The Subsumption Mechanism for XCS using Code Fragmented Conditions

Hsuan-Ta Lin
Institute of Biomedical Engineering, National Chiao-Tung University, Taiwan (R.O.C.).
+886 (3)571 2121#59329
ada19900409@gmail.com

Po-Ming Lee
Institute of Computer Science and Engineering, National Chiao-Tung University, Taiwan (R.O.C)
1001 Ta Hsueh Rd., Hsinchu, Taiwan (R.O.C.)
pmli@cs.nctu.edu.tw

Tzu-Chien Hsiao
Institute of Computer Science and Engineering and with Institute of Biomedical Engineering, National Chiao-Tung University, Taiwan (R.O.C.).
labview@cs.nctu.edu.tw

ABSTRACT
By utilizing a code-fragmented representation of Extended Classifier System (XCS) condition in conjunction with building-block extraction technique, autonomous scaling has been realized in the latest work of XCS. The technique substantially reduces the number of training instances required in various benchmark problems. However, the subsumption mechanism was not included in the former report of the technique. Therefore, we invented the subsumption mechanism for XCS with such technique, and observed the characteristics of such the system in multiplexer problems. The finding indicates that our subsumption mechanism decreased the number of macro-classifiers.

Categories and Subject Descriptors
F.1.1 [**Models of Computation**]: Genetics-Based Machine Learning, Learning Classifier Systems

General Terms
Algorithms, Performance

Keywords
Extended Classifier System (XCS), Building Blocks, Code Fragments, Scalability, Pattern Recognition

1. INTRODUCTION
Machine learning techniques are widely used in many fields, such as data mining and artificial intelligence because of their capability of extracting previously unknown knowledge from datasets. One of the research mainstreams is Learning Classifier Systems (LCSs), especially Extended Classifier System (XCS). XCS is a Learning Classifier System (LCS) that learns the problem using a set (called population set [P]) of condition-action rules called classifiers. Via the learning process, XCS continually updates [P] to extract useful unknown knowledge from the dataset [1].

Because of XCS's excellent performance in a wide range of real world applications, XCS has been successfully applied to various areas, including security [2, 3], finance [4-6], medical research [7, 8], and chip design [9]. In the area of finance, XCS is known for its capability of financial time series forecasting. As a promising classifier, XCS is also used in applications of personalization and user context extractions [10, 11]. These studies provide a comparison of accuracy rates between XCS and other commonly used traditional machine learning techniques to demonstrate that XCS is competitive [12].

CECCO'13 Companion, July 6–10, 2013, Amsterdam, The Netherlands.
Copyright 2013 ACM 978-1-4503-1964-5/13/07...$15.00.

Traditionally, LCSs use binary representations. Later on, different types of encoding representation are proposed to solve problems in different domains. For example, XCSR (see [13]), by adopting real-value representation, can be applied to continuous variables such as stock index, temperature, height and weight. Furthermore, on the path of the attempts of using code-fragment style representations, Lanzi firstly experimented with two different ways to represent classifier conditions: a variable length messy coding [14], and S-expressions [15]. The messy coding scheme translates environmental inputs into bit strings that have no positional linking between bits in classifier condition and any feature of the environmental input [14]. On the other hand, compare to messy coding, S-expressions is a more complex representation for general classifier conditions [15]. The representation methods are complementary and can be used together, for coping more dynamic types of input variables.

Later, in 2003, Lanzi presented a stack-based genetic programming mechanism for XCS [16]. Such the technique represents the condition of XCS by linear sequences of tokens expressed in Reverse Polish Notation. Each token can be a function, a variable, or a constant. The set of used functions includes boolean operators (AND, OR, NOT, EOR), arithmetic operators (+ and -), and comparisons (> and =). The variables were represented by the values of sensory inputs. In 2012, Iqbal et. al, by utilizing a code-fragmented representation of XCS conditions in conjunction with a building-block extraction technique, autonomous scaling was demonstrate effective in LCSs for the first time. The effectiveness and the reduction in the number of training instances required in large problems of the proposed technique was reported [17]. Such the genetic programming (GP) like approach proposed by Iqbal et. al, would be desirable for researchers in LCS field, because such approach may allow XCS to cope with high dimension problems.

GP is an evolutionary algorithm-based methodology that generates the computer programs. These computer programs are represented by a tree structures that perform a user-defined task.

