Nenad Tomašev

Building a CoreWar Optimizer

-Diplomski rad-

Novi Sad, 2008
1 Foreword

In order to better understand the world we live in, we’ve been building models of reality ever since the dawn of scientific thought. This is perhaps best illustrated when observing physical theories. With the ascension of modern mathematics, more delicate models became possible and the tools for their analysis were available. The next big change followed the computational revolution which had occurred in the 20th century. Testing many existing hypotheses became possible. Also, many new models emerged, some of them relying solely on the power of simulation. We have been fascinated by Life for many centuries. We have observed it. We have studied it. We have even clumsily defined it many times over. Suddenly, with computers at hand, the idea formed of trying to create something resembling life as we know it, but located in the silicon world of computer memory instead. This artificial life was a fascination as much as a controversy. It will be many years before the opinions converge on this issue, if ever. When thinking of creating models of life in the simulated environment provided by computers, it is only natural to think of the similarity between organisms and some sorts of computer programs. When trying to represent models of life via this mapping, the next logical thing is to start at the small scale, at the scale of assembly coding. Assembly-like language called Redcode and the simulation known as CoreWar have inspired many artificial life projects, such as Tierra, Coreworld, Avida and Quantum Coreworld. Quantum Coreworld project is even backward-compatible to the basic CoreWar. These projects were never in the focus of the scientific community, but have regardlessly provided valuable insights into many mechanisms implemented in the models and even caused outrage among some conservative circles, as was the case with Avida. I have practised CoreWar for several years, with some success, to say the least. I’ve managed to claim the first place in 5 of the leagues several times over, participated in tournaments and organized some. I’ve also started CoreExplorer, an online beginners magazine. In other words, I have been completely fascinated by this simulation. There is much simplicity in CoreWar and also an unexpected amount of inherent complexity. Apart from that, I have realized something else. It is a perfect playground. A huge state space, many program types, subtleties and variations at each step. Some time ago I have obtained an interest in optimization theory, perhaps due to my interests in data mining which relies on optimization in the hypotheses space. I immediately knew what I was about to do. I wanted to test some of the basic algorithms that I have looked at and as my playground I chose CoreWar. In the recent years, optimization of the existing programs has become a necessity in CoreWar and there were no advanced tools for it, Optimax was great for a start, but
its focus was more on speed than on the search for good solutions. This is why I have created CoreWar Optimizer. It is, just as its name implies, an optimization tool for CoreWar. It has a nice user interface and it offers 5 different optimization methods. It is what this graduation thesis is all about. I hope that you’ll enjoy reading it.

The paper is organized as follows. At the very beginning, project goal is presented. Section 3 introduces the reader to CoreWar, starting with the basic definitions and surveying the most common strategies in some detail. It is far from being a comprehensive tutorial, but that is beyond the scope of this paper. Optimization algorithms in general are presented in section 4. The focus was on the methods implemented in the optimization tool, the theory of optimization is much, much bigger than that and it would be overoptimistic to even try addressing all the relevant issues. Section 5 presents the features of CoreWar Optimizer. Finally, section 6 outlines the directions for future work and gives some perspective to the text presented here.

It is said that the journey of a thousand miles begins with one step. The first steps are always the hardest. It has been some time since I’ve decided to embark upon this journey of mine and follow the winding roads that lie ahead. There are still crossroads to come upon and doors to be opened, and yet I can’t help but gaze in amazement each time I set my eyes on a calendar. I am not on this journey alone and I’ve met many wonderful people who helped along the way. Some helped with advice, some with friendship and love, and some with home-baked cakes and a cup of coffee. It really doesn’t matter. I wouldn’t have made it alone just as nobody else would. Even as I’m typing these lines, I feel the reluctance mounting as I am completely aware that no matter whom I choose to thank on this special occasion - I am bound to leave someone out. However, perfection is quite elusive and not something one should strive for in a foreword of any kind. This is why I have in the end chosen to stick to the formal template. So, without further ado, I will now point towards the ones who were there for me even though I never asked for it.

I would first like to express my gratitude to my mentor Dr Dragan Mašulović for his help in bringing this project and this text to its current shape. I would also like to thank the members of the committee, Dr Mirjana Ivanović and Dr Siniša Crvenković for their time. As it turns out, I have no choice but to thank them once again, Dr Ivanović for her help with the data mining projects and for pointing me in the directions that I have chosen to follow in the future, Dr Crvenković for his selfless devotion to the students on our department and for providing all the inspiration and assistance in the past years.
I would like to thank Mr Miloš Radovanović on his assistance in the data mining projects and for introducing me to the LaTeX text editing environment which I have also used to write this text.

I must wholeheartedly thank the entire staff of the Petnica Science Center. They have introduced me to the scientific method and have organized many fascinating seminars and lectures. I am overjoyed by the fact that I have had the opportunity to pay them back by taking part in some of the recent seminars as a teaching assistant (not to be confused with the formal university title of the same name).

It has been a couple of weeks since my highschool class had come together to celebrate the five years that have passed since we’ve graduated. Many memories emerged, as it is usually the case. I would like to use this opportunity to thank all the teaching staff in my highschool, especially (now retired) prof Ana Harpanj and all the others that have led me to think out of the box.

Mathematics has, in a way, touched my life. It would be unfair if I were not to mention my sincere gratitude towards the members of the Archimedes school of mathematics, Dr Ratko Tošić and prof Milica Prošić.

After long days spent chasing after knowledge and skill, there would be nothing left if it were not for my friends and family. They have painted across the canvas of my life more than anyone else. I thank my mother for her exotic meals, my father for being close although far away. Finally, I would like to thank my cat for being such a cute little terror and making me smile whenever I was stressed and pressed by work.

Novi Sad, July 2008.

Nenad Tomašev
2 Project Goal

The goal of this project was to examine the possibilities for instruction field value optimization in Redcode programs and to construct a software tool that would enable CoreWar experts to easily utilize various optimization methods while tuning the parameters in their programs. Naturally, it’d be too much to expect that such an application could soon become the default CoreWar developers kit. However, the application was designed so that its functionality could easily be extended and plans for future versions are already in motion. The need for such an optimization tool in CoreWar community was immense, even though Optimax \[23\] had reached high popularity in recent years. The particular problem common to all current optimizer tools is that the search through the value space is completely random. Such an approach can not be satisfactory in case where one has to try to find acceptable sub-optimal solutions in number of iterations comprising merely a small fraction of the whole search space, sometimes even \(1/10^{12}\). The focus in previous tools was mainly on choosing a proper benchmark, building fast MARS and results display. It was the authors intention to devise at first a complementary tool, an alternative to the existing ones, and then via the process of iterative adjustments and improvements reach the point where CoreWar Optimizer could be used in each and every part of Redcode program optimization.
3 CoreWar

Majority of artificial life research focuses on the exploration of various artificial environments and possibilities for self-organization, adaptation, competition and most importantly - survival in the imposed circumstances. Sometimes the artificial worlds are designed with the specific intent to mimic our own views of reality, or a particular model of some real interaction. In such cases they are extremely useful in testing scientific hypothesis and gaining insight into what may constitute a basis for the observed process. The simulated environments range from extremely simple ones to those exhibiting chaotic dynamics, phase transitions, multiple equilibria and many interesting properties. On the other hand, some artificial environments are not meant to be simplified models of reality, but instead pose novel problems to be solved, which is most often achieved via implementation of evolutionary processes. Several systems of the latter kind were inspired by CoreWar. [3]

In CoreWar, programs written in a language called Redcode compete against each other in a simulated environment that is being controlled by the MARS(Memory Array Redcode Simulator). The program that seizes complete control of the process queue wins the encounter. CoreWar programs are referred to as warriors. Competing programs are being loaded into a looping memory array and are only given access to that part of the memory. Redcode differs from the usual assembly in several important ways, and all of its peculiarities will be thoroughly explained in section 3.1.

CoreWar was introduced to the scientific community by A.K.Dewdney in 1984 in an article published in Scientific American. [6] That article drew attention of several programmers and it was a year later when the International CoreWar Society was formed. CoreWar represented an interesting, but not completely unfamiliar concept. It was based on a somewhat similar computer simulation, a game called Darwin. Darwin was developed in Bell Labs in 1961. It was devised by Victor Vyssotsky, Robert Morris Sr. and Dennis Richie. The first public description of that game appeared in the column of Software-Practice and Experience, Volume 2 in 1972.

Corewar has a history of more than 20 years of active competitions, periodical journals, tournaments, and more recently - quite common evolutionary approaches to overcoming the challenges inherent to some of the standard environments of its virtual worlds. There is more than one way to characterize it. In a way, it is certainly a game, though an unusual one. A programming challenge. Nevertheless, it has also been used in testing microprocessors and is suitable for exploration of many genetic programming strategies.

The popularity of CoreWar has dropped recently, along with the interest in assembly programming. However, there are still competetions and people
contributing to the CoreWar online community.

Most of the CoreWar competitions are set up as *hills*. Hills are fixed-size leagues. New warriors challenge the hill one at a time. After each challenge, the warrior with the worst score is removed from the hill. There are several standard environment settings, and at least one active hill for each of those variations.
3.1 Basic Concepts

So far, only the historical context of the CoreWar simulation has been given. In this chapter, the focus will be on some of the more basic game concepts and also the Redcode syntax. More subtle considerations will be given in section 3.3.

Redcode is a CoreWar standard since 1994. Previously, a somewhat more restricted language called Corewars had been used. Redcode is an assembly-like language. However, there are no registers in Redcode. All the values obtained in the calculations need to be memorized in the instruction fields (source, destination) themselves! Therefore, by the very design of Redcode, self-adjustment and mutation of the warrior code is not only possible, it is vital for survival.

Redcode is Turing-complete. That basically means that it can be used for all kinds of calculations. However, only few of the contemporary warriors use heavy calculations in determining their strategy. Most of the time, speed is preferable to logic.

The memory array where the warriors are loaded is called the core. The size of the core is predetermined in most of the standard settings. The most common hill configurations are given in table 1. Great diversity is more than apparent. Core size ranges from paltry 80 instructions up to 55440, and even larger cores have been used in some special tournaments. The differences in maximum size of the warriors are more important though. Although it may seem absurd that a competition exists where programs are only 5 instructions long, nano hill is set up for such a competition. Even more surprisingly, it turns out that even some of the more intelligent strategies can be coded in such a restricted space. P-space is a separate array that is sometimes enabled where only numerical values can be stored and it is persisted between the rounds. In other words, a warrior can deduce what happened to it in the past encounters and change its strategy to better confront the opponent. The result of the previous battle is always written in P-space at address 0.

All arithmetics in CoreWar is done modulo CORESIZE. This has some serious implications on the attack patterns which will be discussed later on. All addressing is relative, there are no absolute addresses.

Redcode syntax consists of 19 instructions, 7 instruction modifiers and 8 addressing modes. Each command consists of an instruction name, followed by the instruction modifier, A-field addressing mode, A-field value, B-field addressing mode and B-field value. A-field is the source and B-field is the destination of the operation performed by the instruction. Regardless of the simplicity, the language allows for much creativity in warrior design. There are many combinations of the aforementioned elements, more precisely
Table 1: Common hill configurations

<table>
<thead>
<tr>
<th>Parameter</th>
<th>94</th>
<th>94nop</th>
<th>94t</th>
<th>lp</th>
<th>94x</th>
<th>tiny</th>
<th>nano</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORESIZE</td>
<td>8000</td>
<td>8000</td>
<td>8192</td>
<td>8000</td>
<td>55440</td>
<td>800</td>
<td>80</td>
</tr>
<tr>
<td>NUMPROC</td>
<td>8000</td>
<td>8000</td>
<td>8000</td>
<td>8</td>
<td>10000</td>
<td>800</td>
<td>80</td>
</tr>
<tr>
<td>MAXLENGTH</td>
<td>100</td>
<td>100</td>
<td>300</td>
<td>200</td>
<td>200</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>MINDIST</td>
<td>100</td>
<td>100</td>
<td>300</td>
<td>200</td>
<td>200</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>NUMCYCLES</td>
<td>80000</td>
<td>80000</td>
<td>10000</td>
<td>8000</td>
<td>500000</td>
<td>8000</td>
<td>800</td>
</tr>
<tr>
<td>R/W restrictions</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Pspace</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

8512-CORESIZE², although a portion of the command set exhibits equivalent behavior but would be considered different by some instruction-comparing Redcode instructions and thus cannot be regarded as completely equivalent. More detailed description of the Redcode instructions is given below.

- **DAT** - Removes the process that executes it from the process queue. It is used to store data. The instruction modifiers play no role here.

