A Genetic Program for Symbolic Regression
CS 472 Fall 2011
Project 2

Colby Blair

Due October 26th, 2011

Abstract

In computer science, one area of study is that of optimization. Genetic Algorithms (GAs) optimize functions, while Genetic Programming (GP) tries to find functions to fit data. Genetic Programming creates mathematical expression trees, and are useful for finding the functions that are not known, given some data.

This report presents a GP with mathematical non-terminal symbols '+', '-', '*', and '/', and terminal values as contants and variables. This report demonstrates a GP for a very limited domain, and a few different target equations. It talks about the crossover and selection functions, as well as the population representation. This report uses a generational algorithm for population regeneration. This report demonstrates good results for the limited test set, and suggests improvements for the bad results. Finally, this report shows the code used.

Contents

List of Figures

Part I
Introdution

GPs are used to try to approximate a mathamatical expression **tree** (Section 1.1) that describes a function on a graph. In order to improve the approximation, a random set of expression trees are first randomly generated. This set is called the **population** (Section 1.2). 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.

Individuals in the population are then selected, crossed over (or bred together), and then mutated. Exactly how depends on the **random tree generator**, **selection**, **crossover**, **mutation** functions, and in the **generational algorithm**, which are described in Sections 2.2, 2.3, 2.4, 2.5, and 2.6, respectively.

Part II
Experimentation

1 Representation Description

1.1 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. The '/' division operator is protected by **division by zero** errors by simply returning a 0 result if any denominator is 0. 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

```
+------------------+
|tree              |
|tree_node *tnp=|--------------------------------- +------------------+
|...               |                                 |tree_node         |
|int nchildren=    |                                 |enum node_type    |
| 2 (nonterminal)|                                   |double dval       |
|tree *children[]|------                             |double variable*|
+------------------+     |                           +------------------+
                         |
                         ... (many more non-terminals)
         ----------------|
         |               |
         |        +------------------+
         |        |tree              |
         |        |tree_node *tnp=|--------------- +------------------+
         |        |...               |               |tree_node         |
         |        |int nchildren=    |               |enum node_type    |
         |        | 0 (terminal)     |               |double dval       |
         |        |tree *children[]|--NULL           |double variable*|
         |        +------------------+               +------------------+
+------------------+
|tree              |
|tree_node *tnp=|--------------------------------- +------------------+
|...               |                                 |tree_node         |
|int nchildren=    |                                 |enum node_type    |
| 0 (terminal)     |                                 |double dval       |
|tree *children[]|-------NULL                        |double variable*|
+------------------+                                 +------------------+
```

Figure 1: An expression tree (one per individual)

constants or pointers to elements in a variable array (double, or decimal, values). Both are initialized randomly from their respective sets. The variable array can be of **any practical size** $(x_1, x_2, ...x_n)$.

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 (tree) nodes are mutated using point mutation 90% of the time. 10% of the time, 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 evaluation and fitness functions (re)ran.

During the experiment, **depth issues** became a problem. Tree expanded hugely to protect themselves from destructive changes [1], and tree evaluations hung due to huge tree search times. This report's approach is to simply limit the max tree depth result that a **crossover** or **mutation** would have. If a crossover or mutation would result in a tree max depth greater than a global max depth (12-14), the operation was aborted before it happened, and the tree's state was left alone.

1.2 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.

$$\begin{aligned} P = i_1, i_2, ...i_j & \\ & \text{where} \\ & i_n \text{ is a tree} \\ & j = 500 \end{aligned}$$

Figure 2: The representation of the population

2 Functions and Generators

2.1 Fitness Function

$$\begin{aligned} f_i(expected) &= \text{Error} \\ &= |eval_i() - expected| \\ &\text{where} \\ &eval_i() \text{ is the evaluation function in Figure 4} \end{aligned}$$

Figure 3: Fitness function

The $fitness_i()$ and $eval_i()$ functions only ever consider one set of values for variables in the variable array for an optimization. For example, for each generation, $x_1 = .2, x_2 = .3$. The GP would then find a fairly good equation for these values. But not a very good solution for others. This is admitted a weak point in the experiment, and could be improved by running simulations again with an optimized population, and new variable values. An evaluation could also consider multiple value sets for each variable. Due to develop and compute time restraints, this was not improved, but would be fairly simple to do.

