Ryan Coleman

Learning By Simulating Evolution Using Corewars

Abstract

The goal of this project is to prove that corewarriors can learn by simulating evolution. I will try to show that the corewarriors have learned survival out of evolution. This will be accomplished using a computer program that simulates evolution using genetic algorithms. This paper was first prepared and presented in an Artificial Intelligence class at Wilmington College in the Spring semester of 1998. The class was taught by Jim Fitzsimmons, Ph. D.

Background

Evolution

Evolution is a concept put forward by Charles Darwin in his The Origin of Species. After lots of debate, it has become generally accepted among scientists. (Winston, Patrick Henry, 507). In summary form, the driving principles of evolution are:

"-Each individual tends to pass on its traits to its offspring.
-Nevertheless, nature produces individuals with different traits.
-The fittest individuals--those with the most favorable traits--tend to have more offspring than do those with unfavorable traits, thus driving the population as a whole toward favorable traits.
-Over long periods, variation can accumulate, producing entirely new species whose traits make them especially suited to particular ecological niches." (Winston, Patrick Henry, 507)

These theories are very important in many fields of science today. The third theory is also known as the theory of natural selection, or survival of the fittest.
Today, in genetic algorithms, there are three methods used in genetics to change the code. (Winston, Patrick Henry, 512) They are mutation, reproduction, and crossover. Mutation, the simplest of all, involves changing a single piece of code, or inverting the same piece of code. Reproduction is rewarding good code with more replicas of itself in the set. Crossover involves cutting two pieces of code in half at some point, and then switching the tail end around. Many arguments have been made as to which of these holds the most importance, but, it is still being debated. (Peters, James A.) Also, many different methods of each kind have been developed, such as crossover where two points are picked, and only the code in between are switched.

Corewars

Corewars is a game created by A. K. Dewdney in a series of articles in Scientific American. (Dewdney, A.K., The Armchair Universe) The 'core' in corewar was an early name for memory. A core is a linear looping linear memory with each place in memory able to hold one complete instruction. By linear, I mean that instructions are referenced by a single number, no matrix or array is used. The memory is said to loop because after the last instruction in memory the first instruction can be executed. This could be pictured as a circle. It is also worth mentioning that there are no absolute references in corewars, everything is referenced in terms of the instruction executing. There are seventeen (or nineteen) instructions that can be used in corewars. By far, the most important instruction in corewar is the DAT instruction. No matter what the A and B fields contain, a DAT terminates the process if that process executes the DAT. Notice process is used and not program. A program in corewar can split into two or more processes by use of the SPL instruction. Two programs are loaded into random positions in the core where they battle it out. After a certain number of cycles, if neither program has quit executing, a tie is declared. If one program terminates, the other is declared the winner. Basically, if you can make all your opponent's processes execute a DAT, you win. This is where the 'war' in corewar comes in. A brief description of each instruction, modifier, and addressing mode is given in Appendix A of this paper. To fully understand corewars I suggest the Frequently Asked Questions list of rec.games.corewar. (Durham, Mark, et. al.)

Another aspect of corewars was the early creation of 'King of the Hill' tournaments. (Durham, Mark, et. al.) The name is based on the game played on a dirt pile, or hill, where everyone tries to run up and push the person at the top off. Each hill has a specific set of standards which people write warriors for. Entries are submitted via e-mail, and a script that will run on a web-browser is in the works. After the battles are run, you receive results back. Usually, there are 20 to 25 warriors on the hill that you must score better than to get on the hill and knock one of them off. Usually, 100 battles take place between your warrior and each of the other warriors. As far as standards go, the main hill now has a coresize of 8000, it goes 80000 cycles before a tie, has 8000 processes, a maximum of 100 instructions, awards 3 points for a win, 1 for a tie, and 0 for a loss. This hill also does fights on a one-on-one basis. There are other hills, such as a limited process hill, a multiwarrior hill (everyone fights at the same time), a beginner's hill where 'newbies' are invited to test their code, and a 'baby' hill. The baby hill is interesting because the number of instructions are limited to 20, the coresize is 800, and it only goes 8000 processes before a tie. This makes the baby hill run much faster than all its counterparts.

I should also make mention of the different types of warriors that have been created over the years. There are three basic types that follow the idea of the childhood game paper-rock-scissors. A paper is a replicator. It makes many copies of itself and usually has lots of processes. Papers are usually durable and usually have a small bombing routine at the end of their code to help kill the opponent. A rock or stone is a warrior that bombs the core with DATs or other bombs, hopefully landing a bomb on the opponent. A scissor is a warrior that scans the core for other warriors, and then bombs them. In general, a paper beats a stone, a stone beats a scissor, and a scissor beats a paper. It should also be noted hybrids and many other forms of these warriors exist. No one strategy dominates corewars today, rather a good warrior uses elements from all of them to win.

Corewars Lingo

This is a section introducing the reader to the jargon that has become common to corewars. First, off, as has already been mentioned, bombing. Bombing consists of using a MOV instruction to move one instruction over another. The instruction that gets moved is called the bomb. For example, let's look at a very simple bomber, written by A. K. Dewdney (Dewdney, A. K., The Armchair Universe):

```
mov.i   $3,     $7
add.ab  #4,     $-1
jmp.a   $-2,    #0
dat.f   #0,     #0
```

Execution starts at the MOV instruction. This instruction moves whatever is at the location 3 places ahead of it, in this case the DAT instruction, to 7 instructions ahead of it. Execution continues to the

ADD instruction which adds 4 to the B-field of the instruction before it, the MOV. This changes the 7 to an 11. The JMP instruction sends execution back to the MOV, which bombs again. Let's look at how two of these warriors would fight against each other and how a DAT bomb kills.

```
        mov.i   $3,     $7                      <-- execution for this warrior starts here
        add.ab  #4,     $-1
        jmp.a   $-2,    #0
        dat.f   #0,     #0
(empty line)
(empty line)
(empty line)
(empty line)
        mov.i   $3,     $7              <-- execution for this warrior starts here
        add.ab  #4,     $-1
        jmp.a   $-2,    #0
        dat.f   #0,     #0
(empty line)
(empty line)
(empty line)
(empty line)
. . .
```

First off, the empty lines are initialized as DAT.F #0, #0. By using the empty line I can better show what each warrior is doing. The MOVs bomb the fourth empty line and the ADDs change the values of the B-field of the MOV instruction. Next, the JMPs send execution back the each MOV. The core now looks like this:

```
        mov.i   $3,     $11                     <-- executing here
        add.ab  #4,     $-1
        jmp.a   $-2,    #0
        dat.f   #0,     #0
(empty line)
(empty line)
(empty line)
        dat.f   #0,     #0
(empty line)
        mov.i   $3,     $11             <-- executing here
        add.ab  #4,     $-1
        jmp.a   $-2,    #0
        dat.f   #0,     #0
(empty line)
(empty line)
(empty line)
        dat.f   #0,     #0
. . . (lots more empty lines)
```

Now, the first warrior puts a DAT right on his opponents JMP. The second warrior, because of its position this round, has to bomb all the way around the core before it can bomb the other warrior. Such is chaos. Anyways, after both warriors bomb and add again, the core looks like this:

```
        mov.i   $3,     $15
        add.ab  #4,     $-1
        jmp.a   $-2,    #0    <--executing here
        dat.f   #0,     #0
(empty line)
(empty line)
(empty line)
        dat.f   #0,     #0
(empty line)
```

```
        mov.i   $3,     $15
        add.ab  #4,     $-1
        dat.f   #0,     #0           <-- executing here
        dat.f   #0,     #0
(empty line)
(empty line)
(empty line)
        dat.f   #0,     #0
(empty line)
(empty line)
(empty line)
        dat.f   #0,     #0
. . . (lots more empty lines)
```

Now, the first warrior successfully jumps back to it's MOV instruction. The second warrior executes the DAT statement and that process terminates. If the second warrior had more than one process, it could still run, but since it only has one, it has lost. It should be noted you can use other forms of bombing, such as:

```
        jmp.x   #0,     #0
```

Which 'stun' the opponent and all processes that execute the JMP will jump back to the same line for the rest of the round unless acted upon. This will slow down and perhaps stop the other warrior from executing harmful statements to you.

