Linear Genetic Programming

An assessment of self-programming programs,
using simple esoteric language interpreters in C++
with insight into optimization and parallelization

COIS 4550H – Artificial Intelligence

Assignment #4

Simon Willshire (0491272)

What is Linear Genetic Programming?

A style of genetic programming which is not tree based[1]. Expressions are mapped to chromosome values, and the resultant program is tested by a fitness value[1]. In order to determine the next generation, genetic crossover is applied between two or more chromosomes in order to potentially increase the programs fitness[1]. Syntax generated is generally a very simplified grammatically, as the more complex the language, time to optimize the programs fitness is lengthened, despite the potential to decrease the length of the program (Less instructions overall) [6]. It is also essential that the language is able to compile in a decent amount of time. The language which to compile is one of the more important aspects of this technique, as the performance of such a brute force technique must be efficient in order to remain useful in today's applications.

Other styles of genetic programming typically involve tree-like structures, these methods, or are variants of the linear programming technique discussed in this paper. Multi-Expression Programming (MEP) has proved to be the most popular variant due to its ability to encode several expressions within a programs chromosome. MEP excels in dealing with more complex target expressions, whereas the variant discussed here (LGP) would have to store genetic operators (pre-defined functions) to manipulate its chromosome similarly.

How Does it Work?

Initial chromosome generation

Generally the chromosome generation is simply filled by a linear congruential generator (fast random number generator using an initial seed). However, a method with more entropy can be employed by other means, such as hardware random generators which are supplied by many processors today which make use of atmospheric noise, or an external source of seeding.

Languages

Using a General Purpose Language (GPL)
By using a restricted set of C for example, using limited processor registers and memory allocation for each program. The importance here is restricting the language up to a simplified set of data types and operators. Other available GPLs are higher level, typically script-like, and are faster compilation time for the potentially lower performance; languages such as python. Other low level languages can be compiled in script-like performance using interpreter-like solutions.

Using Machine Code or Assembly
By using random instructions at the lowest level of software, programs are capable of becoming the most efficient on target machines. A step above machine language (Assembly) is sometimes used in order to be more readable to allow for a more readable format. However, this option is capable of causing many damaging instructions, such as accessing and setting memory locations outside of typical operating bounds.

Using a GPL Interpreter
Similar to the above methodology, however, instead of passing each generation's chromosome to generate the GPL code, one can simply query the behaviour of the chromosome, and operate within the host language. This has many advantages over other methods, as the capabilities of the host language do not

necessarily need to be limited: such as error handling, multithreading, and other hardware support. One flaw of this method derives from its advantage: by using an interpreter scheme, we lose the ability to operate at a lower processor level unless otherwise programmed in the host language.

Using a Custom Interpreter

Depending on the program which you intend to solve, you may choose to adjust the grammar of the language you are interpreting. For example, you may choose to restrict memory and operate-able variables, or customize the functions available to the language by defining optimized functions for the language to use in order to solve/accomplish said task.

Chromosome Mutation/Selection Techniques

Many techniques can be employed in order to differentiate the next generations produced program. Here are some examples which can be used, and what their advantages/disadvantages may be.

- **Roulette**: Random segments of the chromosome are mutated.
- **Roulette Crossover**: Random segments of next generation's chromosome are combined with other high fitness programs.
- **Genetic Crossover**: Can use single point or multi-point selection of the chromosome. For example 1/2, 1/2 combination of two programs, or 1/3, 1/3, 1/3 of three, etc. Possibilities may not necessarily be of equal proportions (could use indexed offsets of array), or adjust segments based on the fitness of the programs.

What can it solve?

Technically speaking, anything and everything, with heavy dependence on the fitness and constraints the user provides. This approach is essentially of self-programming design. However, the question as to how much resources and time you wish to allocate towards solving the problem is the limitation.

Realistically speaking, there are many caveats which must be mitigated in order to use this technique with viable results. These caveats come from the nature of the technique, where optimization of both compilation time and process execution time of the bred programs becomes of increasing concern (bloat). To resolve these issues, one may reduce chromosome length (reducing the programs capabilities), reducing the memory allocation a single generated program may use, as well as adjusting the fitness function in order to negatively feedback upon this program (this may also reduce accuracy of results should the bloated program produce good results, but takes its time in order to achieve it).

