Automatic Categorization of Human-coded and Evolved CoreWar Warriors

Nenad Tomašev, Doni Pracner, 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
tomasev@nspoint.net, doni@neobee.net, {radacha,mira}@im.ns.ac.yu

Abstract. CoreWar is a computer simulation devised in the 1980s where programs loaded into a virtual memory array compete for control over the virtual machine. These programs are written in a special-purpose assembly language called *Redcode* and referred to as *warriors*. A great variety of environments and battle strategies have emerged over the years, leading to formation of different warrior types. This paper deals with the problem of automatic warrior categorization, presenting results of classification based on several approaches to warrior representation, and offering insight into ambiguities concerning the identification of strategic classes. Over 600 human-coded warriors were annotated, forming a training set for classification. Several major classifiers were used, SVMs proving to be the most reliable, reaching accuracy of 84%. Classification of an evolved warrior set using the trained classifiers was also conducted. The obtained results proved helpful in outlining the issues with both automatic and manual Redcode program categorization.

1 Introduction

Artificial life research focuses on the exploration of various artificial environments and possibilities for self-organization, adaptation, competition, and most importantly survival in these simulated conditions. Some of those artificial worlds resemble in a way the world we inhabit and are meant to be simplified models of reality, where interesting changes in system dynamics can be observed as the parameters change and the system shifts from one stable state to another. Other artificial worlds are completely different, posing novel problems to be solved by evolutionary processes of self-adjustment through mutation and selection. Several such systems were inspired by CoreWar.

CoreWar was introduced by A. K. Dewdney in 1984 in an article published in Scientific American [1]. International CoreWar Society (ICWS) was formed a year later. CoreWar was based on a game called *Darwin* developed in Bell Labs in 1960, devised by Victor Vyssotsky, Robert Morris Sr. and Dennis Richie. In CoreWar, several programs, referred to as *warriors*, attempt to survive in a looping memory array, avoiding attacks and at the same time trying to eliminate the opposition. The warrior that takes complete control of the process queue wins the battle. A match between two warriors consists of a number of such battles, each time varying the initial positioning in the memory array which is referred to as *the core*.

Over the years, many different battle strategies have emerged, taking advantage of various possibilities embedded in the Redcode language and battle parameters' numerical properties. Some warriors are easily categorized by the combination of strategic components in their code, but finding clear distinctions between warrior types is generally not an easy task due to fuzzy borders induced by the existence of hybrid strategies. Virtually all modern warriors are optimized automatically by some performance improving software. Instruction copy distribution is the main optimizing issue, but code placement and phase thresholds are also significant. The size of the parameter space calls for automation of many warrior development phases. Apart from optimization software, automatic warrior generators were also created, utilizing evolutionary algorithms to create functioning warriors out of sets of randomly generated code sequences.

Performance evaluation in warrior generation is usually done via testing against some predetermined benchmark warrior set of considerable size. To reach reliable score estimation in standard environment settings, at least 250 battles per match-up are required. In evolvers, when a lot of new warriors are being constructed in each generation, determining fitness becomes a very time demanding process.

Automatic warrior categorization would be very useful in CoreWar evolutionary software in order to design control mechanisms for mutation rate adjustment. Mutation rates are usually set to high values in the beginning, slowly dropping to some predetermined constant values. High mutation rates allow the creation of a greater variety of forms, while the low mutation rates instigate convergence of the generation pools to the fittest among the generated types. However, the current lack of automatic categorization renders diversity supervising practically impossible. If one could keep track of the diversity shifts in the evolution process, it would be possible to dynamically change the mutation rates in some selected generation pool subsets.

