Categorizing Evolved CoreWar Warriors Using EM and Attribute Evaluation

Doni Pracner, Nenad Tomašev, Miloš Radovanović, and Mirjana Ivanović

University of Novi Sad
Faculty of Science, Department of Mathematics and Informatics
Trg D. Obradovića 4, 21000 Novi Sad
Serbia
doni@neobee.net, tomasev@nspoint.net, {radacha,mira}@im.ns.ac.yu

Abstract. CoreWar is a computer simulation where two programs written in an assembly language called redcode compete in a virtual memory array. These programs are referred to as *warriors*. Over more than twenty years of development a number of different battle strategies have emerged, making it possible to identify different warrior types. Systems for automatic warrior creation appeared more recently, evolvers being the dominant kind. This paper describes an attempt to analyze the output of the CCAI evolver, and explores the possibilities for performing automatic categorization by warrior type using representations based on redcode source, as opposed to instruction execution frequency. Analysis was performed using EM clustering, as well as information gain and gain ratio attribute evaluators, and revealed which mainly brute-force types of warriors were being generated. This, along with the observed correlation between clustering and the workings of the evolutionary algorithm justifies our approach and calls for more extensive experiments based on annotated warrior benchmark collections.

1 Introduction

Among the many approaches to creating artificial intelligence and life, one is concerned with constructing computer programs which run in virtual environments. Many aspects of these environments may be inspired by the real world, with the overall objective to determine how well the programs adapt. In some cases different programs compete for resources and try to eliminate the opposition.

One of the oldest and most popular venues for the development and research of programs executing in a simulated environment is CoreWar, in which programs (referred to as *warriors*) attempt to survive in a looping memory array. The system was introduced in 1984 by A. K. Dewdney in an article in the Scientific American [1]. Basically, two programs are placed in the array end executed until one is completely eliminated from the process queue. The winner is determined through repeated execution of such "battles" with different initial positioning of warriors in the memory. Online competitions are held on a regular basis, with the game being kept alive by the efforts of a small, but devoted community.

Over the course of more than twenty years of development, a number of different battle strategies have emerged, often combining more than one method for eliminating opponents. These strategies closely reflect programmers' ideas about how a warrior

should go about winning a battle. However, several attempts have been made recently to automatically create new and better warriors, by processes of optimization and evolution. Optimized warriors are essentially human-coded, with only a choice of instruction parameters being automatically calculated to ensure better performance. On the other hand, evolved warriors are completely machine generated through the use of evolutionary algorithms.

In order to evaluate the performance of optimized and evolved warriors, the most common method is to put them against a benchmark set of manually prepared test programs. To get reliable and stable results against every warrior from the benchmark in the usual setting, at least 250 battles are needed, each taking a few seconds to execute. Evolving new warriors from a set of a few thousand programs and iteratively testing them against the benchmark is then clearly a very time demanding process.

The goal of the research presented in this paper is to examine the diversity of warrior pools created by one particular evolver and to test the possibilities of automatic categorization by warrior type (employed strategies), given the information obtained by syntax analysis of warrior source code. The amount of data created by evolver runs usually surpasses the capabilities of human experts to examine and classify the warrior pools. Automated categorization would, therefore, be extremely helpful in the control of diversity levels, and dynamic modification of mutation rates for sustaining the desirable diversity within generations. It would also significantly contribute to our understanding of the nature of the output of evolutionary algorithms, in this case the battle strategies of evolved warriors. Although one may be familiar with every detail of how a particular evolutionary algorithm works, its output is still very much dependent on the performance of warriors against the benchmark, leaving room for many surprises.

There were some attempts in the past to perform automatic categorization of warriors, but these were based on the analysis of execution frequencies of particular instruction types during simulation, which requires the simulation to be run for a certain amount of time [2]. If the source-based approach proved fruitful, it would be possible to come to similar conclusions much quicker, which could, in turn, speed up the whole process of warrior evolution. To the best of our knowledge, this paper presents the first attempt to categorize warriors using *static* (source-based) instead of *dynamic* (execution-based) methods.

