

Evolutionary Programming to Optimize an Assembly Program

Brian Blaha and Don Wunsch
Applied Computational Intelligence Laboratory
Department of Computer Engineering
University of Missouri-Rolla, Rolla, MO, 65401, U.S.A.
Email: bblaha@umr.edu
<http://www.ece.umr.edu/acil/>

Abstract - Evolutionary programming was used to attempt to optimize a program written in the pseudo-assembly language Redcode, invented by A. K. Dewdney. Corewars is the game under which Redcode programs compete. Since 1994, the last standardization of Redcode, many complicated, effective Redcode programs have been written by people, but intense study is required to learn the nuances of the language and perfect programs. Since this is such a difficult task, evolutionary techniques may outperform humans. Multiple point, variable length cross-over and change, insert, and delete mutations were the operators used. Relative fitnesses were calculated within a subset of the population on remote client computers. A food model was used to select the most fit programs. Current results are preliminary, but already one of the resulting programs wins 38% and ties 29% against a common type of human-written program. The best performance is 151 wins, 49 losses, and 0 ties against a typical human program.

I. EXPLANATION OF THE PROBLEM

Corewars is a game invented by A. K. Dewdney in which two or more programs, written in the Redcode pseudo-assembly language, attempt to crash all other programs [1]. The programs reside in a common, circular memory core, at pseudo-random locations unknown to them. All data must be encoded in the instructions, and all instructions can be overwritten by any program. Therefore, self-modifying programs are the rule, not the exception.

Redcode is a programming language which is abstract, but also resembles assembly languages. A complete set of rules may be found at [2]. The most important and interesting instructions are JMP, MOV.I, DAT, and SPL. JMP simply moves the point of execution, as an unconditional branch. MOV.I copies an instruction at its first argument's location and overwrites the instruction at its second argument's location. This is the only way to change the opcode of an instruction in memory, and is frequently used to attempt to overwrite opponents' programs. DAT is an instruction which kills the currently executing thread. This instruction is intended to represent data space, which in a normal computer program, cannot be executed. Many Redcode programs use MOV.I to place many copies of a DAT instruction the execution path of other programs. SPL performs an instruction usually left up to the operating system; it creates a new thread for the executing program. The benefit is that a program does not lose unless all of its threads have died, so a program with more threads is more difficult to defeat. The downside is that programs share

time evenly on a single CPU, regardless of the number of threads in each, so each thread in a many-threaded program receives a small share of time. This can be used offensively, by "capturing" an opponent's thread by MOV.I'ing a JMP instruction into the code path, then forcing it to SPL endlessly, slowing down the opponent's remaining uncaptured threads.

Most Redcode programs use one or two of five major strategies. Bombing is the strategy of attempting to MOV.I DAT instructions over opponents' code. Scanners attempt to find their opponents quickly, so they can attack first. A "vampire" attempts to capture opposing threads by using MOV.I with JMP. A replicator copies itself all over memory, with a thread running each copy, making itself difficult to defeat. A so-called "imp spiral" uses several cooperating threads to swarm over opposing programs [3]. It has been conjectured that Corewars is a rock-paper-scissors game, in which the replicator is paper, the scanner and the vampire are scissors, and the bomber is rock (often called stone). The imp spiral is often thrown in with one of the other strategies, so that if the main program is overwritten, the code for the imp spiral is still executing and has a chance of defeating the opponents [4]. Corewars is a very complicated game, and one in which people are still inventing new strategies, more than a decade after it was first standardized. Therefore, evolutionary strategies have ample chance to outperform humans.

One similar work to this is David Fogel's evolution of a checkers-playing neural network [5]. His evolutionary algorithm had to find the best position-evaluating network, given just the knowledge of how well the network's evaluations win entire games. The difficulty of successfully modifying individual weights in the network is similar to the difficulty of successfully modifying individual instructions in a Redcode program.

