An Analysis of Matching in Learning Classifier Systems

Martin V. Butz⋆, Pier Luca Lanzi†‡, Xavier Llorà‡, Daniele Loiacono†

⋆Department of Cognitive Psychology, University of Würzburg, 97070 Würzburg, Germany
†Artificial Intelligence and Robotics Laboratory, Politecnico di Milano, I-20133, Milano, Italy
‡Illinois Genetic Algorithm Laboratory, University of Illinois at Urbana Champaign, IL 61801, USA

butz@psychologie.uni-wuerzburg.de, pierluca.lanzi@polimi.it, xllora@uiuc.edu,
loiacono@elet.polimi.it

ABSTRACT

We investigate rule matching in learning classifier systems for problems involving binary and real inputs. We consider three rule encodings: the widely used character-based encoding, a specificity-based encoding, and a binary encoding used in ALECSYS. We compare the performance of the three algorithms both on matching alone and on typical test problems. The results on matching alone show that the population generality influences the performance of the matching algorithms based on string representations in different ways. Character-based encoding becomes slower and slower as generality increases, specificity-based encoding becomes faster and faster as generality increases. The results on typical test problems show that the specificity-based representation can halve the time required for matching but also that binary encoding is about ten times faster on the most difficult problems. Moreover, we extend specificity-based encoding to real-inputs and propose an algorithm that can halve the time require for matching real inputs using an interval-based representation.

Categories and Subject Descriptors

F.1.1 [**Models of Computation**]: Genetics Based Machine Learning, Learning Classifier Systems

General Terms

Algorithms, Performance.

Keywords

LCS, XCS, RL, Matching.

1. INTRODUCTION

Learning classifier systems [11, 10, 20] are methods of genetics-based machine learning that can solve both classification and reinforcement learning problems. They compute solutions consisting of condition-action-prediction rules, called classifiers; they apply (i) temporal difference or supervised learning to estimate rule quality in terms of problem solution and (ii) a genetic algorithm to discover better rules that may improve the current solution. In learning classifier systems, the target problem is viewed as an unknown environment that provides feedback in terms of numerical reward. Accordingly, a classifier system solves a problem by interacting with it and by trying to obtain as much reward as possible from it.

A learning classifier system maintains a rule base, called a population of classifiers, which represents the current solution. At each time step, the classifier system receives a problem instance which requires a decision (that is, an action to be performed next). It matches the incoming instance against the rule base and computes a match set containing the rules that apply to the current situation. Based on the value of the actions in the match set, the system decides which action to execute. As a consequence of the executed action, the system receives a numerical reward that is distributed to the rules accountable for it so as to improve the estimates of the action values. While the classifier system is interacting with the problem, a genetic algorithm is applied to the rule base to discover better rules, which may improve the current solution.

In learning classifier systems, matching is the most computationally demanding part of the overall process, which can occupy up to the 65%-85% of the overall time [18]. As a consequence, fast matching algorithms can lead to dramatic speedups. A recent study by Llorà and Sastry [18] compared the typical character-based encoding of rule conditions, an encoding based on the underlying binary arithmetic, and a version of the same binary encoding optimized via vector instructions. The results show that binary encodings combined with optimizations based on the underlying integer arithmetic can speedup the matching process up to 80 times. The analysis in [18] focuses on the cost of matching the entire population but does not consider the influence of rule generality, which, nonetheless, has a significant impact on the cost of matching. In the most typical implementations (see for instance, [2, 15]), the matching of a rule stops as soon as it is determined that the rule does not apply to the current problem instance. Therefore, matching a population of highly specific rules takes much less than match-

ing a population of highly general rules. In addition, the analysis in [18] considers matching alone: a population of random rules was generated, several match operations were performed using a random set of inputs, and the performance was measured as the time needed to match one rule set. However, in learning classifier systems the generality of the population can change dramatically during learning. Accordingly, the number of rules involved in the matching process and the cost of matching changes during evolution.

This paper is a follow-up of the work of Llorà and Sastry [18]. It extends their early analysis in various respects. The first and foremost extension regards the influence of rule generality on the cost of matching, which has not been considered in [18]. As we show in the next section, with the mostly used character-based encoding, the cost of matching depends on the rule generality and it is generally lower for specific conditions and larger for general conditions. Secondly, in this work we consider two additional encodings that were not considered in [18]: one was used in some implementations of ALECSYS which, similarly to the encoding discussed in [18], exploits the underlying integer arithmetic; one is a specificity-based encoding that has been distributed in some versions of XCS-C [2]. We compare the widely used character-based encoding and these two encodings on matching alone (as it was done in [18]) and on typical testbeds used in the learning classifier system literature (Boolean multiplexer, hidden parity, and woods problems [20, 4]). The first set of experiments, involving matching alone, shows the influence of population generality on the cost of matching: character-based encoding becomes slower and slower as generality increases, specificity-based encoding becomes faster and faster as generality increases, binary encoding is not influenced. The second set of experiments, involving known test problems, confirms the results of Llorà and Sastry [18]: binary encoding is the fastest encoding, being on a typical problem around ten times faster than character-based encoding; specificity-based encoding can solely halve the time require to solve a typical problem. At the end, we exploit the knowledge we acquired on binary problems and present a specificity-based encoding for the well-known interval-based representation. We compare the original matching algorithm for interval-based representation with our specificity-based encoding and show that the latter can speed up matching up to 50%, in populations that contain highly general rules.

