COMP 572

Project #2b – Genetic Programming

Sanjeev Shrestha

March 12, 2014

This is the second subproject of the GP project. The goal of this subproject is to finish the pieces of the GP for a symbolic regression problem.

For this subproject you will need to complete the GP including the following functionality (in addition to the functions from the previous assignment):

- Add a conditional to the function set of the expression trees.
- Mutation
- Crossover of two trees
- Selection
- Elitism if you are using a generational model

Test the GP to make sure that it is working.

Project Write-up: For this subproject you only need a description of the general algorithm: generational or steady-state, how mutation works, the selction mechanism, etc. and a description of any problems so far. Note that the write-up may be fairly short.

PROJECT DETAILS:

Algorithm	**Generational**
Population size	100
Selection method	Tournament Selection (Same as before hence not discussed)
Elitism (if used)	-
Crossover method	Sub Tree Crossover
Crossover rate	100% on the 10 offsprings selected
Mutation method	Node Mutation
Mutation rate	50% on the 10 offsprings selected
Operator/non-terminal set	• ADD(+), SUBTRACT(-), DIVIDE(/), MULTIPLY(*), • SIN($\sin\theta$), COS($\cos\theta$), • IFGT(if (a > b) then { c } else { d }), IFLT(if (a < b) then { c } else { d })
Terminal set	INPUTX(X), CONSTANT ([0,1,2,..., 20])
Fitness function	Root Mean Squared Error between generated output values and provided output values.
Size control (if any)	-
Stop Condition	Run for1000 iterations or stop if solution is found early.

MUTATION:

For the genetic programming project, I have implemented the node mutation. Node mutation is used to change the type of the Node from one to another. For example with node mutation a previously SUM node could be changed to DIVIDE node having the same arity. The mutation probability for any gene for the selected off springs is 50%. Note that the change of the node from one type to another is particularly based on arity and the tree traversal is done recursively.

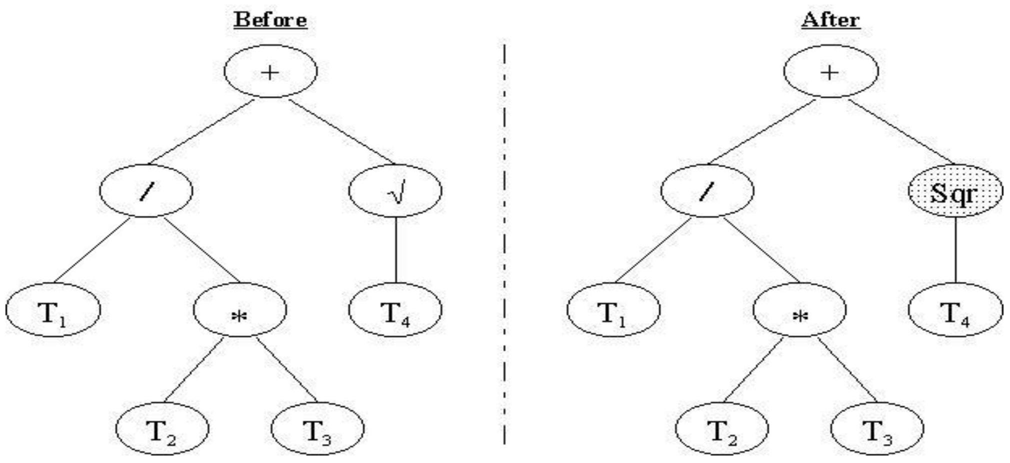

Fig: Before and After Tree States for Node Mutation [Image from http://www.stumptown.com]

Following is the pseudo-code for the mutation operation:

```
private void mutate(Node root) {
       // if the root is null no need for mutation
       if (root == null) {
           return;
       }
       // randomly generate a float between [0,1) and check if greater than 0.5
       if (rdm.nextFloat() > 0.5) {
           // if the type of node is ADD or SUBTRACT OR DIVIDE OR MULTIPLY
           if (root.getType() == Command.ADD
                  || root.getType() == Command.SUBTRACT
                  || root.getType() == Command.DIVIDE
                  || root.getType() == Command.MULTIPLY) {
              // Change Node Type
              root.setType(rdm.nextInt(Command.MULTIPLY + 1));
              // Move to the next branches
              mutate(root.getBranches()[0]);
              mutate(root.getBranches()[1]);
           }
           // if the type of node is IFGT or IFLT
           else if (root.getType() == Command.IFGT
                  || root.getType() == Command.IFLT) {
              // Change Node Type
              root.setType(rdm.nextInt(2) + 4);
              mutate(root.getBranches()[0]);
              mutate(root.getBranches()[1]);
              mutate(root.getBranches()[2]);
              mutate(root.getBranches()[3]);
           }
           // if the type of the node is SINX or COSX
           else if (root.getType() == Command.SINX
                  || root.getType() == Command.COSX) {
              // Change Node Type
              root.setType(rdm.nextInt(2) + 6);
              mutate(root.getBranches()[0]);
           }
           // if the type of the node is INPUTX or CONSTANT
           else if (root.getType() == Command.INPUTX
                  || root.getType() == Command.CONSTANT) {
              // Change Node Type
              root.setType(rdm.nextInt(2) + 8);
              // if type is constant then set corresponding value
              if (root.getType() == Command.CONSTANT) {
                  root.setConstValue(DoubleFormatter
                         .getFormattedSolution(Double.valueOf(rdm
                                .nextDouble() * 2 * Constants.CONST_LIMIT)
                                - (Constants.CONST_LIMIT / 2)));
              }
              // else remove the value for type X
              else if (root.getType() == Command.INPUTX) {
                  root.setConstValue(null);
              }
           }
       }
    }
```

