# An Evolutionary Approach Generates Human Competitive Corewar Programs

Barkley Vowk[1], Alexander (Sasha) Wait[2], Christian Schmidt[3]

[1]*University of Alberta, Department of Mathematical and Statistical Sciences, Edmonton, AB T6G2E1, Canada*
[2]*Harvard Medical School, Department of Genetics, 77 Avenue Louis Pasteur, Boston, MA 02115, USA*
[3]*Glasower Damm 4R, 15831 Mahlow, Germany*
fizmo@corewar.info

## Abstract

A framework for automatic generation of human competitive Corewar programs is presented. The authors believe this approach is applicable to other Corewar-like artificial chemistries.

## Introduction

In 1984 the first Corewar article appeared in the pages of Scientific American [Dewdney, 1984]. That article described a new game in which human players write computer programs to take over a virtual machine. The Corewar game has spawned many important artificial life systems: The Coreworld [Rasmussen et al. 1990], Tierra [Ray, 1992], and Avida [Adami and Brown, 1994] are some of the best known. Corewar itself is also still popular [Philaja, 2004; Birk, 2004].

Many people have noticed a similarity between Corewar programs and biological viruses. Natural viruses, however, have had over 3 billion years—during which an estimated $10^{25}$ phage infections occurred every second—to explore genomic space [Pedulla et al, 2003; Hendrix et al, 1999]. We are a long way from simulating the awesome power of the biosphere.

The Corewar language is extremely "brittle"—nearly any change to a succesful program is fatal—so it was not possible to produce human competive programs in the past. It is also difficult to engineer a new virus by randomly changing a few base-pairs. In this work we have decided not to change the rules of Corewar to make it easier to evolve these programs but instead have decided to apply better ideas and more hardware.

## The Chemistry

The current Redcode standard, DRAFT94, was published in REC.GAMES.COREWAR and implemented in the pMARS software [Ma et al, 1995]. Each memory cell of the "core" consists of an instruction [Table 1], modifier [Table 2], and two operands whose interpretation is determined by their addressing mode [Table 3]. For more details about Redcode please consult one of the many introductory documents. Eg. The Beginners Guide to Redcode [Karonen, 2004] More information can also be found in the pMARS documentation.

| | |
|---|---|
| **DAT** | terminate process |
| **MOV** | move from A to B |
| **ADD** | add A to B, result in B |
| **SUB** | subtract A from B, result in B |
| **MUL** | multiply A by B, result in B |
| **DIV** | divide B by A, result in B if A is not zero, else DAT |
| **MOD** | divide B by A, remainder in B if A is not zero, else DAT |
| **JMP** | execute at A |
| **JMZ** | execute at A if B is zero |
| **JMN** | execute at A if B is not zero |
| **DJN** | decrement B, if B is not zero, execute at A |
| **SLT** | skip if A is less than B |
| **SEQ** **CMP** | skip if A is equal to B |
| **SNE** | skip if A is not equal to B |
| **NOP** | no operation |
| **SPL** | new task at A |

**Table 1. Valid instructions.** Note: Instruction behavior can depend on execution style: biotic (2004) or abiotic (1994).

| | |
|---|---|
| **.A** | Instr. read and write A-fields |
| **.B** | Instr. read and write B-fields |
| **.AB** | Instr. read A-field of A-instr. and B-field of B-instr. and write B-field |
| **.BA** | Instr. read B-field of A-instr. and A-field of B-instr. and write A-field |
| **.F** | Instr. read both A&B fields of A&B instr. and write to both A&B fields (Ato A and B to B). |
| **.X** | Instr. read both A&B fields of A&B instr. and write to both A&B fields (Ato B and B to A). |
| **.I** | Instr. read and write Instr., Modifier, Modes, A & B fields |

**Table 2. Valid modifiers.**

| | |
|---|---|
| **#** | *immediate* |
| **$** | direct |
| **@** | indirect using B-field |
| **<** | predecrement indirect using B-field |
| **>** | postincrement indirect using B-field |
| ***** | indirect using A-field |
| **{** | predecrement indirect using A-field |
| **}** | postincrement indirect using A-field |

**Table 3. Valid addressing modes.**

## Seeding the Pool

Each round of the evolution starts out with *seeding the pool*, this is done by running the "makepop" tool. Makepop creates a new individual from randomly selected Redcode instructions (the individual's DNA) and then checks to see if it can pass a test battle against a random individual from a benchmark set.