2. CHARACTER-BASED ENCODING

Learning classifier systems often assume that the problem space is binary, that is, the problem instances (the inputs) are binary strings and the decisions (the actions) are binary strings. Accordingly, rule conditions are usually strings defined over the ternary alphabet $\{0,1,\#\}$. The match procedure scans all the input bits to check whether the condition contains a don't care symbol ($\#$) or an input bit is equal to the corresponding character in the condition. This ternary representation of rule conditions can be encoded in several ways [12, 18]. Among others, character-based encoding is probably the most straightforward and therefore the most widely spread approach to implement ternary conditions and matching. It defines conditions as strings (or arrays) of '0', '1', '#' symbols. Inputs are strings of 0s and 1s. Matching scans all the input characters and checks whether the condition contains a don't care symbol ($\#$) or an input character

---

Algorithm 1 Character-based encoding and matching.

```
// representation of classifier condition
string  condition;

// representation of classifier inputs
string  inputs;

// matching procedure
int pos = 0;
bool result = true;

while ( (result) && (pos<condition.size()) )
{
        result = ((condition[pos]=='#') ||
                (condition[pos]==inputs[pos]));
        pos++;
}
return result;
```

---

is equal to the corresponding character in the condition. Algorithm 1 reports, using a C-like syntax, a typical implementation of character-based encoding with the corresponding matching procedure [15, 7]. In the worst case, the cost of matching using character-based encoding is $O(n)$ to match one condition of length n and it is $O(Nn)$ when a population of N classifiers is considered. In the average case, however, the cost of matching is proportional to the percentage of $\#$ symbols in the classifier condition, or classifier generality. In fact, matching typically stops as soon as it is clear that the condition does not cover the current problem instance—when in Algorithm 1 the variable `result` becomes false. Thus, matching will perform more comparisons when applied to conditions that have many $\#$s and fewer comparisons on conditions with fewer $\#$s, since specific conditions apply to fewer situations.

3. OTHER ENCODINGS

Character-based encoding is very simple and for this reason widely used but it is also highly inefficient in that (1) it wastes 75% of the memory by using 8 bits characters to encode three symbols and (2) it processes input information that is in principle useless. We now examine two alternative encodings, which try to improve the character-based encoding by focusing the matching process only on the relevant input information and try to reduce memory and computational burden by exploiting the underlying integer arithmetic.

3.1 Specificity-based Encoding and Matching

Character-based encoding scans the entire condition, but checking the don't care symbols ($\#$s) is actually useless since they will match anyway. Accordingly, matching can be limited to the specific bits and a condition can be represented by a list of position-digit pairs $\langle p_i, d_i \rangle$, which specify which specific bits should be matched. A pair $\langle p_i, d_i \rangle$ indicates that the input position p_i must be equal to d_i. For instance, the condition #1##0# is represented by the set $\{\langle 2, 1 \rangle, \langle 5, 0 \rangle\}$, which indicates that the condition will apply when the second input is 1 and the fifth input is 0. By limiting the condition representation to the specific bits, this representation can reduce the memory occupation and the computational cost when the conditions involved are sufficiently general. If σ is the average specificity of the rules in the population [5]

Algorithm 2 Specificity-based encoding and matching.

```
// representation of classifier condition
vector<int>  condition_vector;

// representation of classifier inputs
string       inputs;

...

//
// matching procedure
//
int   pos = 0;
bool  result = true;

// match ends when a pos did not match or
// all the integers have been considered
while (result && pos<condition_vector.size())
{
      result = inputs[condition_vector[pos]/2] ==
               '0'+condition_vector[pos]%2;
      pos++;
}
return result;
```

