A Novel Representation of Classifier Conditions Named Sensory Tag for the XCS in Multistep Problems

Liang-Yu Chen
Institute of Biomedical Engineering, National Chiao Tung University, 1001 University Road, Hsinchu 300, Taiwan, R.O.C..
lychen1211@cs.nctu.edu.tw

Po-Ming Lee
Institute of Computer Science and Engineering, Department of Computer Science, National Chiao Tung University, 1001 University Road, Hsinchu 300, Taiwan, R.O.C..
pmli@cs.nctu.edu.tw

Tzu-Chien Hsiao
Biomedical Electronics Translational Research Center and Biomimetic Systems Research Center, Institute of Biomedical Engineering, Department of Computer Science, National Chiao Tung University, 1001 University Road, Hsinchu 300, Taiwan, R.O.C..
labview@cs.nctu.edu.tw

ABSTRACT

Dynamically adding sensors to the Extended Classifier System (XCS) during its learning process in multistep problems has been demonstrated feasible by using messy coding (XCSm) and s-expressions (XCSL) as the representation of classifier conditions. XCSm and XCSL shown improved performance when new sensors were dynamically added to the agent of these systems in addition to the original available sensors during the learning process. However, these systems may suffer from overspecified problem and some logical operators (or clauses) could lead instability of the performance. Despite studies have suggested that these issues can be solved by appropriate parameter tuning, in the last study, we introduced a novel representation of classifier conditions for the XCS, named Sensory Tag (ST) (called XCS with ST condition, XCSSTC) to achieve the same goal as XCSm and XCSL, but inherent most of the mechanisms of the XCS to solve those issues that the XCSm and XCSL encountered without any parameter tuning. The experiments of the proposed method were conducted in the multistep problems (i.e. Woods1 and Maze4). The results indicate that the XCSSTC is capable of being dynamic added additional sensors to improve performance during the learning process, and moreover, the XCSSTC shown a better performance in regard to learning speed than the other methods.

Categories and Subject Descriptors

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

Keywords

Learning Classifier Systems; XCS, Scalability; Machine Learning

GECCO'15 Companion, July 11 - 15, 2015, Madrid, Spain
© 2015 ACM. ISBN 978-1-4503-3488-4/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2739482.2768446.

1. INTRODUCTION

Researchers have developed various intelligent systems and algorithms inspired from the nature. For instance, John Holland invented the Genetic Algorithms (GA) and tried to apply them to the Machine Learning (ML) field. He put forward the architecture of a rule-based system called Learning Classifier System (LCS) in 1976[1]. Nowadays, LCS has become one of the research mainstreams in the field of intelligent system. Various LCSs have been proposed, the Extended Classifier System (XCS) is one of the most popular ones in the application domain. The XCS is a LCS that learns to solve a given task by using a set (called population set [P]) of "IF condition THEN action" rules called classifiers. The XCS interacts with the environment to extract an optimal policy for collecting maximum reward from the environment by continuously update the parameters of the classifiers [2]. The XCS has been proved to be capable of learning accurate and general rules and has excellent performance in a wide range of real world applications which includes security [3, 4], finance [5-7], medical research [8, 9], and chip design [10]. Numerous studies have shown that the XCS is competitive to the traditional ML techniques [11]. An ordinary LCS uses binary strings of fixed-length as its classifier conditions. The limitation of the learning paradigm of using binary strings as representation of classifier conditions has not been aware until the research path moved from seeking for optimal performance to generalization [12-14]. Different types of representation of classifier conditions were proposed since then to apply the XCS to different problem domains. For instance, Wilson [15] modified the classifier condition of the XCS by adopting real-value representation (i.e. the version of XCS taking real inputs, called XCSR). This modification enable the XCS to cope with continuous input attributes such as stock index, temperature, height and weight. Lanzi [16], [17] implemented XCS with two types of representations of classifier conditions, they are variable length messy coding and S-expressions representation of classifier conditions. The messy coding scheme adopts binary encoding without the restriction of position linkage between the input attributes of the classifiers. The S-expressions is a more complex representation that was introduced to generalize classifier conditions. The S-expressions enable the XCS to cope with dynamic types of input variables. The advantage of variable length messy coding and S-expressions representation are that the method does not required the system to bind classifier syntax to a specific sensory configuration. Accordingly, the XCS with these types of representations of classifier conditions can automatically adjust or replace its input attributes. The messy version of XCS (XCSm) and XCS with LISP s-expressions (XCSL) shown improved performance when new sensors were dynamically added to it in addition to the original available sensors during the learning process [16, 17].

This capability could make XCS more flexibility to solve more general problems. However, the XCSm and XCSL are slower than the ordinary XCS, and moreover, the XCSm may suffer from overspecified problem, and furthermore, some logical operators (or clauses) could cause instability in XCSL's performance. Despite the former studies have suggested that these issues can be solved by appropriate parameter tuning, in the current study, we introduced a novel representation of classifier conditions for the XCS, named Sensory Tag (ST), by inheriting most of the mechanisms of the original XCS, to achieve the same goal as the XCSm and XCSL with a better performance in regard to learning speed, and without having those issues which XCSm and XCSL encountered. The concept of the proposed representation is inspired by the messy coding representation proposed by Lanzi in [16]. The concept of messy coding introduced in [16] was to tag each sensor (i.e. the input attribute) with a tag. However, for the messy coding representation of classifier conditions proposed in [16], the tags were not unique in the classifier conditions. This lead the proposed system to suffer from overspecified problem and the instability of the system. In the current study, we argue that such the mechanism can be simplified in which the tags can be unique. Hash Table (HT) [18] was used in the current study to achieve this goal. We proposed an XCS with ST condition called XCSSTC [23], and its effectiveness has been verified in the Multiplexer (MUX) problem domain [21] and it can dynamically learn multiple problems [24]. This paper validated that the performance of the XCSSTC in the multistep problem domains (i.e. Woods1 and Maze4). The remainder of the article is divided into four main sections. Section 2 describes the components of an ordinary XCS and the algorithmic description of the proposed XCSSTC. The description of the applied problem domain and experimental design are also provided in this section. The result and discussion of this study are presented in Section 3. And finally, the conclusion is presented in the last section.

