SubProject 2A - Genetic Program

Tao Zhang
CS 472

March 7, 2014

Abstract

My Genetic Program (GP) works well. I rewrite the code in Java, since I have other 3 CS classes working on Java , and I want to do some practice. Most of my GP functions are similar to the references posted on the webpage. The GP can successfully generate a full tree by entering the max depth. It could generate both non-terminals and terminals. It has the function to copy the tree from a subroot or just the whole tree. Fitness evaluation and size calculation works well. It has the ability to erace the whole individual. And finally, I have a generation class that could create a population of individuals and find the best and average fitness of the population, also the average size for the population.

1 Algorithm Description

So far, my Genetic Program has 4 classes:

- Genetic_program: The GP main class.
- Generation: The class that generate a population so far. It also has arrays to store the fitness, terminal/non-terminal size of each individual. It will perform crossover and mutation later.

```
public int pop_size;
public Individual [] pop; // array of individuals
public double [] fitness; // fitness of each individual
public int [] t_size;  //sizes for terminals of each individual
public int [] n_size;  //sizes for non_terms of each individual

public double X;    // the input value

public void Generation(int p, double x_in){
    pop_size = p;
    X = x_in;
    // initialize
    pop = new Individual[pop_size];
    fitness = new double[pop_size];
    t_size = new int[pop_size];
    n_size = new int[pop_size];
    generatePop();
}

public void generatePop(){
    Random r = new Random();
    int max;
    int i;
    for(i = 0; i < pop_size; i++){
        max = r.nextInt(5); // random max depth 0~5
        pop[i] = new Individual();
        pop[i].Individual(max);
        fitness[i] = pop[i].fitness(X);
        pop[i].calc_size();
        t_size[i] = pop[i].terms;
        n_size[i] = pop[i].non_terms;
    }
    }
```

- Individual: The tree class. It can generate an individual, delete an individual, calculate the size and fitness, copy part or whole of the tree, and print the tree. (More details in Section 2)
- Node: The class to generate a single node that stores the type, value, and links to left, right, and parent.

```
    public class Node {
        Node left;
        Node right;
        Node parent;

        // 0 = +; 1 = -; 2 = *; 3 = /; 4 = X; 5 = const
        int type;
        double const_value;

        public void Node(){
            left = null;
            right = null;
        }
    }
```

2 Individual Description

The Individual class is a tree class, and I have these functions:

2.1 Generate full random expression

The generate function will read a max depth input and generate a full tree. Each node has the link to their parent.

-
  ```
    public void generate(int max){
        Node n = new Node();
        root = n;    // point root to new start node
        full(0, max, n);
    }

    // creates full trees
    public void full(int depth, int max, Node parent){
        Random randomGenerator = new Random();
        Node p_copy = parent;
        if(depth >= max){    // 4 for X, 5 for const_value
            parent.type = 4 + randomGenerator.nextInt(2);
        }else{  // 0 = +, 1 = -, 2 = *, 3 = /
            parent.type = randomGenerator.nextInt(4);
                // generate left leaf
            Node l = new Node();
  ```

```
            parent.left = l;
            l.parent = p_copy;
            full(depth+1,max,l);
                //generate right leaf
            Node r = new Node();
            parent.right = r;
            r.parent = p_copy;
            full(depth+1,max,r);
        }

        if(parent.type == 5){ // random const_value 0~9
            parent.const_value = randomGenerator.nextInt(10);
        }
    }
```

2.2 Erace

Function that erace the whole individual.

-
  ```
    public void erase(){
        deleteNode(root);   // point to root and start delete
        root = null;
    }

    public void deleteNode(Node n){
            /* if non_terminal, keep tracing the leaves
             * until we delete terminals
             */
        if(n.type < 4){
            if(n.left != null)
                deleteNode(n.left);

            if(n.right != null)
                deleteNode(n.right);
        }
            // delete each node
        n.left = null;
        n.right = null;
        n.parent = null;
    }
  ```

2.3 Copy

The copy function will copy the whole individual to another individual. We just need to change the start position “n” and “source” of copyNode() so that we can copy a subtree of one individual to the subroot of another individual

-
  ```
    public void copy(Node source){
        Node n = new Node();
  ```

```
        root = n;
        copyNode(n,source);
    }

    public void copyNode(Node self, Node source){
        self.type = source.type;
        self.const_value = source.const_value;
        if(source.type < 4){
            if(source.left != null){
                Node l = new Node();
                self.left = l;
                copyNode(l, source.left);
            }

            if(source.right != null){
                Node r = new Node();
                self.right = r;
                copyNode(r, source.right);
            }
        }
    }
```