(i.e., the percentage of specific positions in the population conditions), the cost matching is on average $O(\sigma n)$ for one rule and $O(\sigma N n)$ when a population of N classifiers is considered; if s_c and s_i are respectively the number of bytes required to store a character and an integer, the memory occupation for this representation is $O(\sigma n N(s_c + s_i))$, which makes the representation useful when $\sigma \leq s_c/(s_c + s_i)$. For instance, if s_c is 1 and s_i is 2, as in many C compilers, specificity based is convenient when the conditions have a specificity lower than the 33% (a rather typical situation in classifier systems). This specificity-based encoding is, for example, used in an implementation of XCS-C, where conditions are represented as lists of position-digit pairs [2]. Here, we propose an even slightly simpler implementation of the same idea. A condition c is represented as a set (an array) of integers $\{c_1, \ldots, c_i, \ldots\}$ that encodes both the position and digit information: the position p_i is obtained by the integer division $c_i/2$ while the digit d_i is obtained by the corresponding remainder (i.e., $c_i \mod 2$)[1]. The condition #1##0# is now represented by the set $\{5, 10\}$: 5 encodes $\langle 2, 1 \rangle$, 10 encodes $\langle 5, 0 \rangle$. This implementation is very compact and requires $O(\sigma n N s_i)$ bytes to store the entire population. It occupies less than the usual character-based representation when $\sigma \leq s_c/s_i$, that is, typically when less than 50% of the condition is specified. Algorithm 2 reports the implementation of this version of specificity encoding with the corresponding matching procedure. The condition is now a vector of integers (`vector<int>`), while the inputs are still represented as strings. Matching scans the integers in the vector and checks the corresponding condition positions against the inputs.

---

[1]The representation can be immediately generalized for an input space defined over an $\chi - ary$ alphabet: $p_i = c_i/\chi$ and $d_i = c_i \mod \chi$

Algorithm 3 Binary representation and matching.

```
// representation of classifier condition
bitset  fp;
bitset  sp;

// representation of classifier inputs
bitset  inputs;

// matching procedure

bitset result = ((input^fp) & (input^sp));
return result.none();
```

3.2 Encoding and Matching in Alecsys

The representations considered so far use built-in types (a character or an integer) to represent each condition symbol. As noted by Llorà and Sastry [18], this approach wastes most of the allocated memory (character-based encoding wastes up to the 75% of the memory [18]) and it also prevents the use of bitwise (register-level) logical operations, which can dramatically speed-up the matching process [18]. The memory and speed constraint were particularly strict in the early days of learning classifier system research. So, we went back to one of the first successful learning classifier systems, ALECSYS, developed by Dorigo, Colombetti and colleagues [9] and found out that rule conditions were implemented using arrays of bits packed up inside unsigned integers. In ALECSYS, a condition was represented by two arrays, `fp` and `sp`, of unsigned integers, each array represented part of the condition. A one in the rule condition is represented by a bit set to one in the same position of `fp` and `sp`; a zero is represented by a bit set to zero in the same position of `fp` and `sp`; a don't care (#) can be either represented by a 0 in `fp` and a 1 in `sp` or by a 1 in `fp` and a 0 in `sp`. Given the bit encoded inputs i, a condition matches if `fp^i & sp^i` returns a set of zero bits, where `^` is the bitwise exclusive or and `&` is the bitwise logical and. Algorithm 3 shows our implementation of the encoding used in ALECSYS and the corresponding matching. The condition is represented as two variables, `fp` and `sp`, using the Standard Template Library (STL) `bitset` class [13], which encodes a set of bits, the condition matches if the resulting bitset has all the bits set to zero, i.e., if `result.none()` returns true. While Llorà and Sastry [18] implemented conditions as arrays of integers to reach the maximum speedup possible, we preferred an implementation based on the language layer, which consequently depends on the capabilities of the language layer itself and on the optimization options provided by the compiler.

4. MATCHING ALONE

We performed two sets of experiments aimed at evaluating (i) the performance of the three encodings on matching alone and (ii) the performance of the three encodings on typical classifier system testbeds, namely Boolean multiplexer, hidden parity, and woods environments [20, 21, 4]. All the experiments were conducted with `xcslib` [15] (according to Llorà and Sastry [18], matching is more relevant in `xcslib` [15] than it is in XCS-C [2]) and machines with AMD Opteron CPUs and 1GB of memory, while all the programs have been compiled using `g++ 4.1.2` with the `-O3` option.

Algorithm 4 Procedure to test match performance.

```
// P is a set of N conditions
condition       P[N];

// generate the population
for(i=0; i<N; i++)
{
    // the i-th condition is randomly generate
    // with a dont care probability dcp
    P[i] = condition.random(dcp);
}

// tt is the total time
tt = 0;

// test 2000 input instances
for(ins=0; ins<2000; ins++)
{
    // generate a random input of size n
    input       in = random(n);

    // save the current time
    start = timer();

    // matching time evaluation for all classifiers
    for(i=0; i<N; i++)
    {
        // test if the ith classifier matches
        P[i].match(in);
    }

    // save the end time
    end = timer();

    // update tt
    tt = tt + (end-start);
}

// return the total time spent for matching
return tt
```

