THREE-CORNERED COEVOLUTION LEARNING CLASSIFIER SYSTEMS FOR CLASSIFICATION

by

Syahaneim Marzukhi

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy
in Computer Science.

Victoria University of Wellington
2014

Abstract

This thesis introduces a *Three-Cornered Coevolution System* that is capable of addressing classification tasks through coevolution (coadaptive evolution) where three different agents (i.e. a generation agent and two classification agents) learn and adapt to the changes of the problems without human involvement.

In existing pattern classification systems, humans usually play a major role in creating and controlling the problem domain. In particular, humans set up and tune the problem's difficulty. A motivation of the work for this thesis is to design and develop an *automatic pattern generation and classification system* that can generate various sets of exemplars to be learned from and perform the classification tasks autonomously. The system should be able to automatically adjust the problem's difficulty based on the learners' ability to learn (e.g. determining features in the problem that affect the learners' performance in order to generate various problems for classification at different levels of difficulty). Further, the system should be capable of addressing the classification tasks through coevolution (coadaptive evolution), where the participating agents learn and adapt to the changes of the problems without human participation. Ultimately, Learning Classifier System (LCS) is chosen to be implemented in the participating agents. LCS has several potential characteristics, such as interpretability, generalisation capability and variations in representation, that are suitable for the system.

The work can be broken down into three main phases. Phase 1 is to develop an *automated evolvable problem generator* to autonomously generate various problems for classification, Phase 2 is to develop the *Two-Cornered*

Coevolution System for classification, and Phase 3 is to develop the *Three-Cornered Coevolution System* for classification.

Phase 1 is necessary in order to create a set of problem domains for classification (i.e. image-based data or artificial data) that can be generated automatically, where the difficulty levels of the problem can be adjusted and tuned.

Phase 2 is needed to investigate the generation agent's ability to autonomously tune and adjust the problem's difficulty based on the classification agent's performance. Phase 2 is a standard coevolution system, where two different agents evolve to adapt to the changes of the problem. The classification agent evolves to learn various classification problems, while the generation agent evolves to tune and adjust the problem's difficulty based on the learner's ability to learn.

Phase 3 is the final research goal. This phase develops a new coevolution system where three different agents evolve to adapt to the changes of the problem. Both of the classification agents evolve to learn various classification problems, while the generation agent evolves to tune and adjust the problem's difficulty based on the classification agents' ability to learn. The classification agents use different styles of learning techniques (i.e. supervised or reinforcement learning techniques) to learn the problems. Based on the classification agents' ability (i.e. the difference in performance between the classification agents) the generation agent adjusts and creates various problems for classification at different levels of difficulty (i.e. various 'hard' problems).

The Three-Cornered Coevolution System offers a great potential for autonomous learning and provides useful insight into coevolution learning over the standard studies of pattern recognition. The system is capable of autonomously generating various problems, learning and providing insight into each learning system's ability by determining the problem domains where they perform relatively well. This is in contrast to humans having to determine the problem domains.

Acknowledgments

All praise is due to Allah, the Almighty God, the most Beneficent and the most Merciful, I am forever grateful for His blessings by giving me the strength upon completing this thesis.

Also, my sincerest gratitude goes to

My supervisors for the opportunity to explore this research area and work under their kind supervisions by providing technical guidance, useful discussions and suggestions, reviewing our joint publications, and all the needed support, upon the completion of this thesis;

My beloved husband and the two little ones for their patience and sacrifice in supporting me to undergo this challenging journey and these meaningful years;

My parents for their prayers and support, especially to my mother for her encouragement to help me achieve the highest level of education;

My workmates and friends for sharing their time, thoughtful lunches and joyful moments together;

Several researchers in the LCSs field, especially those who patiently reviewed my work by giving useful feedback and comments, and other LCSs experts for sharing their knowledge and experience;

The members of the faculty and administration staff who provided all the required facilities and needs during my studies here,

and last, but not least, Ministry of Education, Malaysia for awarding me the postgraduate research scholarship and National Defence University Malaysia for the opportunity to further my study abroad.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Scope

Pattern recognition is concerned with the capability of machines to observe the environment, learn to distinguish patterns of interest from their background and make a reasonable decision about the categories of the patterns [20]. Here, the recognition problem is a *classification* or categorization task. Automatic machine recognition, description, classification, and grouping of patterns are important problems in a variety of engineering and scientific disciplines such as biology, psychology, medicine, marketing, computer vision, artificial intelligence, and remote sensing [20]. Recently demands on automatic pattern recognition systems have increased due to rapidly growing and available computing power, which has enabled the production of various datasets in large volumes and faster processing of these large datasets.

Learning from a set of examples that has desired features is crucial and important for most pattern recognition systems. Traditionally, research in pattern recognition involves choosing a domain, creating a source of exemplars, and trying out learning algorithms that seem likely to work in that domain [124]. Previous studies in [73, 39] have shown that a learner's performance (algorithm) depends on either the constraint of the method

or the complexity of the data. Consequently, a public repository (e.g. the UCI repository [5]), has become the most used method in attempting to find a suitable domain. However, relying on the available data in the public repository has the following disadvantages [73]. First, if one hundred percent performance is not reached, the learner's true capability in the domain is unknown, e.g. whether it has achieved the maximum performance or if there is still a room for performance improvements. Secondly, sometimes the critique of the learner's performance is misleading, for example when the characteristics of the problem under study are unknown. Thirdly, identifying these problem characteristics is not easy as there are many factors affecting the problem's difficulty in the real-world problems, such as inconsistency, uncertainty, missing values, and sampling sparsity.

One way of creating the source of exemplars is guided by humans' expert knowledge. However, humans might make a mistake in classification and fail to produce thousands or millions of exemplars within a limited amount of time. Again, identifying the problem's characteristics are not easy as there are many factors affecting the problem's difficulty in the real-world problems. Thus, humans might produce incomplete and noisy exemplars. Nevertheless, the process of creating the exemplars based on humans' expert knowledge incurs another additional cost, i.e. labour cost. In order to learn from the available exemplars, it is very important to produce a reliable source of examples that can easily be extracted and learned from.

In this case, an *automatic pattern generation system* would be valuable to provide the following advantages over the traditional methods: 1) generation of a large set of examples automatically (i.e. a library of problems), 2) capability of producing examples of each class that are diverse and numerous, 3) manipulation of the states related to a particular class through flexible methods, 4) encoding deterministic characteristics into the data, and 5) less human involvement, which can reduce overall cost.

Further, if the pattern generation system can link with a pattern classi-

fication system, then the learner's capability in the domain can be further investigated systematically. Here, the classification system learns the generated examples or problems, whereas the pattern generation system discovers the types of the problem that fall in the domain of the classification system's competence automatically.

1.2 Motivation

Although, *autonomous learning of patterns* by machines has advanced recently, it still requires humans to set up the problem at an appropriate level of complexity for the learning technique to learn. In existing artificial classification systems, the problem domain is created and controlled by humans. If the problem is too complex the system does not learn. Conversely, if the problem is too simple, the system does not reach its full potential. If humans can set up the problem appropriately then machines can extract beneficial knowledge to solve classification tasks. Furthermore, the limit of the learner's or technique's capability can be assessed. Therefore, the challenge is to develop a computer based program, i.e. an *autonomous pattern generation and classification system* that learns to identify whether or not a pattern belongs to a specific class by producing various examples to be learned from autonomously.

Furthermore, the creation of an *automated pattern classification system* is itself a considerably complex problem. Firstly, machines usually have to recognize a pattern characteristic from a large search space and a huge number of examples (or datasets) to build a reliable recognition system. The amount of irrelevant and redundant data might be too large compared to the amount of knowledge available (i.e. important and interesting information) that needs to be learnt. Secondly, machines often perceive incomplete pattern characteristics from the repository as there is inconsistency, uncertainty, missing values, and sampling sparsity, in order to build a formative knowledge representation. Finally, there is no clear guidance for

the selection of a classification system to address classification problems effectively where the cause of the problem's difficulty is unknown.

The main advantage of the automated pattern generation and classification system is to be able to generate a set of more comprehensive exemplars than humans can generate and learn useful knowledge to capture all the pattern characteristics accurately, showing the capability of the system.

There are a number of machine learning methods that are available to improve the design and development of existing pattern recognition systems. A class of computer algorithms called evolutionary computation (EC) can be used. For example, Genetic Algorithm (GA) is used to explore the search space in order to find an interesting problem in the domain (e.g. pattern). Once these problems are found, a set of exemplars is produced. Thus, GA could enable the pattern generation system to automatically create the exemplars associated with the problems in less time than a human. Millions of exemplars can be produced and created under various circumstances. Next, GA can also be used to learn the set of exemplars for classification, where the classification system's performance can be further tested and investigated.

Learning Classifier Systems (LCSs) appear to be a widely applicable agents' model that can provide a framework for a diversity of learning and practical applications [116]. Hence, in this work for the thesis, LCSs are mainly applied as the learning systems. Here, LCSs have been adopted in the learning system based on the potential characteristics, such as interpretability, generalisation capability and variations in representation.

Recently, a theoretical framework of *Three-Cornered Coevolution* [124] was proposed by Wilson in the paper 'Coevolution of Pattern Generators and Recognizers'. The author proposed an automatic system for image transformation with a pattern generator and pattern recognizers. This is to be a human-independent system that may provide a new insight into the pattern recognition problem. However, this theoretical framework had not yet been implemented and tested. It is therefore unclear how well

the framework will work to drive the coevolution within the system (e.g. how the coevolutionary process will actually work between the pattern generator and the pattern recognizers).

Therefore, the *Three-Cornered Coevolution System* will be implemented in order to automate the process of pattern classification, with the aim to autonomously create various problems for classification tasks at different levels of complexity. The *Three-Cornered Coevolution System* will be based on the Three-Cornered Coevolution Framework as suggested by Wilson through implementing LCSs as the participating agents within the system.

1.3 Thesis Statement

The thesis makes the following statement:
The Three-Cornered Coevolution System that consists of three main agents (i.e. the generation agent and two classification agents) is capable of addressing the classification tasks through coevolution (coadaptive evolution) where three different agents learn and adapt to the changes of the problems without human involvement.

The potential benefits of utilising the Three-Cornered Coevolution System are that the system is able to: 1) determine the learner's capability in the domain, 2) determine the features in the problem that effect the problem's difficulty in order to explore the methods of the learners, 3) maximize the difficulty levels of the problem in relation to the features in the problem in order to investigate the learner's capability within certain feature domains.

1.4 Research Goals

Based on Wilson's proposal, the overall goal of the work is to develop and implement the novel *Three-Cornered Coevolution System* for addressing

classification tasks for the first time. The main focus is to develop a system that autonomously creates various problems for the classification task at different levels of complexity, where the problem's difficulty can be adjusted and tuned, based on the agent's ability to learn. Ultimately, LCSs are implemented in the participating agents to evolve their rules and improve their performance.

Given a variety of patterns where the agents need to perform a classification task, the work has to find possible answers to the following research question:

How can the pattern classification tasks be addressed effectively using a Three-Cornered Coevolution System that automatically adjusts the problem's difficulty based on the agents' ability to learn?

In order to execute the overall goal above, a set of research objectives have been established to guide this research.

1. Develop an *automated evolvable problem generator* for generating different types of problems for classification. Here, two new problem domains will be created and can be manipulated autonomously (i.e. scalable and evolvable image-based data, and artificial data). This system is needed in order to establish an appropriate problem domain for the classification tasks that can be evolved and tuned automatically.

2. Develop a new *Two-Cornered Coevolution System* for addressing the classification tasks that consists of two main agents (i.e. the generation agent and the classification agent). The classification agent evolves to learn various problems, while the generation agent evolves to tune and adjust the problem's difficulty based on the classification agent's ability to learn. This system is needed to investigate the coevolutionary approach between the participating agents within the

system in order to establish a framework for the Three-Cornered Coevolution System.

3. Develop a new *Three-Cornered Coevolution System* that consists of three main agents (i.e. the generation agent and two classification agents) for addressing the classification tasks. Here, two classification agents learn and adapt to the changes of the problems using different types of learning technique (i.e. supervised learning or reinforcement learning). Based on the classification agents' ability (i.e. the difference in performance between the classification agents) the generation agent adjusts and creates various problems for classification at different levels of difficulty (i.e. various 'hard' problems).

1.5 Major Contributions

The thesis has made the following major contributions to pattern recognition and classification utilising Learning Classifier Systems (LCSs):

1. The thesis has introduced an *automated evolvable problem generator* that can create various problems for the classification tasks; a new human-independent system for the creation of various classification problems. A novel set of problem domains has been created which can be manipulated autonomously to generate scalable and evolvable problems (i.e. image-based data or artificial data). In addition, the created problem domain can be tuned at various levels of difficulty, such that various exemplars at different levels of difficulty for the classification tasks can be generated autonomously. All of the generated problems provide a convenient way to help in empirically testing the learning bounds of the learners (algorithms).

   Part of this contribution has been published in:

Syahaneim Marzukhi, Will N. Browne and Mengjie Zhang, "Developing An Evolvable Pattern Generator Using Learning Classifier Systems", Proceedings of the 5th International Conference on Automation, Robotics and Applications (ICARA 2011), pages 163-168, IEEE Xplore.

2. The thesis has introduced a new technique for addressing the classification tasks using the *Two-Cornered Coevolution System*. This is the first effort to develop a new human-independent adjustable system for the classification tasks using LCSs. The Two-Cornered Coevolution System was a baseline of coevolution LCSs, where two different agents (the generation agent and the classification agent) interact with each other to adapt to the changes of the problem. The *classification agent* evolves to learn various problems, while the *generation agent* evolves to tune and adjust the problem's difficulty based on the classification agent's ability to learn. Both the problem domain (i.e. the generation agent) and the learner (i.e. the classification agent) evolve in parallel (coadaptive evolution) to adapt to the changes of the problem. Here, various problems for classification at different levels of difficulty can be generated. Further, the problem domain was tuned autonomously based on the learner's ability to learn, i.e. the problem made 'hard' or 'easy' to specified limits or 'harder' or 'easier' based on a certain threshold value. The system was able to tune the problem domain autonomously such that the problem's difficulty can be tuned efficiently to empirically test the learning bounds of the learner by lowering human intervention. The hypothesised degrading effects of noise in the problem on the system's performance were confirmed.

   Part of this contribution was published in:

   Syahaneim Marzukhi, Will N. Browne and Mengjie Zhang, "Two-

Cornered Learning Classifier Systems for Pattern Generation and Classification", Proceedings of the Genetic and Evolutionary Computation Conference 2012 (GECCO 2012), pages 895-902, ACM.

Syahaneim Marzukhi, Will N. Browne and Mengjie Zhang, "Adaptive Artificial Datasets to Discover the Effects of Domain Features for Classification Tasks", Proceedings of the Genetic and Evolutionary Computation Conference 2013 (GECCO 2013), pages 157-158, ACM.

Syahaneim Marzukhi, Will N. Browne and Mengjie Zhang, "Adaptive Artificial Datasets Through Learning Classifier Systems for Classification Tasks", Proceedings of the International Workshop in Learning Classifier Systems 2013 (IWLCS 2013), page 1243-1250, ACM.

Syahaneim Marzukhi, Will N. Browne and Mengjie Zhang, "Adaptive Artificial Datasets Through Learning Classifier Systems for Classification Tasks", Evolutionary Intelligence (October 2013), DOI 10.1007 / s12065-013-0094, Springer-Verlag Berlin Heidelberg.

3. The thesis has introduced a new technique for addressing classification tasks using the *Three-Cornered Coevolution System* for the first time. This is a new coevolution LCS where three different agents evolve to adapt to the changes of the problem. Both of the classification agents evolve to learn various classification problems, while the generation agent evolves to tune and adjust the problem's difficulty based on the classification agents' ability to learn. The classification agents used different types of learning technique (i.e. reinforcement learning and supervised learning) to learn the classification tasks. Based on the classification agents' ability (i.e. the difference in performance between the classification agents) the generation agent adjusts and creates various problems for classification at different lev-

els of difficulty (i.e. various 'hard' problems). The results showed that the coevolutionary (coadaptive evolutionary) process was successfully implemented without human intervention as the generation agent and the classification agents evolved and adapted to the changes of the problems autonomously.

1.6 Organisation of the Thesis

The remainder of the thesis is organised as follows.

Chapter 2 presents a review of the literature on Learning Classifier Systems (LCSs) as a main approach to address classification problems. A detailed description of LCSs is given to provide the background for the readers understanding the main methodology. The chapter also covers important background of EC from an artificial intelligence (AI) and machine learning (ML) perspective. It therefore visits the basic concepts of evolutionary algorithms (EAs) and genetic algorithms (GAs). This chapter then gives a review of the current research on using LCSs for pattern classification problems.

Chapter 3 details the methodology used for the three different phases of research work in order to develop the Three-Cornered Coevolution System. This chapter provides a framework for developing, testing and evaluating the systems for each phase. The Three-Cornered Coevolution System work consists of three main phases. Phase 1 is to develop an automated evolvable problem generator for generating the classification problems. Phase 2 is to develop the Two-Cornered Coevolution System for classification, and Phase 3 is to develop the Three-Cornered Coevolution System for classification. Detailed design of each system, including the problem setup and the experimental setup, is discussed and presented.

Chapter 4 presents the results of the research for each phase. In *Phase 1*, a set of problem domains for the classification task (i.e. image-based data or artificial data) that can be generated automatically has been estab-

lished. Various problem domains have been successfully tested with the classification agent in order to determine its learning bounds and investigate its performance. The results indicated within this domain whether the problem's difficulty can be increased or decreased at the appropriate level, the classification agent's performance can be investigated and further tuning can be performed. In *Phase 2*, the system was further enhanced where the problem's difficulty can be tuned and adjusted whilst lowering human involvement. However, the results suggested that generating the artificial data through specifying features rather than image-based data led to a system that can tune and adjust the problem's difficulty meaningfully. In addition, applying Pittsburgh-style LCS (i.e. A-PLUS) coupled with Tabu Search rather than Michigan-style LCSs helped the generation agent to evolve the rules most effectively. The generation agent was able to generate the next problem based on the classification agent's ability and learned the difficulty of the problem.

Phase 3 is the final research goal. The Three-Cornered Coevolution System for addressing classification tasks has been developed where the system consisted of three different communicating agents. In this phase the classification agents' performance that used a different style of learning technique (i.e. supervised or reinforcement learning technique) on various problem domains was analysed in order to confirm that the coevolution process had occurred. The results showed that the generation agent was able to drive the coevolutionary process (i.e. coadaptive coevolution) within the system successfully.

Chapter 5 discusses the relevance and impact of the novel work in the LCSs field, highlights the limitations of the work and presents a detailed analysis of findings for each phase. Comparison is made with existing literature to place the results in context.

Chapter 6 summarises the findings and contributions from the experiments in each phase of the thesis, and describes future research directions arising from the contributions of this work.

Chapter 2

Background

This chapter gives a general description of the field of the work for this thesis and provides a review of the literature that forms the background to the novel contributions. The chapter contains an overview of artificial intelligence and machine learning in which the work is situated, and the task to which the application of the work is applied, i.e. classification problems. Next, the chapter focuses on the specific machine learning paradigm that is used in the contributions of the work: Genetic-based Machine Learning (GBML). GBML systems are machine learning techniques that use evolutionary computation (EC) in learning tasks. Details of Learning Classifier Systems (LCSs), a subset of GBML systems, are presented as a main approach to addressing the classification problems.

The chapter is structured as follows. First, sections 2.1 and 2.2 provide a description of artificial intelligence (AI) and machine learning (ML) and ML paradigms, particularly on the machine learning task we are dealing with (i.e. the classification problem) by defining the scope and the concepts that are going to be used in the work. Secondly, sections 2.3 and 2.4 give a brief description of evolutionary computation (EC) and evolutionary algorithm (EA) field. Thirdly, section 2.5 describes the main knowledge discovery mechanism of genetic algorithms (GA), and shows GA theory and a formal methodology of its application. Next, section 2.6

describes coevolution that applies to the work. The final three sections focus on GBML related contents, specifically LCSs in section 2.7. Section 2.8 and 2.9 provide details of four LCSs models that are used to address the classification problems (i.e. XCS, XCSR, UCS and A-PLUS). Section 2.10 provides a description on Tabu Search (TS) that is used to enhance the LCSs model, A-PLUS, for exploring the search space of the problem domain. Section 2.11 discusses related works of LCSs for addressing the classification tasks. Finally, section 2.12 provides a summary of the chapter.

2.1 Overview of Artificial Intelligence

Artificial Intelligence (AI) is a cross-disciplinary field of research that is generally concerned with developing and investigating systems that act intelligently [92]. There are four categories of AI as defined by Russell and Norvig [98]:

1. Systems that think like humans: systems that model the cognitive information processing properties of humans. This includes fields such as cognitive science and cognitive neuroscience.

2. Systems that act like humans: systems that can perform specific tasks that humans can do, which includes fields such as the Turing test, natural language processing, automated reasoning, knowledge representation, machine learning, computer vision, and robotics.

3. Systems that think rationally: systems that model the laws of rationalism and structured thought of humans, such as syllogisms and formal logic.

4. Systems that act rationally: systems that can act rationally as human behaviours, such as expected utility maximization and rational agents.

Briefly, artificial intelligence is the study of intelligent mechanisms and intelligent behaviours in the system. In the work for this thesis, we focus on a system that *acts rationally* in order to construct an intelligent agent that can perceive its environment, generate plans, execute those plans and communicate with other agents.

2.2 Overview of Machine Learning

Machine learning (ML) is a major research area in AI that is concerned with the design and the development of algorithms and techniques that allow computers to learn, where computers evolve their behaviours based on empirical data and automatically improve with experience.

Machine learning can be defined as computational methods using experiences to improve performance or to make accurate prediction of specific tasks [79]. The goal of machine learning is to design computer programs which learn to solve problems without explicitly being programmed or instructed [25].

In [78], Mitchell defined machine learning as any computer program that improves its performance at some task through experience as follows.

> "*A machine is said to learn from an experience* ***E*** *with respect to a particular task* ***T*** *and performance is measured by* ***P****, only if the system improves its performance* ***P*** *at task* ***T*** *by following experience of* ***E****.*"

Machine learning is generally categorized according to the type of learning procedure used to generate the output value. Learning methods in machine learning can be distinguished into: supervised, unsupervised and reinforcement learning [79].

In *supervised learning*, the learner receives a set of labelled examples (a training set) consisting of a set of instances that have been properly labeled

with the correct output (i.e. classification, regression and so forth). The goal is to learn a function that can perform an input-output mapping.

In *unsupervised learning*, the learner receives unlabelled examples where the training data has not been labelled (i.e. clustering, dimensionality reduction). The goal is to find inherent patterns in the data that can then be used to determine the relationships between inputs.

In *reinforcement learning*, the learner (the agent) actively interacts with the environment and in some cases affects the environment, and receives an immediate reward for each action. The desired outputs are not directly provided. Instead, a learner has to learn based on rewards and punishments it receives for its actions. The goal is to maximize the reward over a course of actions and iterations with the environment.

This can be illustrated by the following two tasks [58].

- Classifying a dataset of mushroom varieties.
  Task **T**: distinguish and classify between poisonous mushrooms and edible mushrooms; Performance measure **P**: percentage of correctly classified mushrooms between two types of mushrooms; and Training experience **E**: a database of pre-classified mushrooms; the description between two types of mushrooms (i.e. colour, size, shape) with a given class.

- Simulating a frog.
  Task **T**: maximize the number of flies it eats, minimize its energy expended and avoid being eaten itself; Performance measure **P**: number of flies eaten and survive; and Training experience **E**: trial and error practice to maximize the number of flies it eats while minimizing the energy use.

The previous two examples describe two types of learning task where it is possible to apply a machine learning technique for solving those problems. The first example shows a pattern classification task, while the latter shows an on-line control task, where each task is suitable for a different

learning technique. In the first problem, a supervised learning technique is suitable for classifying between the two types of mushrooms, when a training set of pre-classified examples is provided. However, in the second example, when the system is only being given certain conditions, either positive reward (i.e. eating flies) or negative reward (i.e. being eaten itself), then the reinforcement learning technique is more appropriate to solve this problem.

The mushroom classification task can also be categorized as a non-sequential task (single-step task), when the action taken by the learner has no effect on the inputs it will receive in the future. However, the frog simulation task is a sequential task (multi-step task), when an action taken at time t by the learner might influence the inputs it will receive in the future. In general the sequential task is more difficult, as it requires the learner to consider the long term consequences of its actions.

In the work for this thesis we are dealing with the supervised learning and reinforcement learning techniques for addressing the non-sequential task (i.e. classification). Both of the learning techniques will be implemented in the agents for the Three-Cornered Coevolution System. Therefore, the rest of this chapter will focus on these two paradigms of learning techniques for classification.

2.2.1 Machine Learning for Classification

The field of *pattern recognition* is concerned with the automatic discovery of regularities in data through the use of computer algorithms, for further actions such as classifying the data into different categories [11]. An example of pattern recognition is *classification*, which is a task of assigning one of several predefined categories to each object in previously unseen data [19].

Classification is a data mining task which resolves the class of an object by assigning a collection of objects (data) to the target classes (categories).

The goal of classification is to accurately predict the target class for each case in the collection of objects (data). The simplest type of classification problem is binary classification. In *binary classification*, the target class has only two possible values: for example, poisonous mushrooms and edible mushrooms.

A *pattern* is a characteristic of an observation such as a speech signal or a human face image, while a structural characteristic extracted from a pattern is called a *feature* [17]. Each pattern can be viewed as a point (a vector) in the feature space. From a statistical classification point of view, a pattern is represented by a set of d features (attributes), viewed as a d-dimensional feature vector [50]. The process of converting a pattern to features is called feature extraction. A feature selection algorithm is used to select the best features for representing the classes or the distinction between classes.

In the *statistical classification* approach, the goal is to choose features that allow pattern vectors, which belong to different categories, to occupy a d-dimensional feature space [50]. The representation of feature space is considered effective if the patterns are well separated between classes. During training (i.e. given a set of training patterns from each class), the objective is to establish decision boundaries in the feature space which separate patterns to different classes. In the training mode, a classification algorithm finds a relationship between features for representing the input patterns and the classifier is trained to partition the feature space. Different classification algorithms use different techniques for finding those relationships. In the test mode, the trained classifier assigns an unseen input pattern to one of the classes under consideration, based on the measured features.

Pattern classification is performed by assigning an output value (also known as label, category, target class) to a given input value (also known as an instance or an example), according to some specific algorithm. A *pattern* can be represented as a pair of variables: $[x, \omega]$, where x is a vector

of observation, and ω is a label that represents a meaningful concept for a problem domain [100]. *Classification* is a task of learning a target function $f : x \rightarrow \omega$ that maps each attribute set x to one of the predefined class labels ω. The target function can be used as an explanatory tool to distinguish between objects of different classes (*descriptive modeling* [19]) , or can be used as a predictive tool to predict the class label of unknown records (*predictive modelling* [19]).

2.2.1.1 Problem Difficulty

In most existing artificial classification systems, the problem domain is created and controlled by humans. Humans set up and tune the problem domain, such as determining the problem's difficulty. If the problem is too complex, the system does not learn. Conversely, if the problem is too simple, the system does not reach its full potential and can classify all of the examples easily. If humans can set up the problem appropriately then the machines can extract beneficial knowledge to solve the classification task.

A *problem* can be difficult for different reasons [39], which can affect the performance of classifiers. In [39], Ho and Basu identified that problems can be difficult because of a mixture of three effects: (1) class ambiguity, (2) boundary complexity, and (3) sample sparsity and feature space dimensionality. In [73], Macia defined these three effects as follows.

Ambiguity refers to a situation when there are examples in which their features do not allow distinguishing the class. Usually, this ambiguity is due to the problem formulation in which the concepts are intrinsically inseparable or the set of attributes is not sufficient to describe the concepts. Class separability and problem linearity are based on the geometrical complexity of data structure. The classes are ambiguous regardless of sample size or feature space dimensionality.

Boundary complexity is related to the description of the class boundary. Complex decision boundaries and subclass structure can be categorized

by the minimum length of a computer program needed to reproduce it. Boundary complexity is due to the nature of the problem regardless of sample size or feature space dimensionality.

Sample sparsity and feature space dimensionality are concerned with the difficulty which occurs when the sampled instances of a problem do not contain all of the necessary patterns. Moreover, small sample size and high dimensionality are likely to increase this difficulty, making the solution more complex to discover. Therefore, the rules may overgeneralize if they do not encounter examples near the decision boundary. In contrast, simple problems are normally linearly separable with wide margins between decision boundaries.

The major focus of the work for this thesis is mainly in the area of pattern recognition. More specifically, the work is focused on one example of pattern recognition which is classification. Thus, ML can be adapted to address the classification problem in order to achieve better results. Here, the learning algorithm (i.e. the classification agent) learns to recognize the patterns and makes intelligent decisions based on the empirical data (i.e. training set) for classifying the patterns to the correct class. This set of descriptors that characterize different aspects of complexity is useful to estimate the classification agent's performance, as well as to investigate the classification agent's domains of competence.

2.2.2 Machine Learning and Agents

During the last decade, developments in the fields of ML and agent technologies have, in some respect, become complementary and researchers from both fields have seen ample opportunities to profit from solutions proposed by each other [51]. The central goal of ML is to construct a complete intelligent agent that can perceive its environment, generate plans, execute those plans and communicate with other agents [63]. Recently, several agent-based frameworks and applications that utilize ML for mak-

ing intelligent decisions have been reported. Thus, research into learning agent technology, such as reinforcement learning, supervised learning and unsupervised learning, is increasingly becoming an important topic to further investigate for producing valuable applications [52].

Moreover, autonomous agents and multi-agent systems represent a new way of analysing, designing, and implementing complex software systems [52]. The agent-based applications can offer powerful tools and techniques that can potentially improve the implementation of many software systems. There have been considerable agent-based applications ranging from comparatively small systems such as personalised email filters to large, complex, mission critical systems such as air-traffic control.

In [52], an *agent* is defined as a computer system that is situated in an environment and capable of flexible autonomous action in order to meet its design objectives. There are three key main concepts in the definition: *situated*, *autonomy* and *flexibility* that can be described as follows [52].

- *Situated* refers to a situation where the agent perceives sensory input from its environment in which it can perform actions and adapts with the changes of the environment.

- *Autonomy* refers to a condition when the agent is able to act without the direct intervention of humans (or other agents). The agent has power to control its own actions and internal state and is capable of learning from experience.

- *Flexible* can be defined by two terms: responsive and pro-active. Responsive means that the agent should perceive its environment and responds in a timely fashion to the changes of the environment. Pro-active means that the agent should not simply act in response to the environment, but should be able to exhibit goal-directed behaviour by taking the initiative where appropriate.

An *autonomous agent* is defined as a computational system that inhabits some complex dynamic environment, senses and acts autonomously in

this environment, and by doing so realizes a set of goals or tasks for which they are designed [75].

In the work for this thesis, autonomous agents are implemented in the Three-Cornered Coevolution System. The system consists of three main agents, a generation agent and two classification agents that use different techniques of learning (i.e. supervised learning and reinforcement learning). All agents evolve to adapt to and drive the changes of the problem. The classification agents evolve to learn various classification problems, while the generation agent evolves to tune and adjust the problem (i.e. the problem's difficulty) based on the classification agent's ability to learn.

2.3 Overview of Evolutionary Computation

Evolutionary computation (EC) is a research area within AI that models the processes of natural evolution following Charles Darwin's theory of natural selection [6, 23], where the main concept is survival of the fittest. EC may also refer to an optimization methodology inspired by the mechanisms of biological evolution and behaviours of living organisms [127].

In *natural evolution*, survival is achieved through reproduction. The individuals with the best characteristics have greater chances of surviving and reproducing. Offspring are reproduced from two parents (sometimes more than two) containing genetic material of both (or all) parents, hopefully the best characteristics of each parent. Those characteristics will be passed on to the offspring, and will be inherited by the following generations, and (over time) become dominant in the population. The individuals with the best characteristics have greater chances for survival compared to those individuals that inherit weak characteristics. This process is referred to as the *survival of the fittest*. On the other hand, evolution is an optimization process where the aim is to improve the ability of the individual to survive in dynamically changing and competitive environments [22].

The terminology used by EC researchers is strongly influenced by the biological process of natural evolution. An *individual* is a candidate solution to a problem, while a *population* is an entire set of the current solutions. The actual representation (encoding) of the individual is called *genome* (chromosome), where each genome contains a sequence of genes (i.e. attributes that describes an individual). The value of a gene within a certain range is called an *alle* (i.e. value of the attribute). A *genotype* describes the genetic composition of the individual, which is inherited from the parents, while a *phenotype* is the expressed behavioural traits of the individual in a specific environment [23].

Evolution by natural selection simulates evolution based on the processes of reproduction, recombination, mutation, competition and selection [82] as follows. During each generation the individuals compete with others for reproduction. The individuals with the best characteristics have a greater chance to survive and to reproduce following the phenomena of survival of the fittest. The individuals can be modified to produce a new individual through a breeding process (i.e. combining parts of the other individuals (parents)) called recombination (crossover). This new individual is referred to as an offspring (child). Each individual can also be mutated which alters some of the alles of the chromosome through mutation. Next, each individual is evaluated and receives a grade called fitness. Fitness is used to indicate the quality of the individual in the context of a given problem. When the new offspring are introduced to the population and replace the current population, the new population is called a new generation.

Generally, EC algorithms include genetic algorithm (GA), evolutionary programming (EP), evolutionary strategies (ES), genetic programming (GP), learning classifier systems (LCS), differential evolution (DE), and estimation of distribution algorithm (EDA). There are a number of EC algorithms in the EC research community, where all of the algorithms have a similar framework in implementation and procedure as illustrated in Fig-

ure 2.1 [127]. The framework consists of three fundamental operations and two optional operations. The evolutionary iterations start after 'population initialization'. Then follows the two operations of 'fitness evaluation and selection' and 'population reproduction and variation'. Next, the new population is evaluated again and the iteration continues until a termination criterion is satisfied. Besides those three operations, EC algorithms sometimes perform another additional process such as an 'algorithm adaptation' procedure or 'local search' procedure.