- **MOV** - Copies the source to the destination. Depending on the modifiers, it can copy either the content of a specific instruction field or the whole instruction pertaining to the source field.

- **ADD** - Adds the number in the source field to the number in the destination field. Two additions can be done in parallel if the .F or .X modifier is used.

- **SUB** - Performs subtraction. The functionality is the same as in ADD.

- **MUL** - Performs multiplication. It’s not used as frequently as ADD or SUB, however. The reason for this is the fact that effects of iterative multiplications are far less predictable when done modulo CORESIZE than iterative additions.

- **DIV** - Performs integer division. In case of division by zero, the process demanding the execution of the instruction is removed from the process queue. This is another way of removing enemy processes, as an alternative to making them execute a DAT instruction.
• MOD - Gives the remainder of the integer division.

• JMP - The unconditional jump instruction, redirecting the execution to the location pointed at by its A-field. The B-field does not affect the jump, so it can be used either to store data, or to modify some other values via the use of incremental/decremental addressing modes.

• JMZ - Performs the jump, if the tested value is zero. If the modifier is .F or .X, the jump fails if either of the fields is nonzero. As in the jump instruction, the A-field points to the jump location. The B-field points to the test location. If the jump fails, the instruction following the JMZ will be the next instruction to be executed by this process.

• JMN - Performs the jump if the tested value is nonzero. Operates the same way JMZ does.

• DJN - Decreases the destination and jumps if the value is nonzero. The functionality is otherwise the same as in JMZ and JMN.

• SPL - Creates a new process and directs its execution to the source value. The B-field plays no role in its execution and can be used the same way as in JMP. The old process, being the one that executed the SPL is moved to the next memory location. The order in which these two processes are then executed is as follows: the new process is going to be executed right after the old process executes one instruction.

• SEQ - Skips the execution of the instruction at the next memory location if the source and destination are equal. If the instruction modifier is .F, .X or .I, and either of the fields is found not to be equal, the next instruction is the one to be executed. Along with CMP and SNE instructions it is used in fast modern scanners for enemy detection.

• CMP - Has the same meaning as SEQ. It is the remnant of the old Corewars language and only used for backward compatibility.
• SNE - Skips the execution of the instruction at the next memory location if the source and destination are *not* equal. The modifiers operate in the same way as in SEQ.

• SLT - Skips the execution of the instruction at the next memory location if the source value is lower than the destination value. It is often used in scanners to check if the observed location is somewhere in their own code or a potential part of the opponents code.

• NOP - This instruction does nothing.

• STP - Saves the source value to the destination in P-space, if the environment allows the use of P-space. It is mostly used to store data regarding the strategic choices in some of the battle rounds.

• LDP - Loads the value from the source in P-space to the destination in the core, if the environment allows the use of P-space. It is used mostly to recover data pertaining to the results of the previous battles in the same match-up.

Since only a modest number of different instructions is available in Red-code, at first glance it might seem as if it is not very expressive, but it is. And it owes its expressiveness to the enabled instruction modifiers and addressing modes. Short overview of the former is given in table 2. Once all these facts are taken into account it becomes apparent why it is so cumbersome to try and optimize a warrior respective to its overall hill performance. After all, it is not only the bare difficulty of the task at hand, but also the great variety of options that lies ahead, rendering the search space far too big for the mere casual tinkering of the given code to appear promising in the long run. Naturally, as programmers we tend to express our own ideas that we have at that particular moment, and are not that concerned with the suboptimality of the solutions employed. However, thinking about ways to improve the existing solutions, as well as frameworks for building better ones, or more robust ones, is certainly not without merit. It should be emphasized once again that there is more than just one approach to CoreWar. It is certainly a game, a challenge. If we were to observe it exclusively that way, all of these facts might seem superfluous. However, there is also a scientific perspective
Table 2: Redcode instruction modifiers

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.I</td>
<td>This modifier states that the action is conducted on the whole instruction, and used only when copying an instruction or comparing the content of two memory locations.</td>
</tr>
<tr>
<td>.F</td>
<td>Copying, or comparing two fields at the same time.</td>
</tr>
<tr>
<td>.X</td>
<td>Copying, or comparing two fields at the same time, A-field of the source to the B-field of the destination, and B-field of the source to the A-field of the destination.</td>
</tr>
<tr>
<td>.A</td>
<td>Moving, or comparing, the A field of the source to the A-field of the destination.</td>
</tr>
<tr>
<td>.B</td>
<td>Moving, or comparing, the B field of the source to the B-field of the destination.</td>
</tr>
<tr>
<td>.AB</td>
<td>Moving, or comparing, the A field of the source to the B-field of the destination.</td>
</tr>
<tr>
<td>.BA</td>
<td>Moving, or comparing, the B field of the source to the A-field of the destination.</td>
</tr>
</tbody>
</table>

there. One might even further consider the whole process of problem solving as being a two way street. It is not just that we are solving the problems given to us, but the problems themselves are teaching us new things about the methods that we choose to use and even motivate us from time to time to create new approaches so that we could overcome all the difficulties of those particular cases.

As it has been pointed out in the former part of this paper, all the addressing modes in CoreWar are relative, there are no absolute addresses visible to the warriors. There is a total of 8 addressing modes available. They are given in the table.

Before proceeding further, let us now consider the simplest of examples - the imp.

\[
\text{MOV.I $0, $1} \]

This is probably the simplest of all CoreWar warriors. So, what does it do? Let us examine it a bit closer, then. It consists of just one instruction, the MOV instruction. The modifier is .I, so it copies the whole instruction, not just the field values. If we notice the addressing modes and the instruction fields, it becomes quite apparent what it copies and also where. The addressing mode being used is the relative one, in both fields. The instruction copies itself one field down the core array. Since the next position in the core is precisely the one that is about to be executed next, the process
Table 3: Redcode addressing modes

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>This is the immediate addressing mode. In operations, it is interpreted as pointing to the current address of the instruction embedding it. That means that if it is encountered in the instruction that is being executed, in that particular moment it will be the same as if it were a relative mode with offset zero.</td>
</tr>
<tr>
<td>$</td>
<td>Ordinary relative addressing. This is the default addressing mode and if the addressing mode specification is omitted in an instruction, the compiler will interpret it as this default value.</td>
</tr>
<tr>
<td>*</td>
<td>This is the A-field indirect addressing mode. Points to the instruction $x + y$ locations away, where $x$ is the respective field value.</td>
</tr>
<tr>
<td>@</td>
<td>This is the B-field indirect addressing mode. Similarly, it points to the address where the B-field of the target instruction points.</td>
</tr>
<tr>
<td>{</td>
<td>Apart from specifying a location, these modes can also specify some action to be performed. This is the A-field predecrement mode. The value in the target A-field is first decremented by 1, and then it is the same as in case of the A-field indirect mode.</td>
</tr>
<tr>
<td>}</td>
<td>Very similar to the previous one, this is the A-field postincrement mode.</td>
</tr>
<tr>
<td>&lt;</td>
<td>B-field predecrement, analogous to A-field predecrement.</td>
</tr>
<tr>
<td>&gt;</td>
<td>B-field postincrement, analogous to A-field postincrement.</td>
</tr>
</tbody>
</table>

repeats itself and in the next cycle the next MOV instruction executes, copying itself further and further down the core. If not interrupted, this copying will continue ad infinitum. Of course, this is rather simplistic and not very effective, but this simple imp is still sometimes used in some specialist warriors in the LP setting, as a subcomponent of larger structures. In one of the following sections, imp structures will be considered in more detail. Spirals and rings of imps are even today one of the most fierce weapons in the game and also formidable defensive structures.
3.2 Warrior File Format

There is more to the files that are being sent to the competitions than just their Redcode code. There is also a variety of options that are provided and also meta data regarding the warrior itself. The basic structure of the Redcode file is given in Figure 1.

```plaintext
;redcode-hillkey
;name WarriorName
;class WarriorClass [optional]
;password WarriorPassword [optional]
;kill WarriorName2 [optional]
;author AuthorName [optional]
;assert FunctionalConditions [optional]
;strategy StrategyDescription [optional]
[points to the label of the first instruction to be executed]
org start
start ...
...
... [here is where the code is inserted]
...
...
end
```

Figure 1: Basic structure of a redcode file

The ;assert option is used to emphasize the conditions that must be met for the warrior to function properly. The hill key is used to determine the hill that is to be challenged by the submitted warrior and usually refers to the combat conditions as well. The author can choose to remove a warrior already on the hill via the ;kill demand, in case a valid password is given. If a ;password is not included in the warrior file, a password will automatically be assigned to a given warrior. The ;class label specifies a class to which the warrior belongs. That option was used for warrior classification. Anything else following a ";" will be regarded as a comment in the code.

3.3 Overview of Strategies

3.3.1 Imps

In each form of combat, there are two opposing principles - offense and defense. It has been suggested that in order to eliminate the opponent, a
CoreWar warrior must force all the opponents processes to execute either a DAT instruction or a zero division. However, it would be unwise, to say the least, if one would fail to set up a backup strategy to avoid losing when the attack fails. One of the most commonly used ways to achieve this is by including some kind of imp-like structures to accompany the main warrior code. These structures are almost never used alone in a warrior, because of their lack of offensive capabilities. However, the presence of imps in a warrior tends to increase its durability and provides resistance to certain types of attacks. Imps are, in general, structures of the following type:

\[ \text{MOV.I } \$0, \ istep \]

or

\[ \text{MOV.I } \#istep, *0 \]

the former being referred to as the B-field imps and the latter as the A-field imps. More than one process is required for these structures to function correctly. Not all values are suitable for the \textit{istep} value used in imp B-field or A-field, respectively. There has to exist a number \( N \) that satisfies 
\[ N \ast istep \equiv 1(\text{mod CORESIZE}) \]
and for the smallest \( N \) satisfying the equation, the imps in question are called N-point imps, or N-point imp spiral. For the proper functioning of N-point imp spirals at least \( N \) processes are required queued in the following order (according to their location in the core – assuming the imp MOV instruction is located at location \( loc \)):

\[ loc, \ loc + istep, \ldots \ loc + (N - 1) \ast istep \]

Of course, the array of processes in the queue can be longer, and it provides extra durability and resistance to the spiral in question. In the CORESIZE = 8000 environment which is the most widely used one, some of the numbers for imp steps are: 2667 (3-point), 1143 (7-point), 889 (9-point), 3077 (13-point). Generally speaking, all numbers not containing a prime factor that is also a factor of the \textit{CORESIZE} are imp numbers for that \textit{CORESIZE}. The most widely used are 3-point and 7-point imp spirals, because it does not take a lot of time to set them up and running. The A-field imps and the B-field imps, although similar, are not vulnerable to the same set of attacks, so they are often combined in larger warriors and used together. The imp spiral can be used in several ways.

In the vector launch method (Figure 2 a)), a desired number of threads is created, and a table of values is used to designate the place where the execution starts for each of those threads, respectively. Imp pump (Figure 2}
a) vector launch method

```
SPL.B 1, $0
SPL.B 1, $0
SPL.B 1, $0 ; creating 8 processes
SPL.B 1, $0
JMP.B @vt, }0
```

```
vt DAT.F $imp, $imp+istep
DAT.F $imp+2*istep, $imp+3*istep
DAT.F $imp+4*istep, $imp+5*istep
DAT.F $imp+6*istep, $imp+7*istep
```

```
imp MOV.I $0, $istep
```

c) additive launch method

```
SPL.B $1, $0
SPL.B $1, $0
SPL.B $1, $0
SPL.B $2, $0
JMP.B $imp, $0
```

```
vt DAT.F $imp, $imp+istep
DAT.F $imp+2*istep, $imp+3*istep
DAT.F $imp+4*istep, $imp+5*istep
DAT.F $imp+6*istep, $imp+7*istep
```

```
imp MOV.I $0, $istep
```

d) silk imps

```
SPL.B $1, $0
SPL.B $1, $0
SPL.B $1, $0
SPL.B 0, $istep+1
MOV.I }-1, }-1
MOV.I $0, $istep
```

```
pump MOV.I $impl+ioff, $impl+1+ioff
MOV.I {-1, >-1
```

```
leap JMP.B $((impl+2*(istep+1))+ioff),
{((impl+2*(istep)-1)+ioff)-500)
```

```
imp mov.i #istep, *0
```

Figure 2: Example codes of Imp warriors
b)) is used to set up the imp spiral and periodically add new threads to the spiral. There are several different types of imp pumps. By using the additive launch method (Figure 2 c)), one makes the decision to sacrifice some speed in order to reduce the size of the necessary code, because then a vector table is not needed.

