CS 472 Fall 2011
Project 2.2

Colby Blair

Due October 17th, 2011

Abstract

In computer science, one area of study is that of optimizing functions. There are many methods for optimization, and this repor will talk about Genetic Programming (GP). Genetic Programming creates mathematical expression trees, and modifies them to make educated guesses. They are useful for finding the function defintions for curves on a graph.

This report presents a GP with mathematical non-terminal symbols '+', '-', '*', and '/', and terminal values as contants and variables. Although very simple, this report setups a proof of concept GP for later reports. It talks about the crossover and selection functions, as well as the population representation. It also shows two trees before and after crossover, a sample tree over different values, and finally, the code used.

Contents

List of Figures

Part I
Representation Description

GPs are used to try to approximate a mathamatical expression **tree** (Section 1) that describes a function on a graph. In order to improve the approximation, a random set of expression trees are generated. This set is called the **population** (Section 2). When trees are evaluated, they are measured by computing what each mathamatical expression's result is. The fitness is then the error rate, or $valueexpected - valuecomputed$. A **minimum fitness** in this report is, then, the best fitness in the population, and the max is the worst. Inverse to one's first inclination, but an abstract representation nonetheless.

1 Trees

A tree is simply class, that has pointers to child trees. Since our operators ('+', '-', '*', and '/') only take a left hand and right hand expressions, each tree only needs at most 2 children. But more or less can be inserted for future operators, on a per-operator basis. Since a tree simply points to other subtrees, the term **tree** in this report can mean either the whole tree or a subtree.

Our operators are called **non-terminals**, since they rely on the results of child subtrees to compute their results. Our **terminals** then are either constants or pointers to elements in a variable array (double, or decimal, values). Both are initialized randomly from their respective sets.

Each tree class instance points to a tree_node class. This class holds the enumerable type of the tree class; either 'plus', 'minus', 'multi', or 'div' for non-terminal trees (operators), or 'tree_double' or 'tree_variable' for terminal trees.

The terminal trees will be, in future projects, mutated using point mutation, but for now are left alone. The non-terminal trees are mutated by simple regenerating a random tree in place, and selected at random. Trees of type tree_var are, again, pointers to a variable array. This tree's value is initialized to point to a random element in the variable list. Since they are pointers, modifying variable values takes immediate affect throughout the tree. The variables in the variable array can be modified, and the tree

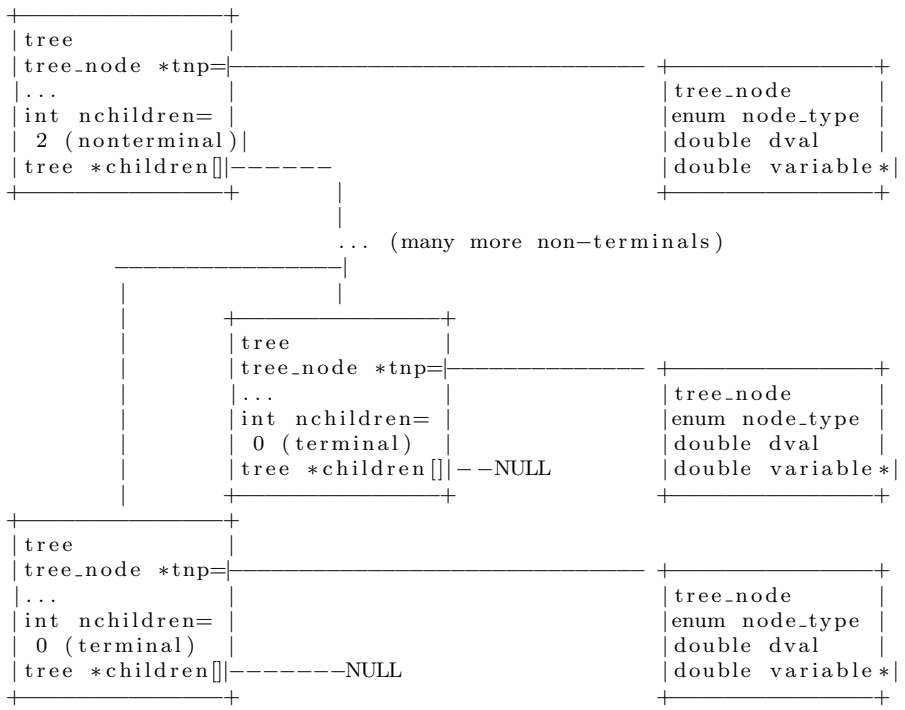


Figure 1: An expression tree (one per individual)

evaluation and fitness functions (re)ran.

2 Population

In order to optimize lots of trees to reach an approximate solution, a **population** (or set) is kept. Our population is just a list (or array) of trees.

$$P = i_1, i_2, ...i_j$$

where
i_n is a tree
$j = 500$

Figure 2: The representation of the population

Part II
Functions and Generators

3 Crossover Function

```
for original tree1:
        select random nonterminal
        replace it with random nonterminal in original tree2
for original tree2:
        select random nonterminal
        replace it with random nonterminal in original tree1
```

Figure 3: Crossover function (see Section 6.7)

The crossover function needs some pressure still on minimizing tree depth. Currently, they grow quite quickly, with less benefit to fitness than they probably should have. Crossover will probably be better too by producing a child, instead of crossing over the parents with themselves. This effectively creates two new children who are probably worse, and the parents are gone from the population.

4 Select Function

```
repeat until bored:
        pick the lowest (best) fitness
        pick the second lowest (second best) fitness
        use crossover to change both trees
```

Figure 4: Selection function (see Section 6.11)

This selection function is very simple, and not effective. It should find local minimums in fitness (best local fitnesses) faster, but will blow away

individuals with possible global minimum fitnesses (best overall) quicker. Instead, a small sample of the population should be selected, and the minimum take from them. This will be a simple improvement in future version of this report, and will look like this:

$$selection(P) = \quad i_1, i_2, ...i_k$$

where
P is the entire population
i_k is a random individual
k is the sample size, specified at run time

Figure 5: The future selection function

Note that a higher k value will find local minimum fitnesses (best fitnesses) faster, while a lower k will leave more variance in the population, because the minimum (best) fitnesses are less likely to reproduce.

