Journal of Artificial Intelligence and Data Mining (JAIDM), Vol. 11, No. 1, 2023, 13-27.

Shahrood University of Technology

Journal of Artificial Intelligence and Data Mining (JAIDM)
Journal homepage: http://jad.shahroodut.ac.ir

Research paper

An Efficient XCS-based Algorithm for Learning Classifier Systems in Real Environments

Ali Yousefi[1], Kambiz Badie[2*], Mohammad Mehdi Ebadzadeh[3] and Arash Sharifi[4]

1. Department of Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran.
2. E-Content & E-Services Research Group, IT Research Faculty, ICT Research Institute, Tehran, Iran.
3. Department of Computer Engineering, Amirkabir University of Technology, Tehran, Iran.
4. *Department of Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran.*

Article Info

Article History:
Received 01 November 2022
Revised 25 November 2022
Accepted 24 December 2022

DOI:10.22044/jadm.2022.12358.2384

Keywords:
Learning Classifier Systems, XCS Algorithm, Identification of Cycle and Overlapping.

**Corresponding author: k_Badie@itrc.ac.ir(K. Badie).*

Abstract

Recently, learning classifier systems are used to control physical robots, sensory robots, and intelligent rescue systems. The most important challenge in these systems, which are models of real environments, is its non-Markov quality. Therefore, it is necessary to use memory to store system states in order to make decisions based on a chain of previous states. In this research work, a memory-based XCS is proposed to help use more effective rules in classifier by identifying efficient rules. The proposed model is implemented on five important maze maps, and leads to a reduction in the number of steps to reach the goal and also an increase in the number of successes in reaching the goal in these maps.

1. Introduction

Learning classifier systems (LCSs) are rule-based systems that are mainly "if conditions, then act". In these systems, an evolutionary algorithm or other intuitive methods can be used to search the space of the existing rules, and at the same time, a learning process can be used to assign applications to the existing rules, which leads the search process to better rules [1, 2]. The term LCS was first introduced by Holland *et al*. as an extension of genetic algorithms [2]. Years later, he developed Cognitive Systems Level 1 (CS-1) in collaboration with Reitman [2]. Then he modified and standardized his previous works. However, the complex structure of the Dutch system prevented it from being easily implemented. Later, Wilson developed another type of LCS called XCS, where the fitness of the rules was determined only by the accuracy of their application [3]. LCSs have various applications in real environments [1-4].

XCS algorithm in problems such as segmentation of underwater images [7], knowledge extraction in factors based on economic models [8-9], understanding patterns in data [10], comparison of exploration and extraction strategies in engineering problems [11], anomaly detection based on cooperative fuzzy algorithms [12], automatic testing in organic computing systems [13], and competitive environments based on learning-based factors [14] is used. Another study presents the application of LCS learning algorithm in simulating drivers' lane selection behavior in microscopic simulation models of toll plazas [15]. LCS is used to evolve a set of ‘control rules’ for a number of Boolean network instances [16].

The biggest challenges are using LCS systems in real environments such as humanoid robots, navigation systems in tortuous paths, learning rescue robots, selecting appropriate environmental features, and human-inspired scaling [26-34]. In all these environments, the model systems must be non-Markov (i.e. work with memory). In many real LCS systems, due to the lack of sensory input in some cases, the current state of the environment cannot be determined by input alone. Therefore, the response of the system to the environment is not only determined by the current state and input but also requires the previous states of the system. Such

a system needs memory to store system states, so another condition called memory condition is used in these classifiers. These memory conditions can affect the system's power to decide on more appropriate responses to input in ambiguous and hidden situations. A stack-based turing machine can be used. Ideally, such a machine would have an infinite memory [35]. Considering the importance of LCS in natural systems, the aim of the present work is to develop a model of memory-based LCS for a real-world system. Using the proposed memory mechanism, the system uses the previous states of the system along with its current state and input to determine the best response to the input. Our approach focuses on the concepts of discrete response, non-Markov learning environment, and cyclic conditions of the system. Obviously, since natural environments are characterized by extensibility and increasing complexity, scalability is considered in our design by combining solutions with an evolutionary algorithm suitable for developing classifiers. This article is organized as what follows. In Section 2, previous works on LCS systems will be reviewed. Section 3 discusses the proposed method to improve the performance of XCS classifier systems. In this section, in addition to explaining the proposed method for optimizing the number of animation steps by identifying effective conditions and measures and adding them to the memory of the classifier system, how to use the genetic algorithm in this type of memory will also be explained. The proposed method that makes animat use its history is described in Section 4. Section 5 explains the implementation and performance evaluation of the proposed method. The final section provides conclusions as well as suggestions for further research works.

2. Literature Review

Holland put forward the notion of LCS to develop reinforcement learning or trial-and-error method. He sought to know how an artificial intelligence system could permanently adapt itself to the new and existing experiences and how an optimum system could be designed to represent knowledge by using continuous and flexible learning based on trial-and-error with increased rewards [37]. For this purpose, he made use of a certain type of genetic algorithms. The combination of evolutionary process and reinforcement learning resulted in the emergence of the CS-1 architecture.

In every discrete time cycle, CS-1 obtains an encrypted binary description of the current state of the environment. Then it determines a response memory space called ‘message list’ on the basis of the input, the previous action, and the current content. This is illustrated in Figure 8. In this system, the rule database consists of N rules in the form of condition-assertion, each of which called a classifier. The conditions are strings of three encryption symbols in the form (#, 0, 1). The symbol # acts as “all-purpose”, and allows the condition to be “0” or “1”. For example, the condition 1#1 matches both 101 and 111. The assertion part of each rule consists of an action and an internal message, which are both represented as binary strings. All parts of a rule are initialized by random values. Also a number of parameters come along with each rule including age of the rule, number of uses, and prediction of the reward for being used (that is also used as a parameter of fitness). Later on, Holland modified CS-1 and explained what needed to be changed to achieve a standard architecture. He named this new system Learning Classifier System (LCS) [2].

The most significant improvement of CS-1 was the fact that a reinforcement learning system was introduced by analogy with the economic metaphor “bucket brigade”, in which the application of a rule was judged by the appointment of credit [37]. In this method, those rules that acted in time cycles and resulted in external reward were considered as the middle individuals between the supply and demand chains. Rules have a parameter called ‘strength’ that is indicative of their credit in this chain. This parameter is both used to select actions and discover new rules through the genetic algorithm. In this model, the message list is extended so that multiple rules can send their assertions. The conditions of other rules do not have a fixed structure for considering the current state of environment, the content of message list, and the final action. Instead, all conditions and assertions have the same length, and the conditions can also include a logical NOT. Assertions use the alphabet used in conditions, i.e. [1], so that the information may pass through a condition or string (external input or internal message) which a rule matches in the present of #.

Booker presented a type of Holland’s standard system, and developed the idea of using genetic algorithms for discovering the configuration of problem space in order to separate the task of learning this support structure from the unit of appropriate actions for external rewards. For each of these two, there exists a different LCS. The first LCS receives an encoded binary description of the environment with the aim of discovering and

learning the appropriate rules within the general visible items. This is equal to learning the representation of the categories of subjects in the environment. The matched rules not only send their messages to the message list but also some of them are passed as input to the second LCS. Therefore, the second LCS is rewarded only when it uses these classifications correctly according to its current task.

Booker's innovations include detailed matching and levels of stimulation of the rules; but his main idea is to limit the process of discovering new rules to only those active rules (and not all the rules in the knowledge base) that have proved to be most effective. In this system, parents are selected from the message list [M]. Therefore, the rules are prevented from being combined with generalities that reveal many aspects of the problem. Booker developed his idea based on stimulation of genetic algorithm during learning, and allowed this process to run at a constant rate like Holland's reinforcement learning. Importantly, in this model of learning, rules contain an estimation of their stability, which is indicative of their changes when being rewarded. If the instability of a certain percentage of the rules in [M] is greater than a specified threshold, the fitness of stable rules will increase and the genetic algorithm will run. As a result, stable classifications are more interesting to the genetic algorithm. XCS system makes use of both fundamental genetics and stimulated genetics with a stability threshold [39-41].

