Evolving Software Applications using Genetic Programming PushCalc: The Evolved Calculator

Kwaku Yeboah-Antwi
Institute for Computational Intelligence
Hampshire College
Amherst, MA
ky10@hampshire.edu

ABSTRACT

This paper describes *PushCalc*, a system that evolves a Calculator, a complete software application. PushCalc is a modified version of Clojush, the clojure implementation of the PushGP genetic programming system[1]. PushCalc supports the definition and storage of names and functions via its naming mechanism, tags. The goal of this system is to utilise this ability to evolve an individual that can create the modular parts of the calculator and also know when and in what situations to use which modular functions and perform the correct operations depending on the input given to the system.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program synthesis*

General Terms

Algorithms

Keywords

Genetic Programming, Dynamically Scaling Genetic Operator Usage, Variable Point Size System, Gecco proceedings, Evolving Software Applications

1. INTRODUCTION

Genetic programming has always been concerned with evolving a population of computer programs to solve problems. These computer programs are traditionally represented as syntax trees and are composed of primitive functions and operands. These computer programs are evolved to generalize from sample input and be able to perform a general task and do it very well. They are in effect, just modular programs/functions that can be integrated into other bigger software applications. This research project is concerned with the broader view of evolving whole and complete software applications that are made up of modular parts. Instead of individually evolving all the parts of the application, the system defines and evolves named functions, automatically defined functions(ADF)[4] that will then be put together to make a complete software application. The concept of naming and defining functions in genetic programming runs has been investigated and implemented in a few genetic programming systems. These systems however require the human to pre-specify in advance, the number and types of names[3] and when they don't, very complex operators are required to ensure all names are defined before they're used and that calls to functions match their definitions[4]. Spector, Harrington, Martin and Helmuth (2011)[2] described and implemented a new approach to naming in genetic programming which allows for automatic definition of functions and names by the gp system during a run. They called their approach the tag based system. The system allows tags to be applied to variables and code and when the tag values are called, they then refer back to the variables/code that the tag was applied to. Tags references implement an inexact matching system where if the tag number does not exist, the closest matching tag gets called thereby making sure a tag value will always return something. The tag system has been implemented in PushCalc and as such allow the genetic programming system to automatically define and name its own functions. This project is based on the hypothesis that, since the PushCalc genetic programming system evolves modular programs and has the capability to automatically define and name these modular functions, we can therefore replicate the software development process by evolving a complete software program that will be a sum of different named modular programs.

2. THE *PUSHCALC* SYSTEM

This software application is evolved using a modified version of the clojure [2] implementation of the PushGP genetic programming system called PushCalc. The PushCalc system evolves programs written in the Push programming language. Push is a strongly typed tree based language with no syntactic constraints. Each data type in Push has its

[1]The PushGP Genetic programming system was invented by Lee Spector. It can be found at http://hampshire.edu/lspector/push.html

[2]http://clojure.org/

GECCO'12 Companion, July 7–11, 2012, Philadelphia, PA, USA.

own stack. There is also a stack for "code" data and code from programs being evolved reside on this stack. This allows Push to support and allow recursion and automatic definition of functions without any human intervention or pre-specification. The code stack also allows programs to perform operations and manipulate all their code or fragments of their code.

PushCalc utilizes the standard stacks for the boolean, integer and string types. A code and exec stack is also utilized to allow for manipulation of program code. A new stack called the symbol stack was created for the storage and manipulation of the set of standard primitive mathematical operators(or symbols), ie `(+ - * /)`.

The function set utilized by programs is listed below.

Table 1: The set of primitive functions that are utilized in program

Types	Default Instructions Used
integer	add eq swap pop dup lt mult div gt max sub mod rot min
boolean	swap eq rot and not or frominteger dup pop
exec	pop eq if when swap dup noop
code	member do* dup quote cons container if extract wrap nth discrepancy size length cdr map atom contains list do*range eq fromboolean frominteger do*count car position do do*times rot
string	pop eq map rot toInt toSymbol oper? concat_Int swap dup
symbol	pop plus sub div mult
tag	tag-instruction-erc tag-when-instruction-erc untag-instruction-erc

3. TECHNIQUES

A number of techniques were created and implemented in this system to reduce program sizes by controllin bloatand helping speed up therat at which succesful individuals are evolved. These techniques are presented below.

3.1 Variable MaxPoints System

. The first technique that was created ad implemented is the **V**ariable Max**P**oints **S**ystem(VPS) PushCalc utilizes the VPS technique that allows variable program sizes in the population and penalises individuals depending on how much bigger they are than other members in the population. For each generation, the average size of the best individual from each of the past 10 generations is calculated(S_b) and during selection, programs are allowed to grow to a limit of maximum size(S_{max}) of

$$S_b + S_r$$

(Sr = size-radius, the max number of points individual sizes can be bigger than max-points)

The average size of the top 10% best individuals in the population is calculated(S_{pop}) and subtracted from the size of each individual. The resulting resulting value is then added to the total error of each individual to get its size-error. During selection, individuals with the lowest size-errors are selected. Code bloat is severely curtailed because increase in size without a corresponding increase in fitness as compared to the other individuals increases the fitness penalty of that individual and individuals are rewarded with a better fitness by growing bigger only when it is necessary. Also, individuals can only grow to be S_r bigger than the max program size. In effect, the population adapts to increasing its max program size only when the increase in size benefits the whole population.