The goal of the research presented in this paper is to explore the possibilities for automatic warrior categorization using representations based on syntax analysis and benchmark scores. The former may quickly and easily be calculated, while the latter are an essential part of fitness evaluation in CoreWar evolvers. The addressed issues include choosing warrior types from a plethora of possible distinctions depending on the desired level of abstraction, specifying the representations, manually categorizing a warrior set while ensuring reasonably proportional strategic distributions, conducting automatic classification and testing it on both human-coded and evolved warrior sets. Our main intention was to spot the obstacles in the categorization process, so that further improvements to the representations could be made in order to overcome those difficulties and, more importantly, to assess the feasibility of automatic warrior categorization. This project is the first attempt to achieve the above mentioned goal using supervised machine learning methods, and also the first to introduce a fully labeled warrior dataset.

The rest of the paper is organized as follows. Section 2 explains the essentials of CoreWar and the Redcode language, while Section 3 discusses possibilities for representing warriors in a form suitable for analysis and describes the details concerning the used warrior benchmark. Section 4 deals with the issues related to both human-coded and evolved warrior datasets used in this research. The analysis of categorization itself is given in 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 compete in a virtual cyclic memory array. These programs, referred to as *warriors*, are written either in Corewars (by the 1988 ICWS standard) or Redcode (by the 1994 ICWS standard). Possible syntax extensions of Redcode have consequently been discussed in the CoreWar community, regarding proposals of new sets of instructions or even new simulation concepts. This potentially new standard is known as *Bluecode*. Warrior confrontation takes place in a virtual memory array called *the core* which is wrapped around so that the successor of the last address in the core is the first one. Execution of instructions and management of threads is performed by the *memory array redcode simulator* (MARS). All addressing is relative and all arithmetic is modular with respect to the size of the core.

A warrior's goal in most competitions is to take complete control of the core by forcing all its opponents to eliminate their threads of execution from the process queue. At the beginning of the simulation the core is initialized to `DAT $0,$0`. Warriors can read from the core, write to the core, perform basic arithmetic instructions, create new threads, mutate, go through numerous stages in their ontogeny, copy themselves, actively search for their opponents, etc. Between rounds, the result of the previous battle is stored in a separate memory array called *Pspace*. Some warriors can read the results from Pspace, and modify their strategies for future rounds, keeping track of the employed approaches by coding their decisions into secret Pspace locations.

Competitions are held regularly on several Internet servers. Leagues are commonly referred to as *hills* and function as fixed-size tournaments where new warriors push out older ones quite often. There are several important parameters defining these standard competitions, namely: size of the core, number of cycles before a draw is declared, number of threads allowed per warrior, warrior size restrictions, read/write limits and Pspace restrictions. The standard parameters (hillkey: 94) are given in Table 1, along with other common settings. The most popular hill is certainly the 94nop hill, differing from the standard only respective to Pspace. Even among the usual hills there is great variance in environment properties, restricting effectiveness of certain warrior types. The rules of periodical tournaments tend to be more diverse, often placing additional constraints on submitted warriors. There is virtually an infinite number of possible simulation environments, affecting overall performance of specific warrior types, allowing greater or lesser complexity and more or less intelligent strategic approaches. It would therefore be practically impossible to categorize CoreWar warriors in a general case. This paper deals exclusively with warriors designed for the 94nop hill.

The Redcode language. According to the ICWS'94 standard, Redcode is the default language for writing CoreWar warriors. It contains 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. The more important instructions include DAT, which is used both to store data and remove the thread executing it from the process queue; the copying instruction MOV; arithmetic instructions ADD, SUB, MUL, DIV and MOD; unconditional jump instruction JMP and conditional jumps JMZ, JMN and DJN; and the thread-creating instruction SPL. There are many combinations of the mentioned elements, more precisely $8512 \cdot \mathrm{CORESIZE}^2$, although a portion of the command set

Table 1. Common hill configurations

Parameter	94	94nop	94t	lp	94x	tiny	nano
CORESIZE	8000	8000	8192	8000	55440	800	80
NUMPROC	8000	8000	8000	8	10000	800	80
MAXLENGTH	100	100	300	200	200	20	5
MINDIST	100	100	300	200	200	20	5
NUMCYCLES	80000	80000	100000	80000	500000	8000	800
R/W restrictions	none	none	none	none	none	none	none
Pspace	true	false	true	true	true	true	true

exhibits equivalent behavior but would be considered different by some instruction-comparing Redcode instructions and thus cannot be regarded as completely equivalent. Nevertheless, it is apparent that such a small instruction set allows for much creativity in warrior design. In fact, the set of all possible nonequivalent Redcode programs in the standard setting is of cardinality $100 \cdot (5447680000000^{100})$. A more detailed description of Redcode is available in [2].