2. THE METHOD AND MATERIALS

2.1 Components of the XCS

The XCS is a rule-based LCS which classifier fitness is based on prediction accuracy but not prediction payoff [2]. This feature makes the XCS keeps both correct and incorrect rules and each rule represent a form of "IF condition THEN action". In the original XCS, each rule is divided into three parts as follows: a ternary alphabet [0, 1, #] representing condition (# indicates a bit can be ignored, also called "don't care"), a binary string representing an action, and three parameters for each classifier. Three parameters representing as: (i) p represents the classifier prediction which evaluates a value to predict the expected reward gaining from environment; (ii) □ represents prediction error which evaluates the error of the prediction p; (iii) F represents fitness value which evaluates the accuracy of the prediction p. Moreover, GA is applied to the XCS to enable it in discovering new classifiers by evolving the rule set to search for the classifiers which are maximal general and accurate. The detailed procedure of the XCS is described as the following. First, the XCS encodes the status of environmental input to binary string for the detector and uses it in matching operation. Second, the XCS builds an empty match set [M]. Third, the XCS searches the population set [P] for classifiers whose condition matches the environmental input, and places all the matched classifiers into the [M]. After that, if the [M] is empty, the XCS will apply an operation called "covering" to create a new classifier whose condition matches the environmental input and the action of the classifier is chosen randomly. Fourth, after the [M] has been generated, the XCS then calculates the weighted fitness from the sets of classifiers that each set of classifiers has same action. The following step is formed a Prediction Array (PA) and all weighted fitness values of each action will place into the PA. Fifth, the XCS then picks out an action which has maximal predicted payoff and occasionally selects an action randomly as output. Sixth, the Q-learning style reinforcement learning occurs after receiving payoff from the environment. After the effectors has performed the selected action to environment. The environment will give a feedback to the system. The system interprets the feedback in numeric which is generated through a payoff function predefined by user and the parameters (p, F) of each classifier are then updated based on the obtained payoff. The update procedure occurs in action set [A]. The [A] is the set of classifiers with same action that responsible for the environmental feedback. The XCS discovers the possible classifiers to adapt the environment state by using GA. GA is triggered occasionally to search for accurate classifiers in the solution space. During the process of updating parameters and rule discovery, there are two important mechanisms enable the XCS to generalize the learned rules. They are macroclassifiers and Subsumption.

The concept of macroclassifier is to add an additional numerosity parameter *num* to the classifiers in the XCS. A classifier with *num* = *n* can be considered as n regular classifiers. When XCS created a new classifier at the stages of covering operation or GA, XCS will scan the [P] to examine whether a macroclassifier exists with the same condition and action as 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. The use of the macroclassifiers technique reduces redundant classifiers and allows XCS extract generalized classifiers from population. The macroclassifiers technique can also reduce the time of searching classifiers to speed up the matching process. Subsumption deletion occurs after the update process of [A] and GA, also called action set subsumption and GA subsumption respectively. Action set subsumption will select an experienced classifier *cl* with $\square < \square_0$. Next, *cl* subsumes all classifiers in [A] for whose classifier condition are less general than *cl* by deleting them and the numerosity of *cl* is increased at the same time. GA subsumption occurs 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 (*exp*, the number of times of being in an [A], larger than θ_{sub}) and more general. Simultaneously, the numerosity of the subsuming classifier is incremented. Otherwise, XCS inserts the generated new classifiers into [P]. Results reported in [14] has shown the effectiveness of this technique in the XCS. For further details of the XCS, readers are recommended to see Butz's algorithmic description of the XCS [19].

2.2 ST as the Representation of Classifier Conditions in Multistep Problems

The method we proposed here is to use the HT to implement the concept of STs as the representation of classifier conditions. The HT stores a collection of (ST, SV) pairs in which each pair has a key and a value. The keys in an HT are unique, and the HT is capable of retrieving a value by using its corresponding key efficiently from itself based on an implementation of a hash function. HT may suffer memory hungry, however, the memory size of HT is decided by the dimensions of input condition. Unlike messy coding and s-expression representation, HT will not produce duplicated dimensions and suffer bloating problem. HT can dynamically extended its array size to fit the dimensions of input condition and it is very suitable extended to large real-world problems. In the proposed XCSSTC, the ST of a sensor is taken as a key whereas its Sensory Value (SV) is taken as the value corresponds to the key. Each classifier has its own HT as the representation of classifier conditions. The HT includes all the (ST, SV) data pairs. The condition bit which is # in a classifier of an ordinary XCS is simply ignored from the HT for the classifiers in the XCSSTC. For instance, in Woods1 problem, the agent has eight sensors to perceive the adjacent situation of corresponding cell in the environment. Each cell in the grid can be an obstacle (an "O" symbol with sensor codes "01"), a food (an "F" symbol with sensor codes "01"), or it can be empty (a "*" symbol with sensor codes "00"). When a condition of a classifier in an ordinary XCS is 00##0000##010111, the left-hand two bits are always those due to the object occupying the cell directly north of "*", with the remainder corresponding to cells proceeding clockwise around it. The classifier in XCSSTC stores only the set of data pairs {(D_0, 00), (D_2, 00), (D_3, 00), (D_5, 01), (D_6, 01), (D_7, 11)} in its HT. The detail mechanisms of the XCSSTC that are different from the original XCS are described as follows.

