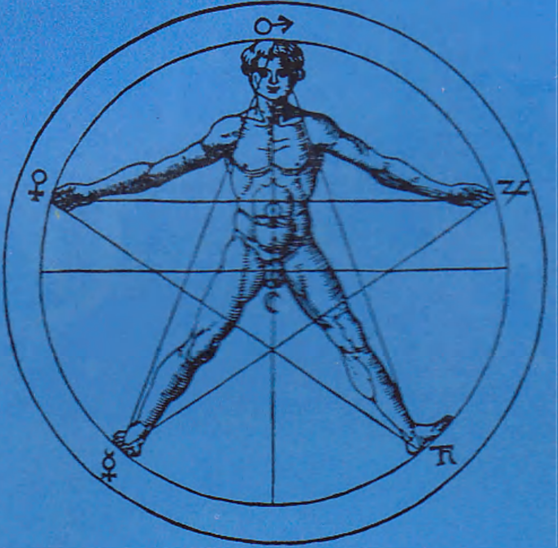
Publications of the Finnish Artificial Intelligence Society (FAIS)



Tekoälyn uudet suunnat

New Directions in Artificial Intelligence

Vol. 1: Esitelmät – Conference Papers

Finnish Artificial Intelligence Conference
Suomen Tekoälytutkimuksen Päivät STeP–92
Teknillinen korkeakoulu, 9–11.6.1992, Otaniemi, Finland

Toimittajat – Editors

Eero Hyvönen, Jouko Seppänen ja Markku Syrjänen

Suomen Tekoälyseura ry – Teknillinen korkeakoulu

Core Wars: A Testbench for Artificial Life?

Heikki Hyötyniemi
Helsinki Univ. of Technology, Control Eng. Laboratory
Otakaari 5 A, 02150 Espoo

Abstract

In the game of *Core Wars* two or more computer programs are run simultaneously by a hyphotetical processor in an imaginary core memory. The purpose of these programs is to locate the enemy processes and destroy them.

In this paper, the rules of Core Wars are repeated, some examples of simple programs are given, and the behavior of these programs is simulated. The possibilities of the Core Wars environment as a workbench for studies of *Artificial Life* are discussed.

1 Introduction

Articicial Life research is the study of the universal properties of living organisms. The research mainly consists of computer simulations. In a computer it is easy to write programs whose behavior strangely resembles that of living organisms. Programs may be designed to act in a characteristic way, to copy and defend themselves against attacks. It would be nice to have an environment for simulating and analyzing properties of Artificial Life ideas.

In *Core Wars game,* a framework is defined for creating 'genetic codes' and simulating their behavior as they interact. Is Core Wars just a computer game among hundreds of others, or should it deserve more attention?

2 What is 'Artificial Life'?

It is not easy to define Artificial Life. The fundamental, more or less philosophical question is, *what is Life*—another deep question is, *what is Artificial!* Perhaps it is best to leave these questions unanswered, as no consensus has been reached. In many respects, there are parallels with Artificial Intelligence research: these fields share an intuitively appealing problem statement. Unfortunately, in Artificial Life, the problem statement is also just as intuitively defined.

Our experiences exclusively with carbon based life make it difficult to distinguish between the properties of down-to-earth life forms and the universal qualities of all living structures. The organisms that we are familiar with are all subordinate to the constraints of physics and chemistry. The target of Artificial Life research is to augment our understanding of *life-as-we-know-it* with *life-as-it-could-be.* 'Silicon life' need not be bound by the real world restrictions, and that is why computer is the basic tool in experiments.

In contrast to biological sciences that try to *analyze* existing life forms, in Artificial Life, new life forms are *synthesized.*

It should not be a surprise that there is no exact theory for Artificial Life. In practice, the research is based on simulation studies and building analogies, the experiments being more or less convincing. Plenty of simulation programs for Artificial Life have been written, each program concentrating on some specific aspect of behavior.

3 Relation to some other fields

The imitation of biological processes has a longer history than the name of Artificial Life. For example, in the early seventies the game of LIFE was introduced by John Conway—it was a *cellular automaton* with surprisingly simple locally controlled rules that were able to create highly complex and fascinating behavior. After that, cellular automata have been applied to many tasks of physical systems modeling.

As the name suggests, the study of *genetic algorithms* has also borrowed its principles from biology. The notion that populations of living organisms have adapted optimally in their environment as a result of stochastic mutations and simple deformations of the genetic code, has encouraged scientists in applying the same idea of optimization in more technical tasks. If compared to the Artificial Life approach, the difference is that when using genetic algorithms for optimization, the *result* of the computation is the most important thing, whereas in Artificial Life applications, the *dynamics* of the ongoing processes plays the major part.