$$
\begin{aligned}
eval_i() \quad &= \textstyle\sum_{x=1}^{n} eval_{child_x}() \text{ if } i \text{ type is 'plus'} \\
&= eval_{child_1}() - eval_{child_2}() - ...eval_{child_n}() \text{ if } i \text{ type is 'minus'} \\
&= \textstyle\prod_{x=1}^{n} eval_{child_x} \text{ if } i \text{ type is 'plus'} \\
&= eval_{child_1}()/eval_{child_2}()/...eval_{child_n}() \text{ if } i \text{ type is 'div'} \\
&= i_{constantvalue} \text{ if } i \text{ type is 'tree\_double'} \\
&= i_{variablevalu} \text{ if } i \text{ type is 'tree\_var'} \\
&\text{where} \\
&n \text{ is the number of children (0 or 2 only for now)}
\end{aligned}
$$

Figure 4: Evaluation function

2.2 Random Tree Generator

```
if random_value in 0...9 equals 0 or at depth 0:
        set this subtree to a terminal type; a randomly a contant or
                a variable
else:
        set this subtree to a nonterminal type; randomly a 'plus',
                'minus','multi','div'
        create each child from rand_tree_generator(depth - 1)
```

Figure 5: Random tree generator

2.3 Select 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.

$$selection(P) = i_1, i_2, ...i_j$$

where
P is the entire population
i_j is a random individual
k is the sample size, specified at run time (default = population size / 10, or 10

Figure 6: The selection function

2.4 Crossover Function

```
MAX_DEPTH = 12...14 #depending on simulation
for original tree1:
        select random tree 1 nonterminal
        select random tree 2 nonterminal
        if tree 1 max depth (including random tree 2 nonterminal)
                < MAX_DEPTH:
                        replace it with random tree 2 nonterminal
        else:
                ignore
for original tree2:
        select random tree 1 nonterminal
        select random tree 2 nonterminal
        if tree 2 max depth (including random tree 1 nonterminal)
                < MAX_DEPTH:
                        replace it with random tree 1 nonterminal
        else:
                ignore
```

Figure 7: Crossover function (see Section 2.13)

2.5 Mutation Function

```
for i in random 1..10
if i == 1:
        #regenerate subtree
        for j in rand 1..number of nonterminal nodes in the tree:
                nonterm tree node j = random_gen_nonterm
                return
else:
        #point mutation
        for i in rand 1..number of terminal nodes in the tree:
                term tree node j = rand_gen_term
                return
```

Figure 8: The mutation function

2.6 Generational Algorithm

```
while not bored:
        select k random individuals using selection function
        select the 2 best (tournament selection) as parents, and crossover using
                the crossover function
        for each of the individuals in the population:
                take the best of the 2 parents, and mutate slightly as new mutatant child
                replace individual with new mutant child

        if minimum fitness <= .0001 or 1000 generations:
                bored = true
```

Figure 9: The generational algorithm

Part III
Results

For the results, the equation $x^3+5y^3-4xy+7$ is considered. For optimization, $x = .2, y = .3$. Below are the graphs and tables for the simulation and actual function.

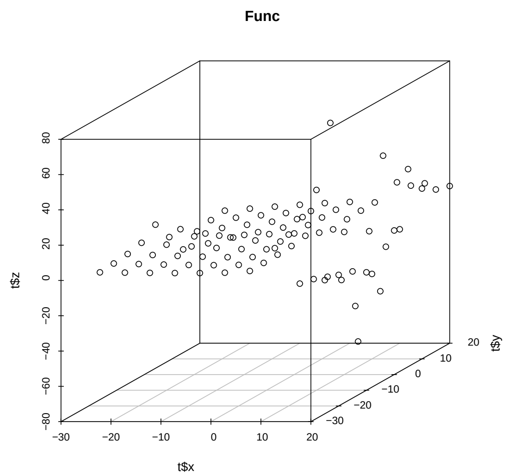

Figure 10: Our best individual results

x, y, z
−25,−25,0.166738
−25,−20,0.730783
−25,−15,1.67356
−25,−10,3.5714
−25,−5,9.4037
−25,0,−2.08205
−25,5,−13.5029
−25,10,−7.70454
−25,15,−5.82272
−25,20,−4.88625
−20,−25,−0.00542462