Above all, the message list had been removed and the matched rules were classified based on actions in the process of human chain to create the action set [A]. In such a system, genetic algorithm is sensitive to the action of rules and partially resembles CS-1. The first parent is selected from the knowledge base according to its strength, whereas the second parent is selected from a subset of the population with identical action. ANIMAT controlled a simple agent in a two-dimensional environment, which was able to sense the content of eight positions in its environment and move in any one of eight paths which was open. Wilson showed that learning was possible, even to the extent that the paths, which led to food reward discovered in his project. However, he found out that the system was unable to learn hurried classifications by reinforcement. In order to stimulate an agent to achieve a reward from an initial position through the shortest path, he modified the structure of rules in a way that an estimation of the number of further steps required to receive the reward would be stored and, on the basis of this estimation, the children of each rule would be updated. In the action selection unit, this idea was implemented through dividing strength by distance. In addition, ANIMAT had a re-combinative operator that replaced dissimilar bits with # under the conditions of the parent to provide a more useful generalization.

Later on, Wilson simplified ANIMAT by developing his Zeroth-Level Classifier System (ZCS) [42, 43]. The major modification in ZCS was the fact that the human chain algorithm had been modified in a way that time differential learning could also be used.

Results from the evaluation of ZCS indicate that it can have a good, but not ideal, performance [36, 40, 42]. In differential learning, Wilson used an algorithm without policy called Q-learning along with the main algorithm [51]. He suggested that a fundamental genetic algorithm be used in the explained form. Bull investigated this issue. It has been shown that ZCS has a good performance in some sample problems but it seems that it is sensitive to some of its parameters. Also ZCS is more efficient than XCS in some noisy environments [21]. Although XCS and ZCS share some of their fundamental features, they have a number of basic differences [18].

After some changes in the architecture of ANIMAT and before presenting ZCS, Wilson developed an architecture specific to reinforcement learning, in which immediate reward was considered.

BOOLE was a system used for binary decision-making [36]. BOOLE borrows the mechanism $[M] \rightarrow [A]$ from ANIMAT but it eliminates the message list [M]. Furthermore, the limitation in selecting the second parent during the intercourse in the genetic algorithm (i.e. the condition of having the same action as the first parent) does not exist anymore in BOOLE, which causes the parents' strength to decrease through transfer to the children by means of an inherent mechanism of ZCS. It has been shown that strength decrease will compel more general rules to update more quickly and receive their rewards sooner [45]. Also in ZCS, rules that exist in [M] and do not exist in [A] will lose their strength with a certain tax rate.

Rules in the structure of Holland's LCS assumed the form condition-action-payoff. The main difference between Wilson's XCS and Holland's architecture lies in the fact that criterion of fitness in the former is accuracy of prediction instead of value of prediction. Although XCS architecture resolved

problems such as generalization, but its lack of internal message list or any other memory structure like CS-1 caused it to be useful only for learning the optimum policy in Markov systems. In such systems, the optimum policy only depends on the state of the sensory inputs and the agent can perfectly determine the state of the environment by means of these inputs. In many applications, the sensory inputs provide the agent with partial information about the current state of the environment. Thus the agent is unable to identify the state of the environment, and the classifier will regard overlapping states as identical. In this case, the classifier agent faces the problem of hidden state or overlapping observations and the action issued by the agent in response to the inputs will have a low accuracy. Such a system is called a non-Markov system, and the environment is said to be partially observable. In an environment that is partially observable, the classifier agent requires a memory with which it could compensate for the deficiency of sensory input data because the optimum policy cannot be recognized only on the basis of sensory inputs [46].

Holland's initial architecture includes an internal message list. This list stores information, and can be used as a temporary memory. However, research findings show that Holland's architecture are usually not successful in non-Markov environments. Various approaches to this issue have been so far proposed.

A pioneering design was proposed by Wilson in 1995 [3] and implemented by Lanzi in 1998 [21]. They added an internal memory to the system in the form of a register with one or a limited number of bits. Then they developed their proposed architecture by adding one condition and one action that were used in sensing and imposing policies on the internal register. Their system was called XCSM. This architecture was used in real non-Markov systems with overlapping sensory inputs. The results of the work of Wilson and Lanzi showed that XCSM was only efficient in simple problems in which the inputs had two or at most four overlapping states, but in other cases such as the problems of twisty paths it could not converge into the optimum solution. This motivation led Lanzi to develop another version of XCSM, which was called XCSMH. This new LCS made use of a new policy for updating memory and utilized a hierarchical strategy for selecting the appropriate action in response to the input. Testing the performance of XCSMH in twisty path problems with different complexities showed that this system could achieve a near-optimum solution in simple problems.

Later, Hamzeh *et al*. proposed two classifier systems with distributed architecture called PSXS and RPXCS [46, 47]. In these systems, the overlap of states is recognized, and the inputs are directed toward XCSs with lower ranks. Low-rank XCSs are equipped with history windows, and are each responsible for one of the overlapping states. The structure of the classifier in their systems is extremely complicated. Their method has proven to be successful in some of the standard tests.

In another study, Preen and Bull used Graph-based Dynamic Genetic Programming (DGP) to represent rules in XCS and XCSF (that was equivalent to a random Boolean network). Their DGP-XCS and DGP-XCSF can solve some twisty path problems but their solution is not optimum [49].

Recently, Xang *et al*. have proposed a new method of using memory in LCS systems to enhance the performance of XCS [31]. In their method, an internal message list is added to the system as the memory list, which has a length equal to the strings created by the identifiers. In this method, a small number of memory-based classifiers are developed to deal with the non-Markov states of the environment. Also a condition developed in the form of (memory, condition) is used along with a mechanism for recognizing the overlap of states to improve the performance of the classifier in hidden states. The performance of XCSMD classifier is closer to the optimum and higher than previous classifiers in real environments. However, the real problem in this method is memory wasting and the overload caused by applying the memory condition (to pass through overlapping states that cause ambiguity). In this case, the development capability of the system will face the problem of overload in computation and memory as the environment grows in size. In the context of more complicated systems, therefore, the solutions will be distant from the optimum solution. Today, classifier systems are used as applied tools to solve a variety of difficult problems. Sample learning without tagging, proper algorithm selection based on reasoning, support vector machine compaction for code fragment in decision making systems are among applications of these systems [50-53].

3. Proposed Structure of LCS

The diagram of our proposed approach to LCS is depicted in Figure 1. This algorithm adds three modules to the XCS architecture.

Memory list: It is used for the non-Markov aspect of the system. The structure of the list varies depending on the environment in which the classifier is used. In other words, the degree of the complexity of the problem will determine the boundaries of the memory. What is important is that, due to the existence of cycles in the states of most real systems, the stored states in the system's memory are limited and the memory should be assumed as limited.

Genetic algorithm (GA): It is used in two parts of our proposed structure. The first part is concerned with development of rules. Here, the population of rules is developed in an efficient and controlled way by using generalization and privatization through genetic algorithm. Secondly, this algorithm is used in the development and generalization of memory states. While interacting with the module that discovers cycles and overlapping states, this process that runs periodically or in an event-oriented manner will generalize the most used states and control the usage of limited memory space.

Module for identification of overlapping states: This module is necessary due to the fact that the memory is limited and bounded. The performance of this module can be either event-oriented or periodical. However it is, the output of the module is given to the genetic algorithm that controls the memory required for storing the states through generalization of frequently used states.

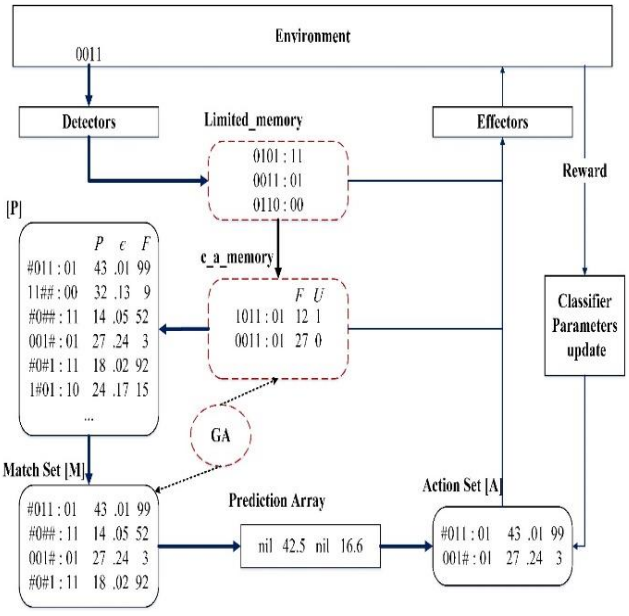


Figure 1. Proposed architecture for an LCS.

