Automating Algorithm Design through Genetic Programming Hyper-Heuristics

Elsa M. Browning
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
brow3924@morris.umn.edu

ABSTRACT

actually write an abstract

Keywords

Evolutionary Computation, Genetic Programming, Hyper-Heuristics, Autoconstruction

1. INTRODUCTION

Mostly notes right now

Traditionally, genetic programming solutions to problems evolve, but almost everything else is specified by the system designer [5]. In autoconstruction, the variation methods are evolving as well.

maybe put in abstract?

In sections 2.1, 2.2, and 2.3 we describe necessary background for understanding the rest of this paper. Next, we describe the history of automating algorithm design (AAD) with a focus on hyper-heuristics (HH) optimization in section 2.4. Then, in section 3, we go over a few different genetic programming variants used in HH and the success of two in particular. Finally, we will go over current research being done with stack-based genetic programming (one of the GP variants) in section 4; the stack-based programming language called Push is used in this example along with a technique for AAD called autoconstruction.

this is long roadmap. Probably need to shorten

2. BACKGROUND

2.1 Evolutionary Computation

Evolutionary Computation (EC) is a subfield of artificial intelligence that uses algorithms based on biological evolution to reach global optimization. These algorithms start with a population of potential solutions to a problem or set of problems, then the worst individuals from the population are removed. The remaining individuals are mutated or recombined in some way to create a new population with the same number of individuals as the first population. Occasionally new individuals are introduced into the next population as well. This process is repeated until the *global optima*, the best solution(s) possible, is found. Sometimes an algorithm is not able to find a global optima (due to the algorithm not being good enough), so it has a designated stopping point to prevent the program from running forever trying to find it. This can be things like a time limit or a limit on the number of generations.

is there a way I can simplify this paragraph (particularly the first sentence) or reword it to be better...?

EC techniques are very useful and can solve a wide variety of problems. There are many different kinds of techniques in EC, some of which are more suited to certain types of problems than others. Some examples include genetic algorithms, gene expression programming, and artificial immune systems to name a few. Genetic programming is the EC technique we will be focusing on in this paper.

2.2 Genetic Programming

Most EC methods produce solutions in the form of a single answer (or set of answers) to a problem. In genetic programming (GP), our solutions are in the form of programs. These programs then solve a problem (or set of problems), adding a level of abstraction to the problem solving process.

I'm not actually sure abstraction is the right word here, but it sounds right...

This also automates a large part of the algorithm design process by allowing an algorithm to evolve rather than designing and revising the algorithm by hand.

GP works by encoding programs into a set of genes and then modifying those genes, usually with a genetic algorithm, to evolve a program which will perform well in a predefined task. The methods used to encode the programs into artificial chromosomes and to evaluate the fitness of a program are still under active research. There are several different GP representations and variations, each of which has pros and cons depending on the problem. We will go into more detail on these in section 3.

2.3 Hyper Heuristics

2.4 History of AAD

Automating algorithm design (AAD) has been investigated by several different areas for about 60 years. In the 1950s, the term "machine learning" was defined as "computers programming themselves" [2]. The definition has changed over time to focus more on the learning aspect, but machine learning is still an important area in AAD. While out of the scope of our paper, machine learning plays a ma-

UMM CSci Senior Seminar Conference, April 2017 Morris, MN.

jor role in the history of AAD and was one of the first fields to do so.

Focusing on the automation of heuristic methods, we can trace the beginnings of hyper-heuristics (HH) to the 1960s (however, the term HH was not coined until 2000) [2]. Early approaches of HH focused on automatically setting the parameters of evolutionary algorithms [2]. A *parameter* used to be thought of as things like mutation rates and crossover, however the definition has expanded to include evolutionary algorithm components like selection mechanisms, and mutation and crossover operators [2]. Many researchers still question which parameters to tune when designing HH systems. Traditionally, parameters are tuned before the evolution starts and controlled during the evolution [2]. There are, however, exceptions.

The idea of *self-adaption*, where an algorithm is able to tune itself to a given problem while solving it, emerged later.

add more about evolving evolution?

Now, we have two major types of HH: *heuristic selection* and *heuristic generation*; the first focuses on selecting the best algorithm from a set of existing heuristics while the latter focuses on generating a new heuristic from the components of existing heuristics [2]. Heuristic generation uses several different methods for the generation process, but we will be focusing on heuristic generation techniques that use genetic programming.

