Applying Genetic Programming to Bytecode and Assembly

Eric Collom

Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA

29 April '14,
UMM Senior Seminar

The big picture

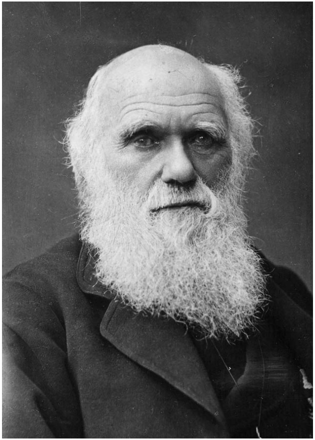
- Evolving whole programs is hard to do with source code.
- Evolving whole programs with bytecode and assembly is not as hard.

Outline

Outline

What is Evolutionary Computation?

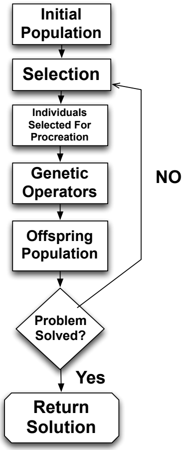
- Evolutionary Computation (EC) is a a technique that is used to automate computer problem solving.
- Loosely emulates evolutionary biology.



Charles Darwin
http://tinyurl.com/lqwj3wt

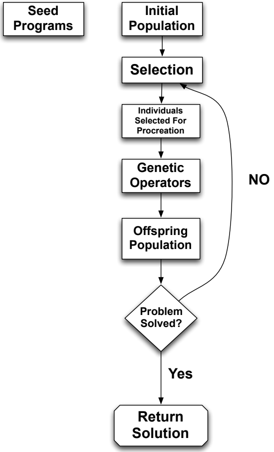
How does it work

- Continuous optimization
- Selection is driven by the *fitness* of individuals
- Genetic operators mimic sexual reproduction and mutation



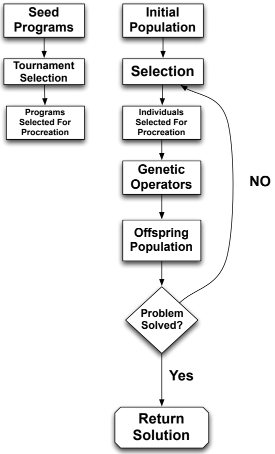
Genetic Programming

- Uses the EC process to evolve programs
- This done by using Evolutionary Algorithm (EA)
- The population consists of programs



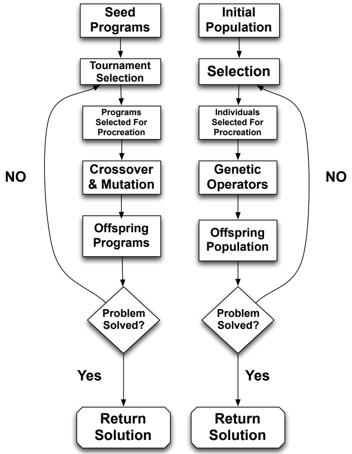
Genetic Programming

- Tournament Selection
  - Randomly select a specified number of programs
  - Pick the program with the highest fitness
  - That program then is selected for reproduction

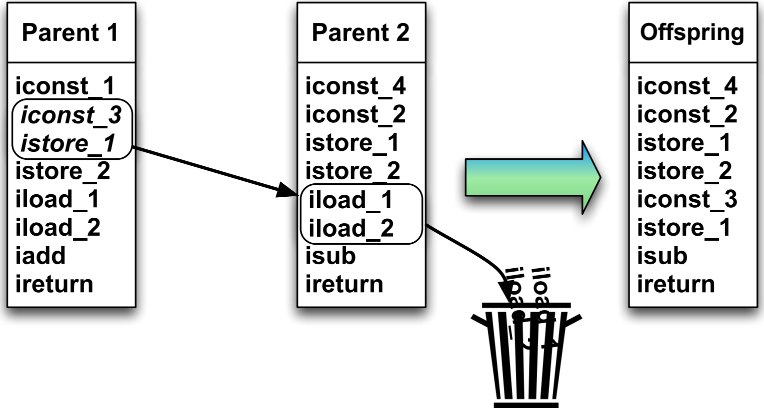


Genetic Programming

- Crossover
  - Sexual reproduction
- Mutation
  - Asexual reproduction
  - Can be used along with crossover

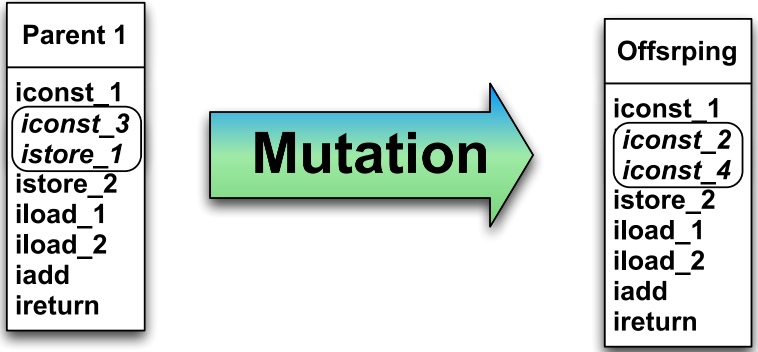


Crossover



Crossover with Java Bytecode

Mutation



Crossover with Java Bytecode

Outline

Source Code Drawbacks

- While it would be useful, it is difficult to apply evolution to an entire program in source code
  - Source code is made to simplify reading and writing programs
  - Source code does not represent the semantic constraints of the program.