The three parts of the proposed architecture will be elaborated in the following phases:

3.1. Selection of appropriate condition-action at each step

The animat is sometimes located at a point where there are some obstacles in the surroundings. In Figure 2-a, for example, there are only three open spots to pass around the animat (*). These spots are marked in red. The remaining five spots are obstructed. The animat can move by selecting actions such as 000, 001, and 100; otherwise, it will be stopped by an obstacle. Figure 2-b illustrates the effect of the selected actions to take against the environment as well as the animat's motion.

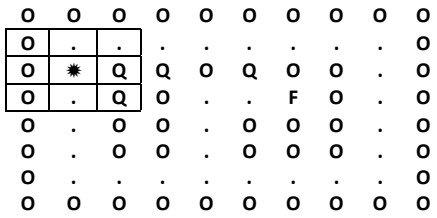


Figure 2-a. Example of an animat in the environment.

7	0	1
6	✷	2
5	4	3

111	000	001
110	✷	010
101	100	011

Figure 2-b. Action performed on the environment and the animat's motion.

Figure 3 shows a detector and the animat's actions in several steps of its motion in Figure 1. In step 1, the action 011 is selected and performed. However, there is an obstacle at the current position and the animat will not move on. In step 2, the action 010 is performed that again leads to an obstacle. In step 3, the action 010 is again selected, which prevents the animat from moving. However, in step 4, the action 100 is selected that causes the animat to move downwards. Therefore, 100 is one of the actions that causes movement with this input from the detector. In step 11, once again the input of the animat resembles that of the steps 1 to 4. In this position, after having performed five different actions on the environment, the animat moves upwards by selecting 000. In step 14, the animat repeats the action of step 1 that causes it to hit the obstacle. Similarly, in step 18, it selects 010 that it has already used in step 2. It is clear that this animat will not use the results of previous steps to improve its behavior.

The next table (figure 3) aims to show that an animat should not select the same action when it receives a repetitive input from the environment, which did not lead to movement in the previous

steps. Instead, it should use actions that have led to movement in the same position in previous steps. In this example, the animat in step 11 should make use of the action selected in step 4. How these condition-actions could be identified will be explained in the next section.

Step#	Condition	Action
1	000000011011000010010010	011
2	000000011011000010010010	010
3	000000011011000010010010	010
4	000000011011000010010010	100
5	000011011010000010010010	010
6	000011011010000010010010	011
7	000011011010000010010010	010
8	000011011010000010010010	001
9	000011011010000010010010	110
10	000011011010000010010010	000
11	000000011011000010010010	101
12	000000011011000010010010	101
13	000000011011000010010010	101
14	000000011011000010010010	011
15	000000011011000010010010	101
16	000000011011000010010010	000
17	010010000011000010010010	100
18	000000011011000010010010	010
19	000000011011000010010010	001
20	010010000011011000000010	110
21	010010000011000010010010	110
22	010010000011000010010010	011
23	010010000011000010010010	010
24	010010000011011000000010	100
25	010010000011011000000010	101
26	000000011011000010010010	001
27	010010000011011000000010	111
28	010010000011011000000010	000
29	010010000011011000000010	010
30	010010000010011011000010	010
31	010010000010010011000010	001
32	010010000010010011000010	100
33	010010000010010011000010	010
34	010010000010010010000010	110

Figure 3. Example of a detector and the animat's action in 34 steps.

3.2. Identification of condition-actions affecting movement

This section describes how to identify a condition and an action that cause the animat to move after having tested more than one action. In each step, both the condition and the action are stored. In addition, if a previously met condition appears in a step, the number of consecutive repetitions of this condition will be stored. Next, if the current condition does not differ from the previous one, the previous condition and action will be identified as the condition-action that affects the animat's movement. Given the values in Figure 3, the condition-action in step 4 has caused the animat to move after three trials. Also the condition-action in step 10 acts as an accurate classifier. If our only criterion is the animat's movement as opposed to its stop, this condition-action can be considered as an accurate classifier in XCS.

Various methods can be developed to use these condition-actions. For this purpose, two different modes will be implemented and examined below.

3.3. Providing XCS with a memory for storing effective condition-actions

In this method, a memory is allocated in XCS to the storage of the obtained condition-actions. Figure 4 shows this memory, which is called c_a_memory. This memory contains four fields, namely, fitness, action, condition, and used. Used is a Boolean field. This field marks condition-actions that were used to achieve the food reward in each problem. At the end of each problem, the value of the used field of the utilized condition-actions is set to 1.

Condition	Action	Fitness	Used
000000011011000010010010	100	1	0
000011011010000010010010	000	1	0
000000011011000010010010	000	1	0
000000011011000010010010	001	1	0
010010000011000010010010	010	1	0
010010000011011000000010	101	1	0
010010000011011000000010	010	1	0

Figure 4. Example of c_a_memory.

Depending on whether it has influenced the animat's achieving food, each condition-action receives a value that is stored in fitness. This field is a double-type field. How fitness is calculated will be explained below.

The allocated memory has a limited space. This memory acts as a FIFO queue. If it exceeds a certain amount of space, one of the condition-actions will be deleted. In the experiments in Section 5, the size of this memory is determined as 100% of the maximum number of classifiers in [p] and, in the experiments on Section 6, its size is determined as 20% of this number.

3.3.1. Storage of condition-actions

When a condition-action is identified, it will be stored in the memory. It should be noted that no new classification will be added to the set [p]. The condition-action is stored without any change (i.e. without using #) in the memory.

3.3.2. Using memory in XCS

This memory lies between the inputs received from detector and effector. The input received from detector is first searched for possible matches in the memory. If the input matches one of the conditions in the memory, the action corresponding to that condition will be sent to effector to be applied to the environment.

If the input does not match any condition, the default steps of XCS will be performed on it. If the input matches several conditions, one of the conditions will be selected at random. This function

returns the number of the line in the memory that contains the desired condition-action.
Now we will examine a specific case that may occur for the animat. For example, suppose the following position of the animat:

o o o o o o o
. ✷ . . . . o
o o o o o o o

If its move to the right is stored in the memory as the effective condition-action, in the next step the animat will again move to the right by checking the memory and matching the input against the existing condition. This action will be done until the animat has reached the rightmost square.

o o o o o o o
. . . . . ✷ o
o o o o o o o

In this step, as the input does not match any existing condition, the animat will use the default XCS and move to the left:

o o o o o o o
. . . . ✷ . o
o o o o o o o

In this step, the memory is searched to find a match for the input. Since the previous condition-action is found, the animat will once again move to the right. This indicates that the animat has fallen into a loop.

o o o o o o o
. . . . . ✷ o
o o o o o o o

For this reason, this memory will be used with a probability of 0.5. Thus the same conditions can be added to the memory with different actions.

3.4. Calculation of fitness

The initial value of this variable for each condition-action is set to 1. In each problem, if the food reward is achieved by a certain condition-action in the memory, the used parameter of that condition-action will be set to 1. In the end (i.e. either the maximum number of steps have been taken or the food has been reached), if the animat has achieved the food, 1 will be divided by all condition-actions that have been used in achieving the food and the obtained values will be multiplied by the previous fitness values. The following equation is used to calculate fitness:

$$Fitness += \left(\frac{1}{all \text{ used condition - actions}} \right) * Fitness \quad (1)$$

3.5. Adding condition-actions to memory of [P]

If the animat has achieved the food, all the condition-actions whose used value is 1 are added to the set [p] as a new classifier. First, Cover command is used to set some bits of the condition to # and, next, these new classifiers are added to [p]. Then if the number of the classifiers of [p] exceeds the maximum number defined, some of the classifiers are removed using an inherent strategy in XCS.

3.6. Using genetic algorithm for producing a new condition-action

If the size of c_a_memory exceeds a certain amount, the genetic algorithm is used to create new condition-actions. First, the entire memory content is sorted according to the fitness value in an ascending order. Next, a new c_a_memory half the size of the total memory is created. The genetic algorithm is executed on two parents with identical actions. 12 bits are taken from the first parent and 12 bits from the second parent to produce a child. The produced child is added to the c_a_memory and replaces the condition-action that has the smallest Fitness value. Also this new child is added to [p] as a new classifier after the Cover operation is performed on it.

3.7. Searching memory for input

The input from detector is first searched for in the memory. If the input matches one of the conditions in the memory, the corresponding action will be selected with a certain probability to be performed on the environment. If there are more than one condition matching the input, the condition with the highest Fitness will be selected. Finally, if there is not match, the conventional XCS method will be used for the next move.

