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

Automating algorithm design is a current subject of research in several different fields. One field that has approached it is hyper-heuristic optimization. Hyper-heuristics are heuristic search methods which seek to automate the process of selecting, generating, or adapting several simpler heuristics in order to solve computational search problems. This process is usually done with machine learning techniques, but in this paper we will be focusing on a family of algorithms within Evolutionary Computation called genetic programming. Hyper-heuristics that use genetic programming are simply called "genetic programming hyper-heuristics." We will focus on one genetic programming hyper-heuristic in particular called autoconstruction.

Keywords

Evolutionary Computation, Genetic Programming, Hyper-heuristics, Autoconstruction

1. INTRODUCTION

Since at least the 1950s, engineers have been trying to automate the design of algorithms [4]. When we say 'automate the design of algorithms' this means we have been trying to reduce the amount of work people put in to the design process and have been trying to increase the amount of work the computer puts in. Two major approaches to this problem are meta-learning, in the field of supervised machine learning, and hyper-heuristics, in the field of optimization [4]. While machine learning is out of the scope of this paper, it plays an important role in tackling the task of automating algorithm design.

One approach to the problem of automating algorithm design would be the use of hyper-heuristics; these are heuristic search methods which seek to automate the process of selecting, generating, or adapting several simpler heuristics in order to solve computational search problems. A *heuristic* is a function that ranks alternatives in a search algorithm at each branching step and uses that information to choose which branch to follow. The goal of a heuristic is to find a solution in a reasonable amount of time that is capable of solving the problem at hand.

Hyper-heuristics can be classified under the field of Evolutionary Computation (EC), which is a subfield of artificial intelligence. EC encompasses algorithms that use techniques based on biological evolution to solve problems. A family of algorithms within EC, called genetic programming (GP), uses biological techniques to evolve programs which then solve problems. When designing hyper-heuristics, engineers can use several different kinds of algorithms to automate the process of selecting, generating or adapting several simpler heuristics; GP algorithms are often chosen for this process. There are many variations of GP, and the variant used may affect the success of the hyper-heuristic.

Traditionally, GP solutions to problems evolve, but almost everything else is specified by the system designer [7]. In a newer technique, called autoconstruction, the variation methods are evolving as well. Autoconstruction can be briefly defined as 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. So autoconstruction can actually be thought of as a hyper-heuristic technique that uses GP to evolve its solutions (which are in the form of programs), also known as a genetic programming hyper-heuristic.

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 with a focus on hyper-heuristics optimization in Section 2.4. Then, in Section 3, we mention an experiment that tests different GP variants used in hyper-heuristics and we go into detail on stack-based GP (one of the genetic programming variants). Next, we will go over current research being done with stack-based genetic programming in Section 4; the stack-based programming language called Push is used in this example along with a technique for automating algorithm design called autoconstruction. Finally we will go over the results of the recent research and some conclusions in Sections 4.3 and 5 respectively.

2. BACKGROUND

In this Section, we introduce the field of Evolutionary Computation and define terminology used throughout the rest of the paper. Next we go over genetic programming, which is a family of algorithms in Evolutionary Computation used for evolving programs. Finally, we define what a hyper-heuristic is and go over a brief history on the development of hyper-heuristics.

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

2.1 Evolutionary Computation

Evolutionary Computation (EC) is a subfield of artificial intelligence that uses algorithms based on biological evolution to solve problems. Much of the terminology in EC is based on biology as well. We will describe the general process of an EC algorithm in the following paragraph and define much of the basic terminology throughout this description.

These algorithms start with a group, or *population*, of potential solutions to a problem or set of problems. These potential solutions, or *individuals*, are tested on the problem (or set of problems) to determine how *fit*, or well suited, the individuals are to solving that problem. This process is referred to as a *fitness test*. Next, the less fit individuals are removed from the population. Finally, mutation and/or sexual reproduction are applied to the remaining individuals and the results are a new population, or the next *generation*, of potential solutions. A *mutation* is an insertion, deletion, or small change in the code or of an individual. It can also be the flipping of a bits in a vector of 1s and 0s (which is often the format for EC algorithm solutions). For our purposes, we will focus on the first definition. *Sexual reproduction* is when two or more individuals are combined in some way to create a new individual. When these techniques are applied to potential solutions, the original individual would be referred to as the *parent*, and the new individual would be referred to as a *child*. Parents can have more than one child. Depending on the EC algorithms used, entirely new individuals, usually consisting of entirely random code, are introduced into the next population as well. This process is repeated until the *global optima*, the best solution (or solutions) possible, is found, or until the algorithm hits a stopping point. Sometimes an algorithm is not able to find a global optima due to the algorithm not being good enough, so all programs have a designated stopping point to prevent them from running forever. The stopping point can be things like a time limit or a limit on the number of generations that can be produced.