2.2.1 Matching Classifiers in the XCSSTC

In the XCSSTC, a matching process decides whether a classifier is matched by enumerating all the SVs in the HT of the cl. and compare them to the compared to the corresponding bit position (by using the ST) in the input string and the absent (ST, SV) pairs are considered as "don't care" and hence ignored. The pseudo code of the procedure of the XCSSTC to match a classifier *cl* from the population [P] whose conditions match against the environment input *s* will return true, otherwise return false is presented below:

```
 1: procedure DOES_MATCH(cl, s)
 2:     STs ← enumerate all STs in cl.cond
              and put all STs into the list
 3:     n ← total number of STs in STs
 4:     for i =  1 to n do
 5:         cl.val ← get the value from STs[i] in
                 cl.cond
 6:         val ← get binary bit from the state s with
                 corresponding STs[i]
 7:         if cl.val ≠ val  then
 8:             return false
 9:         end if
10:     end for
11:     return true
12: end procedure
```

2.2.2 Covering in the XCSSTC

In the covering process of the XCSST, a random classifier whose condition matches the current environmental state *s* and each (ST, SV) pair in the HT has probability $P_{\#}$ to be taken as "don't care" and removed is created. The pseudo code of the covering operation to create a new classifier that match the current input state *s* and advocate an action *a* missing in the match set [M] is provided below:

```
 1: procedure COVERING_OPERATION(s, a)
 2:     initialize classifier cl
 3:     initialize condition cl.cond  with length n
 4:     for i = 1 to n do
 5:        if RandomNumber[0, 1) < P# then
 6:           cl.cond[i] ← null as "don't care"
 7:        else
 8:           val ← get binary bit from the state s with
                     corresponding cl.cond[i]
 9:                   cl.cond[i] ← put (i, val) into hash table
10:         end if
11:     end for
12:     cl.action ← a
13:     return cl
14: end procedure
```

2.2.3 Rule Discovery in the XCSSTC

The rule discovery process of the XCSSTC first selects two parent classifiers from the [A] based on their fitness and produces two offspring by the parents. Then the conditions of two offspring are crossed with probability □. Uniform Crossover (UX) [26] is used for the GA in the XCSSTC. The UX, unlike one- and two-point crossover, evaluates each bit in the parent strings for exchange with a probability □. Empirical evidence suggested that it is a more exploratory approach to crossover than the traditional exploitative approach that maintains longer schemata. This may result in a more complete search of the design space with maintaining the exchange of good information. The procedure of the crossover operation is shown below:

```
 1: procedure CROSSOVER_OPERATION(cl1, cl2)
 2:     STs ← enumerate all STs in
                cl1.cond and cl2.cond, put all STs
                into the list and remove the
                duplicated STs.
 3:     n ← the total number of ST in STs
 4:     for i = 1 to n do
 5:        if RandomNumber[0, 1) < x then
 6:              cond1 ←  get the cl1.cond which ST
                            corresponding to STs[i].
 7:              cond2 ←  get the cl2.cond which ST
                            orresponding to STs[i].
 8:              swap cond1 and cond2
 9:        end if
10:     end for
11: end procedure
```

After the crossover operation, each (ST, SV) pair has a probability μ to be removed in the mutation operation. The absent STs are considered as "don't care" attributes. Then, with probability □, the XCSST randomly chooses an action for child. The pseudo code of the used mutation operation is presented below. The value of the prediction payoff of a child is the average of its' parents' prediction payoff. The prediction error and fitness are the average of the parents' values reduced by constants *predictionErrorReduction* and *fitnessReduction* respectively.

```
procedure MUTATION_OPERATION(cl, s)
    for i = 1 to n do
        if RandomNumber[0, 1) < □ then
            if cl.cond[i] = null then
                val ← get binary bit from the state s with
                    corresponding cl.cond[i]
                cl.cond[i] ← replaced by (i, val)
            else
                cl.cond[i] ← null as "don't care"
            end if
        end if
    end for
    if RandomNumber[0, 1) < μ then
        a ← cl.action
        cl.action ← randomly chosen action other than a
    end if
end procedure
```

2.2.4 Subsumption Deletion in the XCSSTC

Subsumption deletion occurs when a classifier rule is sufficiently accurate and more general than the other classifiers. For instance, if a classifier cl_1 has the same action as the other classifier cl_2. cl_1 is more general than cl_2 while both of classifier condition match against the environmental inputs and has the same accuracy, but cl_1 has more "don't care" symbol. Then cl_2 will be deleted and the num of cl_1 is increased by one. In the XCSSTC, there are two steps to determine which classifier rule is more general than the other classifier rule that can subsume the other. The first step is to count the number of the (ST, SV) pairs in the HT. The number of the absent STs is equivalent to the number of "don't care" bit. This action determines whether the classifier is general or not efficiently. For instance, determining whether classifier cl_1 is more general than classifier cl_2 or not, if cl_1 has more paired number of STs than cl_2, cl_1 will not be identified as general one. The second step is to compare the SVs of the classifiers corresponds to their STs. If cl_1 has a ST that cl_2 does not have, then cl_2 cannot subsumed into cl_1. On the other hand, if each corresponding ST has different SV between cl_1 and cl_2, then cl_1 also cannot be considered as general one. The procedure to determine whether a classifier cl_1 is more general than another classifier cl_2 is shown below:

```
procedure IS_MORE_GENERAL(cl1, cl2)
    x ← total number of STs in cl1.cond
    y ← total number of STs in cl2.cond
    if x ≥ y then
        return false
    end if
    X ← set of STs in cl1.cond
    Y ← set of STs in cl2.cond
    if X ⊆ Y then
        return false
    end if
    STs ← enumerate all STs in
          cl1.cond and put all STs into
          he list
    n ← the total number of ST in STs
    for i = 1 to n do
        cl1.val ← get the value from STs[i] in
              cl1.cond
        cl2.val ← get the value from STs[i] in
              cl2.cond
        if cl1.val ≠ cl2.val then
            return false
        end if
    end for
    return true
end procedure
```