3.2 Dynamically Scaling Genetic Operator Usage

The other technique also created and implemented is the **D**ynamically **S**caling **G**enetic **O**perator **U**sage(DSGOU). This technique scales the probability of each genetic operator being chosen in order to breed the child-individuals depending on how many past individuals have been bred by the operator. Each operator has a standard default probability lower limit(OP_{min}). The number of best individuals generated by each mutation operator(M_{best}) is kept track of and the probability of that operator being chosen to breed child programs is calculated using this equation.

$$\begin{aligned} & (1.0 - (\text{number of operators} \times OP_{min})) \\ \times \quad & (\text{generation number} \div M_{best}) \\ + \quad & OP_{min} \end{aligned}$$

where OP_{min} is the default probability limit and M_{best} is the number of best individuals generated the genetic operator.

This ensures that all operators have at least an OP_{min} chance of being chosen to breed the child program and that, operators that produce better individuals have a higher chance of being chosen to breed the child programs. Preliminary research has shown that for each variable size range(the time it takes for the population to increase or decrease max-points ($\pm S_r$), one operator usually generates the most best individuals during that period. This technique capitalizes on this phenomena and increases the probability of that operator being selected to breed a child during that period thereby ensuring that, the operator generating the best individuals during a period gets used more during that period.

4. SELECTION

During selection, individuals with the lowest error rate after VPS is applied, are selected to breed thereby making sure that individuals that are doing very well at solving most of the problems and with the least amount of bloat get selected to pass on their genome.

5. FITNESS FUNCTION

The fitness function takes in a map of test data. This map contains keys of a list of inputs. These inputs are key presses fom a calculator. The key presses are represented as single characters corresponding to the value of the key

pressed; for example, the operation "3+4" is represented by three keypresses, "3" "+" "4". These sample sets of inputs are all mapped to their corresponding resulting values which are the result of running the key presses specificed in each key. During evaluation, individuals in the population being evolved are executed on a new stack state that has been initialized with the inputs popped onto the string stack in the order of when the keys were pressed, ie. first keypress being pushed onto the string stack first and so on. The fitness of each indiviual is the difference between the final item on the integer stack after executing the individual and the pre-specified expected corresponding output of the initialized input from the map of test data. Each individual is evaluated at the end of every generation and if no individual exists with a total error of 0, a new generation is then initialized with a population bred from the population of the previous generation.

The test cases are composed of possible combinations of the buttons on a calculator (eg. "43" "43+3" "27/4" 45*2-5" ,etc)

6. RESULTS

Four different experiments with different instruction sets were created.

The first experiment, titled *PushCalc-ALL* enabled access to all the instruction sets/primitives listed above for all the programs. The VPS and DSGOU techniques were also enabled. The population was also allowed access to the tagging mechanism present in PushCalc.

The second experiment, titled *PushCalc-NONE* also enabled access to all the instruction sets/primitives listed above except for the tagging instructions for all the programs. The VPS and DSGOU techniques were disabled and the population had no access to tagging mechanisms and as such tags did not exist in this experiment.

The third experiment, titled *PushCalc-TagOnly* enabled access to all the instruction sets/primitives listed above for all the programs. The VPS and DSGOU techniques were disabled but the population had access to the tagging mechanism in PushCalc.

The fourth and final experiment, titled *PushCalc-NoTag* enabled access to all the instruction sets/primitives listed above except for the tagging instructions for all the programs. The VPS and DSGOU techniques were also enabled.

Each experiment was allowed a maximum population of 1000 individuals and a max-generation of 1000.Each population also had an initial program size limit of 100. Numerous runs were done for each experiment and the data for the runs for each experiment were averaged and are presented in Table 2.

Table 2: Standard GP runs for the different experiments

Experiment	Average Best Individual Size	Average Best Initial Error	Average Best Total Error at end of run
ALL	42	635457	3675
NONE	80	782784	34146
NOTAG	32	772780	16968
TAGONLY	62	690435	3015

Seperate runs were also done to measure the effects of each of the techniques that were crafted to help improve the fitnesses of individuals in the population. The average change in error-rate per generation were calculated for the populations that used each technique and this data is also presented in Table 3.

Table 3: Average rate of increase in fitness per generation for each technique

Technique	Average fitness increase per generation
DSGOU	668.149
VS	782.548
DSGOU-VS	752.744
NONE	687.42

The average size of individuals that used each of the techniques was also calculated and is presented in Table 4.

Table 4: Average size of individuals in population for each technique

Technique	Average size of individuals
DSGOU	86
VS	53
DSGOU-VS	44
NONE	88

Since there were no succesful individuals in any of the experiments, the techniques were also tested on the "Dirt-Sensing, Obstacle-Avoiding Robot" problem presented in [6]. The results are presented in Table 5.