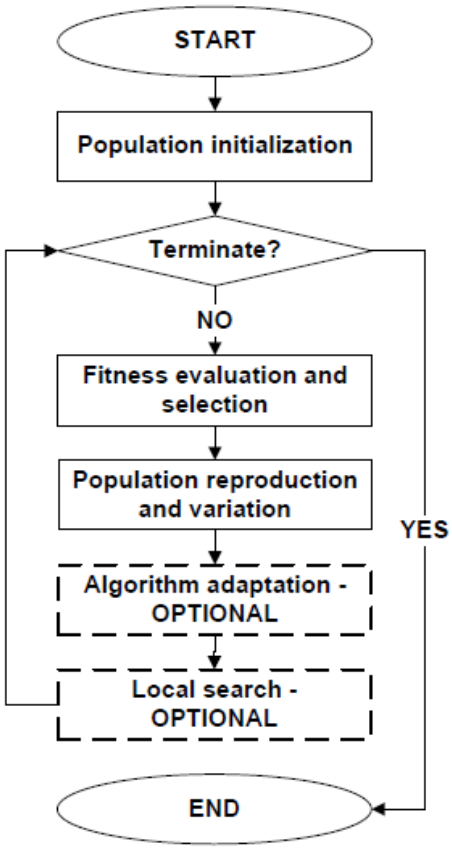


Figure 2.1: The flow of general EC scheme (adapted from [127]).

EC is also seen as the emulation of the process of natural selection in a search procedure of computational systems [23]. Therefore, evolutionary

processes can be simulated using computers to execute millions of generations of the procedure where it can be repeated and investigated under various circumstances [22].

An EC algorithm is implemented in the work for this thesis, specifically LCSs in the autonomous agents. Later, the EC algorithm is enhanced with ML techniques in order to improve its performance. The idea is to use ML techniques to directly guide the EC algorithm, i.e. GA, to search the space of production rules and enhance the search performance, which has been proven to improve the solution's quality.

A main branch of EC is evolutionary algorithms (EAs), which are to be reviewed below (see Section 2.4). Another branch of EC, swarm intelligence, is not used in this work, so will not be described here.

2.4 Overview of Evolutionary Algorithms

Evolutionary algorithms (EAs) are based on the notion of a dynamically changing population due to the birth of new individuals. These new individuals inherit genetic materials from parent individuals with high fitness, while individuals with low fitness are likely not to be selected for reproduction, and may be eliminated from the population.

EAs can also be classified as *guided random search techniques* (non-deterministic search) [8], which are used to attempt to find an optimal solution for a given problem. The guided random search methods are useful in the problems where the search space is huge, multimodal, discontinuous, and a near-optimal solution is acceptable [8].

The main idea of all these EAs is as follows: given a population of individuals, the environmental pressure causes natural selection among the individuals, and improves the fitness (quality) of the population [22]. Figure 2.2 describes the general scheme of EA and this is further elaborated in Algorithm 1.

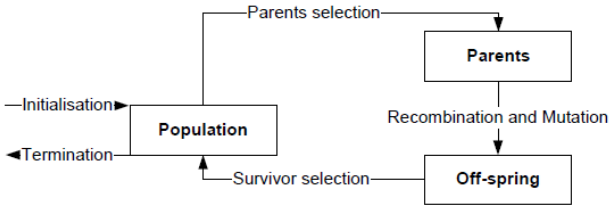


Figure 2.2: The flow of a general EA scheme (adapted from [23]).

Algorithm 1: Algorithmic description of EA (adapted from [22]).

```
1  begin
2      INITIALISE population with random candidate solutions
3      EVALUATE each candidate
4      while (termination condition is not satisfied) do
5          SELECT parents
6          RECOMBINE pairs of parents
7          MUTATE the resulting off-spring
8          EVALUATE new candidates
9          SELECT individuals for the next generation
10     end
11 end
```

EAs have a number of components that must be specified in order to define a particular EA [23]:

1. An encoding of solutions.
2. A function to evaluate the fitness (fitness function).
3. Initialization of the initial population.
4. Selection operators.
5. Reproduction operators such as crossover and mutation.

Different representation schemes of candidate solutions are often used to categorise different EAs into different paradigms. The four major current implementations of EA approaches are briefly described below:

- **Genetic Algorithm** [40, 41, 33, 34]
  A *Genetic algorithm* (GA) models genetic evolution. GA was developed and applied by John Holland. Originally, GA was designed as a formal system for adaption rather than an optimization system. In Holland's original work, each individual is represented by binary strings. GA basic features are: 1) crossover is the main method for reproduction, while mutation plays a minor role; 2) the proportional selection is used for selection purposes. Later, several changes were made to this original GA, such as using different representation schemes, other selection methods, other crossover and mutation operators for creating new individuals, and elitism to retain the best individuals.

- **Genetic Programming** [59, 60]
  Genetic programming (GP) is an extended form of GAs, which genetically breeds a population of computer programs to solve problems. Each individual of the population is represented by a complete computer program in a suitable programming language. The most commonly used representation schemes include trees, binary, machine

code, and many others to represent functions and arguments (e.g. arithmetic operations, mathematical functions and conditional logical operations) in the computer program. GP uses similar reproduction operators such as in GAs.

- **Evolutionary Strategy** [96, 101, 102]
  Evolutionary strategy (ES) typically employs real-valued parameters. Each individual is represented as a genetic building block and a set of strategy parameters to model the behaviour of the individuals in the environment. ES basic features are: 1) the distinction between a parent population (of size μ) and an offspring population (of size $\lambda \geq \mu$); 2) mutation is the main method for reproduction.

- **Evolutionary Programming** [28, 27, 26]
  Evolutionary programming (EP) was originally developed as a method to evolve finite-state machines for solving time series prediction tasks and was later extended to parameter optimization. EP relies on mutation as the main operator for reproduction based on a single parent. EP uses tournament selection to determine the number μ of individuals for survival from the parents and the offspring. EP also uses the self-adaption principle to evolve the strategy parameters during the training.

In the work for this thesis, GA is implemented specifically in the LCS methods as a main tool to evolve the rules. GA is commonly used by the LCS community as it is well understood and improvements to the system can be easily analysed. GA is used repeatedly to refine the rules for a finite number of generations (iterations) until a termination criterion is met. The main aspects of GAs will be reviewed in Section 2.5.

2.5 Genetic Algorithm

Genetic algorithm (GA) [40, 41] updates a population of potential candidate solutions iteratively. GA operates through a simple cycle as follows. At each iteration (generation), GA evaluates candidate solutions and generates offspring based on the fitness of each candidate solution. Offspring are reproduced from two parents, which contain genetic material of both parents; hopefully, the best characteristics of each parent. Substructures of the candidate solutions are then modified through genetic operators, such as mutation and crossover, to form a better candidate solution. GA models the evolution of the population following a general scheme such as in Algorithm 2 and specifies a number of its components as follows [22].

Algorithm 2: Algorithmic description of GA (adapted from [6]).

```
1  begin
2      Set generation t=0
3      INITIALISE the initial population P(t)
4      EVALUATE structures in P(t)
5      while (termination criteria not satisfied) do
6          t=t+1
7          SELECT for reproduction C(t) from P(t − 1)
8          RECOMBINE and MUTATE structures in C(t) forming C′(t)
9          EVALUATE structures in C′(t)
10         SELECT from C′(t) and P(t − 1) forming P(t)
11     end
12 end
```

2.5.1 Knowledge Representation

In order to solve an optimization problem, GA starts with the chromosome representation of a parameter set. The parameter set can be encoded as binary, real-value, integer or permutation representation. A set of the

chromosomes is called a population, where the size of the population may be constant or may vary from one generation to another. The standard practise to generate the initial population is to choose randomly the gene values from an allowed set of values. The goal of random value selection is to ensure that the initial population is a uniform representation of the entire search space [23].

In the *binary representation*, the parameter set is encoded as a finite-length string over an alphabet of '0's and '1's. For example, the string '10011010' is a binary chromosome of length *l* (where *l*=8) and in this case there are $2^l = 2^8$ different chromosomes.

The *integer representation* is used to encode the problem of finding an optimal value for a set of variables that takes all integer values. For example, the values of {0, 1, 2, 3} can represent a path on a square grid of {North, South, West, East}.

In certain cases, a candidate solution to a problem may contain a string of *real-values*, when the solution is from a continuous distribution rather than a discrete distribution. Therefore, these real-values normally are implemented as floating-point values within the specified interval. The genotype for a solution with *k* genes is now a vector $\langle x_1, ..., x_k \rangle$ with $x_i \in \mathbb{R}$.

The work for this thesis is mainly focused on the area of classification (i.e. *image-based data* and *artificial data* for classification). The simplest encoding system to represent the image-based data is binary representation. However, the binary representations are not always suitable for addressing the problem that takes integer or real-value. Thus, the integer and the real-value representation is used to represent attributes (or features) in the problem and instances in the datasets for the artificial data.

2.5.2 Fitness Function

The fitness function is used as a quality measure to find good solutions (i.e. individuals with high fitness values). The fitness function can map a

chromosome representation into a scalar value: $F_{EA} : C^I \rightarrow \mathbb{R}$, where F_{EA} is the fitness function and C represents the I dimensional chromosome.

The fitness function quantifies the quality of each chromosome. The fitness function is used to evaluate each individual in the population in order to select individuals either for reproduction or mutation.

As the work for this thesis mainly addresses the classification tasks, the classification accuracy is used to measure the classification agent's performance for learning various classification problems in the fitness functions.

2.5.3 Selection Operators

The aim of the *selection operator* is to emphasize better individuals in the population. There are a number of selection schemes used in GA to select a number of parent individuals from the population for reproduction. The most common schemes used are: proportionate reproduction (e.g. roulette-wheel), ranking, tournament and steady state selection [35].

The name proportionate reproduction describes a group of selection schemes that choose individuals for reproduction depending on their objective function values [35]. The simplest selection scheme is roulette-wheel selection, also called stochastic sampling with replacement. In *roulette-wheel selection*, the chance of individuals being selected is proportional to their fitness values. First, the individuals are mapped to contiguous segments of a wheel, and each individual's segment is equal to the size of its fitness (see Figure 2.3). Next, a random number is generated and the individual whose segment spans the random number is selected for reproduction. This process is repeated until the desired number of individuals is obtained. It is possible that a few individuals with high fitness will dominate the population due to a wide range of fitness values. Roulette-wheel selection is strongly fitness dependent where selection probability depends on absolute fitness of individual compared to absolute fitness of rest of the population.

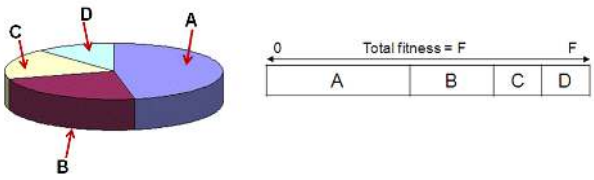


Figure 2.3: Roulette-wheel selection.

In *tournament selection*, at each iteration, a group of k individuals is randomly selected (sampled) from the population. This is called a tournament of size k. The individuals in the group participate in a tournament, where the fittest individual wins the tournament. A winning individual is determined based on its fitness value (i.e. individual's rank). The best individual (one with the highest fitness value) is usually chosen deterministically, though a stochastic selection may be made [6]. As tournament selection depends on the individual's rank rather than the individual's relative fitness, it is not affected by the fitness distribution through the population [22]. The size of the tournament controls the selection pressure, where a bigger tournament size generates a higher selection pressure [54]. In tournament selection, the worst individuals will not be selected, whereas the best individuals will not dominate the population.

Tournament selection runs faster compared to roulette-wheel selection, since roulette-wheel selection needs to compare the fitness of all individuals in the population. In tournament selection, it is easy to control the selection pressure by only varying the tournament size k. Thus, for all of the experiments in the work for this thesis, *tournament selection* is used to select the rules (classifiers) from the population either for reproduction or mutation.

2.5.4 Reproduction Operators: Crossover

Crossover (recombination) refers to the process of creating a new individual (offspring) from the two parent individuals, where the genome (chromo-

some) should contain the most important features from the parent individuals. There are three basic steps to the crossover operation [6]. First, two parent individuals are chosen at random from the population by the selection operator. Secondly, one or more crossover points are chosen as a breakpoint. Finally, the parent chromosomes are exchanged and then combined to produce two new individuals (offspring). The probability of the parent individuals undergoing crossover is controlled by the crossover rate $\chi \in [0, 1]$, which determines how frequently the crossover operator is activated.

2.5.4.1 Crossover for binary representation

Generally, there are three standard crossover methods used for the *binary representation* [22]: 1) single-point crossover, 2) two-point crossover, and 3) uniform crossover.

The *single-point crossover* [42, 53] is performed as follows (see Figure 2.4). First, one crossover point is selected randomly within the range of $[0, l-1]$ (where l is the length of the encoding) to split both parent individuals at this point. Next, the binary string from the beginning of chromosome to the crossover point is copied from its parent and the rest of the binary string is copied from another parent.

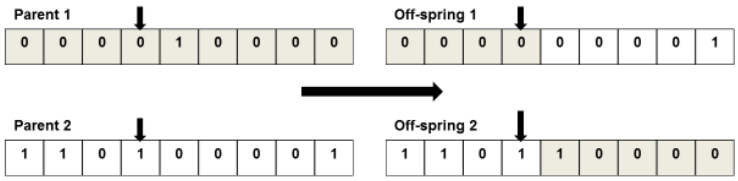


Figure 2.4: Single-point crossover in GA (adapted from [22]).

The *two-point crossover* [53] is performed as follows (see Figure 2.5). First, two crossover points are selected randomly within the range of $[0, l-1]$ (where l is the length of the encoding) to split both parent individuals at this point. Next, the binary string from the beginning of chromosome to

the first crossover point is copied from its parent. The part from the first to the second crossover point is copied from another parent and the rest of the binary string is copied again from its parent.

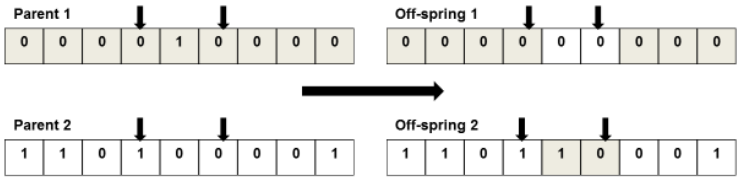


Figure 2.5: Two-point crossover in GA (adapted from [22]).

Figure 2.6 illustrates how *uniform crossover* [24] works by treating each genome independently and making a random choice as to which parent it should inherit from [22]. The binary string m is a mask computed for each invocation of the operator from the set of crossover points [6]. This mask is used to identify which string segments will be exchanged during the crossover operation. Bits are copied either from the first parent or the second parent to create the offspring.

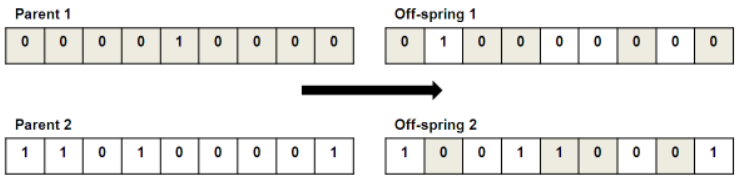


Figure 2.6: Uniform crossover in GA (adapted from [22]).

2.5.4.2 Crossover for floating-point representation

In general, *arithmetic crossover* [23] is usually applied to the *floating-point representation*, where an arithmetic operation is performed to create a new offspring as follows.

$$O_{n1,i} = r_1 C_{n1,i} + (1.0 - r_1) C_{n2,i} \tag{2.1}$$

$$O_{n2,i} = (1.0 - r_2)C_{n1,i} + r_2 C_{n2,i} \tag{2.2}$$

Where O_{n1} and O_{n2} are the generated off-springs, and C_{n1} and C_{n2} are the parents with $r_1, r_2 \in U(0, 1)$.

There are three types of arithmetic recombination used for the floating-point representation [22]: 1) simple recombination, 2) single arithmetic recombination, and 3) whole arithmetic recombination.

The *simple arithmetic recombination* is performed as follows (see Figure 2.7). First, one recombination point k is selected within the range of $[0, l-1]$ (where l is the length of the encoding) to split both of the parents at this point. Next, the first k floats from the beginning of chromosome to the crossover point, is copied from its parent and the rest of the floats are the arithmetic average of both parents.

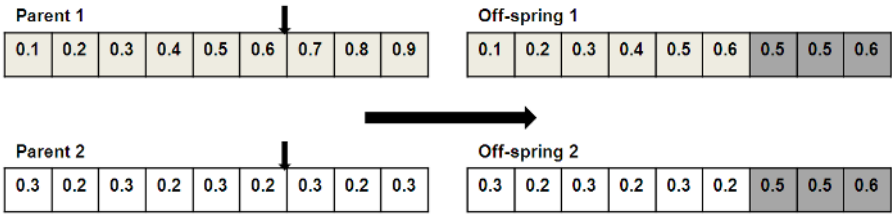


Figure 2.7: Simple arithmetic recombination in GA (adapted from [22]).

The *single arithmetic recombination* is performed as follows (see Figure 2.8). First, one random point k is selected within the range of $[0, l-1]$ (where l is the length of the encoding). At that position k, the offspring's float value is the arithmetic average of both parents, while the rest float values are from its parent.

The *whole arithmetic recombination* is performed as follows (see Figure 2.9). Each of the offspring's float values is the arithmetic average of both parents.

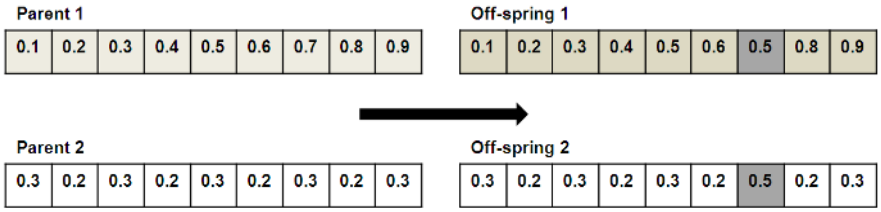


Figure 2.8: Single arithmetic recombination in GA (adapted from [22]).

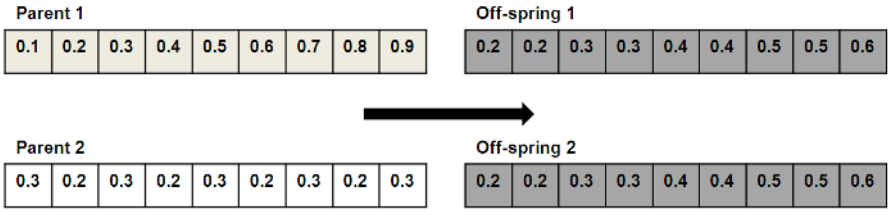


Figure 2.9: Whole arithmetic recombination in GA (adapted from [22]).

2.5.5 Reproduction Operators: Mutation

The aim of *mutation* (variation) is to introduce new genetic material into an existing individual in order to add diversity to the genetic characteristics of the population. Mutation occurs with a certain probability (i.e. mutation rate $\mu \in [0, 1]$) [23]. Usually, a small value of mutation rate is used. Mutation is used to find good solutions (e.g. to keep the fit individuals from distortion, so that the good characteristics of the fit individual are preserved).

2.5.5.1 Mutation for binary representation

There are two mutation methods used for the *binary representation* [23]: 1) random mutation and 2) inorder mutation.

The *random mutation* is applied as follows (see Figure 2.10). First, the bit positions are selected randomly. Next, a random value for each bit is generated. If the random value is less or equal than μ, then flip the corresponding bit value either from '1' to '0' or from '0' to '1'.

Figure 2.10: Random mutation in GA (adapted from [23]).

The *inorder mutation* is applied as follows (see Figure 2.11). First, the two bit positions are selected randomly. Only the corresponding bit values between these positions are mutated. Next, with probability μ flip the corresponding bit value either from '1' to '0' or from '0' to '1'.

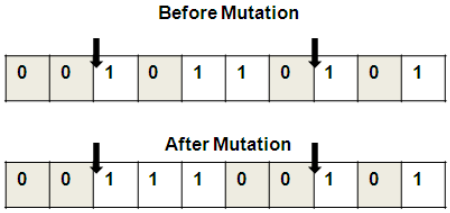


Figure 2.11: Inorder mutation in GA (adapted from [23]).

2.5.5.2 Mutation for floating-point representation

In general the value of each gene is randomly changed within its domain for a given lower bound L_i and upper bound U_i as follow:
$\langle x_1, ..., x_n \rangle \rightarrow \langle x'_1, ..., x'_n \rangle$, where $x_i, x'_i \in [L_i, U_i]$.

There are two mutation methods used for the *floating-point representations* [22]: 1) uniform mutation and 2) non-uniform mutation. In the *uniform mutation*, the values of x'_i are drawn uniformly from $[L_i, U_i]$. However, in the *non-uniform mutation*, a random value is added to the allele, usually sampled from a Gaussian distribution with mean zero and user specified

standard deviation, and then the resulting value is suppressed within the range $[L_i, U_i]$.

In the work for this thesis, mutation is used to support crossover in order to ensure that the full range of allele values is accessible in the search. Mutation is performed by creating one offspring from one parent. Mutation creates random small diversions by staying near (in the area of) the parent. The binary mutation is used for the image-based data, while mutation for floating-point is used for the artificial data, in order to add new material to the solutions.

2.6 Coevolution

According to the Darwinian theory [6], an individual evolves through interaction with the environment. Coevolution is the complementary evolution between closely associated species (individuals) [23], in a situation where two different species interact with each other. In *evolutionary algorithms* (EAs), evolution is viewed as the process that enables the population to adapt in a fixed simulated environment [23]. However, in *Coevolutionary Algorithms* (CEAs), evolution is actually influenced by other independently acting biological populations in the fixed simulated environment [23]. Therefore, the evolution is not just locally within each population, but also in response to environmental changes as caused by other populations. As a consequence, natural evolution actually implies coevolution.

Coevolution is a situation where multiple populations are evolved and the fitness of an individual depends on its interactions with other individuals. For example, predator-prey relationships involve coevolution, where an evolutionary advance in the predator will trigger an evolutionary response in the prey [7]. In coevolutionary systems, different populations interact with each other in such a way that the evaluation function of one population may depend on the state of the evolution process in the other

population(s) [6].

There are two main forms of CEAs [22]: competitive coevolutionary systems and cooperative coevolutionary systems. In *competitive coevolution*, individuals have to coevolve against other individuals (within a population or in opponent population). The evaluation of the individuals is determined by a set of competitions between two or more individuals and the fitness of an individual depends on the fitness of opponent individuals. In contrast *cooperative coevolution* evolves different populations each of which contain partial solutions to a problem. The collaboration between two or more individuals is necessary in order to evaluate one complete potential solution and the fitness of an individual is measured by combining it with one member from each of the other populations to form a complete solution.

Coevolutionary algorithms bring an interesting perspective to evolution as they promote a different manner of the fitness evaluation of a candidate solution, which takes into account its relation to the other surrounding individuals [16]. In addition, coevolutionary evaluation is continuously altered throughout the existence of an individual as a result of better roles achieved at each generation (survival of the fittest). Thus, coevolution has been very successful in many applications, which provide a suitable approach to solve many real-world applications such as game theory, robotics, classification and data mining.

The first work on cooperative coevolution in the GA domain was conducted by Potter and De Jong [93, 95]. In their work, the cooperative coevolution genetic algorithm (CCGA) was initially designed for function optimization, and later a general architecture was proposed for cooperative coevolution with coadapted subcomponents [94]. In [118], a framework for formal analysis of cooperative coevolutionary algorithms (CCEAs) using multi-population symmetric (MPS) games from evolutionary game theory (EGT) was introduced. An enhanced cooperative coevolution genetic algorithm (ECCGA) has also been applied to pattern clas-

sification to further improve the classification performance [128]. A competitive coevolution approach has been applied for developing strategies to play games [112]. The results show that the coevolutionary approach, which allows individuals to play against each other, can lead to much better results compared to those learned with fixed external opponents.

Further, coevolution with LCSs was also employed for addressing many other applications and achieved reasonable successes [48, 70, 76, 117]. In [48], Huang and Sun proposed a coadaptive approach to control coevolution-based eXtended Classifier System (XCS) parameters. The system (i.e. coadaptive XCS, CA-XCS) was tested on the 6-bit multiplexer problem. In this approach one XCS was used to adjust the parameters of the other XCS system. This approach (a coevolution model) allowed two XCS systems to operate in parallel to solve the target and the parameter setting problems simultaneously. Results showed that the coadaptive approach was successful in terms of setting parameters according to target problem properties.

In [70], the paper addressed on how different knowledge representations can be evolved in a fine-grained parallel learning classifier systems (i.e. Genetic and Artificial Life Environment, GALE) for data mining tasks. Experiments were performed with GALE2 to solve two well-known datasets provided by the UCI repository [5]: the Iris dataset and the Wisconsin breast cancer dataset to demonstrate that the fine-grained parallel learning classifier systems can evolve individuals codifying different knowledge representations at the same time. Results showed that when an adequate extinction pattern was used, accurate individuals belonging to different knowledge representations can be coevolved efficiently.

In [76], the paper proposed the CoXCS model, a coevolutionary multi-population XCS. In this model, a number of isolated sub-populations were used to evolve classifiers based on partitioning of the feature (attribute) space. A modified version of XCS was used in each of the sub-populations. Two modifications were made to the base XCS model running in each is-

land: a new algorithm was used to create the match set and a specialized crossover operator was used. Next, the model was compared with some of the representatives of several machine learning paradigms (i.e. j48, NBtree, Random Forest, NN and SVM) on a collection of 6 real-world datasets extracted from the UCI repository. Results showed that the accuracy of the proposed model was significantly better than other well-known classifiers when the ratio of data features to samples was extremely large. Results suggested that the composition strategy played an important role in guiding the trajectory of the evolving populations.

In [117], the paper presented a novel hybrid learning algorithm, namely CoCoLCS MFS (a cooperative coevolution Pittsburgh Learning Classifier Systems embedded with Memetic Feature Selection) in which a memetic feature selection search is embedded into the classifier evolution process of GAssist, a Pittsburgh-style LCS, by means of a cooperative coevolution framework. In the proposed approach, the selected feature subsets and the rule sets of classifiers in GAssist were encoded and evolved by two separate populations. These two coevolving populations cooperate with each other regarding fitness evaluation. Results on several benchmark datasets chosen from the UCI repository illustrated that the proposed CoCoLCS MFS was capable of delivering solutions with better accuracy and higher stability, compared with the original GAssist. Moreover, the incorporated feature selection helped to improve the computational efficiency by reducing the number of features involved in the classier evaluation.

However, in our implementation, the version of coevolution in the Three-Cornered Coevolution LCSs System is implemented differently. The term *coevolution* that is used in this thesis refers to the fact that several LCSs interact and learn with each other to adapt with the changes of the problems. The Three-Cornered Coevolution System is implemented in such a way that the agents interact in a *coadaptive evolution manner*. Each agent adapts and learns the changes of the problems as the parameters setting in each agent are kept unchanged. However, the individuals in each agent

(i.e. rules) are evolved to learn various problems. Also, the three agents are not cooperating in the usual sense, because the agents are opposed in their purposes and objectives.

2.7 Genetic-based Machine Learning

Genetic-Based Machine Learning (GBML) systems are machine learning techniques that use evolutionary computation (EC) to search complex search spaces [91]. The first schemes of GBML systems were introduced by Holland in the 1970s [41], and were formally presented in the paper 'Cognitive Systems Based on Adaptive Algorithms' in 1978 by Holland and Reitman [44]. Research on GBML has been conducted from two perspectives [91]: the Pittsburgh approach and the Michigan approach.

2.7.1 Learning Classifier Systems

Learning Classifier Systems (LCSs) [41] are a subset of GBML systems [41], which are machine learning techniques that incorporate reinforcement learning (RL) and evolutionary computation (e.g. genetic algorithm (GA)) in its main component. LCSs are cross-disciplinary research areas of EC and RL (see Figure 2.12). The RL component is applied to the classifier's predictions to evaluate the classifiers [125] for the identification of the best rules, while the GA component is responsible for discovering potentially better rules [14]. The desired outcome of running a LCS is for the evolved classifiers to collectively model an intelligent decision maker [116].

In [43], Lanzi defined LCSs as follows:

> *"In LCSs an **agent** learns to perform a certain task by interacting with a partially unknown **environment**, using **rewards** and other feedback to guide an internal evolutionary process which modifies its rule-based model of the world. The agent senses the environment through its detectors; based on its current sensations and its past*

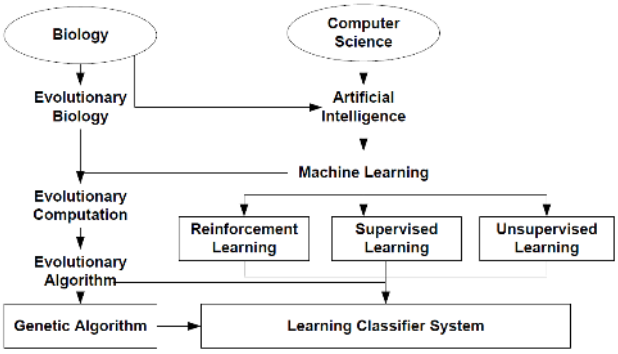


Figure 2.12: LCSs are cross-disciplinary research area of EC and RL (adapted from [116]).

> *experience, the agent selects an action which is sent to the effectors in order to be performed in the environment. Depending on the effects of the agent's action, the agent occasionally receives a reward. The agent's general goal is to obtain as much reward as possible from the environment."*

Learning via reinforcement is an essential mechanism in the LCS architecture, where the learning guides the evolutionary component (e.g. GA) to evolve a better set of rules. In *reinforcement learning* [111], the agent learns from its interaction with its environment (see Figure 2.13). The agent perceives the environment to be in a *state*, and selects an *action* to be executed. In response, the environment returns a numerical *reward* to the agent. The agent seeks to maximize the reward it receives in the long run by effecting appropriate actions to the environment in order to learn a mapping from situations to actions and increase its performance.

Reinforcement learning (RL) is defined as a learning technique for mapping a situation to an action where the main objective is to maximize the scalar reward (reinforcement signal) received by the agent from the environment [111]. In other words, RL can be referred to as learning by *trial-and-error* between the agent and the environment, the agent receives per-

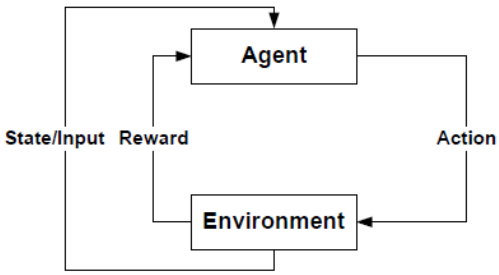


Figure 2.13: RL components (adapted from [111]).

formance feedback in return from the environment. In most of the cases, the actions may affect not only the immediate reward, but also the next situation through to all subsequent rewards [111]. The agent has to *exploit* what it already knows in order to obtain a reward, but it also has to *explore* in order to select a better action in the future. RL provides a flexible approach to design a system in situations for which both supervised learning and unsupervised learning are impractical.

LCSs comprise at least three main components [46]: the Performance Component, the Reinforcement Component and the Discovery Component (see Figure 2.14).

1. **Population**

   LCSs contain a finite population of *condition-action* rules called *classifiers* that represent the current knowledge of the system. Each classifier is usually defined by four main parameters [65]:

   - The *condition*, which specifies the input states in which the classifier can be applied.
   - The *action*, which represents a decision on the problem identified by its condition or specified action that the classifier proposes.

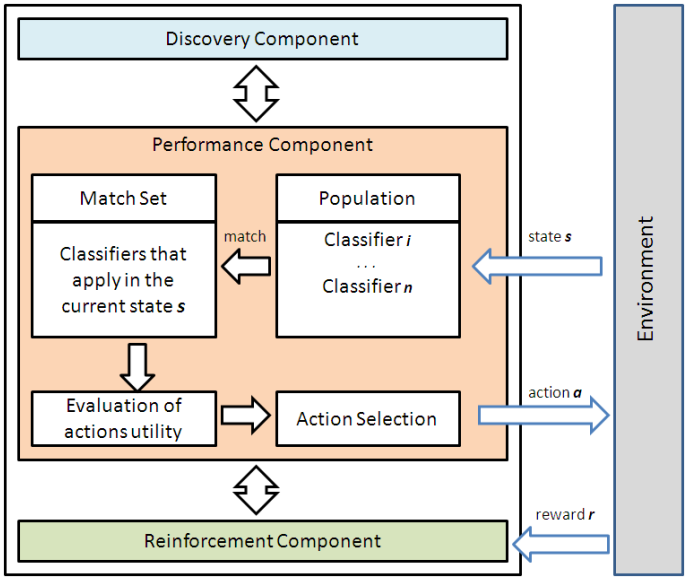


Figure 2.14: Learning Classifier Systems (LCSs) main components.

- The *prediction*, which estimates the amount of reward that the system will receive for the specified action it has performed.
- The *fitness*, which estimates how good the classifier (quality) is at solving the problem.

2. **Performance Component**

   The performance component controls the interaction between the population of classifiers and the environment. At each iteration, the system perceives the current state of the problem (i.e. input). The system initially learns by covering each complete pattern for the input. Next, it builds a match set containing all the classifiers in the population where the condition matches the input. Next, the system evaluates each action in the match set and selects an action to be performed. The selected action is sent to the environment to be executed on the problem. In response, the system receives a scalar reward depending on the effect of the action that has been executed.

3. **Reinforcement Component (Credit Assignment)**

   The reinforcement component distributes the incoming reward from the environment to the classifier that is accountable for the reward obtained, and the associated parameters are updated. This component is also known as credit assignment. In the sequential task (single step task), the credit assignment is normally performed by some form of algorithm (e.g. Q-learning-based strategy or any similar algorithm) [57] to distribute the reward received.