Warrior types. Contemporary warriors are highly sophisticated, a result of over two decades of continuous improvements over the basic ideas, and occasional ascension of new concepts. Most of them represent combinations of certain strategic elements, balanced and optimized to increase durability and reduce variance of score when opposing different warrior types. Some common *strategic concepts* are summarized below.

Imps are among the simplest of components, yet quite often used due to the fact that disposing of them requires some special mechanisms taking valuable time and space. Imps consist only of MOV instructions, copying themselves sequentially through the core, commonly forming structures known as *rings* and *spirals* of various sizes and properties. Most of these structures require multiple threads in order to function properly, which is the only limiting factor in their use. The field values in MOV instructions required for being a part of a spiral or ring structure are called *imp numbers*. Imp numbers of the standard spirals are divisors of 1, respective to modulo $\mathrm{CORESIZE}$. The order of such multiplicative group defined by the mentioned imp number constitutes the minimal number of threads required for proper functioning of the structure.

Bombing is a process of copying some predetermined instructions (referred to as *bombs* in this context) throughout the core with the intention of overwriting a part of the opponent's code and forcing the thread executing it either to remove itself from the process queue or perform some action useful to the bombing warrior. There are many different kinds of bombs specially designed to harm certain opponent types.

Replication is a process performed by warrior components constantly copying themselves and creating new threads to run those copies. The basic idea is to overwhelm the opponent by sheer strength of numbers. Optimized replicators are among the strongest of warriors even today, regardless of many more sophisticated strategies in competition. Combining replicators and imps also proved to be quite a successful approach.

Core clearing is a process of sequential overwriting of the core with some predetermined instruction. It is customary to change the instruction in question after a certain number of cycles is reached. This is usually used as an endgame strategy. There are

several basic types of such components, varying greatly in their performance against different warrior types.

Scanning denotes searching for opponent's code in a heuristic manner, by comparing pairs of instructions, or instruction fields. Scanners attack only those locations where they detect a possibility of enemy code presence, and are therefore able to use more sophisticated types of attack.

Bootstrapping is a process of quickly copying essential components away from the original code to avoid detection.

Stealth is a masking technique where a warrior renders a part of its code undetectable by some scanner types.

Decoy is a technique of leaving chunks of junk code at useful spots in the core, to be mistaken by scanners for potential threats, diverting them from actual enemy code.

Quickscan is a component performing exceptionally fast scanning at the start of the simulation, trying to locate enemy code in an early stage and disable it before it bootstraps and activates all its components. It is very common in contemporary warriors.

For the purposes of categorization in this paper, a relatively modest number of 14 warrior types has been selected to represent the strategic abundance of CoreWar. Choosing too many categories could have been disadvantageous, regarding the low frequency of appearance of some warrior types within any set corresponding to some of the standard environments. The considered warrior types are given in Table 2.