2.2.5 Encapsulating and Reusing the Learned Knowledge When Adding New Sensors to the XCSSTC

Reusing the past learning experience is similar to the learning process of human beings by utilizing knowledge extracted from past learning experience to solve more complex problem in the same or related domains. Suppose we have a robot can move into eight adjacent cells to find the food. In the beginning, for a robot that only have four cardinal sensors to perceive the environment. After learning a period of time, we add four more sensors to the robot. In the XCSSTC, the robot can keep the learned building blocks of knowledge that were learned from the environment by using the original four sensors. We realized this capability of the XCSSTC by using a manner similar to the one proposed in [20]. The anterior classifier rules learned from solved problems is considered as a "last" population set $[P]_{-1}$ and the new empty population set is $[P]_0$ (also abbreviated as [P]). For instance, when applying the XCSSTC to the Maze4 problem, in the first 5000 problem instances, the agent learn from the environment with only four sensors and the learned population set was taken as the $[P]_{-1}$. Later, when adding the additional new four sensors to the XCSSTC, the covering process of the XCSSTC is then altered to a two-step procedure. The first step is to use the ordinary matching process to match the classifier rules in the $[P]_{-1}$ and pick out an experienced classifier ($exp > \theta_{sub}$) which has the highest fitness value. The second step is to apply the covering operation for the additional new STs for the classifier selected from the $[P]_{-1}$. Each ST paired with a SV consistent with the environmental input. For a probability of $P_{\#}$ the new (ST, SV) is not inserted into the classifier's HT. Thus, part of the "learned knowledge" which has been learned from the environment when only four sensors were available is used when new sensors are added to the agent. On the other hand, if there is no classifier matches the environmental input in the $[P]_{-1}$, the XCSSTC applies the ordinary covering operation to create a new classifier.

2.3 Experiment Design

2.3.1 Multistep Problems

The environment of multistep problems is a grid in which an agent is placed in the grid to find a food. The agent has a number of sensors (e.g., four or eight) to perceive the adjacent situation of corresponding cell in the environment. Each cell in the grid can be an obstacle (an "O" symbol with sensor codes 01), a food (an "F" symbol with sensor codes 01), or it can be empty (a "*" symbol with sensor codes 00). The agent can move into any of adjacent cell. If the adjacent cell is an obstacle, the agent unable to move into this cell, the agent must be left in the original place; if the adjacent cell is empty then the agent can move into the cell; finally, if the adjacent cell is a food, the agent will move into the cell and receive a constant reward to end the problem.

The experimental environments used here are Woods1 and Maze4 as show in Figure 1 and Figure 2 respectively. Each experiment consists of number of problem instance for the agent to solve. For each problem instance the agent is put to a randomly selected empty cell of the environment. Then the agent is controlled by the XCSSTC to keep moving until it reaches the food and receive a constant reward to end the problem. The experiment of the XCSSTC on each experiment was repeated for 30 times with a different random seed and all the reports are average result of the 30 runs.

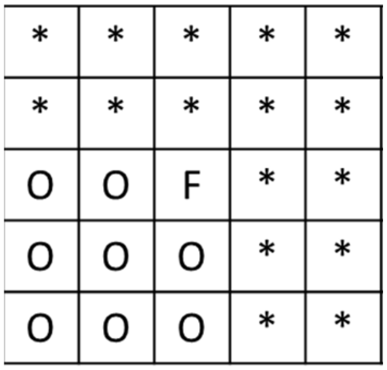

Figure 1 The Woods1 environment

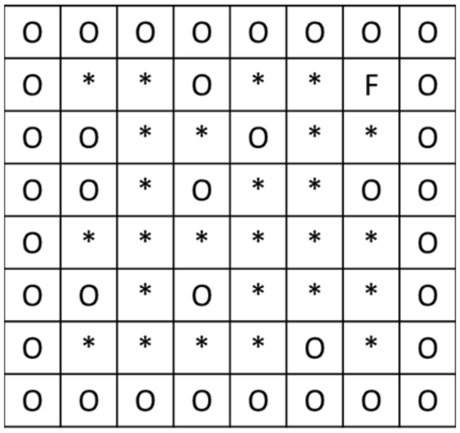

Figure 2 The Maze4 environment

2.3.2 Parameter Setup

This study adopted the setting of parameters mostly based on the suggestion provided in [19] and [16] as follow: fitness fall-off rate $\alpha = 0.1$; learning rate $\beta = 0.2$; the threshold of prediction error $\varepsilon_0 = 0.01$; fitness exponent $v = 5$; the threshold for GA application in the action set $\theta_{GA} = 25$; the probability of crossover operation □ = 0.8; mutation with probability $\mu = 0.01$; experience threshold for classifier deletion $\theta_{del} = 20$; the fraction of mean fitness $\delta = 0.1$; the threshold for subsumption $\theta_{sub} = 20$; probability of "don't care" symbol in covering $P_{\#} = 0.3$; reduction of the prediction error *predictionErrorReduction* = 0.25; reduction of the fitness *fitnessReduction* = 0.1; The number of classifiers used, denoted by N, is 800 and 1600, for the Woods1 and Maze4 multistep problems respectively.