Applications using LGP

Discipulus is modelling software which employs Linear Genetic Programming techniques. Specifically makes use of binary sequences for genomes, and operates completely in machine code, each generation optimizes model parameters during runtime. *Discipulus* claims to be 60-200 times faster than other modelling platforms due to this design choice[3]. The application makes use of very simple techniques: generating random pool of programs, from which 4 are selected randomly (technique unspecified, assuming Linear Congruential Generator), these 4 are compared and 2 are selected with the highest fitness[3]. These programs are now are processed through a genetic crossover technique to create 2 new programs, both of which get added back into the program pool, and removes the 2 worst fitness programs[3]. This process is then repeated until the fitness threshold has been met[3]. The genetic crossover technique between the pair of fit programs uses a basic technique to swap instructions between the two

programs[3]. Afterward, a mutation takes place which swaps operators intelligently, for example the plus (+) operator may become a (*) operator[3]. The advantage of this crossover method comes from the ability to always have safe compilation syntax, as an operator switch is less likely to cause errors (other than divide by 0 and the like). After the fitness threshold has been reached, *Discipulus* offers a decompilation of their machine code into human readable ANSI C, Java, or Intel Assembly instructions [3].The resultant code can then be optimized to the user's liking[3]. Once satisfied, the result can be compiled into a DLL, or COM object[3]. This modelling technique produces an increased performance, and as such the modelling can continue to become more efficient and accurate the more time it spends reaching a higher fitness value[3].

Demonstrations and Comparisons

This simple LPG demonstration was written in C++, and interprets LPG random numbers into instructions for each program. The program is executed from the first instruction to the last instruction; the number of instructions is configurable in order to optimize results. The following crossover, mutation methods, and fitness functions were used to give different results for each LGP language defined.

The Simple 7 Instruction LGP: Memory Management

For the first set of demonstrations, the following 7 instruction language was implemented:
This language is quite similar to popular esoteric languages.

> Increment	Pointer location of programs memory increments by 1
< Decrement	Pointer location of programs memory decrements by 1
+ Add	Memory value at location is incremented by 1
- Subtract	Memory value at location is decremented by 1
[Loop Begin	Loops until Loop End or End/Break before last instruction. Number of iterations determined by current memory value.
] Loop End	End of loop as invoked above.
! End/Break	Breaks loop.

Crossover Methods

a) Roulette Crossover: Instructions are modified between the top two programs at random indexes.
b) One point Crossover: Instructions are split at a random index between top two programs.

Mutation Methods

1. Random with chance of: (Maximum fitness – current fitness) * (1 / maximum instructions) on 2 best fit programs.
2. Increments/Decrements instruction with above chance on 2 best fit programs.

Fitness Methods

A. Sum of all memory locations:

$$F=\left(\sum m\right)$$

B. Sum of all memory locations, with instruction count optimization:

$$F=\left(\sum m\right)-\frac{instruction\ count}{max\ instructions}$$

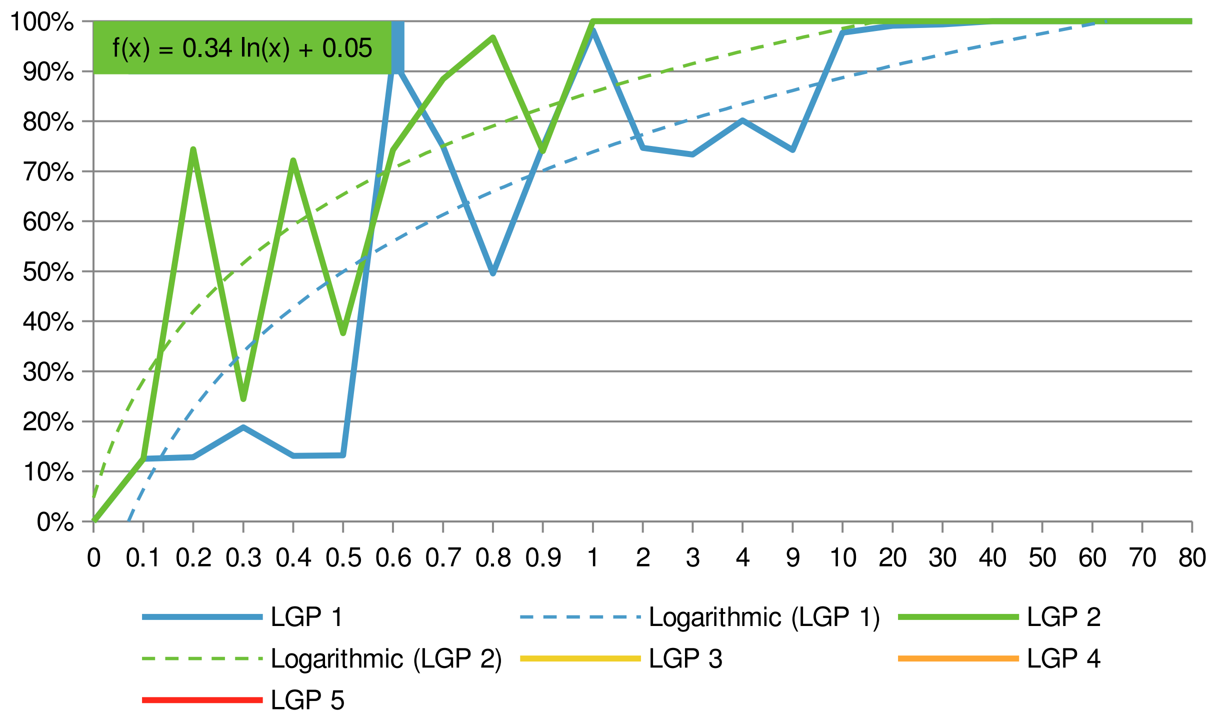
C. Word match memory locations in ASCII, goal = { L, E, A, R, N, I, N, G }