The code generated by makepop is seeded from patterns seen in human coded individuals. The theory here is to emulate natural organisms. In nature there are only a limited number of working combinations of proteins, and the un-workable proteins are suppressed through various means.

The same is true in Corewar, wherein we want to maximize the working combinations we see in "nature" (human coded individuals), and suppress the random ones that we don't see. To do this, a system is devised which is based on instruction distribution tables—called chains in the following discussion—which are calculated from successful human coded individuals. A chain might look like:

```
Start instruction:
   SEQ occurred    69 times (39.884%)
   MOV occurred    49 times (28.324%)
   SNE occurred    23 times (13.295%)
   SPL occurred    17 times (9.827%)
   ADD occurred     7 times (4.046%)
   SUB occurred     3 times (1.734%)
   JMZ occurred     3 times (1.734%)
   DIV occurred     1 times (0.578%)
   NOP occurred     1 times (0.578%)

2nd Instruction:
   If previous instruction was spl:
      SEQ occurred   771 times (60.613%)
      SNE occurred   170 times (13.365%)
      DAT occurred   137 times (10.770%)
      SPL occurred    69 times (5.425%)
      JMP occurred    36 times (2.830%)
      MOV occurred    34 times (2.673%)
      ADD occurred    16 times (1.258%)
      DJN occurred     6 times (0.472%)
      SUB occurred     3 times (0.236%)
      JMZ occurred     1 times (0.079%)

    If previous instruction was djn:
      DAT occurred   128 times (22.980%)
      SEQ occurred   127 times (22.801%)
      JMP occurred   100 times (17.953%)
      MOV occurred    50 times (8.977%)
      SNE occurred    46 times (8.259%)
      SPL occurred    38 times (6.822%)
      DJN occurred    21 times (3.770%)
      ADD occurred    11 times (1.975%)
      JMN occurred    10 times (1.795%)
      SUB occurred     6 times (1.077%)
      MUL occurred     2 times (0.359%)
      JMZ occurred     2 times (0.359%)
```

patterns are examined up to 5 instructions deep, which means the fifth instruction will look like:

```
 If 4 ins ago was spl:
   If 3 ins ago was spl:
     If 2 ins ago was spl:
       If 1 ins ago was spl:
         ADD occurred     1 times (50.000%)
         MOV occurred     1 times (50.000%)
```

So the fifth instruction would be ADD or a MOV.

In addition to the instruction itself, valid Redcode has a modifier and operands. These are tied only to the current instruction to keep look up tables reasonably sized.

When creating the tables for selecting random instructions, a group of successful individuals (obtained from Koenigstuhl's infinite LP hill) is selected and converted to the format that exhaust requires. Then a program called statcode.pl is run. It counts every instruction, tallies all the start instructions it sees, makes a note of which instructions follow which instructions, and tallies which modifiers and operands are seen with any given instruction. The start instructions are tallied separately to ensure that all individuals start with a valid instruction. The program then outputs a C source file, which is compiled and linked into makepop and client programs.

The source file defines several calls:

```
get_start_instruction
get_operands
get_imod
get_omod
get_{second,third,fourth,fifth}_instruction
```

get_start_instruction returns an instruction selected randomly from the list of valid start instructions, weighted by the number of occurrences. So in the above example, one would expect nearly 40% of our start instructions to be SEQ, and nearly 10% to be SPL.

The following is a source snippet from makepop.c showing how an individual is filled using these chains (c is a pointer to the code section of the individual):

```
getstartins(&(c[0]));
op = c[0].in;
getsecondins(&(c[1]),op);
op2 = c[1].in;
getthirdins(&(c[2]),op,op2);
op3 = c[2].in;
getfourthins(&(c[3]),op,op2,op3);
op4 = c[3].in;
getfifthins(&(c[4]),op,op2,op3,op4);
for(i = 5; i < max_length; i++) {
   getfifthins(&c[i],
             c[i - 1].in,
             c[i - 2].in,
             c[i - 3].in,
             c[i - 4].in);
}
```

A second pass would then be made to match each of the instructions with operands and modifiers.