5 Output

5.1 Crossover Function

```
Tree crossover test:
Tree 1 before crossover:
0:multi = -2778.13, children = 2
 1:div = 0.063765, children = 2
  2:multi = 0.0354194, children = 2
   3:plus = 0.18533, children = 2
    4:minus = 0.181647, children = 2
     5:tree_var = 0.2, children = 0
     5:tree_double = 0.0183528, children = 0
    4:multi = 0.00368309, children = 2
     5:tree_var = 0.2, children = 0
     5:tree_double = 0.0184154, children = 0
   3:multi = 0.191115, children = 2
    4:plus = 0.6, children = 2
     5:tree_var = 0.3, children = 0
```

```
      5:tree_var = 0.3, children = 0
     4:plus = 0.318525, children = 2
      5:tree_double = 0.018525, children = 0
      5:tree_var = 0.3, children = 0
  2:plus = 442.769, children = 2
   3:multi = 0.0696918, children = 2
    4:plus = 0.21865, children = 2
     5:tree_double = 0.0186502, children = 0
     5:tree_var = 0.2, children = 0
    4:plus = 0.318736, children = 2
     5:tree_var = 0.3, children = 0
     5:tree_double = 0.0187363, children = 0
   3:plus = 442.699, children = 2
    4:div = 265.972, children = 2
     5:tree_var = 0.2, children = 0
     5:tree_double = 0.0187989, children = 0
    4:div = 176.726, children = 2
     5:tree_var = 0.3, children = 0
     5:tree_double = 0.0188615, children = 0
1:plus = -43568.3, children = 2
 2:minus = -0.0801249, children = 2
  3:plus = 0.0094926, children = 2
   4:multi = 0.00378953, children = 2
    5:tree_double = 0.0189476, children = 0
    5:tree_var = 0.2, children = 0
   4:multi = 0.00570308, children = 2
    5:tree_double = 0.0190103, children = 0
    5:tree_var = 0.3, children = 0
  3:multi = 0.0896175, children = 2
   4:plus = 0.319104, children = 2
    5:tree_var = 0.3, children = 0
    5:tree_double = 0.0191042, children = 0
   4:minus = 0.280841, children = 2
    5:tree_var = 0.3, children = 0
    5:tree_double = 0.019159, children = 0
 2:multi = -43568.2, children = 2
  3:div = 30.6122, children = 2
   4:minus = -0.180771, children = 2
```

```
      5:tree_double = 0.0192294, children = 0
      5:tree_var = 0.2, children = 0
     4:minus = -0.180708, children = 2
      5:tree_double = 0.019292, children = 0
      5:tree_var = 0.2, children = 0
    3:div = -1423.23, children = 2
     4:minus = -0.180638, children = 2
      5:tree_double = 0.0193624, children = 0
      5:tree_var = 0.2, children = 0
     4:multi = 0.00388971, children = 2
      5:tree_double = 0.0194485, children = 0
      5:tree_var = 0.2, children = 0
Tree 2 before crossover:
0:multi = 27.7795, children = 2
 1:minus = 15505.6, children = 2
  2:div = -7.60521, children = 2
   3:minus = -16.6608, children = 2
    4:multi = 0.00586978, children = 2
     5:tree_double = 0.0195659, children = 0
     5:tree_var = 0.3, children = 0
    4:div = 16.6667, children = 2
     5:tree_var = 0.2, children = 0
     5:tree_var = 0.3, children = 0
   3:plus = 0.00789211, children = 2
    4:multi = 0.00393979, children = 2
     5:tree_double = 0.019699, children = 0
     5:tree_var = 0.2, children = 0
    4:multi = 0.00395232, children = 2
     5:tree_double = 0.0197616, children = 0
     5:tree_var = 0.2, children = 0
  2:multi = -15513.2, children = 2
   3:plus = 502.452, children = 2
    4:div = 251.621, children = 2
     5:tree_var = 0.2, children = 0
     5:tree_double = 0.0198712, children = 0
    4:div = 250.831, children = 2
     5:tree_var = 0.2, children = 0
     5:tree_double = 0.0199338, children = 0
```

```
  3:div = -30.875, children = 2
   4:minus = 0.179988, children = 2
    5:tree_var = 0.2, children = 0
    5:tree_double = 0.020012, children = 0
   4:minus = -0.179949, children = 2
    5:tree_double = 0.0200512, children = 0
    5:tree_var = 0.2, children = 0
1:multi = 0.00179158, children = 2
 2:plus = 0.048477, children = 2
  3:multi = 0.0484845, children = 2
   4:plus = 0.220153, children = 2
    5:tree_double = 0.0201529, children = 0
    5:tree_var = 0.2, children = 0
   4:plus = 0.220231, children = 2
    5:tree_double = 0.0202312, children = 0
    5:tree_var = 0.2, children = 0
  3:multi = -7.5204e-06, children = 2
   4:plus = 0.320302, children = 2
    5:tree_double = 0.0203016, children = 0
    5:tree_var = 0.3, children = 0
   4:minus = -2.34791e-05, children = 2
    5:tree_double = 0.0203642, children = 0
    5:tree_double = 0.0203877, children = 0
 2:minus = 0.0369573, children = 2
  3:multi = 0.050171, children = 2
   4:minus = 0.179526, children = 2
    5:tree_var = 0.2, children = 0
    5:tree_double = 0.0204738, children = 0
   4:minus = 0.279464, children = 2
    5:tree_var = 0.3, children = 0
    5:tree_double = 0.0205364, children = 0
  3:multi = 0.0132138, children = 2
   4:plus = 0.0412058, children = 2
    5:tree_double = 0.0205912, children = 0
    5:tree_double = 0.0206147, children = 0
   4:plus = 0.320677, children = 2
    5:tree_var = 0.3, children = 0
    5:tree_double = 0.0206773, children = 0
```

```
tree.cpp: tree 1 crossover on 7
 out of 31
tree.cpp: tree 2 crossover on 21
 out of 31
Tree 1 after crossover:
0:multi = -19.1127, children = 2
 1:div = 0.000438685, children = 2
  2:multi = 5.14838, children = 2
   3:plus = 0.18533, children = 2
    4:minus = 0.181647, children = 2
     5:tree_var = 0.2, children = 0
     5:tree_double = 0.0183528, children = 0
    4:multi = 0.00368309, children = 2
     5:tree_var = 0.2, children = 0
     5:tree_double = 0.0184154, children = 0
   3:multi = 27.7795, children = 2
    4:minus = 15505.6, children = 2
     5:div = -7.60521, children = 2
      6:minus = -16.6608, children = 2
       7:multi = 0.00586978, children = 2
        8:tree_double = 0.0195659, children = 0
        8:tree_var = 0.3, children = 0
       7:div = 16.6667, children = 2
        8:tree_var = 0.2, children = 0
        8:tree_var = 0.3, children = 0
      6:plus = 0.00789211, children = 2
       7:multi = 0.00393979, children = 2
        8:tree_double = 0.019699, children = 0
        8:tree_var = 0.2, children = 0
       7:multi = 0.00395232, children = 2
        8:tree_double = 0.0197616, children = 0
        8:tree_var = 0.2, children = 0
     5:multi = -15513.2, children = 2
      6:plus = 502.452, children = 2
       7:div = 251.621, children = 2
        8:tree_var = 0.2, children = 0
        8:tree_double = 0.0198712, children = 0
       7:div = 250.831, children = 2
```

```
      8:tree_var = 0.2, children = 0
      8:tree_double = 0.0199338, children = 0
    6:div = -30.875, children = 2
     7:minus = 0.179988, children = 2
      8:tree_var = 0.2, children = 0
      8:tree_double = 0.020012, children = 0
     7:minus = -0.179949, children = 2
      8:tree_double = 0.0200512, children = 0
      8:tree_var = 0.2, children = 0
  4:multi = 0.00179158, children = 2
   5:plus = 0.048477, children = 2
    6:multi = 0.0484845, children = 2
     7:plus = 0.220153, children = 2
      8:tree_double = 0.0201529, children = 0
      8:tree_var = 0.2, children = 0
     7:plus = 0.220231, children = 2
      8:tree_double = 0.0202312, children = 0
      8:tree_var = 0.2, children = 0
    6:multi = -7.5204e-06, children = 2
     7:plus = 0.320302, children = 2
      8:tree_double = 0.0203016, children = 0