$$F=\sum\left|goal[i]-m[i]\right|$$

Demonstration Configuration

LCG ID	# Instructions	Memory	# Programs	Crossover	Mutation	Fitness
1	32	8 bytes	128	Method [a]	Method [1]	Method [A]
2	32	8 bytes	128	Method [a]	Method [2]	Method [A]
3	32	8 bytes	128	Method [b]	Method [1]	Method [B]
4	64	8 bytes	128	Method [b]	Method [2]	Method [B]

Results



Number of generations in thousands to 99% fitness threshold averaged over 100 LCG seeds.

The above plot shows how the different configurations of linear genetic programming can influence the number of generations and accuracy of the algorithm. For instance, comparing fitness method A and B (one reduces fitness when more instructions are used in the program, to give the program some optimization), one can see that number of generations has dramatically increased, but with better overall results. Another important parameter to consider is the number of possible instructions available to the program, with large sets of instructions the number of possible program combinations increases, giving the program more complexity, however potentially giving the possibility for an increased fitness value. The other extreme is to reduce the complexity and wait for a sufficient fitness threshold in a reduced programs set of instructions more efficiently. Customizing the fitness function even with the slightest difference may yield completely different results. For instance, one may calculate program execution time and reduce the computation needed in finding the solution, which would also drastically change the time

in order to produce a feasible program within threshold requirements. During this demonstration, we can see that mutation method [2] is more effective at improving overall fitness by roughly 5-10% increase. Logically this makes sense, as instructions are grouped according to type, where pointer increment/decrement are beside one another, and so are value increment/decrement, allowing for more variance as opposed to relying more heavily on entropy during program mutation.

The 10 Instruction LGP: Function Modeling and Estimation

The second style of demonstration will attempt to solve the following situation: The program will accept a set of input values, and output a result correlating to the input. This type of LGP also outlines the use of a strict fitness function, where the fitness is based upon a known outcome. This method of modeling is estimation of the original function (and can become more/less accurate by altering a fitness threshold, +/- the true fitness). The following language modifications were made in order to suite the nature of the problem more accurately to use less instructions and memory locations:

> Increment	Primary pointer location of programs memory increments by 1.
< Decrement	Primary pointer location of programs memory decrements by 1.
+ Add	Memory value at location is incremented by memory value at second memory location.
- Subtract	Memory value at location is decremented by memory value at second memory location.
[Loop Begin	Loops until Loop End or End/Break before last instruction. Number of iterations determined by primary pointers current memory value.
] Loop End	End of loop as invoked above.
) Increment	Second pointer location of programs memory increments by 1.
(Decrement	Second pointer location of programs memory decrements by 1.
***** Multiply	Memory value at location is multiplied by memory value at second memory location.
/ Divide	Memory value at location is multiplied by memory value at second memory location.

Increasing the language size will increase generation count, however permit less instruction count overall. The language is now capable of tracking two pointer locations of the memory, and invoking multiplication and division between them; once the calculation has completed, the new value is stored within the primary pointer location.

Crossover Method

One point Crossover: Instructions are split at a random index between two programs.

Mutation Method

Increments or Decrements with chance of: current fitness * (1 / maximum instructions) on 2 best fit programs. This mutation method was chosen from the last test, as it produced better results as previously discussed.