This method has made a significant difference in the time required to generate a viable start population. The instruction calls are also used to suggest mutations, reducing the chance of destructive mutations by using possibilities that fit chains used to generate the individual. Allowing for a small chance that we will change some part of the instruction based on the chains that match the individuals code.

Once the pool is filled with individuals, the server program master starts. Master reads all the generated individuals, the benchmark individuals, then shuffles the generated individuals into species. The evolver considers a species to be a group of individuals in the pool that breed only with other members of their group. The benchmark typically sits around 6-12 individuals. When the master is on-line and listening, clients starts on any available computer. The clients contact the master, and checks out a section of the pool for which to calculate scores. Once the client finishes calculating the initial individual scores, it returns the information to the server and checks out another small section.

Once all the individuals have an initial score assigned, the server begins sending the clients the completed pool. The clients breed new individuals. Any new individuals that exceed the score of the current flimsiest individual in a species are copied in over that individual, removing it from the pool.

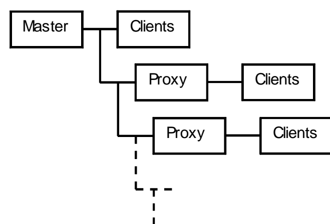
## The Evolutionary Process

The "breeding" process selects two individuals of the same species, one of the parents is chosen to be the source. After each instruction is copied to the child, there is a chance the source will be swapped to a different parent individual, and instructions will be copied from that parent instead. On average there are four cross-overs between parents in each child. As well, there is a chance of mutation for every instruction, this can take the form of either a small change to the current instruction, dropping this instruction completely, or adding a new instruction. Once the child has been filled with code, it is validated against a random benchmark individual to ensure it is worth the cycles to calculate a full score. If it

passes, it will continue as above, if the child scores better than the current flimsiest, it replaces it in the pool.

After the client has found enough improved individuals, it returns its pool to the server, which adds the improved individuals to the main pool. This will continue until someone stops the master, and the clients exit when they can check-out no more work.

There is also a simple proxy server, to allow clients to be run on machines that don't have direct access to the Internet. The proxy checks out work (from either the master, or another proxy), and hands it out to clients that connect to it. Once the proxy receives enough improved individuals, it checks them back into the main server, and requests a new pool.




It was found that best results are obtained by starting the validation score very low, and ramping it up as the baseline score of the individuals improves. When most individuals are capable of scoring >50% of the tie barrier (the score achieved with 100% ties), the validation score is bumped so only individuals that can exceed the tie barrier will have their score calculated. This prevents individuals that have no strong offensive ability from breeding out all the very offensive individuals from the pool, as offensive seem to have a more difficult time finding improvements. The mutation rate is started quite high, about 60% chance per instruction, and is dropped quickly as the individuals improve. Towards the end, the mutation rate is typically held somewhere around 2% chance per instruction. If the mutation rate is not dropped, the mutation rate will cause the pool to flail around aimlessly and never converge. If the mutation rate is started too low, there will not be enough variation in the pool, and it is unlikely to find optimizations. The typical path of the evolver is to start with low scores, quickly increase as we approach the tie barrier, then there is a significant slowdown after the validation score bump described above. Once most of the population starts to shift towards more offensive individuals, the climb upwards starts again. This will continue until the pool converges on a monoculture, where all individuals effectively use the same tactics, and we no longer find further improvements. At this point, the pool can be considered done.

The evolver generally runs several times from a random start until the pool converges on a monoculture, then collect the top individuals from the finished pools into a new pool, and re-run the evolver hoping for further improvements. This strategy has proved to be quite successful in the past. Typically it will take days or weeks to run each pool to completion using the spare CPU cycles of a large cluster of commodity PCs.

## The Evolutionary Environment

As mentioned earlier all individuals are tested against a benchmark. The specific individuals in the benchmark are essential to the evolution of human competitive programs. It usually contains a set of the best scoring individuals from Koenigstuhl’s infinite hill.

If the number of individuals in the benchmark is too low the evolution quickly reaches a dead-end, because of the appearance of individuals which overpower a single individual of the benchmark. Such individuals, called in the following “kryptonites”, inhibit further evolution of the main pool because of their success against the vulnerable individual in the benchmark. But in all cases they are not viable in tournament play.

The effect is less pronounced as more individuals are used in the benchmark. The downside is that the evolution then needs more computing power and the time until the pool converges on a monoculture significantly increases. One can slightly suppress the kryptonite effect by the careful choice of different species and sub-species for an even benchmark.