      8:tree_var = 0.3, children = 0
     7:minus = -2.34791e-05, children = 2
      8:tree_double = 0.0203642, children = 0
      8:tree_double = 0.0203877, children = 0
   5:minus = 0.0369573, children = 2
    6:multi = 0.050171, children = 2
     7:minus = 0.179526, children = 2
      8:tree_var = 0.2, children = 0
      8:tree_double = 0.0204738, children = 0
     7:minus = 0.279464, children = 2
      8:tree_var = 0.3, children = 0
      8:tree_double = 0.0205364, children = 0
    6:multi = 0.0132138, children = 2
     7:plus = 0.0412058, children = 2
      8:tree_double = 0.0205912, children = 0
      8:tree_double = 0.0206147, children = 0
     7:plus = 0.320677, children = 2
```

```
        8:tree_var = 0.3, children = 0
        8:tree_double = 0.0206773, children = 0
  2:plus = 442.769, children = 2
   3:multi = 0.0696918, children = 2
    4:plus = 0.21865, children = 2
     5:tree_double = 0.0186502, children = 0
     5:tree_var = 0.2, children = 0
    4:plus = 0.318736, children = 2
     5:tree_var = 0.3, children = 0
     5:tree_double = 0.0187363, children = 0
   3:plus = 442.699, children = 2
    4:div = 265.972, children = 2
     5:tree_var = 0.2, children = 0
     5:tree_double = 0.0187989, children = 0
    4:div = 176.726, children = 2
     5:tree_var = 0.3, children = 0
     5:tree_double = 0.0188615, children = 0
1:plus = -43568.3, children = 2
  2:minus = -0.0801249, children = 2
   3:plus = 0.0094926, children = 2
    4:multi = 0.00378953, children = 2
     5:tree_double = 0.0189476, children = 0
     5:tree_var = 0.2, children = 0
    4:multi = 0.00570308, children = 2
     5:tree_double = 0.0190103, children = 0
     5:tree_var = 0.3, children = 0
   3:multi = 0.0896175, children = 2
    4:plus = 0.319104, children = 2
     5:tree_var = 0.3, children = 0
     5:tree_double = 0.0191042, children = 0
    4:minus = 0.280841, children = 2
     5:tree_var = 0.3, children = 0
     5:tree_double = 0.019159, children = 0
  2:multi = -43568.2, children = 2
   3:div = 30.6122, children = 2
    4:minus = -0.180771, children = 2
     5:tree_double = 0.0192294, children = 0
     5:tree_var = 0.2, children = 0
```

```
    4:minus = −0.180708, children = 2
     5:tree_double = 0.019292, children = 0
     5:tree_var = 0.2, children = 0
   3:div = −1423.23, children = 2
    4:minus = −0.180638, children = 2
     5:tree_double = 0.0193624, children = 0
     5:tree_var = 0.2, children = 0
    4:multi = 0.00388971, children = 2
     5:tree_double = 0.0194485, children = 0
     5:tree_var = 0.2, children = 0
Tree 2 after crossover:
0:multi = 2500.81, children = 2
 1:minus = 1.39587e+06, children = 2
  2:div = −7.60521, children = 2
   3:minus = −16.6608, children = 2
    4:multi = 0.00586978, children = 2
     5:tree_double = 0.0195659, children = 0
     5:tree_var = 0.3, children = 0
    4:div = 16.6667, children = 2
     5:tree_var = 0.2, children = 0
     5:tree_var = 0.3, children = 0
   3:plus = 0.00789211, children = 2
    4:multi = 0.00393979, children = 2
     5:tree_double = 0.019699, children = 0
     5:tree_var = 0.2, children = 0
    4:multi = 0.00395232, children = 2
     5:tree_double = 0.0197616, children = 0
     5:tree_var = 0.2, children = 0
  2:multi = −1.39588e+06, children = 2
   3:plus = 502.452, children = 2
    4:div = 251.621, children = 2
     5:tree_var = 0.2, children = 0
     5:tree_double = 0.0198712, children = 0
    4:div = 250.831, children = 2
     5:tree_var = 0.2, children = 0
     5:tree_double = 0.0199338, children = 0
   3:multi = −2778.13, children = 2
    4:div = 0.063765, children = 2
```

```
 5:multi = 0.0354194, children = 2
  6:plus = 0.18533, children = 2
   7:minus = 0.181647, children = 2
    8:tree_var = 0.2, children = 0
    8:tree_double = 0.0183528, children = 0
   7:multi = 0.00368309, children = 2
    8:tree_var = 0.2, children = 0
    8:tree_double = 0.0184154, children = 0
  6:multi = 0.191115, children = 2
   7:plus = 0.6, children = 2
    8:tree_var = 0.3, children = 0
    8:tree_var = 0.3, children = 0
   7:plus = 0.318525, children = 2
    8:tree_double = 0.018525, children = 0
    8:tree_var = 0.3, children = 0
 5:plus = 442.769, children = 2
  6:multi = 0.0696918, children = 2
   7:plus = 0.21865, children = 2
    8:tree_double = 0.0186502, children = 0
    8:tree_var = 0.2, children = 0
   7:plus = 0.318736, children = 2
    8:tree_var = 0.3, children = 0
    8:tree_double = 0.0187363, children = 0
  6:plus = 442.699, children = 2
   7:div = 265.972, children = 2
    8:tree_var = 0.2, children = 0
    8:tree_double = 0.0187989, children = 0
   7:div = 176.726, children = 2
    8:tree_var = 0.3, children = 0
    8:tree_double = 0.0188615, children = 0
4:plus = -43568.3, children = 2
 5:minus = -0.0801249, children = 2
  6:plus = 0.0094926, children = 2
   7:multi = 0.00378953, children = 2
    8:tree_double = 0.0189476, children = 0
    8:tree_var = 0.2, children = 0
   7:multi = 0.00570308, children = 2
    8:tree_double = 0.0190103, children = 0
```

```
        8:tree_var = 0.3, children = 0
      6:multi = 0.0896175, children = 2
       7:plus = 0.319104, children = 2
        8:tree_var = 0.3, children = 0
        8:tree_double = 0.0191042, children = 0
       7:minus = 0.280841, children = 2
        8:tree_var = 0.3, children = 0
        8:tree_double = 0.019159, children = 0
     5:multi = -43568.2, children = 2
      6:div = 30.6122, children = 2
       7:minus = -0.180771, children = 2
        8:tree_double = 0.0192294, children = 0
        8:tree_var = 0.2, children = 0
       7:minus = -0.180708, children = 2
        8:tree_double = 0.019292, children = 0
        8:tree_var = 0.2, children = 0
      6:div = -1423.23, children = 2
       7:minus = -0.180638, children = 2
        8:tree_double = 0.0193624, children = 0
        8:tree_var = 0.2, children = 0
       7:multi = 0.00388971, children = 2
        8:tree_double = 0.0194485, children = 0
        8:tree_var = 0.2, children = 0
1:multi = 0.00179158, children = 2
 2:plus = 0.048477, children = 2
  3:multi = 0.0484845, children = 2
   4:plus = 0.220153, children = 2
    5:tree_double = 0.0201529, children = 0
    5:tree_var = 0.2, children = 0
   4:plus = 0.220231, children = 2
    5:tree_double = 0.0202312, children = 0
    5:tree_var = 0.2, children = 0
  3:multi = -7.5204e-06, children = 2
   4:plus = 0.320302, children = 2
    5:tree_double = 0.0203016, children = 0
    5:tree_var = 0.3, children = 0
   4:minus = -2.34791e-05, children = 2
    5:tree_double = 0.0203642, children = 0
```

```
    5:tree_double = 0.0203877, children = 0
  2:minus = 0.0369573, children = 2
   3:multi = 0.050171, children = 2
    4:minus = 0.179526, children = 2
     5:tree_var = 0.2, children = 0
     5:tree_double = 0.0204738, children = 0
    4:minus = 0.279464, children = 2
     5:tree_var = 0.3, children = 0
     5:tree_double = 0.0205364, children = 0
   3:multi = 0.0132138, children = 2
    4:plus = 0.0412058, children = 2
     5:tree_double = 0.0205912, children = 0
     5:tree_double = 0.0206147, children = 0
    4:plus = 0.320677, children = 2
     5:tree_var = 0.3, children = 0
     5:tree_double = 0.0206773, children = 0
```