Fitness Methods

Match memory locations of goal functions (i[] is input, f[] is output memory)

A. $f[0] = -i[0] + i[1]$

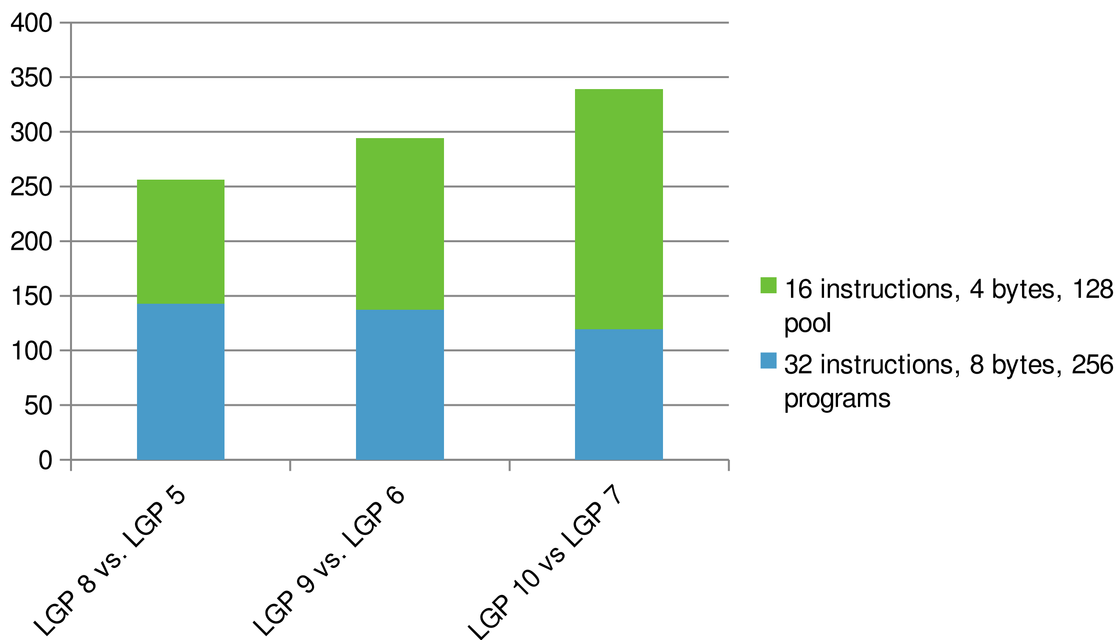
B. $f[1] = (i[0] - i[1])^2$

C. $f[2] = \sqrt{(i[0] * i[1])}$

Demonstration Configuration

LCG ID	# Instructions	Memory	# Programs	Fitness
5	16	4 floats	128	Method [A]
6	16	4 floats	128	Method [B]
7	16	4 floats	128	Method [C]
8	32	8 floats	256	Method [A]
9	32	8 bytes	256	Method [B]
10	32	8 bytes	256	Method [C]

Results



Number of generations in %99.99 accuracy threshold averaged over 1000 LCG seeds.

The above graph shows the function modeling according to three different functions, across two configurations of memory sizes, program length, and the total number of programs to process (similar to a gene pool). With increased execution time (roughly 66% slower by doubling these attributes), decreases the generations required to reach the fitness threshold. However, it is important to note that these demonstrations are run single threaded and linear sorting techniques. Judging by the difference of equation, more generations are required to mimic more complex fitness functions. When implementing very complex models (as would be the use of LGP in software), the time to compute would increase directly with the complexity of the fitness function.

Analysis and Comparison

Linear Genetic Algorithm techniques can apply to a varying set of fitness functions, and can accurately model many features of computing. Our demonstrations have shown that more abstract memory management takes much more computing power than that of simple function modeling; where our tests on average modeled the equation in ~296 generations (of like configurations), over that of our memory sum and word matching, where it took upwards of 1000 generations or more.

Strengths and Weaknesses

An outright strength of linear genetic programming is its ease of implementation mostly due to its brute force design. This technique of genetic programming is also capable of restricting the solution, as well as the ability for completely unrestricted bounds of each generation's capabilities. In short this technique is able to be both restricted and unrestricted in its operating bounds. In addition, Linear Genetic Programming is capable of short execution time because the generated program is capable of operating at both low or high level depending on the complexity of the problem, and what features the host language should wish to surface in the solution to the problem[6].