3. RESULT AND DISCUSSION

3.1 Comparing the XCSSTC to the XCSm and XCSL

Figure 3 shows that for Woods1 problems, when $P_{\#}$ was set to 0.3, the XCSSTC reached accuracy rate 100% at about 500 problem instances. Although not shown in Figure 3, the XCSSTC also reached its minimum system error at the same point, and the number of macroclassifiers of the XCSSTC in Woods1 problem reduced to approximately 185 at the end of the learning process. Notably the original XCSm [16] cannot reach optimal performance in Woods1 problem and after tuning parameters, the enhanced version of XCSm reached accuracy rate 100% at about 3,000 problem instances which is slower than XCSSTC.

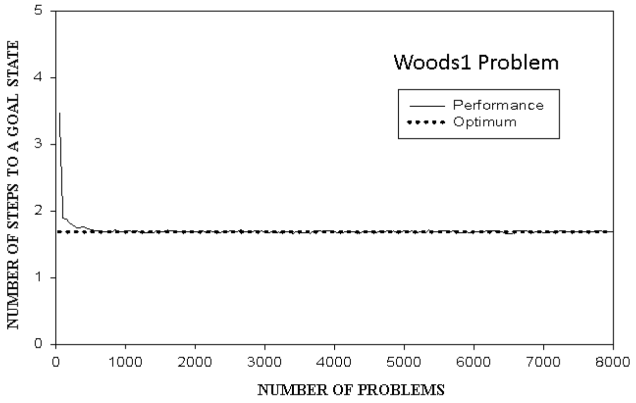

Figure 3 Result of XCSSTC for the Woods1 problem

Figure 4 shows that for the Maze4 problem, when $P_{\#}$ was set to 0.3 the XCSSTC reached accuracy rate 100% at about 750 problem instances. Although not shown in Figure 4, the XCSSTC also reached its minimal system error at the same point, and the number of macroclassifiers of the XCSSTC in Maze4 problem reduced to approximately 400 at the end of the learning process. Notably the original XCSm cannot reach optimal performance in Maze4 problem.

After parameter tuning, the XCSm reached accuracy rate 100% at about 4,500 problem instances [16] and the XCSL can reach accuracy rate 100% at about 1,000 problem instances [17] when an additional sensor that provide the "distance between the agent and the food" information was given. However, the number of problem instances required for the XCSSTC to learn the problem is smaller than both of them.

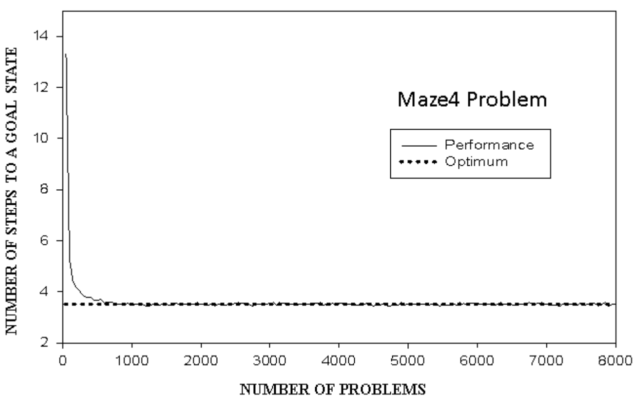

Figure 4 Result of XCSSTC for the Maze4 problem

Unlike the XCSm, the STs in the XCSSTC are unique in the classifier conditions and this lead the proposed system avoid overspecified problem and hence do not need to change the way that the rule-discovery component operates in the original XCS. We believe that is the reason why XCSSTC outperforms XCSm in the multistep problem without tuning parameters of the results. The reason of the XCSSTC for outperforming the XCSL may be due to the difference in degree of freedom since the S-expression enabled a more complex solution space. More problem instances is required for the XCSL to evolve appropriate operators in its classifier conditions than the XCSSTC to evolve appropriate classifier conditions.

3.2 The Effect of Encapsulating and Reusing Building Blocks of Knowledge

We've also tested the effect of dynamically adding new sensors to the XCSSTC, and examined the effect of encapsulating and reusing the building block of knowledge when adding new sensors to the XCSSTC. The experiment was also conducted in the Woods1 and Maze4 problem domains. During the experiments we provided the agent with the tags correspond to only four cardinal sensors in the first 5,000 problem instances. After the 5,000 problem instances, we added four more sensors to the agent to observe the performance changes in the agent. We also specifically reused the $[P]_{-1}$ learned by the XCSSTC in the first 5,000 problem instances by using the technique described in Section 2.2.5.