4. Limited_memory

The innovation aspect of this research work compared with the basic XCS is summarized in the following three points. Firstly, in this research work, for the first time in the states of the learning classifier systems, limited memory has been used to store a set of states depending on the type of problem and the complexity of the system, and this shows the efficiency of the system in not using unlimited space. Secondly: overlap and cycle detection in repetitive pairs (condition-action) is obtained according to the input history from environmental sensors and the history of other previous actions using a non-Markov system. The results of the article confirm the improvement in the number of problems leading to the goal and reducing the number of steps to reach the final states in the maze maps problems. Thirdly: the simultaneous use of evolutionary algorithms such as genetic algorithm improves the development and generalizability of categories as well as the scalability of such problems. Limited_memory

stores the last n moves. In this memory, the conditions received by the sensors from the environment as well as their corresponding actions will be stored. Figure 5 illustrates a memory that contains the last 15 moves of an animat. Window is a small memory that, in each move, stores the last m actions of the animat. To identify the required action, the animat first searches for its window in the Limited_memory. If a specified number of bits from m conditions in window match the contents of the Limited_memory, the last action will be selected for the animat.

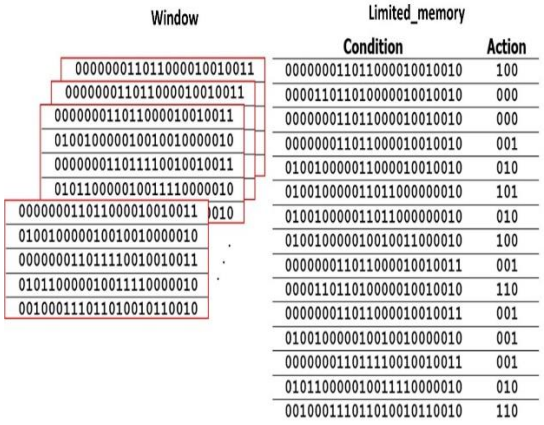


Figure 5. Limited_memory and window.

Before the motion begins, both Limited_memory and window are empty. As long as Limited_memory is smaller than twice the size of Window, no search is conducted. Therefore, the size of Limited_memory should be at least twice the size of Window, in which case only one comparison is made between the two memories.

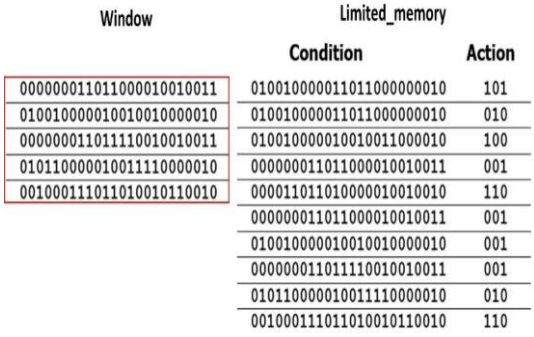


Figure 6. Minimum size of memories.

In Figure 6, the size of Limited_memory is 10 and the size of window is 5. According to the figure, when the memory size reaches 10, a comparison becomes possible between the two memories.

4.1. General flowchart of XCS and c_a_memory

Figure 7-a shows the flowchart of the proposed method, and Figure 7-b shows the corresponding algorithm. These figures explain one step of the animat. The input of this algorithm consists of c_a_memory, Limited_memory, and window. The output is the next move to be done in the environment. The command used to perform the appropriate action on the environment is Environment→perform (action). The functions used in the algorithm are described below. limited_memory.size () is used to calculate the size of limited_memory. window.size () is used to calculate the size of Window. c_a_memory.size () calculates the size of c_a_memory. Random () generates a random number between 0 and 1.

search_detector_c_a_memory () searches c_a_memory for the input from Detector.

insert_new_entry_c_a_memory() adds a condition-action line to c_a_memory. insert_classifier () adds a new classifier to [p]. As Limited_memory reaches a certain size, this memory is examined by window with a probability of 0.5. If window matches the contents of Limited_memory and the corresponding action is performed on the environment, the value of previous_search will be set to 1. Therefore, if the animat has selected an action in the previous step by means of Limited_memory, it is given a chance in the current step to search Limited_memory for its window memory (line 1).

All the bits of window are examined with all possible states in Limited_memory. In every state that matches more than 80% of the bits, the last action is performed on the environment and, after the execution of line 5 in the algorithm, one step of the animat is completed. In Figure 6, for example, there are six different states. In all cases where the existing condition does not hold, c_a_memory is used to find the most effective action. If the input is matched by one of the conditions in this memory and the obtained probability is less than 0.5, the action in the memory is performed on the environment (lines 8-10); otherwise, XCS will be used to determine the action and the function for finding the effective condition-action will be executed (lines 12-13). If the animat has not achieved the food, the next step will repeat the search for the input in the memory. However, if it has achieved the food and used, at least one of the condition-actions of the memory, the fitness of those condition-actions will be calculated. The condition-actions will be then added as new classifiers to the set [p] (lines 15-22). Finally, the algorithm checks whether the memory has exceeded a certain size X (line 23).

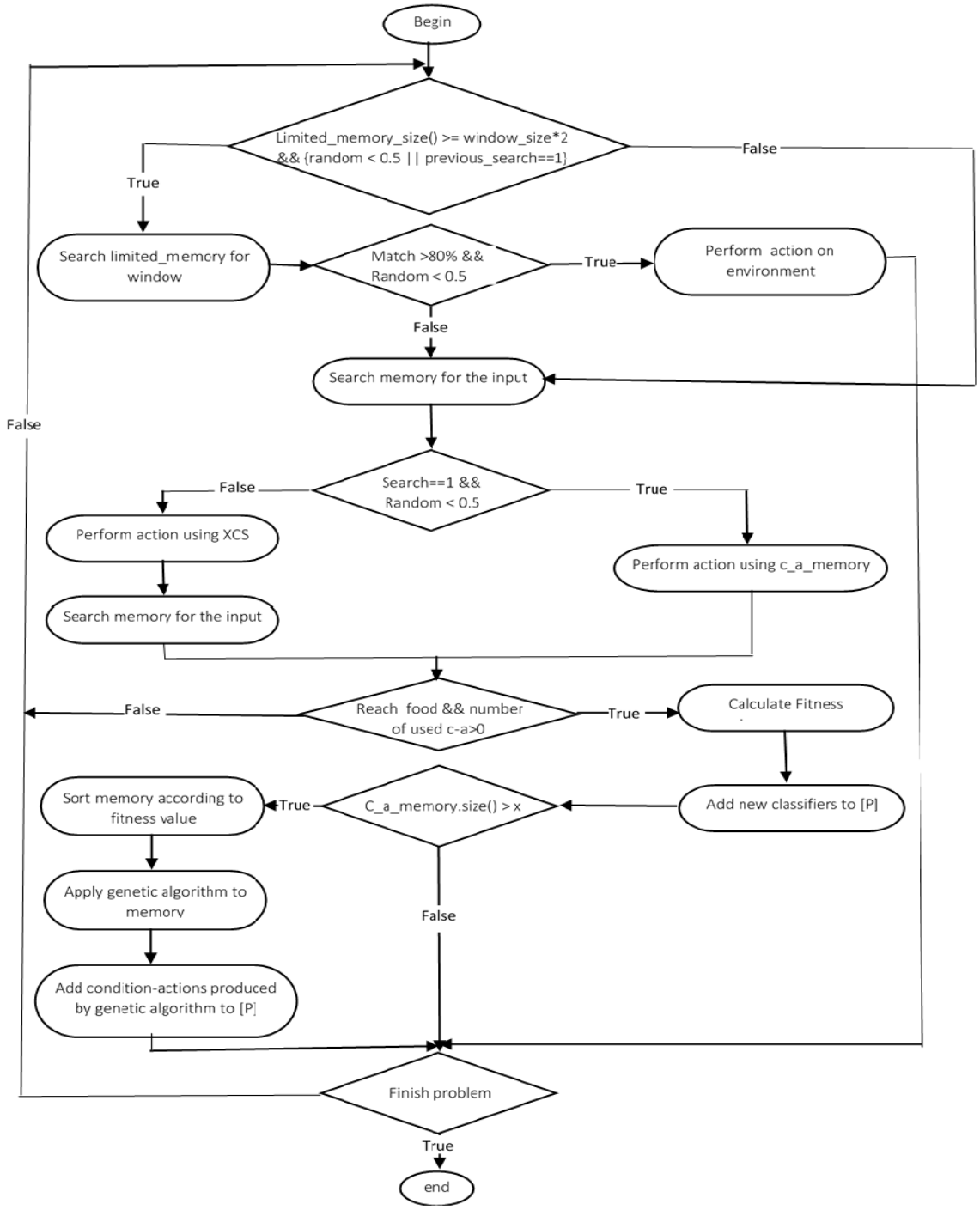


Figure7-a. Flowchart of the proposed method

The value of X in the proposed system is set to Maximum_number_of_ [p] * 0.03.