There are many ways to set up an imp structure in a warrior. Most of them include SPL, MOV, ADD, SUB or JMP instructions. However, those instructions are often met in various kinds of attacking structures, so the best way to automatically detect the presence of imps in a warrior remains the search for A-field or B-field imp MOV instructions, described in the text above. [18]

3.3.2 Coreclears

The only way to ensure a win, in a sense of being positive that the opponent is destroyed, is to rewrite the whole core with process-killing instructions. Such a procedure is referred to as core clearing and the respective warrior parts are therefore called coreclears.

There are several types of coreclears, designed for different types of situations. In many cases, it is not enough to simply rewrite the core with DAT instructions, especially when confronted with multiprocessing warriors, most likely replicators (Section 3.3.4). In such a situation, it is common to first wipe the core content with SPL, slowing down opponents by making them create useless processes, thereby consuming virtual time given to them by the process queue manager. Even more often, a double SPL pass is used before switching to the DAT clearing phase. The coreclear use is strategy-dependent, of course. There are still some warriors using only a simple DAT-clear method. The matter of killing imps is also an important issue, resulting in special types of coreclears designed specifically to perform well on imp-killing tasks. Also, a multipass clearing method is preferred to the one-pass method employed in some older warriors. Here are the basic coreclear types:

The variety of coreclears is a natural consequence of diverse attacking strategies and opposing warrior types. If the opposing warrior possesses only one thread of execution, then a simple DAT clear should suffice. The problem arises in those situations when there is more than one thread active in an enemy warrior. There are, however, several ways to implement even a simple coreclear, such as a DAT coreclear (Figure 3 a)). One strong option is a so-called dirty clear (Figure 3 b)), which aims to increase its forward clearing speed by, in turn, copying DATs and decrementing instruction fields in the consecutive instructions in the core. Decrementing can sometimes also cause
a) simple DAT clear

```
 gate DAT.F $-10, $clrini
 b DAT.F $-10, $6
 clear SPL.B #0, $0
 MOV.I $b, >gate
 DJN.F $-1, {gate
```

e) bi-shot SSD clear

```
 gate DAT.F ini, ini+dist
 b DAT.F 18, <2667
 clear SPL.B #20, >6230
 MOV.I $clear, >gate
 MOV.I $clear, }gate
 DJN.A -2, <stun
```

b) dirty clear

```
 gate DAT.F $0, clrini
 b DAT.F $0, $6
 clear SPL.B #0, >gate
 MOV.I $b, >gate
 DJN.F $-1, >gate
```

f) SSMD clear for destroying A-field imps

```
 gate DAT.F $450, $0
 b DAT.F $-100, $0
 clear SPL.B #20, >6230
 MOV.I $clear, }gate
 MOV.I $clear, }gate
 DJN.A -2, }gate
```

c) SSD clear - simple

```
 gate DAT.F $0, clrini
 b DAT.F $1, $8
 clear SPL.B #dijnini, $8
 MOV.I *b, >gate
 MOV.I *b, >gate
 DJN.F $-2, }clear
```

g) spiral clear the state of the art imp-destruction component

```
 gate DAT.F $14, <2667
 b DAT.F $0, $0
 clear SPL.B #12, $1
 MOV.I $clear, }gate
 MOV.I $clear, }gate
 DJN.A -2, }gate
```

```
d) SSD clear - anti-3 point imp

 gate DAT.F $clear, clrini
 b DAT.F <2667, <5334
 s2 SPL.B #(b-gate), 9
 clear SPL.B #(s2-gate), $9
 MOV.I *gate, >gate
 MOV.I *gate, >gate
 DJN.F $-2, <gate-12
```

```
g) spiral clear the state of the art imp-destruction component

 clear ADD.A #382, $3
 MOV.I @-1, {2
 JMP.B $-2, $0
 SPL.B #1, #4
 DAT.F $-8, $0
```

Figure 3: Example of some Coreclear codes
some structures to malfunction, among which are also imps, which is very important. The field used for indirect pointing to a place for coreclear attack is called a gate. That is the case because imps surviving the initial coreclear attack are expected to arrive at some point at the gate and that is one other reason why dirty clear is so widely used - both decrementing and DAT copying affects the B-field of the gate, protecting the coreclear from B-field imps. SSD coreclears (Figure 3(c), d)) are also often a part of larger warriors. One field is usually used as a pointer to the instruction that is about to be copied in the clearing process. After a certain amount of time, that pointer is modified and another instruction is then copied by the coreclear. Some coreclears use DAT.F <2667, <5334 or DAT.F >2667, >5334 or some other imp-killing instruction.

Bi-shot coreclears (Figure 3(e)) basically operate in the same way as the usual ones, differing only by the fact that they copy instructions along two different paths in the core. It is customary to use the A-field and the B-field of the same gate instruction to point to those locations, because it makes implementation easier.

SSMD coreclears (Figure 3(f)) are a new way of dealing with a lot of imp spirals containing a lot of execution threads. Three different instruction types are used in those coreclears - SPL, MOV and DAT. The first is used to stun the opponent. The second is used to heavily disrupt the functionality of imps. The third is used to kill all the remaining enemies. Spiral clear (Figure 3(g)) attacks though the core along 21 different paths (in CORESIZE==8000 setting), and is specifically designed to destroy imp structures. The downside to using it is that a condition for its proper functioning is that only one thread remains in the warrior in question. All other coreclears are designed to be used by many threads at once, to increase their durability in case of some types of attack.

Generally speaking, each coreclear is a loop, so a JMP, DJN, JMN, or JMZ is almost always present, although there are some rare exceptions to that rule. Also, most coreclears include SPL, and all of them at least one MOV instruction, since it would otherwise be impossible to perform instruction copying. However, this does not distinguish them from some other warrior types. [18]

3.3.3 Stones

One of the first strategies developed was to simply copy DAT instructions over the core following some sort of a plan. Up to this moment, many alternate approaches were devised, resulting in warriors copying other instructions as well, not only DATs. The intent of such an approach is to either stun the
opponent warrior by forcing it to execute SPLs and generate unnecessary processes, or redirect some of its processes to performing tasks helpful to the attacker. However, DAT stones are still used in hybrid warriors, either followed by imp spirals or replicators (Section 3.3.4). The name of this warrior group was given according to the stone/paper/scissor analogy, since there were not as many warrior types at the beginning of the expansion of the CoreWar community, and most of them were able to fit into one of three clusters, each scoring well against one, and not so well against the other. The term paper is used for replicators (Section 3.3.4), and the term scissor for most scanners (Section 3.3.5) and SSD coreclears (Section 3.3.2). There is a great variety of stones being currently used in active warriors, here are some of them:

a) Carbonite-style stone

```
stone SPL.B #0, $0
MOV.I $85, $86
ADD.AB {0, }0
DJN.F $-2, <-2785
FOR 82
DAT.F $0, $0
ROF
DAT.F >-1, >1
```

b) anti-coreclear stone

```
stone SPL.B #2*sstep, <2*sstep
MOV.I $ini, $ini+sstep
MOV.I $sptr, *-1
;hit with DAT >-1, >1
ADD.F $stone, @-1
DJN.F @-2, <ds
DAT.F $0, $0
DAT.F $0, $0
DAT.F $0, $0
sbomb DAT.F >-1, >1
```

Figure 4: Code examples of Stone Warriors

There are many stone types used in contemporary warriors. Carbonite-style stone (Figure 4a)) is often used in P-space warriors, less often in hybrid non-P-space warriors. ADD.AB {0, }0 adds 85 to the B-field of the MOV and in each loop iteration changes the target locations address by 85. Eventually, the ADD is overwritten by DAT.F >-1, >1. A small DAT-coreclear is thus created via intentional mutation of the original code. Each time a mentioned DAT is executed at that location, it increments the B-field value of the MOV, and the whole structure acts as a forward coreclear, copying DATs at consecutive core locations. DAT >-1, >1 is more powerful than it might seem. It is often used with intention of disrupting coreclears, since it could, in many cases, alter the fields in the coreclears MOV instructions, so those would no longer point to the gate, and the coreclear could, in such cases, even rewrite itself with useless code, resulting in either losing its functionality or being destroyed.
The instructions used in stones are: SPL, MOV, some of the jump instructions, and ADD, SUB or MUL. DJN is most widely used, when it comes to choosing the jump instruction, since it can be used for some decremental attack. Arithmetic instructions are also a necessity. MUL is not so frequently used. [18]

3.3.4 Replicators

One of the basic and most effective strategies in CoreWar follows the logic that in order for the warrior to survive, it should create many processes and let them operate on many copies of the main warrior body, therefore ensuring that some of those copies will survive an enemy attack, since it takes a lot of time to destroy them all, and, in the meantime, destroy the enemy process. Sometimes, several replicators are included in a single warrior. These components are almost always accompanied by quickscanners, to increase their attacking capabilities. Some other hybrid strategies are very common, including stone-papers and paper-imps. All of the newly made replicators use the so-called silk-style copying engine. Here are some of the frequent designs:

The silk-style copying engine is really simple, but offers great copying speed. At first, new threads are created and their starting points are set to be where the new copy will be placed. Before the execution of those threads begins, the replicator instance is then copied to the designated place. And then the whole process begins again. This is achieved by SPL/MOV instruction pairs or triplets, depending on the number of threads executing a single instance. All replicators also have attacking options, varying just as in any other warrior type and depending solely on the choice of the warrior author. The replicator instances cannot exceed certain size, because that would make them more vulnerable to all attacks, and reduce the rate at which they spread through the core, since copying would take more time. Other than that, replicators can also contain other small attacking or defensive structures, including coreclears, stones or imps.

There are, generally, many ways to create self-replicating code. However, in recent years, only silk-style replicators are being created, consisting of two or three SPL/MOV pairs or maybe one SPL/MOV/MOV triplet, possibly a MOV/JMP, MOV/DJN, MOV/JMZ or MOV/JMN ending and some attack structure. Ordinary replicators, therefore, use the same basic instruction set as coreclears, which makes automatic type recognition in such cases a little difficult, especially since there are such types like coreclearing papers. [18]
Black chamber paper

boot  SPL.B $1, $0
SPL.B $1, $0
SPL.B $1, $0
MOV.I {p1, {divide
divide SPL.B (p3+1+4000), }c
p1  SPL.B @(p3+1), }ps1
MOV.I }p1, >p1
p2  SPL.B @0, }ps2
MOV.I }p2, >p2
MOV.I #bs2, <1
SPL.B @0, {bs1
MOV.I {p2, {p3
p3  JMZ.A $ps3, *0

Moore-style paper

used in the Water Dragon warrior

boot  SPL.B $1, $0
qtab2 MOV.I $-1, #0
SPL.B $1, $0
p1  SPL.B @0, }ps1
MOV.I }p1, >p1
MOV.I }p1, >p1
p2  SPL.B $ps2, {cpy
MOV.I }cpy, }p2
MOV.I $pbomb, >bs1
cpy MOV.I $p2+numproc, }p2
JMZ.F $p2, *cpy
pbomb DAT.F <2667, <5334

SPL carpet resistant replicator

from Stylized Euforia warrior

boot  SPL.B $2, $0
SPL.B $2, $0
SPL.B $1, $0
MOV.I {3, {2
MOV.I {2, {1
SPL.B $3352, }ff-1429+2
SPL.B 010, }-782
MOV.I }-1, >-1
MOV.I }-2, >-2
SPL.B @0, {2023
MOV.I }-1, >-1
SPL.B @0, }1879
MOV.I }-1, >-1
ff  MOV.I #-3399, }-1429
MOV.I #2908, }-3819
MOV.I #1169, }772

Coreclearing paper

boot  SPL.B $1, $0
SPL.B $1, $0
SPL.B $1, $0
SPL.B @0, }ps2
MOV.I }-1, >-1
SPL.B @0, }ps2
MOV.I }-1, >-1
SPL.B @0, <ps3
MOV.I }-1, >-1
MOV.I #1, <1
DJN.F $-1, #bs

Figure 5: Example code of some Replicators
3.3.5 Scanners

Corewar programs that try to discover the location of enemy code and then start an attack at that location are referred to as scanners. There are several basic scanner designs: SPL/JMP scanners, fixed carpet length scanners, adaptive carpet length scanners, coreclear directing scanners, oneshots, twoshots, and several others. Since the scanner attack has a greater probability of succeeding, due to the intelligent choice of target location, such warrior is usually able to spend more time on the attack against that location. Basically, scanners try to locate the opponent through the extensive use of the comparative instructions, namely SNE, SEQ, JMZ and JMN - searching for the nonzero fields among the unknown core instructions or for the instructions that differ from the initial DAT.F $0, $0 to which the core is initialized at the beginning of each combat simulation. To battle scanners, some warriors use well placed decoys, or separated code placement, reducing the size of the largest code component. Stones are the natural enemy of scanners, since they copy many instructions all over the core, none of which is a part of their code. These instructions trigger the scanner’s attacking sequence, causing the scanner to waste a lot of time. Some recently developed scanners use an anti-stone trick to recognize such instructions and skip the attack, if necessary. Of course, this leads to an increase in size, making the scanner more prone to some other kinds of attack. Here are some of the well known scanners:

Deathstar (Figure 6 a)) is probably the most successful SPL/JMP using scanner of all times and had been written by Roy van Rijn. The coreclear phase is triggered by a self-hit on the ADD instruction. HSA (Figure 6 b)) is one of the best known adaptive carpet length scanners. In the first phase, it scans through the core, and when it locates something other than a DAT.F $0, $0, it starts the attack, copying two SPL instructions at a time, and constantly checking if the next instruction to be copied at is DAT.F $0, $0 or something else. If nothing is found, the attack is stopped and the scanning mode is reinstated. By using the SLT instruction, it is able to detect if the scanning pointer actually point to its own code, and by counting the number of such self-scans, after some time, it changes its behavior - it starts copying DATs instead of SPLs, which is an alternate endgame approach to core-clearing. ()( Figure 6 c)) is a simple, yet powerful, oneshot with an anti-stone trick. It attacks the first thing it detects in the core, if it determines that it is not a single instruction, which would likely be a result of the stone attack. Only if there is more than one instruction at the location in question, the attack is triggered, and it is a standard SSD coreclear.