EC has many applications which have seen success in medicine, engineering, and chemistry to name a few. Part of the field's success comes from its versatility – the techniques based on biological evolution can be applied to just about anything. Some examples that use these techniques include genetic algorithms, which are commonly used to generate high-quality solutions to search and optimization problems, ant colony optimization, which can be reduced to finding good paths through graphs, and artificial immune systems, which are computationally intelligent, rule based machine learning systems. In this paper, we will be focusing on a family of algorithms within EC called genetic programing.

2.2 Genetic Programming

Most EC algorithms 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. 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. These *genes* can be thought of as a reorganization of a program for the evolution process. We then modifying those genes, usually with a genetic algorithm, to evolve a program which will perform well on a predefined task. The methods used to encode the programs into *artificial chromosomes*, a sort of structure that holds the genes, and to evaluate the fitness of a program remain active research areas. There are several different GP representations and variations, but we go into more detail on this in Section 3.

2.3 Hyper-heuristics

To better understand hyper-heuristics, we find it helpful to first define heuristic, discuss its purpose, and to work up from there.

A *heuristic* is a function that ranks alternatives in a search algorithm at each branching step and uses that information to choose which branch to follow. It uses *domain knowledge*, or knowledge about the problem, to find solutions more quickly. If we look at the knapsack problem[1], a heuristic might be, given list X, "take the highest value item out of list X and put it into the knapsack. If the knapsack will be overweight, take the next highest valued item instead, etc. Repeat this process until the knapsack is full." The heuristic uses the total weight a knapsack can hold and value of the items to find a solution to this problem – in other words, the heuristic is using domain knowledge to solve the problem.

Metahueristics, often confused with hyper-heuristics, do not require domain knowledge to solve a problem. Metaheuristics can be thought of as general heuristics because a single metaheuristic could be used on many different problems (such as the knapsack problem, scheduling problems, and more). [8]

Hyper-heuristics are heuristic search methods which seek to automate the process of selecting, generating, or adapting several simpler heuristics in order to solve computational search problems. These work indirectly on the solution space and work directly on the space of metaheuristics or heuristics to solve problems [8]. Hyper-heuristic design has two major components; the first is to create a set of algorithmic primitives appropriate for tackling a specific problem class and the second is searching that algorithmic primitive space [1]. *Algorithmic primitives* is a vague term because it is problem dependent; an example of a *primitive* would be how good or bad your board situation is in chess and the *algorithm* would then find ways to use these primitives so that your search algorithm can find better chess moves. Genetic programming is often used for the second part in hyper-heuristic design – hyper-heuristics that use genetic programming are often referred to as Genetic Programming Hyper-heuristics.

2.4 History of Hyper-heuristics

Automating algorithm design has been investigated by several different areas for at least 60 years. In the 1950s, the term "machine learning" was defined as "computers programming themselves" [4]. This definition has changed over time to focus more on the learning aspect, but machine learning is still used to tackle the problem of automating the design of algorithms. While it plays an important role in the history of attempting to automate algorithm design, machine learning is out of the scope of this paper so it will not be discussed further.

[1]the knapsack problem: given a set of items, each with a weight and value, determine the number of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Focusing on the automation of heuristic development, we can trace the beginnings of hyper-heuristics to the 1960s (however, the term 'hyper-heuristic' was not coined until 2000). Early approaches to developing hyper-heuristics focused on automatically setting the parameters of evolutionary algorithms. 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. Many researchers still question which parameters to tune when designing hyper-heuristic systems. Traditionally, parameters are tuned before the evolution starts and controlled during the evolution. There are, however, exceptions. The idea of *self-adaption*, where an algorithm is able to evolve parameters while solving a given problem, emerged later (see Section 4 for an example of this kind of system). [4]

Today there are two major types of hyper-heuristics: *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 [4]. The *components* can be anything from the code of a heuristic function; this means it could be *instructions*, for example adding two numbers together or determining if two strings are identical, or *literals*, such as strings, integers, booleans, etc. We will be focusing on an example of heuristic generation in Section 4.2.

