Evolving Compiled Code

Eric C. Collom
University of Minnesota, Morris
coll0474@morris.umn.edu

ABSTRACT

Stuff

General Terms

Keywords

1. INTRODUCTION

The limitation of only being able to evolve small parts of programs is an issue in evolutionary computation (EC). Much of these constraints are due to there is no graceful way to evolve programs, as a whole, directly at the source code. This is because source code exists to make it easier for humans to read and write programs. However, it is very hard to create a genetic algorithm (GA) that would evolve source code do to syntactical constraints.

A solution to this problem is to to compile the program to bytecode or machine code and evolve it there. This has been done in Java bytecode and x86 assembly [5] [6] which I will discuss thoroughly in this paper add more.

2. BACKGROUND

2.1 Evolutionary Computation

Definition of genetic operator?

2.1.1 Crossover

2.1.2 Mutation

Mutate-insert,mutate-delete,mutate-swap

2.1.3 Fitness

2.1.4 Tournament Selection

2.2 Assembly

Machine code that runs on a physical machine.

UMM CSci Senior Seminar Conference, April 2014 Morris, MN.

2.3 Java Bytecode

Java bytecode is compiled source code that runs on the Java Virtual Machine(JVM). It "is the intermediate, platform-independent representation of Java programs[1]." Each bytecode operation code (opcode) is one byte in length. However, some opcodes, that take parameters, are multiple bytes long[3]. There are more higher-level languages than just Java that compile into Java Bytecode. These languages are Scala, Groovy, Jython, Kawa, JavaFx Script, and Clojure[5].

Example of Bytcode here

2.4 Grammar Stuff

Talk about syntactic and semantic constraints and BNF.

3. WHY EVOLVE MACHINE CODE?

There are many problems we run into when trying to evolve source code. One problem is that it is extremely difficult to evolve an entire program due to source code syntactical constraints. Another problem is that we cannot just take a program and evolve it, we have to design the program to be evolvable. Evolving bytecode and x86 bypasses both of these problems.

```
float x;
int y=7;
if(y>=0)
    x=y;
else
    x=-y;
System.out.println(x);
```

3.1 Source Code Grammar Constraints

One problem with trying to evolve entire programs at the source code level is that there is a very high risk of producing a non-compilable program. This is due to the fact that high-level programming languages are designed to make it easy for humans to read and write programs. Most high-level programming languages are defined in Backus-Naur Form (BNF) which is purely syntactical[2][5]. This means that the grammar does not represent the semantic constraints of a program. For example, the BNF form does not capture the languages type system, variable visibility and accessibility, and other constraints [5]. In order to write a program to evolve source code we would have to deal with all these constraints. While this task is possible it would require writing a full-scale compiler which would be a very taxing task.

Java bytecode and x86 Assembly contain a small alphabet of primitives [6]. Also they are a lot less syntactical.

This means that there is a lot less risk of producing a non-compilable program during evolution. Sipper et al.[5] simply deals with the cases where a non-compilable program is produced by simply re-evolving the programs. Since, the check to see if a program is compilable is cheap enough and the rate of which a program produced is compilable is high enough this is feasible.

3.2 Evolve Entire Program As Is

For traditional EC, a program has to be designed specifically to be evolved. Evoling, bytcode or x86 doesn't require the program to be designed specifically for evolution [6] [5]. Thus, any language that can compile into either bytecode or x86 can be evolved. The only thing required is an algorithm to compute the fitness. Also, the entire program can be evolved. In traditional EC when a program is designed it is more common to evolve an expressions or formulae [5] within the program. Most of the program as a whole would be already written and remain the same after evolution. In evolving bytecode and x86 most of the program doesn't even need to be written to be able to evolve it and find a solution. Also, the final product can be drastically different from the original program since the entire program was evolved. This allows for a lot more flexibility in evolving a programs. This also allows us to solve the problem as a whole instead of a small section of the problem.

4. HOW THE FINCH WORKS

4.1 Selecting Good Offspring

Evolving Java Byte code only reduces the chance of evolving non-compilable bytecode to a certain extent. Even though byte code is less a lot less syntactical, than source code, it still is syntactical. Sipper et al.[4] addresses this issue by checking if a good offspring has been created before letting it join the evolved population. This is done by checking if it compiles. If it doesn't then another offspring with the same parents is made. This process is repeated until a good offspring is produced or a predetermined number of attempts have been made.