Since the scanners compare either whole instructions or simply instruc-
Deathstar
(main component of the warrior)

ADD.AB #3094, @2
JMZ.B $-1, @2
MOV.I $5, @1
MOV.I $2, *-2
JMN.B $-4, *0
SPL.B #0, <0
MOV.I $6, >-3
JMP.B $-1, @0
FOR 4
DAT.F $0, $0
ROF
DAT.F <2667, $8

HSA
(main component of the warrior)

DAT.F $100, $4096
FOR 5
DAT.F $0, $0
ROF
MOV.I $14, <-6
MOV.I >-7, >-7
JMN.F $-2, >-8
SUB.X #12, $-9
SNE.I *-10, @-10
SUB.X *3, @-2
JMN.F $3, @-12
JMZ.F $-4, *-13
MOV.X @-5, @-5
SLT.B @-6, #27
DJN.B $-10, @-7
DJN.B *-3, #13
JMP.B *-4, }-12
DAT.F $0, $0
SPL.B #1, {1

Figure 6: Example codes of some Scanner warriors
tion field values, SNE, SEQ, CMP, JMZ or JMN are always present, and so are arithmetic instructions: ADD, SUB or MUL. Coreclears are usually a separate component in those warriors, but they can be omitted, which is the case in many adaptive carpet length scanners, or simply created during the simulation via self-mutation, and therefore not be detectable in syntax analysis of the starting code. [18]

3.3.6 Hybrid Strategies

All of the aforementioned strategies represent a different approach to winning the game. They are simple and effective, but also, in a way, a bit extreme. Committing a great deal of resources to a single plan, however fruitful in some particular cases, might also prove fatal in some situations. All of the basic strategies have their vulnerabilities. Replicators are the most solid basic strategy, but they can still easily be defeated by good scanners or coreclears. That is exactly why the hybrid approach yielded so many good warriors in recent years, stressing out robustness instead of placing all the hopes into the fashion of the period and suitable strategic hill composition. There is no need to go into more detail here. P-space warriors deserve to be mentioned as a true mix of various strategies, governed by a common p-switch, a decision structure implementing some form of finite automation. Since the use of P-space isn’t allowed in the 94nop setting, other forms of strategic combinations exist there. Basically, all forms of cooperation are possible, not all of them producing equally stable results in the long run. However, there is still much room for experiments in that subfield of warrior creation.
4 Optimization

The order in which the operations shall be performed in every particular case is a very interesting and curious question, on which our space does not permit us fully to enter. In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selection amongst them for the purposes of a Calculating Engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation. Ada Byron’s notes on the analytical engine 1842

Before we plunge into a discussion involving concrete, practical methods for performing specific optimization tasks, some general remarks concerning optimization would be noteworthy. Let us consider a simple system. The system receives some input data and produces some results as its output, implicitly defining a mapping from the input space onto the output space. If all three parts of the system are previously known, there is basically no need for any kind of scientific inquiry. On the other hand, if any of the system parts is unknown at some point, we could try to deduce it from what is known. Let us consider what those three elementary cases represent. If we are given input data and a transformation, we can easily generate the output and these problems inspire the discipline of simulation. Naturally, the transformation at hand might also be some sort of a hypothetical model devised by scientists with the intention of comparing the generated output to some real-world data. Based on those comparisons, the given model might be either rejected or accepted as a satisfactory approximate solution to the respective real world phenomenon. In another case, the model itself is precisely what we seek to deduce from the input and output data and this is the field of research of data mining. Data mining is a novel field basing its techniques on various machine learning, probabilistic, statistical, information theoretical and numerical methods. The remaining case is when we possess the desired output data and the mapping from input to output space. In those rare cases where the transformation in question is simple, we can simply calculate the inverse transformation and directly solve the problem. On the other hand, it is much more common that the mapping is really complex and that some other methods must be employed to resolve the issue. The size of the input space usually makes it infeasible to perform an exhaustive search, so some form of approximate solution to the problem is sought for instead. This is precisely what optimization is all about. The usual case
revolves around function minimization/maximization. Sometimes it is possible to find the global optima and exact solutions, but in most real world situations, optimization only provides us with some near-optimal ones. The probability of finding proper solutions depends on the convexity/concavity of the local areas of the search space. Concave regions can have multiple local (see figure 7) optima and most algorithms have no way of making a clear distinction between those and the global one.

Practical mathematical theory of optimization emerged in the second half of the twentieth century, boosted by the growth of computer industry and hardware/software that could be used to implement the mathematical solutions and automate the search process. Every new generation of computers enables solving new types of problems and requires new methods to be devised. The optimization theory has become increasingly important in modern engineering, planning and decision making. Data mining can also be viewed as implicitly implementing some sort of optimization procedure in the hypothesis space, while searching for the models that best describe relationships between the data.

There are many subfields of optimization theory. Some of the most important are linear programming, quadratic programming, nonlinear programming, convex programming, semidefinite programming, stochastic programming, constraint satisfaction, combinatorial optimization and trajectory optimization. Linear programming problems can be solved in polynomial time, but even some of the quadratic ones are NP-hard problems, i.e. at least of NP complexity. When looking for extrema, there are two things to consider. In most real world cases, search space is really big and only a small fraction of it can be thoroughly examined. Therefore, a good algorithm must provide some means of search space exploration, thus ensuring that many regions of the search space will be examined by it. The other factor is exploitation, pertaining to how much the algorithm searches in the critical regions where it detects the possibility of extrema. To sum it up, the goal is to first search the state space for critical regions and then intensify the search in those areas where there is a higher probability of finding solutions to the initial problem. [17]

4.1 Optimization Methods

As was previously stated, many different optimization methods exist, each one of them being advantageous in some and disadvantageous in other types of data spaces. In the following sections, the algorithms upon which the implementations in the CoreWar Optimizer are based will be thoroughly discussed.
4.1.1 Hill Climbing

It’s always further than it looks. It’s always taller than it looks. And it’s always harder than it looks. The 3 rules of mountaineering.

There are very complex optimization approaches, but there are also simple ones. Hill climbing is one of the latter. The algorithm starts considering a random state and then proceeds by examining its neighbours in the search space and chooses the first one that is better than the current state. A slight modification to this approach would be to examine all the neighbours of a given state and choose the best one. Both approaches suffer from the fact that they mostly lead to closest local optima if that is nature of the search space. That is why hill climbing is a local optimization method. On the other hand, after discovering some local optimum, the algorithm can be restarted with another random state, so that it performs a local search in another part of the data space. This variation is still used in practice, sometimes even producing results of high quality. The main characteristic of the hill climbing algorithm is that it emphasizes exploitation, completely disregarding exploration of the search space. [10]
4.1.2 Simulated Annealing

Everyone by now presumably knows about the danger of premature optimization. I think we should be just as worried about premature design - designing too early what a program should do. Paul Graham

Undoubtedly, the greatest downside of the hill climbing algorithm is its inability to escape the slopes leading towards local extrema. The most natural improvement to the technique employed there would be to somehow increase exploration and reduce exploitation in the algorithm, at least on average basis. It will become apparent in the next few passages that it is precisely what simulated annealing is all about.

Annealing is a processing technique in metallurgy used to alter some of the properties of a material. It is usually used to induce ductility, refine the material structure and increase the internal dislocation density which leads to strengthening of the material in question. Annealing is performed via heating and maintaining the desired high temperature and then cooling the material. In the heating phase, the material softens due to the fact that existing dislocations disappear and in the cooling process they are formed again by recrystallization. In the process, atoms go through continuous rearrangements, moving towards a lower energy level. The cooling also results in the atoms losing their mobility and the state of the material slowly converges to some energy minimum. However, the decrease in energy is not monotonic, there is a possibility of slight temporary increases in the energy of the current state. If the energy function has local minima, this is a desirable feature and this is where the inspiration for the simulated annealing algorithm is found.

One of the first applications of the insights gained from observing the influence of heat on metals was a stochastic technique simulating the behavior of a system of particles approaching thermal equilibrium. In that procedure, energies of the different particle configurations were compared to each other. The simulated annealing algorithm starts at some random state $s_0$ and proceeds in the following fashion: assume that the last observed state is $s_i$; observe some state $o_i$ drawn at random from some predefined neighbourhood of $s_i$; if $f(o_i) < f(s_i)$, the algorithm moves to the new state, i.e. $s_{i+1} = o_i$; if the new state does not further reduce the function value, there is a positive probability of transition to the new state which depends on $|f(s_i) - f(o_i)|$ and a complementary probability of remaining in the same state. The transition probability also depends on the system temperature, which is a generic parameter introduced to ensure convergence in the latter phases of the algorithm run. In the beginning, the temperature value is set
to some high value and it decreases to some predetermined value to simulate the cooling phase of the system in the annealing process. This temperature time-dependency follows some user-defined law, though it is customary to use the formulas that reflect the nature of the thermodynamical process that gave birth to this optimization approach. The transition probability to the higher energy state is usually given by equation [1]

\[ p(s_{i+1} = o_i | f(s_i) < f(o_i)) = e^{-\frac{(f(o_i) - f(s_i))}{T}} \]  

Of course, there has to exist some sort of stopping criterion, either defined via some function value threshold or the rate of overall decrease or perhaps even the number of iterations (if some constraints are being imposed on the execution time). The basic algorithm can also be slightly extended if we do not consider just one, but a few neighbouring states prior to taking the best one as \( o_i \) and only then performing the probabilistic transition step. Obviously, the simulated annealing procedure in the end reduces to a greedy algorithm and functions as as the steepest descent hill climbing algorithm. It is still quite easy to implement it and it offers better chances of getting closer to the global optima. [14]

There is one similar method that is well worth mentioning - quantum annealing. In the thermic annealing procedure which was referred to as simulated annealing in the former passages due to historical reasons, the sampling in search for a better state is done from some vicinity of the current point. The specification of such a neighbourhood is usually predetermined and application specific. Instead of using temperature to determine transition probabilities, quantum annealing relies on the so-called tunneling field. Tunneling field basically determines the width and shape of the neighbourhood function and changes over time. In the very beginning, it encompasses the entire data space and in the end it reduces to containing only the nearest neighbours. As a consequence of that, in the starting phase of the algorithm run, potential candidates for the next state are being sampled from all over the data set which emphasizes exploration and as the time goes by, the algorithm focuses more and more on satisfying the exploitation criterion, thus reducing itself to basic hill climbing, analogously to the thermic algorithm. This procedure can actually be advantageous over the thermic approach in some specific situations. Assume that there are multiple local minima surrounded by some very high barriers. In such situations it is highly unlikely that the thermic algorithm would be able to direct the search to other areas of the state space once it enters the neighbourhood of such a minimum. If the function landscape better fits the latter description then quantum annealing is to be preferred. [16]
4.1.3 Particle Swarm Optimization

*What is not good for the swarm is not good for the bee.* Marcus Aurelius Antoninus

In the recent years, it has become a common practice to search for patterns and ideas for solutions of some practical engineering tasks in a nature-inspired way. After all, nature has had the time to evolve good solutions that fit well to the environmental conditions, so it is no surprise that so many of the techniques in use today are nothing more than an attempt to copy some natural mechanism that has been around for a long time, right in front of our eyes. The algorithm that is presented briefly in this chapter is one of such approaches.