Table 2. Warrior categories

Type	Description
cds	Clear-directing scanners
clp	A special-purpose strategy designed to oppose oneshots
clr	Warriors basing their activity on clearing the core
clrwi	Core-clearing warriors using imp components
evo	Evolved warriors, a category denoting all automatically created warriors that do not resemble human-coded warriors enough to be considered one of the other types
onesh	Oneshots – a special class of scanners, focusing on the first potential threat
pap	Replicators, also called as *papers*, according to the stone/paper/scissor analogy
pwi	Replicators that also use imps
pws	Replicators that also use stones (see below)
sabi	Stones accompanied by both A-field and B-field imps
sai	Stones accompanied by A-field imps
sbi	Stones accompanied by B-field imps
scn	Scanners other than clear-directing scanners and oneshots. All three classes together are referred to as *scissors*
stn	*Stones* are warriors utilizing a bombing strategy. Their name is derived from the stone/paper/scissor analogy

3 Warrior Representation

One of the main issues in automatic CoreWar warrior categorization is certainly representing warriors in a form suitable for analysis and application of machine learning techniques. The code itself can be viewed as a genotype, while the associated behavior in a certain core corresponds to the phenotype of a warrior. Same warrior code can display different properties in different environments and even belong to different warrior types in the respective core settings! Therefore, it is the emergent behavior that outlines the category generalizing the strategic concepts of a warrior in a certain environment. In the rest of this paper the concept of warrior types will be regarded in this manner, only relative to the 94nop setting considered during the research.

The question arises whether it is possible to draw conclusions about warrior phenotype given the parameters of the observed environment, based on observations of code alone. If it were possible to classify a warrior based on a representation derived from syntax analysis, such a process would be favorable in terms of execution time, and therefore preferable for use in systems performing a lot of calculations, e.g. evolvers.

In the first phase of this research, a simple syntax-based representation was evaluated comprised of frequencies of appearance of instruction names in warrior code, and also of frequencies of some instruction pairs characteristic for some components and warrior types. Instruction pairs used in the representation are: SPLMOV, MOVJMP, MOVDJN, MOVADD, MOVSUB, SEQSNE, SNEJMP and SEQSLT. Instruction modifiers and field addressing modes were not included in the representation. It was thought that too much of the semantics would have been lost that way due to functional equivalence of some commands, and the same holds for the respective instruction pairs. Along with the aforementioned features, a boolean flag called *Impspec* was added to carry information about the potential imp presence within a warrior. We shall refer to this representation as *static*.

In the second phase, a representation formed by benchmark scores was used. The benchmark for this purpose was carefully chosen, more details are given at the end of this section. Performing the confrontations against benchmark warriors takes some time, but it was reasoned that since benchmarking is almost exclusively used in automatic warrior generators as a fitness estimation technique, the scores would already be available. This representation consists of win and loss percentages of the tested warrior against each of the benchmark warriors, and will be denoted *dynamic*.

After both above mentioned representations were tested, a hybrid representation combining the former was used (denoted *combined*). It was additionally attempted to extend the representation with another boolean feature named *Qspec*, containing information regarding the presence of a quickscan component within a warrior. Unlike with *Impspec*, we still have no satisfactory algorithmic solution for determining the value of *Qspec* for an arbitrary warrior, and thus resorted to manual labeling for this attribute. For this reason *Qspec* treated separately from other attributes.

The static representation suffers heavily from its inability to distinguish code that will be executed during the simulation from decoys. Of course, every instruction present in the code affects the reactions of enemy warriors during confrontations, including decoys and other junk code. There was an attempt in the past to overcome this problem by observing frequencies of command executions during simulations [3]. Observing

command execution frequencies may seem to be a good solution, but it has its own downfalls. Namely, when a warrior is placed in a *melee* mode (being the only code loaded into the core), there is a strong possibility that some parts of otherwise active and functional code will never be executed, because no enemy is ever detected. If, on the other hand, a warrior is set to confront some other warriors, it will also be executing commands that other warriors might copy over the location of its code. In that case, such execution can be easily mistaken for execution of its own code and there is no clear way to make a distinction using only elementary techniques that are not time consuming. Since syntax analysis takes the entirety of warrior code into consideration, it does not fall prey to the described problem.

The syntactic part of the representation used in this research is, naturally, only one of many possible solutions. Some improvements will be considered in the future, and the new directions are discussed in Section 6.