It goes without saying that Artificial Life research is a near relative of Artificial Intelligence. The same enthusiasm can be seen in the Artificial Life research community today as there used to be among AI researchers a few decades ago. However, the development of this field is not likely to be as turbulent as it was in the sixties. The researchers do not even claim that their study would be of any use—it is basic research for clarifying the fundamental issues that have only academic interest. No short-sighted projects or massive financing are likely to upset the general public. At least up to now, Artificial Life research has been a branch of interested individuals' spare time activity.

Recalling the ever lasting Artificial Intelligence debate, Artificial Life is not likely to give rise to such vigorous arguments. This time, people seem to be more tolerant about the methodologies and the results. The property of being *alive* is not such a monopoly of humans as being *intelligent* is.

There is a growing interest in complex dynamic systems— chaos theory, fractals, and complex nonlinear systems are explored extensively. Very simple rules, when applied a plenty of times, result in strange emergent behavior that could not be foreseen by looking merely at the rules themselves. Results are also sensitive to changes in the initial values, so that minor changes may result in totally differing large scale dynamics. This astonishing correspondence between simple structures and complex behavior is encouraging. Maybe this works both ways—perhaps the complexity of living things is generated by some basic set of simple rules?

However, a feature that is common to all of these fields, is the need of massive computing capacity. It is easy to understand that none of these paradigms could flourish without the modern hardware and software tools.

4 Core Wars as an Artificial Life example

In the Core Wars game, a set of simple machine code instructions is used to write *battle programs.* In the game, two or more programs run simultaneously, trying to stay alive and trying to eliminate the other programs at the same time. There does not exist real hardware for executing the instructions, but running the programs can be simulated.

Core Wars can be analyzed as an environment for simulating competition for resources between different species. The program code defines the *genotype* that is reflected in the *phenotype* or the visual behavior of the process. However well defined the effects of individual instructions of the code are, the outcome of interactions between processes is unpredictable.

Core Wars programs are easy to write and debug. Could Core Wars program syntax be used as a universal programming language for fast implementation of Artificial Life ideas?

Is Core Wars a good environment for studying Artificial Life? One feature all Artificial Life programs share is that whatever they do and however they do it, the dynamics of the processes is fascinating to look at. In this respect, at least, Core Wars game fulfills the Artificial Life requirements excellently.

5 The REDCODE language

The language for writing Core Wars programs is called REDCODE. REDCODE only includes the most important instructions for arithmetic, testing, copying, branching, and spawning new processes. The instructions that are available in Core Wars are given below:

Mnemonic	Explanation
`DAT A`	A nonexecutable data value *A*
`MOV A B`	Move contents of *eff(A)* to *eff(B)*
`ADD A B`	Add contents of *eff(A)* to *eff(B)*
`SUB A B`	Subtract contents of *eff(A)* from *eff(B)*
`CMP A B`	If contents of *eff(A)* and *eff(B)* are equal, skip the next instruction
`SPL A`	Split execution between the next instruction and *eff(A)*
`JMP A`	Transfer control to *eff(A)*
`JMZ A B`	jump to *eff(A),* if contents of *eff(B)* is zero
`JMN A B`	jump to *eff(A),* if contents of *eff(B)* is not zero
`DJN A B`	Subtract 1 from contents of *eff(B)* and jump to *eff(A),* if result is not zero.

Above, *eff(A)* is used to denote the *effective address* of *A*. The contents of the effective address *eff(A)* is found as follows:

Syntax	Value
`#A`	*Immediate addressing:* the number *A*
`A`	*Direct addressing:* the contents of the memory location *A*
`@A`	*Indirect addressing:* the contents of memory location *A* is taken as the address to the final memory location, calculated starting from *A*
`<A`	Like `@A`, but contents of *A* is first decremented.

Addressing is relative to the memory cell that is currently being read, positive values referring forward and negative backwards in memory. The core memory is circular so that no boundary effects need to be taken into account.

Even if REDCODE is a very low level language, no binary digit manipulation is needed. Numbers can be any size, but all calculation is done using mod N arithmetic, where N is the amount of memory cells in the circular core. The REDCODE compiler is an assembler that allows one to use symbolic names for memory locations, converting the references to relative addresses. When a program is run, the first instruction to be executed is labeled START. After that, the program counter normally steps forward one instruction at a time, if no branching instruction is executed. If a nonexecutable statement is encountered, the process is killed.