3. GENETIC PROGRAMMING VARIANTS

Genetic programming (GP) is often used for heuristic generation in the hyper-heursitc process. However, there are many different *genetic programming variants*, or variations on the setup of GP algorithms. Does it matter which variants engineers use in their hyper-heuristics? Harris et al. [1] addresses this question; they perform an experiment with five different GP variants to see if the variant chosen affects the success of a hyper-heuristic. The variants tested are Tree-based GP, Linear GP, Cartesian GP, Grammatical Evolution, and Stack-based GP (see Harris et al. [1] for details about each of these variants). All of the listed variants were tested on the Boolean Satisfiability Problem (SAT)[2]; the SAT problem is known for its complexity and NP-completeness[3]. SAT is also useful because many other difficult problems can be reduced to it.

We will discuss the details of stack-based GP due to its relevance in Section 4 and then go over the experiment and results of Harris et al. in Section 3.2.

3.1 Stack-based Genetic Programming

Stack-based GP (SGP) uses data-stacks to manage the input and output of operations. To explain data-stacks, it's helpful to look at an example. If we have the program $((2 + 5) * 2)$, we would first put it into *postfix* notation, which is where any arguments or parameters for a command are stated before that command: `2 5 add 2 mult`. We do this because SGP are represented as linear sequences and postfix notation is a good way to put programs into this format. In the program, `add` means take two numbers off the top of the stack, add them together, and push the result back onto the stack. And `mult` means the same thing except multiply instead of add the numbers.

Input	2	5	`add`	2	`mult`
Stack	2	5 2	3	2 3	6

Table 1: Each column shows an input element and the contents of the Stack after processing that input.

In Table 1, this program is shown being input into a stack. When the stack encounters *terminals*, such as strings, integers, booleans, etc., they are simply pushed onto the stack. When the stack encounters *primitives*[4], which are things like subtraction, string length, greater than, etc. they are executed by taking elements off of the stack and pushing their results back onto the stack. For instance, if we look at the first part of the program, `(2 5 add)`, we would push the 2 and 5 (which are terminals) onto the stack and then execute the add (which is a primitive). The add is executed by popping the top integer off of the stack, which would be 5, and popping the second integer off of the stack, which would be 2, and adding them together. We would then push the result, 3, back onto the stack. If, say, our program had started with `2 add` instead, we would push the 2 onto the stack, and skip the add because there would not be enough arguments to execute it. This means the next part of the program would push 2 onto the stack, and execute the `mult` leaving 4 in the place of the 6 in the table.

At the end of Table 1, we can see that 6 is left on the stack. Generally, the top item on the stack at the end of a program is the answer to a problem that the program is trying to solve. This example was very simple and only dealt with integers. What happens when we have multiple *data types*, such as integers, strings, booleans, and more? Some SGP will push and pop all of their data from one stack, regardless of the data type. But other SGP languages have multiple data-stacks, where there is a stack for each data type. This means that there is an execution stack, where the program is evaluated, and separate stacks that the data is pushed onto and popped off of depending on the data type. To see an example of a SGP with multiple stacks, see Section 4.1.

SGP are useful for hyper-heuristics because of the ability to skip instructions. This means, for example, that if a terminal is deleted from a program during the evolution phase, the program can still execute. SGP are also useful because of a special instruction called "`dup`," which allows the program to duplicate a terminal on the stack. The ability to duplicate data is useful, especially when using a first-in-last-out language where important data can get lost in stacks.

3.2 Results

To determine any differences in performance, the five GP variants were implemented in a common framework which facilitated sharing as much code as possible. This implemen-

[2]Harris et al. actually use a subproblem of SAT called 3-SAT. More information about the problem can be found in [1]

[3]This is why Harris et al. chose SAT. To be *NP-complete* means there is no known polynomial solution, and it is believed that one does not exist. There is more to this, but its not relevant to this paper.

[4]This was defined early in the paper under *algorithmic primitives*, but this definition is different because 'primitive' means different things depending on the context

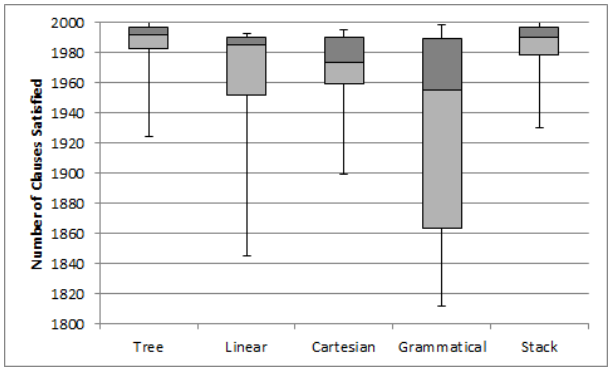

Figure 1: Box plot showing the average number of SAT problem clauses that are satisfied by the best individuals from each run on the test set [1].

tation may cause bias in the results, but this was a necessary risk because the alternative would cause similar bias; the alternative is attempting to maximize the performance of each GP variant at the cost of keeping a common implementation for all of the GP variants [1].