Syntax vs Semantics

- Even though a sentence follows all the rules of the English language it still might be true.

  Semantically Wrong The sun rises in the West
  Semantically Correct The sun rises in the East

Syntax vs Semantics

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

(a)

```
int x = 7;  float y;
if (y >= 0) {
   y = x;
   x = y;
}
System.out.println(z);
```

(b)

Both (a) and (b) are valid syntactically. However (b) is invalid semantically.

WHAT TITLE DO I GIVE THIS SLIDE??????

- EAs are usually designed to avoid dealing with semantic constraints

```
class Robot{
...
  double robotSpeed(){
      double evolvedVariable = valueFromEA;
      return (robot.location + evolvedVariable)/2;
  }
...
}
```

Instruction-Level Code Constraints

- Consists of a smaller alphabets
- Simpler syntactically
- Less semantic constraints to violate

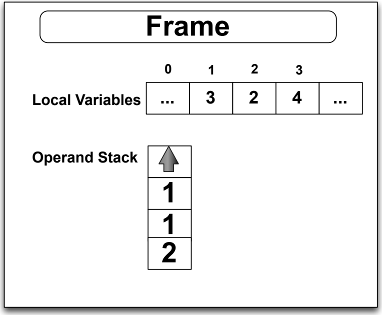
Instruction-level Code Benefits

- Do not need to understand the structure of the program being evolved
- Can evolve a lot from a little
- If there is a compiler for it we can evolve it

Outline

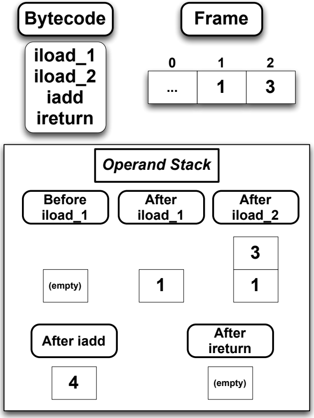
Java Virtual Machine

- Frames
- Array of local variables
- Operand Stack



Java Bytcode and Frames

- Opcodes
- Prefix indicates type



Outline

What is FINCH?

- FINCH is an EA developed by M. Orlov and M. Sipper
- It evolves Java bytecode
- It deals with semantic constraints

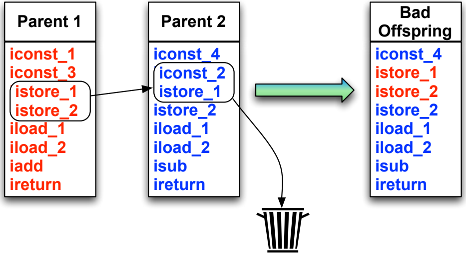
Dealing With Semantic Constraints

- There is still a chance to produce non-executable code
- Solution: check if offspring follows semantic constraints
  - Stack and Frame Depth
  - Variable Types
  - Control Flow

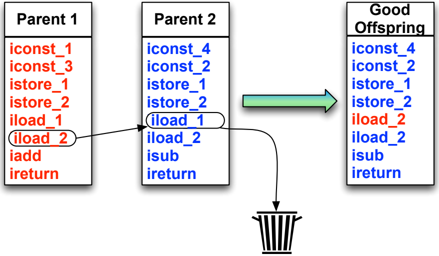
Dealing With Semantic Constraints

1. Apply crossover to two parents
2. Check if they comply to semantic constraints
3. If the program passes the constraint test then it proceeds to offspring generation
4. If it fails the constrain check then another attempt is made with the same parents

Bad Crossover



Good Crossover



Array Sum

- The array sum problem
  - Started with a zero fitness seed program
  - Counted function calls to check for a non-halting state

```
int sumlistrec(List list){
    int sum = 0;
    if(list.isEmpty())
        sum *= sumlistrec(list);
    else
        sum += list.get(0)/2 + sumlistrec(
              list.subList(1, list.size()));
    return sum;
}
```

Array Sum

```
int sumlistrec(List list) {
  int sum = 0;
  if (list.isEmpty())
    sum = sum;
  else
    sum += ((Integer)list.get(0)).intValue() +
        sumlistrec(list.subList(1,
            list.size()));
  return sum;
}
```

Outline

1. Evolutionary Computation
2. Why Evolve Instruction-Level Code
3. Java Bytecode and the JVM
4. FINCH:Evolving Java Bytecode
5. Using Instruction-level Code to Automate Bug Repair
   - How it Works
   - Results

Automating Bug Repair

- Schulte et al., designed EAs that fixes bugs in x86 assembly and Java bytecode
  - Programs at times consist of thousands of lines of code
  - Uses a weighted path due to size of programs
  - No checks if executable instruction-level code is produced

Selecting Offspring

- Each instruction is given a weight
- The weight is determined by what tests execute that instruction
- Each experiment started with one negative test case and multiple positive test cases

Instruction Weight

- Only executed by failing test: weight = 1.0
- Executed by negative test and one positive: weight = 0.1
- Not executed by negative test case: weight = 0

Bugs Fixed

- Buffer overflow
- Infinite loops

Outline

Conclusions

References