The representation benchmark. Two requirements needed to be considered while composing the benchmark for the dynamic part of warrior representation. First, scores against benchmark warriors should be good category indicators. For that to be achieved, warriors must be selected that perform particularly good or particularly bad against certain specific warrior types on average, and achieve close to average scores against most of the other categories. Second, for the benchmarking to be applicable to the required case – as a fitness estimating step in automatic warrior generators – some additional constraints need to be imposed upon the warriors used. The benchmark needs to be balanced in terms of contained warrior types. The benchmark size needs to be within certain limits, to reduce the probability of score oscillations due to parametric dependencies between tested and benchmark warriors, and not to have too great an impact on total evolver execution time.

The benchmark used in this research is comprised of 30 warriors proportionally representing all major strategic types. Special attention was given to warriors performing exceptionally well against A-field or B-field imp structures within certain warrior categories. Note that the distinction between these two types of imp-structures could have been achieved by the static part of the representation. However, it was considered that imps play an important role in warrior behavior and that it is of vital importance to be able to detect not only the potential to create imps in simulation (which is apparent from the *Impspec* attribute) but the structures themselves if they are created in the simulation. Some oneshots achieving truly amazing scores against most types of replicators were also inserted. Warriors containing imps also play an important role, since their scores help make a distinction between warriors that posses some kind of imp-destroying mechanism and the ones that do not.

4 The Datasets

Two warrior sets have been used in this research. The first one, denoted `h1c`, represents a subset of `94nop` Königstuhl set [4]. Königstuhl is an infinite hill where players who decide to share code send final versions of their projects. Inefficient and erroneous warriors were removed from this list to obtain the `h1c` set. All warriors were manually

categorized. The dataset consists of 666 warriors (no pun intended), which are unfortunately not evenly distributed among the considered categories (Table 3). This is related to the fact that some strategies are simply more popular in the CoreWar community. Surprisingly, no instances of the clp class were found. Clps are occasionally used as a component in Pspace-using warriors, and are seldom used alone. There were some clps on the hills in the past, but it seems that none of them found their way to 94nop Königstuhl. Classes with a small number of instances were clrwi and sabi. Balancing of parameters in both of these warrior types is not an easy task, which explains the low frequency of their appearances on the hills, and consequently in h1c. The average warrior length in the dataset disregarding data storing instructions was 35, far below the 100 instruction limit imposed by the 94nop setting. Massive use of quickscanners as an early stage strategic component led to the minimalistic approach in coding to avoid early detection. MOV and SPL were the most commonly used instructions, with the average frequencies of appearance of 9.82 and 7.66, respectively.

Table 3. Class distribution of the h1c dataset

cds	clp	clr	clrwi	evo	onesh	pap	pwi	pws	sabi	sai	sbi	scn	stn	Total
47	0	20	12	66	73	100	40	40	6	33	39	73	117	666

The considered evolved warrior set is a subset of the output generated by the CCAI evolver [5], which was written by Barkley Vowk from the University of Alberta in summer 2003. An island model was used in the evolutionary algorithm [6]. The benchmark used to measure fitness was Optimax [7].

We have already subjected the complete CCAI output to clustering using the static representation in [2]. The respective size of that dataset is 4389 warriors.

5 Warrior Categorization

Classification was performed using the WEKA machine learning workbench [8]. The following classifiers were included in the experiments: SMO – an implementation of the sequential minimal optimization algorithm for training support vector machines [9], which performs multi-class classification by training a binary classifier for each pair of classes; MultilayerPerceptron – a neural network classifier trained using backpropagation [10]; J48 – a decision tree learner based on revision 8 of the C4.5 algorithm [11]; NaiveBayes [10]; BayesNet – a Bayesian network with automatically determined structure as the maximum weight spanning tree [12]; and IBk – which implements the classical k-nearest neighbor algorithm [13]. After initial tests we elected to use SMO with the linear kernel, MultilayerPerceptron trained in 500 epochs with one hidden layer containing $1/2$ of the total number of input and output nodes, and IBk with $k = 5$ neighbors and reciprocial distance weighing when determining the class from the neighbors.