Another piece of jargon--splitting. A program can use the SPL instruction to create two or more processes. It can also be used to add durability. If we add a SPL statement to the beginning of Dewdney's warrior, we get:

```
        spl.a   #0,     #0
        mov.i   $3,     $7
        add.ab  #4,     $-1
        jmp.a   $-2,    #0
        dat.f   #0,     #0
```

What this does is have one process executing the MOV instruction and one still executing the SPL instruction. This will always be true. The SPL instruction will continue to create more and more processes which all execute the MOV/ADD/JMP part of the warrior. If the SPL instruction gets bombed with a DAT, the warrior can still execute fine like the previous version. If the JMP statement gets hit with a DAT bomb, the warrior is still fine because the SPL creates processes all the time which execute the MOV and ADD statements, then terminate on the DAT. If one of the MOV or ADD statements gets hit by a DAT bomb, the warrior still survives, but doesn't do anything useful.

Now let's look at the DJN instruction. This instruction does many things all at once. A simple warrior that uses only the DJN instruction would look like:

```
        djn.a   $0,     <-1
```

What happens when this warrior executes? Let's see how this warrior works:

```
        dat.f   #0,     #0
        djn.a   $0,     <-1         <--executing here
. . . (empty lines)
```

The warrior will use the B-field of the DAT instruction before it, as a pointer to decrement. Because of the .A addressing mode, the A-field of whatever is pointed to is decremented. In this case, the DAT is decremented. But, because of < modifier, the pointer also gets decremented. Also, the DJN instruction jumps to whatever is pointed to by the A-field of the DJN instruction, so it jumps back to the same place. After a couple executions, the core looks like this:

```
        dat.f   #-1,    #0
        dat.f   #-1,    #0
        dat.f   #-1,    #-3
        djn.a   $0,     <-1         <--executing here
. . . (empty lines)
```

So, this warrior will decrement the A-field of every instruction in core eventually. This could interfere with opponent's code, but what if it doesn't and this warrior doesn't win after one pass. Let's see what happens after it has decremented almost every instruction in core. Remember, the core loops around itself, so eventually, it should be expected that the DJN instruction could decrement itself:

```
        dat.f   #-1,    #0
        dat.f   #-1,    #0
        dat.f   #-1,    #-799
        djn.a   $0,     <-1         <--executing here
        dat.f   #-1,    #0
        dat.f   #-1,    #0
        dat.f   #-1,    #0
. . . (empty lines)
```

The -799 is used because I ran all my simulations with a coresize of 800. If the coresize were 8000, then -7999 would be here. Now, if it executes this time, look what happens:

```
        dat.f   #-1,    #0
        dat.f   #-1,    #0
        dat.f   #-1,    #-800
        djn.a   $-1,    <-1         <--executing here
        dat.f   #-1,    #0
        dat.f   #-1,    #0
        dat.f   #-1,    #0
. . . (empty lines)
```

Now, the DJN jumps to whatever is pointed to in its A-field. It has previously been 0, jumping back to the same instruction. But now it is -1. It will jump back to the DAT statement and it will terminate there. This is called a suicidal warrior. If this warrior doesn't successfully terminate the opponent, it will terminate itself.

Throughout this experiment, references will be made as to speed at which warriors move bombs through the core. c is commonly the variable used for the speed of light. It is also used in corewars as the variable representing the speed of light. If a program bombs at 100%c, it bombs one instruction for every instruction it executes. Dewdney's warrior illustrated earlier bombs at 33%c because it bombs one instruction for every three instructions it executes.

Another kind of warrior is called a coreclear. A coreclear works by using either <, >, {, or } to change the position of the bomb each time sequentially through core. Here's a simple coreclear:

```
        mov.i   $2,     <-1
        jmp.a   $-1,    #0
        dat.f   #0,     #0
```

After a few cycles where the pointer is decremented, the core looks like this:

```
        dat.f   #0,     #0
        dat.f   #0,     #0
        dat.f   #0,     #0
        dat.f   #0,     #0
        dat.f   #0,     #-5
        mov.i   $2,     <-1
        jmp.a   $-1,    #0
        dat.f   #0,     #0
. . . (empty lines)
```

This will eventually put a DAT bomb everywhere in the core. This kind is called a backwards core-clear because the bombs are laid down sequentially backwards using the <. If it was a > and you changed the pointer location to 3, then it would be a forward coreclear.

Another change in the standard suicidal coreclear is to make it repeating. A repeating backwards core-clear would look like this:

```
        mov.i   $2,     <3
        jmp.a   $-1,    #0
        dat.f   #0,     #-4
        dat.f   #0,     #-4
```

Obviously, the first DAT instruction is bombed throughout the core backwards. Let's look at what happens when the warrior would commit suicide with the old instructions:

```
(more of the same dat.f #0, #-4) . . .
        dat.f   #0,     #-4
        mov.i   $2,     <3
        jmp.a   $-1,    #0
        dat.f   #0,     #-4
        dat.f   #0,     #-799
        dat.f   #0,     #-4
        dat.f   #0,     #-4
. . . (more dat.f #0, #-4)
```

Now let's see what happens when the MOV executes:

```
(more of the same dat.f #0, #-4) . . .
        dat.f   #0,     #-4
        mov.i   $2,     <3
        jmp.a   $-1,    #0
        dat.f   #0,     #-4
        dat.f   #0,     #-4
        dat.f   #0,     #-4
        dat.f   #0,     #-4
. . . (more dat.f #0, #-4)
```

Now the warrior will begin placing DAT instructions before its code. It will continue to do this until the cycles have run out or it has won.

The next terminology that can be defined is the use of pointers. Let's look at the following version of a coreclear:

```
        mov.i   $2,     <-1
        jmp.a   $-1,    <-2
        dat.f   #0,     #0
```

Now for what happens after the MOV and JMP instructions have been executed once:

```
. . . (more empty lines)
        dat.f   #0,     #0
        dat.f   #0,     #-2
        mov.i   $2,     <-1
        jmp.a   $-1,    <-2
        dat.f   #0,     #0
. . . (more empty lines)
```

Notice at this point in a normal coreclear, the DAT pointer would only be -1, but instead it is -2. Watch how this warrior bombs through core:

```
. . . (more empty lines)
(empty line)
        dat.f   #0,     #0
(empty line)
        dat.f   #0,     #0
(empty line)
        dat.f   #0,     #0
        dat.f   #0,     #-6
        mov.i   $2,     <-1
        jmp.a   $-1,    <-2
        dat.f   #0,     #0
. . . (more empty lines)
```

Every other line gets bombed in this example. The ones in between are not touched, but this kind of warrior is better because it has a better chance of finding and terminating the opposing warrior before a normal coreclear would.