4. **Discovery Component**

   The discovery component which uses different genetic operators (i.e. mutation and crossover) is responsible for discovering better classifiers and improving existing ones. The GA is mainly used as a computational search technique to evolve a population of classifiers, where each classifier represents a potential solution (or piece of a solution) to a given problem. Here, the GA is performed following the standard GA operation methods [53].

2.7.2 Pittsburgh-style LCSs versus Michigan-style LCSs

Research on LCSs have been conducted from two perspectives [91, 116]: *Pittsburgh-style LCSs* and *Michigan-style LCSs*. These two approaches represent two very different ways of interpreting the contribution of EC to ML. A Pittsburgh-style LCS is an optimization tool applied to learning tasks that uses an EC technique as its main driving force, while Michigan-style LCS has been designed specifically for learning and it is a combination of several components, where one of them is an EC technique [1]. Several main distinctions between the two approaches include: population structure, solution structure, cooperation and competition within individuals in the population, and learning style [3].

2.7.2.1 Pittsburgh-style LCSs

The *Pittsburgh-style LCS* was introduced by Smith in his dissertation at the University of Pittsburgh in 1980 [108]. Pittsburgh-style LCSs define a *population* as a collection of multiple competing rule-sets which represent a potential solution (see Figure 2.15). Each *individual* is a rule-set of classifiers. Therefore, each rule-set in the population is a potential solution. The *solution* to the problem is the best individual of the population. The individuals in the population compete to solve the problem and compete for reproduction and evolve following the typical cycle of GAs. The GA operates at the level of the entire rule-set rather than at the level of the individual classifier. At the end of the evolutionary process, the best individual found is treated as the final solution and used to predict the class of unknown examples [125]. Thus, Pittsburgh-style LCSs are very close to the essence of EC techniques where an individual is a *complete solution*, and there is a competition between the candidate solutions in the population, and the search space exploration is made using almost blind genetic operators (without domain knowledge)[1].

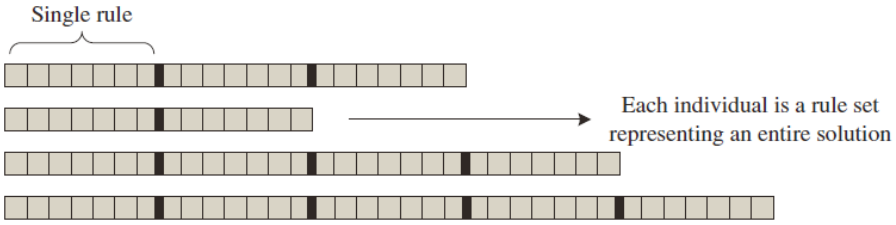


Figure 2.15: Pittsburgh-style LCSs (adapted from [125]).

Pittsburgh-style LCSs need to explore a much larger search space than Michigan-style LCSs, and they are often more computationally expensive as the system has to evolve multiple rule-sets, which requires longer evaluation times because the whole population of multiple rule-sets needs to be evaluated. Additionally, Pittsburgh-style LCSs need to address a *bloat*

effect, which refers to a situation where the size of individuals grows without control [4] (i.e. increases the size of candidate solutions by adding useless rules to individuals) [68]. Conversely, Pittsburgh-style LCSs usually evolve more compact populations involving only few rules in a population [1] compared to Michigan-style LCSs. Therefore, Pittsburgh-style LCSs are more suitable when compact solutions with few rules are expected to solve the problem and are normally applied to *off-line* learning problems. Some of the most successful Pittsburgh-style LCSs include GABIL, GALE, GAssist, and BioHEL (a descendant of GAssist) [116]. Those systems were designed primarily to address classification or data mining problems for which Pittsburgh-style systems are considered to be fundamentally suited. The basic algorithm of Pittsburgh-style LCSs is shown in Algorithm 3.

Algorithm 3: Algorithmic description of Pittsburgh-style's LCSs (adapted from [125]).

```
1  begin
2      Perceive a group of input from the environment.
3      Generate a random population of rule-sets.
4      Evaluate the population of rule-sets.
5      for (each rule-set repeat until stopping criterion is met) do
6          Select a promising rule-set.
7          Apply variation operator on the promising rule-sets.
8          Evaluate the new rule-sets.
9          Replace the population of rule-sets with the new rule-sets.
10     end
11     The best rule-set is treated as the final solution.
12 end
```

2.7.2.2 Michigan-style LCSs

The *Michigan-style LCS* was implemented in Holland's GBML system that was developed at the University of Michigan [45, 18], which merges a

credit assignment scheme with a GA to evolve a population of rules. In Michigan-style LCSs, a *population* consists of a single rule-set which represents the problem solution (see Figure 2.16). Each *individual* is a classifier of the single rule-set. Therefore, the *solution* is represented by the entire set of individual classifiers in the population. The individuals in the population cooperate to solve the problem and compete for reproduction. The RL exploits the incoming reward to estimate the action values in each sub-problem to identify the best classifiers in the population. Meanwhile, GA improves the current solution by means of exploring promising rules [125]. GA periodically operates at the level of each individual classifier referred as 'classifier-based competition'. Michigan-style LCSs typically evolve highly distributed problem solutions involving a large number of classifiers, which make Michigan-style LCSs better suited to distributed types of solution [1]. Michigan-style LCSs are typically applied to interactive *on-line* learning problems. The most popular Michigan-style LCSs are the accuracy-based LCSs (XCS) [119]. The basic algorithm of Michigan-style LCSs is shown in Algorithm 4.

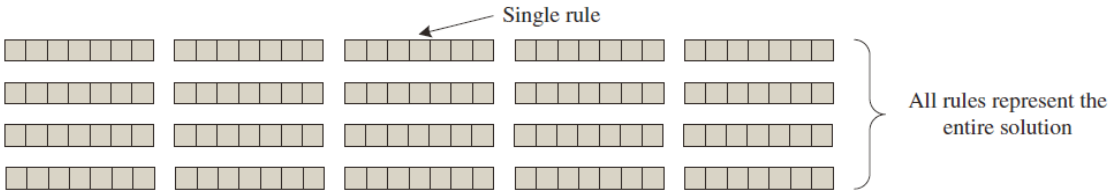


Figure 2.16: Michigan-style LCSs (adapted from [125]).

The work of the Three-Cornered Coevolution System consists of three main agents (i.e. a generation agent and two classification agents). All agents evolve to adapt to and drive the changes of the environment (i.e. the problem). The classification agents evolve to learn various classification problems, while the generation agent evolves to tune and adjust the problem (i.e. the problem's difficulty) based on the classification agent's ability to learn. Pittsburgh-style LCS is used in the generation agent to

Algorithm 4: Algorithmic description of Michigan-style's LCSs (on-line).

```
1  begin
2      Initialise a population of classifiers.
3      for (each classifier repeat until stopping criterion is met) do
4          Perceive input from the environment.
5          Select promising classifiers.
6          Apply variation operator on the promising classifiers.
7          Evaluate the new classifiers.
8          Replace some of the population of classifiers with the new
           classifiers.
9      end
10     The best classifiers are treated as the final solution.
11 end
```

learn the problem, where it can evolve and optimize the generation agent's rules (i.e. to produce compact solutions when only a few rules are expected). Michigan-style LCSs are used in the classification agents to learn various problems for classification (i.e. interactive on-line learning system).

2.7.3 Strength-Based LCSs versus Accuracy-Based LCSs

Traditionally, LCSs have been *strength-based* [119, 56], where the classifier's fitness is based on the prediction (reward) received from its interaction with the environment [107]. The prediction value is used to measure the fitness of each classifier when selecting any competitive or participative classifier. Therefore, the best rewarded classifiers will have a greater chance to be selected than others. As a result, GA will eliminate the less rewarded classifiers in comparison with others from the population.

As a consequence, the strength-based fitness can cause a problem of 'greedy-classification' and 'strong-over-general classifiers', where the over-

general classifier acts correctly in a high reward state and acts incorrectly in a low reward state [107]. In the rules' competition, the over-general classifier has more influence in the low reward state and has a greater chance for reproduction. So, this over-general classifier may displace other reliable classifiers, and the performance of the system can be worsened.

For this reason, Wilson introduced *accuracy-based LCSs* in 1995 [119], called XCS. In accuracy-based LCSs, the classifier's fitness is based on the accuracy with which it predicts the reward received from its interaction with the environment [56], rather than the prediction itself. Further, GA acts in environmental niches instead of on the whole population to maintain the parallel sustenance of equally important sub-solutions. This means that GA searches for the classifiers that are accurate in their prediction, independently from their prediction value, in order to select the classifiers [107].

More importantly, accuracy-based LCSs, e.g. XCS, attempt to evolve a complete map of all possible 'condition-action' rules for each possible level of reward, compared to strength-based LCSs (e.g. ZCS) which attempt to evolve a best action map. The *best action map* contains only the consistently correct rules. However, the *complete action map* contains both consistently correct and consistently incorrect rules. The rules that always receive the highest reward are termed *consistently correct* rules (i.e. rules that predict the correct class in all the inputs that they match). The rules with zero reward and low error correspond to *consistently incorrect* rules (i.e. rules that predict the incorrect class in all the inputs that they match). Therefore, having all rules in the action maps for both correct and incorrect rules enable accuracy-based LCSs able to address various ranges of problems promisingly (see Section 2.11). Table 2.1 describes the major differences between strength-based LCSs and accuracy-based LCS.

Table 2.1: Strength-based LCSs versus Accuracy-based LCSs.

Strength-based LCSs	**Accuracy-based LCSs**
Fitness is based on strength, which is an estimation of the reward the rules receive from the environment. Therefore, the GA searches for the best rewarded rules.	Fitness is based on the accuracy of the prediction, rather than on the prediction itself. This means that the GA searches for rules that are accurate in their prediction, independently from their prediction value.
Strength-based LCSs evolve a best action map, a map that contains only the high-rewarded rules.	Accuracy-based LCSs evolve a complete action map, a map that consists of all accurate rules belonging to the different payoff levels defined by the environment.
Strength-based LCSs perform poorly in the presence of multiple payoff levels, where the greedy classifier allocations assign higher reproductive opportunities to classifiers with higher rewards, that cause strong over-generalisation. Therefore the over-general classifiers are reproduced more often than other reliable classifiers that have low-rewarded states. As a consequence, over-general classifiers may displace other reliable classifiers, and the performance of the system can be worse.	Accuracy-based LCSs avoid these strong over-general effects by being based on the accuracy of the prediction, rather than the prediction itself.

2.8 Accuracy-Based LCSs

Accuracy-based LCSs, e.g. XCS, have been recognized as one of the main representative LCSs 'to date' [9], the most advanced and universal being 'Michigan-style' LCSs [115, 114]. The success factors of XCS are due to two main changes made to the previous LCSs architecture: the classifier's fitness is based on the accuracy of the prediction and 'niche-based' GA. XCS uses a form of restricted mating called 'niche-based' GA which focuses genetic search and provides a strong generalisation to the system [58]. Other interesting features in XCS include [58]: 1) *subsumption* that provides another bias towards generalisation rules, 2) *deletion and GA invocation* to balance rule allocation between niches, and 3) *macroclassifiers* which are classifiers with a numerosity parameter (parameter is used to indicate the number of identical virtual classifiers), which decreases the run-time and provides an important statistic on the worth of each unique rule.

2.8.1 XCS

XCS consists of three main components of LCSs: the Performance Component, the Reinforcement Component and the Discovery Component. There are three important classifier groupings in XCS: the population set [P], the match set [M] and the action set [A]. The population of classifiers (rules) is denoted by [P] and has a maximum number of classifiers. The match set [M] is formed from the current population set [P] and includes all classifiers that match the current input. The action set [A] is formed from the current match set [M] and includes all classifiers from the match set [M] which propose the executed action to the environment. The interaction process of the XCS's Performance Component is illustrated in Figure 2.17 and is described by the basic algorithm shown in Algorithm 5.

Algorithm 5: Algorithmic description of XCS (Performance Component) (adapted from [57]).

```
1  begin
2      Perceive a single input string (e.g. current state of the problem) from
       the environment.
3      Generate a random population of classifiers [P].
4      Build a match set [M] containing all the classifiers in the population [P],
       where the condition matches the input string.
5      if ([M] is empty or some of the classes are not predicted in [M]) then
6          Covering process is activated, a new classifier is created (where the
           condition is a generalized version of the input example, an action is
           the class that not covered in [M]).
7          Add this classifier into the population.
8      end
9      Calculate the prediction values for each action in the match set [M]
       based on p and F values of each classifier in [M] (where p is is an
       estimate of the expected amount of reward that the classifier will
       receive and F is the classifier's accuracy with respect to other
       classifiers).
10     Evaluate each action in the match set [M] based on the calculated
       prediction value.
11     Form the action set [A] containing the classifiers in match set [M] that
       will advocate action to the environment.
12     Send the selected action to the environment and receive a reward.
13     Activate the credit assignment algorithm for classifiers update.
14 end
```

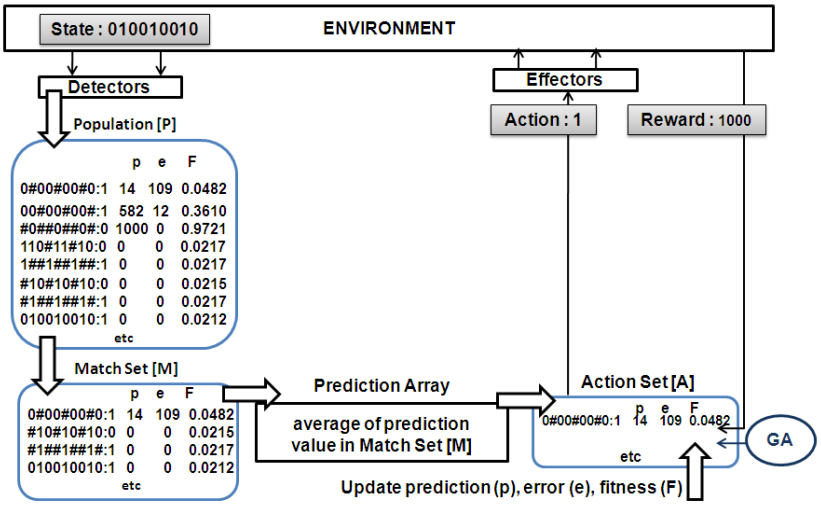


Figure 2.17: Schematic illustration of XCS (adapted from [119]).

2.8.1.1 Knowledge Representation

XCS evolves a population of classifiers [P] to represent the solution to a given problem. Knowledge is commonly represented in the form of $condition \rightarrow action$ rules and a set of parameters, called *classifiers*. Each classifier and its associated parameters are described in Table 2.2.

Classifiers may be created in one of three ways [57]: 1) random initial population, 2) covering, and 3) GA. Each of these methods is described as follows.

1. **Random initial population**
   This method allows the population to be initialised and filled with random classifiers. N classifiers are generated, each with random $condition$ and random $action$. Each bit in a classifier's condition is either a don't care symbol # with probability p#, '0', or '1'.

2. **Covering**
   In this method, the system starts with an empty population. Next, matching rules are produced through covering to fill the empty pop-

Table 2.2: Description of parameters listing in XCS [115, 114].

Notation	Description
The condition C	Specifies the input (state) in which the classifier is applicable (matches). It is commonly represented by fixed length *ternary* strings from {0, 1,#}, where 'don't care' symbol # matches both 0 and 1.
The action A	Specifies the selected action to be sent to the environment when the condition is satisfied. Represented by fixed length *binary* strings from $\{0, 1\}$.
The prediction p	Estimates the expected amount of reward that the classifier will receive for the specified condition and action it performed.
The prediction error ϵ	Estimates the error between the prediction p and the received payoff (reward).
The fitness F	Specifies the relative accuracy of the classifier or computed as inverse function of the prediction error (or the classifier's quality), which evaluates the classifier's accuracy with respect to other classifiers in the same action set [A].
The experience exp	Counts the number of times the classifier has occurred in the action set [A].
The time stamp ts	Denotes how many times the classifier has been evolved by GA in the action set [A].
The action set size as	Estimates the average size of the action set that the classifier belongs to.
The numerosity num	Counts the number of copies of the classifier in the population (i.e. macroclassifier). A macroclassifier is a classifier with a numerosity parameter, which indicates the number of identical virtual classifiers it represents. Instead of having n classifiers with identical conditions and actions, XCS stores a single macroclassifier with numerosity n.
β	Learning rate.
$p\#$	Probability of including don't cares # in the classifier's condition when the classifier is created through covering. A higher $p\#$ value means more general the classifiers are likely.
θ_{sub}	Subsumption trigger threshold, when a classifier is allowed to subsume another classifier when certain other conditions are met.
θ_{GA}	GA threshold to control the rate of applying GA to individual niches (i.e. action set [A]).
θ_{del}	Deletion threshold to delete the classifiers with low fitness.
χ	Probability of applying crossover to two parents.
μ	Probability of applying mutation to a bit in a condition or action.

ulation. When a classifier is created through covering, its condition is a copy of the current environmental input and it is given a randomly chosen action. Each character in the condition is then mutated with probability p# into a #. Covering allows the creation of new classifiers whilst guaranteeing that it matches the current input. Covering only occurs a few times at the beginning of the run to initialise the population.

3. **GA**
   XCS employs *niche-based* GA, which refers to the process of selecting parents from a subset of classifiers in the population (i.e. action set [A] or originally match set [M]) for reproduction. Copies of the classifiers are generated and then transformed using the standard genetic operators for creating the new classifiers. Crossover only occurs in the conditions, while mutation occurs in both the condition and the action.

2.8.1.2 Performance Component

The Performance Component in XCS works as follows:

1. At each *iteration*, the system obtains a single input x from the environment.

2. With the receipt of input x, the population of classifiers [P] is scanned.

3. Any classifier whose the condition matches the input x becomes a member of match set [M]. The classifier is considered to match the input when each symbol in its condition equals either the same symbol at the corresponding position of the input or a 'don't-care' symbol.

4. If the match set [M] is empty (i.e. none of the classifiers in the population [P] match the input), the covering operator is activated. The covering operator creates a new classifier with a matching condition

and a random action. With the addition of a new classifier, the existing classifiers must be removed from the population to keep N (the number of maximum classifiers) constant using any selection mechanism (i.e. tournament selection).

5. Once the match set [M] is formed, a system prediction array $P(a_i)$ is computed for each action a_i in the match set [M] using fitness-weighted average of the prediction of all classifiers in [M] that advocates a_i.

6. Next, the system evaluates each action in the match set [M] based on the prediction value from the prediction array $P(a_i)$ in order to select an action. The prediction array $P(a_i)$ stores the system prediction for each advocated action in preparation for action selection. If a specific action is not advocated (i.e. [M] is empty), then the covering operator is activated.

7. Next, the system forms the action set [A]. The action set [A] contains all classifiers from the match set [M] that proposed the executed action to the environment (i.e. subset classifiers with the highest prediction).

8. Next, the system selects an action to perform either using a simple *explore scheme*[1] or *exploit scheme*[2] alternately. Then, the selected action is performed in the environment and a reward is received in returned.

[1]In *explore scheme*, the system selects an action at random from those advocated by the matching rules; the system performs unbiased exploration from the available options (choosing the action randomly).

[2]In *exploit scheme*, the system deterministically selects the action which is most highly recommended by the matching rules; the system is maximally biased towards exploitation of its current knowledge (choosing the best action).

2.8.1.3 Reinforcement Component

Once the chosen action is sent to the environment, feedback is received and translated to a scalar reward r. Next, the reinforcement learning algorithm (credit assignment algorithm) is performed as described in Algorithm 6.

Algorithm 6: Algorithmic description of credit assignment algorithm in XCS (single-step task) (adapted from [57]).

```
1 begin
2     if (the previous time step's action set [A] is not empty) then
3         UPDATE [A] (see Section 2.8.1.3).
4         Do ACTION SET SUBSUMPTION in [A].
5         if (condition for GA invocation in [A] are met) then
6             Call RULE DISCOVERY algorithm in [A].
7         end
8     end
9 end
```

The parameters of the classifiers are updated with respect to the immediate feedback to the current action set [A]. The experience *exp* of all classifiers in the action set [A] is increased and other related parameters are updated. Parameter update is normally performed with the order: prediction (p), prediction error (E) and fitness (F) as follows [115, 114, 57].

1. Prediction p of each classifier in [A] (single-step task) is updated, given the immediate reward received r:

$$p_j \leftarrow p_j + \beta(r - p_j) \tag{2.3}$$

   Here, each classifier's prediction is being updated, where p_j is the prediction of classifier j, where β is a value controlling the learning rate (i.e. $0 < \beta < 1$).

2. Prediction error is updated :

$$\epsilon_j \leftarrow \epsilon_j + \beta(|r - p_j| - \epsilon_j) \tag{2.4}$$

Each classifier's prediction error is updated accordingly, where ϵ_j is the prediction error of classifier j.

3. The accuracy κ is derived:

$$\kappa_j = \{^{1}_{\alpha(\epsilon_j/\epsilon_0)^{-\nu}} \quad ^{if\ \epsilon<\epsilon_0}_{otherwise} \tag{2.5}$$

In order to calculate the classifier's fitness, the classifier's accuracy is firstly calculated, where ϵ_0 is a maximum error that a classifier can take to be considered as accurate, and α and ν are constants that control the rate of decline in accuracy. If the error is below the accuracy criterion (any rules with $\epsilon < \epsilon_0$), the classifier is considered to be fully accurate. Otherwise, the accuracy κ_j is a scaled of version of the error, where the classifier's accuracy drops off quickly as controlled by parameters α and ν when ϵ_0 is exceeded. The accuracy falloff rate is $0 < \beta < 1$ and the accuracy exponent is $\nu >0$.

4. The relative accuracy κ' for each classifier in [A] is computed:

$$\kappa'_j = \kappa_j \times numerosity(j) / \sum_{x \in [A]} \kappa_x \times numerosity(x) \tag{2.6}$$

Once the accuracy of all classifiers in [A] has being updated, each classifier's relative accuracy is calculated. It causes the sum of all relative accuracies of the classifier in [A] to equal 1. If the sum of the accuracies of the classifier in [A] is greater than 1, the relative accuracies of the classifier are less than their accuracies. Otherwise, the relative accuracies of the classifier are greater than their accuracies.

5. The fitness value of each classifier is updated from the relative accuracy κ':

$$F_j \leftarrow F_j + \beta(\kappa'_j - F_j) \tag{2.7}$$

Note that fitness is shared among the classifiers in the same action set [A] since it is calculated from the relative accuracies.

2.8.1.4 Discovery Component

In XCS, the GA (i.e. *niche-based* GA) is applied to the action set [A] with a frequency fixed by the parameter θ_{GA}, which refers to a process of selecting parents from the action set [A] (a subset of classifiers in the population) for reproduction as described in Algorithm 7.

Classifiers Subsumption is introduced as a way of biasing the system towards general (but still accurate) classifiers [57]. GA sub-sumption (in the version proposed by Martin Butz [14]) is activated for each of the offspring classifiers. Each offspring is checked for subsumption with its parents before it is added to the next generation [115, 114]. If the parent is sufficiently experienced, accurate and more general than the offspring, then the offspring is not introduced, instead the parents' numerosity is increased. However, if the offspring cannot be subsumed by the parents, than the offspring will be added to the population [P]. During the insertion, the offspring is compared to all individuals in [P] to check for any identical classifier. If an exact identical classifier is found, its numerosity is increased instead of adding the offspring.

In order to restore the number of classifiers to N, an appropriate number of classifiers need to be removed from the next generation. If a population size is greater than the maximum value N, a deletion process is performed with a probability proportional to an estimate of the size of the action set as. If the classifier is sufficiently experienced and its fitness F is significantly lower than the average fitness of the classifiers in [P], its chance to be deleted is increased.

Algorithm 7: Algorithmic description of rule discovery in XCS (single-step task) (adapted from [57]).

```
1  begin
2      Reset GA counters of classifiers in [A].
3      SELECT two parents P1 and P2 from [A].
4      CROSS P1 and P2 with probability χ otherwise clone to obtain C1 and
       C2.
5      MUTATE each bit in C1 and C2 with probability μ.
6      Initialise parameters of C1 and C2.
7      DELETE classifiers as needed.
8      if (C1 subsumed by P1 or P2) then
9          Increment numerosity of subsuming parent.
10     end
11     else
12         Insert C1 into [P].
13     end
14     if (C2 subsumed by P1 or P2) then
15         Increment numerosity of subsuming parent.
16     end
17     else
18         Insert C2 into [P].
19     end
20 end
```

2.8.2 XCSR

Accuracy-based Learning Classifier Systems with real-value, XCSR, was introduced later by Wilson [120]. XCSR enhanced XCS to real inputs (i.e. integer and real-valued problem domains). The representation of XCSR is illustrated in Figure 2.18, while the changes from XCS to XCSR [120] in respect to the classifier's representation is as follows.

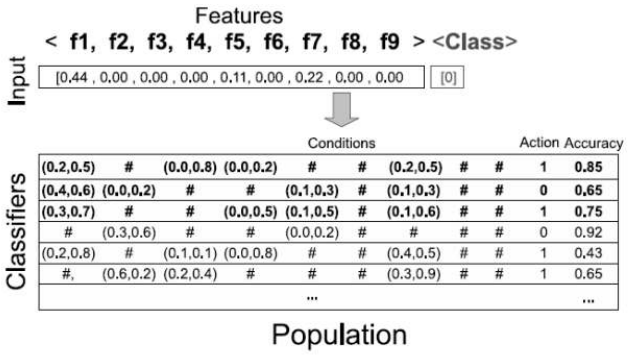


Figure 2.18: Schematic illustration of XCSR (adapted from [76]).

2.8.2.1 Knowledge Representation

XCSR evolves a population of classifiers [P] to represent the solution to a given problem. Each of the N classifiers in the population consists of $condition \rightarrow action$ rule and a set of parameters as follows:

1. The condition C specifies a problem domain by interval encoding within the interval $C = (l_1, u_1, l_2, u_2, ..., l_i, u_i)$; where l_i is a *lower bound* and u_i is an *upper bound* of a real-value number. A classifier matches an input x if and only if $l_i \leq x_i < u_i$ for all x_i.

2. The action part A specifies an available or specified action that the classifier proposes, $A \in a$, where $a = \{a_1, ..., a_m\}$ is the set of all possible actions in the problem.

3. The prediction p estimates the payoff the system will receive if condition C_j matches and its action is chosen by the system.

4. The prediction error ϵ estimates the error in p_j with respect to actual payoffs received.

5. The fitness F specifies the relative accuracy of the classifier.

2.8.2.2 Performance Component

The performance component in XCSR works in a similar way to XCS. The main difference (i.e. on how the input is perceived) is described as follows.

1. XCSR is initialized with an empty population, where initial classifiers are generated by a covering mechanism that creates intervals controlled by parameter r_0. When a new covered classifier is created, each interval $int_i = (l_i, u_i)$ is generated as $l_i = x_i - rand(r_0)$ and $u_i = x_i + rand(r_0)$, where $rand(r_0)$ is a value uniform randomly chosen from $[0, r_0]$ and r_0 is a real constant [49].

2. At the beginning of each iteration, the system detects the current problem instance x.

3. With the receipt of the instance, the population of classifiers [P] is scanned.

4. Any classifier in the current population [P] whose conditions are satisfied by x becomes a member of the match set [M]. The classifier is considered to match the input when x lies in each of the hyperrectangles of all matching classifiers; a classifier matches an input message x if each element x_i belongs to the corresponding interval in C within the lower bound and upper bound range $l_i \leq x_i < u_i$. Every possible action should be represented by at least one classifier in the match set [M].

The Reinforcement Component and the Discovery Component of XCSR also work similarly to XCS as described in the previous section.

2.8.3 UCS

UCS [9] is a learning classifier system derived from XCS that works under a supervised learning scheme [87]. As UCS is specifically designed for *supervised learning*, it benefits directly from known labels during training [105]. The main difference between UCS and XCS is as follows [87]. First, the Performance Component is adjusted based on a supervised learning scheme. Thus, UCS only evolves high-rewarded classifiers (i.e. the best action map). Secondly, UCS calculates accuracy as the percentage of correct classification.

UCS uses the same classifiers representation as in XCS, and 'niche-based' GA is implemented as the main search mechanism. However, niches in UCS are defined by the correct rule-set [C], and it is expected that UCS will generalize over the search space of correct rule-sets [9]. The changes from XCS to UCS are discussed in this section, where the major differences are in the Performance Component [9, 105]. The Reinforcement Component and the Discovery Component of the system work as described in XCS in the previous section.

2.8.3.1 Knowledge Representation

UCS evolves a population of classifiers [P] to represent the solution to a given problem. Each classifier is represented in the form of $condition \rightarrow action$ rules and a set of parameters, called *classifiers*. The main parameters of the classifiers are: 1) acc, the accuracy of the classifier, 2) F, the fitness of the rule which is based on the accuracy acc, 3) exp, the experience of the classifier, 4) the niche size cs, 5) the last time of the GA activation ts, and 6) the numerosity num, which denotes the number of copies of the classifiers in the population.

2.8.3.2 Performance Component

UCS is a supervised learner, thus learning is performed using a supervised learning scheme. During learning, UCS perceives a set of labelled examples where each instance $x = (x_1, ..., x_n)$ and has a corresponding class c. Therefore, UCS is being presented with an input example together with the associated class, $x : c$.

During *learning*, UCS works as follows [87]:

1. At each iteration, an input example x with the associated class c is presented to the system.
2. With the receipt of input x, the population [P] is scanned.
3. Next, the system creates the match set [M] consisting of those classifiers whose conditions match the input x.
4. Those classifiers in the match set [M] which predict the class c form the correct set [C].
5. If the correct set [C] is empty, the covering operator is activated, creating a new classifier with a generalized condition matching x with random # and predicting class c.

During *testing*, UCS works as follows [87].

1. At each iteration, only input example x is presented to the system. Thus, UCS must predict its associated class c.
2. With the receipt of input x, the population [P] is scanned.
3. Any rule whose condition matches the input x becomes a member of the match set [M].
4. Once the match set [M] is formed, the system selects the best class from the vote (weighted by fitness) of all classifiers in the match set [M] (i.e. chosen the most-voted class).

At each time the classifier participates in the match set [M] in the learning mode, the classifier parameters are updated as follows [86, 87]. The accuracy acc is computed as the proportion of correct classification:

$$acc = \frac{NumberOfCorrectClassification}{exp} \tag{2.8}$$

Note, since the classifiers' parameters are updated in the match set [M], the classifiers' experience exp is increased by one every time the classifier participates in the match set [M]. The niche size cs stores the average number of classifiers that participates in the correct set [C]. The niche size cs is updated whenever the classifier belongs to the correct set [C]. The rest of the parameters in UCS (i.e. the accuracy κ, the relative accuracy κ' and the fitness F) are updated in the same way as in XCS.

2.8.3.3 Discovery Component

In UCS, the GA is used as the search mechanism similar to that of XCS (i.e. *niche-based* GA), where GA is applied to the correct set [C]. The system selects two parents from the correct set [C] (a subset of classifiers in the population) for reproduction. The rest of the process works similarly to XCS.

2.9 A-PLUS

Accuracy-based Pittsburgh Learner Using Subsumption (A-PLUS) is a version of Pittsburgh-style LCSs, which incorporated the *rule subsumption deletion* and *inaccurate rule deletion* from XCS into its method. In [109], Stacey implemented 'XCS updates' into A-PLUS, in such a way that the system can reduce the execution time by replacing the specific rules with more general rules and only delete inaccurate rules. The main structure of A-PLUS follows Pittsburgh-style LCSs descending from GABIL (GA Batch-Incremental concept Learner) [53].

2.9.1 Knowledge Representation

In Pittsburgh-style LCSs, knowledge is represented by a *population of rule-sets*. An individual is a *rule-set* (i.e. a variable-length set of rules). Each rule (classifier) has a fixed length, but the number of classifiers of the set is variable. Here, a *classifier* is in the form of *condition-action* rules and a set of parameters. The *condition* is a conjunction of terms, where each of them is related to an attribute of the input instance, whilst the *action* is an associated class. For example, the condition (i.e. nominal and real-valued attributes) can be represented as follows [1].
$attribute_i$ is equal to $value^i_j$
$attribute_i$ is equal to $value^i_{j1}$ or $value^i_{j2}$
$attribute_i$ belongs to interval $[low, high]$
$attribute_i$ is lower than $value$
$attribute_i$ is higher than $value$

A-PLUS used a method similar in GABIL [53] to represent the nominal attributes as follows [1, 125]. Each individual I is a variable-length set of classifiers, where each rule R_i includes the condition part and the corresponding action.

$$I = (R_1 \vee R_2 \vee ... \vee R_n) \tag{2.9}$$

The rule R_i can defined as:

$$(V^1_1 \vee ... \vee V^1_{k_1}) \wedge (V^2_1 \vee ... \vee V^2_{k_2}) \wedge ... \wedge (V^n_1 \vee ... \vee V^n_{k_n}) \rightarrow classification \tag{2.10}$$

where the condition part of each rule is a conjunctive normal form (CNF), V^j_k denotes the k^{th} value of the j^{th} feature. The rule is triggered when the value of the j^{th} feature in the input is equal to one element in $V^j_1 \vee ... \vee V^j_k$.

The *condition* can be mapped into a *binary string* in the following way. For example, there are three features {A, B, C}. Each feature can take any value of {A1, A2}, {B1, B2, B3} and {C1, C2, C3, C4}. One bit represents

each available value for each feature, and if the feature has a value in the condition part then the corresponding bit is set to '1', or else '0'. Therefore the condition {A has value A1 or A2}, {B has value B3} and {C has value C1 or C4} can be represented as '110011001'.