Parallel Linear Genetic Programming is also a possibility, where individual threads may be used to compile and/or compare programs to use increasingly paralleled computing architectures. Results can be further improved through the use. Further parallelization can be implemented within the execution of the program; this option would be most easily implemented within an interpretive language as opposed to the GPL route[2]. More implications of parallelizing the programs execution may occur when combining results of each processing thread, where memory should not be shared between them; otherwise thread execution order will come into effect with different operations[2]. Since typical execution of these programs have simple instruction sets, genetic programming would be ideal for GPU processing, where each processing core could handle a single genome, and pooling the results in separate threads with shared memory[4].

All of these customization methods also may implicate this techniques success in solving the problem. For example, one may not realize that results produced by one configuration of the technique and its fitness function may not approach the intended results, tweaking the fitness function and the techniques operating parameters may produce better/worse results which may take up valuable time - all must be taken into consideration.

By comparing linear methods (i.e. LGP) of genetic programming to multi-expression programming, or any other tree-like structure being used to develop the output, LGP is able to have variable length programs and has proven in more complex tests to be less error prone than that of MEP [6].
LGP typically follows a logarithmic solution curve, where increasing number of generations results in high solution rate at the beginning with less probability of solution with increased generations computed (As seen in our initial Simple LGP demonstration's results). Other variants of genetic programming are more linear in solution, however typically do not gain as much abrupt gain at lower generations, such as the previously discussed MEP method, Genetic Expression Programming (GEP) which is also a linear approach variant, as well as the Grammatical Evolution (GE) method (steady-state algorithm, similar to MEP) [6].

The use of genetic programming in applications like *Discipulus* produce astoundingly simple and fast results which would have previously taken much more time and money spent on hardware to complete. However, the nature of Linear Genetic Programming allowed the solution to be deceptively simple. With new techniques in the parallelization of genes (each program can be considered to be a task in a thread), or even parallelization of tasks (simplification and optimization of sets of instructions) would speed the process even further, allowing more complex programs to be produced.

As with any genetic programming, the expectation as to what needs to be solved needs to be known (its fuzzy fitness measure). Should the fitness test be able to suit a more abstract method; something that is determined by the programs ability to be more *creative* would really makes this method of programming interesting.

Future Applications

The development of parallelization in Linear Genetic Programming are starting to emerge, some have coined this new method as PLGP[2]. PLGP is structured to execute lists of instructions in parallel, whether that execution is on a single machine, a GPU or within several machines in a distributed computing cluster, the same architecture of the program applies. After this execution, all resulting segments are combined to produce a programs output[2]. This method allows for near instantaneous crossover and mutation speeds, as each segment has already been established and executed; mutation and recompilation in a parallel fashion further increases PLGP's performance [2]. The key concept here is to foresee disruptive instructions within a set of instructions, and group the disruptive regions at the beginning or end of the segment to be parallelized; doing so will lessen reprocessing of gene segments[2]. This approach is very similar to Dynamic Programming, where we are building from an existing solution already assumed, and executing the areas of necessity. Further optimization in said PLGP methods can be achieved by reducing redundant instructions in LGP code (Structural Introns) by further segmentation of code[2]. Through this parallelization technique, longer programs can be executed in less time, allowing programs to tackle more severe problems with longer length programs becoming more optimum to solve [2].

Once LGP is capable of this parallelization, computation could be moved to a GPU for even increased capabilities, one such experiment was conducted by Simon Harding, allowing several hundred times faster computation, for reference: these results were produced on a card with roughly 554 GFLOPS, where todays cards are capable of driving 5000 GFLOPS (without breaking into server-line GPUs), never mind the future. The experiment suggested various forms of APIs available, such as SH, Brook, PyGPU, and Accelerator, and chose Accelerator[4]. The experiment yielded results with varying configurations; however the main takeaway here was the sheer yield in performance. Limitations of this method are the interface between CPU and GPU architectures capabilities of data transfer, and as of today not many APIs are designed specifically for the purpose of genetic programming[4]. Possibly through the use of OpenCL, or Nvidia's CUDA further tests in the future for this level of parallelization is needed.

In conclusion, genetic programming whilst making using of linear genetic programming enables many methods of customization to cater to the problem at hand. It is by no means a silver bullet solution to all your needs, and must be tested on various scenarios before optimum use can be achieved. Through optimization in software systems, as well as parallelization on different hardware platforms, this technique will become increasingly useful in the future, where the demand for software is ever increasing, and not enough time and resources to produce a solution as effectively as a machine capable of learning it. One exception remains, defining a fitness method in order for an appropriate destination to be set is entirely another matter, possibly with other computers defining those functions. Who knows, maybe its computers all the way down.