When Core Wars simulation is run as a battle game, competing programs are loaded to arbitrary memory addresses. During a competition, each program is given the same amount of processor time, so that if some program has spawned multiple processes, the execution of that program becomes slower.

Perhaps the simplest interesting Core Wars program is IMP, where the only action is to copy its only instruction one step forward, waiting there for the program counter to proceed to that memory location. Effectively, IMP traverses through the memory destroying everything it touches.

```
START MOV START next
next  ...
```

This IMP is also a good example of programs that are prone to 'steal the soul' of other running programs: if another program counter jumps to code that has been overwritten by IMP, it will start repeating IMP behavior forever.

6 Program examples

Some REDCODE programs that differ much from each other are given below as examples. The first one has proven to be a kind of a standard exemplar for reproducing programs:

MICE

```
ptr   DAT #0
START MOV #12 ptr
loop  MOV @ptr <next      Copy program code tail first
      DJN loop
      SPL @next           Start the child process
      ADD #653
      JMZ START ptr       Redo the copying to yet another place.
next  DAT 833
```

This MICE, by Chip Wendell from Floral Park, New York, is a very aggressive piece of program code. It repeats to copy itself as fast as it can spawning new processes that reproduce again. Because of its fast spreading, it is usually difficult to destroy. However, it gets slower and slower as the amount of processes grows.

The second example illustrates a brute force approach to destroying enemy processes. This program tries to 'wipe out' all other processes:

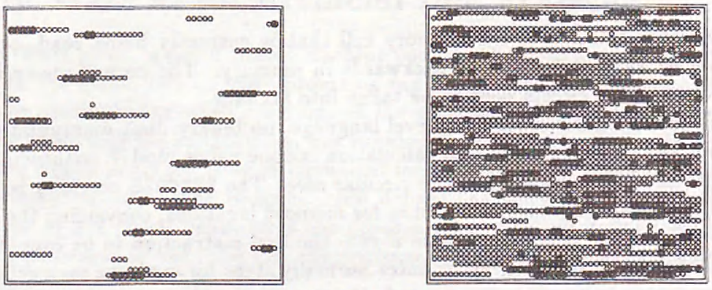


Figure 1. Typical MICE behavior after 500 and after 10000 steps. The square represents a core of 2500 memory cells, with those cells that have been manipulated by the process being encircled. Lighter circles denote data area, and darker ones represent cells that have been pointed to by a program counter.

FOOL

```
buff  DAT #1
fool  JMP fool          Trap command being used as a bomb
START MOV fool @ptr     Send the bomb
      ADD #1 ptr
      JMN START buff    If own code starts becoming corrupted,
      MOV #3 ptr        save it by skipping next memory cells
      MOV #1 buff
      DJN START ctr     Bomb with JMP trap five times,
      MOV kill fool     thereafter kill with DAT.
      JMP START
ptr   DAT #3
ctr   DAT #5
kill  DAT #0
```

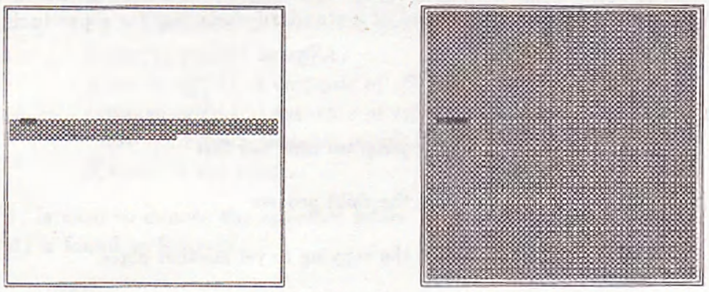


Figure 2. Typical FOOL behavior after 500 and after 10000 steps

This code is due to Fredrik Wilhelmsen from Trondheim, Norway, and it was specially designed to beat MICE. The whole memory is filled repetitively with JMP 0 traps, that efficiently freeze all program counters that happen to execute these commands. The finishing blow is done by killing the frozen processes using data bombs. Because no copies of the code are produced, the survival of FOOL is dependent of its ability to hide in the memory. The larger the core is, the easier it is to hide, but going through the memory takes a longer time, on the other hand. It is interesting to note that even if FOOL is immune against its own bomb attacks, it has no means to defend itself against another FOOL running simultaneously.

