XVII Conference on Applied Mathematics
Đ. Herceg, H. Zarin, eds.
Department of Mathematics and Informatics
Novi Sad, 2006, pp. xx–xx

CATEGORIZING COREWAR WARRIORS

Doni Pracner[1], Nenad Tomašev[1], Miloš Radovanović[1], Mirjana Ivanović[1]

Abstract. CoreWar is a computer simulation where programs, written in an assembly language called redcode, compete in a virtual memory array. These programs are referred to as *warriors*. There are several systems that automatically create warriors, among which are evolvers. This paper describes an attempt to analyze the results of an evolver. Since manual examination would be too time demanding, a code analysis approach was used, which was based on clustering and attribute evaluation techniques from data mining.

1. Introduction

There are many possible ways to create artificial intelligence, and one of the approaches is to construct computer programs in virtual environments. Sometimes these environments try to simulate the real world, often with simplified rules, to see how well the programs can adapt. In other cases, different programs compete for resources and try to eliminate their opponents. One of the oldest and most popular venus for the development and research of programs executing in a simulated environment is CoreWar, in which programs (referred to as *warriors*) try to survive in a looping memory array. The system was introduced in 1984 by A. K. Dewdney in an article in Scientific American [2]. Through more that 20 years of development, many different strategies appeared, often combining more than one method of eliminating opponents. Several attempts were made to automatically create new and better warriors, by the process of optimization and evolution. So far, the best and only reliable method to evaluate warriors was by putting them against a benchmark set of prepared test programs and then examining the results. To get reliable and stable results against each warrior 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 a benchmark is then clearly a very time demanding process.

The goal of the research presented in this paper was to examine the diversity of the warrior pools created by the CCAI evolver (Section 4) and test the possibilities of automatic categorization according to syntax analysis. The

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

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. There were some attempts in the past to perform automatic categorization, but they were based on the analysis of execution frequencies of certain instruction types during simulation, which requires the simulation to be run for a certain amount of time [5]. If the syntax-based approach proved fruitful, it would be possible to come to the same conclusions much quicker, which could, in turn, speed up the whole process of evolution. To the best of our knowledge, this paper represents the first attempt to categorize warriors using *static* (syntax-based) instead of *dynamic* (execution-based) methods.

The rest of the paper is organized as follows. Section 2 presents some machine learning and data mining techniques, focusing on clustering methods used in the paper. Then, in Section 3, essential CoreWar principles and some basic strategies of human-coded warriors are explained. Section 4 describes the data that was used and how it was processed into a representation of knowledge for actual data mining. Section 5 covers the clustering analysis that was done and its results. Finally, in Section 6, a summary of the conclusions is presented, together with plans for future work.

2. Machine Learning and Data Mining

Machine learning is a very broad subfield of artificial intelligence, and is concerned with developing algorithms and techniques that allow machines to "learn." However, the precise definition of "learning" may be somewhat problematic. The notions of learning that are valid for living beings usually can not be applied to machines. Therefore, machine learning is often described in terms of searching for patterns and rules in the data. *Data mining* is usually defined as "finding non-trivial, previously unknown and potentially useful information from data." Therefore, it does not include "philosophic" problems, because it involves learning in a practical sense, not a theoretical one.

Before the data is "mined," it is usually transformed into a form of "metadata," where each instance is described by a set of *attributes*. One of the crucial steps in the process of mining is to determine which attributes are important and will be used, and which attributes can be disregarded.

A process called *clustering* is used to group pieces of data that go "naturally" together. The problem consists of finding the groups and assigning instances to them. Results are often subjectively measured by the usefulness of the clustering to a human user.

In this research the *Expectation Maximization* (EM) clustering algorithm was used. This algorithm is based on a probabilistic method, which 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 either a Gaussian or a normal distribution, but with different means and variances. Then the clustering problem is to deduce these parameters for each cluster based on the input data (the dataset). The EM algorithm provides a solution to this problem.