$-20,-20,0.411167$
$-20,-15,1.10772$
$-20,-10,2.51074$
$-20,-5,6.82914$
$-20,0,-1.66565$
$-20,5,-10.0755$
$-20,10,-5.80985$
$-20,15,-4.42383$
$-20,20,-3.73375$
$-15,-25,-0.150315$
$-15,-20,0.125655$
$-15,-15,0.587389$
$-15,-10,1.51853$
$-15,-5,4.3932$
$-15,0,-1.24925$
$-15,5,-6.78764$
$-15,10,-3.98367$
$-15,15,-3.07047$
$-15,20,-2.61536$
$-10,-25,-0.248672$
$-10,-20,-0.101665$
$-10,-15,0.14472$
$-10,-10,0.643106$
$-10,-5,2.19379$
$-10,0,-0.832846$
$-10,5,-3.73793$
$-10,10,-2.27437$
$-10,15,-1.79479$
$-10,20,-1.55517$
$-5,-25,-0.258277$
$-5,-20,-0.217997$
$-5,-15,-0.149825$
$-5,-10,-0.00956022$
$-5,-5,0.445617$
$-5,0,-0.416444$
$-5,5,-1.14257$
$-5,10,-0.788018$
$-5,15,-0.667263$

```
-5,20,-0.605983
0,-25,-0.0399227
0,-20,-0.0492635
0,-15,-0.0639591
0,-10,-0.0902546
0,-5,-0.14508
0,0,-4.19563e-05
0,5,0.287953
0,10,0.12591
0,15,0.0797593
0,20,0.0581144
5,-25,0.933324
5,-20,1.06349
5,-15,1.28173
5,-10,1.72353
5,-5,3.10046
5,0,0.41636
5,5,-2.14349
5,10,-0.856214
5,15,-0.433356
5,20,-0.221903
10,-25,13.8523
10,-20,17.1148
10,-15,22.5635
10,-10,33.5164
10,-5,67.0533
10,0,0.832762
10,5,-65.6897
10,10,-31.8424
10,15,-20.8874
10,20,-15.442
15,-25,-6.23077
15,-20,-8.10408
15,-15,-11.2313
15,-10,-17.5126
15,-5,-36.7052
15,0,1.24916
15,5,39.6002
```

```
15,10,20.0622
15,15,13.7486
15,20,10.6123
20,-25,-4.24418
20,-20,-5.72411
20,-15,-8.19448
20,-10,-13.1557
20,-5,-28.3093
20,0,1.66557
20,5,31.9836
20,10,16.5348
20,15,11.5439
20,20,9.06494
```

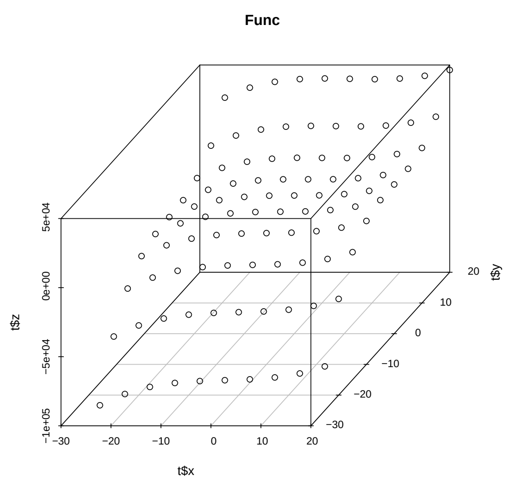

Figure 11: Actual function values

```
x, y, z
-25,-25,-96243
-25,-20,-57618
-25,-15,-33993
-25,-10,-21618
-25,-5,-16743
```

−25,0,−15618
−25,5,−14493
−25,10,−9618
−25,15,2757
−25,20,26382
−20,−25,−88118
−20,−20,−49593
−20,−15,−26068
−20,−10,−13793
−20,−5,−9018
−20,0,−7993
−20,5,−6968
−20,10,−2193
−20,15,10082
−20,20,33607
−15,−25,−82993
−15,−20,−44568
−15,−15,−21143
−15,−10,−8968
−15,−5,−4293
−15,0,−3368
−15,5,−2443
−15,10,2232
−15,15,14407
−15,20,37832
−10,−25,−80118
−10,−20,−41793
−10,−15,−18468
−10,−10,−6393
−10,−5,−1818
−10,0,−993
−10,5,−168
−10,10,4407
−10,15,16482
−10,20,39807
−5,−25,−78743
−5,−20,−40518
−5,−15,−17293

−5,−10,−5318
−5,−5,−843
−5,0,−118
−5,5,607
−5,10,5082
−5,15,17057
−5,20,40282
0,−25,−78118
0,−20,−39993
0,−15,−16868
0,−10,−4993
0,−5,−618
0,0,7
0,5,632
0,10,5007
0,15,16882
0,20,40007
5,−25,−77493
5,−20,−39468
5,−15,−16443
5,−10,−4668
5,−5,−393
5,0,132
5,5,657
5,10,4932
5,15,16707
5,20,39732
10,−25,−76118
10,−20,−38193
10,−15,−15268
10,−10,−3593
10,−5,582
10,0,1007
10,5,1432
10,10,5607
10,15,17282
10,20,40207
15,−25,−73243

```
15,−20,−35418
15,−15,−12593
15,−10,−1018
15,−5,3057
15,0,3382
15,5,3707
15,10,7782
15,15,19357
15,20,42182
20,−25,−68118
20,−20,−30393
20,−15,−7668
20,−10,3807
20,−5,7782
20,0,8007
20,5,8232
20,10,12207
20,15,23682
20,20,46407
```

Part IV
Conclusion

In conclusion, the GP did great for $x = .2, y = .3$. But since it didn't consider any other values yet in optimization, the overall results were very bad. With a little more development and much more compute time, optimizing over all the values that this report graphed for the actual function would lead to a much better GP overall.

