Automating Algorithm Design through Autoconstruction

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

ABSTRACT

Algorithm design can be difficult and time consuming; because of this, since at least the 1950s, engineers have been trying to automate the design of algorithms. One newer approach to this problem is autoconstruction. This is a type of genetic programming hyper-heuristic that evolves programs which can evolve programs to solve problems. A new system called AutoDoG, which uses autoconstruction to automate part of the algorithm design process, has recently solved a problem that other autoconstructive and genetic programming systems have struggled with: Replace Space With Newline. This recent success is promising for AutoDoG, and autoconstruction in general, as an effective means for automating the design of algorithms.

Keywords

Evolutionary Computation, Genetic Programming, Hyper-heuristics, Autoconstruction

1. INTRODUCTION

rewrite entire introduction...

Algorithm design can be difficult and time consuming; because of this, since at least the 1950s, engineers have been trying to automate the design of algorithms by reducing the amount of work people put in to the design process and increasing 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 [3]. We will be focusing on hyper-heuristic optimization in this paper.

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. 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.

When designing hyper-heuristics, engineers can use several different kinds of algorithms to automate the process of selecting, generating or adapting simpler heuristics. When genetic programming (GP) is used for this process, we refer to these as genetic programming hyper-heuristics. GP is a family of algorithms within Evolutionary Computation (EC), a subfield of Artificial Intelligence. EC encompasses algorithms that use techniques modeled after biological evolution to solve problems. GP uses these techniques to specifically evolve programs which then solve problems. There are many variations of GP, and the variant used may affect the success of the genetic programming hyper-heuristic.

Traditionally, the system designer determines how GP solutions evolve [6]. In a newer technique, called autoconstruction, the methods for evolution are evolving as well. Autoconstruction can be thought of as a more complex genetic programming hyper-heuristic.

The rest of the paper is organized as follows. In Sections 2.1, 2.2, and 2.3 we describe background necessary 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 outline an experiment that tests different GP variants used in hyper-heuristics and we go into detail on stack-based GP (a genetic programming variant). Next, we will go over current research being done with stack-based genetic programming in Section 4; we briefly discuss Push, a stack-based programming language, along with a technique for automating algorithm design called autoconstruction. Finally we will go over the results of the recent research in Section 4.3 and some conclusions in Section 5.

2. BACKGROUND

In this section, we introduce a lot of terminology used throughout the rest of this paper. We also briefly go over the history of hyper-heuristic development.

2.1 Evolutionary Computation

Evolutionary Computation (EC) is a subfield of Artificial Intelligence that uses algorithms modeled after biological evolution to evolve potential solutions to problems. We will describe the general process of an EC algorithm and define the basic terminology. Note that this is not how all EC algorithms work – it depicts the most common process.

These algorithms start with a group, or *population*, of randomly generated potential solutions to a problem or set of problems. These potential solutions, or *individuals*, are often in the form of 1s and 0s; these 1s and 0s represent *genes* which are being turned on or off.

is this okay?

We measure the quality of these individuals using a fitness function. This process is referred to as a *fitness test*.

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

Next a *selection method* is used to pick individuals for reproduction, based on the results of the fitness test; this means the more *fit* individuals, or higher quality solutions, are the ones selected for reproduction. We then apply operations like mutation and crossover on these individuals to build new solutions. A *mutation* is an insertion, deletion, or small change in an individual and *crossover* is when two or more individuals are combined in some way to create a new individual. When these operations 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, random individuals may be introduced into the next population as well. The next population would be referred to as the next *generation* of potential solutions. This process is repeated on this new generation. We continue to repeat this process until the *global optima*, the best solution (or solutions) possible, is found, or until the algorithm hits a stopping point, such as a time limit or a limit on the number of generations. Stopping points vary from system to system and are implemented to make sure the algorithm run time is reasonable – an algorithm that can find the global optima, but takes a year to do so, is not a very good algorithm.