There are still further approaches in progress how to break the deadlock provoked by cryptonides while using a benchmark with a small amount of individuals.

## Evolution and a Step Beyond

An example of the evolutionary process described in this article is the individual "189602-1978-xt642-2-eve15". This program was evolved using optiMAX's LP hill benchmark. It showed a very interesting scoring pattern on SAL's LP hill. Although it is a very defensive species it's able to beat some other defensive species (namely replicators) quite nicely.

A closer look at the code showed the individual executes just 8 instructions. The remaining 192 lines are not in use and seems to act as a huge decoy. The 'active' part of the individual, with some additional comments, is shown below:

```
mov.i  $     7 , {     2  ;boot the imp
spl.a  $     5 , # 1867  ;split behind impgate
spl.ab $  2317 , $   931  ;launch booted imp
djn.a  #    -1 , <  -145  ;imp-gate / djn-train
djn.f  #    -1 , <  -145  ;
jmp.ba #    21 , <   -43  ; ->safety instr.?
mov.b  #  2667 , $     8  ;useless instruction
mov.i  #    -1 , $     1  ;the imp instruction
```

As unique the scoring pattern is as interesting is the evolved species, because it contains just two imps and an imp-gate. On a deeper view it unfurls a quite barbaric brilliancy: A strategy which was never seen in a human coded individual before. Both imps are running fast enough to being not caught by a coreclear or a scanner wipe on their way through the core. And if they overwrites an opponent they convert it into an imp. And they are even fast enough to catch the papers while still in process of replicating themselves. At the end of all this waits the imp-gate terminating every imp that approaches. This means if the two imps were able to convert all processes of the paper into imps while running through the core the paper will lose.

The next step after understanding the species was to write a human coded version. For some additional points a quick scanner was added which works also as a decoy; the code is at right:

```
;redcode-lp
;name Eve 15
;strategy No humans were used in the creation
;strategy of this strategy
;author bvowk + Fizmo
;assert 1

;------>qscan constants
zero    equ     qbomb
qtab3   equ     qbomb
qc2     equ     ((1+(qtab3-qptr)*qy) %
                 CORESIZE)
qb1     equ     ((1+(qtab2-1-qptr)*qy) %
                 CORESIZE)
qb2     equ     ((1+(qtab2-qptr)*qy) %
                 CORESIZE)
qb3     equ     ((1+(qtab2+1-qptr)*qy) %
                 CORESIZE)
qa1     equ     ((1+(qtab1-1-qptr)*qy) %
                 CORESIZE)
qa2     equ     ((1+(qtab1-qptr)*qy) %
                 CORESIZE)
qz      equ     2108
qy      equ     243

;------>qbomb constants
qoff    equ     -87
qstep   equ     -7
qtime   equ     14

;------>eve 15 constants
iAwa    equ     4174
train   equ     7903

;------>eve 15 code
pGo     mov.i   imp,            *2
        spl     imp
        spl     iAwa
        djn.a   #0,             <train
        djn.f   #0,             <train
        jmp.ba  #0,             <train
imp     mov.i   #-1,            1


        for     51
        dat     0,              0
        rof

;------>qscan code
qbomb   dat     >qoff,          >qc2
        dat     0,              0
        dat     0,              <qb1
qtab2   dat     0,              <qb2
        dat     0,              <qb3
        for     16
        dat     0,              0
        rof
        dat     zero - 1,       qa1
qtab1   dat     zero - 1,       qa2
        for     42
        dat     0,              0
        rof

qgo sne   qptr+qz*qc2, qptr+qz*qc2+qb2
    seq   <qtab3,  qptr+qz*(qc2-1)+qb2
    jmp   q0, }q0

    sne   qptr+qz*qa2, qptr+qz*qa2+qb2
    seq   <qtab1, qptr+qz*(qa2-1)+qb2
    jmp   q0, {q0

    sne   qptr+qz*qa1, qptr+qz*qa1+qb2
    seq   <(qtab1-1), qptr+qz*(qa1-1)+qb2
    djn.a q0, {q0
    sne   qptr+qz*qb3, qptr+qz*qb3+qb3
```

```
    seq    <(qtab2+1), qptr+qz*(qb3-1)+(qb3-1)
    jmp    q0, }q1
    sne    qptr+qz*qb1, qptr+qz*qb1+qb1
    seq    <(qtab2-1), qptr+qz*(qb1-1)+(qb1-1)
    jmp    q0, {q1

    sne    qptr+qz*qb2, qptr+qz*qb2+qb2
    seq    <qtab2, qptr+qz*(qb2-1)+(qb2-1)
    jmp    q0

    seq    >qptr, qptr+qz+(qb2-1)
    jmp    q2, <qptr

    seq    qptr+(qz+1)*(qc2-1),
           qptr+(qz+1)*(qc2-1)+(qb2-1)
    jmp    q0, }q0

    seq    qptr+(qz+1)*(qa2-1),
           qptr+(qz+1)*(qa2-1)+(qb2-1)
    jmp    q0, {q0

    seq    qptr+(qz+1)*(qa1-1),
           qptr+(qz+1)*(qa1-1)+(qb2-1)
    djn.a q0, {q0
    jmz.f pGo, qptr+(qz+1)*(qb2-1)+(qb2-1)

;------>qbomb code

q0      mul.b    *2,             qptr
q2      sne      {qtab1,         @qptr
q1      add.b    qtab2,          qptr
        mov      qtab3,          @qptr
qptr    mov      qbomb,          }qz
        sub      #qstep,         qptr
        djn      -3,             #qtime
        jmp      pGo

        end qgo
```