Next, a relatively long piece of code that is 'unsinkable'. In principle, continuous validity checking facilitates self-repair:

BIGBROTHER

```
sum   DAT #0               Counters
ctr1  DAT #0                 .
ctr2  DAT #0                 .
bomb  DAT #97              Pointer to the target to be destroyed
START ADD data1 bomb
      MOV #39 ctr1
      MOV #38 ctr2
      ADD data1 ctr2
comp  CMP <ctr1 <ctr2      Compare own code with the child process code
      JMP error
      CMP #3 ctr1
      JMP comp
shoot MOV bomb @bomb       No problems—send data bomb and return
      JMP START
error MOV data5 sum        Some difference was detected
      MOV #42 ctr2
calc  ADD <ctr2 sum        Calculate the check sum for own code
      CMP #2 ctr2
      JMP calc
      JMZ other sum
      MOV bomb other       Own code is corrupted—commit suicide
other MOV #37 ctr1
      MOV #38 ctr2
      ADD data1 ctr2
kill  MOV bomb <ctr2       Kill the child process
      DJN kill ctr1
copy  MOV data1 dataN      Replace child process code with own code:
      MOV #45 ctr1           .
      MOV #43 ctr2           .
      ADD data1 ctr2         .
addr  MOV <ctr1 <ctr2      First, rotate addresses
      CMP #40 ctr1           .
      JMP addr               .
      SUB #1 ctr1            .
rest  MOV <ctr1 <ctr2      Second, copy the rest of the code tail first
      JMN rest ctr1
      ADD #3 ctr2
      SPL @ctr2            Restart the child process
      JMP START
check DAT #506             Check sum value for correct code
data1 DAT #511             Addresses of the other processes
data2 DAT #253               .
```

```
data3 DAT #-1023                .
data4 DAT #2047                 .
data5 DAT #-1788            The last one points back to the first process.
dataN DAT #0
```

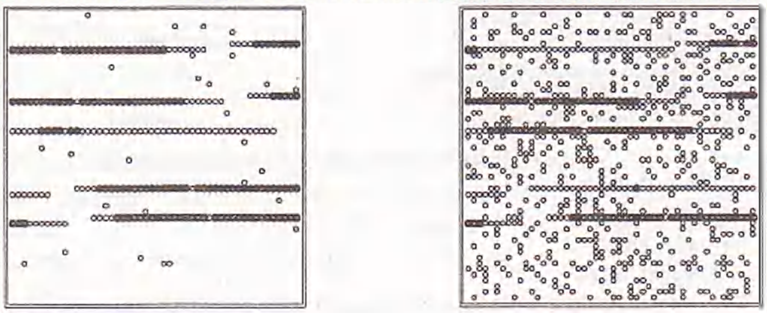


Figure 3. Typical BIGBROTHER behavior after 5000 and after 100000 steps

Among these examples, BIGBROTHER is the most sophisticated program. It continuously compares its own code to code that is found in some other place in the memory. If the contents of the corresponding memory locations is not equal, the check sum is calculated. If own code is corrupted, the process silently terminates execution, otherwise it copies itself to the other place and creates a subprocess that starts executing the new code. This way, arbitrary amount of copies of the same code (five in the program above) check each other, correcting each other's errors.

7 Core Wars battles

When letting two or more programs run simultaneously in the core, the game is a kind of an struggle of life between various species. Two programs are planted randomly in the memory, and the code to survive the most of the duels wins.

Competitions between Core Wars battle programs have been organized—during the first tournament that was held in Boston in 1987, incidentally, MICE, shown above, appeared to be the most tough-lived. The fast breeding of the code was its key to victory.

According to the tournament rules, a program is said to have survived the battle if any of the program counters is still active after a fixed amount of cycles, no matter what instructions it executes. This criterion is not very good—some emphasis should be put on the *robustness* of the code, or how probably the *initial* code is still active and behaving as it was designed. MICE is actually self-destructive, and it could not stay alive for a very long time, not executing its initial code anyhow.

Writing the ultimate winning code seems not to be possible. For memory size 4000, for example, FOOL usually beats MICE, MICE beats BIGBROTHER, but BIGBROTHER beats FOOL. In the table on the next page, quantitative duel statistics are displayed for various memory sizes. The percentages illustrate the winning probability of a program against another program—in a large memory, programs may coexist for a long time, and the sum of figures may not be 100. The normal reason for a draw was that both programs had ended in a cycle of trivial commands.