EC has many applications in medicine, engineering, and chemistry to name a few. Part of the field's success comes from its versatility – some examples of EC algorithms include genetic algorithms, which are commonly used to generate high-quality solutions to search and optimization problems, ant colony optimization, which can aid in 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.

move paragraph to intro?

2.2 Genetic Programming

Most EC algorithms produce solutions in the form of a set of answers to a problem or set of problems. In genetic programming (GP), our solutions are in the form of programs – more specifically, they are usually in the form of trees of operations and values which represent programs.

Jarvis thought I should this, should I drop it?

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.

One common way that GP algorithms work is 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 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 structure that holds the genes, and to evaluate the fitness of a program remain active research areas.

2.3 Hyper-heuristics

Before we can understand hyper-heuristics, we must define heuristic. 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 a list of items 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 knapsack cannot hold any more of the remaining items without becoming overweight, or until you are out of items." The heuristic uses the total weight a knapsack can hold and value of the items to find a solution to this problem.

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 heuristics to solve problems [7]. Hyper-heuristics also work on the space of *metaheuristics*, which are more generalized heuristics that do not need domain knowledge to solve the problem; in other words, metaheuristics are not problem specific [7]. 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; for example, when playing chess, your *primitives* would the set of possible moves for each of your chess pieces for an individual board state. The *algorithm* would then find ways to use these primitives to determine the best move to make in that board state. In hyper-heuristics, genetic programming is often used to search through the algorithmic primitive space [1].

I'm not actually sure this is correct–the Harris et al. [1] paper states "The first step in employing a hyper-heuristic is creating a set of algorithmic primitives appropriate for tackling a specific problem class. The second step is searching the associated algorithmic primitive space. Hyper-heuristics have typically employed Genetic Programming (GP) to execute the second step, but even in GP there are many alternatives." I'm not sure this is correct because wouldn't GP be generating the primitives AND searching that space? or are the primitives supposed to be parameters and GP searches through all possible combinations of those parameters in order to find a solution....?

2.4 History of Hyper-heuristics

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 and crossover rates (numeric values?), however the definition has expanded to include evolutionary algorithm components like selection mechanisms, and mutation and crossover operators (instructions themselves?). Many researchers still question which parameters to tune when designing hyper-heuristic systems.

[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.

I realize this isn't the best description, however I am pretty sure I have to define what a parameter is... And the problem is that definition has changed over time and I wanted to somehow capture that because I really like the history of hyper-heuristic development. I'm not sure the history really adds much to my paper though...

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 in the 1990s. Self-adaption first focused on evolving parameters, but later expanded to evolving operations. In the early 2000s, the definition of self-adaption changed again to include systems that evolve entire algorithms. [3]

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 [3]. 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.

Elena said to define components in a previous draft. You said to drop this... I know I define it better later, but how do I handle this section then?

We will be focusing on an example of heuristic generation in Section 4.2.

3. GENETIC PROGRAMMING VARIANTS

In genetic programming hyper-heuristics, genetic programming (GP) can be used for heuristic and program evolution. However, there are many different *genetic programming variants*, or variations on the representation and setup of GP algorithms. Does it matter which variants engineers use in their hyper-heuristics? Harris et al. [1] addresses this question; they performed an experiment with five different GP variants to see if the variant chosen affects the success of a hyper-heuristic. The variants tested were 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 $((2+1)*2)$, we would first put it into *postfix* notation, which is where any arguments or parameters for a command are stated before that command: `2 1 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	1	`add`	2	`mult`
Stack	2	1 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 1 add)`, we would push the 2 and 1 (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 1, 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 (see Section 4.1 for example), 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.

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.

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 implementation causes bias in the results, but this was selected to maintain a common implementation. The alternative, attempting to maximize the performance of each GP variant

[2]Harris et al. actually uses a subproblem of SAT called 3-SAT. More information about the problem 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.

[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.