Flocking of birds and schooling of fish have been an object of research of many biologists. With computers becoming available, scientists shifted the focus of their attention from more traditional research methods towards the art of computer simulation. Consequently, models of such animal group behavior had been created, mostly depending on inter-individual distances and assuming that the animals tended to preserve some sort of a formation. However, the behavior of such groups tends to be highly unpredictable, they suddenly change directions, scatter, regroup, perform various evasive maneuvers if attacked, etc.

Since such a behavioral pattern has persisted until present time, one must ask oneself why it is that these animals do it. It certainly has to be some sort of an evolutionary advantage. Apart from scaring off the predators, these forms of primitive social behavior in some animal species are also related to the search for food and water. If these resources are distributed unevenly and irregularly across the area, then sharing that valuable information is highly beneficial to the individuals of the population and increases their chances of survival. Now we have come to the point when the potential for applying this principle in optimization becomes quite apparent. Just as these animals search for resources in groups and share the information they obtain, we could use groups of objects in the data space to simultaneously examine points in search for the function optima. This way, since the individual particles of the swarm will be distributed across wide areas of the domain, it is to be expected that the results obtained in this fashion lie closer to the global optima than in the cases of hill climbing and simulated annealing which can be interpreted as special one-particle cases of this more general algorithm. [9]

Particle swarm algorithm shares many similarities with evolutionary computation techniques including the (genetic algorithms) described in section 4.1.4. The process begins with a random initialization of particle population and
proceeds by updating generations until it finds a satisfactory solution. The main difference is that the particle swarm algorithms have no operators to perform mutation, crossover, etc. Some comparisons between the techniques were made and it has been concluded that even though both the genetic algorithms and PSO achieve solutions of similar quality, the time required for such solutions to be obtained differs significantly between the two. Particle swarm algorithm which was tested was shown to be much faster on average in its convergence to good solutions and that is certainly an important quality. \[13\]

Regardless of its obvious advantages over both hill climbing and the simulated annealing methods, PSO still suffers from the fact that it sometimes tends to converge too quickly to some of the more suitable local optima and once it reaches such a state, particles remain in the vicinity of the discovered point. This has been a major issue and in order to resolve it, some variations of the algorithm have been proposed. A common approach was to introduce some kind of inertia weights, similar to the temperature factor in the simulated annealing. \[15\] The one that will be presented next is the one that was implemented in the CoreWar Optimizer.

Following another biological metaphor, a simple method of maintaining diversity in the swarm population regardless of the algorithms phase and the quality of the current solution, was devised. This was achieved by introducing a special kind of particle in the population and this particle will be referred to as the predator. The purpose of the predator is to chase after its prey, i.e. all the other particles in the swarm. Other members of the swarm are able to detect the predator particle as it approaches and turn away from it. What this basically means that if the swarm converges to some local suboptima, the predator particle will also head in that direction and force all the other particles to disperse and continue exploring the state space at some other location. \[1\]

There is only one predator in the swarm and it is chasing after the best individual. The behavior of the predator is described by the equation \[2\]

\[
V_p(t) = \phi_4(X_g(t-1) - X_p(t-1))
\]

\[
X_p(t) = X_p(t-1) + V_p(t)
\]

\[2\]

$V$ denotes the velocity and $X$ the position of the particle at a given time. Index $g$ is the index of the best individual in the swarm at that moment. $\phi$-factor is a random number which determines how quickly the predator approaches its prey. When the predator approaches some other normal particle, it induces fear and the particle moves away from the predator. The behavior of the other particles is given by equation \[3\]
\[ V_{ij} = wv_{ij}(t - 1) + \phi_{1ij}(p_{ij} - x_{ij}(t - 1)) + \phi_{2ij}(p_{gj} - x_{ij}(t-1)) + \phi_{3ij}D(d_i) \]

\[ x_{ij}(t) = x_{ij}(t - 1) + v_{ij}(t) \] (3)

The factor \( w \) is a linear decreasing inertia weight. All the \( \phi \)-factors represent random numbers. The best position that the \( i \)-th particle has found up to the current moment is given by \( p_i \) and the index \( j \) in the above given equation represents the \( j \)-th dimension of the particle vectors. The distance from the \( i \)-th particle to the predator is given by \( d_i \) and \( D \) is the exponential decreasing distance function defined as:

\[ D(x) = ae^{-bx} \] (4)

The influence of the predator grows exponentially with proximity. That feature of the above formula ensures that the effect of the predator will be negligible most of the time but will tend to increase dramatically as the swarm begins to converge. Convergence of the swarm is conditioned by the inertia parameter and the presence of the predator brings the benefit of maintaining population diversity, thus balancing exploration and exploitation.

It has experimentally been proven that this algorithm performs better than the original PSO algorithm. Even though that it has only been a decade since the first papers discussing this optimization approach were published, the PSO has become increasingly popular and a subject of many research projects.

4.1.4 Genetic Algorithms

"Observe constantly that all things take place by change, and ac-
custom thyself to consider that the nature of the Universe loves
nothing so much as to change the things which are, and to make
new things like them." Marcus Aurelius Antoninus, Meditations
(ch. IV, 36)

Genetic algorithms are a stem of evolutionary computing and the phrase is encountered in the literature interchangeably with evolutionary programming, genetic programming and evolution strategies. These algorithms draw inspiration from the process of natural evolution. Using evolution as a method for achieving improvements is quite reasonable, given the undeniable manifestation of its power reflected in the diversity of species populating the world we live in, tailored for the conditions present in their niches. The fundamental
The metaphor of evolutionary computing is encountered in a particular style of problem solving - that of trial and error - which would be expressed as generate and test in terms of evolution. The basic unit of evolution is a population. Algorithms based on evolution rely, in turn, on collections of candidate solutions. The population consists of many different individuals that strive for survival and compete in the conditions imposed by the environment. In the sense of evolutionary computing this means that objects in the collection are put up against the constraints of the task at hand. In nature, such a competition results in survival of the fittest, which was identified by Darwin and his contemporaries. The other driving force of the whole process lies within the wide spectrum of phenotypic variations in populations. Phenotypical traits greatly define response of an individual to the environment. Each individual is nothing but a unique combination of these traits and is being evaluated by the environment. Thus, the semantics of the mentioned phenotypic combination determines the fitness of each individual. The cornerstone of providing so many variations in the population is the process of mutation that occurs when generating offspring. When the new generation ascends, the process continues. The evolutionary pressure depends on many factors ranging from population density, scarcity of resources, predators, parasites, pathogens, natural disasters, pollution, etc.

There is duality here that needs to be emphasized. Even though the phenotype is what defines how well an individual fits the environment it is the respective genotype that encodes the phenotype. It is quite clear how this works in nature, but it raises a few questions when applied to optimization tasks. More specifically, some sort of representation for the objects of the domain space is required and such a representation must be suitable for being subjected to the genetic operators. Naturally, several traits could be represented by a single gene (pleitropy) and in turn, more than one gene might be required to encode some specific trait (polygeny). So, the mapping is not injective in either direction. Genes (sometimes referred to as loci) are also composite objects and comprise a set of alleles. The entirety of the search is performed in the genotype space.

Let us now observe evolution as an optimization process. We could think of adaptive landscape as a plot relating all the possible combinations of traits to their environmental fitnesses. A given population is basically a set of points somewhere on this landscape. The population tend to advance to the higher areas on this landscape in the course of time. It must be mentioned though, that evolution is not a strictly uphill process (due to the definition of the fitness, this process always searches for the maxima of the fitness function) due to the randomness of the choices and the finite size of the population. Another phenomenon of importance is what we refer to as genetic drift. It
occurs when highly fit individuals are lost from the population and some trait combinations are eliminated from the population gene pool. In those cases, it is even possible for a population to be forced towards the low-fitness areas on the adaptive landscape.

When implementing a genetic algorithm, there are a few things that must be previously defined. The matter of choosing a good representation was already discussed. First of all, one needs to carefully choose the fitness function. Sometimes, if used in some classic optimization task, fitness of a genotype is trivially constructed from the function that needs to be optimized and the genotype-phenotype mapping. Once that is set, one must choose the desired population size and desired diversity. If the island model is used, then the same choices must be made for several populations and probabilities of mating of individuals from various subpopulations must also be set. For the process to start, there has to be some sort of a parent selection mechanism and the genetic operators must be properly defined. After the new offspring is produced, a survival selection mechanism is applied to the union of the old and new groups of individuals. Of course, good initialization can greatly reduce the times required for obtaining good solutions. The algorithms can run until being forcefully stopped, however it is useful to define some sort of stopping criterion. Actually, there is hardly any need to run the genetic algorithms for a very long time, due to the nature of the time dependency of the best individual’s fitness in almost any population. It has been empirically shown that one of the basic features of the GA is that it tends to find good suboptimal solutions rather quickly and then slowly progresses towards the better ones. In other words, most of the improvements are done in an early stage of the algorithm run. [7]

Over the years, the GA have been widely accepted and it was believed that they offer great robustness and can be applied to a wide range of problems while preserving their effectiveness. However, it has recently been shown that all nonrevisiting black box algorithms exhibit the same performance, when averaged over the set of all possible problems. [22] This means that if we are using some general GA which has not been tailored specifically for the problem at hand, there is absolutely no guarantee that it will perform better than random.
4.2 Optimization In CoreWar

The nature of the competition in CoreWar evolved over time with the game itself. In those early days unconstrained creativity flourished, new warrior concepts were being proposed quite often and this period is marked by some simple warriors utilizing its strategic superiority to elevate their scores and reach the top of the hills. After a while, new strategies became a rarity, though still emerging every now and then. The objective was now to determine how to improve and refine the existing concepts and ideas. This was the time when many new microcomponents had been developed and combined in new ways. Field values for scanners, stones and replicators were still chosen by some educated guess, based either on some mathematical assumptions or visual criteria, the latter a consequence of the development of test environments having visual displays of the core. However, it was noted in one of the previous chapters that the space of all possible warriors is simply huge, even in the basic environments. There have also been attempts with some much larger cores to allow more complex warriors and this has also been a theme of some tournament rounds. Unfortunately, these experimental settings failed to reach the popularity of those where the years of merciless competition had already provided enough strategic diversity to pose a true challenge to a contestant.

There is no doubt that we have got some prejudice regarding the possibility of computers generating program code. Those prejudices are driven by the blind faith in the superiority of human reasoning and a tendency to denounce the machines a feature of being creative. Nevertheless, it was not long before the first automatically-generated warriors were made. There have been early breakthroughs, evolved warriors that were able to defeat some of the human-coded ones in the competitions. The approach was soon recognized by the community as a powerful tool in some of the environments. Even in the present, a great majority of warriors in the nano hill was not coded by human contestants. Several evolvers were made and used to generate these warriors via genetic algorithms.\[8\]\[5\] The strategic diversity of the CCAI evolved dataset \[21\] has been examined by the author and it was concluded that this particular set lacked diversity, which is one of the major problems in evolving CoreWar warriors.\[12\]\[19\] This phenomenon is related to the asymmetric distribution of mutation resistance among different strategies. Replicators and coreclears are thus most frequently found in almost all evolved warrior sets. More complex strategies rarely emerge, but there are some notable exceptions, e.g. White Noise generated by the MicroGP Collective.

The evolutionary approach was utilized when there was a need to create
either whole new warriors or some new warrior components. There is a completely different level of optimization, related to much smaller but nonetheless important refinements to the warrior code. Once the instruction skeleton of a warrior is set, there is always the question of how to determine some crucial field values. These values can represent scanning/bombing/replicating steps, bootstrap distances, decoy placement, separation between the warrior components, number of cycles before self-mutation, number of positive scans before switching to coreclear mode, etc. Up to this point, the only tool available for this kind of optimization was Optimax. [23] Optimax randomly samples the search space, generates temporary warriors for each sampling and runs them against user-specified benchmarks. Even though it might appear that this is a completely arbitrary solution, since there is no guarantee that good values will be found even after hours of optimization, Optimax introduced a really important idea for speeding up the optimization. The benchmarking is being done in several phases and users can define a different score threshold for each separate phase. The warrior that is being optimized is at first confronted by a single opposing warrior. If it scores higher than a threshold, it goes to the next phase where it has to fight warriors from several strategic groups. The selection of these groups considerably affects the overall optimization performance and care must be taken not to eliminate good warriors too soon in this phase by setting some inappropriate threshold. Only if the warrior passes the two mentioned phases it is allowed to enter the final phase and fights the entire benchmark. Performing all the fights is really time consuming and this approach of basically testing if a warrior is good enough to be a candidate for a good solution enabled better optimization in shorter time intervals.