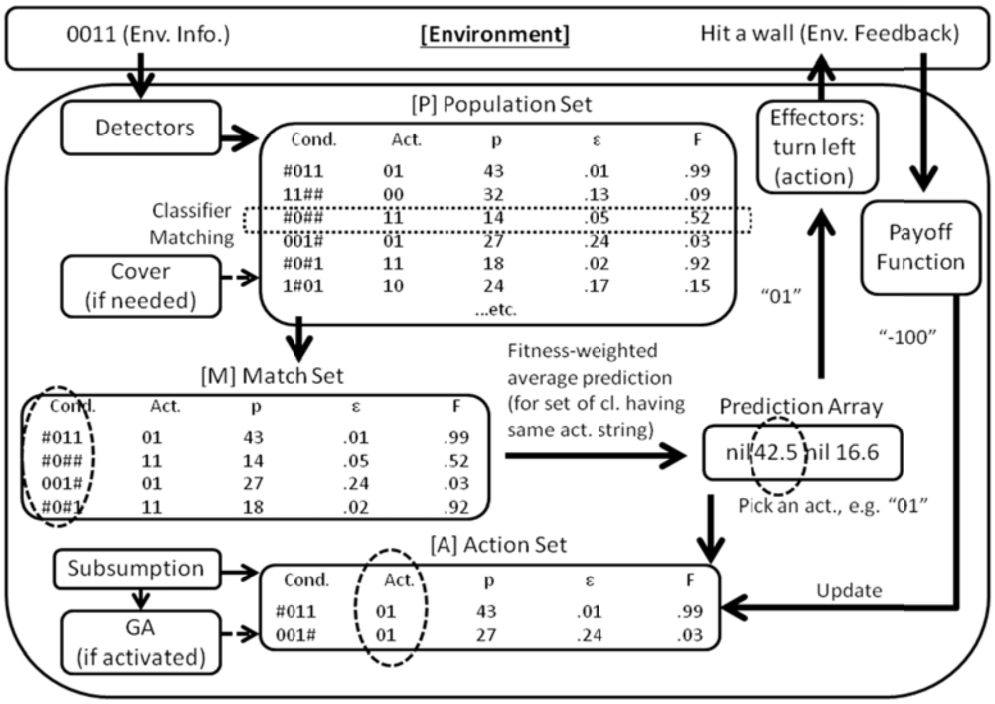

Figure 1 The system architecture of XCS.

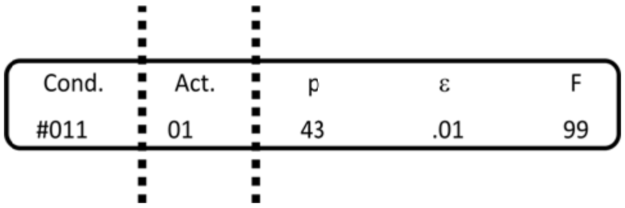

Figure 2 An example of a typical classifier managed by XCS.

The internal nodes of the tree are functions and the leaves are terminal symbols. GP is used to seek the program that delivers the correct output for each known input. For instance, if the set {x,1} is the terminal symbols and {+,-,*,/} is the function set, then GP will generate the program that map the set {(1,2), (2,5), (3,10), (4,17),…} as shown in Figure 3. The expression of the tree is x^2+1 [18].

However, the subsumption mechanism for the modified version of XCS has not been proposed. Since the subsumption technique in XCS results in a small final [P] by reducing the number of specific classifiers, we invented a subsumption mechanism for XCS with code fragments, and observed the characteristics of the system in multiplexer problems.

The remainder of this paper is structured as follows: Section 2 will describe the brief description of extracting building blocks of knowledge and the subsumption mechanism in XCS. The result and discussion of this research are presented in Section 3 and 4. Finally, we will describe an explicit and abbreviated conclusion in Section 5.

2. METHOD AND MATERIALS

2.1 XCS Classifier System

2.1.1 XCS Base System

XCS is a rule-based online learning algorithm which is able to extract knowledge from an unknown dataset in an iterative manner. XCS can also be regarded as a system that manages a set of classifiers that are represented in the traditional production system form of "IF state THEN action". By integrating the Genetic Algorithm (GA) component (also named rule discovery component in XCS), the set of classifiers can evolve occasionally, and search for a set of classifiers that yields the maximal generality and accuracy [1].

A detailed description on the flow of a typical XCS learning iteration is presented as follows: first, as shown in Figure 1, the XCS detector encodes the status of the environment at the beginning of a typical iteration into a binary string, and uses it for the classifier matching process. Second, during classifier matching process, XCS searches for classifiers in [P], in which the condition space represented by its condition string (0, 1, # for each bit, # indicates a bit that should be ignored, also called "don't care" bit) includes the detected current status, and places all matched classifiers into Match Set (denoted by [M]). If the classifiers found during classifier matching process do not meet the predefined criteria, XCS applies cover to generate new classifiers to enable their condition string to match current input (i.e., the environmental status), and action string is chosen at random.

Third, XCS calculates the fitness weighted average prediction from sets of classifiers that suggest the same output (with the same action string) after [M] is generated.

