Project 2: Genetic Algorithms

CS 423/523: Complex Adaptive Systems
Assigned Feb. 20th
Due: Mar. 10th, 6:00pm Mountain Time
Version: 1.2

1 Change Log

1.1 (2-27-2017):

1. Fixed some typos: “tyring” $\implies$ ”trying” and “MB” $\implies$ ”GB”.
2. Specified that warriors cannot be hardcoded into the initial population.
3. Removed the requirement that reward-based selection be implemented and tested.

1.2 (3-8-2017):

1. Removed references to four selection methods since there are only three required.

2 Introduction

You and your assigned group members will implement a genetic algorithm (GA). This GA will optimise a “warrior” in the Core Wars system.

Evolving corewar warriors is not a new idea: a list of projects that evolve warriors can be found here: http://corewar.co.uk/evolving.htm. You may alter and use code you find online; however, you must **document** the source in your paper **and** program source code. Be careful that you do not waste time trying to understand and modify existing code when writing your own would be faster.

There are numerous great tutorials for Redcode and Core Wars. Remember that you are not trying to deign a Core Wars warrior. You are trying to design a program to design a Core Wars warrior. You will mostly need to understand Redcode well enough to design a good encoding for your GA and to understand why the solutions it finds work. You will not need to understand Redcode so well that you can design the best warriors yourself.

Code execution and tracking of warrior threads are all handled by the provided pMARS program. All you have to do is write the GAs.

You will have to specify:

1. The genome encoding.
2. A mutation operator.
3. Various crossover operators.
4. Various selection operators.

5. An island model.

2.1 Goal

The goal of your warrior programs is to still be executing by the end of the run. If more than one warrior is executing at the end of the allotted time then those warriors tie. Warriors can spawn new execution threads with the split command. A warrior's thread is killed when it tries to execute data as though it were an instruction. When a warrior has no threads executing it dies.

2.2 Warriors

Corewar warriors are assembly language programs. The assembly language we will use is called Redcode. Redcode defines the instruction set:

Each of the following instructions takes 0, 1 or 2 arguments. These arguments are addresses.

1. DAT – data (kills the process)
2. MOV – move (copies data from one address to another)
3. ADD – add (adds one number to another)
4. SUB – subtract (subtracts one number from another)
5. MUL – multiply (multiplies one number with another)
6. DIV – divide (divides one number with another)
7. MOD – modulus (divides one number with another and gives the remainder)
8. JMP – jump (continues execution from another address)
9. JMZ – jump if zero (tests a number and jumps to an address if it's 0)
10. JMN – jump if not zero (tests a number and jumps if it isn't 0)
11. DJN – decrement and jump if not zero (decrements a number by one, and jumps unless the result is 0)
12. SPL – split (starts an execution thread at another address)
13. CMP – compare (same as SEQ)
14. SEQ – skip if equal (compares two instructions, and skips the next instruction if they are equal)
15. SNE – skip if not equal (compares two instructions, and skips the next instruction if they aren't equal)

16. SLT – skip if lower than (compares two values, and skips the next instruction if the first is lower than the second)
17. LDP – load from p-space (loads a number from private storage space)
18. STP – save to p-space (saves a number to private storage space)
19. NOP – no operation (does nothing)

Address modes:

1. # - immediate
2. $ - direct
3. @ - indirect
4. < - indirect with predecrement
5. * - indirect using A-field
6. { - predrecement indirect using A-field
7. } - postincrement indirect using A-field

2.3 Formatting the Header Comment

Redcode programs are formatted as follows. The comments are required:

```
;redcode
;name: Warrior Name
;author: Your Group ID
;assert    CORESIZE == 8000 && MAXLENGTH >= 100
add #4, 3           ; execution begins here
mov 2, @2
jmp -2
dat #0, #0

end ; execution ends here
```

2.4 The Core

The core consists of an address space that stores instructions. Each instruction is executed in order. We will use the default coresize of 8000 memory addresses.

2.5 Running Your Programs

Run and debug your Redcode programs using the Portable Memory Array Redcode Simulator (pMARS). You can download pMARS from the course website. To run programs in files A, B, and C call “pmars A B C”. To debug programs run pmars -e A.

2.6 Benchmarking and Fitness

Use the Wilkies Benchmark on the course website to evaluate your warriors. Run your warriors against the benchmark and use the resulting score as their fitness, this will give you a fixed point of reference for creating the figures for your paper.

You should consider evolving your warriors against each other as well, since the the warriors your classmates produce will not look like those in the benchmark.