5.2 Selection Function

The selection function didn't preform very well yet, due to some needed improvement in crossover. The source can be seen in the **tree_gp::ss()** function in Section 6.11. Its output looks as follows:

```
Min fitness is 331 element. Its eval value is 7.54065
Second min fitness is 356 element. Its eval value is 7.68431
...
Min fitness is 430 element. Its eval value is 6.72136
Second min fitness is 97 element. Its eval value is 6.27473
...
Min fitness is 269 element. Its eval value is 6.28595
Second min fitness is 475 element. Its eval value is 7.53898
```

6 Code

6.1 Makefile

```
PROC=eval
CPP=g++
CPPFLAGS=-g -pg -Wno-write-strings
#CPPFLAGS=-g -pg -Wno-write-strings -DDEBUG=1
#CPPFLAGS=-g -pg -Wno-write-strings -DDEBUG_TREE=1
OBJS=tree_gp.o darray.o tree_node.o tree.o main.o test.o

all: $(OBJS)
	$(CPP) $(CPPFLAGS) $(OBJS) -o $(PROC)

main.o: main.cpp
	$(CPP) $(CPPFLAGS) main.cpp -c

tree_node.o: tree_node.cpp tree_node.h
	$(CPP) $(CPPFLAGS) tree_node.cpp -c

tree.o: tree.cpp tree.h
	$(CPP) $(CPPFLAGS) tree.cpp -c

test.o: test.cpp test.h
	$(CPP) $(CPPFLAGS) test.cpp -c

darray.o: darray.cpp darray.h
	$(CPP) $(CPPFLAGS) darray.cpp -c

tree_gp.o: tree_gp.cpp tree_gp.h
	$(CPP) $(CPPFLAGS) tree_gp.cpp -c

clean:
	rm $(PROC) *.o gmon.out
```

6.2 main.h

```
#ifndef _MAIN_H
#define _MAIN_H

#ifdef DEBUG
#define DEBUGMSG(arg) (cout << arg << endl)
#else
#define DEBUGMSG(arg) ;
#endif
```

```
#endif
```

6.3 main.cpp

```
#include <iostream>
#include "darray.h"
#include "tree_gp.h"

#ifdef DEBUG
#include "test.h"
#endif

using namespace std;

int main()
{
        #ifdef DEBUG
        test_nodes();
        test_darray();
        test_trees();
        test_tree_copy();
        test_tree_replace();
        test_tree_crossover();
        exit(1);
        #endif

        //Main eval
        darray *dp1 = new darray(2, false);
        dp1->a[0] = .2;
        dp1->a[1] = .3;
        tree_gp *tgp1 = new tree_gp(500, 5, &dp1);
        //x^3 + 5y^3 - 4xy + 7
        //= (.2)^3 + 5(.3)^3 - 4(.2)(.3) + 7
        //= .008 + .135 - .24 + 7
        //= 6.903
        double dexpected = 6.903;

        for(int i = 0; i < 500; i++)
        {
                tgp1->ss(dexpected);
                tgp1->print_fitnesses(dexpected);
                int mini = tgp1->get_lowest_fitness_index(dexpected);
                cout << "Min fitness is " << mini << " element. "
                 <<  "Its eval value is " << tgp1->get_eval(mini) << endl;
                int mini2 = tgp1->get_second_lowest_fitness_index(dexpected);
```

```
                cout << "Second min fitness is " << mini2 << " element. "
                 << "Its eval value is " << tgp1->get_eval(mini2) << endl;
        }

        return(0);
}
```

6.4 tree_node.h

```
#ifndef _TREE_NODE_H
#define _TREE_NODE_H

#include <iostream>

#include "darray.h"

using namespace std; //for string

//how many types? see tree_node::node_type
// used in tree::gen_rand_node()
#define NTYPES 4

//how many terminal types? see tree_node::node_type
// used in tree_gen_rand_term_tree_node()
#define NTERMTYPES 2

class tree_node
{
//public enum here so private members can see
public:
        enum node_type
        {
                plus,
                minus,
                multi,
                div,
                tree_double,    //terminal
                tree_var,       //terminal
                null
        };

private:
        node_type ntype; //type of node (see node_type)
        double dval; //for tree_double types only
        int dpi;     //index the ddp points to in dp
        darray *dp; //darray pointer
```

```
        double *ddp; //double pointer to rand element in this->dp

public:

        tree_node(tree_node::node_type, double, darray**);
        bool copy(tree_node**);
        double get_dval();
        double get_ddp_val();
        tree_node::node_type get_ntype();

        bool set_ddp(int);

        bool print_ntype();
        bool print_dval();
        bool print_ddp();
        bool print_members();
};

#endif
```