After calculating P_i for each possible output, Prediction Array is formed for output selection process. The output selection regime is usually set to pick up output i, which owns the maximal predicted payoff, max (P_i), in the prediction array, and occasionally picks up an output for exploration purpose arbitrarily.

The Effectors perform its action to the environment according to the selected output. Fourth is the Q-learning style reinforcement learning part of XCS (on the right and the bottom of Fig. 1). After Effectors perform action to the environment, feedback generated by the environment is gathered by XCS. Payoff Function is a function predefined by the user to interpret the feedback into numeric form of payoff, and the computed payoff is used to update parameters (p, ε, and F) of each classifier. The Update process is held on only the current Action Set (denoted by [A]), which represent the set of classifiers (demonstrated in the bottom of Fig. 1) that are responsible for the environmental feedback caused by the classifiers' suggested action performed by the Effectors. Last is the Rule (Classifier) Discovery part of XCS (on the left bottom of Fig. 1). During the learning process, GA is triggered occasionally to search for accurate classifiers in the classifier (condition-action string representation) space. During Update process and GA, Subsumption is performed to enable "Macro-Classifier" (classifiers that are more general than others) to subsume other classifiers to reduce the number of redundant, overlapped classifiers.

As shown in Figure 2, a classifier managed by the original XCS can be divided into three parts, as follows: a ternary string representing condition (for certain situations. 0, 1, # for each bit, # indicates a bit that should be ignored, also called "don't care" bit), a binary string representing an action (the output suggested by the classifier), and three parameters of the classifier, in which p represents predicted payoff, ε represents prediction error, and F represents fitness value. Several versions of XCS were developed over the past decade to suit various types of problems [19]. The original XCS was designed to perform on datasets with discrete inputs and a discrete output.

The detailed description of macroclassifiers and subsumption deletion will be described in Section 2.1.2 and 2.1.3.

2.1.2 Macroclassifiers

To allow XCS extract generalized rules (classifiers) from a previously unknown dataset, the use of the macroclassifiers technique to reduce redundant classifiers is crucial in the design of XCS. By reducing redundant classifiers, the use of the macroclassifiers technique realizes the generalization of classifiers in XCS, and also increases the speed of Classifier Matching process. The idea of macroclassifier is to obtain an additional parameter, numerosity parameter, num, for the classifiers managed in XCS [20], in which a classifier with numerosity num = n is equivalent to n regular classifiers.

When XCS generates a new classifier at the initialization step or at later stages, [P] is scanned to examine whether a macroclassifier exists with the same condition and action as that of the new classifier. If [P] has classifiers with the same condition and action, the numerosity of the existing macroclassifier with the same condition and action is incremented by one instead of inserting the new classifier into [P]. Otherwise, the new classifier is added to the population with its own numerosity field set to one. Similarly, when the macroclassifier experiences a deletion, its numerosity is decremented by one instead of being deleted, and then any macroclassifier with numerosity num = 0 is removed from the population.

2.1.3 Subsumption Deletion

Subsumption deletion is a method to improve the generalization capability of XCS, and occurs after the Update process of [A] and GA (also called Action Set Subsumption and GA Subsumption respectively) [20]. During Action Set Subsumption deletion, XCS selects an experienced classifier G with $\varepsilon < \varepsilon_0$ first; subsequently, G subsumes all classifiers in [A] that are less general than G by deleting them, and the numerosity of G is incremented accordingly. The XCS also operates GA Subsumption deletion when new classifiers (children) are generated through GA; the children are compared to their parent classifiers and subsumed as well if the parent classifiers are experienced and more general. Simultaneously, the numerosity of the subsuming classifier is incremented. Otherwise, XCS inserts the generated new classifiers into [P]. For further details of XCS please see Butz's algorithmic description of XCS [20].

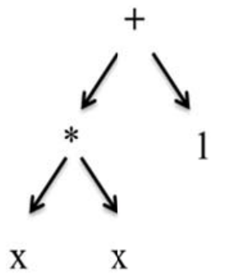

Figure 3 An example of a GP tree structure.

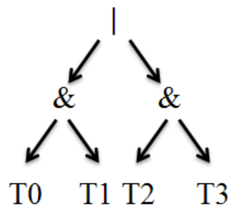

Figure 4 An example of a code fragment.

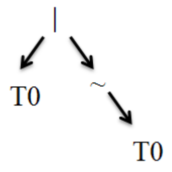

Figure 5 An example of a "don't care" code fragment.