This report doesn't show overall good results. But it does show a reasonable approach to GPs and linear regression problems. Much more could be done with different population algorithms (steady state vs. generational), as well as with elitism vs. no elitism. More could be done also with different k values in the selection function, and with more experimentation, better results would be achievable.

Tree growth in regard to depth was a constant concern in development.

Trees tend to grow to hide entrons and minimize destructive mutations and crossovers[1]. To identify this problem, this report set MAX_DEPTH to a low number, like 5. This lead to an opposite problem; minimum fitnesses were always depth 1 trees, or an expression like $x * 15$. The depth was not enough to statistically have good mutations/crossovers; most chagnes to depth 5 were very destuctive, and immediately destroyed. There seemed to be a sweet spot around MAX_DEPTH = 9 before this problem went away. After that, most tree grew immediately to the MAX_DEPTH, and had many intron branches.

Performance wise, this report used a lower level program language (C++) over interpretive langauges like R or Python. This lead to much higher development time, but much better tree evaluation time compared to our past experience. The performance time was nice, but GP's have an area for parallelism in tree evaluations that is perhaps nicer than GA's fitness evaluations. The deeper the tree, the better the solution, although with diminishing returns. Each tree evaluation is independent of another, so this problem lends itself well to true task parallellism.

Part V
Bibliography

References

[1] Harrison, M.L.; Foster, J.A. "Improving the Survivability of a Simple Evolved Circuit through Co-evolution". *Evolvable Hardware, 2004. Proceedings. 2004 NASA/DoD Conferenc* 24-26 June 2004 : 123-129. Print

Part VI
Code

2.7 Makefile

```
PROC=eval
CPP=g++
CPPFLAGS=-g -pg -Wno-write-strings
#CPPFLAGS=-g -pg -Wno-write-strings -DDEBUG_TREE_GP=1
#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
```

2.8 main.h

```
#ifndef _MAIN_H
#define _MAIN_H

#ifdef DEBUG
```

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

#endif
```

2.9 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_replace();
        test_tree_crossover();
        test_tree_get_subtree();
        */
        test_tree_copy();
        return(0);
        #endif

        //Main eval
        darray *dp1 = new darray(2, false);
        dp1->a[0] = .2;
        dp1->a[1] = .3;
        tree_gp *tgp1 = new tree_gp(100, 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;

        int i = 0;
        bool bored = false;
```

```
while(!bored)
{
        //timing stuff
        clock_t stime, etime, ttime;
        int precision = 1000;
        stime = (clock () / CLOCKS_PER_SEC) * precision;

        //tgp1->ss(dexpected);
        tgp1->gen(dexpected);

        double lowest_fitness = tgp1->get_lowest_fitness(dexpected);
        //int mini = tgp1->select_lowest_fitness_index(dexpected);

        etime = (clock () / CLOCKS_PER_SEC) * precision;
        ttime = (etime - stime) / precision;
        cout    << i << ": min fit = "
                << lowest_fitness
                //<< ", min eval = " << tgp1->get_eval(mini)
                //<< ", avg fitness is "
                //<< tgp1->get_avg_fitness(dexpected)
                << ", time = " << ttime
                << endl;

        if(lowest_fitness <= 0.0001 || i > 200)
        {
                bored = true;
        }
        i++;
}

int mini = tgp1->select_lowest_fitness_index(dexpected);
cout << "Lowest individual: eval = " << tgp1->get_eval(mini)
        << ", tree:\n";
tgp1->print_lowest_fitness_tree(dexpected);

//how did we do?
cout << "original func:\n";
cout << "x, y, z\n";
for(double x = -25.; x < 25.0; x += 5.0)
{
        for(double y = -25.0; y < 25.0; y += 5.0)
        {
                //x^3 + 5y^3 - 4xy + 7
                double z = (x * x * x) + ( 5 * y * y *y) - (4 * x * y) + 7;
                cout << x << "," << y << "," << z << endl;
```

```
                }
        }

        cout << "our func:\n";
        cout << "x, y, z\n";
        for(double x = -25.; x < 25.0; x += 5.0)
        {
                for(double y = -25.0; y < 25.0; y += 5.0)
                {
                        dp1->a[0] = x;
                        dp1->a[1] = y;
                        cout << x << "," << y << "," << tgp1->get_eval(mini)
                                << endl;
                }
        }

        //cleanup
        delete tgp1;
        delete dp1;

        return(0);
}
```