N	gen	char$(\mu \pm \sigma)$	spec$(\mu \pm \sigma)$	bin $(\mu \pm \sigma)$
1000	0.0	0.11 ± 0.02	0.12 ± 0.03	0.08 ± 0.02
1000	0.25	0.13 ± 0.03	0.11 ± 0.03	0.07 ± 0.02
1000	0.5	0.14 ± 0.03	0.12 ± 0.02	0.08 ± 0.03
1000	0.75	0.15 ± 0.03	0.12 ± 0.03	0.08 ± 0.02
1000	0.99	0.17 ± 0.03	0.09 ± 0.02	0.08 ± 0.02
5000	0.0	0.74 ± 0.07	1.14 ± 0.09	0.41 ± 0.05
5000	0.25	0.72 ± 0.06	1.13 ± 0.09	0.44 ± 0.06
5000	0.5	0.82 ± 0.07	1.11 ± 0.08	0.43 ± 0.08
5000	0.75	0.87 ± 0.08	0.94 ± 0.09	0.40 ± 0.05
5000	0.99	0.90 ± 0.08	0.55 ± 0.07	0.44 ± 0.07
10000	0.0	1.69 ± 0.12	2.80 ± 0.11	0.87 ± 0.09
10000	0.25	1.77 ± 0.12	2.83 ± 0.12	0.88 ± 0.09
10000	0.5	1.89 ± 0.1	2.73 ± 0.13	0.89 ± 0.07
10000	0.75	1.94 ± 0.09	2.13 ± 0.14	0.81 ± 0.08
10000	0.99	1.90 ± 0.1	1.53 ± 0.12	0.87 ± 0.08

Table 1: Time required to match 2000 instances when the problem size is 32 bits, the population size N is 1000, 5000, or 10000, and the population generality gen is 0.00, 0.25, 0.50, 0.75, or 0.99. Data are averages over 50 runs.

The first set of experiments was performed following the design of Llorà and Sastry [18]. We generated a set of N ternary conditions of length n with different generality and 2000 random inputs. We matched each random input against the N conditions using one of the three algorithms previously discussed and measured the average CPU time required to perform all the match operations. This procedure was repeated 50 times. The performance was measured as the average CPU time to perform the 2000 matches over the N conditions. Algorithm 4 shows the pseudocode of the overall procedure.

Table 1 reports the results when the condition length n is 32; N is the population size, gen is the population generality (or one minus the specificity σ, 1-σ), char is the average CPU time for the character-based encoding, spec is the average CPU time for the specificity-based encoding, and bin is the average CPU time for the binary encoding used in ALECSYS. The population generality gen was set by tuning the don't care probability parameter $P_{\#}$, see [20] for details. Conditions are short so the time differences are small. Binary encoding is always faster, at most, it is from 30% to 50% faster than character-based encoding depending on the population generality. Since binary encoding is not influenced by rule generality, the time is basically the same, independent of the value of gen. As we anticipated, character-based encoding is between 11% and 36% faster

when the population is more specific (when generality gen is low). In contrast, specificity-based encoding is slower in highly specific populations where there is a small probability that a rule will match (in fact, the experiments show that none of the problem instances matched) and therefore very few comparisons are performed. Accordingly, the overhead introduced by this representation slows down the matching. However, when conditions are general, character-based encoding will scan almost all the condition whereas specificity-based encoding will just compare the very few specific condition elements and therefore will perform faster than plain character-based encoding.

We repeated the same set of experiments with conditions consisting of 1024 symbols (Table 2). In general, the three encodings (alone) perform as in the previous experiment. Character-based encoding becomes slower as generality increases, specificity-based encoding becomes faster as generality increases, while binary encoding takes the same amount of time no matter what the generality is. With very specific conditions, both character-based and specificity-based encodings are faster than binary encoding. This can be explained by considering that with a generality between 0.0 and 0.5 almost no condition matches an input and few positions are checked. In contrast, binary encoding always checks all the 1024 positions (encoded in 128 bytes). With a high generality, character-based encoding is extremely slow but specificity-based encoding is faster, even faster than binary encoding. Again, this is easily explained by noting that, with high generality, the specificity-based encoding will require very few comparisons. In contrast, character-based and binary encoding will check the entire condition.

This set of experiments confirms that when considering the cost of matching in learning classifier systems, in some encodings the generality of the rule population matters and the matching time may be significantly influenced by rule generality. When considering these results we should also note that the binary encoding considered here is not highly optimized (as the one considered in [18]) and it exploits the C++ language layer to encapsulate the underlying integer representation. These two aspects surely have a significant influence on the matching time for the binary encoding,

N	P#	char($\mu \pm \sigma$)	spec($\mu \pm \sigma$)	bin ($\mu \pm \sigma$)
1000	0.0	0.13 ± 0.03	0.15 ± 0.03	0.29 ± 0.04
1000	0.25	0.14 ± 0.03	0.15 ± 0.04	0.27 ± 0.04
1000	0.5	0.15 ± 0.03	0.14 ± 0.03	0.27 ± 0.04
1000	0.75	0.16 ± 0.03	0.14 ± 0.03	0.27 ± 0.04
1000	0.99	0.93 ± 0.05	0.12 ± 0.04	0.30 ± 0.04
5000	0.0	1.50 ± 0.09	1.97 ± 0.12	2.77 ± 0.10
5000	0.25	1.56 ± 0.10	1.91 ± 0.11	2.77 ± 0.11
5000	0.5	1.61 ± 0.12	1.82 ± 0.11	2.76 ± 0.12
5000	0.75	1.77 ± 0.10	1.74 ± 0.11	2.72 ± 0.09
5000	0.99	6.29 ± 0.16	1.11 ± 0.07	2.83 ± 0.12
10000	0.0	3.15 ± 0.11	4.14 ± 0.14	5.60 ± 0.17
10000	0.25	3.33 ± 0.15	4.08 ± 0.17	5.45 ± 0.30
10000	0.5	3.56 ± 0.16	4.02 ± 0.13	5.64 ± 0.14
10000	0.75	3.97 ± 0.15	3.69 ± 0.14	5.69 ± 0.19
10000	0.99	12.73 ± 0.22	2.40 ± 0.14	5.54 ± 0.16