2.2 XCS Using Code Fragmented Conditions

For the XCS using code fragmented conditions, the binary representation of each condition bit of a classifier is changed to a code fragment. A code fragment is a binary tree. Each tree node has at most two child nodes, which can be distinguished as "left" and "right". Each binary tree has depth up to two and so can have maximally seven nodes. The leaves of each code fragments are operands, and the other nodes contain operators. The operator include &, |, d, r, and ~ that denotes the boolean operators of AND, OR, NAND, NOR, and NOT respectively. The code fragment uses postfix notation. Figure 4 is an example of a code fragment which postfix notation is "T0T1&T2T3&|". The terminal symbols may be the combination of the symbol set {*T*0, *T*1, …, *Tn*-1} where n is the length of the classifier rule.

To determine whether a problem input is matched by a classifier, each of the code fragments in a classifier will determine whether the problem input matches it. If each code fragment in the condition of classifier has output 1, the problem input is matched by the classifier. For example, consider the problem instance is "100010" then the code fragment shown in Figure 4 will generate 0 as the output value (not matched). Because T1, T2, T3, T4 are replaced by 1, 0, 0, and 0 respectively.

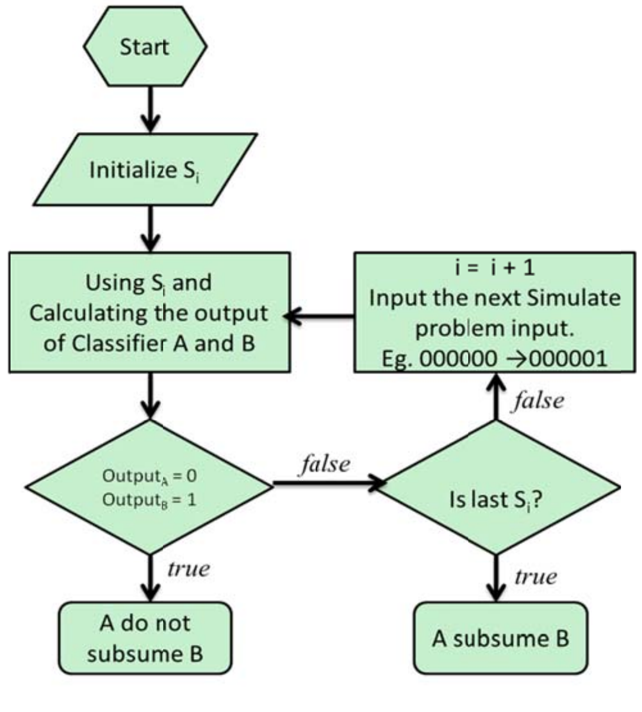

Figure 6 The flow diagram of the mechanism of subsumption.

XCS randomly chooses the code fragments from a code fragment population to generate classifier conditions during the covering mechanism. The Genetic Algorithm (GA) and the crossover point can occur between any two code fragments. The mutation operator will check whether the code fragment is "don't care" (same as "#" used in the ternary representation stated in Section 2.1.1). If not, change the code fragment to "don't care". As shown in Figure 5, the output of the "don't care" code fragment is always 1.

The building blocks of knowledge from a smaller problem seed the population of the code fragments used in the problems that are more complex. The fitter code fragments are used to replace the terminal symbol of the next level problem. The probability of choosing a code fragment from the population of fitter code fragments is set to 0.5.

Each code fragment has a fitness value which is updated by Equation 1, where *specificness* is the number of the code fragments *cf* in the classifier *clfr*. Initially the fitness value of the code fragment is equal to 0. By Equation 1, when XCS updates the classifier fitness, then the fitness of the code fragments in the classifier condition is also updated too.

The system will observe the fitness of each code fragment, and hence identify the fitter and useful building blocks of knowledge.

$$cf.fitness = cf.fitness + (clfr.specificness)^{-v} \qquad (1)$$

2.3 The Mechanism of Subsumption

In this section, we describe the mechanism of subsumption we invented. The main principle of our method is a way to determine a classifier is more general than another classifier. Figure 6 shows the flow chart of our proposed mechanism of subsumption.

The following steps show how we determine classifier A is more general than classifier B. If classifier A is more general than classifier B, for all environmental inputs, classifier A and classifier B must followed the rule that when classifier B matches an input (all code fragments in this classifier rule output 1), A must matches the input as well. On the other hand, when classifier B does not match an input, we don't care whether classifier A matches the input or not.

Based on such the principle, to determine if classifier A is more general than classifier B, a truth table of all inputs can be generated, called the table of simulations. The simulated input is called simulate question input (S_i). Initially, S_i set to 0 in binary representation. For example, if the classifier's condition length is 6, S_i will set to "000000". Then S_i replaces each terminal symbols, such as T0, T1, T2, …, T5 respectively of each code fragment in the classifier rule. Each code fragment will generate the value of CFi (i = 0~5) as shown in Table 1. Finally, each code fragmented value (CFi) will use the boolean operator AND to generate the value of output. For example, $output_A = CF_A0 \wedge CF_A1 \wedge CF_A2 \wedge CF_A3 \wedge CF_A4 \wedge CF_A5$ and $output_B = CF_B0 \wedge CF_B1 \wedge CF_B2 \wedge CF_B3 \wedge CF_B4 \wedge CF_B5$. If classifier A does not subsume B, system will return false. Otherwise, S_i will do the action $i = i + 1$ and get the next S_i. Having the next iteration and generating the next output until $S_i = 1$ in binary representation.