2.10 tree_node.h

```
#ifndef _TREE_NODE_H
#define _TREE_NODE_H

#include <iostream>

#include "darray.h"

using namespace std; //for string


#ifdef DEBUG_TREENODE
#define DEBUG_TREENODE_MSG(arg) (cout << arg << endl)
#else
#define DEBUG_TREENODE_MSG(arg) ;
#endif


//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
```

2.11 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)
{
        DEBUG_TREENODE_MSG("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;
                        DEBUG_TREENODE_MSG(" Node type == plus");
                        break;
                }
                case tree_node::minus:
                {
                        this->ntype = val;
                        DEBUG_TREENODE_MSG(" Node type == minus");
                        break;
                }
                case tree_node::multi:
                {
                        this->ntype = val;
                        DEBUG_TREENODE_MSG(" Node type == multi");
                        break;
                }
                case tree_node::div:
                {
                        this->ntype = val;
                        DEBUG_TREENODE_MSG(" Node type == div");
                        break;
```

```
                }
                case tree_node::tree_double:
                {
                        this->ntype = val;
                        //get the float val
                        this->dval = dval;
                        DEBUG_TREENODE_MSG(" Node type == tree_double");
                        DEBUG_TREENODE_MSG(" Node val == " << this->dval);
                        break;
                }
                case tree_node::tree_var:
                {
                        DEBUG_TREENODE_MSG(" 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];
                        DEBUG_TREENODE_MSG(" Node val from rand index " << this->dpi
                        break;
                }
                default:
                        cerr << "ERROR: Node type not set, got val " \
                                << val << endl;
                        exit(1);
        }
}


bool tree_node::copy(tree_node** to)
{
```

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

	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";
}
```

2.12 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(int);
        double fitness(double);

        bool is_term();
        bool is_nonterm();

        int count_terms();
        int count_nonterms();

        bool crossover(tree**,tree**);
        tree *get_nth_nonterm_subtree(int);
        int max_depth(int);

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

//External tree stuff
int tree_get_safe_new_depth(int);
bool mutate(tree**);
bool mutate_nth_nonterm(tree**,int, int, int, darray**);
bool mutate_nth_term(tree**,int, int, darray**);
bool tree_replace_nth_nonterm(tree**,tree**, int);
bool tree_crossover(tree**,tree**);

#endif
```

2.13 tree.cpp

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

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

int SUM_TEMP;

#define MAX_DEPTH 12

/*
This is the structure I am trying to represent

        +-----------------+
        |tree             |
        |tree_node *tnp=|--------------------------------- +-----------------+
        |...              |                                |tree_node        |
        |int nchildren=   |                                |enum node_type   |
        | 2 (nonterminal)|                                 |double dval      |
        |tree *children[]|------                           |double variable*|
        +-----------------+      |                         +-----------------+
                                 |
                                 ... (many more non-terminals)
                -----------------|
                |                |
                |       +-----------------+
                |       |tree             |
                |       |tree_node *tnp=|--------------- +-----------------+
                |       |...              |              |tree_node        |
                |       |int nchildren=   |              |enum node_type   |
                |       | 0 (terminal)    |              |double dval      |
                |       |tree *children[]|--NULL         |double variable*|
                |       +-----------------+              +-----------------+
        +-----------------+
        |tree             |
        |tree_node *tnp=|--------------------------------- +-----------------+
        |...              |                                |tree_node        |
        |int nchildren=   |                                |enum node_type   |
        | 0 (terminal)    |                                |double dval      |
        |tree *children[]|-------NULL                      |double variable*|
        +-----------------+                                +-----------------+
*/

///////////////////////////////////////////////////////////////////////////
//Tree
///////////////////////////////////////////////////////////////////////////
```

```
tree::tree(int depth, darray **dp)
{
        //set depth
        if(depth > MAX_DEPTH)
        {
                cerr << "WARNING: tree create depth of " << depth
                << " attempted, setting instead to MAX_DEPTH = "
                << MAX_DEPTH << endl;
                //this->depth = MAX_DEPTH;
        }
        else
        {
                //this->depth = depth;
        }

        //set dp
        this->dp = (*dp);

        //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(&(this->dp));
                return;
        }

        //Nonterminal
        this->gen_rand_nonterm_tree_node(&(this->dp));
```

```
        //create the children
        this->nchildren = MAX_CHILDREN;
        for(int i = 0; i < MAX_CHILDREN; i++)
        {
                DEBUG_TREE_MSG("DEBUG: tree.cpp: Gen child " << i
<< " at depth " << depth);
                this->children[i] = new tree(depth - 1, &(this->dp));
        }
}


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

        delete this->tnp;
        this->tnp = NULL;
}


bool tree::copy(tree** to)
{
        if(this == NULL)
        {
                (*to) = NULL;
                return(false);
        }
        //DEBUG_TREE_MSG("DEBUG: tree.cpp:");

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


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

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