The *condition* can also be represented by the *real-value* attributes using real-value intervals as follows [1]. Each individual is a variable-length set of rules:

$$A_1 \in [l_1, u_1] \wedge A_2 \in [l_2, u_2] ..... \wedge A_n \in [l_n, u_n] \rightarrow classc_m \tag{2.11}$$

Where A_i is an attribute or feature of the domain and l_i, u_i are the lower and upper bounds of an interval associated to the attribute A_i. The values of l_i and u_i are calculated using the method as in the XCSR system [120], where the interval is codified as a pair of real values defined by the centre and spread. The lower bound of the interval is defined as $centre - spread$, while the higher bound of the interval is defined as $centre + spread$.

2.9.2 Rule-Set Evaluation

Here, A-PLUS is designed as an on-line system. At each iteration, each rule-set is initialized based on the input example and creates a number of classifiers that classifies the input example correctly . The rule-set is evaluated based on the previous performance of each classifier in the set. The simplest approach of examining the past experience of the classifier and computing its accuracy is as follows (similar to equation 2.8):

$$accuracy = C/T \tag{2.12}$$

Where C is the number of correctly classified instances, and T is the total number of instances that the classifier matched.

Next, each classifier in the rule-set is evaluated. First, the classifier is assigned an accuracy value (i.e. the number of correctly classified instances over the total number of instances that match). Secondly, the average accuracy of the classifiers in each action set [A] is calculated, where the action

set is defined as a set of matched classifiers that correctly classify a given data instance. Thirdly, the mean accuracies are added and finally divided by the total number of data instances, giving an overall fitness value for the rule-set, where fitness is based on the rule-set accuracy. Thus, the rule-set with a higher number of accurate classifiers in each action set will have a higher fitness and ultimately be favoured for reproduction. A-PLUS system is executed as in Algorithm 8.

2.9.3 Rule-Set Evolution

A-PLUS applies a standard GA to evolve individuals (rule-sets) by operating at the level of a single rule-set. The genetic operators which apply to the system are limited to *selection*, *crossover* and *mutation*. In order to restore the number of rule-sets to N, an appropriate number of rule-sets needs to be removed from the next generation. If a population size is greater than the maximum value N, a deletion process is performed.

2.9.3.1 Selection

A-PLUS used the *tournament selection* as suggested by Butz [14]. The individuals compete against each other and only one individual is chosen to reproduce (i.e. the fittest individual wins the tournament). The tournament selection depends on the individual's rank rather than the relative fitness. The selection is therefore not affected by the fitness distribution through the population [22], which makes the individuals in the tournament selection converge faster.

2.9.3.2 Crossover

In A-PLUS, an individual is a population of rule-sets, where each rule-set has a fixed length with a variable number of classifiers in the set. Since all classifiers have the same length, the crossover point can be set to any position of the string and is not restricted to rule boundaries. A-PLUS

Algorithm 8: Algorithmic description of A-PLUS (Accuracy-based Pittsburgh Learner Using Subsumption) (adapted from [109]).

```
1  begin
2      for each ruleSet do
3          ruleSet.meanActionSetSize=0
4          ruleSet.rawFitness=0
5          for each classifier do
6              classifier.meanActionSetSize=0
7              classifier.meanActionSetAccuracy=0
8          end
9          for each classifier do
10             for each dataInstance do
11                 if classifier is matched then
12                     classifier.match++
13                     if classifier.action==dataInstance.class then
14                         classifier.correct++
15                     end
16                 end
17                 classifier.accuracy=classifier.correct/classifier.match
18                 classifier.numActionSet= classifier.correct
19             end
20         end
21         for each dataInstance do
22             Empty the action set
23             for each classifier do
24                 if classifier is matched AND classifier.action==dataInstance.class then
25                     Add classifier to ACTION-SET
26                 end
27             end
28             actionSetAccuracy=0
29             for each classifier in ACTION-SET do
30                 actionSetAccuracy += classifier.accuracy
31                 actionSetAccuracy /= number of classifiers in ACTION-SET
32             end
33             for each classifier in ACTION-SET do
34                 classifier.meanActionSetSize += number of classifiers in ACTION-SET
35                 classifier.meanActionSetAccuracy += actionSetAccuracy
36             end
37             ruleSet.meanActionSetSize += number of classifiers in ACTION-SET
38             ruleSet.rawFitness += actionSetAccuracy
39             carry out SUBSUMPTION-DELETION
40             ruleSet.rawFitness /= number of data instances
41             ruleSet.meanActionSetSize /= number of data instances
42             carry out INACCURACY-DELETION
43         end
44     end
45 end
```

uses *two-point crossover* on each parent, where each point is aligned with the corresponding point on the other parent and swaps the segments lying between the cut points. There are a few reasons for choosing this method [109]. First, it is insufficient to choose the same crossover point for both parents, since the individuals may be of different lengths and the chosen point may not exist on one parent. Secondly, the crossover points cannot be chosen arbitrarily on both parents, as this may result in the creation of invalid classifiers with too many or not enough attribute values. The crossover method applied to A-PLUS is performed as in Table 2.3.

Table 2.3: A-PLUS crossover (adapted from [109]).

Method	Description
Two-Point Crossover	$RuleSetParent_1$, $RuleSetParent_2$
Precondition	none
Postcondition	selects two random crossover points on each 'parent' rule-set, swaps the middle segments, and replaces the parents with the resulting 'child' rule-set.

2.9.3.3 Mutation

A-PLUS uses a *standard mutation* with a parameter specifying the probability of the operator's application [109]. A new value is always chosen when the mutation function is called. If the gene currently has value '0' or '1', then it may either be flipped within a condition string or mutated into a 'don't care'. If a 'don't care' is to be mutated, it has an equal chance of becoming '0' or '1'. The mutation method applied to A-PLUS is performed as in Table 2.4.

Table 2.4: A-PLUS mutation (adapted from [109]).

Method	Description
Mutation	rule-set, integer
Precondition	integer supplied must be a valid position within the rule-set.
Postcondition	assigns a randomly chosen alternative value to the specified bit.

2.9.4 XCS component in A-PLUS

The individuals in Pittsburgh-style LCSs are commonly encoded as a variable-length set of rules. Thus, bloat may be a problem to the system. This problem is related to the unlimited growth of the size of the individuals [4]. In [109], it is hypothesised that the problem could be resolved by evaluating individual classifiers and removing any inaccurate or unnecessary specific classifiers. Therefore, *rule subsumption deletion* and *inaccurate rule deletion* methods in XCS have been adapted to A-PLUS in order to control the bloat effect. The methods were successfully employed within XCS [119], but had not previously been tried in any Pittsburgh-style LCSs. Here, the methods applied to A-PLUS are described in the following way [109].

2.9.4.1 Subsumption Deletion

This method is performed similarly to XCS. However, A-PLUS includes the *action set subsumption*. A-PLUS consists of an action set [A], which refers to a set of matched classifiers that correctly classify a given data instance. There will be a corresponding action set for each instance in the training set, though some of these action sets might be empty. The system searches for the most general and accurate classifier in the current action set. If there is one maximally general and accurate classifier in the action set [A], then all classifiers which are accurate but more specific (can be

subsumed by it) will be removed. Rule subsumption deletion is performed as in Algorithm 9.

Algorithm 9: Algorithmic description of subsumption deletion in A-PLUS (adapted from [109]).

```
1  begin
2      highestGenerality=0
3      index=-1
4      for each classifier in ACTION-SET do
5          if classifier.accuracy==1 AND classifier.numActionSets>1 AND
           classifier.generality>highestGenerality then
6              highestGenerality = classifier.generality
7              index = classifier index
8          end
9      end
10     if index>-1 then
11         for each classifier in ACTION-SET do
12             if classifier.accuracy==1 AND classifier subsumed by
               classifier[index] AND randomNumber<subsumptionProbability
               then
13                 DELETE classifier
14             end
15         end
16     end
17 end
```

2.9.4.2 Inaccuracy Deletion

This method is performed when the classifier is not a member of any action set [A], since it is considered completely inaccurate (i.e. always advocating the wrong classification). In addition, other classifiers are removed as well (i.e. when the classifier's accuracy is below the average accuracy of the classifiers belonging to the same action set). Inaccuracy deletion is executed as shown in Algorithm 10.

Algorithm 10: Algorithmic description of inaccuracy deletion in A-PLUS (adapted from [109]).

```
1  begin
2      for each classifier do
3          if classifier.numActionSets==0 then
4              DELETE classifier
5          end
6          else
7              classifier.meanActionSetSize /= classifier.numActionSets
8              classifier.meanActionSetAccuracy /= classifier.numActionSets
9              if classifier.accuracy<classifier.meanActionSetAccuracy then
10                 if randomNumber<deletionProbability AND
                   classifier.meanActionSetSize≥ruleSet.meanActionSetSize then
11                     DELETE classifier
12                 end
13             end
14         end
15     end
16 end
```

2.10 Tabu Search

Tabu Search (TS) is a neighborhood search method proposed by Glover in 1986 [77]. TS is a 'higher level' heuristic procedure for solving optimization problems, designed to guide other methods (or their component processes) to escape the trap of local optimality [30]. TS uses [31, 32]: 1) flexible memory structures to permit search information to be exploited more thoroughly than by a rigid memory system or a memoryless system, 2) conditions for strategically constraining and freeing the search process (embodied in tabu restrictions and aspiration criterion) and 3) memory functions of varying time spans, from short-term to long-term memory for intensifying and diversifying the search (reinforcing attributes historically found good and driving the search into new regions).

A *memory* (i.e. short-term memory) forces TS to explore a new area of the search space that seeks to make the best move if possible and subject to available choices to satisfy certain constraints. These constraints (embodied in the tabu restrictions) are designed to prevent the reversal (or repetition) of certain moves by rendering the selected attributes of these moves forbidden (tabu) [32]. A known number of solutions that have been examined recently become tabu and cannot be selected when searching for the next solution and they are stored in a memory called the *tabu list* [77].

The basic principle of TS is to improve local search whenever it encounters a local optima by allowing non-improving moves, as cycling back to previously visited better solutions in the tabu list is forbidden [29]. The tabu list records the recent history of the search, essentially the value of the objective function $f(i^*)$ of the best solution i^*, and also keeps information on the trajectory through the last solutions visited [29]. When a new candidate solution is introduced , it goes into the tabu list and it is made tabu for a certain number of iterations (time).

One way of creating the tabu restrictions is to assign a maximum iteration for each candidate solution in the tabu list. For example, each can-

didate solution has a maximum 'tabu tenure' (iteration value assigned to it, e.g. the maximum iteration value is 5). The candidate solution can be revisited again only when the maximum iteration value is 0. For each iteration, all the non-zero values (i.e. the maximum iteration value of the candidate solution) in are decremented by 1. This means, this candidate solution cannot be revisited for the next 4 iterations. Thus, TS allows escaping from sub-optima solutions by improving the efficiency of the exploration process. TS is performed as described in Algorithm 11.

Algorithm 11: Algorithmic description of Tabu Search (adapted from [29]).

```
1  begin
2      s ← s_0 : create an initial solution.
3      BestSolution ← s.
4      TabuList ← null.
5      while (not stopping condition) do
6          Find the best neighbor of the current solution by applying an
           allowed move (non-tabu move).
7          if (a given criteria is meet) then
8              CandidateSolution ← accept as the new current solution.
9          end
10         else
11             CandidateSolution ← find another neighbor (best non-tabu
               neighbor).
12         end
13         if (fitness CandidateSolution greater than fitness BestSolution) then
14             BestSolution ← CandidateSolution.
15             while (TabuList size greater than maximum TabuList size) do
16                 ExpireFeatures(TabuList).
17             end
18         end
19     end
20     return BestSolution : globally best solution found.
21 end
```

2.11 LCSs Applied to Pattern Classification

LCSs appear to be a widely applicable cognitive (agent) model that can be implemented as a framework for a diversity of learning investigations and practical applications [46, 9]. The important work of LCSs apply to multi-step [67], modifications for non-Markov and Markov environments [69, 68], incorporation of continuous-valued actions [47], function approximation problem [37, 64, 121, 123, 15], boolean applications [66], and many others. LCSs have also been applied to various applications in the areas of data analysis and data mining [122, 3, 106], pattern recognition [113, 126, 16, 91], robotics [80, 55], classification [9, 99, 113] and computational economics [110]. Here, we review recent research that applied LCSs to the area of pattern recognition and classification.

In [124], Wilson proposed the *Three-Cornered Coevolution Framework*, which is a theoretical model of an automated image-transformation program. The framework provides a new model of *human-independent system* to address the pattern recognition problem. The Three-Cornered Coevolution Framework consists of three different agents that interact with one another to adapt with and drive the changes of the problems. In this framework, the pattern recognizers evolve to learn each set of the patterns, while the pattern generator evolves autonomously to create various problems with different levels of difficulty for classification. It is proposed that the pattern generator is able to adjust the difficulty of the problem being addressed. However, this theoretical framework had not yet been implemented and tested. There were several issues identified in the paper as follows. First, is the coevolutionary framework relevant to the way natural patterns form? Secondly, if the framework functions as a pattern recognizer, will the system evolve similar methods to human saccades? Thirdly, how well will the framework drive the coevolution in the system? Finally, if the framework works, will the results have wider relevance than image classification? Therefore, it was unclear how well the framework would

work to drive the coevolution among the agents in the system.

A number of GBML models were studied by Orriols-Puig et al. [91] on a collection of 20 real-world datasets extracted from the UCI repository and local repositories in order to determine whether or not the LCS models were competitive for pattern recognition [91]. First, the paper reviewed the most relevant GBML models for pattern recognition: non-fuzzy rule representations (UCS, GAssist, HIDER, HMOF) and fuzzy rule representations (SLAVE, Fuzzy LogitBoost). The six GBML models were compared in terms of their accuracy and readability. UCS (the Michigan-style GBML model for supervised learning) appeared to be the best model as it obtained high accuracy, while SLAVE (fuzzy iterative rule learning approach) appeared to be the best alternative model that obtained high interpretability.

Secondly, the six GBML models were compared with a number machine learning techniques such as C4.5, IBK, Naive Bayes, PART and the support vector machine, SMO. UCS appeared to be the most accurate model among GBML models. UCS evolved the largest rule-set among all the models. However, the larger number of rules in UCS may delay the interpretability of the models. Thus, some reduction techniques were required to remove non-useful rules from the final population in UCS. GAssist (Pittsburgh-style GBML) and HMOF (Pittsburgh-style fuzzy Genetic Fuzzy Rule-based Systems) resulted in more readable models compared to the remaining machine learning techniques, while maintaining statistically equivalent test accuracy. The analysis showed that GBML models were competent for classification and UCS was shown to be one of the best learners. The paper also revealed the strengths and the weaknesses of GBML models. It can be used as a guideline for choosing any GBML model to apply to a problem on hand depending on the overall goal (i.e. either to maximize the accuracy or the interpretability). All these results may encourage a researcher to consider GBML models to apply to other classification and pattern recognition problems.

In [9], the performance of two accuracy-based LCSs (i.e. XCS and UCS) in different types of classification problems was investigated by Bernado-Mansilla, et al. XCS, as is standard, used the reinforcement learning scheme, while UCS used the supervised learning scheme. First, the models were tested on three artificial problems (i.e. binary class, multi-class, and multi-class problem with different proportions of examples per class (class imbalance problem)) in order to understand the behaviour of the models related to the problem's characteristics. Next, the models were further tested with a set of real world classification problems from the UCI data sets and compared to well-known learning algorithms such as ZeroR, IB1, IBk, Naive Bayes, C4.5, PART and SMO. The accuracy rate of each learning model is used as the metric of performance. The results showed that the accuracy-based LCSs were competitive with respect to other learning algorithms. UCS and XCS evolved accurate generalizations of best action maps that consist of both correct and incorrect rules which helped the models to correctly predict the class. The paper also suggested open issues for further improvement such as using reduction techniques that may minimize the number of rules during training in the models.

In [99], Hybrid C4.5 (decision tree induction) and UCS (accuracy-based LCSs for supervised learning) were tested on eight multi-category classifications of real world data sets from the UCI data repository, where the quality of the rule-sets of these two algorithms were evaluated. The accuracy rate of each learning model was used as the metric of performance. The results showed UCS had significantly better performance for six data sets (Glass, Iris, Heart, Wine, Soybean and Vote), compared to C4.5 that achieved less accuracy for all data sets except Soybean. UCS achieved 52% accuracy rate, whereas C4.5 achieved 91.4% accuracy rate for the Soybean data set. In the Soybean data set, there were 35 categorical attributes and 19 classes. The authors strongly suggested applying the accuracy-based LCSs (UCS) to address other supervised learning problems.

Six well known evolutionary rule learning algorithms, XCS, UCS, GAs-

sist, cAnt-Miner, SLAVE and Ishibuchi, were investigated by Tanwani, et al. on 31 publicly available biomedical datasets [113]. The results showed that GAssist (Pittsburgh-style LCSs) gave better classification results on the majority of biomedical datasets among the compared schemes. The greater accuracy was a result of its superior fitness function that used the Minimum Description Length (MDL) principle to evolve optimum rules. However, the results suggested that UCS and XCS were effective in identifying hidden patterns and generating information rich rules compared to GAssist and cAntMiner that produced simple and generic rules. They strongly recommended that other medical experts refine XCS and UCS rules for knowledge extraction. They found that the classifier's accuracy depends on the complexity of a dataset. They also quantified the complexity of a biomedical dataset in nature (i.e. missing values, imbalance ratio, noise and information gain) which played a major role in determining the classification accuracy of the datasets. They proposed a meta-classification model that could be used for determining the problem's difficulty.

From the above discussion, the most significant limitations of the related work are as follows:

1. Little work has been conducted on artificial problem generators and even less in a coevolutionary framework for problem generation in LCSs. Usually, a classification system's performance is assessed on different sets of data, commonly from public repositories. Few works have been proposed investigating the classification system's performance via artificial datasets such as in [71], [38], [72], [73] and [103]. However, the complexity factor is limited for the creation of artificial datasets at various levels of complexity.

2. LCSs are still lacking in complete theoretical frameworks of its implementation in certain areas [124], and only few have been developed, for example, facet-wise analysis [12].

3. LCSs still have low visibility in the machine learning community.

Recent advances on both the GBML theory and applications may help encourage GBML techniques to be used as competent, accurate, and reliable machine learning systems [91].

4. Most LCSs suffer from higher training times, which is associated with the learning algorithm required to evolve accurate generalization of a complete action map, which consists of both correct classification and incorrect classification for each problem sub-solution [91, 9]. For example, the most competent GBML approaches tested in [91] had a higher average run-time compared to C4.5 rule induction algorithm.

5. Most LCSs consume larger computational resources (e.g. CPU time) to evolve accurate generalized rules of complete action maps, which consist of both correct and incorrect rules. This mostly happened in a large search space domain where larger populations were needed and more learning cycles were required [91, 9].

6. Most LCSs have a large number of configuration parameters that need to be tuned and configured in the design of the experiment before any experiment can be performed.

2.12 Summary and Way Forward

This chapter covers concepts and background materials in the fields of artificial intelligence (AI) and machine learning (ML) where the work for this thesis is applied. A general description of the machine learning paradigms (i.e. supervised versus reinforcement learning) and concepts related to the work in particular to address classification problems is presented. Next, a description of evolutionary computation (EC) and evolutionary algorithm (EA), specifically genetic algorithm (GA) is provided. However, GA is implemented as 'niche-based GA'. The concept of coevolution that applies to

the work is also described. Special focus is given to GBML systems, LCSs, where four models of LCSs related to the implementation of the work are elaborated (i.e. XCS, XCSR, UCS, A-PLUS). In order to enhance the capability of the LCSs model, Tabu Search is introduced. A literature review of existing techniques relating to our work especially pattern recognition and classification is also presented.

Some remarks regarding this line of research are as follows. Learning from a set of examples that have desired features using any learning algorithm (e.g. either supervised or unsupervised learning) is crucial and important for most pattern recognition systems. In this domain, several machine learning techniques, such as GBML systems, have been designed to address various classification problems. In the last few years, a public repository (i.e. the UCI repository), has become the most used resource to obtain datasets for classification. The results reported in the literature have confirmed that GBML systems were competitive for classification. The Three-Cornered Coevolution Framework provides a new interesting model of *human-independent system* to address the pattern recognition problem differently. Research in this direction is relatively new, especially 'coevolution LCSs' and many aspects still need to be explored.

The work of *Three-Cornered Coevolution System* is not to identify an ideal implementation, but to provide a new implementation choice (i.e. a new *coevolution system*) that can be used in a comparison with other LCS implementations. The work for this thesis is focused on an autonomous *classification-problem generation approach* for generating various classification problems, where the problem's difficulty can be tuned and adjusted automatically whilst lowering human involvement. The idea here is to tune the problem's difficulty autonomously so that the problem's characteristics may be determined effectively to empirically test the learning bounds of the learners. As the learners learn and adapt with the changes of the problems using different types of learning technique (e.g. supervised learning technique or reinforcement learning technique), a *coevolu-*

tionary approach (i.e. *coadaptive evolutionary* approach) will be activated in order for the system to create various problems for classification based on the learners' ability to learn. The methodology of the system will be introduced in the next chapter.

Chapter 3

Methodology

The overall aim of the work for the thesis is to design and develop the new *Three-Cornered Coevolution System* for addressing the classification tasks. This chapter details the methodology used to develop the system. The methodology consists of three main phases as follows.

1. Phase 1: An Evolvable Problem Generator.

2. Phase 2: Two-Cornered Coevolution System.

3. Phase 3: Three-Cornered Coevolution System.

Phase 1 is necessary to establish an appropriate problem domain for classification that can be evolved and tuned automatically. Therefore, an automated evolvable problem generator is developed for creating two different problem domains for classification (i.e. image-based data and artificial data). The created problem domain is to be flexible in order to be tuned and adjusted (e.g. to make it either a 'hard' or 'easy' problem). Here, the generation agent evolves autonomously for generating different types of image-based data (or artificial data) for the classification task, while the classification agent evolves for classifying image-based data (or artificial data) correctly.

Phase 2 is needed to investigate the generation agent's ability to autonomously tune and adjust the problem's difficulty based on the classification agent's performance. The generation agent must either increase or decrease the problem's difficulty (i.e. either to maximize or minimize the classification agent's performance). Thus, the ability of the generation agent to find an appropriate level of the problem's difficulty is crucial to ensure both agents work in a coevolutionary manner. This phase is important for establishing a baseline for a coevolution system. Phase 2 is a *standard coevolution system, where two different agents evolve to adapt to and drive the changes of the problem*. Here, the classification agent evolves to learn various classification problems, while the generation agent coevolves to tune and adjust the problem's difficulty based on the classification agent's ability to learn.

Phase 3 is the final research goal and the core principle of the research. This phase is a *new coevolution system* where three different agents evolve to adapt to and drive the changes of the problem. Here, two classification agents evolve to learn various classification problems, while the generation agent coevolves to tune and adjust the problem's difficulty based on the classification agents' ability to learn. The overall system will implement the Three-Cornered Coevolution in which each agent autonomously evolves to adapt with the changes of the problem, hence lowering human involvement for setting and tuning the system.

3.1 Phase 1: An Evolvable Problem Generator

The overall aim of this phase is to develop an *automated evolvable problem generator* for generating various problems for classification. The generation agent (i.e. the automated evolvable problem generator termed as the Sender (S)) needs to autonomously evolve the problem and generate different types of image-based data (or artificial data) for classification, while the classification agent (i.e. termed as the Receiver (R) which is an accuracy-based LCS in this case) learns and evolves to learn the image-based data (or artificial data).

3.1.1 Research Objectives

This overall aim can be broken down into the following objectives:

- Develop an *evolvable problem generator* for generating different types of problems for classification, where each problem domain can be created and manipulated autonomously (i.e. scalable and evolvable image-based data or artificial data).
- Test the classification agent on various classification problems to determine limits of the classification agent's performance. The created problem domain should be able to explore the classification agent's ability to work on various classification problems (i.e. ranging from an 'easy' to a 'hard' problem).
- Evaluate the classification agent's performance for learning various classification problems. The system should be able to scale and evolve the problem domains for generating various classification problems to be learned by the classification agent.
- Investigate the classification agent's performance for learning various classification problems, so that an appropriate problem domain develops as a baseline for further enhancement in the next phase of

development. The created problem domain should be flexible to be tuned and adjusted either to make it 'hard' or 'easy' for the classification agent to learn.

- Verify the system as a baseline for generating various problems at different levels of difficulty for classification.

3.1.2 Framework Design

The system consists of two main agents: an automated evolvable problem generator (i.e. the Sender (S)) and the classification agent (i.e. the Receiver (R)) (see Figure 3.1). S is a program to generate various problems for classification, while R is a program to learn the generated image-based data (or artificial data) based on accuracy-based LCS (i.e. XCS or XCSR). The accuracy-based LCS was chosen to be applied to the classification agent as it has been recognized as one of the main representative LCSs 'to date' [9]; the most advanced and universal 'Michigan-style' LCSs [115, 114]. Both agents will evolve using evolutionary computation (i.e. Genetic Algorithm).

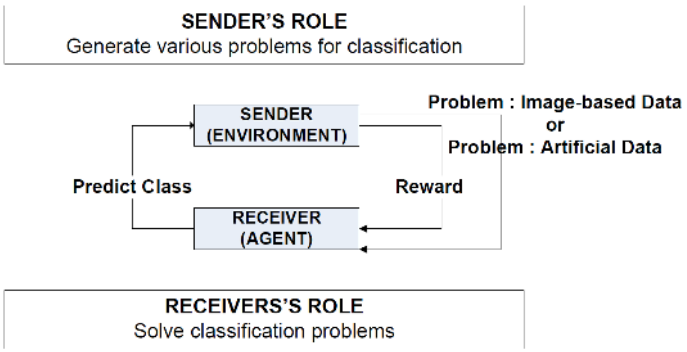


Figure 3.1: An evolvable problem generator.

Figure 3.1 illustrates the overall design of the system. First, S generates variants of problems for classification (i.e. a population of problems referred to as a *meta-problem*) and the associated instance for each individual problem 'on-the-fly' (i.e. image-based data or artificial data). Secondly, S sends each instance of the individual problem to R in turn (i.e. one iteration for S is the individual problem, and one iteration for R is an instance from an individual problem). R is developed based on the accuracy-based LCS (i.e. XCS). Therefore, R learns each instance following XCS standard procedure of learning. At each iteration R perceives a single instance and selects an action (class) based on its past experience to S. R alternately executes the *explore scheme* (choosing an action randomly) or the *exploit scheme* (choosing a best action) to select the action to S. In response, S reads the action and sends a numerical reward back to R. Based on the reward received, R updates its rules that proposed the action.

In Phase 1, the problem is considered either 'hard' or 'easy' based on the classification agent's performance after a certain number of iterations. Here, humans set the levels of the difficulty so that if the classification performance is greater than 95% for each problem, the problem is categorized as 'easy'. However, in Phase 2, the system set the problem's difficulty (i.e. the problem's difficulty is either increased or decreased automatically by the problem generator based on the classification agent's performance). Therefore, the relative features in the problem that make the problem either 'harder' or 'easier' is crucial.

A problem for classification can be difficult for several reasons (see Section 2.2.1.1). Traditionally, research in this area involves choosing a domain, creating a source of exemplars and trying learning algorithms that seem likely to work in that domain [124]. This includes creating problems with large sets of examples that can be learned from (i.e. library of problems). In this case, an automatic problem generator would be valuable if it were capable of producing examples of each class that are diverse and subtle as well as numerous [124]. The following section will describe the

two problem domains (i.e. image-based data and artificial data) for the creation of the classification problems.

3.1.3 Image-based Data for Classification

In [124], Wilson provided some implementation suggestions for a framework to be considered in order to create image-based data. A pattern (image-based data) can be illustrated by a two-dimensional pattern and can be encoded by a gray-scale visual pattern. Next, the pattern can be transformed into another pattern via a transformation process, such as translation, scaling and rotation within a defined range. Therefore, the generation agent would contain a transformation module to take an input image and transform it into an output image. The classification agent needs to translate the output image back to the input image (i.e. classify the pattern to a correct class).

In Phase 1, *image-based data* for classification system is created. However, for simplicity a pattern (image-based data) is encoded in a binary representation of '1' and '0'. The pattern is generated based on a list of defined features such as dimensionality, orientation and angle. There are different types of patterns that can be generated depending on a number of defined features (i.e. $2^{NumberOfFeatures}$). Instead of using the transformation module, the logical operators 'OR' and 'XOR' are used to emphasize a certain feature in the generated patterns, so that it can be classified into 'Class 1' or 'Class 0'. It is hypothesised that by emphasizing a certain feature in these generated patterns, it will make the pattern either 'hard' or 'easy' to learn.

3.1.3.1 Image-based Data Domain

For *simple image-based data*, a *problem* can be described as a list of features containing [`PatternDimension`,`PatternOrientation`,`PatternOperator`]. Table 3.1 describes the encoding scheme for simple image-based data. The

first column describes each feature in the problem, while the second column gives the encoding scheme used to represent different values of the feature.

For example, the *problem* of '0010000' can be translated as '3 by 3 dimension pattern' with 'Horizontal' orientation of three adjacent pixels and applying logical operator 'OR'. The *pattern* can be encoded into a value of either '1' for white or '0' for black. For instance, a pattern resulting from the problem (e.g. '0010000') can be represented by 9 bits (pixels) '010010010' and mapped into 3 by 3 dimensional mapping (i.e. row by row). In this case, if the pattern can be mapped into three adjacent Horizontal '1's (i.e. three adjacent Horizontal pixels), then it can be categorized as belonging to 'Class 1', otherwise 'Class 0'. However, if there are *multiple sets* of the three adjacent Horizontal '1's or three adjacent Horizontal pixels in a single pattern, it is still categorized as belonging to 'Class 1' (see Figure 3.2 (a)) due to the logical operator 'OR' in the problem's feature (i.e. `PatternOperator`).

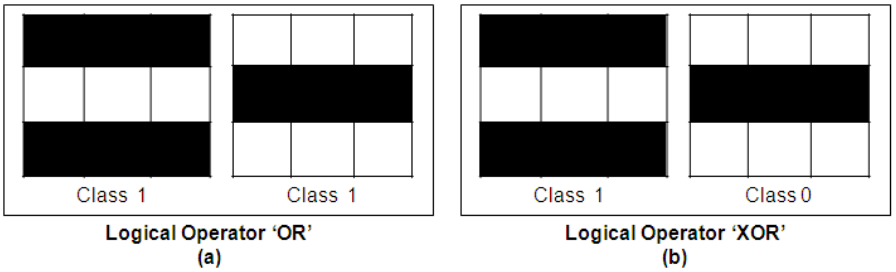


Figure 3.2: Applying logical operator 'OR' and 'XOR' (simple image-based data).

In another example, the *problem* of '0010001' can be translated as '3 by 3 dimension pattern' with 'Horizontal' orientation of three adjacent pixels and applying logical operator 'XOR'. For instance, one of the *patterns* can be represented by 9 bits (pixels) of '010010010'. However, in this case, if there are three adjacent pixels arranged Horizontally, and if *only one set*

Table 3.1: Encoding scheme (simple image-based data).

Feature	**Encoded Scheme**
F1 PatternDimension n by n pattern dimension with n number of rows and n number of columns.	**The first 2 bits**: '00' represents 3 by 3 dimension. '10' represents 4 by 4 dimension. '11' represents 5 by 5 dimension.
F2 PatternOrientation Pattern orientation such as Vertical, Horizontal, Diagonal1 or Diagonal2.	**The next 4 bits**: Each bit represents Horizontal, Vertical, Diagonal1 or Diagonal2. '1000' represents Horizontal, '0100' Vertical and so forth. If **PatternOrientation** '1100', the image will contain patterns with Horizontal and Vertical orientation.
F3 PatternOperator Applying logical operator 'OR' or 'XOR' for classifying the class.	**The next 1 bit**: '0' represents logical OR operator. '1' represents logical XOR operator. **PatternOperator** relates to the fact that the specified pattern can occur in either *one set* or *multiple sets* in the pattern. If **PatternOperator** is 'OR', then *any number* of the specified patterns can occur for it to be categorized as 'Class 1'. If **PatternOperator** is 'XOR', then *one and only one* of the specified patterns can occur for it to be categorized as 'Class 1'.

of three adjacent pixels arranged Horizontally exists, then 'Class 1', otherwise 'Class 0' (see Figure 3.2 (b)). This condition strictly applies when the feature of `PatternOperator` is the logical operator 'XOR'.

The last feature of the problem (i.e. `PatternOperator`) relates to the fact that the specified pattern can occur in either *one set* or *multiple sets* in the pattern. If `PatternOperator` is set to logical operator 'OR', then *any number* of the specified patterns can occur for it to be categorized as 'Class 1' (see Figure 3.2 (a)). However, if the `PatternOperator` is set to the logical operator 'XOR', then *one and only one* of the specified patterns can occur for it to be categorized as 'Class 1', otherwise 'Class 0' (see Figure 3.2 (b)).

3.1.3.2 Knowledge Representation

The Receiver (R) is developed based on accuracy-based LCSs (i.e. XCS) using *ternary* alphabet representation. Table 3.2 illustrates R's *condition-action* rule format. The *condition* specifies each generated pattern, while the class is considered as the *action*. The action can be either '1' for 'Class 1', otherwise '0' for 'Class 0'. R will receive a reward of '1000' for correct classification or '0' for incorrect classification.

Table 3.2: Example of R's condition-action rule format.

R's condition-action rule:
IF<$pattern$>THEN<$class$>
Rule's description:
'01#010010:1' where '01#010010' pattern can be mapped into 3 by 3 pattern dimension mapping and is predicted to be 'Class 1'.

The same R's *condition-action* rule format will be used in all of the experiments throughout the work, unless explicitly otherwise stated.

3.1.3.3 Image-based Data Generation and Classification

Figures 3.3 and 3.4 illustrate the process of the *pattern generation* and *classification* between the Sender (S) and the Receiver (R).