Table 1 The simluate question input.

Simulated Question Input (S_i)	Classifier A							Classifier B						
	CF_1	CF_2	CF_3	CF_4	CF_5	CF_6	$Output_A$	CF_1	CF_2	CF_3	CF_4	CF_5	CF_6	$Output_B$
000000	1	1	1	1	1	1	1	1	1	0	0	1	1	0
000001	1	0	0	0	0	1	0	1	1	1	1	1	1	1
000010	1	1	1	0	1	1	0	0	0	0	1	1	0	0
…														
…														
111111	1	1	0	0	0	1	0	1	1	1	1	1	1	1

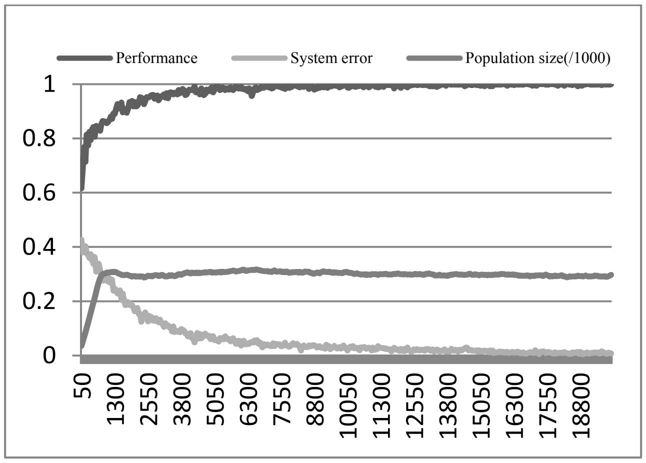

Figure 7 6-bit multiplexer with the “GA subsumption” off.

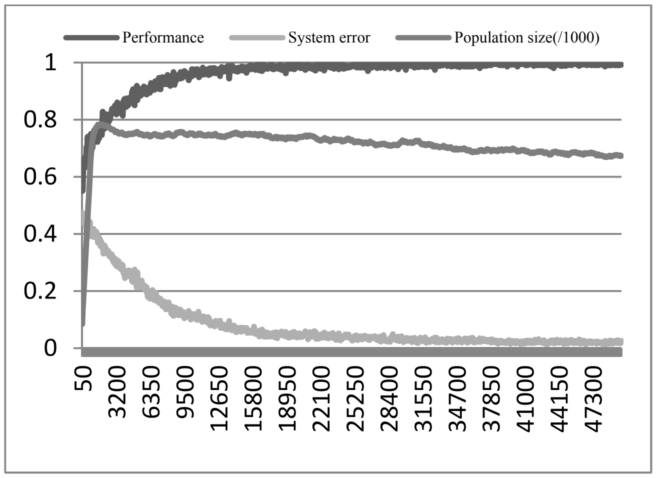

Figure 9 11-bit multiplexer with the “GA subsumption” off.

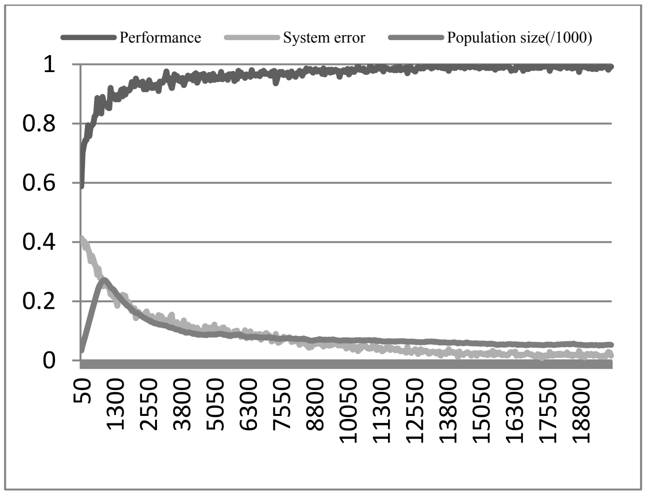

Figure 8 6-bit multiplexer with the “GA subsumption” on.