Crossover:

The crossover for genetic programming implemented is Sub Tree Crossover. Implementing this was fairly tricky and it took up a lot of time. The idea behind sub tree crossover is to select a random point in each of the pair of Individuals and swap the nodes from one tree to the other tree. For sub tree mutation, the branch reference of the parent and child nodes parent reference values have to be updated accordingly.

A typical scenario and corresponding algorithm is presented in the corresponding attached sheet.

Pseudo code is as follow:

```
private Individual[] exchangeInfo(Individual firstParent,
          Individual secondParent) {

      Individual[] resultArray = new Individual[2];

      // Create a temporary node
      Node temp = new Node();
      // Get a random node from firstParent
      Node firstParentRandNode = new Node();

      firstParentRandNode = firstParent.getRandomNode();
      // Get firstRandom Node's Parent
      Node firstRandNodeParent = new Node();
      firstRandNodeParent = firstParentRandNode.getParent();
      // Get a random node from secondParent
      Node secondParentRandNode = new Node();
      secondParentRandNode = secondParent.getRandomNode();
      // Get secondRandom Node's Parent
      Node secondRandNodeParent = new Node();
      secondRandNodeParent = secondParentRandNode.getParent();

      int firstRandNodeBranchLoc = 0, secondRandNodeBranchLoc = 0;

      for (int i = 0; i < Constants.MAX_ARITY; i++) {
          if (firstRandNodeParent != null) {
            if (firstRandNodeParent.getBranches()[i] != null
                  && firstRandNodeParent.getBranches()[i]
                        .equals(firstParentRandNode)) {
                firstRandNodeBranchLoc = i;
                break;
            }
          }

      }

      for (int i = 0; i < Constants.MAX_ARITY; i++) {
          if (secondRandNodeParent != null) {
            if (secondRandNodeParent.getBranches()[i] != null
                  && secondRandNodeParent.getBranches()[i]
                        .equals(secondParentRandNode)) {
```

```
                    secondRandNodeBranchLoc = i;
                    break;
                }
            }
        }

        /*
         * System.out.println("\nStatus of Parents Before Exchange");
         * System.out.println("First Parent: " + firstParent.toString());
         * System.out.println("Second Parent: " + secondParent.toString());
         */

        // Exchange the nodes
        // temp.setParent(firstParentRandNode.getParent());
        temp = firstParentRandNode;
        // firstParentRandNode.setParent(secondParentRandNode.getParent());
        firstParentRandNode = secondParentRandNode;
        // secondParentRandNode.setParent(temp.getParent());
        secondParentRandNode = temp;

        // Adjust the corresponding branches.
        if (firstRandNodeParent != null) {
            firstRandNodeParent.getBranches()[firstRandNodeBranchLoc] =
firstParentRandNode;
        } else {
            firstParent.setTheIndv(firstParentRandNode);
        }

        if (secondRandNodeParent != null) {
            secondRandNodeParent.getBranches()[secondRandNodeBranchLoc] =
secondParentRandNode;
        } else {
            secondParent.setTheIndv(secondParentRandNode);
        }

        /*
         * System.out.println("Status of Parents After Exchange");
         * System.out.println("First Parent: " + firstParent.toString());
         * System.out.println("Second Parent: " + secondParent.toString() +
         * "\n");
         */
        firstParent.calcSize();
        secondParent.calcSize();

        firstParent.evaluate();
        secondParent.evaluate();

        resultArray[0] = firstParent;

        resultArray[1] = secondParent;

        return resultArray;
    }
```

RESULTS:

Two results generated from the GP for the f(x) = x^2 is as follows:

- Formula 1: IFLT (COS (IFGT (SIN (14.77) + X - X > 11.95) { 13.05 } else { 0.89 }) - COS (SIN (X)) < 10.26 * X * IFLT (12.51 < X) { X } else { X } + X * 10.92 + 13.27 + 13.89) { IFGT (IFGT (IFGT (X > X) { X } else { X } > COS (11.58)) { X / -4.1 } else { X - X } > COS (6.21) / 2.77 * X) { X * X * 4.26 * 10.4 } else { IFLT (COS (X) < COS (X)) { 4.9 + X } else { X * X } } } else { SIN (IFLT (X + X < SIN (X)) { X / X } else { SIN (X) }) }