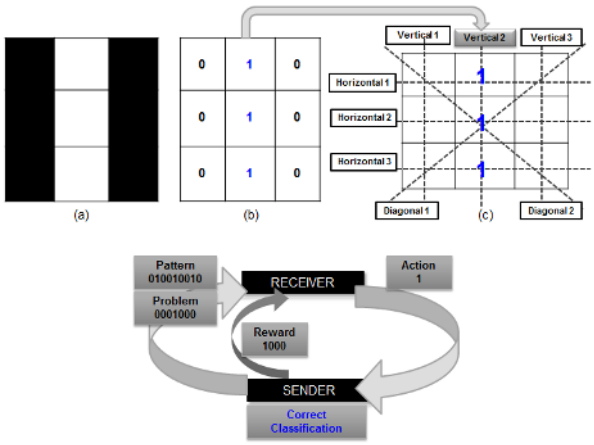


Figure 3.3: Image-based data generation and classification between S and R (*correct classification*).

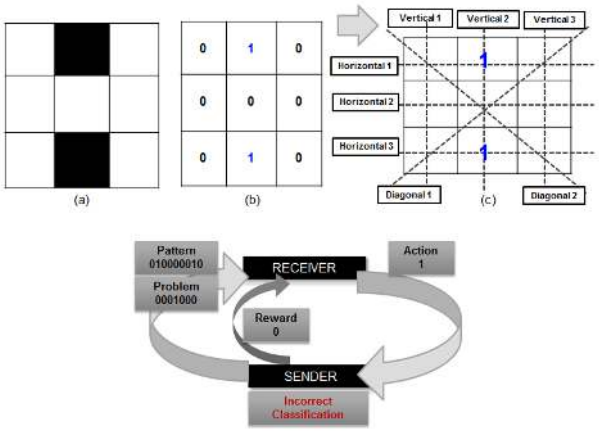


Figure 3.4: Image-based data generation and classification between S and R (*incorrect classification*).

First, S creates a population of problems with different sets of features. A *problem* contains a list of features such as [`PatternDimension`, `PatternOrientation`, `PatternOperator`]. For example (see Figure 3.3), given a problem of [`3 by 3 dimension`, `Vertical orientation`, `operator OR`], the problem can be encoded by 7 bits binary string of '0001000'. At each iteration, an individual problem from the meta-problem is selected randomly from the population of problems.

Secondly, S generates a pattern (image-based data) based on the individual problem's features (i.e. '0001000') so that the generated *pattern* consists of 9 bits binary string, which can be mapped into a 3 by 3 dimensional pattern (see Figure 3.3 (a)). For instance, one of the patterns can be encoded by 9 bits binary string of '010010010' (refer Figure 3.3 (b)).

If the pattern contains three adjacent Vertical '1's (i.e. three adjacent pixels Vertically) or multiple sets of three adjacent Vertical '1's, depending on the problem's feature of `PatternOperator`, then it can be categorized as belonging to 'Class 1', otherwise '0' (see Figure 3.3 (c)). There are 2^n possibilities of different patterns S wishes R to recognize for each problem, where n is the number of bits in the pattern.

Thirdly, R will need to classify a pattern (e.g. '010010010') as either belonging to 'Class 1' or 'Class 0'. Next, R sends either '1' for 'Class 1' or '0' for 'Class 0' as suggested by its rules. In response, S sends a numerical reward of '1000' for correct classification, or '0', back to R.

3.1.4 Artificial Data for Classification

Lately, the study and comparison of different types of learners on various datasets has gained attention among researchers. Often in real-world problems, some features of the problem's difficulty cannot be identified or taken separately because of the overlapping features in the problem's difficulty. There are several complexity factors such as data volume, search space size and type, complexity of the concept, noise in the data, handling

of missing values, and problem of over-fitting.

One approach to investigating the learner's capability in this area is to use artificial data [71]. This method allows the researcher to analyze the learner's capability under a controlled scenario, where the datasets can be generated according to particular known features. Thus, issues of problem difficulty such as noise, missing values, data sparsity and dimensionality, can be introduced in a controlled way. Moreover, the use of artificial data offers better understanding of the learner's behaviour since the complexity of the problem being studied is known [72], in a way that features of the problem that affect the learner's performance are transparent.

Therefore, *artificial data* for classification is developed. Here, a *problem* is defined by a list of parameters. The parameters represent features of a problem's difficulty such as feature space dimensionality, data volume, noise in the data, class ambiguity, sample sparsity and boundary complexity. Next, the datasets are generated based on the list of those defined parameters. In this domain, the work focuses to solve binary classification tasks where two classes need to be classified.

3.1.4.1 Artificial Data Generation

In this domain, a *problem* is defined by a list of parameters (i.e. `[Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd]`). These parameters are real-valued within a specified range, where each feature can take a number of values (see Table 3.3). The datasets are generated based on the list of those defined parameters. It is hypothesised that by tuning a certain feature `F` of the current problem, it may make the problem either 'hard' or 'easy' to learn.

`F` is used to distinguish a feature of the problem and `f` is used in relation to data feature of each instance in the dataset. `Fn` determines the number of data features `f` in the dataset. Once these features `F` are set, the dataset containing a number of instances can be generated and the class of each instance is labeled accordingly, i.e. based on the remaining features

`F` in the problem. The datasets consist of a set of `N` instances, where each instance `n` is defined by the specified problem of features `F`. Each instance `n` is created *on-the-fly* with each data features `f` being within the interval of [0,1] and is labeled accordingly.

Table 3.3 describes the encoding scheme for each feature `F` in the problem (i.e. `[Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd]`). The first column gives details of each feature `F` in the problem, while the second column provides the encoding scheme used to represent the different values of the feature.

3.1.4.2 Knowledge Representation

The Receiver (R) is developed based on accuracy-based LCSs with real-value conditions (i.e. XCSR). Table 3.4 illustrates R's *condition-action* rule format. The $condition$ is encoded to be real-value of $real_n = (l_n, u_n)$, where l_n is the lower bound and u_n is the upper bound within the interval $[0, 1]$. The action can be either '1' for 'Class 1', otherwise '0' for 'Class 0'. The last row illustrates R's rules to specify one instance with two data features where each data feature $data_n$ is within the interval of lower bound l_n and upper bound u_n (i.e. [0,1]). R will receive a reward of '1000' for correct classification or '0' for incorrect classification.

The same R's *condition-action* rule format will be used in all of the experiments throughout the work, unless explicitly otherwise stated.

Table 3.3: Description of features *F* in the problem.

Description.	**Value of feature *F***.
`Fn`: number of data features in each instance.	`Fn`: from 2 to 5. If `Fn=2`, each instance contains two data features `f1` and `f2`.
`Fc`: number of conjunction.	`Fc`: at least 1/2 of `Fn`. If `Fn=2`, then `Fc=1`.
`Fd`: number of disjunction.	$0 \leq$ `Fd` $\leq$ `Fn`. If `Fn=2`, then `Fd` can take any value from 0 to 2.
`Fi`: number of irrelevant features.	$0 \leq$ `Fi` $<$ `Fn`. If `Fn=2`, then `Fi` can take any value from 0 to 1.
`Fr`: number of redundant features.	$0 \leq$ `Fr` $<$ `Fn`. If `Fn=2`, then `Fr` can take any value from 0 to 1.
`Fan`: percentage of noise level apply to class.	`Fan`: from 0%-50%.
`Fcn`: percentage of noise level apply to data features.	`Fcn`: from 0%-50%.
`Fcbl`: percentage of 'Class 1' instances in the dataset.	`Fcbl`: from 0%-100%. If `Fcbl=50`, there will be 50% instances of 'Class 1' and 50% instances of 'Class 0'.
`Fcbd`: percentage of decision boundary to separate between each class (i.e. wide or small decision boundary of the class).	`Fcbd`: from 0%-50%.

Table 3.4: Example of R's condition-action rule format.

R's condition-action rule:
IF<$condition$> THEN<$class$> $condition$ is a list of data feature for each instance containing: [$data_1, data_2, ..., data_n$], where each $data_1 : [l_1, u_1]$
[$data_1, data_2$]: $class$ [5.5,0.98], [0.4,0.8] :**0**

3.1.4.3 Artificial Data Generation and Classification

The process of *artificial data generation* and *classification between S and R is described as follows (see Figure 3.5)*. First, S generates variants of problems for classification (i.e. a population of problems referred as a *meta-problem*). S is initialized with an initial value for an individual problem (i.e. humans specifies each value of features `F`). Secondly, S creates a set of artificial dataset for classification associated with the individual problem that needs to be solved by R. Next, S sends each instance in the dataset to R in turn (i.e. one iteration for S is the individual problem, and one iteration for R is an instance from the individual problem).

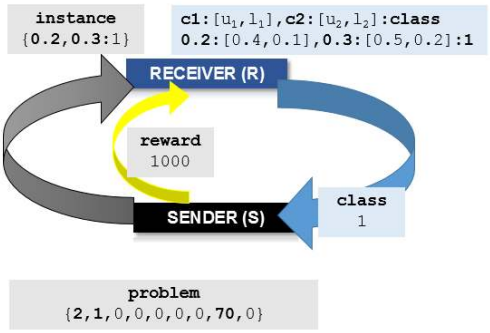


Figure 3.5: Artificial data generation and classification between S and R (*correct classification*).

For example, given a problem of <Fn=2 Fc=1 Fd=0 Fi=0 Fr=0 Fan=5 Fcn=5 Fcbl=70 Fcbd=5>, the problem generator (S) generates each instance n by evaluating the values of F in the order of precedence such as Fn, Fcbl, Fc, Fd, Fcbd, Fr, Fan and Fcn described as follows:

1. The system checks for the value of Fn. In this case, each instance in the dataset consists of two data features such as f1 and f2 when Fn is set to 2 (Fn=2).

2. The system calculates the possible values of f1 and f2 based on the value of Fcbl where each instance is created *on-the-fly*. Here, the class balance Fcbl is set to 70 (Fcbl=70). Therefore, the dataset will contain 70% instances from 'Class 1' and 30% instances from 'Class 0'. The possible value of f1 and f2 is performed as follows when class balance Fcbl is set to 70%.

3. If Fc is set to 1 (Fc=1) and Fd is set to 0 (Fd=0), the probability of f1 and f2 follows P(f1 AND f2) and is computed as P(f1) $\times$ P(f2). In this case, P(f1) $\times$ P(f2) is equal to 0.7 when Fcbl/100 (i.e. 70/100). The probability of f1 and f2 is calculated as $\sqrt[2]{0.7} = 0.836$. Next, the possible values (decision values) of f1 and f2 are calculated as $1.0 - 0.836 = 0.164$ (within the interval [0,1]) and are set to 0.164. Based on this value (decision value), a rule for classifying each instance either belonging to 'Class 1' or 'Class 0' is derived as (IF f1>=0.164 AND f2>=0.164 THEN 'Class 1').

4. However, if Fd is set to 1 (Fd=1) and Fc is set to 0 (Fc=0), the probability of f1 and f2 follows P(f1 OR f2) and is computed as P(f1)+P(f2)-[P(f1) $\times$ P(f2)]. In this case, P(f1 OR f2) is equal to 0.7 when Fcbl/100 (i.e. 70/100). If the probability of f1 is set to 0.6 which less than 0.7, then the probability of f2 is calculated as 0.7=0.6+f2-[0.6$\times$f2], where f2 is equal to 0.25. Next, possible values (decision values) of f1 is calculated as $1.0 - 0.6 = 0.4$ and f2

is calculated as $1.0 - 0.25 = 0.75$ (within the interval [0,1]). Based on this value (decision value), a rule for classifying each instance either belonging to 'Class 1' or 'Class 0' is derived as `(IF f1>=0.4 OR f2>=0.75 THEN 'Class 1')`.

5. The system creates an instance randomly and labels the instance based on the derivation rules. The dataset contains a number of instances such as `<0.6,0.3>` and `<0.1,0.1>`.

6. Next, each data feature `f1` and `f2` is checked with the defined decision boundary `Fcbd`, where `Fcbd=5`. The data value of `f1` and `f2` are recalculated to ensure that it is well separable between 'Class 1' and 'Class 0' (see Table 3.5). First, the system determines the values of DB (DecisionBoundary) and R (Range). Here, DB is 0.164 (i.e. the decision values of `f1` and `f2`, when `Fc=1` and `Fd=0`) and R is calculated as `(Fcbd/100)/2`. Second, each data feature is checked either greater than DB or less than DB. If the data feature greater than DB, its value is recalculated as follows $Xn = (DB + (R/2)) + (Xn - DB/1 - DB) \times (1 - DB - (R/2))$. However, if the data feature is less than DB, its value is recalculated as $Xn = (Xn/DB) + (DB - (R/2)$. For example, the first instance contains `<0.6,0.3>`, where the first data feature `f1` is 0.6 and and the second data feature `f2` is 0.3. Here, `f1` is greater than DB, when $DB = 0.164$. So `f1` is recalculated as $(0.164 + (0.5/2)) + (0.6 - 0.164/1 - 0.164) \times (1 - 0.164 - (0.5/2))$, and the new value is 0.548. Value of `f2` also is recalculated as $(0.164 + (0.5/2)) + (0.3 - 0.164/1 - 0.164) \times (1 - 0.164 - (0.5/2))$ when `f2` is greater than DB, and the new value is 0.337. The new value for the first instance now is `<0.548,0.337>`.

7. The system checks for the value of `Fr`. Here, `Fr` is set to 0 (`Fr=0`), which means there is no redundant value between the data features `f1` and `f2`. The value of the second data feature (i.e. `f2`) remains the same. However, if `Fr` is set to 1 (`Fr=1`), the value of the second data

feature f2 will has the same value with f1.

8. Next, based on the derived rule (i.e. (IF f1>=0.164 AND f2>=0.164 THEN 'Class 1')), each instance is labelled either 'Class 1' or 'Class 0' such as <0.548,0.337:1>.

9. Finally, the system applies noise to the data features f1 and f2 and the class. In this case, each data feature and the *class* has a random value associated with them. As Fcn is set to 5 (Fcn=5), the chance of noise being applied to the data feature f1 (and f2) is 5%. If the generated random value associated to f1 (and f2) is less than 0.05, then the value of f1 (and f2) is reset to a new value within the defined range, otherwise the value of f1 (and f2) remain the same. Here, Fan is set to 5 (Fan=5), so the noise level that applies to the class is 5%. If the generated random value associated with the class is less than 0.05, the class will be flipped either from '1' to '0' or '0' to '1', otherwise the class remains unchanged. In order to avoid the classification agent learning the inverse problem, the value of Fan and Fcn is set within 0-50%.

Thirdly, R needs to successively classify each instance in the dataset as either belonging to 'Class 1' or 'Class 0'. Thus, R sends either '1' for 'Class 1' or '0' for 'Class 0' as suggested by its rules. In response, S sends a numerical reward of '1000' for correct classification, or '0' is returned to R.

Table 3.5 shows sample of the generated instance n and its data features f when features F are set as follows <Fn=2 Fr=1 Fan=5 Fcn=5 Fcbl=70 Fcbd=5>.

Table 3.5: Sample of the generated instance n.

Problem.	Instance and class.
Example 1: conjunction (`Fc=1, Fd=0`) `Class 1: IF(f1>=0.164 AND f2>=0.164)`	`f1 f2 : class` 0.1 0.1 : 1 0.5 0.5 : 0
Example 2: disjunction (`Fd=1, Fc=0`) `Class 1: IF(f1>=0.4 OR f2>=0.75)`	`f1 f2 : class` 0.1 0.1 : 0 0.5 0.8 : 1

Example 3: noise to action (`Fan=5, where noise=0.05`).	`f1 f2:class:rndValue`
	0.1 0.1 : 1 : 0.6
	`f1 f2:class`
	0.1 0.1 : 1
	IF(`rndValue` of `class`>noise)THEN `class` is remain.
	ELSE flip `class` either from '1' to '0' or from '0' to '1'.

Example 4: noise to condition (`Fcn=5, where noise=0.05`)	`f1:rndValue,f2:rndValue:class`
	0.1:0.6, 0.1:0.6 : 1
	`f1 f2 : class`
	0.1 0.1 : 1
	FOR each data feature (`f1` to `f2`)
	IF(`rndValue` of `f1` and `f2` >noise) THEN the value of the data feature (`f1` and `f2`) is remain.
	ELSE the value of the data feature (`f1` and `f2`) will be replaced by a new random value.

Example 5: decision boundary (`Fcbd=5`)
`Class 1: IF(f1>=0.164 OR f2>=0.164)` $DB(DecisionBoundary) = 0.164$
$R(Range) = (Fcbd/100)/2 = 0.25$
	FOR each data feature ($Xn : f1 to f2$)
	IF(value of f1 or f2>decision boundary)
	$Xn = (DB + (R/2)) + (Xn - DB/1 - DB) \times (1 - DB - (R/2))$
	IF(value of f1 or f2<decision boundary)
	$Xn = (Xn/DB) + (DB - (R/2)$

3.1.5 Experimental Design

In our implementation, the classification agent (i.e. the Receiver (R)) is executed following *Wilson's explore/exploit scheme* [119], which has become the standard approach in accuracy-based LCSs. The explore and exploit schemes are run alternately with probability 50% *explore*[1] and 50% *exploit*[2]. For example, to learn 2,000 instances, R is run for 2,000 iterations. In the first iteration, R chooses the action randomly (*explore scheme*), while in the second iteration, R chooses the best action (*exploit scheme*). For 2,000 iterations, there will be 1,000 *explore scheme* and 1,000 *exploit scheme*.

Table 3.6 shows the parameter setting for R (that applied *accuracy-based LCSs*, XCS) for addressing the *image-based data* for classification. The parameters are set according to [54], where a few modifications are made to improve the efficiency, including the number of classifiers and the number of iterations. There will be many iterations of R for each iteration of S (e.g. to learn 10,000 patterns (instances), R is run for 10,000 iterations).

Table 3.7 shows the parameter setting for R (that applied *accuracy-based LCSs with real value*, XCSR) for addressing the *artificial data* for classification. The parameters are set according to [120, 13]. A few modifications are made to improve the efficiency, including the number of classifiers and the number of iterations. In this domain, the population size N is limited to 500 classifiers for each problem, where R learns 2,000 instances (i.e. R runs for 2,000 iterations). There will be many iterations of R for each iteration of S. One iteration for S is a *problem*, and one iteration for R is an *instance*. Both values are low for standard LCSs to reduce training times as the overall 'meta-problem' task is time consuming.

[1] In *explore scheme*, the system selects an action at random from those advocated by the matching rules (choosing the action randomly).

[2] In *exploit scheme*, the system deterministically selects the action which is most highly recommended by the matching rules (choosing the best action).

Table 3.6: Parameters setting for XCS (image-based data).

Parameter		Value
Number of iterations		10,000 (3 by 3 pattern), 50,000 (4 by 4 and 5 by 5 pattern) (one iteration representing one instance of the problem)
Maximum size of the population	N	500, 5,000, 50,000
Selection algorithm		Tournament selection
Tournament size	τ	a fraction $\tau = (0, 1]$ of the current action set size, where ($\tau = 0.4$ is the suggested value)
Crossover algorithm		Two-Point crossover
Crossover probability	χ	0.8
Mutation algorithm		Random mutation
Mutation probability	μ	0.4
Probability of using #	$p\#$	0.01
Accuracy discount factor	α	1.0
Learning rate	β	0.2
Rule's fitness to its accuracy	ν	5
GA threshold	θ_{GA}	25
Subsume threshold	θ_{sub}	20
Rule deletion threshold	θ_{del}	20
Deletion discount factor	δ	0.1
Initial prediction error	ϵ_0	1.0
$GAsubsumption$		YES
$ASsubsumption$		YES

Table 3.7: Parameters setting for XCSR (artificial data).

Parameter		Value
Number of iterations		2000 (one iteration representing one instance of the problem)
Maximum size of the population	N	500
Selection algorithm		Tournament selection
Tournament size		a fraction $\tau = (0, 1]$ of the current action set size, where ($\tau = 0.4$ is the suggested value)
Crossover algorithm		Arithmetic crossover
Crossover probability	χ	0.8
Mutation algorithm		Random mutation
Mutation probability	μ	0.4
Mutation threshold	m_0	0.2
Covering threshold	r_0	0.4
Accuracy discount factor	α	1.0
Learning rate	β	0.2
Rule's fitness to its accuracy	ν	5
GA threshold	θ_{GA}	25
Subsume threshold	θ_{sub}	20
Rule deletion threshold	θ_{del}	20
Deletion discount factor	δ	0.1
Initial prediction error	ϵ_0	10
$GAsubsumption$		YES
$ASsubsumption$		YES

The same parameters setting for XCS and XCSR will be used in all of the experiments throughout the work, unless explicitly otherwise stated. Both of the systems (i.e. XCS and XCSR) are implemented as suggested by Butz [13].

All of the experiments are run 30 times with different random seeds for analysing the results. R's performance was calculated from the exploit trials. After a certain number of iterations, R's classification performance (i.e. average of correct classification performance from the exploit trials over 30 runs) is used to calculate R's performance on the specified problem.

3.1.6 Summary

In Phase 1, two different problem domains for classification (i.e. image-based data and artificial data) are established. Here, both of the problem domains can be tuned and adjusted autonomously (i.e. either to make the problem 'hard' or 'easy') by the problem generator (i.e. the Sender). In this phase, the experimental design, including experimental setup, parameters setting and evaluation metrics are presented. Based on this set-up, the classification agent's (i.e. the Receiver) ability on various problem domains is evaluated to help in empirically testing the learning bounds of the agent. Thus, in the next phase, the two problem domains (i.e. image-based data and artificial data) will be further investigated for the *Two-Cornered Coevolution System*.

3.2 Two-Cornered Coevolution System

The overall aim of this phase is to design a *new problem generation approach* to autonomously generate image-based data (or artificial data) for classification with different levels of difficulty based on the classification agent's ability to learn. In the *Two-Cornered Coevolution System*, the generation agent (i.e. the automated evolvable problem generator) creates various complex problems for classification, whilst the classification agent (i.e. accuracy-based LCSs) learns the problems and adapts to different feedback returns from the generation agent. Ultimately, the problem's difficulty needs to be adjusted (i.e. increased or decreased) based on the classification agent's learning ability once the problem features that alter its performance have been identified.

3.2.1 Research Objectives

This aim can be broken down as the following objectives.

1. Develop a *Two-Cornered Coevolution System* for addressing classification problems, which consists of two main agents: the problem generation agent (i.e. the Sender) and the classification agent (i.e. the Receiver).

2. Evaluate the Receiver's performance for learning different types of problems for classification within a given period of time or after a certain number of iterations.

3. Investigate a method for the Sender to increase or decrease a problem's difficulty based on the Receiver's performance and to autonomously determine how the features in a problem domain affect the learning of the Receiver.

4. Investigate the cooperation between the Sender and the Receiver in this coevolutionary process.

3.2.1.1 Framework Design

The system consists of two main agents, the generation agent (i.e. the Sender (S)) and the classification agent (i.e. the Receiver (R)) (see Figure 3.6). S is a program to create various problems for classification with different levels of difficulty using Pittsburgh-style LCSs. R is a program to learn the generated image-based data (or artificial data) using accuracy-based LCSs (i.e. XCS or XCSR). Both of the agents evolve using evolutionary computation (i.e. Genetic Algorithm).

For this work, Pittsburgh-style LCSs is found more suitable for application to the generation agent considering that Pittsburgh-style LCSs usually evolve more compact population of rules compared to Michigan-style LCSs. Here, Pittsburgh-style LCSs, A-PLUS (Accuracy-based Pittsburgh Learner Using Subsumption), is selected to be applied to the generation agent. A-PLUS is capable of addressing the 'bloat' phenomenon, which refers to increasing any variable-sized solution (in this case, a set of rules) [1].

Figure 3.6 illustrates the overall design of the system. In Phase 2, S generates various image-based data (or artificial data) from different problems for classification that will need to be solved by R. Here, S adjusts the problem's difficulty autonomously based on R's learning ability. Therefore, S activates a program for problem generation, while R activates a program for classification. R learns the classification problems and adapts to different feedback returns from S. Both S and R maintain their own population of candidate classifiers and evolve using evolutionary computation (i.e. Genetic Algorithm). Based on R's performance, S needs to identify and discover the difficulty of the problem (e.g. the effect of varying feature values in the problem) in order to generate a new problem at the appropriate levels of difficulty for R to learn. If S's objective is to increase R's performance, S tunes and makes the problem 'easier' to learn. If S's objective is to decrease R's performance, S makes the problem 'harder' to learn. Here, S is trying to discover features in the problem that either make the

problem 'easier' or 'harder' for R to learn, with respect to a fixed R implementation (fixed parameters setting).

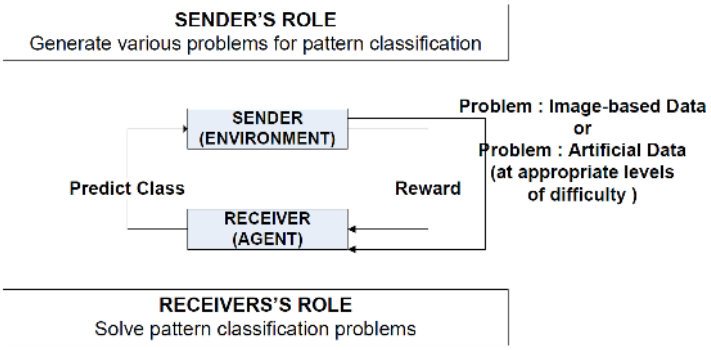


Figure 3.6: Two-Cornered Coevolution System.

3.2.2 Image-based Data for Classification

3.2.2.1 Image-based Data Domain

In Phase 1, the *simple image-based data*, which is a *non-sparse problem* with a low number of conditions has been established. The *problem* can be described as a list of features that contains [`PatternDimension`, `PatternOrientation`, `PatternOperator`] and each feature can take a number of values such as [`3 by 3 dimension`, `Vertical orientation`, `operator OR`] (see Table 3.1).

In Phase 2, the simple image-based data is further extended to a *complex image-based data*, which has a higher number of conditions. The *problem* can be described as a list of features (containing more than three features) such as [`PatternDimension`, `PatternStroke`, `PatternOrientation`, `PatternAngle`, `PatternOperator`]. Each feature can take a number of values such as [`3 by 3 dimension`, `1 stroke`, `Vertical orientation`, `0 degree angle`, `OR operator`]. Table 3.8 describes the encoding scheme for each feature

in the *complex image-based data*. The first column gives a detailed description of each feature in the problem, while the second column provides the encoding scheme used to represent different values for the feature.

3.2.2.2 Knowledge Representation

S generates various problems for classification, while R learns the image-based data (patterns) for each problem and adapts to different feedback returns from S in order to predict the class. R's *condition-action* rules are similar to Phase 1 (see Table 3.2). Table 3.9 illustrates S's *condition-action* rule format for the *complex image-based data*. S is developed based on Pittsburgh-style LCSs (i.e. A-PLUS) using *ternary* alphabet representation. The *condition* specifies the problem containing a list of parameters [`PatternDimension`, `PatternStroke`, `PatternOrientation`, `PatternAngle`, `PatternOperator`], where each feature can take on a number of values. The problem's difficulty is considered to be the *action*, where it can be either '1' for 'hard' problem, or '0' for 'easy' problem. The *ProblemDifficulty* is based on the classification performance after a certain number of iterations.

The same S's *condition-action* rules format will be used in all of the experiments for addressing the image-based data in this phase, unless explicitly otherwise stated.

3.2.2.3 Image-based Data Generation and Classification

Figure 3.7 illustrates the process of the *image-based data generation* and *classification* between S and R. First, S creates a population of problems with different sets of features. A *problem* contains a list of features [`PatternDimension`, `PatternStroke`, `PatternOrientation`, `PatternAngle`, `PatternOperator`], where the value can be [`3 by 3 dimension`, `2 stroke`, `Vertical and Horizontal orientation`, `90 degree angle`, `operator XOR`] that can be encoded by an 11 bits binary string of '00101100011'. At each iteration, an individual problem is selected randomly from the population.

Table 3.8: Encoding scheme (complex image-based data).

Feature	Encoding Scheme
F1-PatternDimension n by n pattern dimension with n number of rows and n number of columns.	**The first 2 bits**: '00' 3 by 3 pattern dimension. '01' 4 by 4 pattern dimension. '10' 5 by 5 pattern dimension.
F2-PatternStroke total number of strokes in pattern (e.g. 1 stroke equivalent to 3 adjacent pixels in 3 by 3 dimension pattern).	**The next 2 bits**: '00' if total number of stroke in pattern equal to 0. '01' if total number of stroke in pattern equal to 1. '10' if total number of stroke in pattern equal to 2.
F3-PatternOrientation orientation of pattern such as Vertical, Horizontal, Diagonal1 or Diagonal2 on the pattern mapping.	**The next 4 bits**: Each bit represents Horizontal, Vertical, Diagonal1 or Diagonal2. '1000' represents Horizontal, '0100' Vertical and so forth.
F4-PatternAngle if there is any desired angle from four angles with degree of 0, 90, 180 and 360 in pattern; an angle is formed by intersecting strokes.	**The next 2 bits**: '00' if there is 0 degree angle in pattern. '01' if there is 90 degree angle in pattern. '10' if there is 180 degree angle in pattern. '11' if there is 360 degree angle in pattern.
F5-PatternOperator applying logical operator OR or XOR for classifying the class.	**The next 1 bit**: '0' represents logical OR operator. '1' represents logical XOR operator.

Table 3.9: Example of S's condition-action rule format.

S's condition-action rule:
IF<$problem$>THEN<$ProblemDifficulty$> $problem$ is a list of features containing : [`PatternDimension, PatternStroke, PatternOrientation, PatternAngle, PatternOperator`]
Rule's description:
'00011000000:0' where '00' represents 3 by 3 pattern dimension, '01' represents total stroke is 1, '1000' represents Horizontal orientation, '00' represents 0 degree angle in the pattern, '0' represents operator 'OR' and the problem is predicted to be a 'easy'.

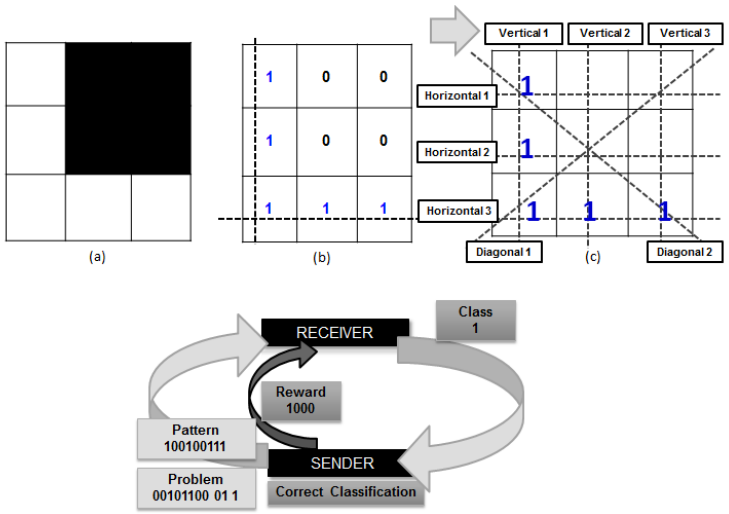


Figure 3.7: Image-based data generation and classification between S and R.

Secondly, S generates a *pattern* (image-based data) based on the individual problem's features (i.e. '00101100011'). For example, the generated pattern consists of a 9 bits binary string, which can be mapped into 3 by 3 dimensional pattern. The pattern can be encoded into a value of either '1' for white or '0' for black, and read row by row (see Figure 3.7 (a)). For instance, the pattern can be encoded by a 9 bits binary string of '100100111' (see Figure 3.7 (b)). If the pattern contains of three adjacent Vertical '1's (i.e. three adjacent pixels Vertically) and three adjacent Horizontal '1's (i.e. three adjacent pixels Horizontally), where the total stroke is 2 (i.e. 2 straight lines of Vertical and Horizontal lined) and can build a 90 degree angle, then it can be categorized as belonging to 'Class 1', otherwise 'Class 0' (see Figure 3.3 (c)).

Thirdly, R needs to classify the pattern of '100100111' as either belonging to 'Class 1' or 'Class 0'. Next, R sends either '1' for 'Class 1' or '0' for 'Class 0' as suggested by its rules. In response, S sends a numerical reward of '1000' for correct classification, or '0' back to R.

After a certain number of iterations, S will evaluate R's performance for learning different types of patterns. Based on its prediction, S is tasked to create the next problem at the appropriate levels of difficulty to R, where S autonomously determines the features in the problem that alters R's performance, and S can either increase or decrease the problem's difficulty based on R's performance.

However, in this domain, there is a case where S can create a sparse problem such as creating a valid encoding, which does not produce a valid problem for R to learn. Thus, it is impossible for S to predict R's learning ability correctly. For example, the *problem* '00011000011' can be translated as `[3 by 3 dimension, 1 stroke, Horizontal orientation, 90 degree angle, operator XOR]`. This example is considered *sparse* when valid encodings do not produce valid problems, e.g. one stroke and 90 degree angle. Therefore, considering this 'sparse problem', an *appropriate problem domain* (i.e. artificial data) is needed to address the classification

problems and is discussed in detail in the next section. In addition, within this domain there are a few other problems that limit the system's capability of producing flexible and adjustable problems for the classification tasks which are discussed in Section 5.1.

3.2.3 Artificial Data for Classification

3.2.3.1 Artificial Data Generation

In this domain, the problem of defining a problem with an appropriate level of difficulty is considered a *meta-problem*. The meta-problem itself is difficult in practice due to the execution time needed to solve for any individual problem. Starting from the beginning for each problem or repeating a given problem is time consuming. Thus, a local search method is required to adjust the problem domain so the generation agent can commence from the previous learnt problems, without requiring to repeat the same problem domain. The process of *artificial data generation* is described as follows:

First, S generates variants of problems for classification (i.e. a population of problems referred as *meta-problem*). S is initialized with an initial value for an individual problem containing a list of parameters (i.e `[Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd]`) for generating a set of artificial data (dataset). The dataset is generated based on the list of those defined parameters associated with the individual problem that need to be solved by R (see Section 3.1.4.1 and Table 3.5).