The rest of the paper is organized as follows. Section 2 explains the essentials of CoreWar and some basic strategies of human-coded warriors, while Section 3 outlines the principles of the EM clustering algorithm. Section 4 describes the dataset of evolved warriors and how it was processed into the representation suitable for analysis. The analysis, which relies on clustering and attribute evaluation techniques, is the subject of Section 5. The last section provides a summary of the conclusions together with plans for future work.

2 CoreWar

CoreWar is a computer simulation where programs written in an assembly language called corewars (by the 1988 ICWS standard) or redcode (by the 1994 ICWS standard) compete in a virtual memory array. Those programs are referred to as *warriors*.

The simulated memory array is called *the core*. It is wrapped around, so that the first memory location in the address space comes right after the last one. The basic unit of memory in the core is one instruction, instead of one bit. The memory array redcode simulator (MARS) controls the execution of the instructions in the core. The execution of instructions is consecutive, apart from the situations arising after executing jump instructions. All arithmetic is modular, depending on the size of the core. All addressing modes are relative.

The goal of a warrior is to take complete control over the core by making the opponent eliminate its own thread of execution from the process queue. There are many ways to achieve this effect, and various different strategies of attack have emerged over time. CoreWar warriors can copy the memory content, read from the core, perform various calculations, mutate and change their behavior, make copies of themselves, place decoys, search for their opponents etc. The starting placement of warriors in the core is done at random, and a predetermined number of fights are staged to decide the winner (3 points are awarded for a win, 1 for a draw, 0 for a loss). Between rounds, the result of the previous fight is stored in a separate memory array called P-space. In some competitions warriors are allowed to access this memory and change their strategy, if necessary, to ensure better performance in future rounds.

CoreWar was introduced by A. K. Dewdney in 1984, in an article published in the Scientific American [1]. Today, CoreWar exists as a programming game with ongoing online competitions on several servers, among which are `www.koth.org/` and `sal.math.ualberta.ca/`. There are many competition leagues, depending on battle parameters, and each of these is called a hill. The warrior currently holding the first place is appropriately called the *king of the hill* (KOTH).

Although the competitions were originally meant as a challenge for testing human skill in making successful CoreWar programs, there were also those who chose to create software capable of autonomously generating or evolving and later evaluating competitive CoreWar programs. On several occasions such warriors were able to outperform warriors coded by humans. This is usually done via the implementation of evolutionary algorithms.

2.1 The Redcode Language

Redcode is a language that is being used as a standard for making CoreWar warriors since 1994. It consists of 19 instructions, 7 instruction modifiers and 8 addressing modes. The warrior files are stored on the disk as *WarriorName.RED*.

The redcode instruction set, although not huge, allows for much creativity and diversity. Each command consists of an instruction name, instruction modifier, A-field addressing mode, A-field value, B-field addressing mode, and the B-field value. The source address is stored in the A-field and the destination address in the B-field. Table 1 summarizes the more important redcode instructions, while Tables 2 and 3 describe all redcode modifiers and addressing modes, respectively. Figure 1(a) depicts the source of an example warrior.

Table 1. Overview of some redcode instructions

Instruction	Description
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.
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 is not used as frequently as ADD or SUB, however.
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.
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. Otherwise functions like JMZ.
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 old process, being the one that executed the SPL is moved to the next memory location. The new process is executed right after the old process.

Table 2. Overview of redcode instruction modifiers

Modifier	Description
.I	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.
.F	Copying, or comparing two fields at the same time.
.X	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.
.A	Moving, or comparing, the A field of the source to the A-field of the destination.
.B	Moving, or comparing, the B field of the source to the B-field of the destination.
.AB	Moving, or comparing, the A field of the source to the B-field of the destination.
.BA	Moving, or comparing, the B field of the source to the A-field of the destination.

