Project 2

Alex Cochrane

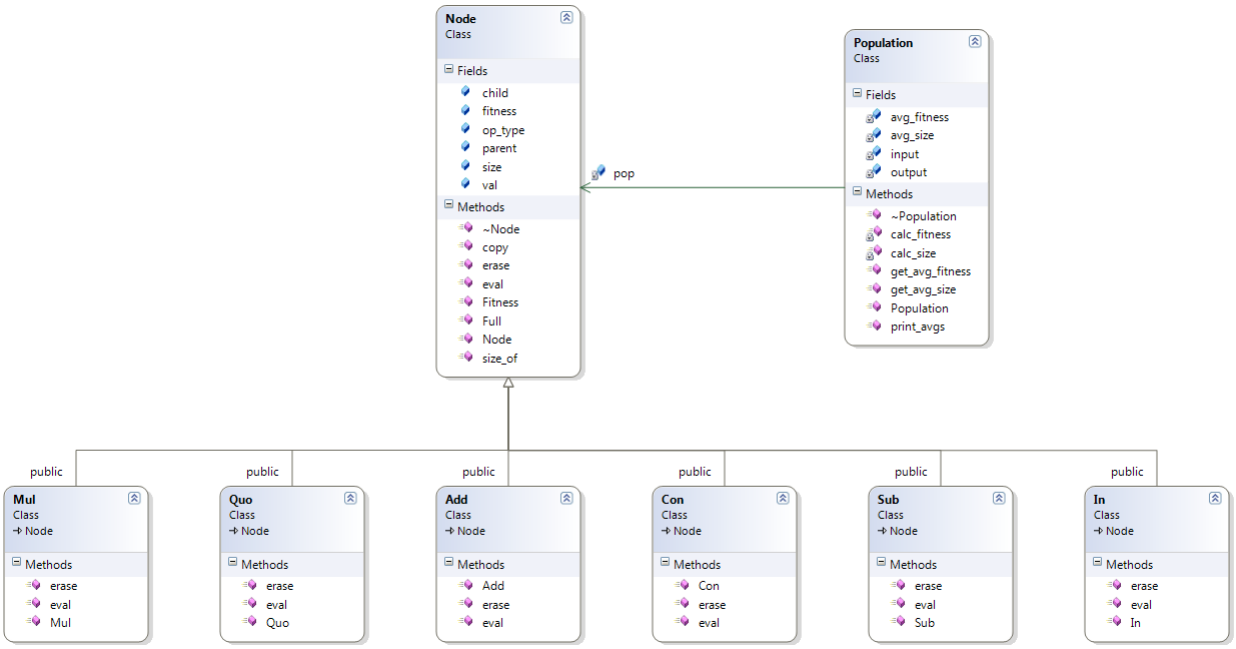
March 4, 2014

1 ABSTRACT

For this project we were tasked with adding functionality to create a Genetic Program that had certain functionality and could easily be expanded upon later. Doing this in C++ I chose to do the typing of individual nodes through polymorphism and inheritance. I decided it would be easiest to build my individuals up as abstract nodes. The nodes themselves will be typed off of there operation. Therefore when I build my populations of individuals I really just have to build a population of nodes and I will not have to worry about their types.

2 ALGORITHM AND INDIVIDUAL DESCRIPTION

For now there are only operations for building up a population of individuals, getting there fitness and removing/copying them. The easiest way to show how these methods work is to show the class diagram for there creation and then do pseudo code for the functions that need an explanation. This class diagram shows both how individuals, or nodes in my code, are formed and structured as well as the population of individuals. For now since all we need is to print out average size and average fitness as well as our best fitness, I have just created these functionalities in my population.



As the diagram shows, each node has the capability of being an addition, subtraction, multiplication, division, constant, and an input. The general attributes of a Node are its fitness, size, children, parent, and for certain individuals average fitness, average size, and value are necessary. These attributes are set by functions and will be discussed more during the following algorithm descriptions.