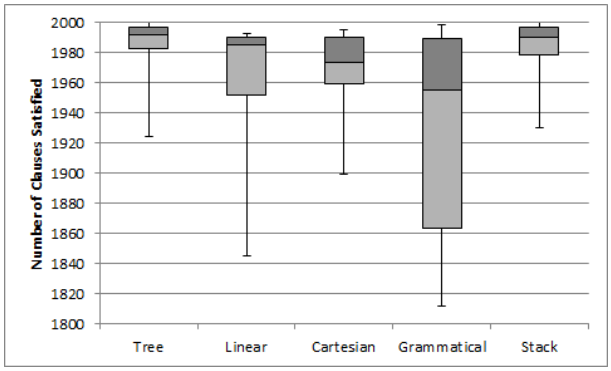

Figure 1: Box plot showing the average number of SAT problem clauses that are satisfied by the best individuals from each run on the test set [1].

at the cost of keeping a common implementation for all of the GP variants, would cause similar bias [1].

The GP variants were tested on an instance of the SAT problem. Specifically, the hyper-heuristics (using these GP variants) were evaluated on all 2000 subproblems, or *clauses*, within the instance given to determine how well each hyper-heuristic solves the 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 clauses 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 determine how the GP variant used can cause a hyper-heuristics system to excel.

4. AUTOCONSTRUCTION

Autoconstruction is a genetic programming hyper-heuristic (GPHH) that uses genetic programming to evolve programs which have heuristic functions available to them to select the best program possible for a problem.

In most GPHH, the individual programs are evolving, but everything else is specified by the engineer. This means engineers might set half the population to produce children through a mutation that inserts an *element*, a literal or instruction, every ten elements in a program. And they might set the other half to experience crossover where the first half of the code for the child comes from one individual and the second half comes from another individual. In autoconstruction, engineers don't specify how programs construct their offspring. Instead, the methods of variation are encoded into the programs that are evolving so that, as a program evolves, its variation methods also evolve. An example of this is provided in Section 4.2.

: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.

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 many genetic programming systems have struggled with: Replace Space with Newline [6].

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 [6]. Push programs are sequences of instructions, constants, and parentheses with only one syntax requirement: the parentheses must be balanced [4]. 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. [5]

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 [6]. 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 [6]. This allows for methods of variation in a population to be encoded and evolved more easily. We can just think of genomes as the structure which holds programs for the reproduction process.

4.2 AutoDoG

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

When designing AutoDoG, Spector et al. [6] 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 operators, like muatation and crossover. In PushGP, the rate of mutation and crossover and other parameters are set by the designers. In AutoDoG, the building blocks for instructions that manipulate genomes are created by the designers, but the rate at which these instructions are used and how they are combined into higher level operations changes through the evolution of programs. One example of a building block would be `genome_uniform_addition`; this is an instruction which inserts random instructions or literals 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 instruction/literal insertion may be very different from one program to the next. There is an ongoing discussion about how much guidance to give the AutoDoG system. Some of the designers think building blocks like `genome_uniform_addition` are giving the system too much instruction – some think we should not give any building blocks and should let AutoDoG evolve all instructions from scratch. The problem with evolving everything from scratch is that the system would take much longer to develop solutions and may not develop solutions at all.

In AutoDoG, the reproduction process works as follows: A program that has performed well on the fitness tests has been selected, using Lexicase Selection, for reproduction. We will call this program Mom. Another program has also performed well and is selected – 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. [6] 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 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.

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 into the next generation. The child genome, created by Mom earlier, contains the code of a program that we will call Pat; this test begins by having Pat take itself as a mate (the way Mom took Dad as a mate). Pat is run and makes temporary children. If the children differ enough from Pat and from each other, Pat will be inserted into the next population and the children will be discarded. If Pat fails this test, a random genome, meaning a genome with randomly generated code in it, is created 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 (this blank genome does not undergo the diversification test). This test allows for a more diverse set of potential solutions and allows exploration of the problem space.