Categorization of human-coded warriors. Figure 1 summarizes the performance of the considered classifiers in experiments involving 10 runs of 10-fold cross-validation

on the h1c dataset. The highest accuracy was exhibited by SMO, 84.26%, on the representation including both static and dynamic components, including specification of quickscanner presence. Generally, the use of the *Qspec* attribute only slightly improved the performance of classification with all representations, and its contribution could not be statistically verified using the corrected resampled t-test, at $p = 0.05$.

Performance of all classifiers except for the MultilayerPerceptron was statistically verified as worse when compared to SMO on the complete (combined+Qspec) representation. NaiveBayes proved to be the worst among the tested classifying methods, reaching accuracy of only 64.35%. Apparently, it was not able to cope with the dependencies between attributes, both static (concerning co-occurrences of instructions) and dynamic (regarding the pairwise and other dependencies between win and loss percentages), which may be observed on the partial representations.

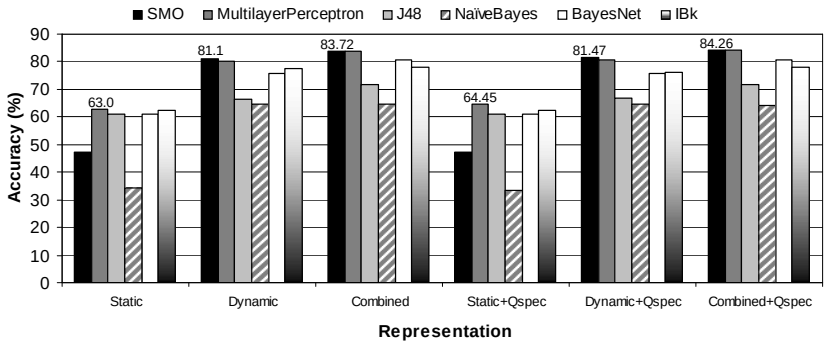

Fig. 1. Performance of various classifiers on the h1c warrior dataset, by representation

The classification success rate varied respective to warrior categories in question. Replicators were easily recognized. Most classifiers managed to make a clear distinction between stones using A-field and B-field imps (classes sai and sbi, respectively). It follows that such distinction can be made through use of special-purpose benchmark warriors, since no such distinction had been made possible in the static part of the representation. Some scanners belonging to the scn class had been confused with stones. The scanners in question were mostly the less efficient ones, thus differing in crucial benchmark scores from the rest of their group, instead scoring similar to some *incendiary stones*. The lowest accuracy was present in classification of warriors belonging to those categories represented by a small number of instances in the dataset, namely clr, clrwi, sabi. Core-clearing warriors are not so common, due to their lack of adaptability to various enemy strategies, and are mostly used as components in other, larger and more complex warriors. There are also several subcategories scoring quite differently against most other strategies. The total of 20 instances proved to be too few to ensure automatic recognition of that warrior group. Breaking the clr class into several distinct classes will also be considered in the future. It is also worth mentioning that most of the evolved warriors were correctly classified as evo.

Classifier accuracy on the complete representation was superior over isolated use of static features. Also, adding the static feature vector to the dynamic representation improved classification results significantly in cases of SMO, MultilayerPerceptron, BayesNet and J48 classifiers.

Even though SMO performed best on dynamic and combined representations, its accuracy of 47.15% on the static representation was significantly inferior to the performance of J48, BayesNet, IBk and MultilayerPerceptron. This can be partially remedied by employing polynomial kernels of higher degree (4–6), with the performance being able to reach that of the best classifiers, but at the expense of worsening accuracy on dynamic representations. The highest accuracy of classification on the static representation was achieved by MultilayerPerceptron – 64.45%.