I focus pretty heavily on HH, which is good because that is the focus of paper, but feels weird because it's less of 'history of AAD' and more 'history of HH'

3. GENETIC PROGRAMMING VARIANTS

3.1 Tree-based Genetic Programming

3.2 Stack-based Genetic Programming

4. AUTOCONSTRUCTION

In genetic programming hyper-heuristics (GPHH), the goal is to use genetic programming to evolve a program that will solve hard computational search problems. The individual programs serve as potential solutions. In most GPHH, the potential solutions are evolving, but everything else is specified by the system designer.

not sure how to phrase previous sentence. Also not sure if I need to clarify what I mean by "potential solutions"

In autoconstruction (a form of GPHH), evolution is evolving as well. This happens by encoding the methods of variation into the programs that are evolving so that, as a program evolves, its variation methods also evolve (see section 4.1 for how this works). A simplified definition of autoconstruction would be one or more parents producing one or more children, where the parents and children are programs which serve as potential solutions to a set of computational search problems.

TO NIC: is there more I could say to summarize autoconstruciton quickly or do you think this is fine? Should I say something about how autoconstruction = only genetic operator?

Prior work on autoconstruction has explored a variety of system designs, but, until recently, none of them have been able to solve hard problems. A new system called Autoconstructive Diversification of Genomes (AutoDoG) has broken this trend by solving at least one hard problem: Replace Space with Newline [5]. And, in recent unpublished work, AutoDoG has actually solved a problem that has never been solved before [1].

Not sure how much of this is introduction stuff

4.1 Push and Plush

Push is a stack-based programming language with a separate stack for each data type. It was developed for program evolution and autoconstruction was one of the driving forces behind the original design [5]. This is the programming language that AutoDoG, the system we describe in section 4.2, uses. Push programs are strings of instructions, constants, and parentheses with only one syntax requirement: the parentheses must be balanced [3]. Instructions are executed by putting them on the `exec` stack. If the necessary arguments are not available for the instruction to be executed, the instruction is skipped. For example, assume all stacks are empty and assume a program says `(1 2 integer_add)`. This means push 1 and 2 onto the `integer` stack and push the `integer_add` onto the take the `exec` stack [4]. The `integer_add` instruction will try to execute: the first two integers (in this case 1 and 2) are popped off the top of the `integer` stack, added together, and the result (in this case 3) is pushed back on to the `integer` stack. Since there were enough arguments, the instruction executes. If, instead, the program was `(1 integer_add)` then it would push 1 onto the `integer` stack, but skip the `integer_add` because there are not enough integers in the `integer` stack.

not sure I should include example inline like this, might want to separate out to a figure and try to shorten somehow

Plush is a linear genome format for Push [5]. This can be thought of as computerized DNA. It is a program in a linear format to make program evolution easier.

TO NIC: Is this accurate? This is how I have been thinking about linear genomes. Is this actually linear GP and using the registers (if so, should I talk about registers) or is it different?

In section 4.2, AutoDoG is actually evolving linear genomes and then translating those genomes back into Push programs [5]. This allows for methods of variation in a population to be encoded and evolved more easily. To aid in the translation process, Plush uses "epigenetic close markers" (in other words, parenthesis) to separate blocks of code [5]. It also uses "epigenetic silencing markers" to indicate when a part of the code in a genome should not be translated into a Push program [5]. This is arguably quite similar to how DNA and evolution work in biology in that some people are carriers for specific traits, but do not show them. This allows for these traits to be passed on to the next generation.

TO NIC: I might want to talk about why this is useful... why is this useful?

4.2 AutoDoG

In this section, we briefly describe some of the key features of AutoDoG.

When designing AutoDoG, Spector et al. [5] wanted to maintain diversity in parent selection. To do this, AutoDoG uses Lexicase Selection, described below:

In each parent selection event,

- start with pool of entire population
- filter pool based on how each program performs on individual fitness cases (performed in random order, and one at a time)

HOW do I phrase prev bullet point?

- In each fitness case, only retain programs best on that case.

List or paragraph?

More/better description of lexi (reread section in helmuth's dissertation)

Its name comes from the way it filters the population using a kind of "lexigraphic ordering" of cases [5]. When tested on a benchmark suite of problems taken from introductory programming textbooks and compared to the results of other current GP parent selection techniques, Lexicase Selection allows for the solution of more problems in fewer generations [5].

rephrase prev sentence?