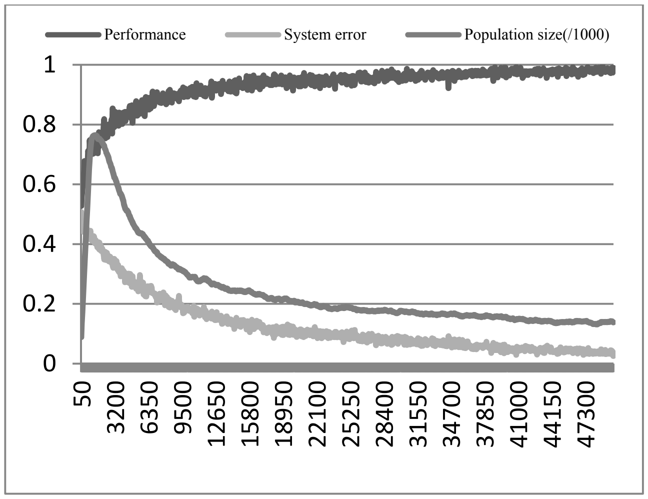

Figure 10 11-bit multiplexer with the “GA subsumption” on.

2.4 Experiment Design

2.4.1 L-bit Multiplexer

The experiment uses *L*-bit multiplexer to be the testing problem. L-bit multiplexer is a simple *L*-bit Boolean function that inputs Boolean strings with length *L* and outputs with a single bit binary output of 0 or 1. The function's output values are determined by regarding the first k bits as an address that indexes the remaining 2^k bits, and returns the indexed bit. For example, in the 6-bit multiplexer problem ($k = 2$), the output corresponds to the input string, 100010 is 1, because the "address," 10, indexes bit 2 of the remaining four bits.

2.4.2 Parameter Setup

XCS parameters use as follows: fall of rate of the fitness $\alpha = 0.1$, learning rate $\beta = 0.2$, the threshold of prediction error $\varepsilon_0 = 10$, fitness exponent $v = 5$, the threshold for GA application in the action set $\theta_{GA} = 25$, the threshold for subsumption $\theta_{sub} = 20$, the fraction of mean fitness $\delta = 0.1$, the probability of mutation $\mu = 0.04$, the probability of crossover $x = 0.8$, the probability of using don't care symbol when covering $P\# = 0.5$, $N = 500$, 1000, and 2500 for 6-, 11-, and 20-bit multiplexer respectively. The number of the code fragments used is twice the number of classifiers. The number of the fitter code fragments from 6- to 11-bis multiplexer is 22, and from 11- to 20-bit multiplexer is 40. Only "GA subsumption" which occurs in the mechanism of GA is turned on in our experiment.

3. RESULT

The result of the experiment is discussed in this section. We divided the experiment into three parts, as follows: 6-bit, 11-bit, and 20-bit multiplexer problems. Figure 7 shows the average of 30 runs, the population size of 6-bit multiplexer with the GA subsumption off is approximately 300. Performance reaches its maximum approximately 99% at about 6000 problems. System error reaches a minimum at a similar point. Figure 8 shows the multiplexer with the GA subsumption on, we observe that the population size reduces to about 60. Performance reaches its maximum approximately 99% at about 12000 problems. System error reaches a minimum at a similar point.

Figure 9 shows the average of 30 runs, the population size of 11-bit multiplexer with the GA subsumption off is approximately 750. Performance reaches its maximum approximately 99% at about 15000 problems. System error reaches a minimum at a similar point. Figure 10 shows the multiplexer with GA subsumption on, we observe that the population size reduces to about 160. Performance reaches its maximum approximately 99% at about 36000 problems. System error reaches a minimum at a similar point.

4. DISCUSSION

4.1 The Performance of Experiments

In our experiments (6-bit, 11-bit multiplexer), the performance do not reach 100% occasionally, so the average of the accuracy reached in 6000 iteration (6-bit multiplexer) was provided in the result section as 99%. The average and standard deviation of performance, system error, and population size are provided in Table 2.

Table 2 shows the average of the experiment after 30 runs and the standard deviation indicates how much variation or dispersion exists from the average. In the 6-bit multiplexer with GA subsumption off, the average of the performance is 0.99±0.02, system error is 0.047±0.032, and the population size is 314±12 at about 6000 iteration. In the 6-bit multiplexer with GA subsumption on, the average of the performance is 0.99±0.02, system error is 0.032±0.027, and the population size is 64±13 at about 12000 iteration. In the 11-bit multiplexer with GA subsumption off, the average of the performance is 0.98±0.02, system error is 0.069±0.025, and the population size is 750±13 at about 15000 iteration. In the 11-bit multiplexer with GA subsumption on, the average of the performance is 0.97±0.02, system error is 0.059±0.040, and the population size is 160±22 at about 36000 iteration.

The statistic t-test is also be used to analyze the results. The t-test is the most powerful parametric test for calculating the significance of a small sample mean. The results shows that the population size in the 6-bit and 11-bit multiplexer reach the significant differences (p-value < 0.001). The p-value of the performance and system error are 0.50 and 0.03 in the 6-bit multiplexer. In the 11-bit multiplexer, the p-value of the performance and system error are 0.09 and 0.15.