References

[1] A Field Guide to Genetic Programming. (n.d.). Retrieved December 3, 2014, from http://cswww.essex.ac.uk/staff/poli/gp-field-guide/71LinearGeneticProgramming.html

[2] Downey, C., & Zhang, M. (n.d.). Parallel Linear Genetic Programming. Retrieved December 7, 2014, from http://ecs.victoria.ac.nz/foswiki/pub/Main/TechnicalReportSeries/ECSTR10-25.pdf

[3] Francone, F. (n.d.). Discipulus™ Linear-Genetic-Programming Software: How it Works. Retrieved December 7, 2014, from http://www.rmltech.com/doclink/Discipulus How It Works.pdf

[4] Harding, S., & Banzhaf, W. (n.d.). Fast Genetic Programming on GPUs. Retrieved December 2, 2014, from http://www.cs.mun.ca/~banzhaf/papers/eurogp07.pdf

[5] Nordin, P., Banzhaf, W., & Francone, F. (n.d.). Efficient Evolution of Machine Code for CISC Architectures using Blocks and Homologous Crossover. Retrieved December 7, 2014, from http://www.rmltech.com/doclink/aigp31.pdf

[6] Oltean, M., & Gros¸an, C. (n.d.). A Comparison of Several Linear Genetic Programming Techniques. Retrieved
November 27, 2014, from https://www.complex-systems.com/pdf/14-4-1.pdf

Appendices

The following code was used for the first and second demonstrations with slight modifications to behaviour using preprocessor defines for fitness, mutation and crossover methods.

GP.h

```
// COIS-4550H – Assignment #4 – LGP Demonstration
// Simon Willshire (0491272)
#include <float.h>
#include <stdlib.h>
#include <stdio.h>
#include <random>
#include <time.h>
#include <thread>
#include <chrono>

#define ITERATIONS  1000

#define FITNESS_A
//#define FITNESS_B
//#define FITNESS_C

#define FITNESS_A_FUNC     (-i[0] + i[1])
#define FITNESS_B_FUNC     ((i[0] - i[1]) * (i[0] - i[1]))
#define FITNESS_C_FUNC     (sqrt(i[0] * i[1]))

#define MAX_INSTRUCTIONS        32
#define MAX_MEMORY              8
#define INITIAL_STRANDS         256
```

```
#define MAX_FITNESS             0.f
#define RAND_INSTR              (((float)rand() / RAND_MAX) * 10)

#define SCREEN_UPDATE           100000
#define SLEEP_WAIT              1

#define FITNESS_THRESHOLD       -0.001f

enum INSTRUCTIONS
{
        INSTR_INCR = 0,         // Increment            Increment memory location
        INSTR_DECR = 1,         // Decrement            Decrement memory location
        INSTR_ADD = 2,          // Add                  Add current memory value
        INSTR_SUB = 3,          // Subtract             Subtract current memory value
        INSTR_LPB = 4,          // Loop Begin           Following instructions until
        INSTR_LPE = 5,          // Loop End              This instruction, mem ptr
times.
        INSTR_SINC = 6,                 // Increment Secondary
             INSTR_SDEC = 7,            // Decrement Secondary
             INSTR_MULT = 8,            // Multiply primary / secondary memory values
             INSTR_DIV = 9              // Divide primary / secondary memory values
};

class Genome
{
public:
        Genome(void);
        ~Genome(void);

        void status(void);
        void clear(void);
        void reset(void);

        unsigned int numInstruc;

        int layers;
        int location;
        int second;

        float* memory;
        unsigned char* instruction;
};

static bool interpret(Genome* g, int* index);
static void interpret(Genome* g);

static void parse_incr(Genome* g);
static void parse_decr(Genome* g);
static void parse_add(Genome* g);
static void parse_sub(Genome* g);
static void parse_lpb(Genome* g, int* index);

static float fitness(Genome* g);
static void crossover(Genome* g1, Genome* g2);
static void mutate(Genome* g1, Genome* g2, float curFitness);
```