A self-mutating warrior is meant to change it's own form at specific, hopefully key instances. Let's change Dewdney's warrior that bombs a DAT every fourth position to the following:

```
        spl.a   #0,     #0
        mov.i   $3,     $5
        add.ab   #4,    $-1
        jmp.a       $-2,          #0
        dat.f   >-1,    #0
```

Now, instead of bombing with a DAT instruction with numbers of 0, it uses those to it's advantage. Execution is exactly the same, until it has bombed all the way around the core and is about to bomb over itself. Here is what the core would look like:

```
. . .(more empty lines with dat.f >-1, #0 thrown in every fourth position)
        dat.f   >-1,    #0
(empty line)
        spl.a   #0,     #0
        mov.i   $3,     $1      <--- about to execute this instruction
        add.ab   #4,    $-1
        jmp.a       $-2,          #0
        dat.f   >-1,    #0
(empty line)
        dat.f   >-1,    #0
(empty line)
(empty line)
(empty line)
        dat.f   >-1,    #0
. . .(more empty lines with dat.f >-1, #0 thrown in every fourth position)
```

Now look what happens. The MOV puts the DAT bomb over its own ADD instruction. Now look how the warrior executes:

```
. . .(more empty lines with dat.f >-1, #0 thrown in every fourth position)
        dat.f   >-1,    #0
(empty line)
        spl.a   #0,     #0
        mov.i   $3,     $1
        dat.f   >-1,    #0      <--- about to execute this instruction
        jmp.a       $-2,          #0
        dat.f   >-1,    #0
(empty line)
        dat.f   >-1,    #0
(empty line)
(empty line)
(empty line)
        dat.f   >-1,    #0
. . .(more empty lines with dat.f >-1, #0 thrown in every fourth position)
```

Now look what happens. The DAT terminates the process, but not before it increments the previous instructions B-field. This just happens to be where the MOV will throw it's next DAT bomb. This warrior that initially bombed every fourth position in the core with a DAT bomb turns itself into a 33%c forward coreclear. This improvement is called self mutation.

It has been mentioned that only a DAT instruction terminates a process. This isn't always true. Two other instruction can serve to terminate processes. These are DIV and MOD. Both of these instructions deal with divison. If either of them are asked to divide by zero, they terminate as this is mathematically impossible. This could be compared to a Divide by Zero error message you may receive while debugging programs.

What's been done with Evolutionary Computing Before

Non-corewars

The first mention of the words genetic algorithm occurs in a paper published in 1967 by Bagley. (Goldberg, David E., 92) He used genetic algorithms to develop game player controls for a 'hexapawn' game, basically a 3 x 3 matrix with each opponent having three pawns at either end of the board. He used mutation, reproduction, and crossover to produce strategies for winning this game.

Many other early pioneers worked out genetic algorithms to develop answers to specific problems only, not applicable to any other field or program. An excellent list of these and many other exploits into genetic algorithms are contained in Genetic Algorithms in Search, Optimization, and Machine Learning. (Goldberg, David E., 126-127)

John H. Holland did the first major series of works on genetic algorithms 23 years ago at the University of Michigan. (Gibbs, W. Wayts, 1) His methods required using only problems that were analogous to actual chromosomes. Therefore, the problems that could be tackled were very limited. Even with these restrictions Holland did quite a large number of tests. (Goldberg, David E., 126-127)

In 1992, John R. Koza, extended Holland's methods to solve programs of any size and form. (Gibbs, W. Wayts, 1) The field has virtually exploded. Koza's methods have inspired a variety of work, at least two Genetic Programming Conferences, and he claims 51 Ph.D. theses are in progress dealing with genetic computation of some sort. (Genetic Programming.org Home Page) (Koza, John)

A few examples where evolved programs have been comparable to human-written programs appeared in (Gibbs, W. Wayts, 1). One example used genetic algorithms to create a program to control a prosthetic hand. A program has also been evolved that is capable of maneuvering a spacecraft 10 percent faster than a solution designed by an expert. Of course these are only a few examples of what has been done. Over 800 papers have been written since 1992 that deal primarily with genetic or evolutionary programming. (Langdon, W. B.).

One of the major problems with these evolved programs is their serious lack of readability. (Gibbs, W. Wayts, 3) The genetic algorithms do not produce programs that use comment sections, procedures, or any means of understanding the code. Typically, they become monolithic pieces of code. These take much more time to read and decipher, and even more time for humans to modify than normal programs. They may even contain useless instructions that could not be readily separated from the useful ones.

Corewars

The first published work on evolving corewarriors was a paper written by John Perry. (Perry, John, 1) In his paper he describes an experiment he did in which he eventually submitted two of his evolved warriors into the International CoreWars Society Tournament. His warriors got second to last and third to last. Perry's experiments were generally inconclusive because he only ran 2 generations and the 'seed' warriors were handwritten proven warriors. He did another test with 1000 randomly generated seed warriors and ran 3 generations, but in these he tried to mimic the function of assembly programs. I should point out that assembly programs are rather different than corewarriors. Although, it should be noted, when he did his experiment, the processing speed of computers was very far behind today's standards and he was limited by that. Even though his experiments were slow, they still showed promise and had good ideas.

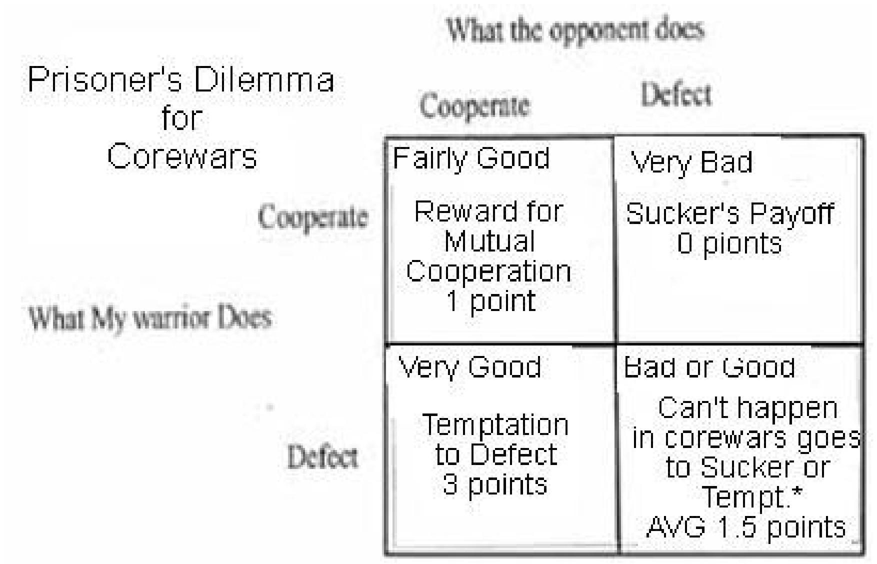
The next work that I could find on evolved corewarriors was not a 'paper' but rather a long piece of source code with a rather long comment section. (Boer, Jason, ga_war.c) I eventually ran across his homepage devoted to evolving corewarriors (Boer, Jason, Jason's Corewar Project Page). This system used mutation and reproduction in a somewhat controllable (by means of a configuration file) population of warriors. I eventually decided to base my experiments on his source code, but that comes later. Eventually, with personal computer processing speeds much higher than ever before dreamed of, Jason was able to evolve some very tough little warriors. It should be noted these warriors ran on a 'standard' corewar size of 8000. No outside warriors are ever tested against the evolved ones to determine fitness, so it is a closed system. This is something good because no clear evolutionary goal is present, except to defeat your brothers/sisters. It's drawbacks are speed limitations because of lots of hard disk accesses. It also takes a longer time to test programs that wait 80000 cycles until a tie is declared.