If this condition is true, the condition-actions in the memory will be sorted according to their Fitness value (line 24). Next, using genetic algorithm, a child is produced from two parents with identical actions and replaced by the condition-action that has the smallest Fitness value (line 25). Production of condition-actions continues for half of the memory size and the condition-actions are then added to [p] as

4. Implementation and Evaluation

This section reports on the implementation of the proposed method on the following maps. The evaluation criteria include the number of times the animat achieves the food in 10000 problems, the average number of steps in 10000 problems, and the average number of times the animat achieves the food in 10000 problems. The results are calculated as the average of 10 executions. The size of c_a_memory is 3% of the maximum number of the classifiers in [p]. Given that on these maps the maximum number of steps to

achieve the food is not large, memory size is considered as small as possible so that new classifiers could be more easily added to [p]. To evaluate method in different environments, the maps were selected based on the complexity of their environments. The complexity of the maps increases progressively.

```
        Input:      Limited_Memory
                    Window
                    c_a_memory

        Function    limited_memory.size() : size of limited_memory
                    window.size(): size of window
                    c_a_memory.size(): size of c_a_memory
                    random(): create a random between 0 to 1
                    search_detector_c_a_memory(): search
                    detector(input) in c_a_memory()
                    insert_new_entry_c_a_memory(): insert a new
                    entry in c_a_memory
                    insert_classifier(): insert a new classifier to [p]

        Output:     Next step of animat

        Begin
1               If limited_memory.size() >= window_size*2
                &&(random() > 0.5 || previous_search==1) do
2                   Max_matched= maximum number of bits of
                        entry which matched with detector
3                   If max_matched > 80% && random()<0.5 do
4                       Environment→perform(limited_memory
                        [max_offset].action)
5                       Goto Line 28
6                   End if
7               End if
8               search = search detector in c_a_memory()
9               If search  && random()<0.5 do
10                  Environment→perform(c_a_memory[search].
                    action)
11              Else
12                  Environment→perform(XCS.action)
13                  Insert XCS.action as new entry in c_a_memory
14              End If
15              If Environment->reward() == 1000 && used != 0
                do // reach food && number of used c-a >0
16                  For i=0;i<c_a_memory.size() do
17                      If c_a_memory[i].used == 1do
18                          c_a_memory[i].fitness +=
                            (c_a_memory[i].fitness * (1/used)) //
                             calculate Fitness
19                          insert_classifier(c_a_memory[i]) // add
                new classifiers to [p]
20                      End If
21                  End For
22              End If
23              If c_a_memory.size() > X do
24                  Sort c_a_memory
25                  GA
26                  insert n/2 of c_a_memory to [p]
27              End if
28      End
```

Figure 7-b. Pseudo-code for one step of the motion in the proposed method.

3.2. Benchmark problem 1 (Woods1: [p] =800)

Woods1 map is illustrated in Figure 8. In this map, the maximum number of classifiers in [p] is 800.

Also the maximum number of steps to reach the food is 5. The size of Limited_memory is 4 and the size of Window is 2. Figure 9 illustrates the results of the implementation of the proposed method as well as XCS.

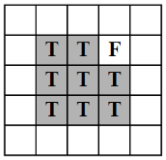
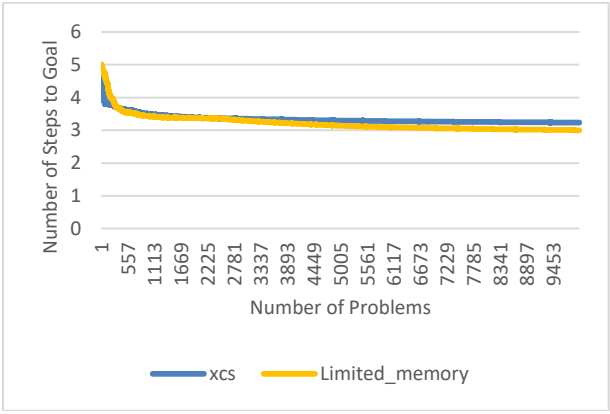
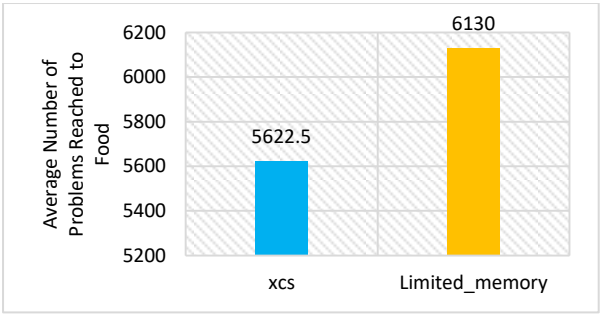


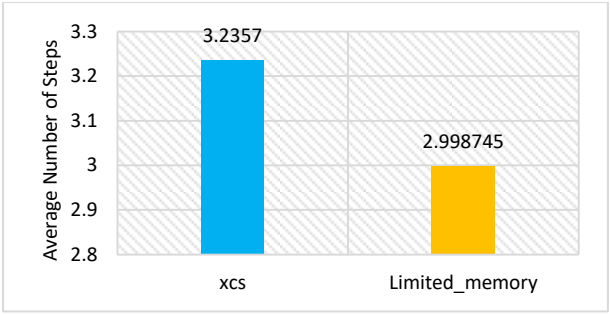
Figure 8. Woods1 map.



a) Number of steps required to achieve the food on Woods1.



b) Average number of problems ending in food on Woods1.



c) Average number of the animat's steps in 10000 problems on Woods1.

Figure 9. Results of implementation on Woods1.

3.3. Benchmark problem 2 (Maze7: [p] =1600)

In this map, the maximum number of classifiers in [p] is 1600. Also the maximum number of steps to reach the food is 20. The size of Limited_memory is 15 and the size of Window is 5. Figure 10 shows Maze7 map, and Figure 11 shows the

results of the implementation of the method in 10000 problems.

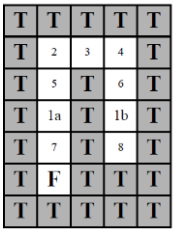
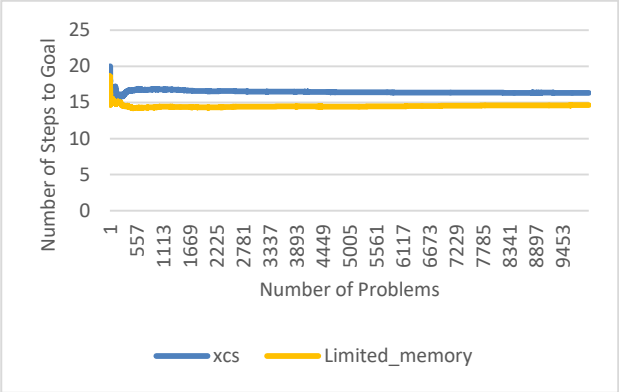
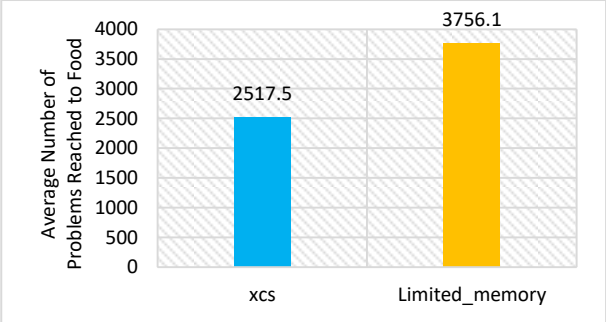


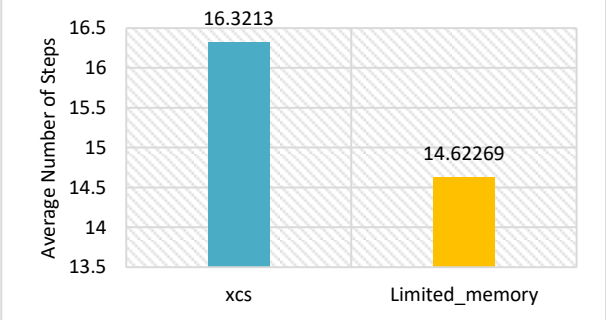
Figure 10. Maze7 map.



a) Number of steps required to achieve the food on Maze7.



b) Average number of problems ending in food on Maze7.



c) Average number of the animat's steps in 10000 problems on Maze7.