6.5 tree_node.cpp

```
#include <iostream>
#include <stdarg.h>
#include <typeinfo>
#include <cstdlib>

#include "tree_node.h"
#include "tree.h"
#include "main.h"

using namespace std;

tree_node::tree_node(tree_node::node_type val, double dval, darray **dp)
{
        DEBUGMSG("DEBUG: tree_node.cpp: Setting node type");

        //init members to default vals
        this->dval = 0;
        this->dp = NULL;
        this->ddp = NULL;

        switch (val)
        {
                case tree_node::plus:
                {
```

```
        this->ntype = val;
        DEBUGMSG(" Node type == plus");
        break;
}
case tree_node::minus:
{
        this->ntype = val;
        DEBUGMSG(" Node type == minus");
        break;
}
case tree_node::multi:
{
        this->ntype = val;
        DEBUGMSG(" Node type == multi");
        break;
}
case tree_node::div:
{
        this->ntype = val;
        DEBUGMSG(" Node type == div");
        break;
}
case tree_node::tree_double:
{
        this->ntype = val;
        //get the float val
        this->dval = dval;
        DEBUGMSG(" Node type == tree_double");
        DEBUGMSG(" Node val == " << this->dval);
        break;
}
case tree_node::tree_var:
{
        DEBUGMSG(" Node type == tree_var");

        this->ntype = val;
        //get the float val
        //get darray pointer from va_args
        //TODO: pass dp by reference instead

        /* initialize random seed: */
        srand ( clock() );

        //set dp to point to reference of passed in dp
        this->dp = (*dp);
```

```
                        //set dpi
                        /* generate secret number: */
                        // select random element in dp
                        this->dpi = rand() % this->dp->get_size();

                        //set ddp to point to a random element of dp->a
                        this->ddp = &this->dp->a[this->dpi];
                        DEBUGMSG(" Node val from rand index " << j << "== " << *this
                        break;
                }
                default:
                        cerr << "ERROR: Node type not set, got val " \
                                << val << endl;
                        exit(1);
        }
}


bool tree_node::copy(tree_node** to)
{
        switch (this->ntype)
        {
                case tree_node::tree_double:
                {
                        (*to) = new tree_node(this->ntype, this->dval, NULL);
                        return(true);
                }
                case tree_node::tree_var:
                {
                        //TODO: not sure how stable this is exactly
                        (*to) = new tree_node(this->ntype, 0.0, &this->dp);
                        //TODO: set ddp to dp index
                        (*to)->set_ddp(this->dpi);
                        return(true);
                }
                default:
                {
                        (*to) = new tree_node(this->ntype, 0.0, NULL);
                        return(true);
                }
        }
}

double tree_node::get_dval()
```

```
{
        if(this == NULL)
        {
                return(NULL);
        }

        return(this->dval);
}


double tree_node::get_ddp_val()
{
        if(this == NULL)
        {
                return(NULL);
        }

        return(*this->ddp);
}


tree_node::node_type tree_node::get_ntype()
{
        if(this == NULL)
        {
                return(tree_node::null);
        }

        return(this->ntype);
}


bool tree_node::set_ddp(int i)
{
        if(i >= this->dp->get_size())
        {
                return(false);
        }

        this->dpi = i;
        this->ddp = &this->dp->a[i];

        return(true);
}
```

```
bool tree_node::print_ntype()
{
        if(this == NULL)
        {
                cout << "(!null!)";
        }
        else
        {
                switch (this->ntype)
                {
                        case tree_node::plus:
                        {
                                cout << "plus";
                                break;
                        }
                        case tree_node::minus:
                        {
                                cout << "minus";
                                break;
                        }
                        case tree_node::multi:
                        {
                                cout << "multi";
                                break;
                        }
                        case tree_node::div:
                        {
                                cout << "div";
                                break;
                        }
                        case tree_node::tree_double:
                        {
                                cout << "tree_double";
                                break;
                        }
                        case tree_node::tree_var:
                        {
                                cout << "tree_var";
                                break;
                        }
                } //end switch
        }
}
```

```
bool tree_node::print_dval()
{
        cout << this->dval;
        return(true);
}


bool tree_node::print_ddp()
{
        if(this->ddp == NULL)
        {
                cout << "   : ";
        }
        else
        {
                cout << *this->ddp << " : ";
        }

        return(true);
}

bool tree_node::print_members()
{
        this->print_ntype();
        cout << " : ";
        this->print_dval();
        cout << " : ";
        this->print_ddp();
        cout << "\n";
}
```

6.6 tree.h

```
#ifndef _TREE_H
#define _TREE_H

#include <time.h>

#include "tree_node.h"
#include "darray.h"


#ifdef DEBUG_TREE
#define DEBUG_TREE_MSG(arg) (cout << arg << endl)
#else
```

```
#define DEBUG_TREE_MSG(arg) ;
#endif


extern int SUM_TEMP;


#define MAX_CHILDREN 2

class tree
{
private:

public:
        //members
        tree_node *tnp;
        darray *dp; //darray pointer, for tree_double use only
        int nchildren;
        int depth;  //how deep the current tree is
        tree *children[MAX_CHILDREN];

        //methods
        tree(int, darray*);
        ~tree();
        bool copy(tree**);
        tree_node *gen_rand_nonterm_tree_node(darray*);//[non]terminal vals
        tree_node *gen_rand_term_tree_node(darray*); //terminal vals

        double eval();
        double fitness(double);

        bool is_term();
        bool is_nonterm();

        int count_terms();
        int count_nonterms();

        bool crossover(tree**,tree**);

        bool print(int);
        bool print_tnp_ntype();
};

//External tree stuff
bool mutate_nth_nonterm(tree**,int, int, int, darray*);
```

```
bool tree_replace_nth_nonterm(tree**,tree**, int);
bool tree_crossover(tree**,tree**);

#endif
```