2.7 Code Submission

Submit your GA as a github repository. Using machines with an Intel Xeon E3-1280 (4x3.6GHz) processor with 8 GB RAM, I will compile and run your GA over spring break to produce two warriors. Evolution will be limited to 24 hours wall time. Those two warriors will be your entrants into the tournament. The warriors must be evolved by your GA given the assembly commands listed in section 2.2. The largest unit of code that can be hard-coded in your GA is a single instruction with its arguments. You may not hard-code multiple instructions together (this is to prevent including warriors that are known to do well as part of the genome). This applies to the initial population for your GA. The genomes for the initial population may not have more than one instruction and its arguments hard-coded. This is to prevent the initial population from being seeded with known good warriors. Good warriors may be generated in the initial population by chance and you may seed your population with instructions taken from warriors that are known to be good.

2.8 Tournament

The two warriors produced by your GA will be entered into a tournament to be held in class. There are 16 teams so the tournament will progress as follows:

Entrants will be randomly assigned to head to head elimination stages. Each pair of entrants are run for 1001 rounds. I will make sure entrants from the same group do not compete against each other during stages 1 and 2. Half the warriors will be eliminated in each stage until there is a single winner.

There will also be a bonus free-for-all round. The last surviving warrior is the winner of this round. We will use the default pMARS scoring formula.

In the event of a tie in any round, the shortest program wins. If tied programs are the same length, a coin toss decides the winner.

2.8.1 Scoring

The following percentages will be added to your project score depending on how your evolved warriors perform:

1. Winner: 20%.
2. Runner up: 15%.

3. Round 2 Winners: 10%.
4. Round 1 Winners: 5%.
5. Bonus round: 10%.

The points earned are added to your final score for this project. No final project's score can exceed 100%. These bonuses do not carry beyond project 2. Percentages are not cumulative. For example, if team members receive a score of 80% on the paper and coding portions of the project and have 2 entrants that survive round 2 their final project score would be 90%. A group that wins both the tournament and the free-for-all with a base score of 80% would have a final project score of 100%. These percentages are added to all group members' project grades equally.

3 The Paper

3.1 Encoding

Describe the genetic encoding you used and why. If you experimented with multiple encodings, describe how you selected which one to use.

3.2 Comparison of Selection Methods

1. Generate a figure that compare the rate of convergence for your GA using *roulette*, *tournament*, and by random replacement of the lowest fitness half of the population with members of the upper half.

2. Generate an interquartile box-plot over many samples showing the quality of your solutions for these 3 types of selection. Provide p-values using the Mann-Whitney (also called rank-sum) statistical test to show whether the distribution of solution qualities is statistically different from one another.

3. Report whether elitism improves performance for each of the 3 selection methods. Verify your claim using the Mann-Whitney test.

3.3 Comparison of Crossover Methods

1. Generate a figure that compares the rate of convergence for your GA using *no crossover*, *uniform crossover*, and *1-point crossover*.

2. Generate an interquartile box-plot over many samples showing the quality of your solutions for these 3 types of crossover. Provide p-values using the Mann-Whitney (also called rank-sum) statistical test to show whether the distribution of solution qualities is statistically different from one another.

3.4 Mutation Rate and Crossover Rates

Define a mutation operator. Produce figures showing the effects of changing the mutation rate and crossover rate on your genetic algorithm.

3.5 GA Termination

Describe your GA termination criteria.

3.6 Island GA

Choose your best mutation rate, crossover method and rate, and selection method from your previous sections and use them with an island GA. Describe how the island GA compares to your previous version.

3.7 Fitness landscape

Describe the fitness landscape defined by the Core Wars problem. Is it simple, complex? What is the dimensionality of the space your GA had to search?

3.8 GA Selected Warrior

Provide a printout of two warriors selected by your GA, include the warrior with the highest fitness and one other. Describe how your warriors function and what makes your warriors successful. Are you able to describe the solutions in terms of schemas (i.e. what matters and what doesn't in the genome)?

4 Author Contributions

Include a contributions statement before the introduction section. The contributions may fall into three categories: analysis, code, and writing. For example your author contribution statement might look like this:

\section*{Author contributions}
J. C. wrote the code that generated Figs. 1, 3, and 5. V. W. wrote the code that generated Fig. 4. Both authors wrote the code that generated Figs. 6 and 7. J.C. wrote sections 1 though subsection 2.3, and section 3.6 of the paper. V. W. wrote subsection 2.4 through 3.5. The authors wrote sections 4 and 5 together. J. C. performed stability analysis for the map and V.W. identified the fixed points for the flow.

5 Notes

Format your paper as described in the project section of the web syllabus. Use the ACM paper template provided. Organise the paper into the following sections:

1. Abstract
2. Introduction

3. Related Work
4. Methods
5. Results
6. Conclusions
7. References

The paper may not exceed 5 single spaced pages including references.