Figure 11. Results of implementation on Maze7.

3.4. Benchmark problem 3(MazeF4: [p] =1600)

In this map, which is shown in Figure 12, the maximum number of classifiers in [p] is 1600. Also the maximum number of steps to reach the food is 20. The size of Limited_memory is 15 and the size of Window is 5. The plots in Figure 13 show the results of the implementation of the proposed method on MazeF4.

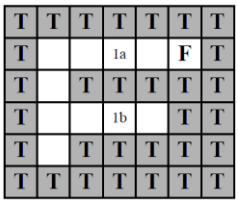
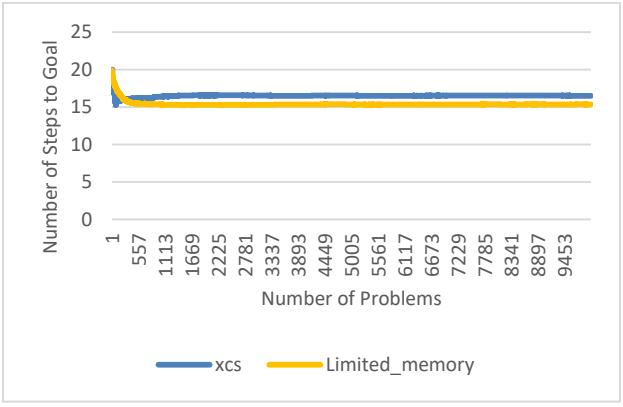
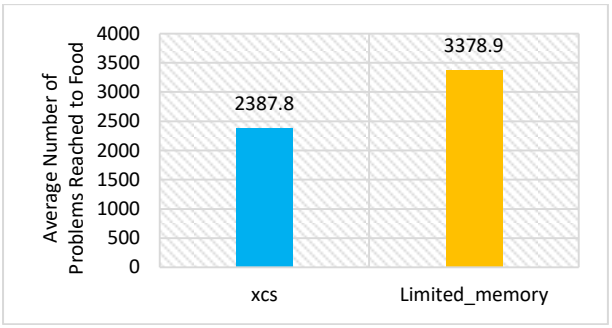


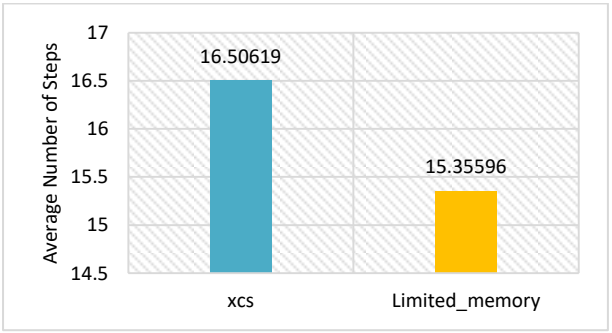
Figure 12. MazeF4 map.



a) Number of steps required to achieve the food on MazeF4.



b) Average number of problems ending in food on MazeF4.



c) Average number of the animat's steps in 10000 problems on MazeF4.

Figure 13. Results of implementation on MazeF4.

3.5. Benchmark problem 4 (Maze10: [p] =2800)

In this map, the maximum number of classifiers in [p] is 2800. Figure 14 shows Maze10 map. Also the maximum number of steps to reach the food is 20. The size of Limited_memory is 15 and the size of Window is 5. The plots in Figure 15 show

the results of the implementation of the proposed method on Maze10.

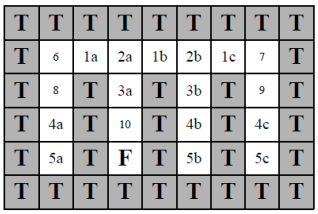
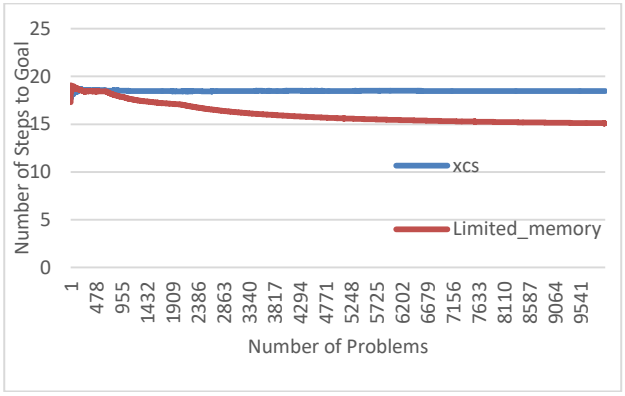
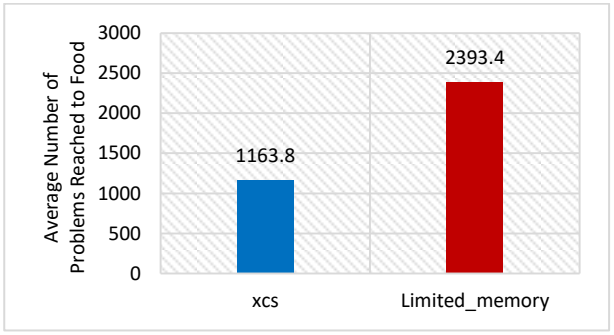


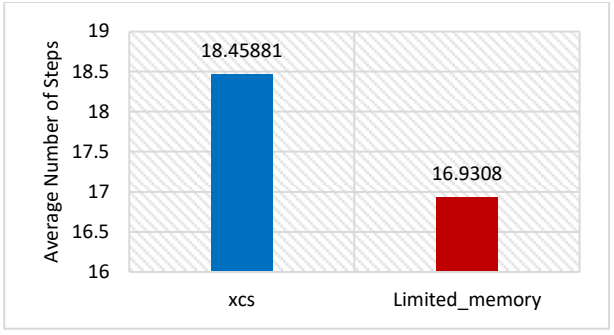
Figure 14. Maze10 map.



a) Number of steps required to achieve the food on Maze10.



b) Average number of problems ending in food on Maze10.



c) Average number of the animat's steps in 10000 problems on Maze10.

Figure 15. Results of implementation on Maze10.

3.6. Benchmark problem 5 (Woods102: [p]=2800)

In this map, the maximum number of classifiers in [p] is 2800. Also the maximum number of steps to reach the food is 20. The size of Limited_memory is 15 and the size of Window is 5. Figure 16 shows Woods102 map. The plots in Figure 17 show the results of the implementation of the proposed method as well as XCS on Woods102.

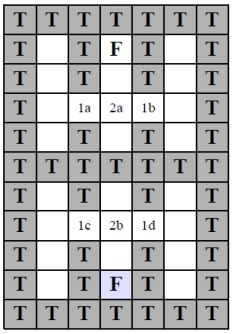
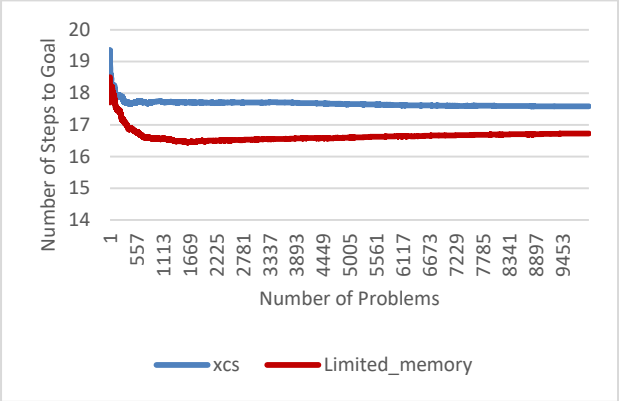
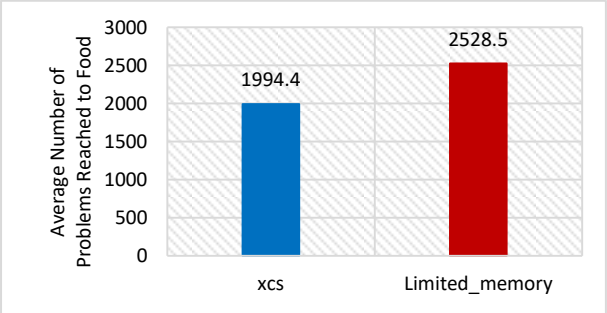


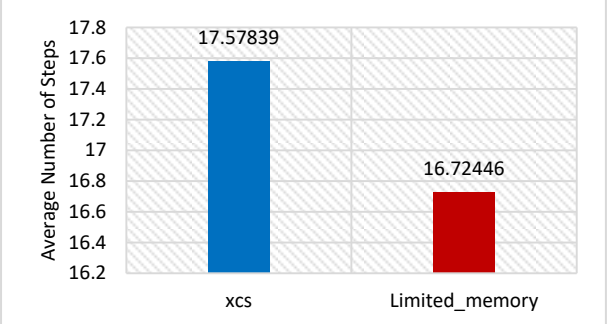
Figure 16. Woods102 map.



a) Number of steps required to achieve the food on Woods102.



b) Average number of problems ending in food on Woods102.



c) Average number of the animat's steps in 10000 problems on Woods102.