It should be noted that the best program Jason Boer ever evolved came in 25th, at the very bottom of the beginner hill. He ran thousands of generations, and changed the variables a lot, and could only evolve a semi-decent program for a ’beginner’ hill. This was quite an accomplishment in itself, but still leaves much to be desired.

There are a few people who have used Jason Boer’s code to evolve their own warriors. Besides that, they changed little except varying the mutation rates and number of warriors in each population. Some have evolved better warriors, but little has been done to improve results to the point where they could be competitive with human written warriors.

Why Corewars

In The Selfish Gene, there is a situation called the Prisoner’s Dilemma. (Dawkins, Richard, 209) In it he describes the conditions that must be met for a good dilemma of this type. They are best represented by the following chart.



Prisoner's Dilemma for Corewars

What the opponent does

What My warrior Does	Cooperate	Defect
Cooperate	Fairly Good Reward for Mutual Cooperation 1 point	Very Bad Sucker's Payoff 0 pionts
Defect	Very Good Temptation to Defect 3 points	Bad or Good Can't happen in corewars goes to Sucker or Tempt.* AVG 1.5 points

A corewarrior can defect by executing offensive code like bombing with DATs. It can cooperate by keeping to itself, and not interfering with the opponent. This strategy doesn’t occur frequently, because almost all corewarriors are programmed to defect.

As you may notice, corewars is an excellent tool to use to examine the consequences of this dilemma. Unfortunately, the game does not rely simply on if you cooperate or defect, but mainly how well you defect due to the fact that Defect-Defect cannot occur. In fact, if you notice, the best choice is to defect no matter what your opponent does. So, the method you use to defect becomes the factor that determines the best program.

John Perry, in his paper gave several reasons for using corewars. (Perry, John, 1-4) Corewars has many advantages because it does not mimic any natural form of life. The main similarity between real life and corewars is that in both places, the main urge is survival. He also believes that Dawkins would have liked a corewars experiment with randomly seeded code because there is no clear evolutionary goal. People have used corewars to evolve warriors that function well against a test warrior, but these warriors often fail miserably when they fight against a different warrior. Warriors in a closed system like I will be using have no goal, they may evolve into unpredictable forms.

Setup of Experiment

I used Jason Boer’s ga_war.c (Boer, Jason, ga_war.c) as the primary basis for my experiments. I changed the source code to run under a baby-size hill. This, I believe, led to much faster results, because in theory it should take 1/10th the time to run a program that goes 8000 cycles before a tie than one that goes 80000. It also let me reduce the maximum warriors length from 100 to 20. This reduced the possible number of warriors, and it eliminated more processing time. I also excluded the instruction using P-space (see Appendix A) because of their complexity and use only in advanced warriors.

The following is a mathematical representation of the possible numbers of different warriors for a baby-size hill assuming a maximum number of 100. It also assumes non-usage of the two P-space instructions:

```
Number of possible Opcodes per instruction =    17
Number of possible Modifiers per instruction =  14
Number of possible Addressing Modes per instruction =   14
Number of possible numbers -100 to +100 =       201
                --------
Number of possible instructions =       669732
Where x is the number of instructions = x^20 + x^19 + . . . x^2 + x^1
                        --------------
Total number of possible warriors.               3.296 x 10^116

        That’s a lot of possible warriors.  That is way too many to search then entire space randomly.  Now the same for a standard size hill, here are the same calculations:
Number of possible Opcodes per instruction =    19
Number of possible Modifiers per instruction =  14
Number of possible Addressing Modes per instruction =   14
Number of possible numbers -1000 to +1000 =     2001
                -------
Number of possible instructions =       7451724
Where x is the number of instructions =                x^100 + x^99 + . . . x^2 + x^1
                        --------------
Total number of possible warriors =             1.681 x 10^687
```

If the number of possible baby warriors was bad, this is much worse. This number is amazingly huge. To search through the entire space would be nearly impossible. By reducting the number of warriors to baby hill standards, things are improved, but even there the number is much too big to work through entirely. It will be many times more likely to hit upon a very good warrior in a baby size hill than a standard size hill if you speak only in random terms.

It is necessary to explain what running a generation means with ga_war.c. The first warrior in the population randomly fights for a certain number of battles one other warrior. If they tie, they are modified according to the chances dictated by the configuration file. If one wins outright, the winner replaces the loser’s code with his own code. If one has a limited win, the winner replaces the loser’s code with a modified version of itself. All these variables are controlled by the configuration file. Each warrior in a population does this once per generation. If the random selection of opponents is good, this means that each corewarrior will fight twice per generation.

All of these factors put together led me to develop my corewarriors with baby size standards rather than normal. Mainly among these was processor and hard drive speed constraints. The next step was to decide what variables to use for the configuration file. These can be changed after the program has stopped running however many generations you told it to. All the options except population size and number of battles were functions of percentages.

I decided on a population size of 100. I hoped 100 would be big enough to promote diversity, but not too big to slow down the system. The next variable was mutation rate, or how often a random piece of instruction would be modified. Several sources, including Jason’s homepage had some suggestions as far as this goes. (Boer, Jason, Jason’s Corewar Project Page) I eventually settled on initial mutation rates of around 15 and varied them down to 4 or 5 as time went along. The insertion and removal rates were used to insert or remove additional individual instructions. I decided they should be kept close together, as to avoid all warriors having a maximum length of 20 or a minimum length of 1. Hopefully by keeping the rates for insertion and removal similar and from around 5 to 15 I could keep these two extremes from happening. The resurrection rate was how often an older version of the warrior would be brought back from the dead. This was added in case all the current warriors were worse than the previous ones. The resurrection rate should be kept somewhat low, and to promote new mutations instead of constantly overwriting any progress the warriors have made. I kept it below 10.

Again, processor time was a constraint on the number of battles for each warrior to fight against its brethren. I chose to start at around 30 and vary it up to 50 and even 100 for some of the final generations.

I had two computers on which I did all of these experiments. They are both IBM compatible PCs. One is MS-DOS based, and the other is Windows '95 based, although it was restarted in MS-DOS mode to run the simulations. One has a true Pentium chip, the other has an equally fast 5x86 processor. The 5x86 machine has 40 megabytes of memory and a rather slow hard drive. The Pentium machine has 16 megabytes of memory and a much faster hard drive. Hard drive speed proved to be important, as the evolution program accesses the disk quite a bit.

I set up both computers to do, originally two separate populations of warriors. Because of file naming requirements, each set was identified by only two letters. The set run on the 5x86 machine was called AB. The set run on the Pentium machine was called AC.

The ga_war.c program uses pMARS to test each warrior. pMARS is a simulator that loads the programs, runs them, and gives the results. (Durham, Mark, et. al.) Everyone in the corewars community is indebted to Albert Ma, Nandor Sieben, Stefan Strack, and Mintardjo Wangsaw for writing pMARS.

I also used a freeware program called MARS Tournament Scheduler to compare all 100 warriors in each generation gap, so to speak, when I would tweak variables. They were stored in separate directories for comparison later. The tournament scheduler also accesses pMARS to run the battles. Again thanks go out to Stefan Strack for writing this handy program.

It should also be noted that the ga_war.c program uses only mutation and reproduction. It accomplishes mutation by randomly selecting a modifier, number, addressing mode, or opcode and changing it. Again, I used a mutation rate between 5% and 15%. It accomplishes reproduction in an unusual way. The population size remains constant throughout the entire simulation. Instead of actually reproducing, any corewarrior that can beat its opponent consistently replaces the opponent with a verbatim copy of itself. If a corewarrior beats its opponent by some margin, but not a clear victory, it replaces the opponent with a mutated version of itself. If the two tie, insertion, removal, mutation, or resurrection can occur.