4.3 Results of AutoDoG

One challenging problem AutoDoG has solved is Replace Space with Newline (RSWN). This problem is taken from an introductory level textbook and is part of a GP benchmark suite of problems used to evaluate how successful a GP system is. RSWN 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 solves the RSWN problem 5–10% of the time, whereas PushGP solves it about 50% of the time [2]. Why do we care about AutoDoG when a relatively standard GP system performs much more reliably? We care because autoconstruction automates more of the algorithm design process than other GP systems, so while it may not perform as well on RSWN, it performs better on the problem of automating algorithm design; 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 construct offspring as well as learn how to solve problems.

AutoDoG is unique among autoconstructive systems in that it can solve problems that are challenging for other

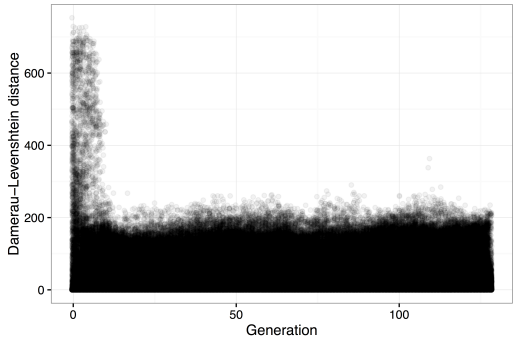

Figure 2: DL-distances between parent and child during a single PushGP run on RSWN [6].

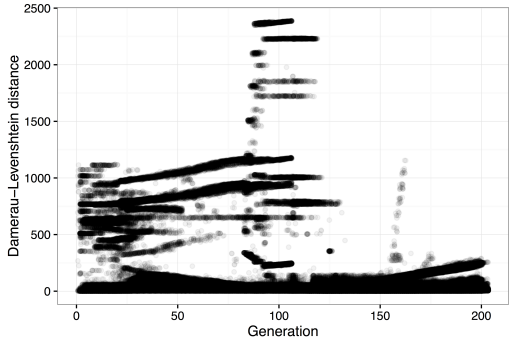

Figure 3: DL-distances between parent and child for a single autoconstructive run on RSWN [6].

autoconstructive and GP systems[5]. However, as stated by Spector et al. [6],

> 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 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 Damerau-Levenshtein distance (DL-distance) between the genomes of parents and their children in Figure 2. A *Damerau-Levenshtein distance* can be thought of as a measure of change between the genomes of a parent and child over the course of a single generation. In Figure 2, over the 150 or so generations, change occurs at a steady rate and is not very dramatic from one generation to the next – the most change that occurs is about 200 differences between a parent and child. This is because engineers set the parameters pertaining to biological evolution before they run the program. The scattered dots near the first generation occurs because the first set of individuals is randomly generated. In AutoDoG, there are big differences between the genomes of some parents and their children as shown in Figure 3. Notice how the y-axis goes up to 2500 DL-distance in the AutoDoG graph compared to only about 800 in the PushGP graph. This is due to the fact that, in AutoDoG, the programs are evolving the parameters pertaining to biological evolution. In the AutoDoG graph, we are essentially seeing evolution evolve.

5. CONCLUSIONS AND FUTURE WORK

Automating algorithm design is a complex problem that has been investigated for over 60 years. We have recently automated more of the algorithm design process through the use of genetic programming hyper-heuristics, specifically using a technique called autoconstruction. These techniques are important because the closer we come to systems that can reliably generate algorithms from scratch, the closer we come to drastically changing computer science; we could potentially solve problems people have never solved before. At the very least we would not need programmers to write simple algorithms anymore because the computer would do this job using genetic programming hyper-heuristic systems, which would change the programming industry.

However, according to Pappa et al. [3] 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 machine learning is better than hyper-heuristic optimization, the two have different methodologies and ways of approaching the problem of automating algorithm design. Pappa et al. states that the next step is working on getting hyper-heuristic optimization to the same level. They state that both heuristic and algorithm selection/generation are useful for all types of domains in which many parameters and methods are available, but no clear criteria or methodology exists for selecting them.

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.

[5]These problems are not difficult for people to solve – they are 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]

Acknowledgments

Special thanks to Nic McPhee and Elena Machkasova for their time, feedback, and constructive comments. And thanks to Alex Jarvis and Michael Bukatin, my external reviewers.

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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.