Table 3. Overview of redcode addressing modes

	Addressing Mode	Description
$	direct	Points to the instruction x locations away, where x is the respective field value in the executed instruction. It can be omitted.
#	immediate	Points to the current instruction, regardless of the field value.
$*$	A-field indirect	Points to the instruction $x + y$ locations away, where x is the respective field value and y is the value in the A-field of the instruction x locations away.
@	B-field indirect	Analogous to A-field indirect.
{	A-field predecrement	Indirect mode, also decreasing the A-field value of the instruction pointed to by the respective field in the executed instruction. The decrement is done before calculating the source value of the current instruction.
}	A-field postincrement	Indirect mode, also increasing the A-field value of the instruction pointed to by the respective field in the executed instruction. The increment is done after calculating the source value of the current instruction.
$<$	B-field predecrement	Analogous to A-field predecrement.
$>$	B-field postincrement	Analogous to A-field postincrement.

2.2 Warrior Types

As mentioned before, over twenty years of CoreWar competitions had lead to a great increase in diversity of warrior types. Some of the most important warrior categories are given below.

Imps are the simplest kind of warriors which just copy themselves to another memory location in each execution cycle, that way "running around" the core. Imps barely have any offensive capabilities, and are seldom used on their own.

Coreclears attempt to rewrite the whole core with process-killing instructions, that way ensuring a win, in a sense of being positive that the opponent is destroyed.

Stones simply copy DAT instructions over the core, trying to overwrite a part of the enemy code. Up to this moment, many alternate approaches were devised, resulting in warriors copying other instructions as well, not only DATs.

Replicators (papers) follow 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. In the meantime, the warrior tries to destroy the enemy process. The warrior in Fig. 1(a) is, in fact, a replicator, referred to as the "black chamber paper."

Scanners (scissors) try to discover the location of enemy code and then start an attack at that location. Since the scanner attack has a greater probability of succeeding, due to the intelligent choice of target location, such a warrior is usually able to invest more time in the attack against that location.

Hybrid warriors combine two or more warrior types in their code, and are nowadays most frequently used in CoreWar tournaments.

Generally, each non-hybrid type of CoreWar warrior is effective over one other warrior type, and is at the same time especially vulnerable to another, with the relationships between types being in line with the rock-paper-scissor metaphor (hence the naming of some warrior types). For more information about the redcode language and warrior types, see [3].

3 Expectation Maximization

The research described in this paper utilizes the *expectation maximization* (EM) clustering algorithm [4] (p. 265), implemented in the WEKA machine learning workbench. This algorithm is probabilistic by nature, and takes the view that while every instance belongs to only one cluster, it is almost impossible to know for certain to which one. Thus, the basic idea is to approximate every attribute with a statistical *finite mixture*. A mixture is a combination of k probability distributions that represent k clusters, that is, the values that are most likely for the cluster members. The simplest mixture is when it is assumed that every distribution is Gaussian (normal), but with different means and variances. Then the clustering problem is to deduce these parameters for each cluster based on the input data. The EM algorithm provides a solution to this problem.

In short, a procedure similar to that of *k-means* clustering ([4], pp. 137–138) is used. At the start, the parameters are guessed and the cluster probabilities calculated. These probabilities are used to re-estimate the parameters, and the process is continued until the difference between the overall log-likelihood at consecutive steps is small enough. The first part of the process is "expectation," i.e. the calculation of cluster probabilities, and the second part – calculating the values of parameters – is the "maximization" of the overall log-likelihood of the distributions given the data.

WEKA's implementation of EM provides an option to automatically determine the number of clusters k using 10-fold cross-validation. This is done by starting with $k = 1$, executing the EM algorithm independently on every fold and calculating the average log-likelihood over the folds. As k is incremented the process is repeated until the average log-likelihood stops increasing.

4 The Dataset

The analyzed data represents a subset of warriors generated by the CCAI evolver [5], which was written by Barkley Vowk from the University of Alberta in summer 2003. The evolutionary approach used in this evolver was the island model [6].

The dataset consists of 26795 warrior files, and is summarized in Table 4. The data was divided into four smaller parts in chronological order. The respective sizes of the parts are 10544, 6889, 4973, 4389, and will be referenced in the text as "generation 1," "generation 2" etc. The first pool was randomly generated, and the others represent the consecutive generations in evolving. One of the reasons why each group is smaller than the previous one is that evolvers reduce diversity in each step, and duplicates are removed before proceeding to the next generation. The benchmark used for this evolution was Optimax [7].