4.2 The Population of Code Fragments

One of the possible reasons that impacts the system performance may occur in the generation of the population of code fragments. To define the uniformly random when the system generates the population of code fragments is very important. The method about the population of code fragments is not clearly described in the previous works [17]. Hence, in this section, we provide our method to generate the population of code fragments. First, the system randomly selects the number of nodes to generate the tree (node 1~7). Second, the system will consider all nodes in the tree and randomly replace the internal node to an appropriate operator (AND, OR, NAND, NOR, and NOT) and the leaves to terminal symbols (*T*0, *T*1, …, *Tn*-1). The system repeats the above steps and generates the trees until it reaches the maximum population size of the code fragments.

4.3 The Technique for Speeding Up

Twenty-bit or higher dimension multiplexer problem were not tested in our research due to the major drawback of our proposed method, that is, speed. Since each time when the function of determining if a classifier is more general than another classifier is called, the 2^L truth table should be traversed. However, improvements in speed can be done by improvements in implementation, for example, storing the unmatched input string of classifier A.

Such the technique for speeding up could be based on the fact that the most crucial concept of the proposed isMoreGeneral function showed in the flow diagram in Figure 6 is that, because if any S_i is unmatched by Classifier A but matched by Classifier B, the function can immediately judge that Classifier A is not more general than Classifier B and return. Hence, the unmatched S_i of Classifier A is important. So, we can develop speeding up technique based the fact. We can store the unmatched S_i of Classifier A for future use (not need to go through all S_i every time when an isMoreGeneral function is used). Every time that S_i is established in the isMoreGeneral function, S_i that was previously matched by Classifier A should be ignored (see Figure 11 for illustration).

Table 2 The average and standard deviation of 6-bit and 11-bit multiplexer. i represents the i^{th} iteration and *represents $p < 0.001$.

The average of 30 runs	Average and standard deviation		
	performance	System error(/0.001)	Population size
6-bit multiplexer with the GA subsumption off (6000i)	0.99±0.02	47.33±32.62	314.07±12.87*
6-bit multiplexer with the GA subsumption on (12000i)	0.99±0.02	32.43±27.89	64.90±13.72*
11-bit multiplexer with the GA subsumption off (15000i)	0.98±0.02	69.03±25.82	750.20±13.85*
11-bit multiplexer with the GA subsumption on (36000i)	0.97±0.02	59.96±40.41	160.90±22.28*

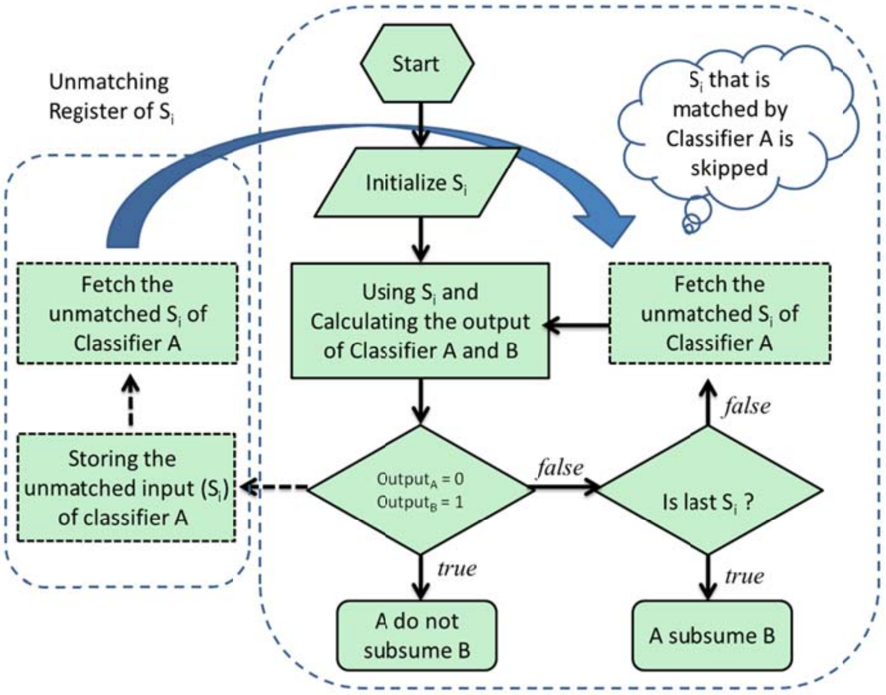
Unmatching Register of S_i
Start
S_i that is matched by Classifier A is skipped
Initialize S_i
Fetch the unmatched S_i of Classifier A
Using S_i and Calculating the output of Classifier A and B
Fetch the unmatched S_i of Classifier A
false
Storing the unmatched input (S_i) of classifier A
$Output_A = 0$ $Output_B = 1$
false
Is last S_i ?
true
true
A do not subsume B
A subsume B

Figure 11 The possible speeding up technique that has not yet been implemented in this study.