The CoreWar Optimizer is the next logical step in this succession of software tools for warrior generation and optimization. It offers more possibilities for traversing the state space while searching for better field values and has a greater potential for finding good solutions. All the options of this new tool are given in the next section.
5 CoreWar Optimizer

CoreWar Optimizer is a new tool for CoreWar programs development and optimization. It has been implemented in Java 1.5.0. The code could have also been written in any other general purpose language, but Java was chosen for the task. Java dates back to 1995 and has become increasingly popular since. It is platform independent and object-oriented. It would be pointless in this context to discuss the overhead arising from the fact that it is an interpreted language. It is true that one of the main objectives of the CoreWar Optimizer is to increase the speed of warrior optimization, however - almost all of the execution time is spent on MARS execution and the contribution of the Java-based components to the total time of the optimization run is negligible.

In the following sections, all the most important features of CoreWar Optimizer will be introduced.

5.1 Functionality

The purpose of CoreWar Optimizer is rather straightforward. Therefore, the functions that are at user’s disposal represent those options that are of importance to the CoreWar warrior optimization. More details on how those functions are used will be given while describing the application GUI.

Corewar Optimizer enables a user to edit the desired warrior and specify the variables for optimization (for more details, see section 5.2), choose the optimization parameters, specify an optimization method, run the optimization, observe the updated statistics while the optimization is being run. The tool also persists the results in HTML format and also persists 20 best candidate solutions up to that point. All these files are being updated as time goes on. That would be a short summary of what the application does.

The speed-up trick used in Optimax is also used in CoreWar Optimizer. There are three phases for testing of each candidate solution. It is first being put up against a single warrior, then against the subset of offered strategic groups. If it proves to be satisfactory, the candidate solution is then passed on to the final stage where it is confronted by the entire benchmark. There is no redundancy in calculations, since CoreWar Optimizer only performs those match-ups in the final stage that haven’t already been performed in phases 1 and 2 and then recalculates the score based on the 3 partial scores.

As it was previously stated, the application has a GUI and that is the default way of using it. All the relevant methods have been made public, so it would also be possible to incorporate CoreWar Optimizer in some other
application and use it as an auxiliary tool in the process. The optimization parameters can be saved/loaded to/from a file, which enables a user to run the CoreWar Optimizer even without a GUI. However, most users would certainly like to take advantage of the built interface. An example of how the parameters configuration file looks is given below:

```
;warrior E:\diplomski_rad\code\examples\TestWarrior.red
;name TestWarrior
;method 2
;instances 2000
;rounds 250
;phase1 E:\diplomski_rad\code\bench\scn\willow.red
;phase2 cds|scn|stn
;threshold1 179.0
;threshold2 169.0
```

The configuration specification is self-explanatory. The `warrior` keyword denotes the path string to the warrior file. Warrior name determines what is going to be written in the results files and in the candidate solution files. Each method is encoded by a number and the number 2 in the above example means that the method to be used is simulated annealing. The `instances` keyword specifies the number of candidate solutions to be performed unless the process is stopped by a user. It is also necessary to specify the number of rounds in each match-up. It is customary to set this to a number of 250, for reasons of statistical accuracy of the results. Phase 1 warrior is given by its file path and phase 2 warrior sets by their names. The configuration file also contains the specification of the two thresholds for the first two optimization phases.

Since the optimizer keeps track of the best warrior found during the run, one might ask oneself why is it that CoreWar Optimizer persists no less than 20 best candidate solutions and all their HTML score files. It certainly doesn’t hurt to do so, but it may seem superfluous. There are two very significant reasons for such an implementation decision. First of all, one must always keep in mind that the goal of optimization in the context of CoreWar simulation is to produce warriors that will perform well against warriors currently on the hill or entries of some tournament, etc. In other words, the optimized warrior will not be fighting against the warriors from the benchmark, but against some other warriors. In such situations, optimization can lead to overfitting to the current benchmark (test set which implicitly defines a warrior’s fitness), so even though a good benchmark score suggests that the generated solution would probably be a good one, it is not necessarily
so. This can, of course, be resolved via creating larger and more diverse benchmark warrior sets, but this has a direct impact on the execution time which depends almost linearly on the size of the benchmark. Benchmark size for 94nop environment is already at the upper limits. However, there is another reason as well. The fitness of a warrior is defined by its average score against all the benchmark warriors. However, a warrior that loses 0 - 250 to half of the benchmark and wins 250 - 0 against the other half would have obtained the same average score as the warrior that scores 125-125 against all the warriors. This is a simplified situation, with only wins and losses taken into account (there are also ties) and also extreme, but it makes it easier to describe why the average score can not be the sole quantifier for describing how well a warrior performs. More specifically, if a warrior loses a lot to certain strategies and wins a lot against the other ones, it can never last for long in the CoreWar leagues, since the hill sizes are usually small and never represent equally all the strategies. It would be even less likely to assume that they reflect the strategic composition of the CoreWar Optimizer benchmark. That is why, at the end of each run, CoreWar Optimizer calculated the standard deviation for a given warrior. Standard deviation is the usual dispersion measure and here it is used to suggest a user how much does the score of the warrior that is being optimized deviate from the average one. It would not be possible to create a hybrid formula that would take advantage of both the average score and the dispersion of the result set, due to the fact that the average dispersion is both benchmark-dependent and also strategy-dependent. For instance, replicators have much more stable scores than stones(section 3.3.3) or HSA-type scanners(section 3.3.5).

The path to where the results and the warrior are to be persisted is not given by a user, since there is no need for so much flexibility. In the CoreWar Optimizer directory, there is a directory name work and in that directory, the application makes a directory with the name equal to the warrior name. All the data related to a particular optimization project are kept there. Candidate solutions are named in an incremental fashion, concatenating the number of the solution with the warrior name.
5.2 Optimizing Variables

Since CoreWar Optimizer makes an attempt at improving some of the critical instruction field values in the selected Redcode program, it is necessary to specify which instruction fields are to be optimized and to place some constraints on the acceptable values. Actually, the application offers a bit more flexibility than this, since it doesn’t directly optimize the values in the instruction fields, it optimizes Redcode variables instead. What is the difference? Well, Redcode allows for the use of user-defined variables in the code, which makes the code much more readable and easier to maintain. Also, it helps in modelling some dependencies between the various instruction field values in a warrior, which certainly do exist. That means that a single variable is often used in more than just one instruction field, being a part of the mathematical expression of the field which may include more than one variable. That way, if the variable is changed, all the dependent fields values change as well, and they do so consistently, by preserving the functionality of the code in question. Back to the topic - since CoreWar Optimizer optimizes variables, it is able to optimize multiple instruction fields at once, which reduces the dimensionality of the search space. Also, if some of the field values that are being optimized and mutually dependent, this ensures that the candidate solutions with be valid in that respect. Here is how the specification should be given:

```redcode
;optimizer_start
a equ !(minaVal, maxaVal)
b equ !(minbVal, maxbVal)
c equ !(mincVal, maxcVal)
;optimizer_end
```

The declaration of the critical variables must be contained within a block delimited by the `;optimizer_start` and `;optimizer_end` keywords. There may be more than one such block within a warrior code. In the example above, variables `a`, `b`, and `c` are declared by associating each one of them with a valid values interval. In the example above, the dimensionality of the search space would equal \((\maxaVal - \minaVal)(\maxbVal - \minbVal)(\maxcVal - \mincVal)\). If the respective intervals represent the full possible range given the environment, in the 94nop setting that would amount to \(5.12 \cdot 10^{11}\) variations. Naturally, an arbitrary number of variables is allowed. As the number of variables increases and with the increase of the core size, search space becomes simply too huge to allow any hopes of getting near the global maximum. However, local search would still produce its suboptimal solutions.
By looking at the above specification, one immediately realizes that there are possibilities to further extend it and increase the number of options. For instance, valid value sets could be given as a union of valid intervals. Values could also be declared by some sort of a mathematical expression. The former would certainly be useful to a user, but would require major changes in some of the algorithms, since the search space would no longer be connected. On the other hand, the latter might make the variable declaration more readable, but is in a way unnecessary, since mathematical expressions are allowed in Redcode and the variable value generated by the CoreWar Optimizer can be processed afterwards by the existing mechanisms in the Redcode compiler. There may still exist a few situations where this would not suffice, but they nearly never occur, so this feature has been omitted from the first versions of the CoreWar Optimizer tool.

5.3 Brief Description of Components

CoreWar Optimizer comprises several components. The public classes are: CorewarOptimizer, Score, MarsComm, CorewarOptimizationGenotype and Particle. They are all contained in the same package. So, let us review the components one by one.

The Score class contains the results for all 3 optimization phases for a given warrior. It also implements some simple comparison methods. The most important method in that class, however, is the calculateFitness() method. Explicit fitness values is needed, especially in the GA optimization method where even those warriors that didn’t compete in all three phases (since they scored less than a specified threshold value) need to be associated a fitness score. This is how it was resolved:

```java
public double calculateFitness() {
    if (PH3 > EPSILON) {
        return 200. + (PH3 / 3.);
    } else if (PH2 > EPSILON){
        return 100. + (PH2 / 3.);
    } else {
        return PH1 / 3.;
    }
}
```

The conditions in the if-else clauses basically check if the warrior had made it to 3rd, 2nd or only the 1st phase. This would be a good moment to mention that the average score can not exceed 300 (since 3 points are given
for a win, 1 for a tie and 0 for a loss) and MARS returns the average score percentage regardless of how many rounds there are in a given match-up. Therefore, $1/3$ of the average score does not exceed 100. With that taken into account, the fitness function that was used throughout the CoreWar Optimizer ensures that all the warriors entering phase 3 will have higher fitness than all those that have made it only to phase 2 and those will be more fit than the ones that failed in the very beginning. The EPSILON constant is used as a precaution everywhere in the code because of floating-point operations inherent imprecision.

The MarsComm class contains the logic behind calls to MARS and is used to rout demands to MARS and return the results to the methods that requested it.

The CorewarOptimizationGenotype class represents a data instance - a particular candidate solution. It contains the genes, alleles and some other data relevant for the GA implementation, but that will be introduced in section 5.5.5. The class also contains mutation and recombination operators for the GA algorithm. However, some of the methods in its interface are more general and important for all the algorithms involved. The most important ones are $\text{birth(File directory)}$ and the class constructor taking the warrior specification file as a parameter and producing a new CorewarOptimizationGenotype instance. The $\text{birth}$ method generates a Redcode file corresponding to a CorewarOptimizationGenotype instance. This file is then afterwards used by MARS for benchmarking.

The Particle class contains much of the logic related to the PPPSO algorithm. This will be further addressed in the corresponding section.

The CorewarOptimizer class is the main component in the whole package. It contains most of the program logic and is a java application that can be called either from the command line or a web browser. All the optimization methods are implemented as methods within this class. Their specifics will be given in the following sections.

There aren’t many components in the package and it would have probably been beneficial to allow for a bit more modularity. The code will probably be separated into several smaller classes in the next versions, separating each optimization method into a different class, etc.
5.4 User Interface

One of the best features of CoreWar Optimizer is that it offers a comfortable work environment. The only other instruction field value optimizer in the corewar community, Optimax [23], does not have a GUI and its methods are called from the command line.

When the application is called, a user is presented with the input window for the optimization parameters. This is shown in figure 8. The visual display consists of a JTabbedPane containing three JPanel containers. The first JPanel, input, is there for optimization parameter initialisation. The second one, named warriors, displays a table of the best 20 candidate solutions. The last one, named statistics, displays the maximal, minimal and average scores as well as numbers of warriors for each three optimization phases. Values within these last two panels are updated each time a candidate solution finishes an optimization phase. This is being taken care of by the method resolveStatistics() in the CorewarOptimizer class. There is also a help menu.