The results of our first experiment in both Woods1 (see Figure 5) and Maze4 (see Figure **6**) can be divided into two parts. First, in the first 5,000 problem instances where the XCSSTC only had 4 sensors, the performance improved over problem instances, but does not reached the optimal performance since it did not have sufficient sensors to build a complete payoff map of the environment. Second, after the first 5,000 problem instances, the result indicates that by reusing the learned $[P]_{-1}$, the problem instances required for the XCSSTC to reach the optimal performance was reduced compare to the XCSSTC without reusing the $[P]_{-1}$. However, this improvement was small. XCSSTC with reusing the $[P]_{-1}$ (365±231.71) is not significantly (t(58) = 4.258, p > .05) faster than XCSSTC without reusing the $[P]_{-1}$ (603.33±192.75) in the Woods1 problem. XCSSTC with reusing the $[P]_{-1}$ (1093.33±838.92) is not significantly (t(58) = 0.081, p > .05) faster than XCSSTC without reusing the $[P]_{-1}$ (1108.33±547.88) in the Maze4 problem. We investigate the population set of the XCSSTC at the 5,000th problem instance and we found that there were too many experienced classifiers with high fitness value that were overgeneral. The final [P] of the XCSSTC at the 5,000th problem instance in Maze4 is show in Table 1. If the reused classifier's classifier condition is empty (i.e. overgeneral), the mechanism of reusing the learned knowledge will become meaningless and even misleading in certain circumstances.

To avoid this issue, we modified the procedure described in 2.2.5 which reuses the building blocks of knowledge during the covering procedure. The modified procedure picks out an experienced classifier ($exp > \theta_{sub}$) which has the highest fitness value and also not empty in its condition (i.e. the HT). Figure 7 and Figure 8 show the effect of the modification in the procedure of reusing $[P]_{-1}$.

It is obvious that the modification of the procedure greatly improved the effectiveness of reusing the learned $[P]_{-1}$ in Maze4 problem. XCSSTC with reusing the $[P]_{-1}$ (241.66±141.47) is not significantly (t(58) = 8.146, p > .05) faster than XCSSTC without reusing the $[P]_{-1}$ (603.33±192.75) in Woods1 problem. XCSSTC with reusing the $[P]_{-1}$ (433.33±200.55) is significantly (t(58) = 6.230, p < .05) faster than XCSSTC without reusing the $[P]_{-1}$ (1108.33±547.88) in Maze4 problem. Furthermore, the final [P] of the XCSSTC is shown in Table 2. It is found that in the end of the learning process, the classifiers that are both accurate and general were evolved. After the examination of the $[P]_{-1}$ and [P] we found that the reuse of the $[P]_{-1}$ has been effective in Woods1 and Maze4 problems because of the non-overgeneral learned rules that were related to the end of the path (i.e. last step near the food).

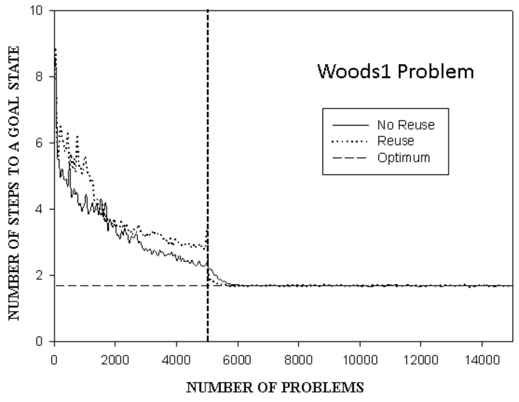

Figure 5 Result of XCSSTC by reusing the four cardinal sensors and without reusing the four cardinal sensors in Woods1 problem

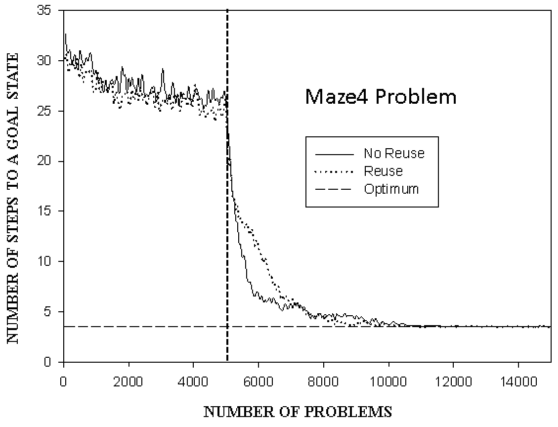

Figure 6 **Result of XCSSTC by reusing the four cardinal sensors and without reusing the four cardinal sensors in Maze4 problem**

Although the agent here was given only 4 sensors and cannot accurately identify the environment, the agent can still be accurate about the "next step" when the food is sensed in the given 4 sensors. On the other hand, in the environmental states those 4 sensors were insufficient to identify the state and cannot suggest a promising classifier action, the [P] would lead to more "overgeneral" classifiers with high fitness value. The "overgeneral" here means empty in HT.

We've conducted another experiment described below to validate to support this finding about the reason of why reused $[P]_{-1}$ being effective. We were curious about the performance changes when the agent already has its own $[P]_{-1}$ but was learned from different problem domain. The experiment was conducted by applying the XCSSTC (with only four cardinal sensors) to Woods1 problem for 10k problem instances then store the $[P]_{-1}$, then reuse the learned $[P]_{-1}$ with full eight cardinal sensors, in the Maze4 problem for the later 10k problem instances. We suspected that if the statement about the reason of why the reuse of $[P]_{-1}$ being effective was true, we hypothesize that this reuse of building blocks of knowledge should lead to "no improvement" at least in learning performance. Since the direction of the food in the end of the problem is totally different in Woods1 and Maze4 problems, proved that when prior learning creates behaviors that reduce or eliminate necessary experience in the new context, the past experience will not help [22].