Regardless of the apparent advantages of both score-based and combined representations over the static representation, the obtained results indicate that classification according to static features alone might be possible in the future if some modifications were made. First of all, according to the current static representation, it is absolutely impossible to distinguish A-field imps from B-field imps. That can easily be solved by adding new imp presence indicators. Also, it appears that the use of characteristic instruction pairs was insufficient in carrying information about the context in which instructions were used. Possible solutions to that problem will be considered in Section 6.

To analyze the effects of different attributes to class affiliation, *information gain* and *gain ratio* evaluators were utilized. Due to the known shortcomings of both evaluation methods, the approach was used where gain ratio is considered, but only if the respective information gain is above average [8]. *Impspec* was determined to be of greatest importance in the `h1c` warrior set. Practically all other attributes with high gain ratios belonged to the dynamic part of the representation. It is interesting to note that those attributes represent almost exclusively the winning percentages of special-purpose warriors in the benchmark inserted with the intention of detecting imp structures. Due to the high significance of these structures in determining warrior types, this is not surprising. The attribute evaluators were also utilized to estimate the importance of attributes respective to *Impspec*, and once again the same attributes were determined to have high gain ratios, differing from the class-based scores only in their relative importance.

Categorization of the evolved dataset. Generation 4 of the CCAI dataset was already clustered [2], so the categorization of that warrior set was meant both as a test of the reliability of trained classifiers and also to provide insight into the structure of the dataset and its clusters. Clustering had been done using the static representation, without *Qspec*. For purposes of classification in the current research, benchmark scores for generation 4 were produced, and the classifiers trained on the combined representation. A random 400-warrior sample was extracted from the dataset and manually categorized.

Evolved warriors usually greatly differ from their human-coded adversaries. One of the main characteristics of evolved warriors is the presence of considerable amounts of junk code. Evolved datasets consist mainly of mutation resistant forms – core-clearing warriors and replicators being the dominant types [2]. Also, such warriors rarely utilize advanced strategic tricks, which distinguishes them from analogous human-coded warrior types. However, the final generation of the CCAI set exhibits somewhat different properties. The evolver in question was not merely devised as an experiment – instead

it aimed to generate strong, competitive warrior forms. The final product was a famous warrior that defeated many human-coded state-of-the-art warriors. It was appropriately named *Machines Will Rule*. Hence, warriors in the CCAI set tend to be stronger than typical evolved warriors and bear more resemblance to their human-coded counterparts.

Manual inspection detected only two classes in the 400-warrior sample of the CCAI set: pap and pwi. Replicators were mostly using anti-imp core-clearing techniques, and achieved great scores against imp-type warriors in the benchmark. Imp-containing replicators, however, were not nearly as well optimized and rarely benefited from the presence of defensive imp structures.

The following classifiers were tested: SMO, MultilayerPerceptron, BayesNet, and IBk, with the results summarized in Table 4. The highest accuracy on the sample was achieved by the BayesNet classifier – 96.5%. SMO and IBk had also proved satisfactory, both reaching accuracy of 88.5%, and performing very similar classifications of the complete dataset. On the other hand, MultilayerPerceptron was considerably worse, with accuracy of only 53.25%. Most of the misclassified instances were interpreted as either pws or pwi by MultilayerPerceptron. That was generally a consequence of imps and also arithmetic instructions being often present within the junk-code in the evolved warriors. The misclassification rate of the other tested classifiers mostly originated from pap being interpreted as sbi, which was initially quite surprising. Such results were later attributed to the presence of imps within the junk code, as well as high resistance of the warriors in question to some common scanner attack techniques.