Secondly, *Tabu Search* is used in S to discover the combination of features `F` in the problem in order to generate the next problem at the appropriate levels of difficulty. S can either make the problem 'harder' or 'easier' for R to learn (i.e. maximize or minimize R's performance). The basic principle of TS is to pursue local search whenever it encounters a local optima by allowing non-improving moves. When a new candidate solution is introduced (i.e. a new best solution is found), it goes into the tabu list and it

is made tabu for a certain number of iterations (tabu tenure). The tabu list records the recent history of the search and also keeps information on the itinerary through the last solutions visited. Here, one way of creating the tabu restrictions (constraint) is to assign a maximum iteration for each candidate solution in the tabu list. The approach for TS maintaining the tabu list is by remembering an index of features that have been swapped and the time (number of iterations) when it was swapped [77] (see Algorithm 11 in the Section 2.10).

In this set-up, a maximum time (iteration) for each index is 5. This means each feature can remain in the tabu list for only 5 times (iterations). Here, S objective is to make the problem 'harder' for R to learn (i.e. minimize Rs performance), where TS iteration is set to 10. S starts with an initial solution (i.e. `[Fn=5 Fc=1 Fd=0 Fi=0 Fr=4Fan=5 Fcn=5 Fcbl=50 Fcbd=5]`). Changes of features `F` (the solution) and tabu list content for the first two iterations is described as follows. For each iteration, each feature value `F` is changed within a certain limit. In the first iteration, the best solution is `[5, 1, 0, 1, 4, 5, 5, 50, 5]`, when R's performance is 67% and the swap in tabu list is at index 4 `[0, 0, 0, 5, 0, 0, 0, 0, 0]`. In this case, the swap at index 4 gives the best solution, thus the feature at index 4 is not allowed to be used and not available for the next 5 iterations. In the second iteration, the best solution is `[5, 1, 0, 1, 4, 5, 5, 50, 4]`, when R's performance is 62.0% and the swap in tabu list is at index 9 `[0, 0, 4, 0, 0, 0, 0, 0, 5]`. Here, index 4 and 9 is not available to be swapped for the next iterations, while other indexes are available to be swapped in any iteration.

Table 3.10 illustrates changes in feature `F` when applying TS in S to *increase* the problems difficulty. S starts with an initial solution (i.e. `[Fn=5 Fc=1 Fd=0 Fi=0 Fr=4 Fan=5 Fcn=5 Fcbl=50 Fcbd=5]`), where R's performance is 66%. However, after 10 iterations, TS is able to find the best combination of features F in the problem that decreases R's performance (i.e. minimize R's performance) from 66% to 51% by changing certain fea-

tures F. Using TS, S is able to search for the best combination of features `F` for the set task. Based on R's classification performance, S changes the values of features F (i.e. finds the best combination of features that can either make the problem 'harder' or 'easier') for generating the next problem for R to learn. R needs to learn again for each new problem send by S.

Table 3.10: Changes in features `F` using Tabu Search in S.

INITIAL SOLUTION: 5, 1, 0, 1, 4, 5, 5, 50, 5
INITIAL PERFORMANCE: 66.0
Search COMPLETE!
BEST PERFORMANCE: 51.0
BEST SOLUTION: 5, 0, 1, 0, 3, 29, 1, 3, 1
NOTE:
R's classification performance decreased from 66% to 51% using TS to adjust `[Fc Fd Fi Fr Fan Fcn Fcbl Fcbd]`.

3.2.3.2 Knowledge Representation

S generates various problems for classification, while R learns the datasets for each problem and adapts to the resulting feedback, returning from S in order to predict the class. R's *condition-action* rules are similar to Phase 1 (see Table 3.4). Table 3.11 illustrates S's *condition-action* rule format. S is developed based on Pittsburgh-style LCSs (i.e. A-PLUS). The *condition* (c_i) specifies the problem containing $state_1$ and $state_2$. Each $state$ is a list of features `F` in the problem (i.e `[Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd]`). The $state_n$ is encoded to be real valued at $real_n = (l_n, u_n)$, with a lower bound and an upper bound within a specified interval. The *ProblemDifficulty* is considered to be the *action*, where it can be either '1' for a 'harder' problem, or '0' for an 'easier' problem. In Phase 2, S must either increase or decrease the problem's difficulty to either maximize or minimize R's

performance. The *ProblemDifficulty* is therefore changed either from 'hard' to 'harder' or 'easy' to 'easier' compared to Phase 1. If R's performance for $state_2$ is greater than R's performance for $state_1$, then the problem is considered 'easier', otherwise 'harder'.

Table 3.11: Example of S's condition-action rule format.

S's condition-action rule:
IF<$state_1, state_2$> THEN<$ProblemDifficulty$> $state_n$ is a list of features containing: [`Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd`]
$state_1, state_2 : ProblemDifficulty$ $state_1 : [l_1, u_1], [l_2, u_2], [l_3, u_3], [l_4, u_4], [l_5, u_5], [l_6, u_6], [l_7, u_7], [l_8, u_8]$
$state_1$:[0,1], [0,1], [0,1], [0,1], [40,50], [40,50], [50,100], [0,50] $state_2$:[0,1], [0,1], [0,1], [0,1], [0,5], [45,50], [70,100], [0,25] $ProblemDifficulty$: **0** (easier)

3.2.3.3 Artificial Data Generation and Classification

The process of *artificial data generation* and *classification* between S and R is described as follows. First, S generates variants of problems for classification (i.e. a population of problems referred as *meta-problems*). S is initialized with a random *meta-problem* containing a list of parameters (i.e `[Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd]`). Next, S effectively generates a set of datasets for an individual problem and sends each instance in the dataset to R.

Thirdly, R needs to successively classify each instance in a dataset as belonging either to 'Class 1' or 'Class 0'. Thus, R sends either '1' for 'Class 1' or '0' for 'Class 0' as suggested by its classifiers. In response, S sends a numerical reward of '1000' for correct classification or '0' is returned to R.

Fourth, based on R's performance, S needs to generate a new problem

where S's objective can be either to make the problem 'easier' by decreasing the problem's difficulty or 'harder' by increasing the problem's difficulty. In the first approach tested, S uses TS to vary the features `F` in the current problem for generating the subsequent problem in the next set of iterations (see Table 3.10).

However, using TS alone in S resulted in TS becoming stuck in local optima (see Section 4.2.2.1). Thus, in order to overcome stagnation in the local optima, a Pittsburgh-style LCSs, A-PLUS, is adapted in S to evolve S's rules. S implements A-PLUS system as an *on-line* version rather than off-line to suit the design of the Two-Cornered Coevolution System. The on-line system offers several advantages over the off-line system which is discussed in Section 5.1. In the original implementation of the A-PLUS system, the datasets are read in batch-mode or off-line mode. Now the datasets are created *on-the-fly*, thus, the system can directly process the datasets instantly. In addition, TS is used to discover the best combination of features `F` in the problem that alters R's classification performance, while A-PLUS evolves S's single rule (i.e. the problem).

An algorithmic description to describe the main task of S and R for problem generation and classification tasks (restricted to binary classification) is shown in Algorithm 12.

3.2.4 Experimental Design

In our implementation, the classification agent (i.e. the Receiver) is executed following *Wilson's explore/exploit scheme* [119], which has become the standard approach in accuracy-based LCSs and uses the same parameter setting as in Phase 1. The generation agent (i.e. the Sender) is a version of *Pittsburgh-style LCSs*, A-PLUS, and is executed in *on-line mode* and implemented as suggested in [109].

Algorithm 12: Algorithmic description for problem generation and classification, Subscript R the Receiver, Subscript S the Sender.

```
1  begin
2      problem ← Sender : generate initial problem to Receiver.
3      while (number of problem less than maximum problems) do
4          while (instance less than maximum instance in dataset) do
5              instance ← Sender : generate instance based on problem.
6              Receiver ← pattern : perceive instance from Sender.
7              GENERATE MATCH SET [M]_R out of [P]_R using instance.
8              GENERATE PREDICTION ARRAY PA_R out of [M]_R.
9              class ← SELECT ACTION according to PA_R .
10             GENERATE ACTION SET [A]_R out of [M]_R according to class.
11             Receiver : execute action class.
12             reward ← Sender : Sender check class and send reward back to Receiver.
13             prediction ← reward : update prediction with current reward of Receiver.
14             UPDATE SET [A]_R using prediction possibly deletion in[P]_R.
15             RUN GA in [A]_R considering instance insertion in [P]_R.
16             if instance equal to maximum instance in dataset then
17                 classificationPerformance : calculate Receiver classification performance.
18             end
19         end
20         Sender : read classificationPerformance
21         problem ← Sender : APPLY TS on problem based on Receiver
           classificationPerformance.
22         ruleset ← Sender : generate rule set based on problem.
23         ruleset ← Sender : APPLY GA to rule set.
24         for (each rule set) do
25             Set rawFitness to zero.
26             EVALUATE accuracy of each classifier.
27             for (each problem) do
28                 CREATE ACTION SET [A]_S of correct matching classifiers.
29                 CALCULATE meanAccuracy of classifiers in [A]_S and add to rawFitness.
30                 if (action set contains a totally accurate classifier) then
31                     DELETE other classifiers in [A]_S that are accurate but subsumed by it.
32                 end
33             end
34             rawfitness = rawFitness/totalProblems.
35             Set rawfitness to Fitness
36             DELETE any totally inaccurate classifiers.
37             DELETE weak classifiers that appear in [A]_S.
38         end
39         SELECT next problem.
40         problem ← Sender : next problem.
41     end
42 end
```

Table 3.12 shows the parameter setting of S for addressing the classification problems (i.e. image-based data and artificial data). The parameters are set according to [109], where a few modifications are made including the number of classifiers in the population and the number of iterations.

Table 3.12: Parameters setting for A-PLUS.

Rule set parameters	Value
Initial number of rules	10
Generality	0.33
Deletion probability	1.0
Subsumption probability	1.0
GA parameters	Value
Maximum population size	100
Selection algorithm	Tournament selection
Tournament size	5
Crossover algorithm	Two-point crossover (binary representation) Arithmetic crossover (real-value representation)
Crossover probability	0.6
Mutation algorithm	Random mutation (binary representation) Random mutation (real-value representation)
Mutation probability	0.001
Fitness threshold	0.005
$ASsubsumption$	YES

The same parameters setting for A-PLUS will be used in all of the experiments throughout the work, unless explicitly otherwise stated. All of the experiments are run 30 times with different random seeds for analysing the results. R's classification performance is calculated from the exploit trials, similar to Phase 1.

3.2.5 Summary

In Phase 2, the Two-Cornered Coevolution System is implemented as a *coadaptive evolution* rather than a real coevolution such as proposed in the Three-Cornered Coevolution Framework by Wilson. In the Two-Cornered Coevolution System, the generation agent (i.e. the Sender) and the classification agent (i.e. the Receiver) are evolving at different levels of structure. S evolved on the level of architecture (i.e. the problem creation), while R evolved on the the level of components (i.e. the classifiers or solutions), where the parameter setting for S and R are fixed. Here, both the rate of convergence to a classification performance level and the maximum achievable performance level itself are not known a priori, so it is difficult to determine when the generation agent should adjust the problem's difficulty. There will be a case when the whole learning system is stable, where the classification agent has successfully addressed the classification problems, hence becoming inactive (i.e. the the classification agent's performance cannot be improved). This is one reason why the Three-Cornered Coevolution System is preferred over the Two-Cornered Coevolution System.

In the Three-Cornered Coevolution System, the problem domain can be tuned autonomously, depending on the two different classification agents' ability to learn (i.e. the Receiver (R) and the Interceptor (I), which will use different techniques of learning) [124], and an active learning system can be established. By introducing a third agent (i.e. the Interceptor), the difference in classification performance between the two classi-

fication agents can be used to direct the generation agent to change the problem's difficulty, whilst the difference in performance encourages exploration of the problem's difficulty. The third agent can also assist in determining whether the problem should be made 'harder' or 'easier' when the classification agents' performance have stagnated. In addition, the Three-Cornered Coevolution System is needed in order to investigate the *coevolutionary* process between the participating agents within the system if the framework is practical. This coevolutionary approach will be further investigated.

3.3 Three-Cornered Coevolution System

The *Three-Cornered Coevolution System* is important as it addresses both how computers can solve interesting problems and how to find interesting problems to solve, which was previously the human investigators task. The overall aim of this phase is to develop the Three-Cornered Coevolution System where three different agents evolve to adapt to and drive the changes of the problem. The name 'three-cornered' derives from having one generation agent, one favoured classification agent, and one classification agent to monitor (i.e. intercept) the learning between the first agent in case it becomes stagnated.

The generation agent (i.e. the Sender) to autonomously generate various problems (i.e. various sets of artificial data) for classification, whilst the classification agents (i.e. the Receiver and the Interceptor) learn the problems and adapt to different feedback returns from the generation agent. Here, the classification agents use different techniques of learning (i.e. either the supervised learning or the reinforcement learning technique) to learn the problem (i.e. one similar problem at a time without interaction), whilst the generation agent determines the type of problems that can be solved by the classification agents. Ultimately, the problem's difficulty is adjusted (i.e. increased or decreased) based on the classification agents'

learning ability (i.e. the difference in classification performance).

3.3.1 Research Objectives

This aim can be broken down into the following objectives.

1. Develop a *Three-Cornered Coevolution System* for addressing classification problems. The system consists of three main agents: the Sender (S), the Receiver (R) and the Interceptor (I). S wants to send a variety of problems and its associated dataset for classification to R and I, whilst R and I need to classify the datasets effectively. R may use different techniques compared to I without interaction between R and I.

2. Evaluate the difference in performance between R and I for learning various types of datasets after a certain number of iterations, as indication for S to increase or decrease the problem's difficulty based on the R's and I's performance.

3. Investigate the competitive relationship between R and I in this coevolutionary process.

4. Investigate the coevolutionary process between S, R and I in the Three-Cornered Coevolution System as each agent adapts and learns the problems while evolving the individual's rules.

3.3.2 Framework Design

The system consists of three main agents: the generation agent (i.e. the Sender (S)) and two classification agents (i.e. the Receiver (R) and the Interceptor (I)) (see Figure 3.8). S is a program to generate various problems for classification, while R and I are programs to learn the artificial data using different techniques of learning (i.e. either the supervised learning or

reinforcement learning). All of the agents evolve using evolutionary computation (i.e. Genetic Algorithm). The system is an extended version of the *Two-Cornered Coevolution System* for addressing the classification problem in Phase 2 (see Section 3.2).

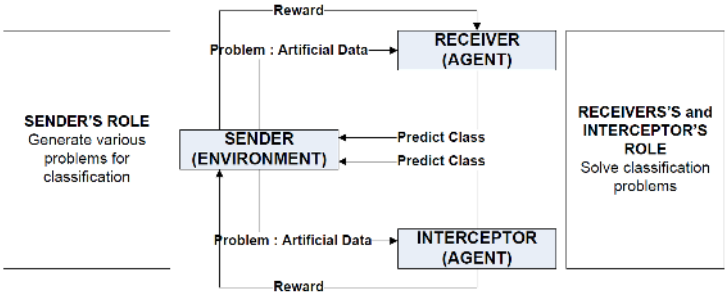


Figure 3.8: Three-Cornered Coevolution System.

Figure 3.8 illustrates the overall design of the system. In this setup, S generates artificial data from various problems for classification that will need to be solved by R and I. Therefore, S activates a programme for problem generation, while R and I activate a programme for classification. R and I learn the classification problems and adapt to different feedback returns from S. However, the action of I may or may not always be competitive with R. Here, R and I use different techniques of learning (i.e. either the supervised learning or reinforcement learning). In the Three-Cornered Coevolution System, all agents evolve to adapt to and drive the changes of the problem and the difference in performance between R and I is used to direct S to change the problem's difficulty.

Here, S can change the problem's difficulty based on the difference in performance between R and I (i.e. below a certain threshold). I as the third agent can assist in determining whether the problem should be made 'harder' or 'easier' when the performances have stagnated. The difference in performance helps to separate whether a current non-optimal performance level is likely due to the problem or to the techniques' abilities, i.e. when there is a difference between the former agents' performances and

the latter. Next, S will adjust the problem's difficulty either to make the problem 'harder' or 'easier'. If the difference in performance between R and I is greater than the threshold value, S will change the problem's difficulty and make the problem to be 'easier'. However, if the difference in performance between R and I is less than the threshold value, S will change the problem's difficulty and make the problem to be 'harder' (see Figure 3.9). Therefore, S needs to consider the ability of R and I when generating a variety of problems to be solved which can be determined based on the difference in performance between R and I (i.e. determining the type of problems that can be solved by R and I).

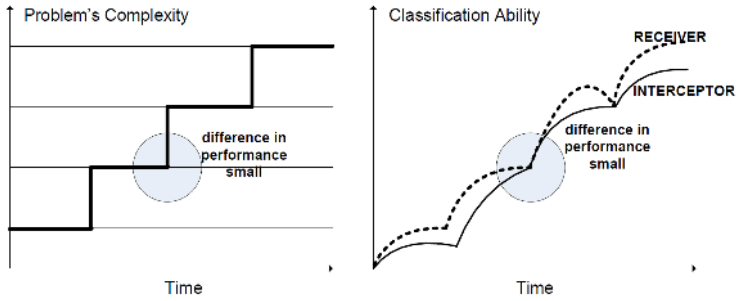


Figure 3.9: Three-Cornered Coevolution System (coevolutionary process).

3.3.3 Artificial Data for Classification

3.3.3.1 Knowledge Representation

S generates various sets of artificial data (problems) for classification, while R and I learn the datasets for each problem and adapt to different feedback returns from S in order to predict the class. R's *condition-action* rule format is similar to Phase 1 (see Table 3.4) and S's *condition-action* rule format is similar to Phase 2 (see Table 3.11).

Table 3.13 illustrates I's *condition-action* rule format. The *condition* specifies each instance in the dataset, while the class is considered to be the *action*. The $condition$ is encoded to be real-value of $real_n = (l_n, u_n)$, where l_n is the lower bound and u_n is the upper bound within the interval $[0, 1]$. The action can be either '1' for 'Class 1', otherwise '0' for 'Class 0'. The last row illustrates I's rules to specify one instance with two data features where each data feature $data_n$ is within the interval of lower bound l_n and upper bound u_n (i.e. [0,1]). I will receive a reward of '1000' for correct classification or '0' for incorrect classification.

Table 3.13: Example of I's condition-action rule format.

I's condition-action rule:
IF<$condition$> THEN<$class$> $condition$ is a list of data feature for each instance containing: [$data_1, data_2, ..., data_i$], where each $data_i : [l_i, u_i]$
[$data_1, data_2$]: $class$ [0.3,0.6], [0.5,0.8] :**0**

3.3.3.2 Artificial Data Generation and Classification

The process of *artificial data generation* and *classification* between S, R and I are provided as follows. First, S generates variants of problems for classification (i.e. a population of problems referred as *meta-problem*). S is initialized with a random *meta-problem* containing a list of parameters (i.e `[Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd]`). Next, S creates a set of artificial data for an individual problem and sends each instance in the dataset to R and I.

Secondly, R and I need to classify each instance as either belonging to 'Class 1' or 'Class 0' and send either '1' for 'Class 1' or '0' for 'Class 0' as suggested by its rules. However, R will use different techniques of learn-

ing compared to I, for classifying the instances (i.e. either the supervised or the reinforcement learning technique). In response, S sends a numerical reward of '1000' for correct classification otherwise '0' returned to both R and I.

Thirdly, S coevolves to tune and adjust the problem's difficulty (i.e. either to increase or decrease the difficulty levels) based on R's and I's ability to learn. S's objective is either to explore maximizing or minimizing one favoured classification agent's performance (i.e. R or I) by adjusting the problem's difficulty. Here, R and I use different techniques of learning (i.e. either the supervised learning or reinforcement learning). Thus, S needs to determine the type of problems that can be solved by R and I by adjusting the difficulty levels and related parameters. S aims to autonomously discover the connection between problem characteristics and the ability of R and I, which affect the classification performance. Based on the difference in performance (i.e. depending on a given threshold value) between R and I, S uses TS to vary the feature `F` in the problem for generating a new meta-problem in the next set of iterations (i.e. either to make the problem 'harder' or 'easier') effectively. Further, R and I learn and evolve to solve the problem (see Figure 3.8).

Finally, S increases the problem's difficulty and generates various 'hard' problems for classification to R and I. S's objective is to minimize one favoured classification agent's performance (i.e. R or I) when increasing the problem's difficulty. Again, R and I evolve to solve the problem, where the capability of using two different learning techniques (i.e. the supervised learning and reinforcement learning) in R and I to solve a certain problem is investigated.

3.3.4 Experimental Design

In our implementation, the generation agent (i.e. the Sender) is executed and set similar to Phase 2, where both of the classification agents (i.e. the Receiver and the Interceptor) are executed following *Wilson's explore/exploit scheme* [119]. However, if the classification agent is a supervised learner (i.e. UCS system), the parameters are set according to [105, 84, 85]. Table 3.14 shows the parameters setting for UCS for addressing the classification problems (i.e. artificial data). UCS is implemented as suggested in [62, 61] and follows the algorithm in Algorithm 13. The same parameters setting will be used in all of the experiments throughout the work, unless explicitly otherwise stated.

All of the experiments are run 30 times with different random seeds for analysing the results. R's and I's performance are calculated from the exploit trials. There will be many iterations of R and I for each iteration of S (i.e. to learn 2,000 instances, R and I are run for 2,000 iterations). One iteration for S is a problem, and one iteration for R and I is an instance from the dataset. After a certain number of iterations (instances), R's and I's classification performance (i.e. average of correct classification performance from the exploit trials over 30 runs) are recorded to measure R's and I's performance on the specified problem.

Table 3.14: Parameters setting for UCS.

Parameter		Value
Population size	N	500
Number of iterations		2,000 (one iteration representing one instance of the problem)
Selection algorithm		Tournament selection
Tournament size	τ	a fraction $\tau = (0, 1]$ of the current action set size, where ($\tau = 0.4$ is the suggested value)
Crossover algorithm		Arithmetic crossover
Crossover probability	χ	0.8
Mutation algorithm		Random mutation
Mutation probability	μ	0.4
Accuracy discount factor	α	0.1
Learning rate	β	0.2
Optimum rule accuracy	acc_0	0.99
Rule's fitness to its accuracy	ν	10
Mutation threshold	m_0	0.2
Covering threshold	r_0	0.4
GA threshold	θ_{GA}	50
Subsume threshold	θ_{sub}	200
Rule deletion threshold	θ_{del}	200
Deletion discount factor	θ	0.1
$GAsubsumption$		YES
$ASsubsumption$		NO

Algorithm 13: Algorithmic description of UCS's (Performance Component) (adapted from [105]).

```
1  begin
2      Perceive a single input string (e.g. current state of the problem) from
       the Sender.
3      Generate a random population of classifiers [P].
4      Build a match set [M] containing all the classifiers in the population [P],
       where the condition matches the input string.
5      Update all classifiers participating in [M].
6      Form the correct set [C] containing the classifiers in match set [M] that
       predict the same class as the label of the current input.
7      if ([C] is empty) then
8          Covering process is activated, a new classifier is created (a
           condition is a generalized version of the input example, an action is
           the same class label).
9          Add this classifier in the population.
10     end
11     Send the selected action to the environment and receive a reward.
12     Activate the credit assignment algorithm.
13 end
```

3.3.5 Summary

The *Three-Cornered Coevolution* is the final research goal. The *Three-Cornered Coevolution System* is a *new coevolution system* where three different agents evolve to adapt to and drive the changes of the problem. The system consists of a generation agent which is referred to as the Sender (S) and two classification agents which are referred as the Receiver (R) and the Interceptor (I). Here, the classification agents evolve to learn various classification problems, while the generation agent coevolves to tune and adjust the problem's difficulty based on the classification agent's ability to learn. Further, as the generation agent increases the problem's difficulty and generates various 'hard' problems for classification, the system will implement the Three-Cornered Coevolution.

3.4 Summary and Way Forward

This chapter provides a detailed design for the *Three-Cornered Coevolution System* that consists of three main phases. All the phases need to be completed sequentially in order to develop a fully operational *Three-Cornered Coevolution System*. The problem domains for each phase is described and the knowledge representation of each agent in the system is illustrated. The experimental design of the system including the experimental setup, the parameter settings, the evaluation metrics and the algorithm of the sub-system for each phase is presented. The next chapter will present the results of implementing the system phase by phase and discuss the relevant findings.

Chapter 4

Results

The main goal of the work for this thesis is to design and develop a new implementation of *Three-Cornered Coevolution System* for addressing the classification tasks. In order to achieve this goal, there are three main phases that need to be established as follows. Firstly, Phase 1 is necessary to create an appropriate problem domain for classification as a test bed that can be evolved and tuned automatically. Secondly, Phase 2 is needed to investigate the generation agent's ability to autonomously tune and adjust the problem's difficulty based on the classification agent's performance. Phase 2 is important to establish a baseline for the coevolutionary system. Finally, Phase 3 is the main research goal to develop the Three-Cornered Coevolution System, which is a *new coevolution LCS*, where three different agents evolve to adapt to and drive the changes of the problem. The details of the three phases have been described in Chapter 3. This chapter will present the experiments and the results of the system phase by phase (see Table 4.1).

Table 4.1: Summary of the experiments for each phase.

Phase	**Experiments**
Phase 1: image-based data.	1 - evaluate R-XCS's performance on the 3 by 3 dimensional patterns. 2 - evaluate R-XCS's performance on the 4 by 4 dimensional patterns. 3 - evaluate R-XCS's performance on the 5 by 5 dimensional patterns.
Phase 1: artificial data.	1 - evaluate R-XCSR's performance on the four problem domains (i.e. `Fn=2,3,4,5`). 2 - investigate R-XCSR's performance on the four problem domains (i.e. `Fn=2,3,4,5`), when `Fan`, `Fcn` and `Fcbl` are increased by 5.
Phase 2: image-based data.	1 - evaluate R-XCS's and S-XCS's performance on simple image-based data. 2 - evaluate R-XCS's and S-XCS's performance on complex image-based data. 3 - evaluate S-XCS's and S-APLUS's performance on simple image-based data.
Phase 2: artificial data.	1 - evaluate S-TS's performance that can either maximize or minimize R-XCSR's performance on the four problem domains (i.e. `Fn=2,3,4,5`). 2 - evaluate S-APLUS-TS's performance that can either maximize or minimize R-XCSR's performance on the four problem domains (i.e. `Fn=2,3,4,5`).
Phase 3: artificial data.	1 - evaluate S-TS's and S-APLUS-TS's performance that can either maximize or minimize I-UCS's performance on the four problem domains (i.e. `Fn=2,3,4,5`). 2 - investigate I-XCSR's and I-UCS's ability to trigger S-APLUS-TS to change the problems difficulty when the threshold values were set to: 1) 10%, and 2) 20% on the four problem domains (i.e. `Fn=2,3,4,5`). 3 - investigate I-XCSR's and I-UCS's ability either suitable to be as a triggering agent or learning agent on the four problem domains (i.e. `Fn=2,3,4,5`). 4 - investigate R-XCSR's and I-UCS's capability when S-APLUS-TS increases the difficulty levels for the four problem domains (i.e. `Fn=2,3,4,5`), where S-APLUS-TS is tasked either minimize I-UCS's or R-XCSR's performance one at a time.

4.1 Phase 1: An Evolvable Problem Generator

The overall aim of Phase 1 is to develop an *automated evolvable problem generator* for generating various problems for classification. A set of experiments is performed to verify the automated evolvable problem generator. This is tested by adopting an accuracy-based LCSs in the classification agent for addressing the evolved problems. The generation agent (i.e. the automated evolvable problem generator, termed as the Sender (S)) autonomously evolves the problem and generates different types of image-based data (or artificial data) for classification, while the classification agent (i.e. the accuracy-based LCS, termed as the Receiver (R)) evolves to learn the image-based data (or artificial data). In Phase 1, the classification agent was trained and tested on various problem domains in order to investigate limits of its performance, so that appropriate problem domains have been developed as a test bed that can be further enhanced in Phase 2. In this phase there was no feedback from the Receiver to the Sender.

4.1.1 Image-based Data for Classification

In this section, the results of the classification agent (i.e. the Receiver (R)) that used an *accuracy-based LCS*, XCS, in both of the *training mode* and the *testing mode* on the three problem domains[1] are presented.

First, R-XCS was trained[2] to learn various patterns (image-based data) on various dimensionalities to determine the limits of its performance. Next, R-XCS was tested[3] on unseen patterns (image-based data) of the

[1]Note: *problem domain* refers to the three problem domains of image based-data (i.e. `3 by 3`, `4 by 4` and `5 by 5 pattern dimensional mapping`), *instance* refers to a generated pattern from any problem domain, *problem* refers to one problem domain.

[2]In the *training mode*, the system learns each pattern alternately using *explore* and *exploit* scheme.

[3]In the *testing mode*, the system exploits previous knowledge in the training mode to learn a new pattern.

same problem domains to verify that sufficient knowledge for classifying the patterns (image-based data) to the correct class had been created.

4.1.1.1 Experimental Results

For all of the experiments, the classification agent (i.e. the Receiver) was executed following *Wilson's explore/exploit scheme*[4] [119], which has become the standard approach in accuracy-based LCSs as described in Section 3.1.5. In *explore scheme*, the system selects an action at random from those advocated by the matching rules (choosing the action randomly). In *exploit scheme*, the system deterministically selects the action which is most highly recommended by the matching rules (choosing the best action). The results were recorded from the *exploit scheme*, where R-XCS selected the best action to effect S in order to obtain the highest reward. R-XCS was designed as an *on-line* learner (or agent).

The **first set of experiments** was performed in order to evaluate R-XCS's classification performance on the `3 by 3 dimensional pattern mapping`. The population size N was set to 500, and the number of iterations (patterns) was set to 220,000. These parameters were set according to [54], where a few modifications were made to improve the efficiency, including the population size N and the number of iterations. Figure 4.1 shows the average of R-XCS's classification performance in the *training mode*. R-XCS is able to achieve 100% performance by classifying the patterns correctly after 18,100 iterations of learning. Figure 4.2 shows R-XCS's classification performance in the *testing mode*. The average of R-XCS's classification performance in the testing mode to learn 100 patterns (exemplars) is approximately 99.0%, and for 500 patterns is approximately 99.2%.

[4]For example, to learn 2,000 patterns R-XCS is run for 2,000 iterations. In the first iteration, R-XCS chooses the action randomly (*explore scheme*), while in the second iteration, R-XCS chooses the best action (*exploit scheme*). For 2,000 iterations, there will be 1,000 *explore scheme* and 1,000 *exploit scheme*.

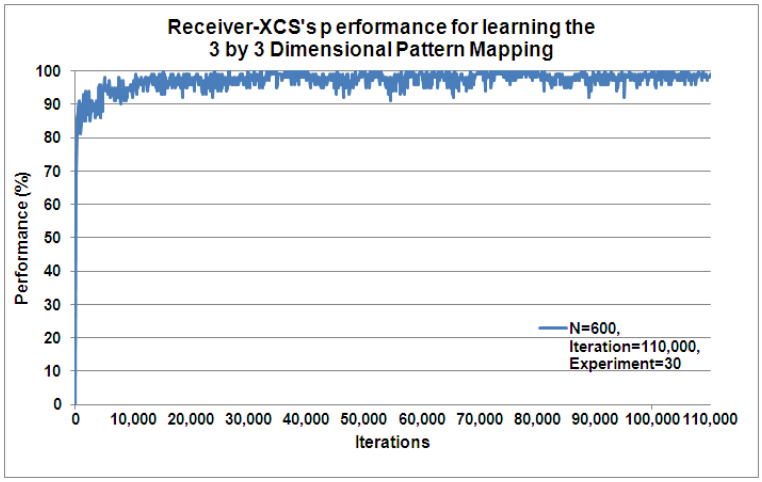


Figure 4.1: Average of R-XCS's classification performance in the *training mode* to learn the 3 by 3 dimensional pattern mapping from Exploit trials for 220,000 patterns (110,00 Exploit, 110,00 Explore) over 30 runs.

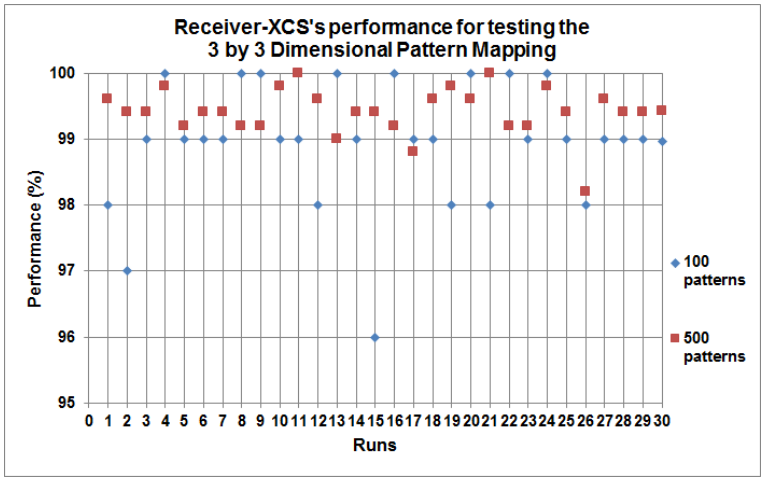


Figure 4.2: R-XCS's classification performance in the *testing mode* to test the 3 by 3 dimensional pattern mapping from 30 runs.

Based on R-XCS's performance, R-XCS's rules was further studied. Table 4.2 describes R-XCS's rules on the `3 by 3 dimensional pattern mapping`. The classifier's experience is given in the second column. The classifier's associated parameters (i.e. fitness, prediction and accuracy) are given in the next three columns. The final two columns provide the classifier's condition and action for mapping the pattern.