```
        (*to)->dp = this->dp;

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

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

        DEBUG_TREE_MSG("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 << "ERROR: 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;

        DEBUG_TREE_MSG("DEBUG: tree.cpp: Generating rand term node with type " << ty
        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 << "ERROR: tree.cpp: No term type for node, got type "
                        exit(1);
                }
}


double tree::eval(int depth)
{
        if(depth > MAX_DEPTH)
        {
```

```
		cerr	<< "WARNING: bailing on tree::eval(), MAX_DEPTH "
			<< " exceeded\n";
		return(0.0);
	}

	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(depth + 1);
			}
			return(sum);
		}
		case tree_node::minus:
		{
			double sum = this->children[0]->eval(depth + 1);
			for(int i = 1; i < this->nchildren; i++)
			{
				sum -= this->children[i]->eval(depth + 1);
			}
			return(sum);
		}
		case tree_node::multi:
		{
			double prod = 1;
			for(int i = 0; i < this->nchildren; i++)
			{
				prod *= this->children[i]->eval(depth + 1);
			}
			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(depth + 1) == 0)
				{
					quot = 0;
				}
```

```
                                else
                                {
                                        quot /= this->children[i]->eval(depth + 1);
                                }
                        }
                        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(0) - dexpected));
}


bool tree::is_term()
{
        if(this == NULL)
        {
                return(false);
        }

        if(this->nchildren <= 0)
        {
                return(true);
        }
```

```
        return(false);
}


bool tree::is_nonterm()
{
        if(this == NULL)
        {
                return(false);
        }

        if(this->nchildren <= 0)
        {
                return(false);
        }

        return(true);
}


int tree::count_terms()
{
        if(this == NULL)
        {
                return(0);
        }

        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()
{
        if(this == NULL)
```

```
        {
                return(0);
        }

        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);
}


tree *tree::get_nth_nonterm_subtree(int n)
{
        if(this == NULL || this->is_term())
        {
                //false if I am a child that didn't get a value
                return(NULL);
        }
        else if(this->is_nonterm())
        {
                SUM_TEMP++;
                if(SUM_TEMP >= n)
                {
                        //DEBUG_TREE_MSG("returning " << n << " subtree node\n");
                        return(this);
                }
                //do stuff here

                for(int i = 0; i < this->nchildren; i++)
                {
                        tree *result = \
                         this->children[i]->get_nth_nonterm_subtree(n);

                        if(result != NULL)
                        {
                                return(result);
```

```
                }
            }
        }

        DEBUG_TREE_MSG("didn't find " << n << " subtree node. ? SUM_TEMP = " << SUM_T
        return(NULL);
}


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

        //init SUM_TEMP
        if(depth == 0)
        {
                SUM_TEMP = 0;
        }

        if(depth > SUM_TEMP)
        {
                SUM_TEMP = depth;
        }

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

        return(SUM_TEMP);
}



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(0);
	//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());
}


int tree_get_safe_new_depth(int new_depth)
{
	int safe_new_depth = new_depth;
	if(new_depth > MAX_DEPTH)
	{
		DEBUG_TREE_MSG("restricted to depth " << safe_new_depth);
		safe_new_depth = MAX_DEPTH;
	}