GP.cpp

```
// COIS-4550H – Assignment #4 – LGP Demonstration
// Simon Willshire (0491272)

#include "GP.h"
```

```
Genome::Genome(void)
{
        instruction = new unsigned char[MAX_INSTRUCTIONS];
        memory = new float[MAX_MEMORY];
        this->reset();
}
Genome::~Genome(void)
{
       delete[] instruction;
       delete[] memory;
}

void Genome::status(void)
{
        for(int i = 0; i < MAX_INSTRUCTIONS; ++i)
        {
                switch(this->instruction[i])
                {
                        case INSTR_INCR: printf(">"); break;
                        case INSTR_DECR: printf("<"); break;
                        case INSTR_ADD: printf("+"); break;
                        case INSTR_SUB: printf("-"); break;
                        case INSTR_LPB: printf("\n\t["); break;
                        case INSTR_LPE: printf("]\n"); break;
                        case INSTR_SINC: printf(")"); break;
                        case INSTR_SDEC: printf("("); break;
                        case INSTR_MULT: printf("*"); break;
                        case INSTR_DIV: printf("/"); break;
                        default: printf("?"); break; // Oh-oh!
                }
        }
}

void Genome::clear(void)
{
        for(int i = 0; i < MAX_MEMORY; i++)
                this->memory[i] = 0;

        location = 0;
             numInstruc = 0;
             layers = 0;
             second = 0;
}

void Genome::reset(void)
{
       int i;

       location = 0;
       numInstruc = 0;
       layers = 0;
       second = 0;

       for(i = 0; i < MAX_INSTRUCTIONS; i++)
             instruction[i] = RAND_INSTR;

       clear();
}

// Parse Functions

void parse_incr(Genome* g)
```

```
{
    if(g->location + 1 >= MAX_MEMORY)
        g->location = 0;
    else
        g->location++;
}
void parse_decr(Genome* g)
{
    if(g->location - 1 < 0)
        g->location = MAX_MEMORY;
    else
        g->location--;
}
void parse_add(Genome* g){ g->memory[g->location]++; }
void parse_sub(Genome* g){ g->memory[g->location]--; }
void parse_lpb(Genome* g, int* index)
{
       // Go to next instruction!
        if(*index + 1 > MAX_INSTRUCTIONS)
            return;

        g->layers++;

        *index = *index + 1;

        int t;
        int b = MAX_INSTRUCTIONS;       // If no proper end to the loop!

        for(int n = 0; n < g->memory[g->location]; n++)
        {
                // Look ahead for INSTR_LPE
                for(t = *index; t < MAX_INSTRUCTIONS; t++)
                {
                        if(g->instruction[t] == INSTR_LPE)
                        {
                            b = t;
                            break;
                        }
                        else
                            interpret(g, &t);
                }
        }

        *index = b;     // Jump to the end of the loop!
}
void parse_sinc(Genome* g)
{
    if(g->second + 1 >= MAX_MEMORY)
        g->second = 0;
    else
        g->second++;
}
void parse_sdec(Genome* g)
{
    if(g->second - 1 < 0)
        g->second = MAX_MEMORY;
    else
        g->second--;
}
void parse_mult(Genome* g)
{
    g->memory[g->location] *= g->memory[g->second];
}
```

```
void parse_div(Genome* g)
{
    if(g->memory[g->second] != 0)
       g->memory[g->location] /= g->memory[g->second];
       else
           g->memory[g->location] = 0.f;
}

// Interpret Functions

bool interpret(Genome* g, int* index)
{
    g->numInstruc++;

    switch(g->instruction[*index])
    {
    case INSTR_INCR: parse_incr(g); break;
    case INSTR_DECR: parse_incr(g); break;
    case INSTR_ADD: parse_add(g);  break;
    case INSTR_SUB: parse_sub(g);   break;
    case INSTR_LPB: parse_lpb(g, index);  break;
    case INSTR_LPE: break;
    case INSTR_SINC: parse_sinc(g); break;
    case INSTR_SDEC: parse_sdec(g); break;
    case INSTR_MULT: parse_mult(g); break;
    case INSTR_DIV: parse_div(g); break;
    }
    return true;
}
void interpret(Genome* g)
{
    for(int i = 0; i < MAX_INSTRUCTIONS; i++)
       interpret(g, &i);
}

// Genome Functions

float fitness(Genome* g)
{
    float f = 0;
    const int i[2] = { rand(), rand() };

#if defined FITNESS_A
    f -= abs(FITNESS_A_FUNC - g->memory[0]);
#elif defined FITNESS_B
    f -= abs(FITNESS_B_FUNC - g->memory[1]);
#elif defined FITNESS_C
    f -= abs(FITNESS_C_FUNC - g->memory[2]);
#endif
    return f;
}

void crossover(Genome* g1, Genome* g2)
{