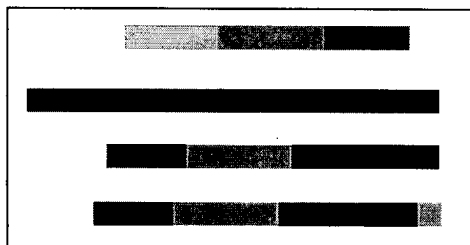
II. EXPLANATION OF EVOLUTIONARY PROGRAMMING TECHNIQUE

In Corewars, an absolute fitness is meaningless. Programs can only be measured relative to other programs. Considering both runtime and accuracy, fitness for the evolutionary programming is calculated by conducting round-robin competitions among random groups of three to six, with two competing at a time. Rewards are set such that a win and a loss together are significantly more valuable than two ties, as is the case for the Corewars game. Calculation of fitness requires interpreting and executing instructions from each

program for up to 80000 cycles, at which point a tie is declared. This threshold cannot be lowered too much, because some human-written programs, such as the vampire, actually use a large portion of this total. Therefore, the main program delegates fitness evaluation to other computers. The evaluation of a group of four (six pairings) takes about two seconds on a typical computer. This is, by far, the most expensive part of a generation.

After about two-fifths of the population competes in this way, the programs that had competed are selected, and each is assigned a share of "food" that increases exponentially based on its fitness. Each program receives its share of the total food available, which oscillates to prevent the population from stagnating. An average warrior in an average-size population would receive roughly 1.1 food. The cost to live one more generation is one food. The cost to breed with another program, producing a child, is one-half food for each. Any left-over food can be stored indefinitely. As a result, children start with one food. Any program with less than zero food has a chance to survive that decays exponentially as its food decreases. In addition, especially old programs require increasing amounts of food to live an additional generation. Therefore, no elitism exists. To preserve success, programs are given many opportunities to breed and pass on their successful code. Furthermore, successful Redcode programs must be tolerant of small changes to their code, of the sort that would be introduced by mutations.

Programs with enough food to breed are paired randomly, with no genders. Breeding is then conducted by selecting several crossover points in each parent and reassembling the segments in roughly the order they were in. The child can be shorter or longer than both parents, or, more likely, it may have an intermediate length. Currently, zero to four small mutations are applied to the child. Small mutations include: change one instruction, insert one instruction, and delete one instruction. No large mutations are applied. Additionally, to enforce population diversity, inbreeding is forbidden. Each program keeps a record of its parents and grandparents. Potential mates may not appear in each other's record, and cannot share any entries. Unfortunately, forbidding inbreeding also has the consequence of causing populations with less than roughly 25 members to go extinct. Additionally, a population with an



The parents, at top, have crossover points selected at the changes in color. The third from top is a possible child, before mutation. The bottommost is a possible child after mutation, specifically a delete in orange, an insert in the middle of teal, and a change at the end of teal.

average size of 70 will drop to that threshold roughly every 30000 generations. The ultimate goal is to produce a program that outperforms the best existing

human-written programs.

III. PRELIMINARY RESULTS

A batch file and set of human-written programs were used to determine the quality of the results [6]. The set of programs are fairly representative of the field, including four each of paper, stone, and scissors.

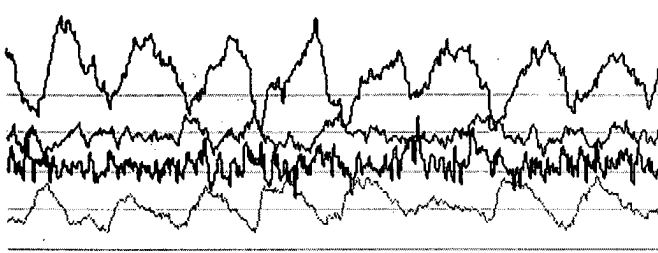
Program ID	vs. Paper	vs. Stone	vs. Scissors
17535	0-787-13	0-799-1	4-792-4
39676	0-783-17	0-800-0	0-799-1
54398	1-768-31	1-796-3	2-796-2
145430	0-535-265	58-616-126	8-787-5
268837	0-672-128	164-328-308	73-632-95
427492	0-772-28	222-577-1	133-658-9
522618	5-761-34	301-270-229	100-661-39

The opponent type is listed across the top. The program ID, which is assigned sequentially to new children, is on the left. The performance of each program against each opponent type is listed as wins-losses-ties. Oddly, program 522618 loses to paper, which indicates it is a stone, defeats stone, which indicates it is a paper or stone, and loses somewhat to scissors, which indicates it is a paper or scissors, contradicting the rock-paper-scissors model.