4.1 Selecting the Representation

Inspired by the classical bag-of-words representation for text documents, and the fact that it works for many types of data mining and machine learning problems, we opted for an analogous "bag-of-instructions" representation for CoreWar warriors. Since each instruction may be accompanied by an instruction modifier, two addressing modes and two field values, there are plenty of choices for deriving attributes, possibly leading to a high dimensionality of the representation.

In the end, the decision was made to use a vector with just the bare instruction counts from the warrior source code. The resulting vector has 16 coordinates (attributes), one for each of the command types. The name of the warrior was also added as an attribute. To transform the data into vector form a Java command line application was written, details of which are presented in [3].

Some modifications were introduced to make the information more specific to redcode, and the first alteration was to treat ADD and SUB as the same instruction, being that they can perform the same operation by simply toggling the minus sign in the address field.

The next alteration was done in order to add more information about the structure of the warriors to the representation. For many types of warriors there are specific pairs of commands that appear one after the other. Based on our previous experience with warrior types and coding practices, eight pairs of these two-command combos were added to the representation, namely SPLMOV, MOVJMP, MOVDJN, MOVADD, MOVSUB, SEQSNE, SNEJMP and SEQSLT.

Finally, there are sets of commands specific to some types of *imps*, so a true/false field named "Imp spec" was introduced. Examples of such commands are `MOV.I 0,1` and `MOV.I #x,1`. The presence of any of the commands suggests that an imp structure could be embedded within a warrior.

Figure 1 shows the representation of an example warrior (a) as a vector of attributes (b) described above.

4.2 Removing Duplicates

Besides choosing an appropriate representation, a method for speeding up calculations, as well as improving results, is to remove "too similar" warriors. When clustering the data, warriors which are close to each other in terms of distance between the appropriate vectors in the state-space (containing all the vectors), could easily gravitate smaller groups toward them, thus creating a larger cluster than it should be.

A decision was made to ignore the *address fields*, and therefore duplicates would be any two warriors that have the same sequence of instructions with identical instruction modifiers and address modifiers.

Table 4 summarizes the results of the duplicates search. Most duplicates were removed from generations 1 and 2, 12% and 8% respectively. From generation 3 only about 1% of the files were removed as duplicates, and in set 4 about 4%. In summary, a total of 2153 duplicates were found, which is about 8% of the initial 26795 warriors.

```
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
```

(a)

```
DAT:      0
MOV:      5
ADD/SUB:  0
MUL:      0
DIV:      0
MOD:      0
JMP:      0
JMZ:      1
JMN:      0
DJN:      0
SPL:      7
SEQ:      0
SNE:      0
SLT:      0
NOP:      0
SPLMOV:   4
MOVJMP:   0
MOVDJN:   0
MOVADD:   0
MOVSUB:   0
SEQSNE:   0
SNEJMP:   0
SEQSLT:   0
Imp spec: false
```

(b)

Fig. 1. Example code of a warrior (a), and its attribute vector representation (b)

Table 4. Summary of datasets and results of duplicate detection

Dataset	**Files**	**Duplicates**	**Reduction**
Generation 1	10544	1345	12%
Generation 2	6889	559	8%
Generation 3	4973	56	1%
Generation 4	4389	193	4%
Complete	26795	2153	8%

5 Analysis of Evolved Warriors

5.1 Clustering

First, clustering was performed independently on all warrior generations (and also on the complete set) using the implementation of EM from the WEKA workbench. The number of clusters was automatically determined by cross-validation (see Section 3).

The number of discovered clusters per warrior set and the number of instances per cluster are given in Table 5. In generation 1, only two clusters were found. After examining a portion of the warriors in this set, it appeared that the two clusters that were found consist mostly of various kinds of replicators and some coreclears. This was determined by taking a random sample of 50 warriors from each of the clusters. The only way to achieve absolute confirmation is to manually examine all warriors, which we considered infeasible. However, some insights provided by attribute evaluation (Section 5.2) give additional support to the finding.