Core size	MICE vs. FOOL		FOOL vs. BIGBROTHER		BIGBROTHER vs. MICE	
1000	71 %	18 %	$\approx$ 100 %	$\approx$ 0 %	$\approx$ 0 %	$\approx$ 100 %
2000	67 %	27 %	67 %	33 %	$\approx$ 0 %	88 %
4000	33 %	57 %	$\approx$ 0 %	$\approx$ 100 %	$\approx$ 0 %	70 %
10000	50 %	43 %	$\approx$ 0 %	$\approx$ 100 %	$\approx$ 0 %	69 %
20000	50 %	42 %	$\approx$ 0 %	$\approx$ 100 %	$\approx$ 0 %	40 %

Data bombs that are sent in order to hit the enemy code are the most common way to attack. As compared to DAT instructions, fragments of executable code that are sent as bombs can be very efficient as the enemy program may be not only killed but *trapped* to execute something else. This kind of dead code fragments are like viruses, that are activated only when they have landed in a living process. For example, FOOL is a 'virus generator' that tries to infect competing programs making them execute a trivial branching command for ever.

8 Artificial Life aspects

Even one of the simplest REDCODE programs, the one-line IMP that was given earlier, can 'reproduce' and 'move'. As a battle program, it is likely to survive if the core is small enough. More complex codes can accomplish more complex schemes. It should seem likely that REDCODE is a good substrate for Artificial Life.

In principle, Core Wars can be interpreted as a testbench for experimenting the idea of 'the survival of the fittest'. Programs may not only survive, but new hybrid codes may emerge as copied programs overlap. Interesting mutations and mixing of code often happens in the course of the game, and visual surprises are common. However, there are some problems.

Finding a good set of basic operations is one of the main difficulties in Artificial Life programs. The results are highly dependent of the rules—in Core Wars, very easily a simple degenerated winning strategy emerges and more complex life forms vanish. The problems with Core Wars viruses illustrate well the problems of finding good rules assuring interesting behavior. For example, remember that there are no mechanisms to control or keep track of all program counters a process has created. Consider the following virus bomb:

```
bomb   SPL bomb
```

If a process happens to execute the above instruction, it will spawn infinite amount of new processes. If the newest process is the next one in the row of processes to be run, *nothing else will ever be done by that program.* Each step it will only spawn a new process.

The contemporary rules encourage wild, uncontrolled duplication of processes. In addition to this, there exist trivial winning strategies, like sending pathological virus bombs as the one above. It is funny to see how difficult it is to predict large scale processes even if the lowest level is thoroughly known: the creators of the game have been complementing the rules many times after it was first introduced, and still there are loopholes.

The instructions of REDCODE are perhaps too simple. Long programs needed to achieve interesting behavior are fragile and prone to get fragmented and void. A minor change in the code probably causes self-destruction. It could be useful to define more powerful parameterized commands—being able to copy a segment of memory in one cycle would usually make programs much shorter, for instance.

One problem with the contemporary Core Wars standard is its lack of *internal control:* there are no means to resolve the amount of running processes or the location of program counters or the age of processes. It is impossible to write a program that would always be able to control its state.

According to the experiments, the most probable outcome at the moment is the gradual degeneration of the genetic code during simulations. The *evolution of complexity* that should be characteristic to Artificial Life processes, is never achieved. Following this line of reasoning, Core Wars cannot be a useful Artificial Life simulation environment.

9 Conclusions

Usually, programs simulating Artificial Life generate new, more complex life forms. In this respect, Core Wars is not a good environment for Artificial Life experiments—applying the contemporary rules, living is merely sustaining the inevitable degeneration. Rather than resembling Artificial Life, nowadays Core Wars processes simulate *artificial death!*

After some changes to the REDCODE language the evolution might become positive, not negative—but what kind of enhancements to the rules are needed?

Acknowledgement

The initial version of mars simulation program that was used here was written by Fredrik Wilhelmsen at the Norwegian Institute of Technology, Division of Engineering Cybernetics, in Trondheim, Norway.

References

Different aspects of Artificial Life have been discussed, for instance, in "Artificial Life", edited by Christopher G. Langton (Addison-Wesley, 1989). This book is a collection of papers presented during the *Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems* that was held in Los Alamos, New Mexico, in September 1987.

Core Wars has been discussed by A.K. Dewdney in the "Computer Recreations" column of *Scientific American:*

- *A Core War bestiary of viruses, worms and other threats to computer memories,* in March 1985 issue, pages 14–19,
- *A program called MICE nibbles its way to victory at the first Core War tournament,* in January 1987 issue, pages 8–11, and
- *Of worms, viruses and Core War,* in March 1989 issue, pages 90–93.