As is evident from the large program IDs, the population advances very slowly. Generations occurred at a rate of roughly eight per minute. The slow advance can be partially explained by the large portion of awful offspring. Children are frequently recombined from the parents in such a way that they have or place DAT instructions in their own code paths, thus losing nearly all of the time. Even if that does not happen, previously successful code is frequently altered in such a way that it becomes useless. The population loses some good code if a successful program fails to produce a successful child. This may explain the large decrease in performance against paper between 145430 and 427492. Although successful code must endure unexpected changes, some of the changes introduced in crossover are too large for the code to endure. Crossover must be responsible for the awful offspring because small mutations have only a low probability of badly damaging a program. On the other hand, it allows rapid advance from a random starting population. As the population advances, the frequency of crossover should be reduced to allow convergence.

IV. FUTURE WORK

Currently, population members are picked at random, sent to a client computer, and compete. Therefore, entirely new programs must be transmitted at each competition. The bandwidth required to do this is on the order of 3 kilobytes per sec-



The blue line, at top, is the population size over time, with the bottom of the graph at 0 and light grey lines every 20 until 80. Next, the red line is the average age, the black line is the average fitness, and the green line is the average amount of stored food, with light grey lines every 1 until 4. The population size, average age, and average stored food oscillate, as intended. The average fitness is near 2, which indicates a high frequency of wins and losses, and very infrequent ties.

ond per typical client. This could be reduced substantially by caching the most recent programs on the client and having the server tend to pick cached programs for competitions. This is the island model of multiprocessing [7].

Similarly, the population could be split into sub-populations, or tribes, to encourage diversity, and, in conjunction with the island model, reduce bandwidth usage further [8]. To decrease the risk of extinction while still keeping pressure on the population, a neural network could be used to control the amount of food available better than a simple periodic formula. Occasionally comparing the best programs to a set of human-written programs may provide useful information to the evolutionary algorithm, so it can apply more pressure if the population is not advancing, or write to disk as separate files programs that perform particularly well. Most importantly, the breeding process must be improved. Some ways to do this include competing potential offspring, strongly altering the crossover operation to keep useful segments intact, and using a neural network to attempt to predict advantageous crossover points.

Corewars has a provision for programs to cooperate in and compete as teams, but it is fairly difficult for people to use effectively. This could be an even better opportunity for evolved programs to shine.

V. RELATED WORK

Others have attempted to evolve Redcode programs with various strategies. In large tournaments, having a few evolved programs is not uncommon. However, as far as we know, no evolved program has greatly outperformed the field. Reference [9] is an incomplete description of two attempts to evolve programs, based on genetic algorithms, using existing human-written warriors to aid the evolution. The results are interesting, and were an inspiration, but the resultant programs, by the author's admission, are not and, with more generations, likely would not become effective.

References

- [1] A. K. Dewdney, *Computer Recreations*. Scientific American, 1984, p. 14-22. <http://www.koth.org/info/sciam/> and <http://www.koth.org/info/akdewdney/index.html>.
- [2] Ilmari Karonen, "Beginner's Guide to Corewars," 1998. <http://www.sci.fi/~iltzu/corewar/guide.html>.
- [3] Greg Lindahl, "Modern Core Wars," 1994. http://www.koth.org/info/greg_lindahl_corewars.html.
- [4] Stephen Morrell, "My First Corewar Book." <http://www.koth.org/info/chapter1.html>.
- [5] David B. Fogel, "Blondie24: Playing at the Edge of AI," Morgan Kaufmann Publishers, 2002.
- [6] John Wilkinson, "JKW's Beginner's Benchmark." <http://www.koth.org/wilkies/>.
- [7] Xin Yao, *Global Optimisation by Evolutionary Algorithms*. Proceedings of the 2nd AIZU International Symposium on Parallel Algorithms / Architecture, 1997, p. 282-291.
- [8] Cara MacNish, *Evolutionary Programming Techniques for Testing Students' Code*. Proceedings of the conference on Australasian computing education, 2000, p. 170-173.
- [9] John Perry, *Core Wars Genetics: The Evolution of Predation*. http://www.KOTH.org/info/evolving_warriors.html