In short, a procedure similar to that of *k means* ([9], pp. 137–138) is used. At the start, the parameters are guessed and then the cluster probabilities are calculated. These probabilities are used to re-estimate the parameters, and this is continued until the difference between the steps is small enough (close to zero). The first part of the process is the "expectation," because the parameters are to the expected values, and the second part is the "maximization" of the likelihood of the distributions given the data.

For more information on EM, see [9], page 265.

3. CoreWar

CoreWar is a computer simulation where programs written in an assembly language called corewars (by the '88 ICWS standard) or redcode (by the '94 ICWS standard) compete in a virtual memory array. These 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, which is usually the case. The MARS (Memory Array Redcode Simulator) controls the execution of the instructions in the core. The execution of instructions is consecutive, apart from the situations arising after execution of 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 his opponent eliminate his own thread of execution from the process queue. There are many ways to achieve this, so various 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 CoreWar warriors in the core is done at random, and a predetermined number of fights is staged to decide the winner (3 points are awarded for a win, 1 for a draw, 0 for a loss). The result of the previous fight is stored in a separate memory array called P-space between the rounds. In some competitions, the warriors are allowed to access this memory, as well, and change their strategy if necessary, to ensure better performance in the future rounds.

CoreWar was introduced by A. K. Dewdney in 1984, [2]. 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 the battle parameters, and each of these is called a hill. The warrior currently holding the first place is, naturally, called *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 which were able, on several occasions, to outperform the warriors coded by humans. This is usually done via implementation of genetic algorithms.

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, 8 addressing modes and 7 instruction modifiers. The warrior files are stored on the disk as *WarriorName.RED*.

The redcode instruction set, although not huge, allows 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, and Fig. 1a shows the source of an example warrior.

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 1: Overview of some redcode instructions.

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 have no offensive capabilities, and are seldom used on their own.

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

Stones simply copy DAT instructions over the core, following some sort of plan. Up to this moment, many alternate approaches have been 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. 1a 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 warrior is usually able to spend more time on the attack against that location.

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

Generally, each non-hybrid type of warrior is effective over some 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 [4].

4. The Dataset

The data analyzed in this paper represent a subset of warriors generated by the CCAI [6] evolver created by Barkley Vowk from the University Of Alberta in summer 2003. The evolutionary approach used in this evolver was the island model [8].

The dataset consists of 26795 warrior files, and is summarized in Table 2. The data was divided into 4 smaller parts, chronologically. The respective sizes of these parts are 10544, 6889, 4973, 4389. These parts will be referenced in

the text simply as "dataset 1", "dataset 2", etc. The first pool was randomly generated, and the others represent the generations in evolving. One of the reasons that each group is smaller 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 [10].

Selecting the representation. To perform any data mining analysis, it is needed first to format the data into a *representation of knowledge* in a way that the software could understand. Usually these are n-dimensional vectors representing, in each dimension, a value for a certain attribute. For each data object a vector is created. To transform the data into vector form a Java command line application was written, details of which are presented in [4].

When selecting a working representation of knowledge, the decision was made to use a vector with just the bare instructions being counted. The resulting vector has 19 coordinates (dimensions), one for each of the command types. Also, the name of the warrior was added as a coordinate. Then some modifications were added, to make the information more specific to redcode, and the first alteration was to treat *ADD* and *SUB* as the same instruction, since they can do the same thing 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. Eight pairs of these two-command combos were added to the representation of knowledge, 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.