The result of the experiment is shown in Figure **9**. The result indicate that the misuse of the $[P]_{-1}$ not only did not enhance the performance of learning in the later 10k problem instances, it furthermore lead to the restriction of learning in the later problem. This may be due to that the reuse of $[P]_{-1}$ since it may be misleading and hard to re-learn. The reason of classifiers in $[P]_{-1}$ for being hard to re-learn is that the SV of the STs (or being empty) related to the 4 sensors given in the first 10k problem instances were incorrect should be revised. The only way to revise them was through global search. However, there is only the mutation operator and no covering process involved.

We also found the without the reuse of the learned $[P]_{-1}$, the performance of the XCSSTC was slightly slower than the XCS [19]and equal to the XCS when the $[P]_{-1}$ was reused. It is worth to mention that this is caused by that in the current study we applied two-bit encoding for the SVs of classifiers in the XCSSTC. While the XCS applies the mutation operator in a one-bit level, the XCSSTC applies the mutation operator in a two-bit encoding.

Table 1 The [P] of the XCSSTC for learning first 5,000 problems with 4 cardinal sensors in Maze4

condition	a	p	□	F	n	exp	as
{(D0, 11)}	0	1000	0	1	33	1117	38.69
{(D2, 11)}	2	1000	0	1	35	662	40.44
{}	5	100.23	15.40	.99	146	37197	149.50
{}	4	120.50	32.98	.98	116	37946	130.37
{}	6	123.47	36.46	.97	132	38731	139.91
{}	7	96.046	13.94	.96	87	37524	113.51
(...)							
{(D2, 01), (D0, 00)}	1	142.49	103.89	0	1	19	93.24
{(D0, 01)}	5	67.43	24.36	0	1	129	102.52
{(D6, 01), (D2, 00)}	1	190.25	304.00	0	1	24	88.55

a: action, p: predicted payoff, □: system error, F: fitness value, n: numerosity, exp: *exp*, as: estimated action set size

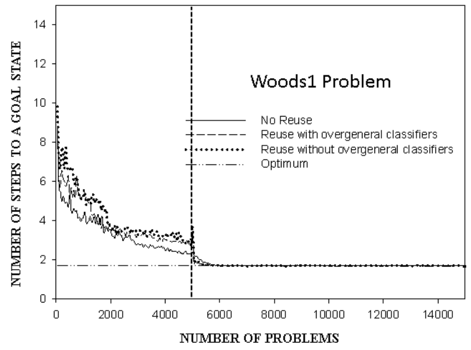

Figure 7 Result of XCSSTC by reusing the four cardinal sensors without overgeneral classifier for Woods1 problem

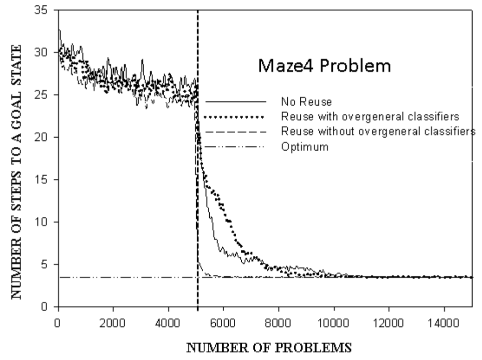

Figure 8 Result of XCSSTC by reusing the four cardinal sensors without overgeneral classifier for Maze4 problem

Table 2 The final [P] of the XCSSTC for learning 15,000 problems with 8 cardinal sensors in Maze4

condition	a	p	□	F	n	exp	as
{(D0, 11)}	0	1000	0	1	5	2669	7.31
{(D2, 11)}	2	1000	0	1	11	2015	8.12
{(D1, 11)}	1	1000	0	1	18	5433	16.82
{(D5, 00), (D4, 01), (D0, 01)}	1	502.24	5.59	1	24	1914	23.58
{(D7, 00), (D6, 00), (D5, 01), (D4, 00), (D3, 01), (D1, 01)}	7	356.83	1.12	1	11	119	12.40
{(D5, 01), (D4, 01), (D3, 01)}	3	181.39	.47	1	19	7628	22.64
(...)							
{(D6, 01), (D4, 00), (D1, 00)}	2	324.73	109.48	0	1	15	15.02
{(D5, 01), (D2, 01), (D0, 00)}	0	283.16	52.95	0	1	14	22.15
{(D5, 01), (D4, 00)}	1	437.92	138.36	0	1	17	16.44

a: action, p: predicted payoff, □: system error, F: fitness value, n: numerosity, exp: *exp*, as: estimated action set size

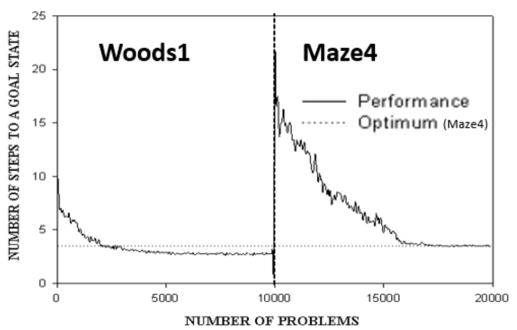

Figure 9 **Result of XCSSTC by reusing the four cardinal sensors without overgeneral classifier for Woods1 problem in the Maze4 problem.**

This will lead to the increase of the number of problem instances required to learn the problems. Hence, the comparison between the learning speeds between the XCS and the XCSSTC in the Woods1 and Maze4 should not be considered as a proof that the reuse of learned $[P]_{-1}$ cannot lead to better performance. In the feature, the authors will test XCSSTC in a more complex domain and its scalability.