## Conclusion

The “brittleness” of Corewar's Redcode has prevented significant evolution of competitive Redcode programs in the past. With advances in computer hardware and by seeding initial populations of evolving programs with patterns seen in human coded individuals, we can significantly decrease the time required to generate a viable start population. Also the use of a dynamic validation score and mutation rate during the "breeding" process assists the pool's convergence on a human competitive monoculture. Together with a well chosen benchmark it is now possible to efficiently evolve new and unique “species” that are effective in tournament play.

## Acknowledgments

We are very grateful to Joonas Pihlaja, John Metcalf and Will 'Varfar' for all their ideas. A special thanks to Joonas for repeatedly rewriting exhaust to suit our whims.

## References

Adami, C. and Brown, T. 1994. Evolutionary Learning in the 2D Artificial Life System 'Avida'. In Brooks, R. and Maes, P. editors, *Artificial Life IV,* 377-381. Cambridge, MA: MIT Press.

Birk, C.C. 2004. Koenigstuhl. On the web at: http://www.ociw.edu/~birk/COREWAR/koenigstuhl.html

Dewdney, A.K. 1984. In the game called core war hostile programs engage in a battle of bits. *Scientific American*, 250:14–22.

Hendrix, R; Smith, MCM; Burns, RN; Ford, ME; Hatfull, GF. 1999. Evolutionary relationships among diverse bacteriophages and prophages: All the world's a phage. PNAS, 96(5):2192-2197.

Karonen, I. 2004. Beginner's guide to Redcode. On the web: http://vyznev.net/corewar/guide.html

Ma, A.; Sieben, N.; Strack, S.; and Wangsaw, M. 1995. PMars. Software on the web at: http://www.koth.org/pmars

Pedulla, ML; Ford, ME; Houtz, JM; Karthikeyan, T; Wadsworth, C; Lewis, JA; Jacobs-Sera, D.; Falbo, J; Gross, J; Pannunzio, NR; Brucker, W; Kumar, V; Kandasamy, J; Keenan, L; Bardarov, S; Kriakov, J; Lawrence, JG; Jacobs, WR; Hendrix, RW; Hatfull, GF. 2003. Origins of Highly Mosaic Mycobacteriophage Genomes. Cell, 113(2):171-182.

Pihlaja, M.J. 2004. New KOTH server. Usenet post in REC.GAMES.COREWAR on the web at: http://groups.google.com/groups?th=1f82856630ed460d

Ray, T.S. 1992. An approach to the synthesis of life. In C. G. Langton, C. Taylor, J.D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, 371–408, Redwood City, CA, Addison-Wesley.

Rasmussen, S.; Knudsen, C.; Feldberg, R; and Hindsholm, M. 1990. The Coreworld: Emergence and evolution of cooperative structures in a computational chemistry. *Physica D*, 42:111–134.

Zapf, S.; Schmidt C. 2004 optiMAX. Software on the web at: http://www.corewar.info/optimax/