The checks that are made make sure that all the variables will be read, written, type-compatible, and will not cause stack underflow[1].

It is important that only good offspring is selected because this provides good variability in the population. Non-compilable code would result in a fitness score of zero. Since noncompilable code would occur frequently enough it would cause a large portion of the population to have a zero fitness score.

4.2 Crossover

The FINCH uses two-point crossover to evolve bytecode. It takes two programs A and B and extracts sections a and b of bytecode respectively. It then takes section a and inserts it into where section b used to be. It only selects an a that will compile after being inserted into B. Take into account that just because a can replace b in B does not imply that b can replace a in A. In other words it is not a biconditional relationship.

4.3 Non-Halting Bytecode

Another issue that arises from evolving unrestricted bytcode is that the resulting program might enter a non-halting state. These problems don't arise when the check to see if program compiles. Instead, this is a runtime issue. This, is especially true when evolving programs that contain loops and recursion. The way that Sipper et al.[5] deal with this, before running the program, is count how many calls are made to each function. If too many calls are made to a function than an exception is thrown.

4.4 Variable Access Sets

[4] much hard

5. EVOLVING ASSEMBLY CODE

Forrest et al.[6] unlike Sipper et al.[5] does not check to see if their evolved bytecode is noncomilable. Also, instead of checking if their program will enter a non-halting state during runtime, instead they terminate program after an eight second wait.

6. RESULTS

6.1 FINCH

6.1.1 *Symbolic Regression*

Symbolic regression is a method of finding a mathematical function that best fits a a set of points between two intervals.

Simple Regression.

As a simple problem, Orlov et al. [5], decided to first solve for twenty random points between -1 and 1 from the polynomial $x^4+x^3+x^2+x$. An initial population of 500 programs was used with a crossover probability of 90%. Binary tournament selection was used to determine what individuals to evolveBinary Tourney. Selection is between 2 individual?. The initial population were all copies of a zero fitness program. The program contained the function set {-,+,*,%}. Bytcode segments where chosen at random with 1000 possible retries.

The average number of retries was between 16 and 24 and an ideal individual was found 99% of the time.

complex regression.

As a more complex problem Orlov et al. decided to evolve the exact set up as for simple regression. However, the fitness algorithm was changed to reward individuals that provided a partial solution. This was done by checking "if all twenty points are within a 10^-8 of a distance of a degree-n polynomial's output"more detail. Also, the growth factor was changed from four to five.

With these changes, FINCH was able to find an optimal solution every single run.

6.1.2 *Artificial Ant*

The artificial ant problem is a problem that involves moving an ant on a square grid trying to find all the food on the grid. The and starts off on the upper right hand corner of the grid facing to the left. The terminals used to control the ant are left,right, and move. Left and right turn the and 90 degrees to its left and right respectively. The move function moves the and forward. Additional functions that can be used is IF-FOOD-AHEAD,PROGN2,and PROGN3. IF-FOOD-AHEAD is a test which checks if there is food in

front of the ant. PROGN2 and PROGN3 allow for two and tree sequence operations respectively.

6.1.3 *Intertwined Spirals*

6.1.4 *Array Sum*

6.1.5 *Tic-Tac-Toe*

6.2 x86

6.3 Future Work

7. CONCLUSION

8. ACKNOWLEDGMENTS

9. REFERENCES

[1] *Genetic Programming Theory and Practice VIII*. Springer New York, 2011.

[2] Backus-naur form. January 2014.

[3] Java bytcode. Febuary 2014.

[4] M. Orlov and M. Sipper. Genetic programming in the wild: Evolving unrestricted bytecode. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 1043–1050, New York, NY, USA, 2009. ACM.

[5] M. Orlov and M. Sipper. Flight of the finch through the java wilderness. *Evolutionary Computation, IEEE Transactions on*, 15(2):166–182, April 2011.

[6] E. Schulte, S. Forrest, and W. Weimer. Automated program repair through the evolution of assembly code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 313–316, New York, NY, USA, 2010. ACM.