The input panel consists of various input forms. This includes 7 buttons, 6 text fields, 2 combo boxes, a list and a text area. These correspond to JButton, JTextField, JComboBox, JList and JTextArea swing classes. Above these elements is a JProgressBar element which displays the percentage of the task that has been completed. Let us begin by reviewing the buttons
Two of the buttons are related to the optimization parameters persistence. One is for the load of the parameters from the configuration file described in section 5.1 and the other for saving the selected options. Open/save menus memorize the last visited folder, so that a user doesn’t need to browse all the way through each time when using them. The next two buttons are the open/save buttons for the warrior file. When the warrior is loaded, it is displayed in the text area on the same panel. There it can be viewed and edited by a user. The text area component is contained in the JScrollPane, so the size of the warrior file poses hardly any problems for a user. The next 3 buttons are: reset, optimize and stop. The optimize button is colored in red. Reset button does what is apparent, it resets all the fields and all the variable values in the code that correspond to those input fields. The stop button stops the optimization process. The process is not momentarily stopped, to be more precise. There is a boolean field runningOptimization that is set to true when the optimization starts and that is set to false when the stop button is pressed. This is because at the moment when a user wants to stop the process, it is most likely that the thread performing the optimization is waiting for the MARS to finish some rounds. For the sake of consistency of statistics, this small concession has been made. The algorithms check quite often to see if the optimization flag is on or off. In practise, the thread will end within less than a minute from the moment the optimization is formally stopped. In the next versions, the algorithmic parts of the code will be moved away from the GUI class, which will enable parallel execution of several optimization processes. However, even though it is a small nuisance, this does not pose a serious problem. As for the optimize button, once pressed, locks all the input fields, starts the progress bar and begins the optimization run. If not all the parameters have been entered, it issues an error message in a separate window. Also, during the optimization, the name of the current warrior candidate file that is being processed is displayed under the progress bar, along with the time that has passed, which is printed out in red. This way, a user can always have the information about how long the process has been running, simply by looking at the first panel.

The combo boxes are there for the selection of the phase 1 warrior and the optimization method. The methods available are: random, hill climbing, simulated annealing, predator-prey particle swarm, genetic algorithms.

On the leftmost part of the first panel is the JList component containing all the strategic warrior groups. User can select an arbitrary subset of the offered benchmark groups ensemble.

The text fields are there to allow the input of the project name, number of simulation cycles, number of rounds and the thresholds for the first two
optimization phases.

The help menu offers several help tips: about the warrior format, about importing parameters, frequently asked questions, credits and general information about the current version of the application. All the items in the menu, as well as the menu itself, can be opened by pressing the corresponding hotkey on the keyboard. This was set up by the `setMnemonic(KeyEvent key)` method. One of the help tip windows is shown in figure 9. Those help windows are given by the static `showMessageDialog` method of `JOptionPane`.

All the elements on display have a tool tip text set to them, so that when a user places the mouse cursor over the element, this help text is shown next to it. The event model used is, of course, that of many ActionListener implementations for the components in question.

In the second panel, the best 20 instances are displayed. In the table, there is data about the phase 1, phase 2, phase 3 score, standard deviation and average scores against every single strategic subgroup of the warrior benchmark.

The final component, displaying the remaining statistics data is pretty simple and there is no need to further discuss it here.
5.5 Implementation of Optimization Methods

This section is where the core of the project is to be presented, i.e. the implementations of the optimization algorithms used in the CoreWar Optimizer. After all, what’s the use of all the flashy graphical components if there is no merit in what the application does? It is, of course, a rhetorical question and if any answer is to be given, it is presented below. Each algorithm is described separately and this is the only place where some Java code from the application is presented to the interested reader. There is still a textual description, so that the algorithms can be fully understood without examining the given code.

5.5.1 Random

The first algorithm offered by the CoreWar Optimizer mimics what has already been done in Optimax.\cite{23} I will just once again repeat the general idea. The optimization procedure is separated into 3 phases. In each phase, a warrior fights some benchmark warriors and proceeds to the next phase if its average phase score exceeds some apriori threshold. Other than that, the path through the search space is completely random and there is no logic attempting to detect score-critical areas of the adaptive landscape for the given warrior. Some score surfaces that have been plotted for some Redcode warriors and one-one match-ups showed that it is possible to distinguish regions where the score is much higher than average and, conversely, regions with very bad scores. This is a consequence of the fact that the variables that are being optimized are not mutually independent with regard to the high score probabilities.

This method has been implemented only to allow for some sort of comparisons to be made and it was simple enough not to require any extra effort once all the auxiliary methods have been coded. No code will be given for this method, since it doesn’t require any additional explanations.

5.5.2 Hill Climbing

This is the simplest of among the non-random algorithms implemented in CoreWar Optimizer. It is a slight modification to the basic hill climbing algorithm given in \cite{3,1,1}. It is not the iterative hill climbing algorithm variation, either. First of all, it is still designed so that it has 3 optimization phases, thus implementing the speed-up procedure from Optimax.\cite{23} This might not seem important at first, but it slightly changes how the good solutions are sought for. Now there is not one, but 3 adaptive landscapes, for each of the phases. This method first climbs the 1st landscape, until it
reaches some initial point on the 2nd where the first threshold condition is satisfied. After that, the process is repeated for the second landscape and so on. The algorithm doesn’t really pick a neighbour in the search space by some predetermined algorithm, it rather takes advantage of the genetic mutation method, creating a similar candidate solution. It also keeps track of the search history and only proceeds if the new item hasn’t been visited in the past. This ensure that it will never get caught up in some endless loops in some small region of the landscape. However, there is also an important tweak which I have devised and used to increase the probability of finding good solutions. This tweak relies on the fact that there are 3 optimization phases and that the most important one is the 3rd, since those warriors that are passed on to phase 3 are defined as acceptable by the user, since they satisfy both score threshold constraints. The algorithm has a counter which has a sole purpose of showing how many candidate solutions have been generated since the last solution that was good enough to enter phase 3. This number is then observed modulo some upper limit. Before it becomes apparent how this is used, it is necessary for me to point out a nice feature of the mutation operators used throughout the CoreWar Optimizer. They receive two parameters, \texttt{prob\_big} and \texttt{prob\_small}, denoting the respective probabilities of both big and small mutations. Large mutations can completely change the character of a candidate warrior and the small ones will produce some similar warrior. The probabilities are applied separately to each allele. Small mutations are used to move slowly through the search space and large mutations to change between the regions of the adaptive landscape that are being examined. The tweak related to the mentioned counter is that as time goes by and generated warriors fail to satisfy the constraints and enter the final optimization phase, which implies that the region of the landscape that is being examined is some sort of a plateau populated with low score values, the probability of big mutations taking place increases! In other words, if the examined solutions are bad, the algorithm will have a tendency to switch to some other area and start over. Also, there is always a finite positive probability of big mutations, so that the algorithm won’t get stuck near a local suboptimum indefinitiely, which means that the parameter also increases the globality of the search. There is one more thing to mention. All the algorithms keep track of 3 warriors: the \texttt{current} one, the \texttt{previous} warrior and the \texttt{last} warrior, the last name being slightly misleading. This is due to the fact that there is often a need for comparing the current and the previous candidate solution to determine which is going to be used, and the \texttt{last warrior} is simply the next in the reverse time succession, i.e. a previous warrior of a previous warrior. The reference to it is kept so that its file gets properly deleted from the general warrior file pool. The code of the algorithm is given
below to illustrate how the implemented optimization algorithms look like.

```java
public void HCOptimizer() {
    int timeSincePH3 = 0;
    int upLimit = 300;
    CorewarOptimizationGenotype o =
        new CorewarOptimizationGenotype(warriorFile);
    o.becomeRandom();
    history.add(o);
    o.birth(projectFolder);
    while (stillGo()) {
        timeSincePH3 = Math.max ((timeSincePH3 + 1) % 300, 1);
        deleteLast();
        combostat(PHASE1, o);
        while ((currentPH1Score < thresholdPH1) && stillGo()) {
            if (currentPH1Score < previousPH1Score) {
                o = previousWarrior;
            }
            o = o.mutate((float)((float)timeSincePH3 / (float)upLimit),
                (float)0.6);
            timeSincePH3 = Math.max ((timeSincePH3 + 1) % 300, 1);
            while (history.isPastGenotype(o) && stillGo()) {
                o = o.mutate((float)((float)timeSincePH3 / (float)upLimit),
                    (float)0.6);
            }
            history.add(o);
            o.birth(projectFolder);
            deleteLast();
            combostat(PHASE1, o);
        }
        combostat(PHASE2, o);
        while ((currentPH2Score < thresholdPH2) && stillGo()) {
            timeSincePH3 = Math.max ((timeSincePH3 + 1) % 300, 1);
            o.birth(projectFolder);
            deleteLast();
            combostat(PHASE1, o);
            while ((currentPH1Score < thresholdPH1) && stillGo()) {
                if (currentPH1Score < previousPH1Score) {
                    o = previousWarrior;
                }
                o = o.mutate((float)((float)timeSincePH3 / (float)upLimit),
```
timeSincePH3 = Math.max((timeSincePH3 + 1) % 300, 1);
while (history.isPastGenotype(o) && stillGo()) {
    o = o.mutate((float)((float)timeSincePH3 / (float)upLimit),
                 (float)0.6);
}
history.add(o);
o.birth(projectFolder);
deleteLast();
combostat(PHASE1, o);
}
combostat(PHASE2, o);
if (currentPH2Score < previousPH2Score) {
    o = previousWarrior;
}
o = o.mutate((float)((float)timeSincePH3 / (float)upLimit),
               (float)0.6);
timeSincePH3 = Math.max((timeSincePH3 + 1) % 300, 1);
while (history.isPastGenotype(o) && stillGo()) {
    o = o.mutate((float)((float)timeSincePH3 / (float)upLimit),
                 (float)0.6);
}
history.add(o);
}
combostat(PHASE3, o);
history.add(o);
timeSincePH3 = 1;
if (previousScore > currentScore) {
    o = previousWarrior;
}
o = o.mutate((float)((float)timeSincePH3 / (float)upLimit),
               (float)0.6);
history.add(o);
o.birth(projectFolder);
deleteLast();
combostat(PHASE1, o);
combostat(PHASE2, o);
while (stillGo() &&
       (currentPH1Score > thresholdPH1) &&
       (currentPH2Score > thresholdPH2)) {
    combostat(PHASE3, o);
timeSincePH3 = Math.max ((timeSincePH3 + 1) % 300, 1);
if (previousScore > currentScore) {
    o = previousWarrior;
}
o = o.mutate((float)((float)timeSincePH3 / (float)upLimit),
        (float)0.6);
while (history.isPastGenotype(o) && stillGo()) {
    o = o.mutate((float)((float)timeSincePH3 / (float)upLimit),
        (float)0.6);
    history.add(o);
    o.birth(projectFolder);
    deleteLast();
    combostat(PHASE1, o);
    combostat(PHASE2, o);
}

5.5.3 Simulated Annealing

As an optimization algorithm, simulated annealing offers a better trade-off between exploration and exploitation, stressing out the former in the early iterations and focusing on the latter in the remaining ones. The algorithm which was implemented slightly differs from the one that was described in section 4.1.2 as a default algorithm for performing simulated annealing. The implemented approach was based on [4]. It follows the update formula:

\[ p(s_{i+1} = o_i | f(s_i) < f(o_i)) = e^{-(f(o_i) - f(s_i))/Td} \] (5)

The difference lies in the factor \( d \), which denotes the average score change. The first several iterations of the algorithm are being run solely for the purpose of obtaining this value. The ratio \( (f(o_i) - f(s_i))/d \) might be considered to measure effectiveness of the change. The function \( f \) has been given in the above formula as if the task was function minimization, for easier comparisons with section 4.1.2 and historical reasons. It may be considered to represent a complement of the fitness function.
The annealing schedule determines the degree of the uphill movement permitted during the search. Choosing the annealing schedule in practical situations is far from being easy. The initial temperature can be determined according to the following rule:

\[ T_0 = \frac{-d}{\ln p_0} \]  

(6)

This enables us to define the initial temperature so that it provides us with the desired initial probability of uphill change. Setting \( p_0 \) to 0.8 is considered to be a good empirical choice. Such an approach was implemented in CoreWar Optimizer. The final temperature for the optimization run was also calculated by the same formula as in equation 6, only by taking \( p_f = 0.05 \). The temperature is updated by the following simple law:

\[ T_{k+1} = \alpha \cdot T_k \]  

(7)

Naturally, \( \alpha \) is easily deduced so that it produces the final temperature when the maximum number of iterations is given apriori. The simulated annealing implementation also takes into account the 3 optimization phases. Neighbour candidate solutions are being generated until the new one satisfies the threshold constraints. Once it does, the next state is chosen with respect to the transition probability described in equation 5.
5.5.4 Particle Swarm Optimization

The logic behind the particle swarm implementation is mostly contained within the Particle class. Those methods are then called from the CorewarOptimizer class method that runs the iterative process. The Particle class implements two methods of the outmost importance for this algorithm, namely updateVelocity() and move(). These methods implement the equations given in section 4.1.3. The only difference is that there has to be some caution regarding the possible violations of the upper and lower bound constraints for the field values. Another small difference arises from the fact that the fitness function domain is not a continuous one, the function takes discrete values as its arguments. Therefore, the calculations performed in the code should be viewed as integer approximations of the real-valued general case.