	return(safe_new_depth);
}


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

```
bool tree_crossover(tree **tp1, tree **tp2)
{
        int rand_val;

        //Get subtrees
        // get tp1 subtree
        /* initialize random seed: */
        srand ( clock() );
        /* generate secret number: */
        rand_val = rand() % (*tp1)->count_nonterms(); //0-n values
        tree *tp1_sub = (*tp1)->get_nth_nonterm_subtree(rand_val);
        tree *tp1_sub_orig = NULL; //the original subtree for tp2 crossover
        tp1_sub->copy(&tp1_sub_orig);

        //get tp2 subtree
        /* initialize random seed: */
        srand ( clock() );
        /* generate secret number: */
        rand_val = rand() % (*tp2)->count_nonterms(); //0-n values
        tree *tp2_sub = (*tp2)->get_nth_nonterm_subtree(rand_val);

        //Replace tp1 rand subtree with rand tp2 subtree
        /*TODO: need max depth
        if( (tp1_sub->depth + tp2_sub->depth) <= MAX_DEPTH)
        {
                delete tp1_sub;
                tp1_sub = NULL;
                //TODO: this is where tree_gp->gen() seg faults, ?
                tp2_sub->copy(&tp1_sub);
        }
        //else, abort crossover on these two
        else
        {
                cout << "Aborting crossover 1\n";
        }

        //Replace tp2 rand subtree with original rand tp1 subtree
        if( (tp1_sub_orig->depth + tp2_sub->depth) <= MAX_DEPTH)
        {
                delete tp2_sub;
                tp2_sub = NULL;
                tp1_sub_orig->copy(&tp2_sub);
        }
        //else, abort crossover on these two
```

```
	else
	{
		cout << "Aborting crossover 2\n";
	}
	*/

	return(true);
}


bool mutate(tree **tp)
{
	//reset the temp counter
	SUM_TEMP = 0;

	int rand_val;

	//every 1 in 10 times, do a subtree mutation. otherwise, do point
	// mutation
	/* initialize random seed: */
	srand ( clock() );
	/* generate secret number: */
	// 0-n values
	rand_val = rand() % 10;

	//subtree mutation
	if(rand_val == 1)
	{
		DEBUG_TREE_MSG("nonterm mutation");
		//get random n value
		int rand_n = rand() % (*tp)->count_nonterms();
		//get random new depth value
		int rand_depth = rand() % MAX_DEPTH;
		//cout << "nonterm mutation: " << rand_depth << "\n";
		mutate_nth_nonterm(&(*tp), rand_n, 0, rand_depth,
					&((*tp)->dp));
	}
	else
	{
		DEBUG_TREE_MSG("term mutation");
		//get random n value
		int rand_n = rand() % (*tp)->count_terms();
		mutate_nth_term(&(*tp), rand_n, 0, &((*tp)->dp));
	}
}
```

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

        if( (*tp)->is_term())
        {
                SUM_TEMP++;
        }

        #ifdef DEBUG_TREE
        cout << string(depth, ' ') << depth << ":";
        (*tp)->print_tnp_ntype();
        cout << " = " << SUM_TEMP;
        #endif

        if(n >= SUM_TEMP && (*tp)->is_term())
        {
                #ifdef DEBUG_TREE
                cout << " !mutating!";
                #endif

                //set this tree node to a new rand tree until it is a
                // terminal. TODO: bad, have external gen
                do
                {
                        delete (*tp);
                        //should always get set to terminal with depth = 0
                        (*tp) = new tree(0, &(*dp));
                } while ((*tp)->is_term() != true);

                #ifdef DEBUG_TREE
                cout << endl;
                #endif

                return(true);
        }

        #ifdef DEBUG_TREE
        cout << endl;
        #endif
```

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

        return(true);
}


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++;
        }

        #ifdef DEBUG_TREE
        cout << string(depth, ' ') << depth << ":";
        (*tp)->print_tnp_ntype();
        cout << " = " << SUM_TEMP;
        #endif

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

                //set this tree node to a new rand tree until it is a
                // nonterminal
                do
                {
                        int safe_new_depth = \
                                tree_get_safe_new_depth(depth + \
                                                        new_depth);
                        if(safe_new_depth == 0)
                        {
                                DEBUG_TREE_MSG("WARNING: bailing from "
                                        << "mutate_nth_nonterm, safe depth can "
```

```
                                        << "only be 0 for nonterm. Nothing "
                                        << "possible to do but abort.");
                                return(false);
                        }
                        delete (*tp);
                        (*tp) = new tree(safe_new_depth, &(*dp));
                } while ((*tp)->is_nonterm() != true);

                #ifdef DEBUG_TREE
                cout << endl;
                #endif

                return(true);
        }

        #ifdef DEBUG_TREE
        cout << endl;
        #endif

        //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,
        }

        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);
			(*tp) = NULL;
			(*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);
}
```

2.14 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
```

2.15 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);
}
```

2.16 tree_gp.h

```
#ifndef _TREE_GP_H
#define _TREE_GP_H

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

#define MAX_TREE_BUF 500

#ifdef DEBUG_TREE_GP
#define DEBUG_TREE_GP_MSG(arg) (cout << arg << endl)
#else
#define DEBUG_TREE_GP_MSG(arg) ;
#endif

//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:
        int k; //percentage of selected indiv for tournament selection

        tree_gp(int, int, darray**);
        ~tree_gp();

        //Selection
        int select_lowest_fitness_index(double);
        int select_second_lowest_fitness_index(double);
        void select_lowest_tournament_fitness_indices(double, int*, int*);

        double get_lowest_fitness(double);
        double get_avg_fitness(double);

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

```
        //gp modes
        bool ss(double); //steady state
        bool gen(double); //generational

        bool print_fitnesses(double);
        bool print_depths();
        bool print_lowest_fitness_tree(double);
};