I feel the exclusion of crossover was a good idea because of the way corewarriors work in general, and as I found out later, work with evolved corewarriors. Most warriors either had some non-useful code followed by the functional code or had functional code followed by code that was never executed. This is the case for some human written warriors, but every corewarrior I looked at had this form. Let's do a crossover here:

```
Warrior 1       Warrior 2
[Good, functional code] [non-useful code]
--------------------------------------Crossover here----------------------------------------------------
[non-useful code]       [Good, functional code]
        The warriors become:
Warrior 1       Warrior 2
[Good, functional code] [non-useful code]
[Good, functional code] [non-useful code]
```

Now, what do we have? One warrior which will never access the second part of it's code even though it is useful, and a warrior that will probably die very quickly. I can see no reason to turn two perfectly decent warriors into one decent warrior and one that is junk. For that reason, I believe not using crossover is a good idea for corewarriors.

Results of Experiment

Population ’AB’

This was the population run on the 5x86 machine. I ran 1000 generations of 100 warriors. Using the MARS Tournament Scheduler program I compared the populations at intervals of 100, 150, 200, 250, 300, 375, 450, 525, 600, 725, 900, and 1000 generations. These numbers were more an arbitrary convenience than planned setup. The number of generations was set by how long I could leave the computer running. Instead of providing complete reports, I have chosen to highlight some of the developments that took place during each run.

After 100 generations, the warriors had learned quite a bit. The first trick they learned to use was self-splitting by use of the SPL opcode. This usually took the form of:

```
        spl.a   #x,     y
```

It really doesn’t matter what numbers you put in. Splitting to a number addressed using # basically causes the program to execute that instruction and the one after it unless acted upon. If the one after it is bombed, the SPL can survive by itself. The second trick was using one of the decrements or increments <, >, {, or } to bomb consecutively through the core using a mov.i instruction like:

```
        mov.i   $x,     <-y
```

This combined with a SPL bombed backwards through the core once before bombing itself. Basically, if the other warrior has avoided dying by way of DAT statement, it wins because this warrior has put DATs over its own code.

One problem with the warriors at this point was the fact that they usually terminated on a DAT following their MOV line. This slowed down the bombing. They also usually had to progress through some beginning code that really had little effect on the outcome except to slow themselves down more.

After 150 generations, the warriors only learned one new trick. This trick was exemplified by the following line of code after the basic SPL instruction:

```
        mov.i   {10,     >-21
```

This piece of code is very interesting. The instruction that is used to bomb through the core is constantly decremented, so therefore, it is usually a different instruction. For example, the first time the warrior executes the MOV line, whatever is pointed to by the A-field of the instruction 10 ahead is used. Then, it is decremented and the instruction previous to that is used as the bomb. In an empty core, this leads to bombing DAT everywhere since that is what the core is initially full of. And, since the location that is bombed is incremented, the DAT usually will get put over the enemy’s code.

More than that, this coreclear is repeating. The only reason for this is because the first execution moves an instruction with a B-field of 73 to the pointer location for the position to be bombed. If this pointer got bombed by an instruction with a B-field number of less than 21, this coreclear would commit suicide. Barring that and getting a DAT thrown on it by the other program, this corewarriors will bomb until all 8000 cycles are used up. This has an advantage just in case the other warrior survived the first pass. It also increases overall survivability, as most of the previous warriors committed suicide after bombing once through the core, they bombed themselves and died.

For this warrior, it takes 3 instructions to bomb 1 instruction. It is a 33%c coreclear. It also decrements one instruction per 3 instructions, which can help mess up other warriors.

After 200 generations, a major change happened. A warrior emerged which was 13% better than the next warrior after a complete comparison. Usually before, a margin of 1% was rare, but now one warrior was consistently beating its brethren. Here are the functional (the parts that are used more than once) of this warrior:

```
        spl.f   # 9,    {-39
        mov.i   < 11,   >-20
        dat.ba  < 10,   @-63
```

The reason this warrior was so much better has to do with the placement of the pointers. The MOV line uses a pointer 11 ahead of it, while the DAT line (which terminates, but still decrements), uses one only 10 ahead of it. Since the DAT line is immediately after the MOV line, the are using the same line for their pointer. In effect, the MOV executes, decrementing once, and then the DAT executes, decrementing a second time. This doesn't improve the speed of the warrior, it is still 33%c, but it changes the bomb used more frequently. By not simply copying data around, instead copying every other location to every location at some other point, the odds of the opponent surviving are considerably less. Another advantage this warrior had was that it was a forward coreclear. All other warriors in this population seem to be backward coreclears. Apparently, the reasons this warrior scored well weren't just in it's code, but in the code of the rest of the population.

This was an amazing evolutionary jump. This tough little warrior represents the fact that these warriors can learn better ways to defeat their brethren. This one did it, and I have decided this warrior learned a new way to defeat anything else present at that time. And to think, all this after only 200 generations.

After 250 generations, nothing really new happened. Apparently, the advantages evolved in the previous round was lost when everyone got it, and since it wasn't helping, it got mutated away. The warriors were still repeating coreclears with a speed of 33%c.

After 300 generations, another very important piece of code came up. The basic warrior still consisted of a SPL and MOV line, but instead of terminating on a DAT, or jumping to random consecutive places through the core, this warrior evolve a line that made use of that line. It was to use the following:

```
        jmp.f   $ -1,   #y
```

The first part is all that really matters. Instead of terminating that process, it jumps backwards 1 instruction, to the MOV instruction. This does help out speed, as it approaches an upper limit of 50%c as more and more processes get caught in the MOV-JMP loop. It also helps out durability. Now if either the first or last functioning line gets hit, the warrior still functions by continuing its coreclear. If the SPL line is hit, the processes keep jumping back to the MOV line. If the new JMP line is hit, the SPL keeps producing new processes that keep the coreclear going. So, instead of being completely vulnerable to a single hit, this program can survive and still function properly as long as the MOV isn't hit. If the MOV is hit by a DAT, the SPL would keep producing new processes that would terminate on the DAT, keeping it alive.

Now, after 375 generations, the warriors learned a new trick. Instead of evolving a new line of code, they got rid of one. They kept the new JMP instruction, but lost the SPL instruction. There is one key advantage to this. The speed is improved to bombing one location every 2 instructions instead of 3. This makes the efficiency in the order of 50%c for the entire time. This is just slightly faster than the best warriors from the previous round. Now, if you care to figure it out, a program with 50%c efficiency will consistently win over a program with 33%c or 25%c efficiency. It will even beat out a warrior that has a speed approaching 50%c. Once again, the corewarriors learned a new, faster strategy, and took advantage of less fortunate siblings to gain more wins. Of course, now, the population was saturated with 50%c warriors. Since they won more, they reproduced more. Something new had to develop, or real stagnation could occur.

Well, unfortunately, after 450 rounds, no new strategies had evolved. The top 75 warriors represented a difference of less than 10%. The population had surely become stagnant, and all that could be hoped for was a luck mutation that somehow helped the warriors.