Table 5. Clusters per generation and number of warriors per cluster

Dataset	Clusters	Cluster sizes											
Generation 1	2	8081	1146										
Generation 2	4	3456	1857	572	468								
Generation 3	12	88	2112	644	543	526	47	671	36	47	103	38	80
Generation 4	5	2364	1197	357	94	184							
Complete	3	6571	2671	15469									

Compared to generation 1, the number of clusters increases in generations 2 and 3, more precisely 4 and 12 respectively, but this was expected. The warriors in each set were evolved from the previous, and new strategies that had good results were preserved. This means that new groups of warriors with similar strategies should appear in generations 2 and 3, and the clustering algorithm did notice this.

In the last generation, the fourth, the number of clusters decreased to 5. This is most probably due to the reduction of diversity in the warriors that takes place at the end of the process of evolution.

The clustering of the whole dataset resulted in 3 clusters. The reduction of the number may be a consequence of the island model – the larger clusters most likely "absorbing" the smaller ones.

5.2 Attribute Evaluation

To analyze the effects of different attributes on cluster selection, *information gain* (IG) and *gain ratio* (GR) attribute evaluators were used [4]. Because of known shortcomings of both evaluation methods[1], the approach that was utilized was to choose the attributes

[1] IG favors attributes with many distinct values, while GR may give unrealistically high scores to attributes with a low value count.

with the highest gain ratio, but only if their information gain is larger that the average information gain for all attributes ([4], p. 105).

Since attributes in the warrior representation mostly correspond to instructions and instruction pairs, we expected their (in)significance with regards to the clustering to give us some idea about the types of warriors that were grouped together, and also to shed some light on the process of warrior evolution.

Table 6 summarizes the results of attribute evaluation on the complete dataset. It shows that the most informative feature is 'DJN,' being that others with higher GR values have very low information gain. It is interesting to note that the second best is 'MOVDJN' and that these two are also the best two in IG values. However, the rest of the information gain list does not follow in the same order. The 'SPLMOV' attribute also has high information gain, but shows less in terms of gain ratio. Looking from the CoreWar perspective, 'SPLMOV' and 'MOVDJN' are instruction pairs appearing frequently in both coreclears and replicators, so this result is not surprising. It also suggests that the results might have been significantly different if these attributes had not been used, and an ordinary bag-of-instructions was employed instead.

After the analysis of the complete dataset, an attempt was made to get more information on the actual evolution process by examining the individual generations, keeping in mind that the first generation is (in big part) random.

Table 7 lists the most informative attributes for generations 1–4, with their GR and IG scores. On generation 1, analysis showed that the most informative attribute is 'JMN,' being the first selection or both information gain and gain ratio. 'SLT' is also close, followed by 'MOVSUB' and 'MOVADD' after a large gap.

An interesting observation is that the best attributes for the complete dataset, 'DJN' and 'SPLDJN,' are at the very bottom of the list in generation 1. An interpretation for this is that there was greater variety in the original pool, which did not affect the global dataset at a greater measure, especially when considering the fact that subsequent generations increasingly resemble the complete dataset, as demonstrated below.

In the second warrior set the situation was significantly different compared to generation 1, with 'JMZ' and 'DIV' leading the GR scores, but with low IG. After filtering with the average IG, the list is as follows: 'MOVADD,' 'ADD/SUB,' 'MOVDJN,' 'SPLMOV,' 'DJN,' and 'SPL'. Here 'MOVDJN' and 'DJN,' which were important for the complete dataset, do appear in the list. Also, most of the best attributes from generation 1 do not show, or are a lot lower in the list, except 'MOVADD' and 'ADD/SUB'. This all indicates that much code from generation 1 was discarded during evolution. This is also evident in the reduction of size by 40% between generations 1 and 2.

