performed on the training cases. However, to demonstrate that a program generalises, its functionality should be demonstrated on an independent set of test cases, as is the practice in supervised machine learning. It is also a case with the methods discussed here: they are trained on one set of test cases and their behaviour is validated on a second set of independence test cases drawn from the same probability distribution [2]. In the case of parameter tuning, we wish to find a set of parameters which perform well on the training set, but also perform well on a second set of independent problems. We can say the parameters are tuned to that probability distribution. Over-fitting has been recognized as an issue by the GI community [8]. When evaluating the correctness of a repair, in the *same* test set, programs which pass most test cases are just a likely to fail test as pass them. Therefore, we should employ a second set of test cases to validate any claimed fixes.

Conversely, the phenomenon of over-fitting is connected with specialization. There are a number of papers which use automatic tuning or programming techniques to align the program with the probability distribution of data it will be expected to see. [7] specialize SAT solvers to classes of problems from Combinatorial Interaction Testing. Similarly [4] specialize optimizers to particular classes of functions.

Automatically tuning "parameters" whether they be plain numerical, or components of a program, does a number of things. When we compare two metaheuristics we allow them the same number of evaluations of the objective function. This may seem fair; however, one metaheuristic may have been tuned more than another metaheuristic before the comparison began. APT ensures metaheuristics received the same amount of tuning prior to the validation.

Often authors do not claim optimality of parameters which have only been tuned using a trial and error process. However, this is unsystematic, and lacks reproducibility (i.e. manually retuning may involve a different number of trial and error evaluations). Automatic tuning parameters obviously replaces manual tuning, but is more explicit (i.e. it is an algorithm), and is therefore a more transparent process.

The contribution of this paper is placing APT, deep parameter tuning, GPHH and GI in relation to one another. We are not claiming that one method is better than another. Indeed, there may be a sweet spot where one method is better for one type of application than another. With all of these methods, the resulting program is part human-made, and part machine made. As we move toward the original goal of automatically designing algorithms from scratch, as our methods become more successful, the amount of code generated automatically will increase in proportion to the amount of human-generated code.

2. REFERENCES

[1] M. Birattari. *Tuning metaheuristics: a machine learning perspective*, volume 197. Springer, 2009.

[2] C. Giraud-Carrier and F. Provost. Toward a Justification of Meta-learning: Is the No Free Lunch Theorem a Show-stopper? In *Proceedings of the ICML-2005 Workshop on Meta-learning*, pages 12–19, 2005.

[3] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark. The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 1–14, New York, NY, USA, 2012. ACM.

[4] L. Hong, J. Woodward, J. Li, and E. Ozcan. Automated design of probability distributions as mutation operators for evolutionary programming using genetic programming. In K. Krawiec, A. Moraglio, T. Hu, A. S. Uyar, and B. Hu, editors, *Proceedings of the 16th European Conference on Genetic Programming, EuroGP 2013*, volume 7831 of *LNCS*, pages 85–96, Vienna, Austria, 3-5 Apr. 2013. Springer Verlag.

[5] H. H. Hoos. Programming by optimization. *Commun. ACM*, 55(2):70–80, Feb. 2012.

[6] W. B. Langdon and M. Harman. Genetically improving 50000 lines of C++. Research Note RN/12/09, Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK, 19 Sept. 2012.

[7] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In M. Nicolau, K. Krawiec, M. I. Heywood, M. Castelli, P. Garcia-Sanchez, J. J. Merelo, V. M. Rivas Santos, and K. Sim, editors, *17th European Conference on Genetic Programming*, volume 8599 of *LNCS*, pages 137–149, Granada, Spain, 23-25 Apr. 2014. Springer.

[8] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 532–543, New York, NY, USA, 2015. ACM.

[9] T. Stützle and M. López-Ibáñez. Automatic (offline) configuration of algorithms. In *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '13 Companion, pages 893–918, New York, NY, USA, 2013. ACM.

[10] J. Swan and N. Burles. Templar–a framework for template-method hyper-heuristics. In *Genetic Programming*, pages 205–216. Springer International Publishing, 2015.

[11] A. Teller and D. Andre. Automatically choosing the number of fitness cases: The rational allocation of trials. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, 13-16 1997.

[12] J. R. Woodward and J. Swan. Automatically designing selection heuristics. In G. L. Pappa, A. A. Freitas, J. Swan, and J. Woodward, editors, *GECCO 2011 1st workshop on evolutionary computation for designing generic algorithms*, pages 583–590, Dublin, Ireland, 12-16 July 2011. ACM.

[13] J. R. Woodward and J. Swan. Template method hyper-heuristics. In J. Swan, K. Krawiec, J. Woodward, C. Simons, and J. Clark, editors, *GECCO 2014 Workshop on Metaheuristic Design Patterns (MetaDeeP)*, pages 1437–1438, Vancouver, BC, Canada, 12-16 July 2014. ACM.

[14] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke. Deep parameter optimisation. In *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*, pages 1375–1382. ACM, 2015.