5. CONCLUSION

The subsumption mechanism in the code fragment representation of XCS was successfully implemented for low dimensionality problems. In the results of the experiment using multiplexer problem from 6-bit to 11-bit, we found that the number of the macroclassifiers in the final [P] reduced. This work shows that the benefits of subsumption transferred to code fragment based systems, but additional work is required to develop a scalable subsumption mechanism.

6. ACKNOWLEDGEMENTS

This work was fully supported by the Taiwan National Science Council under grant numbers NSC-101-2220-E-009-039 and NSC-101-2627-E-010-001. This work was also supported in part by the UST-UCSD International Center of Excellence in Advanced Bioengineering sponsored by the Taiwan National Science Council I-RiCE Program under grant number NSC-101-2911-I-009-101.

7. REFERENCES

[1] Wilson, S. W. Classifier fitness based on accuracy. *Evol. Comput.*, 3, 2 1995), 149-175.

[2] Akbar, M. A. and Farooq, M. Application of evolutionary algorithms in detection of SIP based flooding attacks. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation* (Montreal, Canada, 2009). ACM.

[3] Gandhe, A., Yu, S.-H., Mehra, R. and Smith, R. *XCS for Fusing Multi-Spectral Data in Automatic Target Recognition*. Springer Berlin / Heidelberg, 2008.

[4] Armano, G., Murru, A. and Roli, F. Stock Market Prediction by a Mixture of Genetic-Neural Experts. *International Journal of Pattern Recognition and Artificial Intelligence (IJPRAI)*, 16, 5 2002), 501-526.

[5] Chen, A.-P., Hsu, Y.-C. and Chang, J.-H. Applying Extensible Classifier System to Inter-market Arbitrage with High-

Frequency Financial Data. In *Proceedings of the 2007 International Conference on Convergence Information Technology* (2007). IEEE Computer Society.

[6] Tsai, W.-C. and Chen, A.-P. *Global Asset Allocation Using XCS Experts in Country-Specific ETFs*.2008.

[7] Passaro, A., Baronti, F. and Maggini, V. Exploring relationships between genotype and oral cancer development through XCS. In *Proceedings of the 2005 workshops on Genetic and evolutionary computation* (Washington, D.C., 2005). ACM.

[8] Baronti, F., Micheli, A., Passaro, A. and Starita, A. *Machine learning contribution to solve prognostic medical*. Elsevier, 2007.

[9] Bernauer, A., Arndt, G., Bringmann, O. and Rosenstiel, W. Autonomous multi-processor-SoC optimization with distributed learning classifier systems XCS. In *Proceedings of the 8th ACM international conference on Autonomic computing* (Karlsruhe, Germany, 2011). ACM.

[10] Shankar, A. and Louis, S. *Learning classifier systems for user context learning*.2005.

[11] Shankar, A. and Louis, S. J. XCS for Personalizing Desktop Interfaces. *Evolutionary Computation, IEEE Transactions on*, 14, 4 2010), 547-560.

[12] Orriols-Puig, A., Casillas, J. and Bernadó-Mansilla, E. Genetic-based machine learning systems are competitive for pattern recognition. *Evolutionary Intelligence*, 1, 3 2008), 209-232.

[13] Wilson, S. W. Get real! XCS with continuous-valued inputs. *Learning Classifier Systems*2000), 209-219.

[14] Lanzi, P. L. Extending the Representation of Classifier Conditions Part I: From Binary to Messy Coding. In *Proceedings of the genetic and evolutionary computation conference* (1999).

[15] Lanzi, P. L. and Perrucci, A. Extending the Representation of Classifier Conditions Part II: from messy coding to S-Expressions. In *Proceedings of the genetic and evolutionary computation conference* (1999).

[16] Lanzi, P. L. *XCS with Stack-Based Genetic Programming*, 2003.

[17] Iqbal, M., Browne, W. N. and Zhang, M. Extracting and using building blocks of knowledge in learning classifier systems. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference* (Philadelphia, July 7-11, 2012). ACM.

[18] Koza, J. Genetic programming as a means for programming computers by natural selection. *Stat Comput*, 4, 2 (1994/06/01 1994), 87-112.

[19] Wilson, S. W. Mining oblique data with XCS. *Advances in Learning Classifier Systems. Third International Workshop, IWLCS 2000. Revised Papers (Lecture Notes in Artificial Intelligence Vol.1996)*(2001), 158-174.

[20] Butz, M. and Wilson, S. *An Algorithmic Description of XCS*. Springer Berlin / Heidelberg, 2001.