Figure 17. Results of implementation on Woods102.

According to the results, increase in the number of problems would decrease the number of steps

required by the animat to achieve the food. The reason is that by applying genetic algorithm to the memory in each problem, its c_a_memory has improved and the classifiers can act more efficiently on the next steps. In Maze10 map in Figure 17-a, for example, increasing the number of problems has not caused XCS algorithm to show any remarkable change in the number of steps to achieve the food. Instead, in the proposed method, increasing the number of problems has significantly decreased the steps to achieve the food. On the other hand, the proposed method requires fewer steps than XCS to achieve the food. In Woods102, for example, the average number of steps is 16.7 in the proposed method while it is 17.6 in XCS.

Another criterion that is of great importance in maze problems is the average number of problems in which the animat has been able to achieve the food. Our implementation results suggest that, in all the maps, the proposed method has made the animat achieve the food in more problems. In Woods1, for example, XCS has achieved the food only in 5622 problems out of 10000 problems while the proposed method has achieved the food in 6130 problems. This difference is observed in all the maps.

The last criterion is the average number of steps that an animat takes to achieve the food. Here, too, the proposed method of memory usage has been able to reduce the number of the animat's steps. In Maze10, the results of which are shown in Figure 17-c, the average number of steps are 16.9 in the proposed method and 18.45 in XCS. In other words, our method has decreased the number of steps by 8.4%.

The plots describing the number of steps for achieving the food have fluctuations over the 10000 problems. These fluctuations can be partly explained by the fact that the beginning point of the animat and the initial classifiers in [p] were selected randomly. The animat's beginning point could be any empty point on the map. Therefore, the initial distance of the animat from the food varied from one problem to another. On the other hand, the set [p] was initially empty and, based on the functioning of XCS, a number of random classifiers were created using Cover operation. Thus the classifiers may have been different in the animat's motion towards the food.

Concerning the forward motion of the animat as opposed to its remaining in place or being stopped by obstacles, the proposed method has improved the efficiency, speed, and performance of the animat. Given the limitations set for the animat, it can take a certain number of steps to achieve the food. Therefore, if the steps do not include being stopped by obstacles, the animat can traverse a longer path in search of food, which in turn increases the time of achieving food.

4. Conclusion

Achieving the goal in real environments such as winding problems faces a challenge due to different degrees of overlap in the movement path and the lack of sensory inputs in some cases. To solve such a challenge, in this article, a new memory-based XCS algorithm was used, which acted as a non-Markovian system and was able to identify and maintain optimal rules in overlapping states. In this algorithm, a limited chain of final system states that has led to success in previous experiments is also maintained by relying on limited memory. This algorithm was implemented on 5 famous maze problems. The increase in the number of problems leading to the goal and the decrease in the number of steps and steps to reach the goal in this new algorithm compared to the basic XCS algorithm indicate the high efficiency of the new method. The limitation of the amount of memory that can be used in real learning classifier systems in solving complex problems is the dependence of the memory model used on the type of application of learning classifier systems in solving various problems, among the limitations raised in this article. Considering the static nature of the problem environment for future work, the readers are suggested to make the winding problem environment dynamic in order to increase the flexibility and generalizability of the problem by designing more optimal XCS algorithms based on memory.

References

[1] Lanzi, Pier L. "Learning classifier systems: from foundations to applications.", No. 1813. Springer Science & Business Media, 2000.

[2] J. Holland, L. Booker, M. Colombetti, M. Dorigo, D. Goldberg, S. Forrest *et al.*, "What Is a Learning Classifier System?," *In Learning Classifier Systems*. vol. 1813, Springer Berlin Heidelberg, pp. 3-32, 2000.

[3] S. W. Wilson, "Classifier fitness based on accuracy," *Evol. Comput.*, vol. 3, pp. 149-175, 1995.

[4] Bernadó-Mansilla, Ester, and Josep M. Garrell-Guiu. "Accuracy-based learning classifier systems: models, analysis and applications to classification tasks." *Evolutionary computation*, vol .11, no. 3 pp. 209-238, 2003.

[5] J. H. Holmes, P. L. Lanzi, W. Stolzmann, and S. W. Wilson, "Learning classifier systems: New models, successful applications," *Information Processing Letters*, vol. 82, pp. 23-30, 2002.

[6] M. Shariat Panahi, A. Karkhaneh Yousefi, and M. Khorshidi, "Combining accuracy and success-rate to improve the performance of eXtended Classifier System (XCS) for data-mining and control applications," *Engineering Applications of Artificial Intelligence*, vol. 26, pp. 1930-1935, 2013.

[7] Irfan, Muhammad *et al*. "Enhancing learning classifier systems through convolutional autoencoder to classify underwater images." *Soft Computing* , vol . 25, no . 15, pp. 10423-10440, 2021.

[8] Irfan, Muhammad *et al*. "Knowledge extraction and retention based continual learning by using convolutional autoencoder-based learning classifier system." *Information Sciences* 591, pp. 287-305, 2022

[9] Kato, Jefferson Satoshi, and Adriana Sbicca. "Bounded Rationality, Group Formation and the Emergence of Trust: An Agent-Based Economic Model. " *Computational Economics*, pp. 1-29, 2021.

[10] Liu, Yi. *Learning Classifier Systems for Understanding Patterns in Data*. Diss. Open Access Te Herenga Waka-Victoria University of Wellington, 2021.

[11] Hansmeier, Tim, and Marco Platzner. "An experimental comparison of explore/exploit strategies for the learning classifier system XCS." *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 2021.

[12] Guendouzi, Wassila, and Abdelmadjid Boukra. "A new differential evolution algorithm for cooperative fuzzy rule mining: application to anomaly detection. " *Evolutionary Intelligence*, pp. 1-12, 2021.

[13] Hochberger, Christian, Lars Bauer, and Thilo Pionteck. *Architecture of Computing Systems*. Springer International Publishing, 2021.

[14] Büttner, Johannes, and Sebastian Von Mammen. "Training a Reinforcement Learning Agent based on XCS in a Competitive Snake Environment." *2021 IEEE Conference on Games (CoG)*. IEEE, 2021.

[15] B. Bartin, "Use of learning classifier systems in microscopic toll plaza simulation models," *IET Intelligent Transport Systems*, vol. 13, pp. 860-869, 2019.

[16] M. R. Karlsen and S. Moschoyiannis, "Evolution of control with learning classifier systems," *Applied network science*, vol. 3, pp. 1-36, 2018.

[17] M. Butz and D. Goldberg, "Generalized State Values in an Anticipatory Learning Classifier System," in *Anticipatory Behavior in Adaptive Learning Systems*. vol. 2684, M. Butz, O. Sigaud, and P. Gérard, Eds., ed: Springer Berlin Heidelberg, pp. 282-301, 2003.

[18] M. V. Butz, T. Kovacs, P. L. Lanzi, and S. W. Wilson, "Toward a theory of generalization and learning in XCS," *Evolutionary Computation, IEEE Transactions on*, vol. 8, pp. 28-46, 2004.

[19] P. Gérard and O. Sigaud, "YACS: Combining Dynamic Programming with Generalization in Classifier Systems," in *Advances in Learning Classifier Systems*. vol. 1996, P. Luca Lanzi, W. Stolzmann, and S. Wilson, Eds., ed: Springer Berlin Heidelberg, pp. 52-69, 2001.

[20] J. H. Holland, "Escaping brittleness: the possibilities of general-purpose learning algorithms applied to parallel rule-based systems," in *Computation & intelligence*, F. L. George, Ed., ed: American Association for Artificial Intelligence, pp. 275-304, 1995.

[21] P. L. Lanzi, "An analysis of generalization in the xcs classifier system," *Evol. Comput.*, vol. 7, pp. 125-149, 1999.