Table 4.2: Examples of R-XCS's rules (`3 by 3 dimensional pattern mapping`).

Number	Experience	Fitness	Prediction	Accuracy	Condition	Action
1	187480	0.99	1000	1	0###0###0	0
2	182599	0.83	1000	1	#1##1##1#	1
3	181647	0.73	1000	1	1##1##1##	1
4	181013	0.62	904.1	0	#0#0####0	0
5	182079	0.62	1000	1	1###1###1	1
6	90903	0.61	1000	1	0####000#	0
7	186188	0.59	1000	1	######111	1
8	362112	0.51	786.8	0	##0#0####	0
9	90863	0.46	1000	1	#0##0#0#0	0
10	181862	0.03	903.5	0	###1##1#1	1

The **second set of experiments** was performed to evaluate R-XCS's classification performance on the `4 by 4 dimensional pattern mapping`. The population size N was set to 5,000 and R-XCS was trained for up to 100,000 iterations. These parameters were set according to [54], where a few modifications were made to improve the efficiency, including the population size N and the number of iterations. Figure 4.3 shows the average of R-XCS's classification performance in the *training mode*, where R-XCS is able to achieve 100% performance after 22,600 iterations. Figure 4.4 shows R-XCS's classification performance in the *testing mode*, where the average of R-XCS's classification performance to learn 100 patterns (exemplars) is 100.0%, and for 500 patterns is approximately 99.9%.

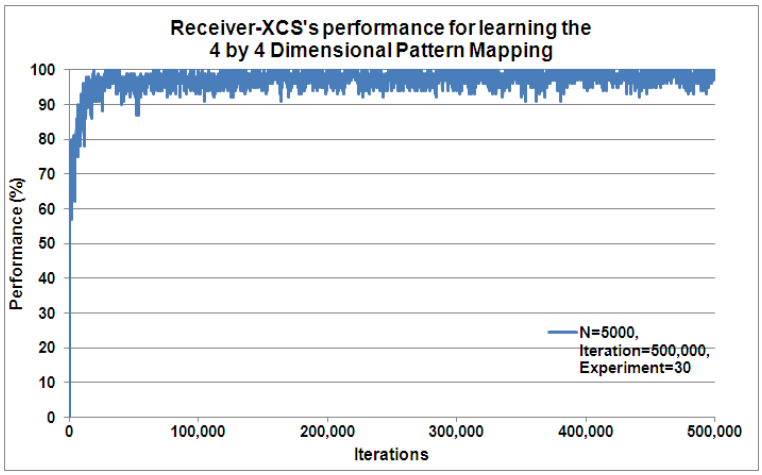


Figure 4.3: Average of R-XCS's classification performance in the *training mode* to learn the 4 by 4 dimensional pattern mapping from Exploit trials for 1000,000 patterns (500,00 Exploit, 500,00 Explore) over 30 runs.

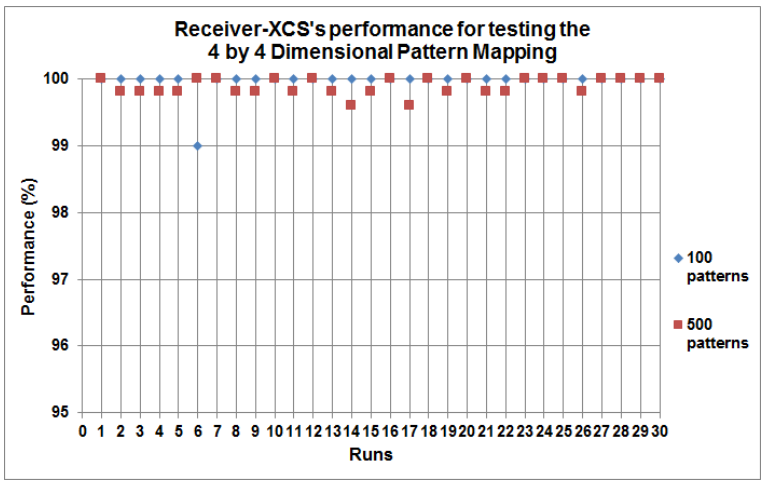


Figure 4.4: R-XCS's classification performance in the *testing mode* to test the 4 by 4 dimensional pattern mapping over 30 runs.

The **third set of experiments** was performed to evaluate R-XCS's classification performance on the `5 by 5 dimensional pattern mapping`. The population size N was set to 50,000 and R-XCS was trained for up to 100,000 iterations. These parameters were set according to [54], where a few modifications were made to improve the efficiency, including the population size N and the number of iterations. Figure 4.5 shows that R-XCS is able to achieve 100% performance in the *training mode* by correctly classifying the patterns after 68,500 iterations. Figure 4.6 shows R-XCS's classification performance in the *testing mode*, where the average of R-XCS's classification performance to learn 100 patterns (exemplars) is 100%, and is approximately 99.9% for 500 patterns.

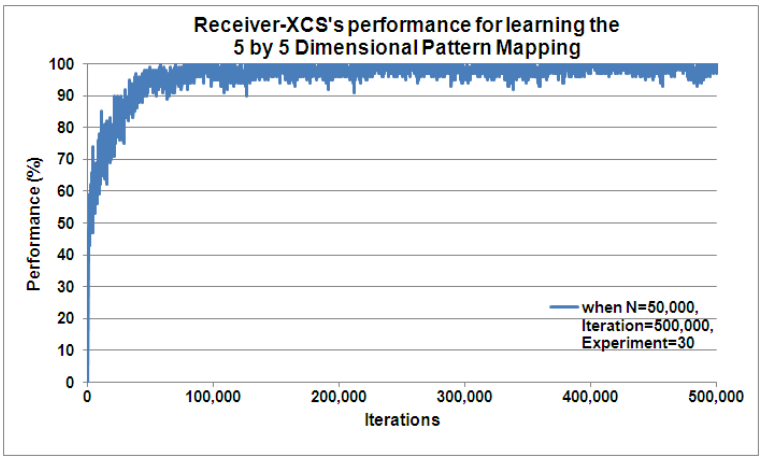


Figure 4.5: Average of R-XCS's classification performance in the *training mode* to learn the 5 by 5 dimensional pattern mapping from Exploit trials for 1000,000 patterns (500,00 Exploit, 500,00 Explore) over 30 runs.

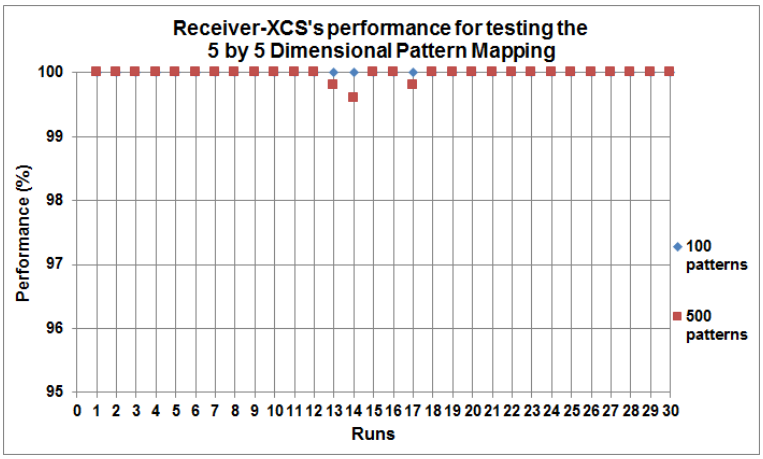


Figure 4.6: R-XCS's classification performance in the *testing mode* to test the 5 by 5 dimensional pattern mapping from 30 runs.

Table 4.3 gives R-XCS's computational time for classifying the patterns either belonging to 'Class 1' or 'Class 0' in the *training mode*. The second column indicates the number of iterations (instances) that R-XCS needs to learn, the third column gives the number of classifiers in R-XCS and the last column gives the computational time for R-XCS to solve the problem.

Table 4.3: Average of R-XCS's computational time over 30 runs (*training mode*) to learn image-based data.

Problem Domain	**Instances (Iterations)**	**Classifiers**	**Computational Time (minutes)**
3 by 3	220,0000	500	1.0
4 by 4	1000,000	5,000	2.4
5 by 5	1000,000	50,000	88

Figure 4.7 plots the average of R-XCS's computational time for classifying the patterns on the three problem domains in the *training mode* over 30 runs. The computational time increases exponentially when the dimensionality grows.

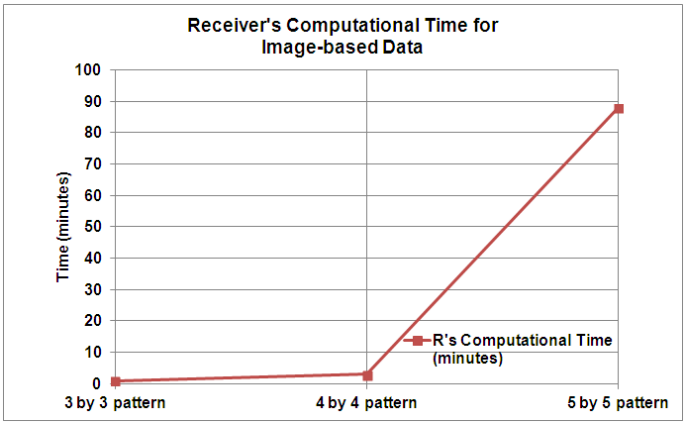


Figure 4.7: Average of R-XCS's computational time over 30 runs (*training mode*) to learn image-based data.

4.1.1.2 Discussions and Findings

The goal to develop an *automated evolvable problem generator* in S has been achieved. S was able to generate a scalable problem domain (i.e. image-based data or (patterns)) that can be evolved in both of the dimensionality and the problem's difficulty. Further, R-XCS learned the generated patterns successfully. However, the total execution time required was increased as would be expected when the dimensionality and the difficulty of the problem grew exponentially.

First, R-XCS has been applied to the three problem domains to solve and learn various image-based data (patterns); mapping all the patterns to the associated class. The dimensionality ranges from the `3 by 3, 4 by 4, 5 by 5` with the length of 9, 16 and 25 bits input state (string) and can take up to any number of n by n dimensional patterns. S was successfully

created and manipulated the problem domain autonomously. S was able to generate different types of patterns at various dimensionalities, while R-XCS evolved to learn the generated patterns and correctly classify the patterns according to the specified class.

Next, R-XCS was tested on the three problem domains to determine the limits of its performance. R-XCS was able to learn the generated patterns successfully for the three problem domains (i.e. `3 by 3, 4 by 4` and `5 by 5 dimensional pattern mapping`). The results suggested that R-XCS was able to learn the generated patterns within the set parameters successfully (i.e. different setting of the population size and the number of iterations). Utilising accuracy-based LCS, XCS, R was able to classify the patterns to the correct class in these three problem domains. Since XCS is designed to be an *on-line learning system* through reinforcement learning, R-XCS requires a longer training time to achieve its best performance. In this problem domain, the `3 by 3 dimensional pattern mapping` was considered as 'easy' problem when R-XCS was able to classify the generated patterns based on its previous knowledge (i.e. 500 rules) compared to the `4 by 4` and `5 by 5 dimensional pattern mapping` which is considered as 'hard' problem.

It was found that the problem's difficulty was related to the size of *n* by *n* dimensional pattern mapping. As the dimensionality increases, a higher number of patterns can be generated by S, thus a bigger search space needs to be solved by R-XCS. Furthermore, the results showed that the execution time was different between *n* by *n* dimensional pattern mapping. In fact, the computational time was related to the size of *n* by *n* dimensional pattern mapping. The total number of the patterns increased dependent on 2^k (where $k = n \times n$ of the *n* by *n* dimensional pattern mapping). For example, for the `3 by 3 dimensional pattern mapping`, *k* can be calculated as $3 \times 3 = 9$, therefore at least $2^9 = 512$ various patterns need to be learned, in order for R-XCS to achieve optimal performance. Meanwhile, the number of patterns which R-XCS needs to distinguish can become very large

as the dimensionality increases (i.e. 25 bits pattern to be mapped `5 by 5 dimensional pattern mapping`).

4.1.2 Artificial Data for Classification

In this section, the results of the classification agent (i.e. the Receiver (R)) that used an *accuracy-based LCSs with real-value*, XCSR, in the training and the testing mode on the four problem domains[5] are presented. The surface landscape of R-XCSR's classification performance is illustrated to show the *trade-off surface* for each problem domain that alters R-XCSR's classification performance when a certain feature in the problem-specific parameters (i.e. `[Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd]`) is adjusted.

First, R-XCSR was trained with various artificial data on the four problem domains to determine the limits of its performance. Secondly, R-XCSR was tested on various artificial data on the same problem domains to verify that sufficient knowledge for classifying the instances to the correct class had been developed. Thirdly, R-XCSR was evaluated on various problems, where different combinations of features of the problem had been altered (i.e. increasing and decreasing value of certain features, such as class balance (`Fcbl`) and noise levels (`Fan` and `Fcn`)), to explore the surface landscape of R-XCSR's classification performance.

[5]Note: *problem domain* refers to various datasets with different number of data features in each dataset depending on value *Fn* (i.e. `Fn=2` to `Fn=5`), *instance* refers to an instance in each dataset, *problem* refers to a problem from any problem domain that contains a problem-specific parameters (i.e. `[Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd]`.

4.1.2.1 Experimental Results

The **first set of experiments** was performed in order to evaluate R-XCSR's classification performance on the four problem domains (i.e. `Fn=2` to `Fn=5`), where the features `F` in the problem were set to `[Fc=1 Fd=0 Fi=0 Fr=0 Fan=0 Fcn=0 Fcbl=50 Fcbd=0]` and the dataset contained 2,000 instances. Figure 4.8 illustrates the average of R-XCSR's classification performance over 30 runs in the *training mode* on the four problem domains. The results show that R-XCSR is able to learn the dataset with varied performance on the four problem domains. Figure 4.9 shows R-XCSR's best classification performance in the *training mode* on the same problem set-up of the four problem domains. R-XCSR obtains an average performance of 99.5% when `Fn=2`, 98.8% when `Fn=3`, 98.9% when `Fn=4` and 98.6% when `Fn=5`. Figure 4.10 shows R-XCSR's classification performance in the *testing mode* on the same problem set-up on the four problem domains. R-XCSR obtains average performance of 85.4% when `Fn=2`, 79.7% when `Fn=3`, 77.7% when `Fn=4` and 76.9% when `Fn=5`. The results suggest that the problem domain `Fn=5` is the 'hardest' problem compared to the others. Meanwhile, the problem domains `Fn=2` and `Fn=3` are the 'easiest' problems, as expected.

Table 4.4 provides R-XCSR's computational time for classifying 2,000 instances for the given problem setting `[Fc=1 Fd=0 Fi=0 Fr=0 Fan=0 Fcn=0 Fcbl=50 Fcbd=0]` on the four problem domains in the *training mode*. The second column indicates the number of iterations (instances) that needs to be learned by R-XCSR, the third column gives the number of classifiers in R-XCSR and the last column gives R-XCSR's computational time to solve the problem.

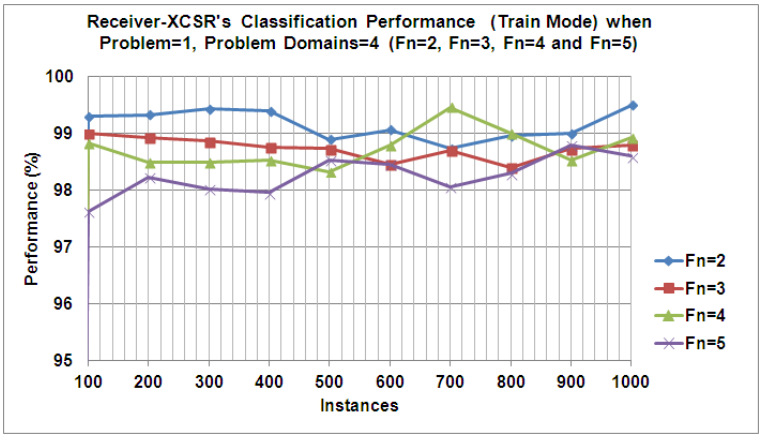


Figure 4.8: Average of R-XCSR's classification performance in the *training mode* on the four problem domains from Exploit trials for 2,000 instances (1,00 Exploit, 1,00 Explore) over 30 runs.

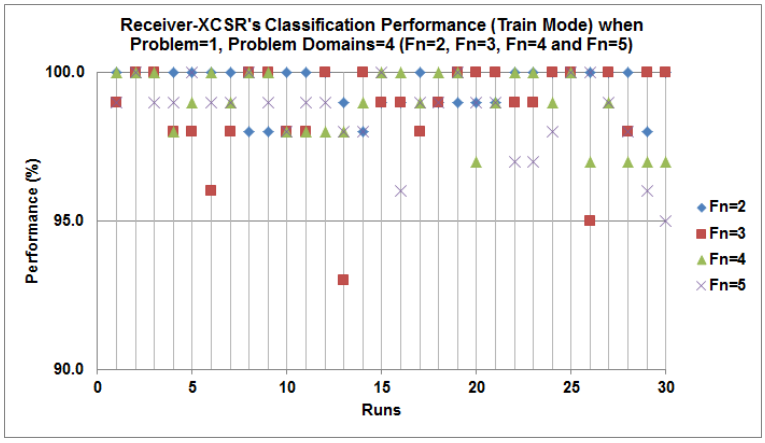


Figure 4.9: R-XCSR's best classification performance in the *training mode* on the four problem domains from 30 runs.

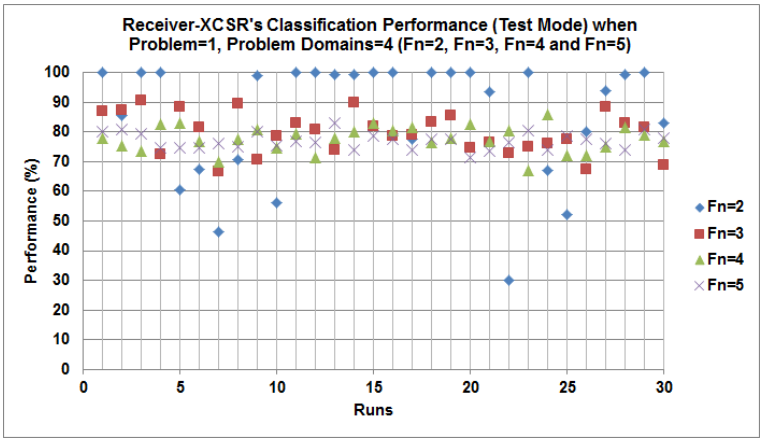


Figure 4.10: R-XCSR's classification performance in the *testing mode* on the four problem domains from 30 runs.

Table 4.4: Average of R-XCSR's computational time over 30 runs (*training mode*) to learn artificial data.

Problem Domain	**Instances**	**Classifiers**	**Computational Time (minutes)**
Fn=2	2000	500	1.5
Fn=3	2000	500	2.5
Fn=4	2000	500	3.5
Fn=5	2000	500	4.5

Figure 4.11 plots the average of R-XCSR's computational time for classifying 2,000 instances on the four problem domains in the *training mode* over 30 runs to learn artificial data. The computational time increases linearly when the dimensionality (i.e. number of data features in the dataset) grows.

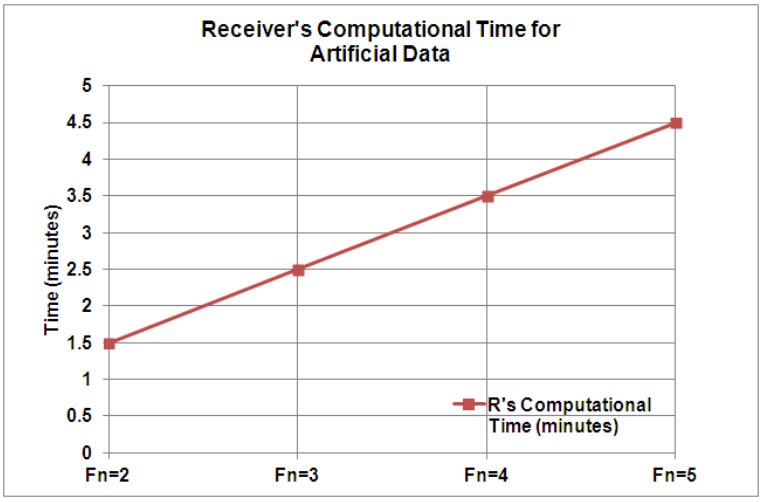


Figure 4.11: Average of R-XCSR's computational time over 30 runs (*training mode*).

The **second set of experiments** was conducted to investigate R-XCSR's classification performance with various problems, where different combinations of features `F` in the problem was altered (i.e. by increasing and decreasing the value of certain features `F` such as *class balance* (i.e. `Fcbl`) and *noise level* (i.e. `Fan` and `Fcn`)). In each problem domain, the value of noise that applies to the action (i.e. `Fan`) and the value of noise that applies to the condition (i.e. `Fcn`) are increased by 5 within the range of 5 to 50. The value of class balance (i.e. `Fcbl`) is also increased by 5 within the range of 50 to 100, while other feature values are set to `[Fc=1 Fd=0 Fi=0 Fr=0]`.

Figure 4.12 shows R-XCSR's classification performance when noise that applies to the action `Fan` is set within the range of 5 to 50, while the class balance `Fcbl` is set within the range of 50 to 100. Both values are increased by 5. R-XCSR achieves a good performance of 85-100%, when the class balance `Fcbl` is within 50-100% and the noise level that applies to the action `Fan` is within 0-10% for the problem domain `Fn=2`.

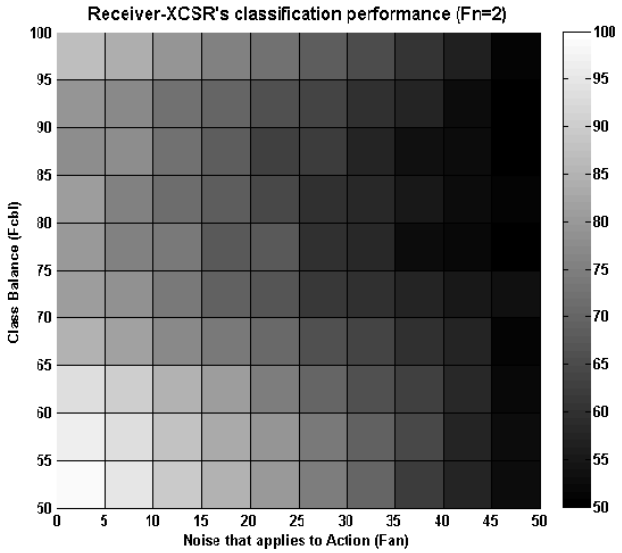


Figure 4.12: Trade-off surface of R-XCSR's classification performance (based on the average of R-XCSR's classification performance in *training mode* for learning a binary classification problem over 30 runs), when `Fan` and `Fcbl` is increased by 5 for `Fn=2`.

Figure 4.13 shows R-XCSR's classification performance when noise that applies to the condition `Fcn` is set within the range of 5 to 50, while the class balance `Fcbl` is set within the range of 50 to 100. Both values are increased by 5. R-XCSR achieves a good performance of 90-100%, when the class balance `Fcbl` is within the range of 50-65% and the chances of noise level `Fcn` being applied to data features `f` is in the range 0-50% for the problem domain `Fn=2`.

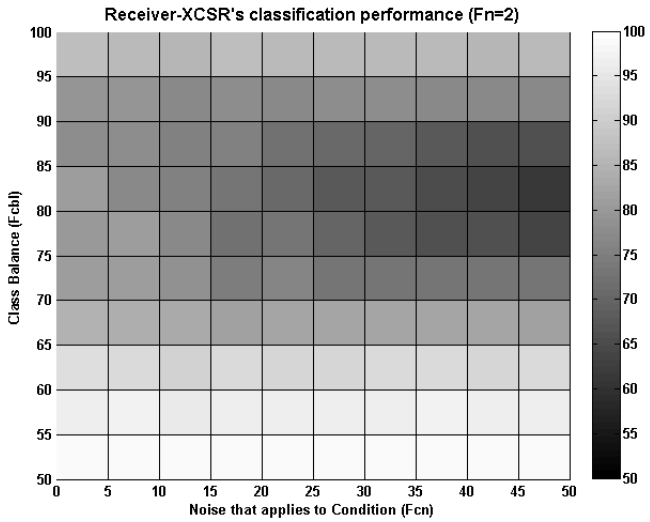


Figure 4.13: Trade-off surface of R-XCSR's classification performance (based on the average of R-XCSR's classification performance in *training mode* for learning a binary classification problem over 30 runs), when `Fcn` and `Fcbl` is increased by 5 for `Fn=2`.

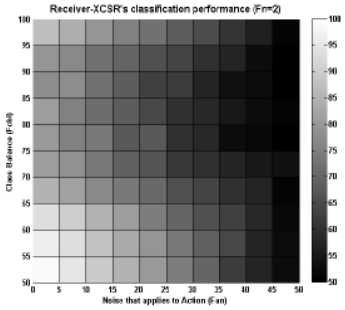
Figure 4.14 shows the *trade-off surface* of the four problem domains (i.e. `Fn=2` to `Fn=5`) that alters R-XCSR's classification performance. R-XCSR's classification performance on the four problem domains are varied but show the same pattern (i.e. the effects of changing the value of `Fan`, `Fcn` and `Fcbl` either will increase or decrease R-XCSR's classification performance). If there is no gradient in difficulty that exists in the problem, then it would be impossible for S to tune the problem to being either 'harder' or 'easier' for R-XCSR to learn.

Note, 100% performance is not reached due to limiting the number of classifiers and training instances. The classification performance improves as the class imbalance ratio increases beyond 90% as the majority class facilitates general (possibly over-general) classifiers and the crude classifiers (deliberately). Fitness function does not take this into account. The results show that a gradient in difficulty exists in relation to features in the problem. This enumerated-information is useful to set up an initial problem for S and determine whether S can vary the difficulty levels appropriately in Phase 2.

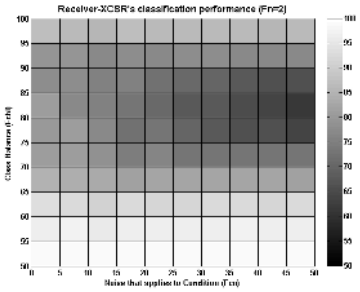
Table 4.5 gives R-XCSR's computational time for one surface landscape in the *training mode*, when a different combination of features in the problem is altered (i.e. `Fc=1, Fan=0 to 50, Fcbl=50 to 100`). The second column indicates the number of iterations (instances), the third column gives the number of classifiers and the last column gives the computational time for R-XCSR to solve the problem.

4.1.2.2 Discussions and Findings

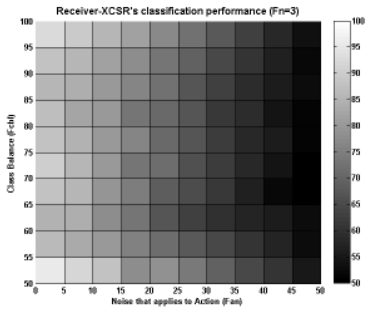
The goal of this section is to develop an *automated evolvable problem generator* in S to generate various problems for classification (i.e. artificial data). The goal has been achieved because S was able to generate various datasets at different levels of difficulty. Further, R-XCSR was able to learn the datasets successfully. However, the computational time required is increased when the number of data features and the instances grow and



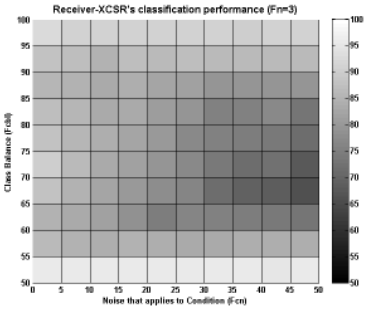
(a) Fan and Fcbl is increased by 5.



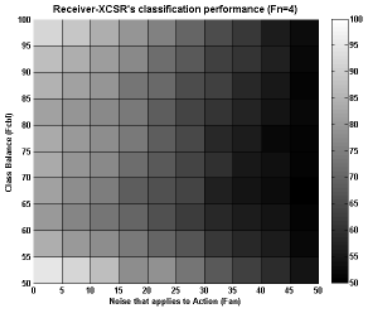
(b) Fcn and Fcbl is increased by 5.



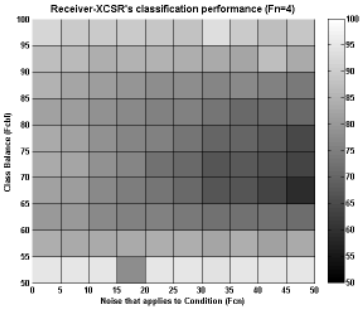
(a) Fan and Fcbl is increased by 5.



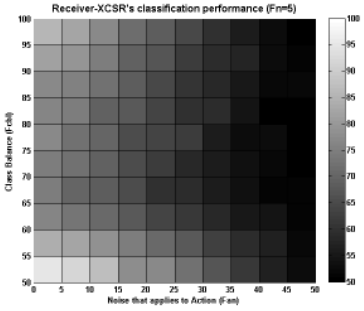
(b) Fcn and Fcbl is increased by 5.



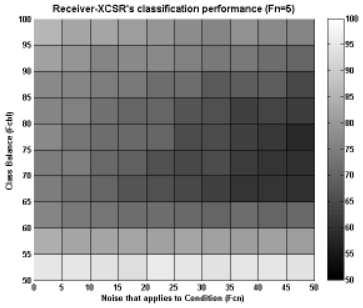
(a) Fan and Fcbl is increased by 5.



(b) Fcn and Fcbl is increased by 5.



(a) Fan and Fcbl is increased by 5.



(b) Fcn and Fcbl is increased by 5.

Figure 4.14: Trade-off surface of R's classification performance (based on the average of R's classification performance in training mode for learning a binary classification problem over 30 runs).

Table 4.5: R-XCSR's computational time for one surface landscape (*training mode*) to learn artificial data.

Problem Domain	**Instances**	**Classifiers**	**Computational Time (minutes)**
Fn=2	2000	500	5.0
Fn=3	2000	500	8.3
Fn=4	2000	500	11.6
Fn=5	2000	500	13.3

the problem's difficulty become harder, as would be expected.

In the first set of experiments, R-XCSR was applied to the four problem domains to solve and learn the generated datasets, determining the class for all instances of the four problem domains namely `Fn=2` to `Fn=5`. S successfully created and manipulated the problem domain autonomously. S generated various problems for classification based on the list of parameters `[Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd]`, while R-XCSR evolved to learn all instances for each dataset.

In order to explore R-XCSR's performance landscape on the four problem domains, a second set of experiments was performed, where the values of certain features `F` in the problem (i.e `Fan`, `Fcn` and `Fcbl`) were increased and decreased. The results showed that R-XCSR was able to learn the datasets with different performances depending on the 'set-up of the problem'. Based on R-XCSR's classification performance from the surface landscape, it was identified that a gradient in difficulty exists in relation to features `F`. The results suggested that S should be able to vary the difficulty levels of the problem by using any appropriate method once the features that altered R-XCSR's classification performance were known. The problem's difficulty in this domain was related to the the parameters `[Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd]`. Therefore, the problem's difficulty can be increased or decreased by changing the values of the features `F` in

the problem.

In all of the experiments, the computational time was varied between different 'set-ups of the problem'. The computational time was not only related to the number of features `F` in the problem, but was also related to the total instances in the dataset. For example, the computational time to train R when the problem was set to `[Fn=2 Fc=1 Fd=0 Fi=0 Fr=0 Fan=5 Fcn=0 Fcbl=50 Fcbd=0]` was 1 minute 30 seconds, and to repeat the whole process for 30 runs, R-XCSR required approximately 45 minutes (see Table 4.4). It was estimated that to complete the whole problem domain R-XCSR required approximately 23 hours (45 minutes $\times$ total attributes which is 2^9).

This means, increasing the number of features `F` will also increase the computational time and the problem's difficulty as R-XCSR needs to learn a larger number of instances and requires more iterations (time) to solve the problem. In all of the experiments, the population size N was limited to 500 classifiers and R-XCSR learnt only 2,000 instances (i.e. R-XCSR was run for 2,000 iterations). Both values are low for standard LCSs, in order to reduce the training times as the overall 'meta-problem' task is time consuming. Due to this constraint, 100% performance was not reached as the number of classifiers and the number of training instances were limited.

Although the system can be extended to produce a maximum value of features `F` and `f`, the work for this thesis only focuses on identifying feature values `F` that affect R-XCSR's performance (i.e. controlling for possible confounding variables to the problem's difficulty). Table 4.5 provides the computational time to complete one surface landscape in order to explore R-XCSR's classification performance when different combinations of features `F` in the problem are altered.

Instead of covering every possible combination of features `F` in the problem, the problem is divided to the precision of 5% (i.e. `Fan`, `Fcn` and `Fcbl` is increased by 5) in order to evaluate R-XCSR's classification performance, which has reduced the computational time. For example, the

initial problem is set to `[Fn=2 Fc=1 Fd=0 Fi=0 Fr=0 Fan=5 Fcn=0 Fcbl=50 Fcbd=0]`, later the value of `Fan` and `Fcn` are increased by 5% within the range of 5% to 50%, and value of `Fcbl` is also increased by 5% within the range of 50% to 100%. Completing one problem landscape took only approximately 5 minutes. Thus, the system used less computational time through this method compared to the enumerated method (i.e. to cover every single possible combination of features `F` in the problem), in order to investigate R-XCSR's classification performance.

4.1.3 Summary and Way Forward

In Phase 1, an automated method (i.e. automated evolvable problem generator) to generate various image-based data (patterns) and artificial data for classification, which were directly tested by two Learning Classifier Systems (i.e. XCS or XCSR), was created. Both the problems (i.e. the image-based data and the artificial data) and the solution (i.e. XCS or XCSR) evolved in parallel, with the accuracy-based LCSs attempting to learn the evolving image-based data (or artificial data). This method helps to empirically test the learning bounds of accuracy-based LCSs in the problem domain. The accuracy-based LCSs, were able to learn the generated image-based data (or artificial data) successfully. However, the time required increased when the dimensionality increased as the classification agent (i.e the Receiver) needed to solve a higher number of the generated image-based data (or artificial data).