In the third group, analysis shows that 'SPLMOV,' 'MOVDJN,' 'DJN,' 'ImpSpec,' and 'SPL' had most impact on the clustering process. Compared to the second generation, 'ADD/SUB' and 'MOVADD' which were "inherited" from generation 1 are now gone, leaving a result much closer to the complete set.

In generation 4, 'SPLMOV,' 'MOVJMP,' 'MOVDJN,' and 'DJN' were the most significant attributes with regards to clustering. The only big difference between this and generation 3 is the "climbing" of 'MOVJMP'. This lack of differences is also consistent with the earlier explained way the CCAI evolver works, in the sense that when there are no great improvements to the warriors in the next generation the process is stopped.

Table 6. Gain ratio and information gain for the complete dataset

Gain Ratio		Information Gain	
0.40350	MUL	**0.49217**	**DJN**
0.38200	SLT	**0.43917**	**MOVDJN**
0.37550	SEQSNE	0.31378	MOV
0.34570	MOVSUB	0.22883	SPLMOV
0.33690	MOD	0.20510	SPL
0.31520	SNEJMP	0.17605	Imp spec
0.26310	**DJN**	0.17025	DAT
0.24970	SEQSLT	0.15134	ADD/SUB
0.24820	JMZ	0.14137	MOVJMP
0.24070	**MOVDJN**	0.09816	SEQ
0.18620	NOP	0.09345	MOVSUB
0.18270	ADD/SUB	0.07179	JMP
0.18040	Imp spec	0.06768	MOVADD
0.17430	DIV	0.06380	SNE
0.15190	MOVADD	0.06001	JMZ
0.11840	JMN	0.03642	MUL
0.11630	SNE	0.02582	SLT
0.10140	SEQ	0.02370	NOP
0.09350	MOVJMP	0.02184	SEQSNE
0.08930	MOV	0.02126	JMN
0.08240	SPLMOV	0.01484	DIV
0.05780	SPL	0.01138	MOD
0.05420	JMP	0.00730	SNEJMP
0.05110	DAT	0.00117	SEQSLT
0.192436	**AVERAGE**	**0.1191416**	**AVERAGE**

Table 7. Most informative attributes for generations 1–4, together with GR and IG scores

Generation 1

Attribute	GR	IG
JMN	0.47461	0.11450
SLT	0.41082	0.04551
MOVSUB	0.13870	0.05287
MOVADD	0.12117	0.04122

Generation 2

Attribute	GR	IG
MOVADD	0.48960	0.34157
ADD/SUB	0.37800	0.34566
MOVDJN	0.22310	0.43554
SPLMOV	0.22280	0.56428
DJN	0.20880	0.43173
SPL	0.20350	0.54848

Generation 3

Attribute	GR	IG
SPLMOV	0.36100	0.47913
MOVDJN	0.33800	0.31149
DJN	0.31500	0.32649
Imp spec	0.26400	0.18135
SPL	0.22700	0.49328

Generation 4

Attribute	GR	IG
SPLMOV	0.52450	0.56816
MOVJMP	0.43380	0.44600
MOVDJN	0.42800	0.33516
DJN	0.41400	0.33717

6 Conclusions and Future Work

Exploration and generation of CoreWar warriors, assisted by computers, has become increasingly popular in the recent years. Majority of work, however, has concentrated on warrior parameter optimization [7] and the evolution of competitive warriors [8,5,9]. Exploratory analysis (albeit motivated by warrior evolution), by means of automatic categorization based on the analysis of execution frequencies of certain instruction types during simulation, was performed, with some results available in [2], but with no published findings. In the research described in this paper, on the other hand, we attempted to utilize a *static* (source-based) instead of a *dynamic* (execution-based) approach to the analysis and categorization of a set of warriors. The used dataset was the result of warrior evolution conducted by the CCAI evolver [5].

The clustering of the CCAI evolver output was done using the EM algorithm incorporated in the WEKA workbench. Three clusters were detected in the complete dataset. This indicates that the overall diversity of the complete dataset was rather low, which can be explained by the fact that it is difficult for evolutionary algorithms to generate complex structures within the warriors in the evolved population, because small changes and mutations usually render good complex warriors useless, and there is a huge gap between different warrior strategies. Therefore, the most mutation resistant forms prevailed, namely replicators and coreclears.