The GP variants were tested on 2000 different instances, or *clauses*, of the SAT problem. First each hyper-heuristic (which is the same except for the GP variant used) is run 30 times on a separate set of SAT clauses to 'train' the hyper-heuristic. After each run, the best individual program from each hyper-heuristic is tested on the 2000 test clauses 3 separate times. These results are shown in Figure 1.

The results from Harris et al. [1] show that the GP variant chosen has a significant impact on the success of the hyper-heuristic. Tree-based GP and stack-based GP performed the best and performed similarly to one another. They solved more variations of the SAT problem on average than the other GP variants. Linear and Cartesian GP performed similarly to each other, but were not quite as good. Grammatical Evolution performed the worst. This does not mean that tree and stack are inherently better than other GP variants – it means that GP variants have different strengths and some are more suited to certain problem spaces than others. More testing is needed to find where each GP variant excels.

4. AUTOCONSTRUCTION

Autoconstruction is a genetic programming hyper-heuristic (GPHH). This means it is a hyper-heuristic that uses genetic programming to evolve programs. In most GPHH, the individual programs are evolving, but everything else is specified by the engineer; in autoconstruction, 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. This means that these programs are responsible for solving a problem *and* responsible for constructing their offspring. An example of this is provided in Section 4.2.

Prior work on autoconstruction has explored a variety of system designs, but, until recently, they have only been able to solve simple problems, such as Scrabble Score, Vector Sums, etc (see [2] for details). A new system called Autoconstructive Diversification of Genomes (AutoDoG) has broken this trend by solving a problem that other genetic programming systems have struggled with: Replace Space with Newline [7]. And, in recent unpublished work, AutoDoG has solved a problem called String Differences that has not been solved by other GP systems before [3].

:exec	1	2	'hi'	string_length	integer_add
:string			'hi'		
:integer	1	2 1	2 1	2 2 1	4 1

Table 2: Each column shows an element on the :exec stack and the contents of the other two stacks after processing that element.

In Section 4.1, we introduce Push, the programming language used by AutoDoG, and Plush, the linear genome format (defined in Section 4.1) of Push that AutoDoG uses. In Section 4.2 we highlight some key features of AutoDoG and in Section 4.3 we go over AutoDoG's recent success.

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 [7]. This is the programming language that AutoDoG uses. Push programs are sequences of instructions, constants, and parentheses with only one syntax requirement: the parentheses must be balanced [5]. For example, `((1 'hello') 2 integer_add string_length integer_gt)` is a simple Push program with its parenthesis balanced.

Instructions are executed by putting them on the `:exec` stack. For example, assume all stacks are empty and assume a program says `(1 2 'hi' string_length integer_add)`. In Table 2, we illustrate the execution of this program (there are more types of data stacks, but we only use two for this example). We push the entire program onto the `:exec` stack to start. Then we push 1 and 2 onto the `:integer` stack. Next we push the string 'hi' onto the `:string` stack. We then encounter `string_length` on the `:exec` stack. The `string_length` instruction takes a string off the `:string` stack and returns the string length. To execute this instruction, we pop 'hi' off the `:string` stack and push the string length, in this case 2, onto the `:integer` stack. Next `integer_add` will try to execute: the first two integers on the `:integer` stack (in this case 2 and 2) are popped off, added together, and the result (in this case 4) is pushed back onto the `:integer` stack. Since there were enough arguments, the instruction executes. [6]

If, instead, the program was `(1 2 string_length integer_add)` then `string_length` would not execute, but `integer_add` still would. What would happen is 1 and 2 would be pushed onto the `:integer` stack and `string_length` would be skipped because there were no strings on the `:string` stack for the `string_length` instruction to use. Then `integer_add` would execute as normal with 1 and 2 instead, which would leave us with 3 on the *:integer* stack in the end instead of 1 and 4.