Removing duplicates. Another method to speed up the calculations, and on the other hand, improve the results, is removing "too similar" warriors. When clustering the data, the warriors that 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 2 summarizes the results of the duplicates search. Most duplicates were removed from datasets 1 and 2: 12% and 8%, respectively. From dataset 3 only about 1% of the files were removed as duplicates, and in set 4 about 4%. In summary, a total of 2150 duplicates were found, which is about 8% of the

```
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)

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

initial 26795 warriors.

dataset	files	duplicates	percent	time (m:s)
1	10544	1345	12%	66:00
2	6889	559	8%	44:25
3	4973	56	1%	29:19
4	4389	193	4%	188:55*
complete	26795	2153	8%	N/A

*old method used

Table 2: Summary of datasets and results of duplicate detection.

5. Clustering

Clustering was done by using the Weka [7] (*Waikato Environment for Knowledge Analysis*) free software package. Weka was developed at the University of Waikato, New Zealand. The software works in the Java programming language, and comes with a set of available classes and methods for use on datasets. The method used in this work for data analysis was the EM clusterer (see Section 2).

The number of discovered clusters per dataset and the number of instances per cluster are given in Table 3. In dataset 1, only two clusters were found. After going through some of the warriors in this dataset, it appears that the

two clusters that were found contain exactly replicators in one, and coreclears in the other. This was determined by taking random samples from the clusters, and it showed promising results. The only way to confirm this is to manually examine all of the warriors, which we considered infeasible.

dataset	clusters	cluster sizes											
1	2	8081	1146										
2	4	3456	1857	572	468								
3	12	88	2112	644	543	526	47	671	36	47	103	38	80
4	5	2364	1197	357	94	184							
complete	3	6571	2671	15469									

Table 3: Clusters per dataset and number of warriors per cluster

The number of clusters increases in datasets 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 datasets 2 and 3. The clustering algorithm did notice this.

In the last dataset, the fourth, the number of clusters decreased to 5. This was probably due to the reduction of diversity in the warriors that happened at the end of the process of evolution.

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

Attribute evaluation. To analyze the effects of different attributes to the cluster selection, two features from Weka were used, namely, the *information gain (IG)* and *gain ratio (GR)* attribute evaluators [9]. Because of known shortcomings of both evaluation methods, the approach that was utilized was to choose an attribute with maximum gain ratio, but only if its information gain is larger than the average information gain for all attributes ([9], p. 105).

Table 4 summarizes the results of attribute evaluation on the complete dataset. It shows that the most informative column is 'DJN', since the 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 info 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 at it 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 the ordinary bag of instructions was used instead.

After the complete dataset analysis, an attempt to get more information on the actual evolution process was made by examining the individual datasets,

Gain Ratio			Information Gain		
0,4035	6	MUL	0,49217	12	DJN
0,382	16	SLT	0,43917	20	MOVDJN
0,3755	23	SEQSNE	0,31378	4	MOV
0,3457	22	MOVSUB	0,22883	18	SPLMOV
0,3369	8	MOD	0,2051	13	SPL
0,3152	24	SNEJMP	0,17605	26	Imp spec
0,2631	12	DJN	0,17025	3	DAT
0,2497	25	SEQSLT	0,15134	5	ADD/SUB
0,2482	10	JMZ	0,14137	19	MOVJMP
0,2407	20	MOVDJN	0,09816	14	SEQ
0,1862	17	NOP	0,09345	22	MOVSUB
0,1827	5	ADD/SUB	0,07179	9	JMP
0,1804	26	Imp spec	0,06768	21	MOVADD
0,1743	7	DIV	0,0638	15	SNE
0,1519	21	MOVADD	0,06001	10	JMZ
0,1184	11	JMN	0,04186	2	Species
0,1163	15	SNE	0,03642	6	MUL
0,1014	14	SEQ	0,02582	16	SLT
0,0935	19	MOVJMP	0,0237	17	NOP
0,0893	4	MOV	0,02184	23	SEQSNE
0,0824	18	SPLMOV	0,02126	11	JMN
0,0578	13	SPL	0,01484	7	DIV
0,0542	9	JMP	0,01138	8	MOD
0,0511	3	DAT	0,0073	24	SNEJMP
0,0105	2	Species	0,00117	25	SEQSLT
0,192436		Average	0,1191416		Average

Table 4: Gain ratio and info gain for the complete dataset.

keeping in mind that these are generations of warriors and that the first dataset is (in big part) random.

For dataset 1, the analysis showed that the most informative attribute is 'JMN', being the first selection or both info gain and gain ratio. 'SLT' is also close, followed by 'MOVSUB' and 'MOVADD' after a large gap.

A very interesting observation is that the complete dataset best attributes, 'DJN' and 'SPLDJN', are at the very bottom of the list in dataset 1. This presumably means that there was a greater variety in the original pool, which did not affect the global dataset at a greater extent.