The complete dataset was divided into 4 subsets, in chronological generational order. After processing, 2, 4, 12 and 5 clusters had been found in generations 1, 2, 3 and 4, respectively (see Table 5). The general tendency of this result was expected, because of varying mutation rates which were decreased at the end of the evolution process, producing a general decrease of diversity in the evolved population.

Information gain and gain ratio analysis showed that 'DJN' and 'MOVDJN' were the most significant attributes in the clustering of the whole dataset (see Table 6). 'SPLMOV' and 'MOVJMP' were also important in clustering of some of the subgroups. This can be explained by the fact that most of the warriors in the dataset were either replicators or coreclears, and these instructions and instruction pairs are seen quite frequently in such warriors.

Attribute analysis generation by generation also showed consistency with the way the evolver works. The greatest changes were exhibited between the original pool and the next generation, and attribute evaluation did register large differences in the informativeness of attributes.

It is also possible to cluster warrior sets according to the scores of evolved warriors against a predetermined benchmark. A diverse benchmark of human-coded warriors manually annotated with their types was created for this purpose, and the score tables have already been generated. The clustering according to the score tables will be conducted and the results compared to those obtained via source-based clustering described in this paper.

An issue with the static source-based warrior representation used in the presented work may be the "garbage" often left over in the source code of evolved warriors – instructions which never actually execute, but effectively introduce noise to the representation. Warriors written by humans, on the other hand, are usually "clean" in this sense. Dynamic representations based on counts of instruction execution are able to

deal with this kind of noise, but at the expense of a considerable increase in warrior preprocessing time.

The noted correspondences between the workings of the evolutionary algorithm and clustering indicate that our choice of static warrior representation was to some extent appropriate. However, in order to determine exactly to what extent, and whether the syntax analysis can produce good categorization of evolved warriors, precise measurements are necessary. This may be done through comparison of source-based and score-table-based clusterings, and additionally by training classifiers and comparing classification results with the clusters.

The warrior population evolved by the CCAI evolver was not as diverse in a strategic sense as any human coded warrior group. To see how well clustering and classification algorithms can cope with more diverse datasets, and also to see if the data representation chosen in this project does well in such situations, the whole process will be repeated on some human coded warrior set. Being that human coded warriors often mix several strategies, it would be especially interesting to use probabilistic methods to gain insight into the probabilities of a warrior belonging to classes which were previously identified and annotated. A comparison of static and dynamic representations, on both human-coded and evolved warrior datasets, should then give more definitive answers concerning the feasibility and applicability of automatic warrior categorization.

References

1. Dewdney, A.K.: Computer recreations: In the game called core war hostile programs engage in a battle of bits. Scientific American **250**(5) (1984) 14–22
2. 'Varfar', W.: Wilfiz scores of warriors on the 94nop. `http://redcoder.sourceforge.net/?p=kepler-wilfiz`
3. Tomašev, N., Pracner, D.: Categorizing corewar warriors. Seminar paper, Department of Mathematics and Informatics, Faculty of Science, University of Novi Sad (2006)
4. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques. 2nd edn. Morgan Kaufmann Publishers (2005)
5. Vowk, B.: CCAI. `http://www.math.ualberta.ca/~bvowk/corewar.html`
6. Whitley, D., Rana, S., Heckendorn, R.B.: Island model genetic algorithms and linearly separable problems. In: Selected Papers from AISB Workshop on Evolutionary Computing. LNCS 1305, London, UK, Springer-Verlag (1997) 109–125
7. Zap, S.: Optimax. `http://www.corewar.info/optimax/`
8. Corno, F., Sanchez, E., Squillero, G.: Exploiting co-evolution and a modified island model to climb the core war hill. In: Proceedings of CEC03 Congress on Evolutionary Computation. (2003) 2222–2229
9. Hillis, D.: Redrace: Evolving core wars page. `http://users.erols.com/dbhillis/`