Plush is a linear genome format for Push [7]. This means that Plush is a format of the Push language that allows the

programs to be stored in linear genomes. *Linear genomes* are an example of an artificial chromosome (see Section 2.2). In Section 4.2, AutoDoG is actually evolving linear genomes and then translating those genomes back into Push programs [7]. This allows for methods of variation in a population to be encoded and evolved more easily. There is much more to Plush and linear genomes, but it is beyond the scope of this paper. For now, we can just think of genomes as the structure which holds programs for the reproduction process. A *random genome* would be a program made from completely random code held in the genome structure and a *blank genome* would be a genome that does not have any program code in it.

4.2 AutoDoG

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

When designing AutoDoG, Spector et al. [7] wanted to maintain diversity in parent selection. To do this, AutoDoG uses Lexicase Selection. Its name comes from the way it filters the population using a kind of "lexigraphic ordering" of cases. 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.

AutoDoG works similarly to PushGP, a reasonably standard genetic programming system, but is run with autoconstruction as the sole genetic operator rather than using human designed mutation and sexual reproduction operators. In PushGP, the rate of mutation and crossover and other parameters are set by the designers. In AutoDoG, instructions for manipulating genomes are created by the designers, but the rate at which these instructions are used and how they are combined changes through the evolution of programs. For example, `genome_uniform_addition` is an instruction which inserts random instructions into a program with a likelihood taken from the top of the `:float` stack. The number at the top of the `float` stack changes as a program executes. So depending on where the `genome_uniform_addition` occurs in the program, the rate of `genome_uniform_addition` may be very different from one program to the next.

In AutoDoG, the reproduction process works as follows: A program that has performed well on the fitness tests is selected for reproduction. We will call this program Mom. Another program which has also performed well is selected and we will call this program Dad. Mom and Dad's genomes are both on the :genome stack. Mom (the program) is run with the purpose of making a child. Part of Mom's code is dedicated to reproduction – for example, she may have `genome_uniform_addition` in her code which would take her genome and insert code into it. So Mom uses her reproduction code, which may or may not take pieces of dad's genome, and Mom makes a new genome. Mom pushes this onto the `:genome` stack. This genome holds the code of her child.

Once this child is constructed, there is the concern of whether or not this child will be passed into the next generation. Spector et al. [7] was especially concerned about cloning when designing AutoDoG. *Cloning* is when a parent makes a child that is an exact copy of the parent and this child moves on to the next generation. This is a major concern with evolutionary systems because if a program performs well enough on the fitness tests to move on to the reproduction phase, making a child that is an exact copy allows the child to perform equally well on the fitness tests. Most programs often do this if allowed because changing the code is risky; if the change is for the worse, then the program's child will not pass its code on to the next generation. Cloning is bad because it prevents the system from exploring more possible solutions, which makes it difficult to find the global optima. It 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. AutoDoG's version of this requires offspring to pass a more stringent diversification test in order to enter the population. This test starts by taking an the new child's genome. This child, named Child, then takes itself as a mate and makes temporary children. If the children differ enough from Child and from each other, Child enters the population and the children are discarded. If Child fails this test, a random genome is generated and the test is repeated on the random genome. If the random genome fails, a blank genome (with no code in it) is generated and inserted into the next generation.

4.3 Results of AutoDoG

One challenging problem AutoDoG has solved is Replace Space with Newline (RSWN). This is a software synthesis problem that takes in a string and prints that string with all spaces replaced by newlines; it also must return the integer count of the non-whitespace characters [2]. This is complex for GP systems because it involves multiple data types, such as strings and integers, and multiple outputs.

AutoDoG does not actually do better than standard PushGP on this problem. AutoDoG solves the RSWN problem 5–10% of the time, where PushGP solves it about 50% of the time [2]. AutoDog's success 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 found solutions to problems, such as String Differences, that have never been solved with regular PushGP (or any other GP system) [3].

AutoDoG is unique among autoconstructive systems in that it can solve problems that are challenging for other autoconstructive and GP systems[5]. However, as stated by Spector et al. [7],

> We do not know which of these [AutoDoG's features], 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

[5]These problems are not difficult for people to solve – in fact they are fairly easy problems taken from introductory level textbooks. However these problems are difficult for GP systems to solve and serve as a set of benchmarks for GP systems to reach [2]

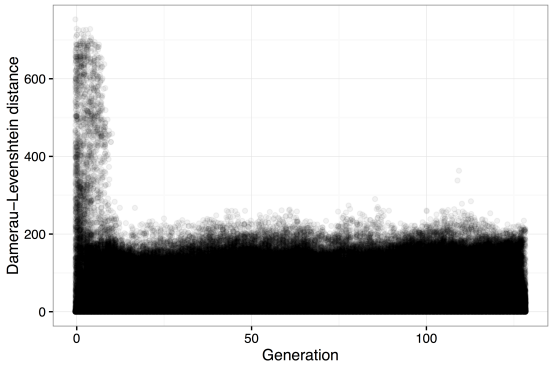

Figure 2: DL-distances between parent and child during a single PushGP run on RSWN [7].