Table 2: Time required to match 2000 instances when the problem size is 1024 bits, the population size N is 1000, 5000, or 10000, and the population generality gen is 0.00, 0.25, 0.50, 0.75, or 0.99. Data are averages over 50 runs.

explaining the slower matching performance in comparison with character- and specificity-based encoding.

Despite this fact, the results may still seem to contradict what had been reported by Llorà and Sastry [18], who showed that the binary encoding is at least an order of magnitude faster than the character-based encoding. However, the character-based match used for comparison by Llorà and Sastry [18], available for download at [17], checks *all the condition symbols* whereas the match considered here for character-based encoding stops as soon as the match fails (see the `result` flag in Algorithm 1). Accordingly, the comparison performed by Llorà and Sastry [18] does not take into account classifier specificity/generality.

5. MATCHING WHILE LEARNING

In the previous set of experiments, we compared the three encodings on matching alone and noted that the time required for matching, for some encodings, depends on the generality of the rule population. With character-based encoding, matching is faster on specific populations and slower on more general ones; in contrast, with specificity-based encoding, matching is slower on specific populations and faster on general ones; while with binary encoding the cost of matching solely depends on the condition length. In learning classifier systems, the generality of the rules in the population changes during learning. Initially, the population contains rules of a predetermined generality [5]. When the exploration of the solution space begins, the population often contains many overly specific rules. As learning proceeds and effective problem representations are developed, the population converges to a compact, general set of rules. Therefore, the cost of matching changes during the evolution both because the population generality changes and also because the number of rules in the population changes. The character-based encoding will be favored by more specific populations and will be penalized by general populations. For specificity-based encodings, the pattern is reversed. Binary encodings always keep the same matching speed.

The second set of experiments aimed at evaluating the three encodings on well-known problems taken from the learning classifier system literature: Boolean multiplexer, hidden parity, and woods problems [20, 21, 4]. We selected these problems because they were available in one or more learning classifier system distributions and more precisely, we used the configuration files available with `xcslib` [15], the only parameter we changed is the number of runs performed for each problem—originally it was set to ten, here it is twenty. Since the encoding also influences how the genetic operators are implemented, to study the effect of the condition encoding on matching, we used rules with two encodings: the character-based encoding, as implemented in the original source code [15] (which was used for covering and for the genetic algorithm [20]), and the encoding used for matching only (that is, the character-based, the specificity-based, or the binary encoding). Accordingly, the genetic operators were not redefined for each representation so that the analysis does not take into account the time required by each representation for crossover and mutation.

The Boolean multiplexer is defined over binary strings of length n where $n = k + 2^k$; the first k bits, $x_0, \ldots, x_{k-1}$, represent an address which indexes the remaining 2^k bits, $y_0, \ldots, y_{2^k-1}$; the function returns the value of the indexed bit. For instance, in the 6-multiplexer function, mp_6, we have that $mp_6(100010) = 1$ while $mp_6(000111) = 0$.

Hidden parity was first used with XCS in [14] to relate the problem difficulty to the number of accurate maximally general classifiers needed by XCS to solve the problem. Hidden parity problems are defined over binary strings of length n in which only k bits are relevant; the hidden parity function ($\text{HP}_{n,k}$) returns the value of the parity function applied to the k relevant bits, that are *hidden* among the n inputs. For instance, given the hidden parity function $\text{HP}_{6,4}$ defined over inputs of six bits ($n = 6$), in which only the first four bits are relevant ($k = 4$), then we have that $\text{HP}_{6,4}(110111) = 0$ while $\text{HP}_{6,4}(001111) = 1$.

Woods problems are grids with obstacles, goal positions, and empty positions. The system can stay in any of the empty positions and it is able to move to any adjacent position that is empty. The system has to learn how to reach a goal position from any empty position. `Woods1` [20] and `Woods2` [21] are the first two woods problems that were used to test the performance of Wilson's XCS.