Table 4. Categorization of generation 4 of the CCAI dataset, with class counts on the complete dataset and the sample, and accuracy calculated on the sample

	clr		evo		pap		pwi		pws		sbi		stn		
Classifier	all	smp	all	smp	all	smp	all	smp	all	smp	all	smp	all	smp	**Accuracy**
SMO	0	0	0	0	3703	359	24	0	2	0	467	41	0	0	88.50%
M. Perceptron	4	0	108	11	2120	217	180	14	1529	136	254	21	1	1	53.25%
BayesNet	0	0	0	0	4086	392	103	7	0	0	7	1	0	0	96.50%
IBk	0	0	0	0	3695	359	48	3	0	0	453	38	0	0	88.50%
Manual	–	0	–	0	–	392	–	8	–	0	–	0	–	0	100%

All of the warriors from the sample misclassified by either of the classifiers as sbi or pwi were examined. In 37% of the instances, an interesting structure was discovered, used by the replicators as a strong anti-imp feature, but also forming some sort of imp-like structure, thus enhancing defensive capabilities. Such pseudo-spirals were set up in a similar fashion to imp spirals, the difference being in the instruction used, namely `MOV.I #1169, }2667`. Apart from this interesting replicator subtype, core-clearing papers were quite frequent in the considered sample.

As for the syntactic clustering described in [2], we are now able to conclude that it was unable to detect the subtle differences between the two classes in the dataset, being mislead by junk code within the warriors. However, this does not mean that the clustering was not a good indicator of the diversity of warrior genotype, since junk code can also be combined in future generations to produce working warriors.

6 Conclusions and Future Work

Recently, attention of the CoreWar community shifted from devising new tricks in the existing strategies to exploring new settings, parameter optimization [7], and automatic warrior generation. Quick and reliable warrior categorization would be of great importance in many such automated optimizing systems. The results obtained from this research indicate that automatic categorization is indeed possible to achieve, at least in the standard 94nop environment. However, further research is necessary in order to improve classification accuracy and possibly form a more universal categorization model, applicable to a wider range of environments. We believe that changing the static part of the representation alone may be enough to ensure the desired increase of accuracy. The benchmark could, of course, be revised and constructed exclusively of special-purpose warriors, but it would inevitably lead to the decrease in its general performance-estimating ability. On the other hand, adding new n-grams to the representation, as well as modifying and decomposing the *Impspec* feature would certainly improve static-based categorization, and probably the combined representation as well.

A completely different approach to warrior categorization would be not to try detecting a fixed number of warrior types, but strategic components instead. This would solve the problem of classifying some hybrid warriors. Another problem that will be considered in the future is automatic categorization based on the round-robin score table as the dynamic component in the representation, rather than benchmark scores, to facilitate effective categorization in large tournaments.

References

1. Dewdney, A.K.: Computer recreations: In the game called core war hostile programs engage in a battle of bits. Sci. Am. **250**(5) (1984) 14–22
2. Pracner, D., Tomašev, N., Radovanović, M., Ivanović, M.: Categorizing evolved corewar warriors using EM and attribute evaluation. In: Proc. MLDM'07, 5th Int. Conf. on Machine Learning and Data Mining in Pattern Recognition, Leipzig, Germany (2007) Forthcoming.
3. 'Varfar', W.: Wilfiz scores of warriors on the 94nop. `redcoder.sourceforge.net/?p=kepler-wilfiz`
4. Birk, C.: CoreWar "Koenigstuhl". `www.ociw.edu/~birk/COREWAR/koenigstuhl.html`
5. Vowk, B.: CCAI. `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. Zapf, S.: Optimax. `www.corewar.info/optimax/`
8. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques. Second edn. Morgan Kaufmann Publishers (2005)
9. Platt, J.: Fast training of support vector machines using sequential minimal optimization. In: Advances in Kernel Methods – Support Vector Learning. MIT Press (1999)
10. Mitchell, T.M.: Machine Learning. McGraw-Hill (1997)
11. Quinlan, R.: C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers (1993)
12. Friedman, N., Geiger, D., Goldszmidt, M.: Bayesian network classifiers. Mach. Learn. **29**(2–3) (1997) 131–163
13. Aha, D., Kibler, D., Albert, M.K.: Instance-based learning algorithms. Mach. Learn. **6**(1) (1991) 37–66