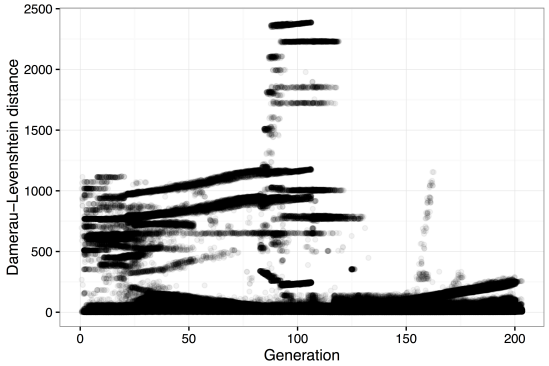

Figure 3: DL-distances between parent and child for a single autoconstructive run on RSWN [7].

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.

One interesting thing to note about AutoDoG is how it evolves. In standard PushGP, one can see a steady incline in Demerau-Levenshtein distances (DL-distance) between the genomes of parents and their children in Figure 2. A *DL-distance* can be thought of as a measure of change between a parent and child over the course of a generation. There is more to DL-distances than this, but this simple definition suffices for the purposes of this paper. Over time, change occurs at a steady rate and is not very dramatic from one generation to the next in PushGP. This is because engineers set the parameters pertaining to biological evolution before they run the program. In AutoDoG, there are big differences between the genomes of some parents and their children as shown in Figure 3. This is likely due to the fact that the programs are evolving the parameters pertaining to biological evolution. This is interesting because the methods for evolution are encoded into the genomes in AutoDoG; these graphs show us how evolution is affected from one generation to the next. In the AutoDoG graph, we are essentially seeing evolution evolve.

5. CONCLUSIONS AND FUTURE WORK

Automating algorithm design has been undergoing research for at least the last 60 years. Hyper-heuristic optimization is a field that has devoted a lot of attention to this subject. Hyper-heuristics are heuristics that seek to automate the process of selecting, generating, or adapting simpler heuristics in order to solve computational search problems. Genetic programming hyper-heuristics, hyper-heuristics that use genetic programming for evolution, are making real progress in the field.

However, according to Pappa et al. [4] in 2014, machine learning was ahead of hyper-heuristics optimization on the subject of automating algorithm design and "can operate over different datasets, from different problem domains, and even with different features." This is not to say that hyper-heuristic optimization is no longer useful – using autoconstruction, a genetic programming hyper-heuristic for heuristic generation, engineers have had recent success solving the String Differences problem; this is exciting because it is a problem which has not been solved by a genetic programming system before [3].

It would be interesting to see machine learning techniques combined with hyper-heuristics. Cartesian genetic programming (one of the GP variants in harris et al. [1]) has a similar structure to neural networks, a type of machine learning technique. It might be interesting to try and combine the two structures. It would also be useful to find out what search spaces each genetic programming variant excels in – Harris et al. mentions further investigation in the end of their paper, so we may see work on this in the near future.

Acknowledgments

Special thanks to Nic McPhee and Elena Machkasova for their time, feedback, and constructive comments.

6. REFERENCES

[1] S. Harris, T. Bueter, and D. R. Tauritz. A Comparison of Genetic Programming Variants for Hyper-Heuristics. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, pages 1043–1050, New York, NY, USA, 2015. ACM.

[2] T. Helmuth and L. Spector. General Program Synthesis Benchmark Suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1039–1046, New York, NY, USA, 2015. ACM.

[3] E. Moshkovich. private communication.

[4] 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.

[5] 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.

[6] L. Spector and N. F. McPhee. Expressive Genetic Programming: Concepts and Applications. In

Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion, GECCO '16 Companion, pages 589–608, New York, NY, USA, 2016. ACM.

[7] L. Spector, N. F. McPhee, T. Helmuth, M. M. Casale, and J. Oks. Evolution evolves with autoconstruction. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, GECCO '16 Companion, pages 1349–1356, New York, NY, USA, 2016. ACM.

[8] D. R. Tauritz and J. Woodward. Hyper-Heuristics. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, GECCO '16 Companion, pages 273–304, New York, NY, USA, 2016. ACM.