    unsigned char t;

    for(int i = 0; i < MAX_INSTRUCTIONS; ++i)
    {
       for(int a = 0; a < std::floor(MAX_INSTRUCTIONS / 2); a++)
       {
           t = g1->instruction[i];
           g1->instruction[i] = g2->instruction[i];
           g2->instruction[i] = t;
```

```
        }
        for(int a = std::ceil(MAX_INSTRUCTIONS / 2); a < MAX_INSTRUCTIONS; a++)
        {
            t = g2->instruction[i];
            g2->instruction[i] = g1->instruction[i];
            g1->instruction[i] = t;
        }
    }
}

void mutate(Genome* g1, Genome* g2, float curFitness)
{
    // Mutate bit instruction at random index
  int nm = (int)((float)(MAX_FITNESS - curFitness * 0.1f) * (1.f / MAX_INSTRUCTIONS));
        if (nm > MAX_INSTRUCTIONS || nm < 0)
                nm = MAX_INSTRUCTIONS;

    int ni1, ni2;

    for(int i = 0; i < nm; ++i)
    {
        ni1 = (int)((MAX_INSTRUCTIONS) * ((float)rand() / RAND_MAX));
        ni2 = (int)((MAX_INSTRUCTIONS) * ((float)rand() / RAND_MAX));

        if((rand() & 0x1) == 0)
                (g1->instruction[ni1])++;
        else
                (g1->instruction[ni1])--;

        if((rand() & 0x1) == 0)
                (g2->instruction[ni2])++;
        else
                (g2->instruction[ni2])--;

        if(g1->instruction[ni1] < 0)
                g1->instruction[ni1]=10;
        else if (g1->instruction[ni1] > 10)
                g1->instruction[ni1]=0;

        if(g2->instruction[ni2] < 0)
                g2->instruction[ni2]=10;
        else if (g2->instruction[ni2] > 10)
                g2->instruction[ni2]=0;
        }
}

int main(int argc, char *argv[])
{
        int curSeed = 0;
        int generations, i;
        int maxFitIndex;

        float maxFit, tempFit;

        std::vector<int> times;

        char stime[9];

        Genome mGenomes[INITIAL_STRANDS];

        for(int s = 0; s < ITERATIONS; s++)
        {
                for(i = 0; i < INITIAL_STRANDS; i++)
```

```
                    mGenomes[i].reset();

            generations = maxFitIndex = 0;
            maxFit = tempFit = -99999999;

            while(1)
            {
                    for(i = 0; i < INITIAL_STRANDS; ++i)
                            interpret(&mGenomes[i]);

                    if((generations < SCREEN_UPDATE && ((generations <= 500 &&
generations % 100 == 0) || (generations < 10000 && generations % 1000 == 0) ||
generations % 10000 == 0))
                    || generations % SCREEN_UPDATE == 0
                    || maxFit >= FITNESS_THRESHOLD)
                    {
                   std::this_thread::sleep_for(std::chrono::milliseconds(SLEEP_WAIT));

                   if(maxFit >= FITNESS_THRESHOLD)   // Fitness threshold met?
                    {
                            system("cls");
                            printf("\nIt: %i, Fitness threshold met, took %i
generations.\n", s,  generations);
                            printf("Final Program: \n");
                            mGenomes[maxFitIndex].status();
                            printf("\n");

                            times.push_back(generations);
                            break;
                    }
            }

            // Select the highest fitness of the strands
            for(i = 0; i < INITIAL_STRANDS; ++i)
            {
                    tempFit = fitness(&mGenomes[i]);

                    if (maxFit < tempFit)
                    {
                            maxFit = tempFit;
                            maxFitIndex = i;

                            if (tempFit < FITNESS_THRESHOLD)
                            {
                                crossover(&mGenomes[maxFitIndex], &mGenomes[i]);
                                mutate(&mGenomes[maxFitIndex], &mGenomes[i], tempFit);
                            }
                            else
                                break;
                    }
                    else
                            mutate(&mGenomes[i], &mGenomes[i], tempFit);
                            mGenomes[i].clear();
                    }

            ++generations;
        }
    }

    system("cls");

    int tcount = 0;
    float averageGen = 0.f;
```

```
    int msize = 0;

    for(auto t : times)
      averageGen += t;

      averageGen /= times.size();

      printf("Average generations for all seeds: %f\n", averageGen);

      scanf("%c");

      return 0;
}
```