Table 5: Standard GP run for techniques on Dirt Sensing Obstacle Avoiding Robot Problem

Techniques	Average Best Individual Size	Average Best Generation
DSGOU	40	69
VS	180	21
DSGOU-VS	149	27
NONE	147	108

7. CONCLUSION

We have described four experiments that were used to test the hypothesis that complete software applications can evolved by a genetic programming system that has the ability to automatically evolve and name its own modular functions.

No individuals in any of the populations were able to evolve a complete calculator within 1000 generations. Individuals that had access to the tagging mechanism present in PushCalc had smaller average initial error rates than individuals without tags and these individuals also succeeded the most at evolving the calculator by having the smallest errors. At the end of 1000 generations, individuals with access to the tagging mechanism had error rates that were over 400% smaller than individuals that had no access to the tagging mechanism. This supports the hypothesis that tags would be very useful in solving the problem.

The data also proved that the merits of the new techniques that were implemented to help improve the genetic programming system. As can be seen Table 3, the two new techniques increased the problem solving abilities of invdividuals as compared to the standard genetic programming system. VPS accounted for the highest increase in fitness per generation followed b a combination of the VPS and DSGOU techniques. The VPS techniques average fitness increase per generation was approximately a 100 points bigger than the standard genetic programming system. Individuals that were in populations that utilised these techniques were also found to have a smaller size as shown in Table 4. Infact, individuals in populations that utilised a combination of both VPS and DSGOU had an average size that was half that of individuals in the standard population.

The techniques were also tried out on the Dirt Sensing Obstacle Avoiding Robot(DSOAR) problem[6] with a grid of 8x8. This problem seeks to evolve a program that causes a robotic mopper to mop the floor in all of the squares in a grid. The grid has irregularly placed objects through which the robot cannot move. The robot has sensors that allows it tell which adjacent grids have dirt and or obstacles. As seen in Table 5, the two new techniques allowed the genetic programming system to evolve programs of smaller sizes that were succesful in the least amount of generations. Infact, a combination of the two techniques resulted in individuals that were more succesful in 1/4th the number of generations individuals in the standard gp system took to solve this problem. The succesful individuals in populations utilising DSGOU were 1/3rd the size of succesful individuals in the standard gp population.

These results demonstrate that the VPS and DSGOU techniques are very useful and can help evolve better populations of individuals who are less bloated and solve problems faster than individuals in the standard genetic programming system.

I beleive that the shown success of tags in evolving individuals that have fitnesses that over 5 times better than individuals that do not have access to tags and the demonstrated benefits from the VPS and DSGOU techniques mean, that the problem is solvable and thus, my next steps are focused on continuing the experiment.

8. FUTURE WORK

Future work is going to explore how the problem is represented to the system. Currently as detailed above, the system is initalized with the inputs as characters. Future work will focus on representing the input as tags. All the buttons on the calculator will be randomly tagged at the beginning of the experiment and the tags will be evenly spread out across the tag space. The fitness function will take an individual, and run the individual and then will have the tag referring to the first keypress pushed onto the exec stack and run. After that tag has been run, the tag for the next keypress will be also pushed onto the exec stack and also run. This process will be repeated until all tags have been run and the fitness will be then calculated. This alternate way of representing the problem to the system is more inline with how calculators work(ie. operations are peformed before being applied to the next input being given).

Another future avenue that I'm planning to explore is developing a co-evolution system where there are populations for each of the fitness cases. These populations will specialize in each fitness case and at the end of each generation will be allowed to interact with the other populations. The best individuals in each population will then be selected and added to a master population that will be applied to all the fitness cases and which should hopefully allow for generalization.

9. ACKNOWLEDGEMENTS

I am grateful to Lee Spector of the Hampshire College School of Cognitive Science and the members of the Institute for Computational Intelligence for providing me with this great opportunity of working with them.

This material is based upon work supported by the National Science Foundation under Grant No. 1017817. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

10. REFERENCES

[1] J. Klein and L. Spector. 3d multi-agent simulations in the breve simulation environment. In M. Komosinski and A. Adamatzky, editors, *Artificial Life Models in Software*, pages 79–106. Springer London, 2009.

[2] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Complex Adaptive Systems. MIT Press, 1992.

[3] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[4] J. R. Koza, F. H. Bennett, D. Andre, and M. A. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann, May 1999.

[5] J. Rosca. Generality versus size in genetic programming. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 381–387. MIT Press, 1996.

[6] L. Spector. Advances in genetic programming. chapter Simultaneous evolution of programs and their control structures, pages 137–154. MIT Press, Cambridge, MA, USA, 1996.

[7] L. Spector, K. Harrington, B. Martin, and T. Helmuth. What's in an Evolved Name? The Evolution of Modularity via Tag-Based Reference. In *Genetic Programming Theory and Practice IX*. Springer New York, 2011.