In the current implementation, the swarm comprises 10 particles - 9 prey particles and one predator particle. Swarm size doesn’t have to be fixed and it will be parameterizable in the next versions of this optimization tool. It was thought that it would be cumbersome for a user to go through a multitude of menus while being forced to make decisions about the algorithms a user has very little knowledge of. However, it is always good to leave the options open to those who can handle their proper use, so some form of standardized algorithmic configurations file would probably be a good choice. The optimization method keeps track of the current swarm champion, the overall best candidate solution during the run, the predator and the remaining 8 prey particles.

Implementation of the PPPSO algorithm does not differ much from the procedure that has already been fully described, so there is no need to go into further details.
5.5.5 Genetic Algorithms

Optimization relying on mutation and selection under simulated evolutionary pressure represents a popular way of dealing with multimodal problems. CoreWar Optimizer implements one variation of a genetic algorithm. Its implementation is separated into two classes, CorewarOptimizer and CorewarOptimizationGenotype.

The most important GA methods in CorewarOptimizationGenotype are becomeRandom(), generateRandom(), mutateSelf(float p), mutate(float p), mutateSelf(float pbig, float psmall), mutate(float pbig, float psmall), crossOverSwapping(...), crossOverAffine(...), resolveLimit(), updateSigmaArray(float tau), sigmaMutateSelf. It can easily be seen that there are several types of methods in this list. Their purpose is apparent when taking into account their names. Some of them produce randomized candidate solutions. Some produce neighbour solutions in the search space, as is the case with various mutation methods. There are two types of mutations methods implemented, those that only produce small mutations with some probability and those that produce both small and big mutations with respective probabilities psmall and pbig. Crossover methods are basically recombination methods and are also used to increase the diversity in the offspring. The swapping recombination method allows for the solutions to mutually exchange some of their genotype. The affine recombination method produces a child candidate solution which is similar to both its parent and for each allele x, it holds $x_{parent_1} < x_{child} < x_{parent_1}$. The resolveLimit() method ensures the consistency respective to the upper and lower bound conditions. The final two methods that have been mentioned above pertain to the mutation factors referred to as the $\sigma$ array.

The mutation operators discussed so far have all been global ones - the probability of mutation occurring was equal over the set of alleles. However, this is not very flexible. The conditional adaptive surfaces obtained when fixing certain dimensions might be of a different nature. Therefore, it would be useful to allow for different mutation strategies when it comes to different feature subspaces.

An individual, a candidate solution, represents a point in the search space. It quantifies the expected quality in its vicinity, given the set of observed solutions. Since various individuals represent various regions of the domain, it would be reasonable not to impose the same restrictions on all of them, i.e. to use local mutation operators instead of the global ones.

If the mutation operators were to be parameterized and localized, a mechanism for deduction of such operators based on the information contained within each of the candidate warriors needs to be properly defined. It is
only natural to think of the same type of strategies that are being used for
the optimization of the solutions, since choosing a proper mutation operator
accounts to no more than choosing the optimal one from the space of all
mutation operators. Of course, that would be an impossible task. On the
other hand, it would be quite enough to simply allow for the use of better
mutation operators than in the simplified version of the algorithm.

A really simple idea that was described in [7] is used in the GA algorithm
of CoreWar Optimizer. A separate mutation vector, $\sigma$ array, is associated
with every candidate solution. The mutation weights are optimized alongside
the rest of the warrior genotype. This literally means that the basic warrior
genotype is extended by these weight factors. Good $\sigma$ values will propagate
through the generations and hopefully improve the quality of the search.

According to the above proposed solution, the asexual mutation process
is governed by two equations - the one for the mutation rate update and the
other for the basic genotype update. The basic alleles are modified under
the mutation operator by the following formula:

$$\text{allele}_i(t + 1) = \text{allele}_i(t) + \lfloor \sigma_i(t + 1) \cdot N(0, 1) \rfloor$$

(8)

In other words, Gaussian noise is introduced to the variables. Mutation
weights are amplitudes regulating offsets of such mutations. The probability
distribution is Gaussian which results in smaller changes being more likely
than the larger ones. Variability of mutation weights is regulated by the
equation 9.

$$\sigma_i(t + 1) = \sigma_i(t) \cdot e^{\tau \cdot N(0, 1)}$$

(9)

The difference between the two rules is apparent. The changes to alleles
are of absolute nature, while the changes to mutation rates are relative ones.
The motivation for using the Gaussian randomness remains the same as in
the previous update rule. There is also a learning rate, $\tau$. In CoreWar
Optimizer implementation, the value of $\tau$ was set to 0.13118. This was
done intentionally to ensure a 95% confidence interval for the event that the
relative change to the mutation weight will not exceed 30%. This relative
threshold is arbitrary in nature, but it adheres to principles of common sense.
It is clear that the mutation rates should change at some relatively stable
pace. Oscilatory changes would only lead to instability and the population
might even not converge to the optimum in such case.

Population size in the algorithm is determined upon the total number
of candidate solutions specified by the user. However, its size is confined
to the interval $[40, 80]$, thus ensuring some degree of variability within the
population. Offspring is produced via some of the described mechanisms.
The initial offspring size is 4 times the size of the population. One quarter of the offspring is produced via crossOverSwapping(...), one quarter via crossOverAffine(...), and two quarters asexually.

Parents for the sexual reproduction are chosen with probabilities proportionate to their fitness. This is the usual approach. The speed-up trick from Optimax [23] which is utilized in all the other algorithms is used in the GA approach as well, while evaluating the fitness function for each individual. This function has already been described in section 5.3.

5.6 Benchmark

Analysing a warrior’s efficiency is never an easy task. There are many warrior types, too many factors to consider and it simply infeasible because one could never cover every aspect of the combat with the model and the calculations would also probably require too much time. In the end, it is always better to test the warrior in actual CoreWar simulation. If one were to work carelessly and without a system, it would be possible to run several match-ups against some warriors and then proceed depending on the evaluation of those results. However, such a lenient approach has a serious drawback. It becomes very hard to conduct any sort of comparison between the warriors. That is why the benchmarks exist. They standardise the testing and allow for a wider perspective.

If the benchmark is to be reliable, it needs to be representative in terms of strategic abundance. The interstrategic ratios need to resemble the actual frequencies of occurrences of the respective strategies in the tournaments. When there are many warrior types and subtypes, this becomes a very difficult task and can never be done with absolute precision. Also, one must update the benchmark occasionally, following the fashion of the period and also introducing new concepts and ideas that are being used instead of the new ones. Benchmark has to contain almost exclusively strong, optimized elements, for this is the kind that the warriors fight in real tournaments. The alternative idea, more suitable for warrior evolution (where the instructions are evolved, as well as field values), consists of more strategically pure elements and small components that make up bigger warriors. This is due to the fact that most of what the evolvers generate starts out as a weak warrior and would thus be utterly defeated by the strong opponents in the benchmark. This would lead to the loss of precision of the fitness measure, because all the warriors would end up with a similar fitness. However, in the optimization setting addressed by the CoreWar Optimizer, such issues do not exist and a strong benchmark is required, not only because of the relative scores of different candidate solutions and different warriors in general, but because
the average score is meant to approximate the expected hill score.

The 94nop CoreWar Optimizer benchmark comprises 12 warrior types, 110 warriors total. Most of the warriors in the benchmark are state-of-the-art from the current perspective, majority of them present in the 94nop Hall of fame. 16 benchmark warriors were created by the author of this paper over the past years. The benchmark is not fixed and a user can modify its composition.

The twelve strategies that are present in the 94nop benchmark correspond to the ones described in [19], namely oneshots (onesh), coreclears (clr), coreclears with imps (clrwi), coreclear-directing scanners (cld), other scanners (HSA type, Rave type, Deathstar type) (scn), stones (stn), A-field stone-imps (sai), B-field stone-imps (sbi), stones with both A and B field imps (sabi), replicators (pap), stone-papers (pws), papers with imp structures (pwi). Of course, this strategic representation of 94nop corewar warriors is merely one out of many possible levels of abstraction. Each of the above described groups can be further decomposed according to some other criteria. There is no need to go into more detail. A visual plot of the strategic distribution within the benchmark is given in figure 10.
6 Conclusions and Future Work

CoreWar Optimizer is a new tool for instruction value optimization of Redcode programs. It has been implemented in Java. CoreWar Optimizer supports several advanced options and is the first CoreWar development tool to apply global optimization methods to the problem of iterative warrior improvement reflected in the problem of field value selection. The algorithms embedded in this tool are: random, hill climbing, simulated annealing, predator-prey particle swarm and one of the genetic algorithms. The 94nop benchmark has been specified and used in the optimization process. CoreWar Optimizer keeps track of the optimization statistics and persists the scores in an easily readable HTML format. It is the first CoreWar optimizer to have a GUI and it represents a comfortable work environment. Despite all the novelties described above, there is still much room for improvement. In the future versions, there will be more configuration files and everything is going to be parameterized. Code will be separated to achieve better modularity and an optimization interface will be defined, so that everyone would become able to write the optimization algorithm classes used in the rest of the application. The optimization genotype class could, even now, be used to represent not only field value alleles, but also the instructions themselves if properly encoded. This suggests that the application could also be transformed into a CoreWar evolver with very little coding effort. Also, extensive tests are necessary to determine exactly how much improvement was achieved by the implemented techniques, averaged over various strategies and environment settings. I will go one step further to state that there is not a shred of doubt that even more ideas on how to improve the current concept will be sprung to life once the aforementioned ones get implemented. There are hardly ever perfect solutions, only better ones. There are many obstacles to overcome. Per aspera ad Astra.
References


Kratka biografija


Tokom studija dodeljena mu je izuzetna nagrada univerziteta za temat iz oblasti “Data Mining”. Autor je tri rada (iz iste oblasti) objavljena na međunarodnim konferencijama. Primio je nagradu "Aleksandar Saša Popović" za naučnoistraživački rad u oblasti računarskih nauka.

Novi Sad, July 2008.

(Nenad Tomašev)
Univerzitet u Novom Sadu  
Prirodno-matematički fakultet  
Ključna dokumentacijska informacija

Redni broj: RBR  
Identifikacioni broj: IBR  
Tip dokumentacije: Monografska dokumentacija  
TD  
Tip zapisa: Tekstualni štampani materijal  
TZ  
Vrsta rada: Diplomski rad  
VR  
Autor: Nenad Tomašev  
AU  
Mentor: dr Dragan Mašulović  
MN

Naslov rada: Optimizacija CoreWar programa  
NR  
Jezik publikacije: engleski  
JP  
Jezik izvoda: srpski/engleski  
JI  
Zemlja publikovanja: Srbija  
ZP  
Uže geografsko područje: Vojvodina  
UGP  
Godina: 2008  
GO

Izdavač: autorski reprint  
IZ  
Mesto i adresa: Novi Sad, Trg D. Obradovića 4  
MA

Fizički opis rada:  6/66/13/0/16/0/0  
(broj poglavlja/strana/lit. citata/tabela/slika/građa/priloga)

63
IZ
Datum prihvatanja teme od strane

DP

DO
Članovi komisije:
(Naučni stepen/ime i prezime/zvanje/fakultet)

KO
Predsednik: dr Mirjana Ivanović, redovni profesor, Prirodno-matematički fakultet, Novi Sad

Mentor: dr Dragan Mašulović, vanredni profesor, Prirodno-matematički fakultet, Novi Sad

Član: dr Siniša Crvenković, redovni profesor, Prirodno-matematički fakultet, Novi Sad
University of Novi Sad
Faculty of Science
Key Words Documentation

Accession number: NO
Identification number:INO
Document type: Monograph documentation
Type of record: Textual printed material
Contents code: Graduation thesis
Author: Nenad Tomašev
Mentor: Dr Dragan Mašulović
Title: Building a CoreWar Optimizer
Language of text: English
Language of abstract Serbian/English
Country of publication: Serbia
Locality of publication: Vojvodina
Publication year: 2008
Publisher: Author’s reprint
Publ. place: Novi Sad, Trg D. Obradovića 4
Physical description: 66/13/0/16/0/0
(no. chapters/pages/bib. refs/tables/figures/graphs/appendices)
Abstract: CoreWar is a computer simulation where programs written in an assembly-like language called Redcode compete in attempt to seize total control of the virtual machine executing them. One of the prime issues in this area is the instruction fields optimization with intent of improving average program performance. The paper presents the first software solution for this problem containing more sophisticated functionality. The constructed tool was named CoreWar Optimizer and its functions and components are described in the paper.