In the second dataset, things looked different, with 'JMZ' and 'DIV' leading the GR list, but with low IG values. After filtering with the average IG, the list is as follows: 'MOVADD', 'ADD/SUB', 'MOVDJN', 'SPLMOV', 'DJN', 'SPL'. Here 'MOVDJN' and 'DJN', which were important for the complete set, appear in the list. Also, most of the best attributes from dataset 1 do not show, or are a lot lower in the list, excluding 'MOVADD' and 'ADD/SUB'. All this indicates that a lot of codes from the first dataset were excluded in the process of evolution. This is also evident in the reduction of size, dataset 2 is only 60% of the original pool's size.

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

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

Tables corresponding to datasets 1–4 are omitted due to space restrictions, and are available in [4].

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 [10] and the evolution of competitive warriors [1, 6, 3]. 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 described in [5]. In the research described in this paper, on the other hand, we attempted to utilize a *static* (syntax-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 [6].

The clustering of the CCAI evolver output was done using the EM algorithm incorporated in the Weka library. Three clusters were detected in the complete dataset.

The complete dataset was divided into 4 subgroups, chronologically. After processing, 2, 4, 12 and 5 clusters had been found in datasets 1, 2, 3 and 4, respectively (see Table 3). This result was expected, because of varying mutation rates, which decreased in the end of the evolution process, resulting in the overall decrease of diversity in evolved population.

The overall diversity in the complete dataset was rather low, which can be explained by the fact that it is difficult for the genetic algorithms to generate complex structures in the warriors in the evolved population, because small changes and mutations usually render good warriors useless, and there is a huge gap between different warrior strategies. Therefore, the most mutation resistant forms prevailed, namely replicators and coreclears.

Info gain and gain ratio analysis showed that 'DJN' and 'MOVDJN' were the most significant attributes in the clustering of the whole dataset (see Table 4). '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.

The analysis set by set also showed consistency with the way the evolver works. The greatest differences are between the original pool (dataset 1) and the first generation (dataset 2), and attribute evaluation did register large differences in the informativeness of attributes.

It is also possible to cluster the dataset according to the scores against a predetermined benchmark. Diverse benchmarks were created for that purpose, and the score tables have already been generated. The clustering according to the results within these score tables will also be conducted and the results compared to those obtained via syntax analysis clustering described in this paper.

In order to determine whether the syntax analysis can produce reasonably good clustering results, some sort of testing is necessary. It will probably be done through the comparison with the score table clustering results, more specifically, by training classifiers based on the score table clustering and then testing the classification on the syntax clustering results.

Warrior population evolved by the CCAI evolver was not as diverse 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 as 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 a class.

References

[1] F. Corno, E. Sanchez, G. Squillero, Exploiting co-evolution and a modified island model to climb the core war hill, Proceedings of CEC03 Congress on Evolutionary Computation, pp. 2222–2229, 2003.

[2] A. K. Dewdney, Computer recreations: In the game called core war hostile programs engage in a battle of bits, Scientific American, 250(5) (1984), 14–22.

[3] D. Hillis, RedRace: Evolving core wars page, http://users.erols.com/dbhillis/

[4] N. Tomašev, D. Pracner, Categorizing CoreWar warriors, Seminar paper, Department of Mathematics and Informatics, Faculty of Science, University of Novi Sad, 2006.

[5] W. 'Varfar', Wilfiz scores of warriors on the 94nop, http://redcoder.sourceforge.net/?p=kepler-wilfiz

[6] B. Vowk, CCAI, http://www.math.ualberta.ca/ bvowk/corewar.html

[7] WEKA software from Waikato University, http://www.cs.waikato.ac.nz/ ml/weka/

[8] D. Whitley, S. Rana, R. B. Heckendorn, Island model genetic algorithms and linearly separable problems, Selected Papers from AISB Workshop on Evolutionary Computing, pp. 109–125, LNCS 1305, Springer-Verlag, London, UK, 1997.

[9] I. H. Witten, E. Frank, Data Mining: Practical Machine Learning Tools and Techniques, Morgan Kaufmann, 2nd edition, 2005.

[10] S. Zap, Optimax, http://www.corewar.info/optimax/