Well, fortunately for the population, something new did develop after 525 generations. The JMP to the previous instruction was tossed away as something better was found. It didn't decrease the efficiency, it didn't make it less durable (or more durable), but it did something completely different. It took the form of:

```
        djn.a   $ -1,   < -9
```

This particular line, when executed, will decrement the value of the A-field pointed to by the B field, in this case, -9, and decrement the 'B' field. In this case, whatever is at the -9 location is also decremented, so next time the same thing will be done to the previous instruction and so on. Next, control is passed to the previous instruction, as the -1 eludes to. So, it functions the same as a JMP -1 instruction, but adds the bonus of decrementing a number. Why decrement? Well, it can help. For instance if the opponent's warrior consisted of the following:

```
        mov.i   $10,    <-1
        jmp.a   $-1,    #0
```

You could defeat him by only using the DJN instruction. This can happen because if his critical A-field of the JMP instruction is decremented, it changes it to -2. The next time it is executed, your opponent will jump back 2 instead of 1. If the previous instruction is a DAT, the he terminates, and you win. If it isn't, let's say it's a NOP (no operation), just for fun, his efficiency is decreased to 33%c. If yours is still at 50%c, then your chances of bombing him first are much higher than his against bombing you.

After 600 generations, there was a major change as far as strategy. The four functional lines of the best program were:

```
        spl.b   # 15,   @-62
        add.x   <-28,   { 34
        mov.i   * 51,   < -8
        djn.a   $ -1,   < -9
```

The first SPL line functions the same way that all of them have so far. The ADD instruction may at first seem useless and time consuming, but it is not. The MOV instruction and the DJN instruction function exactly like the previous version except these two use the same pointer. At first, the MOV bombs an instruction, the DJN decrements the A-field of the instruction before it, and the MOV bombs the one before that. At first, the speed of this warrior may seem slow, but as more and more processes are caught up in the MOV-DJN loop, the speed approaches an upper limit of 50%c. Since the two primary instruction use the same pointer, the speed at which they bomb/decrement the core is actually 100%c. This only goes at this speed for 2/3 of the core before the ADD instruction changes the DJN string to jump back 2 instead of 1. This causes the speed to drop back to an upper limit of 33%c. Eventually the ADD instruction also changes the pointer of the DJN instruction, causing it to start over and decrement every instruction backwards while the MOV instruction bombs every instruction backwards. In effect, this is called a self mutation. (Durham, Mark, et. al.) This warrior actually does not commit suicide like some earlier warriors. It runs several different passes at the core that all serve to hopefully kill the opponent.

After 725 generations, no major changes happened. The next warrior used a SUB instruction instead of an ADD. It also never changed the pointers of the MOV and DJN instructions.

After 900 generations, nothing had happened yet. The warriors hadn't evolved much at all. I think this may be due to using low mutation rates in the final rounds like 4 and 5.

After 1000 generations, there were still no new advancements even with a high number of battles, 100. The final best warrior's functional lines were:

```
        spl.b   #-65,   # 80
        sub.f   {-48,   < 40
        mov.i   @ 54,   < -8
        djn.f   $ -1,   < -9
```

The only advantage of this warrior to the previous one is use of numbers in the SUB fields that don't stop the backwards bomb/decrement coreclear until it has almost bombed itself.

Population 'AC'

This, again, was the population run on the Pentium machine. I ran 1000 generations of 100 warriors. Using the MARS Tournament Scheduler program I compared the populations at intervals of 80, 160, 300, 500, 625, 800, 900, 1000 generations. Again, these were more arbitrary numbers that allowed me to check them before I went somewhere, went to sleep, etc. Instead of providing complete reports, I have chosen to highlight some of the developments that took place during each run.

After 80 generations, the best scoring warrior used two of the SPL instructions described above consecutively. The next instruction was an original:

```
        mov.i   # 28,   > 29
```

This amounted to moving the current instruction 29 instructions ahead. The next time it is executed, it would move 30 instructions and so forth. If any other warrior executed these, they would create more MOV instructions. The next instruction was also a MOV, but it worked differently. It effectively was a waste instruction. It did nothing but move itself onto itself. The next instruction was an interesting variation on the JMP instruction. It was:

```
        jmp.ba  > 27,   <-47
```

This, considering it comes 2 instructions after the MOV instruction, uses the same pointer. Everywhere a MOV copies itself, the JMP does not jump to. It jumps only to places the MOV isn't. Anytime another warrior is bombed by this, MOV are copied over it. If it executes an instruction not bombed, so does this warrior. Basically this leads to lots of ties. Thanks to this, this warrior won half the time and tied the other half. Nothing in this generation could beat it very well as they all consisted of basic SPL/MOV instructions that terminated on DATs.

This warrior was actually a pretty crude replicator. Eventually, it would bomb itself with MOV instructions. These would then be split to by the JMP statement. Eventually, this warrior was running all 800 possible processes, but they were in one third of the core. Why didn't this replicator break the stone barrier? Well, for one thing it was slow. Good replicators, or papers, rely on having multiple copies running at completely different places in the core to win. This one could still be defeated by even the slowest of coreclears.

This was actually, in hindsight, the closest this experiment came to producing a replicator. Unfortunately, as the stones in the population evolved into faster forms of themselves, this warrior did not. This replicator had a speed of about 20%c.

After 160 generations, the previous warrior disappeared from the scene as 33%c SPL/MOV/DAT warriors dominated the killing grounds. They easily killed the slow replicator/bomber that dominated the last population.

After 300 generations, the best warrior wasn't a three or two line warrior like the rest. The functional portion was:

```
        spl.x   # 31,   > 99
        mul.ab  $-84,   >-11
        mul.ab  { 31,   @ 11
        spl.ba  > 96,   $ 81
        mul.b   * 35,   * 50
        spl.ab  #-61,   *-70
        mov.i   > 26,   >  7
        mod.x   * 73,   #-41
```

This warrior has two of the standard SPL instructions, three more or less useless MUL instructions, a SPL instruction that could help it tie with other warriors, one functioning MOV instruction, and a terminating MOD instruction. It has a very low speed, but seems to do well because it can survive a DJN attack or a couple DAT hits. The majority of the other warriors in this population seem to be SPL/MOV/MOD or SPL/MOV/MOV/MOD. The warriors with two MOVs usually only only have one that bombs with DATs and one that bombs with itself.

Whatever advantage the winning warrior in generation 300 had, it was lost after generation 500, where the warrior below reigned supreme:

```
        spl.f   # 99,   $-32
        spl.ab  # 75,   @ 56
        mov.i   *-23,   {-25
        mov.i   # 28,   > 50
        mod.f   * 77,   >-16
```

The twin SPL instructions provide extra durability, where the twin MOV instructions give an overall speed of 18%c. The first MOV bombs DATs and the second bombs itself. the MOD is another terminating instruction. If a process survives the MOD, it is followed by a DIV which also has the possibility to terminate. Apparently these warriors are evolving more durably than set ’AB’. Set ’AB’ evolved primarily low durability, fast bombers, where these are evolve slow bombing, but highly durable warriors.

After 625 generations, the following warrior dominated:

```
spl.a   #-66,   <-39
mov.i   $100,   { -2
mov.i   # -9,   > 98
mod.ba  *-71,   > 83
```

This warrior shed the extra SPL instruction in favor of speed against durability. It still contains the MOV instruction that only moves itself. I really wonder why it doesn’t evolve a second MOV that functions like the first. The MOD instruction is the typical terminator for this test set.

After 800 generations, the following warrior dominated:

```
spl.f   #-68,   <-48
mov.i   <-62,   >  4
mov.i   <-49,   >  3
dat.x   < 46,   $ 52
```

This warrior functions with a speed of 50%c, bombing primarily with DATs or whatever lies at the pointer. The DAT instruction insures termination where the MOD did not. The twin MOV instructions both use the same pointer, so the core is bombed forward at twice the rate the MOVs would if the used separate pointers. In other words, instead of one line of DAT bombs following the other, they are placed one after another, halving the time it takes to bomb over the opponent.