Table 3 reports the time required to perform 20 runs for each problem using the parameter settings included in the original distribution. In the problems considered, the number of inputs is rather limited, in that the condition size n ranges from 6 to 37, therefore the settings are similar to those discussed for Table 1, where binary encoding performed better. In fact, in all the problems binary encoding is the fastest, achieving a speedup of 91% for the most difficult problem, the 37-multiplexer, while specificity-based encoding yields speedups of about 50% on average. The result in Table 3 suggests that, metaphorically speaking, learning is like a marathon: it is better to keep the same (fast) matching pace, as binary encoding does, since the overhead time spent matching specific conditions earlier (as done by the specific-based encoding) piles up and significantly weighs on the overall matching speed.

	settings			char		spec		bin	
	n	N	$\#probs$	$t\ (\mu \pm \sigma)$	$t\ (\%)$	$t\ (\mu \pm \sigma)$	$t\ (\%)$	$t\ (\mu \pm \sigma)$	$t\ (\%)$
mp6	6	400	20000	0.54±0.03	100.0	0.35±0.02	64.8	0.21±0.01	38.9
mp11	11	1000	20000	2.85±0.20	100.0	1.54±0.10	54.0	0.41±0.02	14.4
mp20	20	2000	400000	66.82±2.24	100.0	35.43±0.88	53.0	9.24±0.24	13.8
mp37	37	5000	2000000	1893.87±376.37	100.0	804.85±123.39	42.5	169.34±24.76	8.9
par20-5	20	2000	600000	121.65±19.52	100.0	58.64±9.02	48.2	14.40±1.38	11.8
woods1	16	1600	4000	15.00±1.71	100.0	8.51±0.94	56.7	2.90±0.28	19.3
woods2	24	1600	20000	22.09±2.58	100.0	11.79±1.25	53.3	4.88±0.22	22.1

Table 3: Time required to solve each problem 20 times using the configuration available in the xcslib distribution. Data are averages over 20 runs.

6. BEYOND BINARY DOMAINS

Over the recent years, an increasing number of studies have applied learning classifier systems to real-valued problems. For this purpose, several other condition representations have been used, including center-based intervals [22], simple intervals [23, 19], convex hulls [16], ellipsoids [3], and hyper-ellipsoids [6]. All these representations came with their own matching algorithm and, to our knowledge, there has not been any work that tried to speedup matching for real inputs. Character-based and binary encodings do not provide any hint about how to speed up the matching for real inputs, but specificity-based encoding actually suggests a way to speed up matching in real domains by dealing with more specific information first. In binary domains, the specificity-based encoding considers specific bits first and ignores the generalized positions. Although the distinction between specified and generalized position is blurred in real domains, we can still exploit the same principle and extend the widely used interval-based representation by a specificity-based encoding.

6.1 Interval Based Representation & Specificity-Based Matching

In the interval-based case [23], a condition is represented by a concatenation of n real interval predicates, $int_i = (l_i, u_i)$; given an input $\vec{x}$ consisting of n real numbers, a condition matches $\vec{x}$ if, for every $i \in \{1, \dots n\}$, the predicate $l_i \leq x_i \wedge x_i \leq u_i$ is verified. The matching is straightforward and its pseudocode is reported as Algorithm 5: the condition (identified by the variable `condition`) is represented as a vector of intervals; the inputs are a vector of real values (in double precision); the n inputs (i.e., `inputs.size()`) are scanned and each input is tested against the corresponding interval; the process stops either when all the inputs matched or as soon as one of the intervals does not match (when `result` in Algorithm 5 becomes false).

This matching procedure can be sped-up by changing the order in which the inputs are tested: if more specific intervals are tested first, the match is more likely to fail early so as to speed up the matching process. As an example, consider a condition represented by a set of intervals of decreasing generality (i.e., $(u_i - l_i) > (u_{i+1} - l_{i+1})$ for all $i \in \{1, \dots, n-1\}$); for each i, the interval i is more likely to match than the interval $i+1$. Therefore, the typical matching procedure, which scans the intervals one after another, is more likely to fail as it moves from testing interval i to interval $i+1$. However, if we reverse the order and scan the condition from interval n to interval 1, the match is likely to

---

Algorithm 5 Matching for interval-based conditions.

```
// representation of classifier condition
vector<interval>  condition;

// representation of classifier inputs
vector<double>    inputs;

// matching procedure
int pos = 0;
bool result = true;

while ( (result) && (pos<inputs.size()) )
{
    result =
        ((inputs[pos]>=condition[pos].lower) &&
        (condition[pos].upper>=inputs[pos]));
    pos++;
}
return result;
```

---

fail earlier since interval $i + 1$ is more specific than interval i. Note however that, how much earlier matching will fail cannot be estimated, since this depends on the distribution of the interval specificities and on the input distribution.

We generalize this principle and improve the interval-based encoding by adding a *match index* vector `mi` that represents the sorting of the intervals from the more specific interval to the most general one, i.e., `mi[0]` is the index of the most specific interval, `mi[1]` is the index of the second most specific interval, etc. The match vector `mi` is computed only once when the condition is created and then used for matching purposes. This specificity-based encoding for the interval-based representation and the corresponding matching procedure is shown as Algorithm 6: the match vector `mi` is a vector of integers; matching works as before but the interval to be tested is determined by the match index vector `mi`.