#endif
```

2.17 tree_gp.cpp

```
#include <iostream>
#include <stdlib.h>
#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;
        //default percent of selected indiv for tournament selection
        this->k = 10;

        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];
```

```
        }
}


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

        return(min);
}


int tree_gp::select_second_lowest_fitness_index(double dexpected)
{
        int min2 = 0;
        int min = this->select_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);
}
```

```
void tree_gp::select_lowest_tournament_fitness_indices(double dexpected,
                                                        int *min1, int *min2)
{
        (*min1) = 0;
        (*min2) = 1;

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

        int subset = this->size / this->k;

        for(int i = 1; i < subset; i++)
        {
                DEBUG_TREE_GP_MSG(
                        "select_lowest_tournament_fitness_indices: " << i);

                int j = rand() % this->size;
                if(this->a[j]->fitness(dexpected) < \
                        this->a[(*min1)]->fitness(dexpected))
                {
                        (*min1) = j;
                }
        }

        for(int i = 1; i < subset; i++)
        {
                DEBUG_TREE_GP_MSG(
                        "select_lowest_tournament_fitness_indices: " << i);

                int j = rand() % this->size;
                if(this->a[j]->fitness(dexpected) < \
                        this->a[(*min2)]->fitness(dexpected)
                        && j != (*min1))
                {
                        (*min2) = i;
                }
        }

        return;
}


double tree_gp::get_lowest_fitness(double dexpected)
{
```

```
        if(this == NULL)
        {
                cerr << "ERROR: tree_gp individual is NULL\n";
                exit(1);
        }

        return(this->a[this->select_lowest_fitness_index(dexpected)]->fitness(dexpec
}


double tree_gp::get_avg_fitness(double dexpected)
{
        double sum = 0.0;
        for(int i = 1; i < this->size; i++)
        {
                sum += this->a[i]->fitness(dexpected);
        }

        return(sum / this->size);
}


//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->select_lowest_fitness_index(dexpected);
        int min2 = this->select_second_lowest_fitness_index(dexpected);

        //Crossover
        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));
}
```

```
//generational.
bool tree_gp::gen(double dexpected)
{
        for(int i =0; i < this->size; i++)
        {

        //Selection
        //int min1 = this->select_lowest_fitness_index(dexpected);
        //int min2 = this->select_second_lowest_fitness_index(dexpected);
        int min1;
        int min2;
        this->select_lowest_tournament_fitness_indices(dexpected, &min1, &min2);


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

                DEBUG_TREE_GP_MSG("gen: mutate [" << i << "]");

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

                tree *mutant_child = NULL;
                this->a[min1]->copy(&mutant_child);
                /*
                SUM_TEMP = 0;
                mutate_nth_nonterm(&mutant_child, rand_val, 0, 5,
                                &(mutant_child->dp));
                */
                mutate(&mutant_child);

                //simple replacement that works ok
                /*
                delete this->a[i];
                this->a[i] = new tree(5, &(mutant_child->dp));
                delete mutant_child;
                */

                //replace individual of the population
```

```
                delete this->a[i];
                this->a[i] = NULL;
                mutant_child->copy(&(this->a[i]));
                delete mutant_child;
        }

        return(true);
}


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);
}


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

        cout << endl;

        //how to fail?
        return(true);

}
```

```
bool tree_gp::print_lowest_fitness_tree(double dexpected)
{
        if(this == NULL)
        {
                return(false);
        }

        int min = this->select_lowest_fitness_index(dexpected);
        this->a[min]->print(0);
        return(true);
}


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

        return(this->a[i]->eval(0));
}
```

2.18 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_get_subtree();
bool test_tree_crossover();
bool test_tree_max_depth();

#endif
```

2.19 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(0) << endl;
```

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

        //x^3 + 5y^3 - 4xy + 7
        //
        dp->a[0] = 13;
        dp->a[1] = 20;
        cout << "Tree eval 3: " << tp->eval(0) << 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(0) << endl;
        delete tp1;
        cout << " Tree 2:\n";
        //tp2->print(0);
        cout << " " << tp2->eval(0) << 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_get_subtree()
{
        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);

        SUM_TEMP = 0;
        tree *tp3 = tp1->get_nth_nonterm_subtree(7);
        cout << "Tree get subtree: original tree:\n";
        tp1->print(0);
        cout << "subtree:\n";
        tp3->print(0);

        cout << "Replacing subtree with:\n";
        delete tp3;
        tp3 = new tree(5, &dp1);
        tp3->print(0);
```

```
        cout << "Original tree now:\n";
        tp1->print(0);
}


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);
}
```