After 900 generations, the warrior changed a little:

```
spl.f   #-68,   <-74
mov.i   <-62,   >  4
mov.i   @-25,   >  3
dat.b   * 23,   > -3
```

The only major change to this warrior is the fact that the second MOV instruction uses the same DAT location to bomb with unless interfered with, unlike the previous version which changed upon execution.

After a final run to 1000 generations, the following warrior came out on top:

```
spl.x   # 61,   <-75
mov.i    $-65,  >  4
mov.i   > 57,   >  3
jmz.a   $ -2,   @ 94
```

This warrior bombs more or less like the last one, except it bombs with a speed approaching an upper limit of 66%c. This is due to the JMZ instruction which causes the processes to jump back to the first MOV instruction. As more and more processes accumulate here, the speed improves. Eventually, if a nonzero instruction is bombed to a position 94 lines after the JMZ line, the processes will run whatever comes after this instruction. In most of these types of warriors, this was another terminating instruction which kept the speed below a new upper limit of 50%c.

Comparison of the Populations

The final warriors from each generation were compared. The ’AB’ warriors consistently beat the ’AC’ warriors. This shows a higher level of sophistication in the SPL/SUB/MOV/DJN warriors than in the SPL/MOV/MOV/JMZ. In general, the warriors in the AB population went for speed rather than durability whereas the warriors in the AC population went for durability rather than speed.

A lot of early evolved corewarriors never developed beyond an imp or a single DJN instruction. An imp is:

```
        mov.i   $0,     $1
```

This warrior copies itself one line ahead, and then goes to the next line. If you can find it, it is easy to kill. It also has no offensive capability, it can only tie. A single DJN instruction warrior would look like this:

```
        djn.f   $ 0,    <-1
```

This kind of warrior only decrements instruction, which some warriors can survive, whereas this warrior is suicidal after 800 cycles. Another kind of evolutionary dead-end I have heard of is a hider like:

```
        jmp.x   $0,     $0
```

This little warrior just sits there. His offensive and defensive capability is zero. Now, my warriors never reached these evolutionary dead-ends. The evolved, in my opinion, much farther.

However, it should also be noted, all of the warriors developed were stone or rock types. None of them scanned for opponents, and none replicated themselves. I haven’t heard of any evolved corewarrior breaking this barrier. I think if it does happen, it will be a paper that evolves. This would be due to the fact that papers beat stones, so they would win and reproduce more, whereas if a scissor evolved in a stone population, it would be killed off.

When I did a test consisting of the five best final warriors from each population, two papers, two scissors, and four stones, the results showed how these evolved warriors were still not competitive with the best human written warriors. The papers came in first and second, then a scissors, stone, another scissor, the three other stones, and then the evolved warriors. However, it should be noted that one of the stones, which was written with the latest strategies in mind, barely beat the best of the evolved warriors. Many thanks go to those that gave me the code for their human written warriors including Philip Kendall, Brian Haskin, David Matthew Moore, and Franz.

I also submitted some of the best warriors to the baby hill. None of them were good enough to make it. However, comparing tests of earlier warriors against the best of 1000 generations warriors showed an improvement of almost doubling the score. Of course, the bottom warrior on the hill also had twice the scores of the best evolve warrior I submitted.

Conclusion

The big question here is did these warriors learn. The answer is yes. These warriors made serious improvements toward learning to survive. Some of the instructions they used, like using a DJN instruction to loop instead of a simple JMP were and are still used by good corewarriors today. Also, some warriors had two sections of executing code, which dramatically improve chances of not being killed. In my opinion, these corewarriors learned quite a bit in only 1000 generations. They were constantly improving upon themselves, up until the last couple hundred generations. I believe this was partly my fault, as I kept the mutation rates lower for the final rounds. I think, to a point, a higher mutation rate helps more than it hurts.

Even though the results may not be great, I believe they are a great accomplish for these evolved core-warriors. I don’t know how many generations it would take for them to be competitive with warriors written by humans. However, I do believe by what they’ve done in 1000 generations proves that it may be possible. These warriors went from nothing but a single random instruction to rather complicated warriors that used every advantage they could to win.

Future Work

I believe there are two things keeping evolved corewarriors from being competitive with human-written warriors. The first is the lack of a diversity scoring principle like the Rank-Space method. (Winston, Patrick Henry, 518) With this help for rewarding diversity, I believe the stone barrier could be broken. Also, I believe the second thing necessary is processor time. In his experiments with satellite-control programs, Brian Howley used a fast workstation that took 83 hours of straight computation to evolve a program comparable to human-written programs. (Gibbs, W. Wayt, 2) If someone could either gain use of or use a workstation or a supercomputer, I believe this would also help the results. It would also take some major rewriting of ga_war.c to test for diversity as well as fitness in a population, which would also slow down the computations. But, then again, faster processors and hard drives could be the answer.

Appendix A

The following is a brief description of the corewars code set. It was put together from pieces of two textfiles, which, I am sorry, I do not know where they came from. This is meant to be used as a very general guide to understanding the concepts in this paper.

Opcodes:

DAT	terminate process
MOV	move from A to B
ADD	add A to B, store result in B
SUB	subtract A from B, store result in B
MUL	multiply A by B, store result in B
DIV	divide B by A, store result in B if A <> 0, else terminate
MOD	divide B by A, store remainder in B if A <> 0, else terminate
JMP	transfer execution to A
JMZ	transfer execution to A if B is zero
JMN	transfer execution to A if B is non-zero
DJN	decrement B, if B is non-zero, transfer execution to A
SPL	split off process to A
SLT	skip next instruction if A is less than B
CMP	same as SEQ
SEQ	Skip next instruction if A is equal to B

SNE	Skip next instruction if A is not equal to B
NOP	No operation
LDP	(*) Load P-space cell A into core address B
STP	(*) Store A-number into P-space cell B

(*) LDP and STP are instructions used in accessing a new feature of corewars, P-space. Because this paper focuses on warriors for the baby core with a limit of 20 instructions, these instructions were not used in any warriors. Basically, the baby warriors cannot fit proper P-space routines into 20 instructions. The instructions are primarily meant for normal size cores with limits of 100 instructions.

Modifiers:

.A	Instructions read and write A-fields.
.B	Instructions read and write B-fields.
.AB	Instructions read the A-field of the A-instruction and the B-field of the B-instruction and write to B-fields.
.BA	Instructions read the B-field of the A-instruction and the A-field of the B-instruction and write to A-fields.
.F	Instructions read both A- and B-fields of the the A- and B-instruction and write to both A- and B-fields (A to A and B to B).
.X	Instructions read both A- and B-fields of the the A- and B-instruction and write to both A- and B-fields exchanging fields (A to B and B to A).
.I	Instructions read and write entire instructions.

Addressing modes:

#	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

The complete proper form for a single instruction in corewars is:

```
(Opcode)(Modifier) (Addressing Mode)Number, (Addressing Mode)Number
                  |                     |  |                     |
                   \____________________/   \__________________ /
                              A-field                                              B-field
```

Reference List

Boer, Jason. (1997). ga_war.c. http://www.avalon.net/~jboer/projects/corewar/ga_war.c.

An excellent piece of C code to evolve corewarriors. The help introduction section is very helpful and the code is very readable.

Boer, Jason. (1997). Jason’s Corewar Project Page.
http://www.avalon.net/~jboer/projects/corewar/corewar.html.