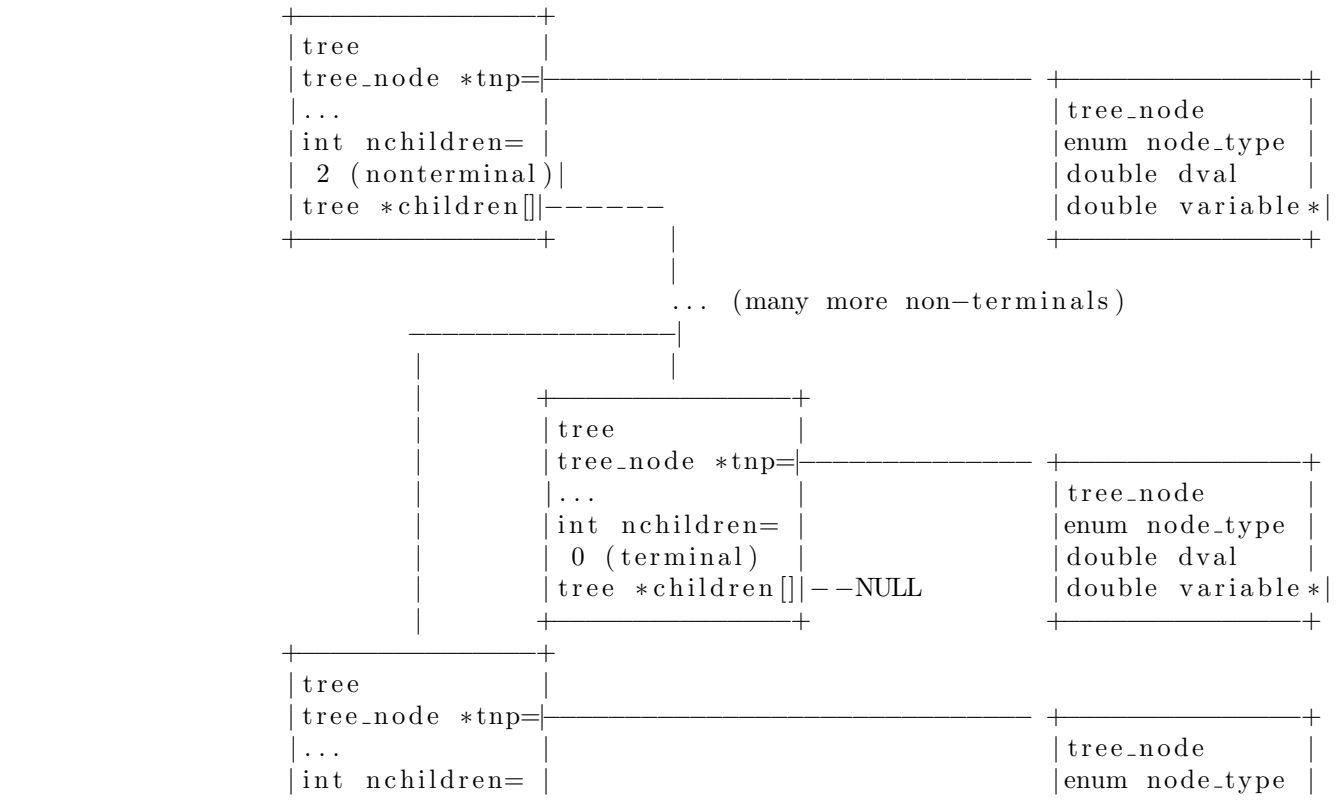
6.7 tree.cpp

```
#include <time.h>
#include <iomanip>
#include <cmath>
#include <cstdlib>

#include "tree.h"
#include "tree_node.h"
#include "main.h"

int SUM_TEMP;

/*
This is the structure I am trying to represent
```



```
        | 0 (terminal)  |                                  |double dval    |
        |tree *children[]|-------NULL                      |double variable*|
        +----------------+                                 +---------------+
*/

////////////////////////////////////////////////////////////////////////////
//Tree
////////////////////////////////////////////////////////////////////////////

tree::tree(int depth, darray *dp)
{
        this->dp = dp;
        this->depth = depth;

        //init null children
        this->nchildren = 0;
        for(int i = 0; i < MAX_CHILDREN; i++)
        {
                this->children[i] = NULL;
        }

        //Terminal
        // if we've reached the bottom, or a random fraction of total nodes
        // be a terminal
        /* initialize random seed: */
        srand ( clock() );
        /* generate secret number: */
        int rand_val = rand() % 10; //0-9 values
        //cout << "DEBUG: tree.cpp: rand_val = " << rand_val << endl;
        // 1 out of 10 rand nodes get set to terminal
        bool rand_term = (rand_val == 0);
        if(depth <= 0 ) //|| rand_term == true)
        {
                this->gen_rand_term_tree_node(dp);
                return;
        }

        //Nonterminal
        this->gen_rand_nonterm_tree_node(dp);

        //create the children
        this->nchildren = MAX_CHILDREN;
        for(int i = 0; i < MAX_CHILDREN; i++)
        {
                DEBUGMSG("DEBUG: tree.cpp: Gen child " << i  << "at depth " << depth
```

```
                this->children[i] = new tree(depth - 1, dp);
        }
}


tree::~tree()
{
        for(int i = 0; i < this->nchildren; i++)
        {
                delete this->children[i];
        }

        delete this->tnp;
}


bool tree::copy(tree** to)
{
        //DEBUG_TREE_MSG("DEBUG: tree.cpp:");

        //init the 'to' tree with 'this's depth
        //(*to) = new tree(this->depth, this->dp);
        (*to) = (tree*)malloc(sizeof(class tree));
        (*to)->depth = this->depth;


        //copy tnp
        this->tnp->copy(&(*to)->tnp);

        //copy dp
        this->dp->copy(&(*to)->dp);

        //copy nchildren
        (*to)->nchildren = this->nchildren;

        //copy children
        for(int i = 0; i < this->nchildren; i++)
        {
                this->children[i]->copy(&(*to)->children[i]);
        }
        return(false);
}

tree_node *tree::gen_rand_nonterm_tree_node(darray *dp)
{
```

```
        /* initialize random seed: */
        srand ( clock() );

        /* generate secret number: */
        int type = rand() % NTYPES;

        DEBUGMSG("DEBUG: tree.cpp: Generating rand node with type " << type);
        switch (type)
        {
                case 0:
                {
                        this->tnp = new tree_node(tree_node::plus, 0.0, NULL);
                        break;
                }
                case 1:
                {
                        this->tnp = new tree_node(tree_node::minus, 0.0, NULL);
                        break;
                }
                case 2:
                {
                        this->tnp = new tree_node(tree_node::multi, 0.0, NULL);
                        break;
                }
                case 3:
                {
                        this->tnp = new tree_node(tree_node::div, 0.0, NULL);
                        break;
                }
                default:
                        cout << "DEBUG: tree.cpp: No type for node, got type" << typ
                        exit(1);
                }
}


tree_node *tree::gen_rand_term_tree_node(darray *dp)
{
        /* initialize random seed: */
        srand ( clock() );

        /* generate secret number: */
        int type = rand() % NTERMTYPES;

        DEBUGMSG("DEBUG: tree.cpp: Generating rand term node with type " << type);
```

```
        switch (type)
        {
                case 0:
                {
                        /* initialize random seed: */
                        srand ( clock() );

                        /* generate random double: */
                        double d = ((double)rand()/(double)RAND_MAX);

                        this->tnp = new tree_node(tree_node::tree_double, d, NULL);
                        break;
                }
                case 1:
                {
                        this->tnp = new tree_node(tree_node::tree_var, 0.0, &dp);
                        break;
                }
                default:
                        cout << "DEBUG: tree.cpp: No term type for node, got type "
                        exit(1);
                }
}


double tree::eval()
{
        switch(this->tnp->get_ntype())
        {
                //nonterminals
                case tree_node::plus:
                {
                        double sum = 0;
                        for(int i = 0; i < this->nchildren; i++)
                        {
                                sum += this->children[i]->eval();
                        }
                        return(sum);
                }
                case tree_node::minus:
                {
                        double sum = this->children[0]->eval();
                        for(int i = 1; i < this->nchildren; i++)
                        {
                                sum -= this->children[i]->eval();
```

```
                }
                return(sum);
        }
        case tree_node::multi:
        {
                double prod = 1;
                for(int i = 0; i < this->nchildren; i++)
                {
                        prod *= this->children[i]->eval();
                }
                return(prod);
        }
        case tree_node::div:
        {
                double quot = 1;
                for(int i = 0; i < this->nchildren; i++)
                {
                        //divide by zero safety
                        if(this->children[i]->eval() == 0)
                        {
                                quot = 0;
                        }
                        else
                        {
                                quot /= this->children[i]->eval();
                        }
                }
                return(quot);
        }

        //terminals
        case tree_node::tree_double:
        {
                return(this->tnp->get_dval());
        }
        case tree_node::tree_var:
        {
                return(this->tnp->get_ddp_val());
        }
        default:
        {
                cerr << "ERROR: No type for eval()\n";
                exit(1);
        }
}
```

```
}


//set / change values in dp, and then run
double tree::fitness(double dexpected)
{
        return(abs(this->eval() - dexpected));
}


bool tree::is_term()
{
        if(this->nchildren <= 0)
        {
                return(true);
        }

        return(false);
}


bool tree::is_nonterm()
{
        if(this->nchildren <= 0)
        {
                return(false);
        }

        return(true);
}


int tree::count_terms()
{
        if(this->is_term() == true)
        {
                return(1);
        }

        int sum = 0;
        for(int i = 0; i < this->nchildren; i++)
        {
                sum += this->children[i]->count_terms();
        }
```

```
        return(sum);
}


int tree::count_nonterms()
{
        int sum = 0;

        if(this->is_nonterm() == true)
        {
                sum = 1;
        }

        for(int i = 0; i < this->nchildren; i++)
        {
                sum += this->children[i]->count_nonterms();
        }

        return(sum);
}


bool tree::print(int depth)
{
        if(this == NULL)
        {
                //false if I am a child that didn't get a value
                return(false);
        }

        cout << string(depth, ' ') << depth << ":";
        this->tnp->print_ntype();
        cout << " = " << this->eval();
        //more debugging stuff
        //cout << ", term:nonterm == " << this->is_term() << ":" << this->is_nonterm
        //cout << " nterm:nnonterm == " << this->count_terms() << ":" << this->count
        cout << ", children = " << this->nchildren;
        cout << endl;

        for(int i = 0; i < this->nchildren; i++)
        {
                this->children[i]->print(depth + 1);
        }

        return(true);
```

```
}


bool tree::print_tnp_ntype()
{
        if(this == NULL)
        {
                return(false);
        }
        return(this->tnp->print_ntype());
}


////////////////////////////////////////////////////////////////////////////
//External tree functions
////////////////////////////////////////////////////////////////////////////

bool mutate_nth_nonterm(tree **tp, int n, int depth, int new_depth, darray *dp)
{
        if( (*tp) == NULL)
        {
                return(false);
        }

        if( (*tp)->is_nonterm())
        {
                SUM_TEMP++;
        }
        cout << string(depth, ' ') << depth << ":";
        (*tp)->print_tnp_ntype();
        cout << " = " << SUM_TEMP;

        if(n == SUM_TEMP && (*tp)->is_nonterm())
        {
                cout << " !mutating!";

                //set this tree node to a new rand tree until it is a
                // nonterminal
                do
                {
                        delete (*tp);
                        (*tp) = new tree(new_depth, dp);
                } while ((*tp)->is_nonterm() != true);

                cout << endl;
```

```
                return(true);
        }

        cout << endl;

        //if we've already see the node to mutate

        if(n < SUM_TEMP)
        {
                return(true);
        }

        for(int i = 0; i < (*tp)->nchildren; i++)
        {
                mutate_nth_nonterm(&(*tp)->children[i], n, depth + 1, new_depth,
                                                                dp);
        }

        return(true);
}


bool tree_replace_nth_nonterm(tree **tp, tree **with, int n)
{
        if((*tp) == NULL)
        {
                return(false);
        }

        if((*tp)->is_nonterm())
        {
                SUM_TEMP++;

                if(n == SUM_TEMP)
                {
                        //copy
                        delete (*tp);
                        (*with)->copy(&(*tp));
                        return(true);
                }
                else
                {
                        for(int i = 0; i < (*tp)->nchildren; i++)
                        {
                                bool status = \
```

```
                                        tree_replace_nth_nonterm(
                                                        &(*tp)->children[i],
                                                        &(*with),
                                                        n
                                                        );
                                        if(status == true) {return(true);}
                                }
                        }
                }

        return(false);
}


bool tree_crossover(tree **tp1, tree **tp2)
{
        //make temp; tp1 original to crossover with tp2
        tree *tp1_temp;
        (*tp1)->copy(&tp1_temp);


        //crossover - tp1
        //int rand_val;
        // create random nonterm index
        /* initialize random seed: */
        srand ( clock() );
        /* generate secret number: */
        int rand_val = rand() % (*tp1)->count_nonterms(); //0-n values
        // replace
        DEBUG_TREE_MSG( "tree.cpp: tree 1 crossover on " << rand_val);
        #ifdef DEBUG_TREE
        cout << " out of " << (*tp1)->count_nonterms() << endl;
        #endif
        tree_replace_nth_nonterm(&(*tp1), &(*tp2), rand_val);

        //crossover - tp2
        // create random nonterm index
        /* generate secret number: */
        rand_val = rand() % (*tp2)->count_nonterms(); //0-n values
        DEBUG_TREE_MSG( "tree.cpp: tree 2 crossover on " << rand_val);
        #ifdef DEBUG_TREE
        cout << " out of " << (*tp2)->count_nonterms() << endl;
        #endif
        // replace
        tree_replace_nth_nonterm(&(*tp2), &tp1_temp, rand_val);
```

```
}
```