4. CONCLUSIONS

The current study has demonstrated the capability of the XCSSTC in being dynamically added sensors to learn multistep problems. The results indicate that the XCSSTC solved the instability issues of systems reported in related previous studies and furthermore improve the learning speed. Moreover, we have develop and tested a technique of encapsulating and reusing the learned $[P]_{-1}$ and proved the effectiveness of it in improving the learning speed. We believe that the proposed approach can lead to building intelligent systems that can reuse the learned building blocks of knowledge and also automatically adjust / replace its input attributes and learn to perform complex tasks in larger scale.

5. ACKNOWLEDGMENTS

This work was fully supported by Taiwan Ministry of Science and Technology under grant numbers MOST 103-2221-E-009-139. This work was also supported in part by the "Aim for the Top University Plan" of the National Chiao Tung University and Ministry of Education, Taiwan, R.O.C..

6. REFERENCES

[1] Holland, J. 1976. Progress in Theoretical Biology in R. Rosen and FM Snell editors. Academic Press.

[2] Wilson, S. W. 1995. Classifier fitness based on accuracy. Evolutionary computation, 3, 2, 149-175.

[3] Akbar, M. A. and Farooq, M. 2009. Application of evolutionary algorithms in detection of SIP based flooding attacks. ACM, 1419-1426.

[4] Gandhe, A., Yu, S.-H., Mehra, R. and Smith, R. E. 2008. XCS for fusing multi-spectral data in automatic target recognition. Springer, 147-167.

[5] Armano, G., Murru, A. and Roli, F. 2002. Stock market prediction by a mixture of genetic-neural experts. International Journal of Pattern Recognition and Artificial Intelligence, 16, 05, 501-526.

[6] Chen, A.-P., Hsu, Y.-C. and Chang, J.-H. 2007. Applying Extensible Classifier System to Inter-market Arbitrage with High-Frequency Financial Data. IEEE, 709-714.

[7] Tsai, W.-C. and Chen, A.-P. 2008. Global Asset Allocation Using XCS Experts in Country-Specific ETFs. In Convergence and Hybrid Information Technology (Novotel Ambasador Busan, Busan, Korea, November 11-13, 2008). ICCIT'08. IEEE, 2, 1170-1176.

[8] Baronti, F., Micheli, A., Passaro, A. and Starita, A. 2006. Machine learning contribution to solve prognostic medical problems. Outcome Prediction in Cancer, 261.

[9] Passaro, A., Baronti, F. and Maggini, V. 2005. Exploring relationships between genotype and oral cancer development through XCS. In Proceedings of the 7th annual workshop on Genetic and evolutionary computation (Washington, DC, USA, June 25 - 29, 2005). GECCO '05. ACM, New York, NY, 147-151.

[10] Bernauer, A., Arndt, G., Bringmann, O. and Rosenstiel, W. 2011. Autonomous multi-processor-SoC optimization with distributed learning classifier systems XCS. In Proceedings of the 8th ACM international conference on Autonomic computing (Karlsruhe, Germany, June 14 - 18, 2011). ACM, New York, NY, 213-216.

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

[12] Kovacs, T. 1998. XCS classifier system reliably evolves accurate, complete, and minimal representations for Boolean functions. Springer, 59-68.

[13] Lanzi, P. L. 1999. An analysis of generalization in the XCS classifier system. Evolutionary computation, 7, 2, 125-149.

[14] Wilson, S. W. 1998. Generalization in the XCS classifier system.

[15] Wilson, S. W. 2000. Get real! XCS with continuous-valued inputs. Springer, 209-219.

[16] Lanzi, P. L. 1999. Extending the representation of classifier conditions part i: from binary to messy coding. In Proceedings of the genetic and evolutionary computation conference (Morgan Kaufmann, August 17, 1999). GECCO '99. 1, 337-344.

[17] Lanzi, P. L. and Perrucci, A. 1999. Extending the Representation of Classifier Conditions Part II: from messy coding to S-Expressions. In Proceedings of the Proceedings of the genetic and evolutionary computation conference (Morgan Kaufmann, August 17, 1999). GECCO '99. 1, 345-352.

[18] Horowitz, E. and Mehta, D. 2006. Fundamentals of data structures in C++. Galgotia Publications.

[19] Butz, M. V. and Wilson, S. W. 2001. An algorithmic description of XCS. In Advances in Learning Classifier Systems. Springer, 253-272.

[20] Iqbal, M., Browne, W. N. and Zhang, M. 2012. Extracting and using building blocks of knowledge in learning classifier systems. In Proceedings of the Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference (Philadelphia, PA, USA, July 07 - 11, 2012). ACM, New York, NY, 863-870.

[21] Chen, L.-Y., Lee, P.-M., Hsiao, T.-C. 2015. A Sensor Tagging Approach For Reusing Building Blocks of Knowledge. In Learning Classifier Systems. IEEE Congress on Evolutionary Computation (Japan, Sendai, May 25-28).

[22] Potter, M. A., Wiegand, R. P., Blumenthal, H. J. and Sofge, D. A. 2005. Effects of Experience Bias When Seeding With Prior Results. In IEEE Congress on Evolutionary Computation (Edinburgh, UK, Sept. 02 - 05, 2005). IEEE, 2730-2737.

[23] Chen, L.-Y., Lee, P.-M., Hsiao, T.-C. 2015. Dynamically Adding Sensors to the XCS in Multistep Problems: a Sensor Tagging Approach. IEEE Congress on Evolutionary Computation.

[24] Wu, Y.-M., Chen, L.-Y., Lee, P.-M., Hsiao, T.-C. 2015. Enable the XCS to Dynamically Learn Multiple Problems: A Sensor Tagging Approach. In Proceedings of the The Genetic and Evolutionary Computation Conference.