A very nice homepage devoted to evolving corewarriors using his own code. Lots of warriors that he has evolved, as well as their results. Also include ideas on the best settings to use to develop better warriors.

Dawkins, Richard. (1989). The Selfish Gene. (2nd. ed.). Oxford: Oxford University Press.

An excellent book which details many functions of genetics from the gene-level instead of the individual. Good reading and brings up several ways evolution could work to produce diversity among other things.

Dewdney, A. K. (1988). The Armchair Universe: An Exploration of Computer Worlds. New York: H. Freeman.

This is where corewars began. This publication is actually Dewdney's first articles put together from their original publication in Scientific American. Dewdney proposes the rules, instructions, and gives the most readable introduction to corewars even though many things have changed.

Durham, Mark., et. al. (1998). Core War Frequently Asked Questions (rec.games.corewar). 1-22. http://www.mcs.vuw.ac.nz/~amarsden/corewars/corewar-faq.html.

An excellent help guide for anyone interested in designing, testing, and putting their warriors against others. Also provides many links to warrior and newsletter archives and personal homepages of many current people in the corewars field.

Genetic-Programming.org Home Page. (1998). http://www.genetic-programming.org/.

This contains the topics, speakers, and location of the next Genetic Programming Conference. Also has links to where to submit your papers for this conference, and links to some of the more prominent members of the genetic programming field.

Gibbs, W. Wayts. (1996). Programming with Primordial Ooze. Scientific American. 1-3. http://www.sciam.com/1096issue/1096techbus3.html.

A summary of what has been done in recent years with genetic algorithms. Contains examples of ways programs made by genetic algorithms have proved themselves equal to or better than human written programs.

Goldberg, David E. (1989). Genetic Algorithms in Search, Optimization, and Machine Learning. Amsterdam: Addison-Wesley.

The official guide to using genetic algorithms. Contains dozens of samples and code to do your own tests with. Also written as a textbook, so it contains very clear explanations of everything.

Koza, John. (1998). John Koza's Home Page. http://www-cs-faculty.stanford.edu/%7Ekoza/index.html#anchor5393849.

An excellent list of resources, contacts, current and future projects from the forerunner in the genetic programming field. This homepage contains hundreds if not thousands of links all related to Genetic Programming and what is being done with it.

Langdon, W. B. (1997). Genetic Programming Bibliography. http://www.cs.bham.ac.uk/~wbl/biblio/gp-bibliography.html.

A comprehensive list of papers that have been done using genetic programming. If it's been done, I'm pretty sure it is here. They claim over 800 resources, and I believe it.

Perry, John. (198?). Core Wars Genetics: The Evolution of Predation. 1-12. http://www.koth.org/evolving_warriors.html.

The first paper done using genetic algorithms to evolve corewarriors. Although it was done awhile ago, and many things have changed, it is still an excellent introduction to evolving corewarriors, although not very good at explaining corewars.

Peters, James A. (1959). Classic Papers in Genetics. Englewood Cliffs, N. J.: Prentice-Hall.

A collection of papers that form the basis for genetics today. Contains everything from basics like Gregor Mendel’s Plant Hybridization to the genetic results of atomic bombs of Hiroshima and Nagasaki.

Winston, Patrick Henry. (1993). Artificial Intelligence. (3rd. ed.). Milan: Addison-Wesley. 505-528.

An excellent textbook containing a complete overview of the Artificial Intelligence field. Contains lots of examples and problems. Brief section of Learning by Simulating Evolution which is a very basic introduction to genetic algorithms.

Bibliography

Boer, Jason. (1997). ga_war.c. http://www.avalon.net/~jboer/projects/corewar/ga_war.c.

An excellent piece of C code to evolve corewarriors. The help introduction section is very helpful and the code is very readable.

Boer, Jason. (1997). Jason’s Corewar Project Page.
http://www.avalon.net/~jboer/projects/corewar/corewar.html.

A very nice homepage devoted to evolving corewarriors using his own code. Lots of warriors that he has evolved, as well as their results. Also include ideas on the best settings to use to develop better warriors.

Dawkins, Richard. (1989). The Selfish Gene. (2nd. ed.). Oxford: Oxford University Press.

An excellent book which details many functions of genetics from the gene-level instead of the individual. Good reading and brings up several ways evolution could work to produce diversity among other things.

Dewdney, A. K. (1988). The Armchair Universe: An Exploration of Computer Worlds. New York: H. Freeman.

This is where corewars began. This publication is actually Dewdney’s first articles put together from their original publication in Scientific American. Dewdney proposes the rules, instructions, and gives the most readable introduction to corewars even though many things have changed.

Dewdney, A. K. (1990). The Magic Machine: A Handbook of Computer Sorcery. New York: H. Freeman.

More updates and new products of corewars appear in this compilation. Also details on the first International Core Wars Society tournament and the better scoring warriors.

Durham, Mark., et. al. (1998). Core War Frequently Asked Questions (rec.games.corewar). 1-22.
http://www.mcs.vuw.ac.nz/~amarsden/corewars/corewar-faq.html.

An excellent help guide for anyone interested in designing, testing, and putting their warriors against others. Also provides many links to warrior and newsletter archives and personal homepages of many current people in the corewars field.

Genetic-Programming.org Home Page. (1998). http://www.genetic-programming.org/.

This contains the topics, speakers, and location of the next Genetic Programming Conference. Also has links to where to submit your papers for this conference, and links to some of the more prominent members of the genetic programming field.

Gibbs, W. Wayts. (1996). Programming with Primordial Ooze. Scientific American. 1-3. http://www.sciam.com/1096issue/1096techbus3.html.

A summary of what has been done in recent years with genetic algorithms. Contains examples of ways programs made by genetic algorithms have proved themselves equal to or better than human written programs.

Goldberg, David E. (1989). Genetic Algorithms in Search, Optimization, and Machine Learning. Amsterdam: Addison-Wesley.

The official guide to using genetic algorithms. Contains dozens of samples and code to do your own tests with. Also written as a textbook, so it contains very clear explanations of everything.

Koza, John. (1998). John Koza's Home Page. http://www-cs-faculty.stanford.edu/%7Ekoza/index.html#anchor5393849.

An excellent list of resources, contacts, current and future projects from the forerunner in the genetic programming field. This homepage contains hundreds if not thousands of links all related to Genetic Programming and what is being done with it.

Langdon, W. B. (1997). Genetic Programming Bibliography. http://www.cs.bham.ac.uk/~wbl/biblio/gp-bibliography.html.

A comprehensive list of papers that have been done using genetic programming. If it's been done, I'm pretty sure it is here. They claim over 800 resources, and I believe it.

Perry, John. (198?). Core Wars Genetics: The Evolution of Predation. 1-12. http://www.koth.org/evolving_warriors.html.

The first paper done using genetic algorithms to evolve corewarriors. Although it was done awhile ago, and many things have changed, it is still an excellent introduction to evolving corewarriors, although not very good at explaining corewars.

Peters, James A. (1959). Classic Papers in Genetics. Englewood Cliffs, N. J.: Prentice-Hall.

A collection of papers that form the basis for genetics today. Contains everything from basics like Gregor Mendel's Plant Hybridization to the genetic results of atomic bombs of Hiroshima and Nagasaki.

Winston, Patrick Henry. (1993). Artificial Intelligence. (3rd. ed.). Milan: Addison-Wesley. 505-528.

An excellent textbook containing a complete overview of the Artificial Intelligence field. Contains lots of examples and problems. Brief section of Learning by Simulating Evolution which is a very basic introduction to genetic algorithms.

```
geovisit();
```