- Formula 2: IFLT (IFGT (IFLT (X < -1.27) { X } else { X } > 1.79 - 14.36) { -3.65 / X } else { X / 2.11 } < COS (14.48 + X)) { IFLT (IFGT (6.48 > X) { X } else { X } < 10.79 / 0.82) { IFGT (X > 4.44) { X } else { X } } else { IFLT (X < X) { 3.44 } else { 10.33 } } } else { SIN (13.67) / SIN (X) } * IFLT (8.17 + 6.32 < X - 5.34) { 9.53 * 7.18 } else { IFGT (-1.25 > 11.85) { -3.17 } else { X } } + IFGT (X * 5.59 > IFGT (X > X) { 10.35 } else { X }) { X - X } else { -1.41 + X }

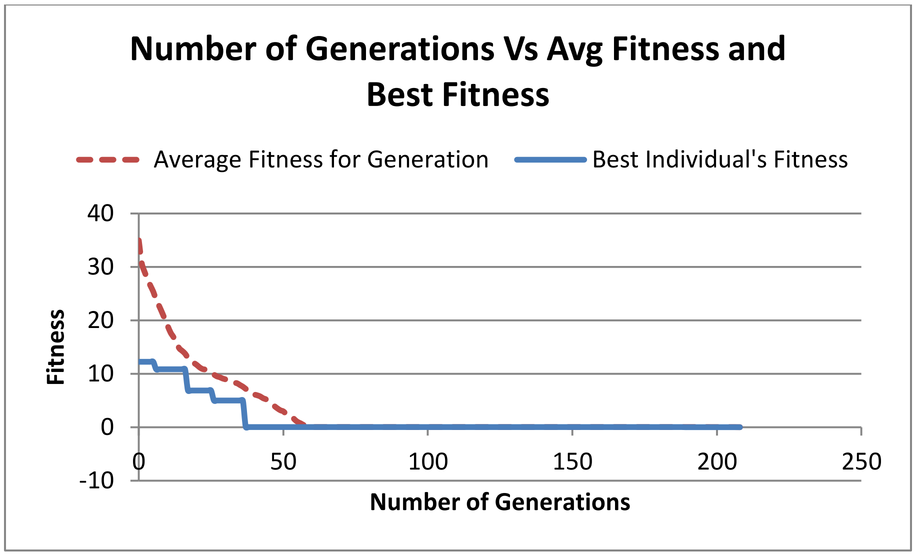

Fig: Number of Generations vs Average Fitness and Best Fitness. The Average fitness had a smooth descent towards achieving better fitness. However, the best fitness was rather step size in fashion until it reached a certain threshold.

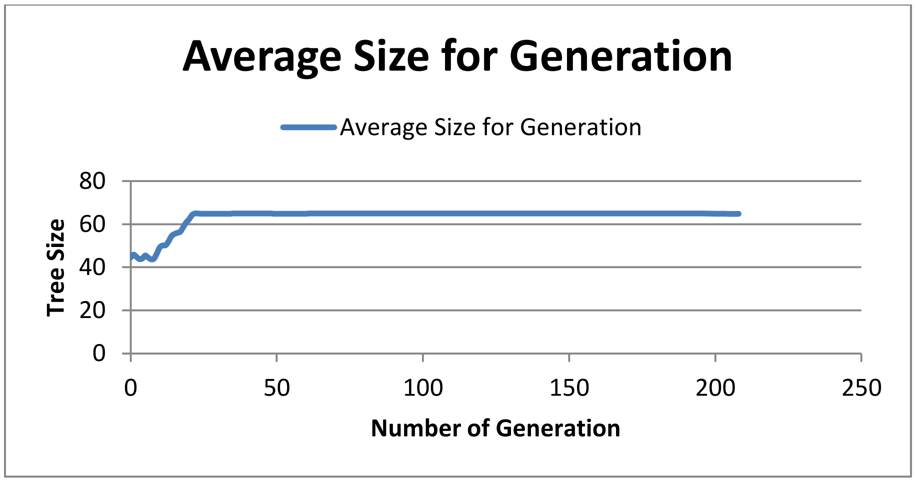

Fig: Number of Generations vs Average Size of Tree. The size of the tree grew as the generations progressed and after a while became steady indicating no growth in the overall population size.

DISCUSSION:

I kept running into the problem where I updated the nodes but the parent pointed branches were not updated. Hence after a lot of trial and error I was finally able to fix the trees. The program is successful in generating the different versions of x^2 at tree heights of T=3, 4, 5.

CONCLUSION:

The genetic program was successful in generating the output formula. The formulae seemed to overly complex however. In conclusion, I think that genetic programming works for generating a distinct formulae in case of given data points. However, I really have the same opinion that Genetic Programming is only to be used in case standard statistical methods cannot be used or standard search and optimization techniques are rendered useless.