2.4 Fitness

Recursive calculation according to the type of each node. When the type comes division, if the divisor is 0, I defined the result as 0.

```
    public double fitness(double X){
        double z;
        z = evaluate(X, root);
        return z;
    }

    public double evaluate(double X, Node node){
        double l;
        double r;
        switch(node.type){
            case 0: // +
                l = evaluate(X, node.left);
                r = evaluate(X, node.right);
                return l+r;
            case 1: // -
                l = evaluate(X, node.left);
                r = evaluate(X, node.right);
                return l-r;
            case 2: // *
                l = evaluate(X, node.left);
                r = evaluate(X, node.right);
                return l*r;
            case 3: // /
                l = evaluate(X, node.left);
                r = evaluate(X, node.right);
                if(r == 0){
                    return 0;
                }else{
                    return l/r;
                }
            case 4: // X
                return X;
            case 5: // const_value
                return node.const_value;
            default:
                System.out.println("Error, unknown instruction");

        }
        return -1;
    }
```

2.5 Size

I keep the “terms” and “non_terms” as public int

-
  ```
  public void calc_size(){
          // initialize
      terms = 0;
      non_terms = 0;
      count(root);
  }

  public void count(Node n){
      if(n.type >= 4){
          terms++;
      }else {
          non_terms++;
          count(n.left);
          count(n.right);
      }
  }
  ```

2.6 Print

This function will print all the termimals and non terminals in math order.

-
  ```
  public void printTree(){
      if(isEmpty())
          System.out.println("The tree is emtpy");
      else{
          printNode(root);
          System.out.println(" = ?");
      }
  }
  ```

```
    public void printNode(Node n){
        if(n.left != null){
            printNode(n.left);
        }

        switch(n.type){
            case 0:
                System.out.printf(" + ");
                break;
            case 1:
                System.out.printf(" - ");
                break;
            case 2:
                System.out.printf(" * ");
                break;
            case 3:
                System.out.printf(" / ");
                break;
            case 4:
                System.out.printf("X");
                break;
            case 5:
                System.out.printf("%.2f", n.const_value);
                break;
            default:
                System.err.println("Error: unknown node type");
                break;
        }


        if(n.right != null){
            printNode(n.right);
        }
    }
```

3 Results

3.1 A single individual

Here are the results that test the generate, erace, fitness and size calculation, copy, and print.

-
```
        Individual t = new Individual();
        t.generate(3);
        t.printTree();
        int X = 1;
        double z = t.fitness(X);
        System.out.println("Fitness: y( "+ X + " ) = "+z);

        t.calc_size();
        System.out.printf("Terms: %d, Non-Terms: %d%n",t.terms, t.non_terms);

        // copy t to a new individual c
        Individual c = new Individual();
        c.copy(t.root);

        //erase individual t
        t.erase();
        t.printTree();

        c.printTree();
        X = 2;
        z = c.fitness(X);
        System.out.println("Fitness: y( "+ X + " ) = "+z);

        c.calc_size();
        System.out.printf("Terms: %d, Non-Terms: %d%n",t.terms, t.non_terms);
```

- Result:

```
X + X + 3.00 * 6.00 + 2.00 + X * X + X = ?
Fitness: y( 1 ) = 26.0
Terms: 8, Non-Terms: 7
The tree is emtpy
X + X + 3.00 * 6.00 + 2.00 + X * X + X = ?
Fitness: y( 2 ) = 38.0
Terms: 8, Non-Terms: 7
```

It seems pretty good!

3.2 Generation sample

-
  ```
    Generation g = new Generation();
    g.Generation(5, 1);
    System.out.printf("BestFit: %.4f%n", g.bestFit());
    System.out.printf("AvgFit: %.4f%n", g.avgFit());
    System.out.printf("Avg Term Size: %.2f%n", g.avgTsize());
    System.out.printf("Avg Non-term size: %.2f%n", g.avgNsize());
  ```

- Results:

```
    BestFit: 8.0000
    AvgFit: -128.9444
    Avg Term Size: 8.40
    Avg Non-term size: 7.40
```

4 Conclusion

Generally, the GP works well. However, I can still do some work to make it looks neat and easier for others to understand.