6.2 Matching Alone

We compared the standard matching algorithm for interval-based conditions in [23] with the introduced specificity-based encoding (cf. Algorithm 6), using the same experimental design already used for the binary problems in Section 4. We generated random populations of 1000 and 10000 rules of different generalities and 20000 instances; we measured the time required to match all the 20000 instances against the 10000 rules in the population. The generality of a random population was determined by setting an adequate value of the parameter r_0 (see [23] for details). Ta-

Algorithm 6 Specificity-based matching for interval-based conditions.

```
// representation of classifier condition
vector<interval>  condition;

// matching index
vector<int>       mi;

// representation of classifier inputs
vector<double>    inputs;

// matching procedure
int pos = 0;
bool result = true;

while ( (result) && (pos<condition.size()) )
{
  result =
   ((inputs[mi[pos]]>=condition[mi[pos]].lower) &&
    (condition[mi[pos]].upper>=inputs[mi[pos]]));
  pos++;
}
return result;
```

ble 4 reports the data from this set of experiments: n is the number of inputs, **gen** is the population generality, N is the number of conditions involved, **plain** is the CPU time required by the standard matching algorithm [23], **spec** is the CPU time required by the specificity-based matching algorithm [23]. The results reported in Table 4 are similar to those reported for the binary case. The cost of the usual matching, which scans the whole condition in a predetermined order, increases with generality whereas the cost of specificity-based matching decreases as the generality increases and as the number of inputs n increases. Accordingly, in populations with less general rules the usual matching is faster whereas the specificity-based matching is faster (up to 60%), when applied to populations containing rules of high generality.

7. CONCLUSIONS

We analyzed matching in learning classifier systems in binary and real domains. Our aim was to extend the previous analysis of Llorà and Sastry [18] in two respects. We wanted to analyze the role of population generality on the cost of matching and we also wanted to evaluate the performance of the different algorithms on typical testbeds taken from the literature. We considered three encodings and the corresponding matching algorithms: the widely used character-based encoding, a specificity-based encoding used in some XCS-C implementations [2], and the binary encoding used in some implementations of ALECSYS [9]. Our results on matching alone show the influence of population generality on character-based and specificity-based encodings: as generality increases (that is, as specificity decreases), the character-based encoding becomes slower and slower, whereas specificity-based encoding becomes faster and faster. Overall, specificity-based matching is 50% faster than character-based encoding when general populations are involved, but it can be slower than character-based encoding if more specific populations are considered. Binary encoding confirms to be the fastest options, as previously shown in [18]. Our results on typical testbeds show that binary encoding provides the highest speedup of up to 90% compared

n	gen	N	plain	spec
10	0.25	1000	0.07± 0.02	0.07± 0.02
10	0.50	1000	0.08± 0.01	0.08± 0.02
10	0.75	1000	0.10± 0.01	0.08± 0.03
10	0.90	1000	0.13± 0.02	0.11± 0.02
10	0.95	1000	0.13± 0.03	0.13± 0.02
50	0.25	1000	0.37± 0.03	0.41± 0.06
50	0.50	1000	0.42± 0.06	0.41± 0.06
50	0.75	1000	0.54± 0.04	0.42± 0.04
50	0.90	1000	0.75± 0.05	0.44± 0.06
50	0.95	1000	1.07± 0.06	0.59± 0.04
100	0.25	1000	0.84± 0.07	0.82± 0.06
100	0.50	1000	0.88± 0.08	0.83± 0.05
100	0.75	1000	1.04± 0.07	0.80± 0.06
100	0.90	1000	1.45± 0.08	0.84± 0.08
100	0.95	1000	2.21± 0.10	0.95± 0.08
10	0.25	10000	0.72± 0.08	0.81± 0.04
10	0.50	10000	0.80± 0.07	0.88± 0.04
10	0.75	10000	1.05± 0.05	1.07± 0.06
10	0.90	10000	1.23± 0.05	1.25± 0.06
10	0.95	10000	1.32± 0.11	1.88± 0.12
50	0.25	10000	4.08± 0.10	4.45± 0.20
50	0.50	10000	5.69± 0.16	4.86± 0.15
50	0.75	10000	9.38± 0.42	6.03± 0.67
50	0.90	10000	11.92± 0.17	7.39± 2.21
50	0.95	10000	15.86± 0.81	13.30± 0.25
100	0.25	10000	8.46± 0.34	8.39± 0.22
100	0.50	10000	13.02± 1.96	8.76± 0.42
100	0.75	10000	16.93± 2.17	8.83± 0.29
100	0.90	10000	27.49± 2.04	9.23± 0.28
100	0.95	10000	36.48± 0.34	13.56± 0.38

Table 4: Time required to match 20000 instances when the problem consists of 10, 50 or 100 real inputs, the population size N is 1000 or 10000, and the population generality gen varies between 0.00 and 0.99. Data are averages over 20 runs.