6.8 darray.h

```
#ifndef _DARRAY_H
#define _DARRAY_H


#define MAX_BUF 200
class darray
{
private:
        int size;

public:
        double a[MAX_BUF];

        darray(int, bool);
        bool copy(darray**);

        //getters
        double get_val(int);
        int get_size();

        //debug
        bool print_vals();
};

#endif
```

6.9 darray.cpp

```
#include <stdlib.h>
#include <iostream>
#include <time.h>

#include "darray.h"

using namespace std;

darray::darray(int size, bool rand_gen)
{
        this->size = size;

        //init with nulls
```

```
        for(int i = 0; i < MAX_BUF; i++)
        {
                this->a[i] = NULL;
        }

        if(rand_gen == true)
        {
                //re-init with rand vals
                for(int i = 0; i < this->size; i++)
                {
                        /* initialize random seed: */
                        srand ( clock() );

                        /* generate secret number: */
                        this->a[i] = ((double)rand()/(double)RAND_MAX);
                }
        }
        //else, need to set manually, ie darray->a[0..n] = 1,2,...
}


bool darray::copy(darray **to)
{
        //init 'to' with our size
        (*to) = new darray(this->size, false);
        //TODO: return false if new failed

        //copy over all the elements in this->a
        for(int i = 0; i < this->size; i++)
        {
                (*to)->a[i] = this->a[i];
        }

        return(true);
}

int darray::get_size()
{
        return(this->size);
}


double darray::get_val(int i)
{
        if(i >= this->size)
```

```
        {
                return(NULL);
        }

        return(this->a[i]);
}


bool darray::print_vals()
{
        for(int i = 0; i < this->size; i++)
        {
                cout << this->a[i];
                //I dunno, 5 vals per line sounds good
                if( (i % 5) == 0 && i != 0)
                {
                        cout << endl;
                }
                else //a delim
                {
                        cout << " : ";
                }
        }

        cout << endl;

        return(true);
}
```

6.10 tree_gp.h

```
#ifndef _TREE_GP_H
#define _TREE_GP_H

#include "tree.h"
#include "darray.h"

#define MAX_TREE_BUF 500

//collects other tree classes and instances for gp ops
class tree_gp
{
private:
        int size;
        tree *a[MAX_TREE_BUF]; //an array of tree pointers
```

```
public:
        tree_gp(int, int, darray**);
        ~tree_gp();

        int get_lowest_fitness_index(double);
        int get_second_lowest_fitness_index(double);
        double get_eval(int); //return value from ind. eval

        //gp modes
        bool ss(double); //steady state

        bool print_fitnesses(double);
};

#endif
```

6.11 tree_gp.cpp

```
#include <iostream>
#include <time.h>

#include "tree_gp.h"
#include "darray.h"

using namespace std;

tree_gp::tree_gp(int size, int tree_depth, darray **dp)
{
        if(size > MAX_TREE_BUF)
        {
                this->size = 0;
                return;
        }

        this->size = size;

        for(int i = 0; i < this->size; i++)
        {
                this->a[i] = new tree(tree_depth, (*dp));
        }
}


tree_gp::~tree_gp()
{
        for(int i = 0; i < this->size; i++)
```

```
        {
                delete this->a[i];
        }
}


int tree_gp::get_lowest_fitness_index(double dexpected)
{
        int min = 0;
        for(int i = 1; i < this->size; i++)
        {
                if(this->a[i]->fitness(dexpected) < \
                        this->a[min]->fitness(dexpected))
                {
                        min = i;
                }
        }

        return(min);
}


int tree_gp::get_second_lowest_fitness_index(double dexpected)
{
        int min2 = 0;
        int min = this->get_lowest_fitness_index(dexpected);

        for(int i = 1; i < this->size; i++)
        {
                if(this->a[i]->fitness(dexpected) < \
                        this->a[min2]->fitness(dexpected)
                        && i != min)
                {
                        min2 = i;
                }
        }

        return(min2);
}


//simple crosses over the two lowest (best) fitnesses
// converges to locals, but needs tournament selection on subset of
// pop. to get global, etc.
bool tree_gp::ss(double dexpected)
```

```
{
        //Selection
        int min1 = this->get_lowest_fitness_index(dexpected);
        int min2 = this->get_second_lowest_fitness_index(dexpected);

        tree_crossover(&this->a[min1], &this->a[min2]);

        //Mutate
        int rand_val;
        /* initialize random seed: */
        srand ( clock() );
        /* generate secret number: */
        // 0-n values
        rand_val = rand() % this->a[min1]->count_nonterms();
        mutate_nth_nonterm(&this->a[min1], rand_val, 0, 5,
                                this->a[min1]->dp);
}


bool tree_gp::print_fitnesses(double dexpected)
{
        for(int i = 0; i < this->size; i++)
        {
                cout << i << ":" << this->a[i]->fitness(dexpected) << " ";
                if(i != 0 && (i % 5) == 0)
                {
                        cout << endl;
                }
        }

        cout << endl;

        //how to fail?
        return(true);
}


double tree_gp::get_eval(int i)
{
        if(this == NULL || i >= this->size)
        {
                return(-1.0000);
        }

        return(this->a[i]->eval());
```

```
}
```