Pseudo Code: Copy

```
temp = new Node(); // based off of t's op type
while all children have not be reached
    temp->child = temp->copy( t->child );
return temp;
```

This function is the only reason why I have to keep track of types in my solutions. Without this I would not have to worry about them at all. But nonetheless we have to create a new node and therefore we have to return a node at the end. therefore we cannot just have a straight copy, we have to return it from the function. This is one of the negatives to my solution.

Pseudo Code: Evaluate

```
// if non term
return eval(child[0]) op eval(child[1]); // where op is the operator based off of which no
// if const
```

```
return val; // Our constant value
// if input
return in; // return the input value for evaluation
```

For evaluation of non terminals we want to return the evaluation of the right and left branches of the tree using the op for what node it is to perform an operation on the two returned values. If it is a constant we return a constant value generated on creation of the tree and if we are input we return the value of the input given on the call to evaluation.

PSEUDO CODE: ERASE

```
// if non term
while all children are deleted
    child->erase();
delete this;
return;
// if const or input
delete this; // we don't have children
return;
```

If we are a non terminal we need to erase our children. If we are a constant or an input we can just delete ourselves and return because we have no children.

PSEUDO CODE: SIZE

```
temp = 1; // size
while all children have not been looked at
    if( child != NULL )
        temp += size( child );
    else // child == NULL const or input
        temp = 1;
size = temp; // local assignment so it is saved in the node/individual
return temp; // this returns so we can calculate
```

The essence of this code is that it uses recursion to go down to the end nodes and build up the size from the bottom. If the node is a constant or input node, we can say it is of size one and start returning up our recursive calls that were made when we were non terminals.

PSEUDO CODE: FULL

```
parent = node // this is the node passed in

if depth is not greater then our max depth
    while all children are not filled
```

```
        generate a random individual // Add,Sub,...
        call full( depth+1, this); // Fill the child
else at max depth
    while all children are not filled
        generate a random individual // Constant or input
```

This can be used to fill an entire individual structure. If we are not yet at max depth we add nodes that are ops, otherwise we add values that are either constants or input values. This is a quick and effective way at generating the tree, especially when we need to generate multiple ones.

Now I will go into the functions that will be used by the population to compute fitness and size values. This list of functions will be expanded to include our future mutation and crossover functions. For now however we just have the ability to get at fitnesses and sizes, which automatically calculate the fitness and size values when trying to retrieve them. Another function prints out these values in a nice way.

PSEUDO CODE: POPULATION

```
// for when a new population is created
```

---

3 RESULTS

---

4 CONCLUSION

The Genetic Algorithm seems to be working well in the since that it is taking good individuals and mutating them along with crossover to get a better next generation. The problem with this solution however is that a great individual is not present in the population at the start. Without this individual that is almost at the minimum at the start, it is very unlikely to actually find the global minimum.

Some improvements that could have been done were to add an individual to the population that was not random but was in fact very close to the actual minimums vector/fitness. This would guarantee that I had an individual that would be able to move to the minimum. This does take away from the whole purpose of the GA but as stated before, it guarantees a solution is found.

While it would be interesting to get these problems to all work correctly, the run time for them with 1000 individuals in the population takes quite some time. Each fix could take up to 5 minutes to run that fix to see if an improvement was made. Therefore I would like to rewrite the way I do sorting a bit but as just an exercise I have learned quite a bit about GA's and how they operate. The most interesting bit was the affect or the mutation ratio as well as the likely hood for crossover to occur. Changing these values had the greatest affect on the shape as the graph as well. The lower the mutation value was the more linear the graph looked which is why most of these graphs are linear. The values took a steep initial two or three steps and then hill climbed as a group to a minima.

Crossover had this same affect. The more likely two individuals were to crossover, the slower it took to hill climb. This is because good individuals are not always taken. Especially without elitism in the GA, it was difficult to assume that the best individual always stayed. Nonetheless it worked out to were the algorithms were working well together to at least find minima and with more tweaking of values and intervals it is plausible to think that the global minimum would eventually be found.