to the usual character-based encoding, while specificity-based encoding is twice as fast as the character-based encoding. Specificity-based encoding provides limited speedups when compared to the binary encoding, but it can be easily extended to real domains. We compare the usual interval-based representation for real-valued inputs with our simple specificity-based representation for real domains on matching alone. The results presented show that, by focusing matching on the more specific part of a condition first, the time required for matching can be almost halved. We also performed some initial experiments, not reported here, with typical testbeds introduced in the learning classifier system literature such as the learning of sum predicates [23] and the checker board [1]. The initial results show basically no significant difference between the time required using standard and specificity-based matching, but more experiments are still needed and they are a topic for future research.

8. REFERENCES

[1] E. Bernadó-Mansilla and T. K. Ho. Domain of competence of XCS classifier system in complexity measurement space. *IEEE Trans. Evolutionary Computation*, 9(1):82–104, 2005.

[2] M. V. Butz. XCS (+ tournament selection) classifier system implementation in c, version 1.2. Technical Report 2003023, Illinois Genetic Algorithms Laboratory – University of Illinois at Urbana-Champaign, 2003.

[3] M. V. Butz. Kernel-based, ellipsoidal conditions in the real-valued XCS classifier system. In H.-G. Beyer and U.-M. O'Reilly, editors, *GECCO*, pages 1835–1842. ACM, 2005.

[4] M. V. Butz. *Rule-Based Evolutionary Online Learning Systems: A Principled Approach to LCS Analysis and Design*. Studies in Fuzziness and Soft Computing. Springer Verlag, Berlin-Heidelberg, Germany, 2006.

[5] M. V. Butz, T. Kovacs, P. L. Lanzi, and S. W. Wilson. Toward a theory of generalization and learning in XCS. *IEEE Transaction on Evolutionary Computation*, 8(1):28–46, Feb. 2004.

[6] M. V. Butz, P. L. Lanzi, and S. W. Wilson. Hyper-ellipsoidal conditions in XCS: rotation, linear approximation, and solution structure. In Cattolico [8], pages 1457–1464.

[7] M. V. Butz and S. W. Wilson. An algorithmic description of XCS. *Journal of Soft Computing*, 6(3–4):144–153, 2002.

[8] M. Cattolico, editor. *Genetic and Evolutionary Computation Conference, GECCO 2006, Proceedings, Seattle, Washington, USA, July 8-12, 2006*. ACM, 2006.

[9] M. Dorigo and M. Colombetti. *Robot Shaping: An Experiment in Behavior Engineering*. MIT Press/Bradford Books, 1998.

[10] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, Mass., 1989.

[11] J. H. Holland and J. S. Reitman. Cognitive systems based on adaptive algorithms. 1978. Reprinted in: Evolutionary Computation. The Fossil Record. David B. Fogel (Ed.) IEEE Press, 1998. ISBN: 0-7803-3481-7.

[12] K. A. D. Jong and W. M. Spears. Learning Concept Classification Rules using Genetic Algorithms. In *Proceedings of the Twelfth International Conference on Artificial IntelligenceIJCAI-91*, volume 2, 1991.

[13] N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley Professional, 1999.

[14] T. Kovacs and M. Kerber. What makes a problem hard for XCS? In P. L. Lanzi, W. Stolzmann, and S. W. Wilson, editors, *Advances in Learning Classifier Systems*, volume 1996 of *LNAI*, pages 80–99. Springer–Verlag, Berlin, 2001.

[15] P. L. Lanzi. The XCS library. 2002.

[16] P. L. Lanzi and S. W. Wilson. Using convex hulls to represent classifier conditions. In Cattolico [8], pages 1481–1488.

[17] X. Llorá and K. Sastry. Software for fast rule matching using vector instructions. `http://www.illigal.uiuc.edu/web/xllora/2006/01/19/`, Last checked on March 21th 2008.

[18] X. Llorà and K. Sastry. Fast rule matching for learning classifier systems via vector instructions. In Cattolico [8], pages 1513–1520.

[19] C. Stone and L. Bull. For real! XCS with continuous-valued inputs. *Evolutionary Computation*, 11(3):298–336, 2003.

[20] S. W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

[21] S. W. Wilson. Generalization in the XCS classifier system. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 665–674. Morgan Kaufmann, 1998.

[22] S. W. Wilson. Get real! XCS with continuous-valued inputs. In P. L. Lanzi, W. Stolzmann, and S. W. Wilson, editors, *Learning Classifier Systems*, volume 1813 of *Lecture Notes in Computer Science*, pages 209–222. Springer, 1999.

[23] S. W. Wilson. Mining oblique data with XCS. In P. L. Lanzi, W. Stolzmann, and S. W. Wilson, editors, *IWLCS*, volume 1996 of *Lecture Notes in Computer Science*, pages 158–176. Springer, 2000.