Another thing that Spector et al. [5] was concerned about when designing AutoDoG was cloning. This happens when an exact copy of a program moves on to the next generation. This is concerning because it can cause early convergence – it is possible that a population could lose all diversity and this could prevent the system from reaching a solution.

TO NIC: is this (prev sentence) actually a concern? I assume it is, but maybe it's so unlikely that it's not

Cloning also significantly slows down the rate of evolution. This is bad because we want to reach a solution quickly. Most autoconstruction systems have some form of the "no cloning rule," and AutoDoG has a form of this as well [5]. AutoDoG's version of this requires offspring to pass a more stringent diversification test in order to enter the population [5]. The test starts by taking an individual genome for the next generation. Then, the individual takes itself as a mate and makes temporary children. If the children differ enough from the individual and from each other, the individual enters the population and the children are discarded. If the individual fails, a random genome is generated and takes the test. If the random genome fails, a blank genome (with no code in it) is generated and passed on to the next generation.

AutoDoG works similarly to PushGP, a reasonably standard generational programming system, but is run with autoconstruction as the sole genetic operator rather than using human designed mutation and crossover operators.

say more about this?

4.3 Results

AutoDoG is unique among autoconstructive systems in that it can solve hard problems. However, as stated by Spector et al. [5],

> We do not know which of these, or which combinations of these, may be responsible for the fact that AutoDoG appears to be capable of solving more difficult problems than previous autoconstructive evolution systems.

This is because it's hard to separate the pieces; there are a lot of intertwined elements of AutoDoG and Push. Separating them to find exactly which elements contribute to the success without unraveling the entire system is difficult and has not yet been accomplished.

TO NIC: Is this (above) more correct?

One challenging problem AutoDoG has solved is Replace Space with Newline (RSWN). This is a software synthesis problem which, when given a string, requires the solution to print the string replacing spaces with newlines. It also requires the return of the integer count of the non-whitespace characters. This is complex because it involves multiple data types (such as strings and integers) and outputs.

AutoDoG does not actually do better than standard PushGP on this problem. It succeeds less reliably than PushGP. AutoDoG solves the RSWN problem 5–10% of the time, producing general solutions [5]. This may not seem like much, but the fact that AutoDoG solves the problem at all is actually quite impressive. It is not surprising that AutoDoG performs less reliably because AutoDoG has to learn how to evolve as well as learn how to solve the problem. In recent unpublished work however, there may be examples of autoconstruction performing 'better' than PushGP in a more meaningful sense; autoconstruction has managed to find solutions to problems that have never been solved with regular PushGP [1].

Add more info about new AutoDoG results

One interesting thing to note about AutoDoG is how it evolves. In standard PushGP, one can see a steady incline in differences between the genomes of parents and their children. Over time, change occurs at a steady rate and is not very dramatic from one generation to the next. In AutoDoG, there are big differences between the genomes of some parents and their children. This is interesting because the methods for evolution are encoded into the genomes in AutoDoG. These DL graphs

insert DL graphs or cut this paragraph

show us how evolution is affected from one generation to the next.

Clean up, more info about weird AutoDoG evolution.

5. CONCLUSIONS AND FUTURE WORK

6. ACKNOWLEDGMENTS

7. REFERENCES

[1] E. Moshkovich. private communication.

[2] G. L. Pappa, G. Ochoa, M. R. Hyde, A. A. Freitas, J. Woodward, and J. Swan. Contrasting meta-learning and hyper-heuristic research: the role of evolutionary algorithms. *Genetic Programming and Evolvable Machines*, 15(1):3–35, March 2014.

[3] L. Spector. Autoconstructive evolution: Push, pushgp, and pushpop. In L. Spector, E. Goodman, A. Wu,

W. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. Garzon, and E. Burke, editors, *GECCO '01: Proceedings of the 2001 on Genetic and Evolutionary Computation Conference Companion*, pages 137–146. Morgan Kaufmann Publishers.

[4] L. Spector and N. F. McPhee. Expressive genetic programming: Concepts and applications. In T. Friedrich and H. P. Institute, editors, *GECCO '16: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, pages 589–608.

[5] L. Spector, N. F. McPhee, T. Helmuth, M. M. Casale, and J. Oks. Evolution evolves with autoconstruction. In T. Friedrich and H. P. Institute, editors, *GECCO '16: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, pages 1349–1356. ACM, July 2016.

Add more of references to bib