[22] P. L. Lanzi, D. Loiacono, S. W. Wilson, and D. E. Goldberg, "Generalization in the XCSF Classifier System: Analysis, Improvement, and Extension," *Evol. Comput.*, vol. 15, pp. 133-168, 2007.

[23] M. Iqbal, W. Browne, and M. Zhang, "XCSR with Computed Continuous Action," in *AI 2012: Advances in Artificial Intelligence*. vol. 7691, M. Thielscher and D. Zhang, Eds., ed: Springer Berlin Heidelberg, pp. 350-361, 2012.

[24] M. Iqbal, W. N. Browne, and Z. Mengjie, "Reusing Building Blocks of Extracted Knowledge to Solve Complex, Large-Scale Boolean Problems," *Evolutionary Computation, IEEE Transactions on*, vol. 18, pp. 465-480, 2013.

[25] F. Freschi and M. Repetto, "Multi-objective optimization by a modified artificial immune system algorithm," *presented at the Proceedings of the 4th international conference on Artificial Immune Systems, Banff, Alberta, Canada*, 2005.

[26] H. Asadul Rehman, M. Iqbal, I. Younas, and M. Bashir, "Learning Regular Expressions using XCS-based Classifier System," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 34, no. 10, p. 2051011, 2019.

[27] J. Khan, A. Alam, J. Hussain, and Y.-K. Lee, "EnSWF: effective features extraction and selection in conjunction with ensemble learning methods for document sentiment classification," *Applied Intelligence*, vol. 49, pp. 3123-3145, Aug 2019.

[28] K. Shafi and H. A. Abbass, "A survey of learning classifier systems in games," *IEEE Computational Intelligence Magazine*, vol. 12, pp. 42-55, 2017.

[29] I. M. Alvarez, W. N. Browne, and M. Zhang, "Human-inspired scaling in learning classifier systems: Case study on the n-bit multiplexer problem set," in *Proceedings of the Genetic and Evolutionary Computation Conference* , pp. 429-436, 2016.

[30] M. Tubishat, M. A. M. Abushariah, N. Idris, and I. Aljarah, "Improved whale optimization algorithm for feature selection in Arabic sentiment analysis," *Applied Intelligence*, vol. 49, pp. 1688-1707, May 2019.

[31] Z. Zang, D. Li, and J. Wang, "Learning classifier systems with memory condition to solve non-Markov problems," *Soft Computing*, vol. 19, pp. 1679-1699, June 2015.

[32] A. L. Thomaz and C. Breazeal, "Teachable robots: Understanding human teaching behavior to build more effective robot learners," *Artificial Intelligence*, vol. 172, pp. 716-737, 2008.

[33] L. M. Saksida, S. M. Raymond, and D. S. Touretzky, "Shaping robot behavior using principles from instrumental conditioning," *Robotics and Autonomous Systems*, vol. 22, pp. 231-249, 1997.

[34] M. Dorigo and M. Colombetti, "Robot shaping: developing autonomous agents through learning," *Artificial Intelligence*, vol. 71, pp. 321-370, 1994.

[35] D. Cliff and S. Ross, "Adding temporary memory to ZCS," *Adapt. Behav.*, Vol. 3, pp. 101-150, 1994.

[36] S. Wilson, "Classifier systems and the animat problem," *Machine Learning*, vol. 2, pp. 199-228, Nov 1987.

[37] G. E. P. Box, "Evolutionary operation: a method for increasing industrial productivity," *Applied statistics : a journal of the Royal Statistical Society*, vol. 6, pp. 81-101, 1957.

[38] L. B. Booker, "Intelligent behavior as an adaptation to the task environment," University of Michigan, 1982.

[39] L. B. Booker, "Improving the Performance of Genetic Algorithms in Classifier Systems," presented at the Proceedings of the 1st International Conference on Genetic Algorithms, 1985.

[40] L. B. Booker, "Classifier systems that learn internal world models,"*Mach. Lang.*, vol. 3, pp. 161-192, 1988.

[41] L. B. Booker, "Triggered Rule Discovery in Classifier Systems," presented at the Proceedings of the 3rd International Conference on Genetic Algorithms, 1989.

[42] S. W. Wilson, "Zcs: A zeroth level classifier system," *Evol. Comput.*, vol. 2, pp. 1-18, 1994.

[43] R. S. Sutton and A. G. Barto, "Toward a modern theory of adaptive networks: expectation and prediction," *Psychol Rev*, vol. 88, pp. 135-70, 1981.

[44] S. W. Wilson, "Classifiers that approximate functions," vol. 1, pp. 211-234, 2002.

[45] L. Bull, "Two Simple Learning Classifier Systems," in *Foundations of Learning Classifier Systems*. Vol. 183, L. Bull and T. Kovacs, Eds., ed: Springer Berlin Heidelberg, pp. 63-89, 2005.

[46] L. Bull, "A brief history of learning classifier systems: from CS-1 to XCS and its variants," *Evolutionary Intelligence*, pp. 1-16, Jan 2015.

[47] A. Hamzeh, S. Hashemi, A. Sami, and A. Rahmani, "A Recursive Classifier System for Partially Observable Environments," *Fundam. Inform.*, vol. 97, pp. 15-40, 2009.

[48] LaterA. Hamzeh and A. Rahmani, "A New Architecture for Learning Classifier Systems to Solve POMDP Problems," *Fundam. Inform.*, vol. 84, pp. 329-351, 2008.

[49] R. Preen and L. Bull, "Discrete and fuzzy dynamical genetic programming in the XCSF learning classifier system," *Soft Computing*, vol. 18, pp. 153-167, Jan 2014.

[50] T. Ke, L. Jing, H. Lv, L. Zhang, and Y. Hu, "Global and local learning from positive and unlabeled examples,"*Applied Intelligence*, vol. 48, pp. 2373-2392, August 2018.

[51] L. Lu, Q. Lin, H. Pei, and P. Zhong, "The aLS-SVM based multi-task learning classifiers,"*Applied Intelligence*, vol. 48, pp. 2393-2407, August 2018.

[52] I. M. Alvarez, W. N. Browne, and M. Zhang, "Compaction for Code Fragment Based Learning Classifier Systems," Cham, pp. 41-53, 2016.

[53] M. Moradi and J. Hamidzadeh, "Ensemble-based Top-k Recommender System Considering Incomplete Data," *Journal of AI and Data Mining*, vol. 7, no. 3, pp. 393-402, 2019.

یک الگوریتم کارآمد مبتنی بر XCS برای سیستم‌های طبقه بند یادگیر در محیط های واقعی

علی یوسفی[۱]، کامبیز بدیع[۲،*]، محمدمهدی عبادزاده[۳] و آرش شریفی[۴]

[۱] گروه مهندسی کامپیوتر، دانشکده مکانیک، برق و کامپیوتر، واحد علوم و تحقیقات، دانشگاه آزاد اسلامی، تهران، ایران.

[۲] پژوهشکده فناوری اطلاعات، پژوهشگاه ارتباطات و فناوری اطلاعات، تهران، ایران.

[۳] دانشگاه صنعتی امیرکبیر، تهران، ایران.

[۴] دانشکده مکانیک، برق و کامپیوتر، واحد علوم و تحقیقات، دانشگاه آزاد اسلامی، تهران، ایران.

ارسال ۲۰۲۲/۱۱/۰۱؛ بازنگری ۲۰۲۲/۱۱/۲۵؛ پذیرش ۲۰۲۲/۱۲/۲۴

چکیده:

اخیراً از سیستم‌های طبقه بند یادگیر برای کنترل ربات‌های فیزیکی، ربات‌های حسی و سیستم‌های نجات هوشمند استفاده می‌شود. مهم‌ترین چالش در این سیستم‌ها که مدل‌هایی از محیط های واقعی هستند، کیفیت غیر مارکوفی آن است. بنابراین لازم است از حافظه برای ذخیره حالت‌های سیستم استفاده شود تا بتوان بر اساس زنجیره‌ای از حالت‌های قبلی تصمیم‌گیری کرد. در این مقاله، یک XCS مبتنی بر حافظه برای کمک به استفاده از قوانین مؤثرتر در طبقه‌بندی با شناسایی قوانین کارآمد پیشنهاد شده‌است. مدل پیشنهادی بر روی پنج نقشه مهم ماز پیاده‌سازی شده و منجر به کاهش تعداد مراحل رسیدن به هدف و همچنین افزایش تعداد موفقیت‌های رسیدن به هدف در این نقشه‌ها می‌شود.

کلمات کلیدی: سیستم‌های طبقه بند یادگیر، الگوریتم XCS، شناسایی چرخه و همپوشانی.