6.12 test.h

```
#ifndef _TEST_H
#define _TEST_H

bool test_nodes();
bool test_darray();
bool test_trees();
bool test_tree_copy();
bool test_tree_replace();
bool test_tree_crossover();

#endif
```

6.13 test.cpp

```
#include <iostream>
#include <stdlib.h>
#include "main.h"
#include "tree_node.h"
#include "tree.h"


bool test_nodes()
{
        //create nodes
        darray *dp = new darray(200, true);
        tree_node *tp;
        tp = new tree_node(tree_node::plus, 0.0, NULL);
        delete tp;
        tp = new tree_node(tree_node::minus, 0.0, NULL);
        delete tp;
        tp = new tree_node(tree_node::multi, 0.0, NULL);
        delete tp;
        tp = new tree_node(tree_node::div, 0.0, NULL);
        delete tp;
        tp = new tree_node(tree_node::tree_double, 2.001, NULL);
        delete tp;
        tp = new tree_node(tree_node::tree_var, 0.0, &dp);
        delete tp;
        delete dp;

        //copy test
        cout << "Tree node copy test\n";
```

```
        cout << " Plus:\n";
        darray *dp1 = new darray(10, true);
        tree_node *tnp1;
        tree_node *tnp2;
        tnp1 = new tree_node(tree_node::plus, 0.0, NULL);
        tnp1->copy(&tnp2);
        tnp1->print_members();
        delete tnp1;
        tnp2->print_members();
        delete tnp2;
        delete dp1;

        cout << " Tree var:\n";
        dp1 = new darray(2, false);
        dp1->a[0] = 5;
        dp1->a[1] = 7;
        cout << "TS45: " << dp1 << endl;
        tnp1 = new tree_node(tree_node::tree_var, 0.0, &dp1);
        tnp1->copy(&tnp2);
        tnp1->print_members();
        delete tnp1;
        tnp2->print_members();
        dp1->a[0] = 9;
        dp1->a[1] = 9;
        tnp2->print_members();

        delete dp1;
}


bool test_darray()
{
        darray *dp1 = new darray(5, true);
        darray *dp2;
        dp1->copy(&dp2);

        cout << "test_darray: dp1: \n";
        dp1->print_vals();
        delete dp1;
        cout << "test_darray: dp2: \n";
        dp2->print_vals();
        delete dp2;
}
```

```
bool test_trees()
{
        tree *tp;
        darray *dp = new darray(200, true);
        dp->a[0] = 0.2;
        dp->a[1] = 0.3;

        //test making lots of trees
        for(int i = 0; i < 500; i++)
        {
                tp = new tree(5, dp);
                delete tp;
        }
        delete dp;
        cout << "Finished bulk tree creation test\n";

        //eval a tree
        cout << "Eval tree test\n";
        dp = new darray(2, false);
        dp->a[0] = 0.2;
        dp->a[1] = 0.3;
        tp = new tree(5, dp);
        tp->print(0);
        cout << "Tree has " << tp->count_terms() << " terminal(s).\n";
        cout << "Tree has " << tp->count_nonterms() << " non-terminal(s).\n";
        delete dp;
        delete tp;

        //deep tree eval
        cout << "Deep tree eval time test: ";
        clock_t stime, etime, ttime;
        int precision = 1000;
        stime = (clock () / CLOCKS_PER_SEC) * precision;
        dp = new darray(2, false);
        dp->a[0] = 0.2;
        dp->a[1] = 0.3;
        tp = new tree(16, dp);
        etime = (clock () / CLOCKS_PER_SEC) * precision;
        ttime = (etime - stime) / precision;
        cout << ttime << " second(s)\n";
        //tp->print(0);
        delete dp;
        delete tp;

        //Mutate test
```

```
        tp = new tree(5, dp);
        dp = new darray(2, false);
        dp->a[0] = 0.2;
        dp->a[1] = 0.3;
        int n = 10;
        cout << "Term mutation on " << n << " terminal\n";
        SUM_TEMP = 0;
        mutate_nth_nonterm(&tp, n, 0, 5, dp);
        cout << "After mutation:\n";
        tp->print(0);

        //x^3 + 5y^3 - 4xy + 7
        //= (.2)^3 + 5(.3)^3 - 4(.2)(.3) + 7
        //= .008 + .135 - .24 + 7
        //= 6.903
        dp->a[0] = 0.2;
        dp->a[1] = 0.3;
        cout << "Tree fitness: " << tp->fitness(6.903) << endl;
        cout << "Tree eval 1: " << tp->eval() << endl;

        //x^3 + 5y^3 - 4xy + 7
        //
        dp->a[0] = 5;
        dp->a[1] = 7;
        cout << "Tree eval 2: " << tp->eval() << endl;

        //x^3 + 5y^3 - 4xy + 7
        //
        dp->a[0] = 13;
        dp->a[1] = 20;
        cout << "Tree eval 3: " << tp->eval() << endl;

        delete dp;
}


bool test_tree_copy()
{
        //Crossover test
        darray *dp1 = new darray(2, false);
        darray *dp2 = new darray(2, false);
        dp1->a[0] = 0.2;
        dp1->a[1] = 0.3;
        dp2->a[0] = 5;
        dp2->a[1] = 7;
```

```
        tree *tp1 = new tree(5, dp1);
        tree *tp2 = NULL;

        tp1->copy(&tp2);
        cout << "Tree copy test\n";
        cout << " Tree 1:\n";
        //tp1->print(0);
        cout << " " << tp1->eval() << endl;
        delete tp1;
        cout << " Tree 2:\n";
        //tp2->print(0);
        cout << " " << tp2->eval() << endl;

        delete tp2;
        delete dp1;
        delete dp2;
}


bool test_tree_replace()
{
        darray *dp1 = new darray(2, false);
        dp1->a[0] = 0.2;
        dp1->a[1] = 0.3;

        tree *tp1 = new tree(5, dp1);
        tree *tp2 = new tree(5, dp1);

        cout << "Tree replace test\n";
        cout << "Tree 2:\n";
        tp2->print(0);

        cout << "Tree 1 before replace:\n";
        tp1->print(0);
        SUM_TEMP = 0;
        tree_replace_nth_nonterm(&tp1, &tp2, 4);
        delete tp2;
        cout << "Tree 1 after replace:\n";
        tp1->print(0);


        delete tp1;
}

bool test_tree_crossover()
```

```
{
        darray *dp1 = new darray(2, false);
        dp1->a[0] = 0.2;
        dp1->a[1] = 0.3;

        tree *tp1 = new tree(5, dp1);
        tree *tp2 = new tree(5, dp1);

        cout << "Tree crossover test:\n";
        cout << "Tree 1 before crossover:\n";
        tp1->print(0);
        cout << "Tree 2 before crossover:\n";
        tp2->print(0);
        tree_crossover(&tp1, &tp2);
        cout << "Tree 1 after crossover:\n";
        tp1->print(0);
        cout << "Tree 2 after crossover:\n";
        tp2->print(0);
}
```