It is hypothesized that there will be a scenario when the image-based data (or artificial data) is too simple, and the classification agent learns the image-based data (or artificial data) easily and becomes stagnant. There will be also a scenario when the image-based data (or artificial data) is too complex, and the classification agent cannot learn and again becomes stagnate. Within this domain, if the problem's difficulty can be increased or decreased at an appropriate level, the classification agent's performance can

be investigated and further tuning can be performed. In the next section, a novel method on how to increase or decrease the problems' difficulty based on the classification agent's learning ability will be investigated. Thus, a study on how the classification agent's performance depends on the parameter values of the problem or how to choose parameter values in the problem that can optimize the classification agent's performance , will be conducted.

4.2 Phase 2: Two-Cornered Coevolution System

The overall aim of Phase 2 is to develop an *automated evolvable generation agent* for generating various classification problems that have an ability to autonomously tune and adjust the problem's difficulty based on the classification agent's performance. A set of experiments was performed to investigate the generation agent's ability and the classification agent's performance in this domain. The generation agent (i.e. the Sender (S)) autonomously evolves the problem and generates different types of image-based data (or artificial data) for classification, while the classification agent (i.e. the accuracy-based LCSs, termed as the Receiver (R)) evolves to learn the image-based data (or artificial data). The generation agent either increases or decreases the problem's difficulty in order to maximize or minimize the classification agent's performance.

4.2.1 Image-based Data for Classification

In this section, the results of the generation agent (i.e. the Sender (S)) and the classification agent (i.e. the Receiver (R)) in the *Two-Cornered Coevolution System* are presented. Both the problem domain and the solution are evolved autonomously (i.e. the generation agent created various problems and the associated sets of patterns, while the classification agent learnt each set of patterns).

The first set of experiments was performed in order to investigate R-XCS's and S's performance on the two problem domains (i.e. simple image-based data and complex image-based data). Next, the second set of experiments was performed in order to evaluate the effectiveness of using different style LCSs in S (i.e. either Michigan-style LCSs or Pittsburgh-style LCSs) to evolve S's rule. In Phase 2, S should be able to tune the problem's difficulty autonomously depending on R-XCS's ability to learn, to make the problem either 'hard' or 'easy'.

First, R-XCS and S were applied to the two problem domains: *simple image-based data* (i.e. three features in the problem can be adjusted) and *complex image-based data* (i.e. more than three features in the problem can be adjusted). Next, S was evaluated to predict the difficulty levels of the problem based on R-XCS's classification performance. If S is able to predict the difficulty levels correctly, then S can tune the problem's difficulty of the next problem appropriately based on R-XCS's ability to learn (i.e. either to predict the problem 'hard' or 'easy'). A level of 95% classification performance was chosen to separate 'hard' from 'easy' problems. This value was chosen based on the previous results in Phase 1, where humans set the levels of the difficulty so that if the classification performance is greater than 95% for each problem, the problem is categorized as 'easy', otherwise 'hard'.

Furthermore, the effectiveness of using two different styles of LCSs in S to evolve its rules on the simple image-based data was compared, such that S can generate the next problem at the appropriate levels of difficulty more effectively. Here, R-XCS sends feedback to S, therefore S can tune and adjust the problem's difficulty based on R-XCS's performance.

4.2.1.1 Experimental Results

The previous results in Phase 1 showed that R-XCS was able to learn the generated patterns (image-based data) for the `3 by 3 dimensional pattern mapping` successfully (see Figure 4.15). Figure 4.15 shows R-XCS's performance for learning three problems that is varied (i.e. above 90%) for the first 10,000 iterations. Therefore, 10,000 iterations are chosen to be set in learning the *simple image-based data* in order to distinguish the difficulty levels in the problem.

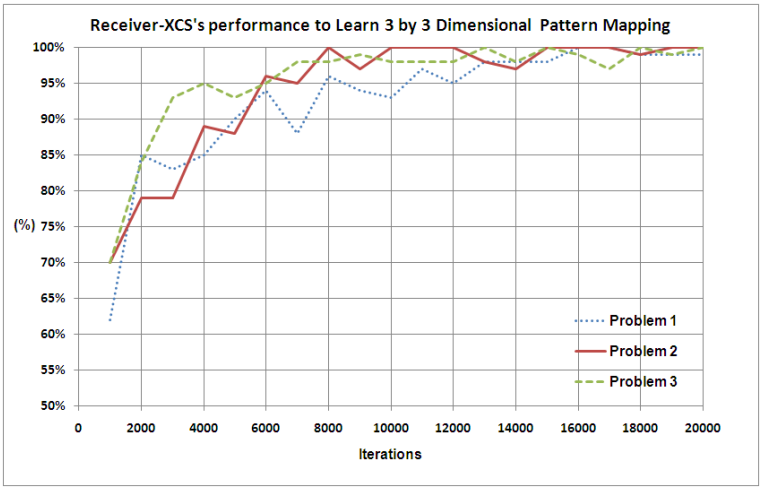


Figure 4.15: R-XCS's classification performance on a series of the generated problems from S-XCS. Note the performance at 10,000 iterations is shown in Figure 4.16 for each problem.

The **first set of experiments** was performed to evaluate R-XCS's and S-XCS's performance on the *simple image-based data* (i.e. three features can be adjusted such as [`PatternDimension`, `PatternOrientation`, `PatternOperator`]). Figure 4.16 presents the average of R-XCS's classification performance for each problem, where R-XCS needs to recognize the generated patterns (image-based data) from various problems (i.e. 10,000 patterns for each individual problem, where the total number of problems

is 1000). However, results suggest that the separation value of 95% between 'hard' and 'easy' problems does not lead to a balanced class distribution, where R-XCS's performance is approximatley 95% and above for only a small number of problems.

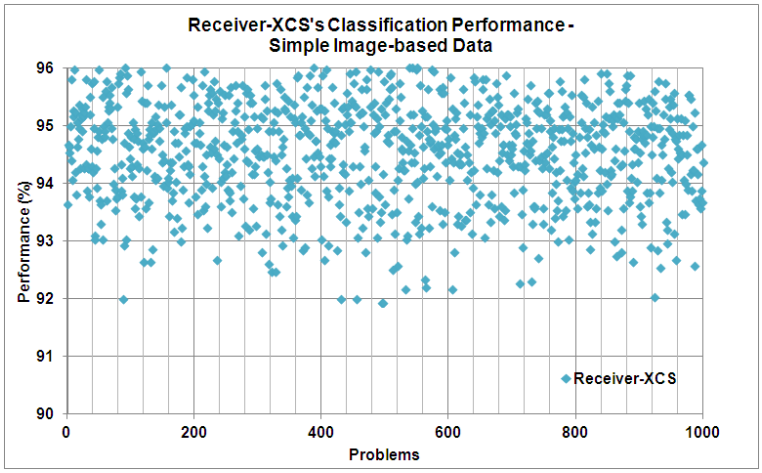


Figure 4.16: R-XCS's classification performance in the *training mode* to learn different types of binary problems for 1,000 problems.

Figure 4.17 depicts the average of S-XCS's performance[6] for predicting the problem's difficulty for 1,000 problems. R-XCS sends feedback to S-XCS for predicting R-XCS's learning ability to learn different types of problem for classification (i.e. the problem is categorized as either '1' if 'hard' or '0' if 'easy').

[6]S-XCS learns each problem alternately using *explore* and *exploit* scheme.

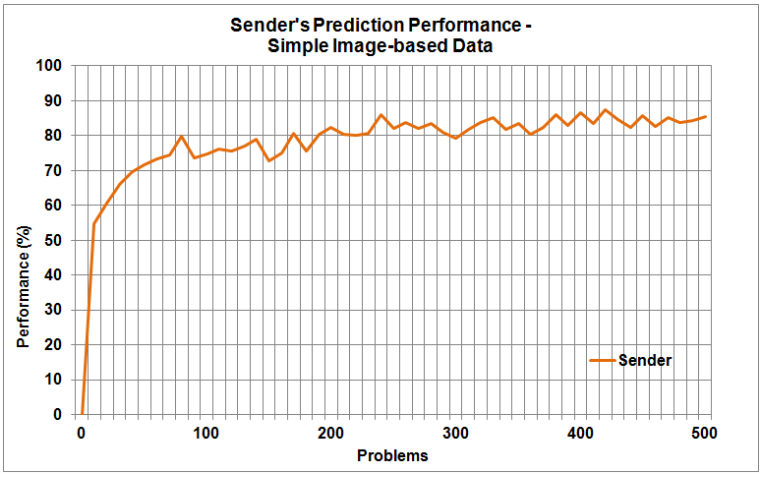


Figure 4.17: Average of S-XCS's performance in the *training mode* for predicting R-XCS's learning ability/problem's difficulty (either 'hard' or 'easy') from Exploit trials for 1,000 problems (500 Exploit, 500 Explore) over 30 runs.

Based on S-XCS's performance, S-XCS's rule was further studied. Table 4.6 provides samples of S-XCS's rules on the *simple image-based data* (i.e. three features in the problem can be adjusted). The second column gives the value of the first feature in the problem (i.e. `PatternDimension` that can take any value of '3 by 3, 4 by 4 and 5 by 5 dimensional pattern mapping'). The third column gives the value of the second feature in the problem (i.e `PatternOrientation` that can take any value of orientation 'Horizontal, Vertical, Diagonal1 and Diagonal2'). The fourth column gives the value of the third feature in the problem (i.e. `PatternOperator` that can take a value of logical operator 'OR' or 'XOR'). The fifth column indicates either the problem is 'easy' or 'hard'. The last two columns gives S-XCS's prediction value and the fitness of S-XCS's rule.

S-XCS predicts that the problems with the features of `PatternOperator` where the logical operator is 'OR' are considered to be a 'hard' problem. The problems received the highest prediction value (i.e. rule *7* and *5*,

Table 4.6: Example of S-XCS's rules (simple image-based data).

Rule	Condition			Action	Prediction	Fitness
	F1	F2	F3			
1	##	##00	#	1	1000	0.69
2	##	###1	1	0	1000	0.64
3	##	##00	#	0	0	0.61
4	##	####	1	1	361.25	0.5
5	0#	1#0#	0	1	997.57	0.48
6	##	0##0	1	0	999.61	0.45
7	11	0#10	0	1	1000	0.44
8	0#	0#11	#	0	840	0.41
9	##	##11	1	1	0	0.36
10	##	#0##	1	0	1000	0.33

where the conditions are '110#100' and '0#1#0#0', with the action '1'). The problems with the logical operator 'XOR' are considered to be an 'easy' problem (i.e. rule *2*, *6* and *10*, where the conditions are '1#####11', '##0##01 and '###0##1', with the action '0'). However, rules *4* and *9* incorrectly predict the logical operator 'XOR' are considered to be 'hard' problem when both the prediction and fitness values are low.

It was expected that R-XCS would achieve a good classification performance for a problem that applies the logical operator 'OR' to the problem, due to the fact that *any number* of the specified pattern can occur in the generated pattern and is considered an 'easy' problem. In contrast, R-XCS's classification performance actually achieved more than 95% for the problems that contain the logical operator 'XOR', when *one and only one* of the specified patterns can occur in the generated pattern which is considered to be a 'hard' problem.

The **second set of experiments** was performed to evaluate R-XCS's and S-XCS's performance on the *complex image-based data* (i.e. more than three features can be adjusted such as [`PatternDimension`, `PatternStroke`, `PatternOrientation`, `PatternAngle`, `PatternOperator`]). Figure 4.18 shows the average of R-XCS's classification performance for the complex image-based data. However, for almost all the problems, R-XCS cannot learn the generated patterns, which are too difficult to be classified correctly within 10,000 iterations. Based on R-XCS's classification performance, S-XCS was able to predict the difficulty levels of the problem (see Figure 4.19).

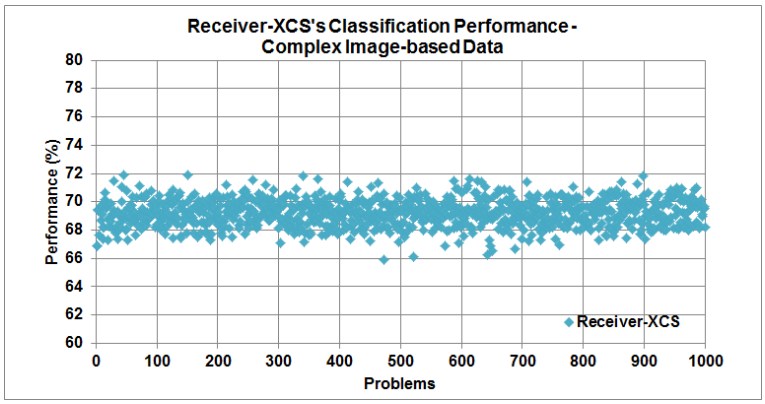


Figure 4.18: R-XCS's classification performance in the *training mode* to learn different types of binary problems for 1,000 problems.

Next, S-XCS's rule was further investigated. Table 4.7 gives samples of S-XCS's rules for predicting the problem difficulty for the *complex image-based data*, where more than three features can be adjusted. The second column gives the value of the first feature in the problem (i.e. `PatternDimension` that can take any value of '3 by 3, 4 by 4 and 5 by 5 dimensional pattern'). The third column gives the value of the second feature in the problem (i.e. `PatternStroke` that can take any value of '0, 1, 2 and 3'). The fourth column gives the value of the third feature in the problem (i.e. `PatternOrientation` that can take any value of orientation 'Horizon-

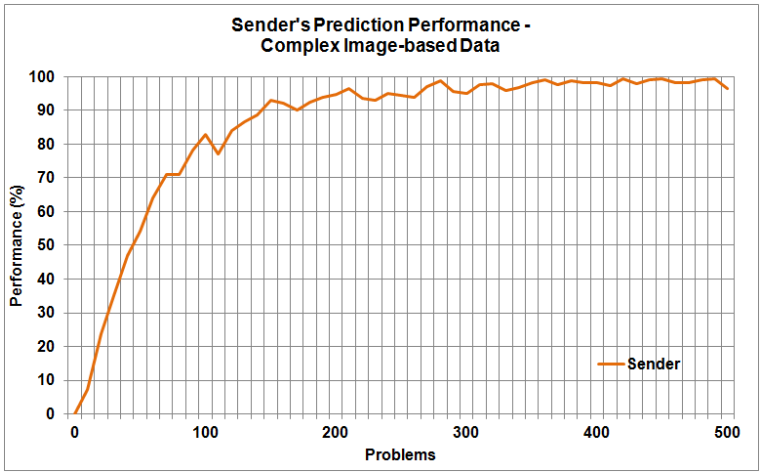


Figure 4.19: Average of S-XCS's performance in the *training mode* for predicting R-XCS's learning ability/problem's difficulty (either 'hard' or 'easy') from Exploit trials for 1,000 problems (500 Exploit, 500 Explore) over 30 runs.

tal, Vertical, Diagonal1 and Diagonal2'). The fifth column gives the value of the fourth feature in the problem (i.e. `PatternAngle` that can take any value of '0, 90, 180 and 360 degree'. The sixth column gives the value of the fifth feature in the problem (i.e. `PatternOperator` that can take a value of logical operator 'OR' or 'XOR'). The seventh column indicates that the problem is either 'easy' or 'hard'. The last two columns give S-XCS's prediction value and the fitness of S-XCS's rule.

Note all of the problems were 'hard' except for rules *8* and *9*, so it was trivial for S-XCS to learn. After a certain number of iterations, S-XCS was able to formulate appropriate knowledge in its population of rules. However, the criteria S-XCS created for the *Problem-PatternDifficulty* mapping was only partially accurate due to the generalisation caused by the class imbalance and the sparse coding (i.e. generating patterns for certain problem is not easy). For example, rules *8* and *9* incorrectly predicted that the problems were considered 'hard' when both the prediction and fitness

Table 4.7: Examples of S-XCS's rules (complex image-based data).

Rule	Condition					Action	Prediction	Fitness
	F1	**F2**	**F3**	**F4**	**F5**			
1	##	#0	11#1	#0	1	1	1000	0.15
2	10	0#	11##	##	0	1	1000	0.11
3	#0	#0	####	#0	1	1	1000	0.21
4	##	##	10##	0#	#	1	1000	0.21
5	0#	#1	#1##	0#	1	1	1000	0.02
6	##	#0	#1#1	0#	0	1	1000	0.02
7	##	##	1###	#0	#	1	1000	0.81
8	10	##	110#	10	0	0	0	0.21
9	1#	#1	1#1#	10	0	0	0	0.11

values are low.

Based on S-XCS's performance on both of the problem domains (i.e. simple and complex image-based data), a **third set of experiments** was conducted in order to investigate S's performance by applying two different styles of LCSs in S. In this experiment, S was applied to the *simple image-based data*. First, the *Michigan-style LCSs*, XCS, was applied to S to evolve S's rules, so that S could predict the difficulty levels of the problem effectively. Figure 4.20 shows the average of S-XCS's performance for predicting the problem's difficulty for 1,000 problems, while Figure 4.21 shows the average of S-XCS's performance for 50,000 problems. The results indicate that S-XCS takes more iterations to predict the problem's difficulty to be considered either 'hard' or 'easy', and S-XCS's prediction performance increases on average from 80% to 90% (see Figure 4.21). The results also suggest that S-XCS is required to evaluate the whole problem domain in order to build appropriate knowledge of the problem.

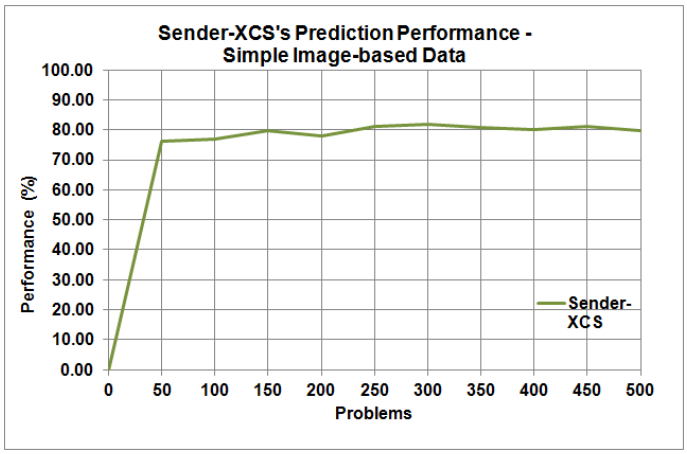


Figure 4.20: Average of S-XCS's performance in the *training mode* for predicting R-XCS's learning ability/problem's difficulty (either 'hard' or 'easy') from Exploit trials for 1,000 problems (500 Exploit, 500 Explore) over 30 runs.

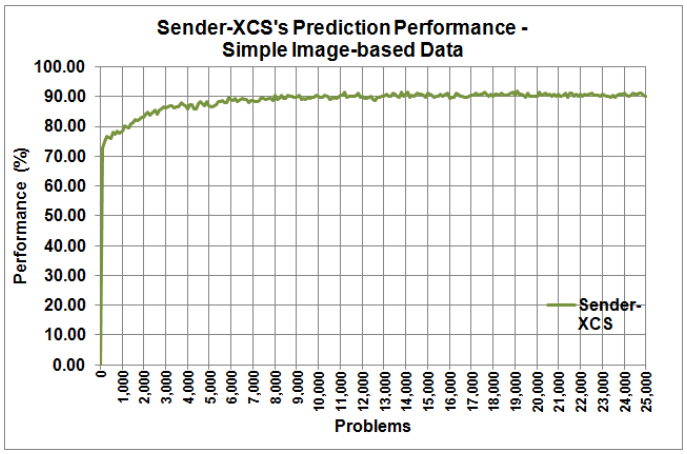


Figure 4.21: Average of S-XCS's performance in the *training mode* for predicting R-XCS's learning ability/problem's difficulty (either 'hard' or 'easy') from Exploit trials for 50,000 problems (25,000 Exploit, 25,000 Explore) over 30 runs.

Table 4.8 describes samples of S-XCS's rules on the *simple image-based data*, where three features can be adjusted in the problem (i.e. [`PatternDimension`, `PatternOrientation`, `PatternOperator`]), when S-XCS's is executed for 50,000 iterations. The second column gives the value of the first feature in the problem (i.e. `PatternDimension` that can take any value of '3 by 3, 4 by 4 and 5 by 5 dimensional pattern mapping'). The third column gives the value of the second feature in the problem (i.e. `PatternOrientation` that can take any value of orientation 'Horizontal, Vertical, Diagonal1 and Diagonal2'). The fourth column gives the value of the third feature in the problem (i.e. `PatternOperator` that can take a value of logical operator 'OR' or 'XOR'). The fifth column indicates either the problem is 'easy' or 'hard' (0 or 1 respectively). The sixth column gives S-XCS's prediction value.

Table 4.8: Examples of S-XCS's rules (simple image-based data).

Rule	**Condition**			**Action**	**Prediction**
	F1	**F2**	**F3**		
1	##	#101	#	0	1,000
2	##	0#11	#	0	1,000
3	##	1#10	#	0	1,000
4	##	##11	#	0	1,000
5	##	101#	#	0	1,000
6	10	0#00	#	1	1,000
7	10	#00#	1	1	1,000
8	10	00#0	#	1	1,000
9	10	000#	#	1	1,000
10	10	00#0	#	1	1,000

S-XCS predicted that the problem containing the feature of `PatternOrientation` and `PatternDimension` (i.e. 4 by 4 dimensional patterns) was considered a 'hard' problem as the problem received the highest prediction value (i.e. rules *6* to *10*). In contrast, rules *1* to *5* suggest that the problem's feature of `PatternDimension` does not affect the problem's difficulty as 'any dimensionality' is considered 'easy' including the problem's feature of `PatternOrientation`. After a certain number of iterations (i.e. S-XCS was executed for 50,000 iterations to learn 50,000 patterns), S-XCS was able to formulate appropriate knowledge in its population of rules. Again, the criteria S-XCS created for the *Problem-PatternDifficulty* mapping was only partially accurate due to the generalisation caused by the class imbalance and the sparse coding (i.e. generating patterns for certain problem is not easy and the generated image-based data were not properly separable), even though S-XCS was trained longer.

Second, the *Pittsburgh-style LCS*, A-PLUS, is applied to S to evolve S's rules, such that S can predict the difficulty levels of the problem effectively. Figure 4.22 shows the average of S-APLUS's prediction performance for predicting the problem's difficulty for 1,000 problems on the *simple image-based data*. S-APLUS was able to predict the problem's difficulty based on R-XCS's performance. However, S-APLUS was unable to find a perfect rule-set to describe the problem's features that affected the problem's difficulty. Instead, the proposed classifiers in the rule-set contained a sparse coding, meaning that not all binary values were legal, making them only partially accurate. There was only a small change in S-APLUS's performance after the first three to four hundred iterations in learning the problem. The results suggest that premature convergence may have occurred. It is encouraging that the performance was around 96% on average. Some improvement might be achieved through experimentation by adjusting a few parameters (i.e. increasing the size of classifiers in the rule-set).

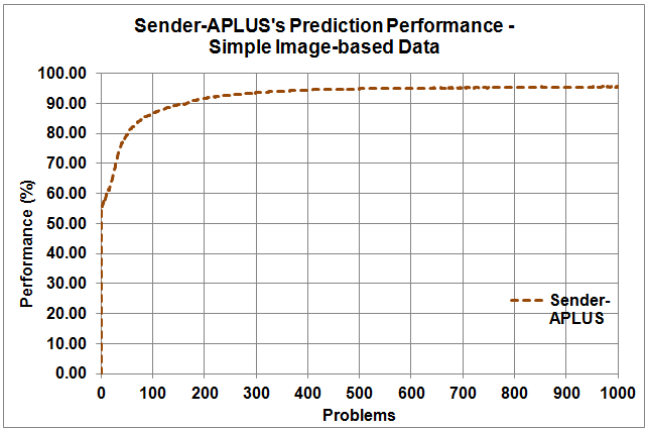


Figure 4.22: Average of S-APLUS's performance in the *training mode* for predicting R-XCS's learning ability/problem's difficulty (either 'hard' or 'easy') from Exploit trials for 2,000 problems (1,000 Exploit, 1,000 Explore) over 30 runs.

Table 4.9 describes samples of S-APLUS's individual rule-set on the *simple image-based data*, where three features can be adjusted in the problem (i.e. [`PatternDimension`, `PatternOrientation`, `PatternOperator`]). The second column gives the value of the first feature in the problem (i.e. `PatternDimension` that can take any value of '3 by 3, 4 by 4 and 5 by 5 dimensional pattern mapping'). The third column gives the value of the second feature in the problem (i.e. `PatternOrientation` that can take any value of orientation 'Horizontal, Vertical, Diagonal1 and Diagonal2'). The fourth column gives the value of the third feature in the problem (i.e. `PatternOperator` that can take a value of the logical operator 'OR' or 'XOR'). The fifth column indicates either the problem is 'easy' or 'hard' (0 or 1 respectively). The sixth column gives a number of the same rules in the S-APLUS population (macroclassifier).

Table 4.9: Example of S-APLUS's individual rule-set (simple image-based data).

Rule	Condition			Action	Numerosity
	F1	F2	F3		
1	##	#11#	#	0	18
2	00	###1	#	0	17
3	00	#00#	#	0	13
4	##	##11	#	0	11
5	##	1##1	#	0	8
6	##	0#00	1	1	23
7	##	00#0	1	1	19
8	##	1100	0	1	16
9	10	#000	#	1	10
10	10	0#0#	#	1	23

After 1,000 iterations, S-APLUS was able to formulate appropriate knowledge as described in Table 4.9. However, the criteria S-APLUS created for the *Problem-PatternDifficulty* mapping was only partially accurate due to the generalisation caused by the class imbalance and the sparse coding. The results showed that the generated image-based data were not properly separable; one pattern might be labelled to more than one class, which led to data ambiguity and class imbalance problems. Nevertheless, the quality of solutions was reasonably good and readable compared to S-XCS that applied Michigan-style LCSs. In fact, the features that affected the problem's difficulty had become apparent. The results suggested that the problems containing the feature of `PatternOrientation` and `PatternDimension` do affect the problem's difficulty. For example, if the problem's feature `PatternDimension` was '4 by 4 dimensional pattern mapping' (i.e. rule *9* and rule *10*) it was considered to be 'hard' problem. On the other hand, if the problem's feature `PatternDimension` was '3 by 3 dimensional pat-

terns mapping' (i.e. rule *2* and rule *3*) and had either 'Diagonal1' or 'Diagonal2' in `PatternOrientation`, it was considered an 'easy' problem.

4.2.1.2 Discussions and Findings

The overall aim of the section was to develop the *Two-Cornered Coevolution System*, where the generation agent (i.e the Sender) should be able to tune the problem's difficulty autonomously depending on the classification agent's ability to learn (i.e. to make the problem either 'hard' or 'easy'). Nevertheless, in this domain (i.e. image-based data), the underlying feature relationships which control the ease of learning within the problem domains were not easily separated. This makes the generation agent unable to change the difficulty levels effectively.

In the first set of experiments, S was able to generate various problems for classification (i.e. simple image-based data and complex image-based data). However, if R-XCS's learning is halted too early, then R-XCS was not able to learn to classify the patterns to the correct class accurately. Therefore, an autonomous method is required to determine when to halt R-XCS's learning to identify its true capability. The 95% performance rate that was used to separate 'hard' from 'easy' problems based on the previous results was not appropriate for the complex image-based data problem and reducing the knowledge about the problem features that S was able to learn, is necessary.

Next, two different styles of LCSs (i.e. Michigan-style LCSs and Pittsburgh-style LCSs) were applied to S in order to investigate the effectiveness of both approaches for evolving S rules in predicting the problem's difficulty. A-PLUS is the *Pittsburgh-style LCSs*, that incorporated a few methods of Michigan-style LCSs, XCS, such as *rule-subsumption* and *inaccurate rule-deletion* in its component. The adoption of *rule-subsumption* method in A-PLUS is important to reduce the size of the final rules, by removing large numbers of specific rules through generalisation. Meanwhile, the *inaccurate rule-deletion* helps the system focus its search for optimal

rules more quickly. For all of the experiments, A-PLUS was run in *on-line* mode, where only a single training instance is presented to the system. This evaluation approach was in contrast to the original Pittsburgh-style LCSs that had been designed only for *off-line* learning tasks, where a set of pre-classified data instances was generated in advance. This approach was introduced in order to suit the system design (i.e. generating each patterns *on-the-fly* or instantly). According to S's rules, it has been confirmed that S-APLUS was able to produce more compact rules with less running time (i.e. number of iterations). Furthermore, features in the problem that affected R-XCS's performance became apparent.

However, the findings showed that the generated image-based data were not properly separable; one pattern might be labelled to more than one class, which led to data ambiguity and class imbalance problems. There was no underlying relation between the resulting pattern and the features in the problem to distinguish the class clearly. Besides, S randomly generated image-based data without having any mechanism that could control a certain critical feature in the problem such as data sparsity, noise and class balance, which can adjust the difficulty of the problem. Therefore, if S is able to identify features in the problem that affect R's performance and predict the difficulty levels correctly, S can tune the problem's difficulty of the next problem more effectively.

4.2.1.3 Summary and Way Forward

Implementing a Pittsburgh-style LCS to the generation agent (i.e. the Sender) in the Two-Cornered Coevolution System improved the exploration of the rules structure, where the effect of the problem's features to the difficulty levels has become apparent. Furthermore, the proposed approach was able to reduce both the number of iterations and the running time of the generation agent for predicting the problem's difficulty, while achieving solutions with the same or even higher accuracy compared to the generation agent that implemented a Michigan-style LCS.

However, the generation agent was unable to identify the problem's difficulty effectively due to sparse coding and class imbalance. This resulted in the vast majority of randomly initialised rules being deleted, because of not being accurate and leaving very few rules for the GA to evolve. This problem can be solved by applying a new problem for classification (i.e. artificial data), where certain critical features in the problem (i.e. class balance, noise, decision boundary, number of instances, and many other parameters) can be controlled to generate various classification problems with different levels of difficulty. Therefore, in the next section, the work focuses on implementing an equivalent system for addressing the artificial data instead of image-based data. Further investigation will also be performed in order to study the differences between different approaches of LCSs in the generation agent (e.g. whether to use Michigan-style LCSs or Pittsburgh-style LCSs, i.e. to evolve complete rules or subsets of rules to map the problem's features to the difficulty levels).

4.2.2 Artificial Data for Classification

In this section, the results of the generation agent (i.e. the Sender (S)) and the classification agent (i.e. the Receiver (R)) in the *Two-Cornered Coevolution System* are presented for addressing the artificial data. Both the problem domain and the solution evolved autonomously (i.e. the generation agent created various problems and the associated datasets, while the classification agent learned each instance in the datasets).

The Tabu Search (TS) [77] technique was adapted on the two different systems (i.e. the plain Sender [7] and the Sender that applied Pittsburgh-style LCSs, A-PLUS) either to maximize or minimize the classification agent's performance (i.e. the Receiver). The first set of experiments was performed in order to evaluate the effectiveness of TS technique in vary-

[7] no learning system involved, i.e. the automated evolvable problem generator as implemented in the Section 3.1 for addressing the artificial data

ing features value `F` in the problem, either to make the problem 'harder' or 'easier' for R-XCSR to learn. The second set of experiments was performed to find the best system between the two systems (i.e. most effective or efficient system) that can automatically adjust the problem's difficulty based on R-XCSR's ability to learn.

First, TS is applied to S to search for the best combination of features `F` in the problem that can vary the difficulty levels based on R-XCSR's performance. Next, TS is used either to maximize or minimize R-XCSR's performance by adjusting the problem's difficulty. Third, both of the systems (i.e. the plain Sender and the Sender that applied Pittsburgh-style LCSs, A-PLUS) were evaluated so that S could generate the next problem at the appropriate levels of difficulty.

4.2.2.1 Experimental Results

In **the first set of experiments**, *TS* was applied to S (i.e the plain Sender), either to maximize or minimize R-XCSR's performance. Figures 4.23 and 4.24 show the average of R-XCSR's classification performance when TS is applied to S (i.e. S-TS) to search for the best combination of features `F` (i.e. `[Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd]`) for a two-class classification problem, with the aim to *maximize* R-XCSR's performance. S-TS was initialised with a predefined problem, where `[Fc=1 Fan=50 Fcn=50 Fcbl=50 Fcbd=25]` and was likely to be a 'hard' problem. TS was used to vary the features `F` in the problem except for `Fn`.

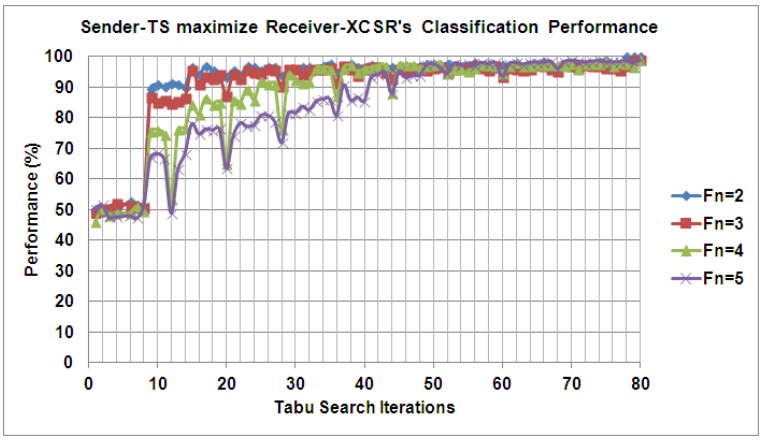


Figure 4.23: Average of R-XCSR's classification performance in the *training mode* to learn a two-class classification problem in the four problem domains. TS is used in S for adjusting the difficulty levels (i.e. from 'hard' to 'easy').

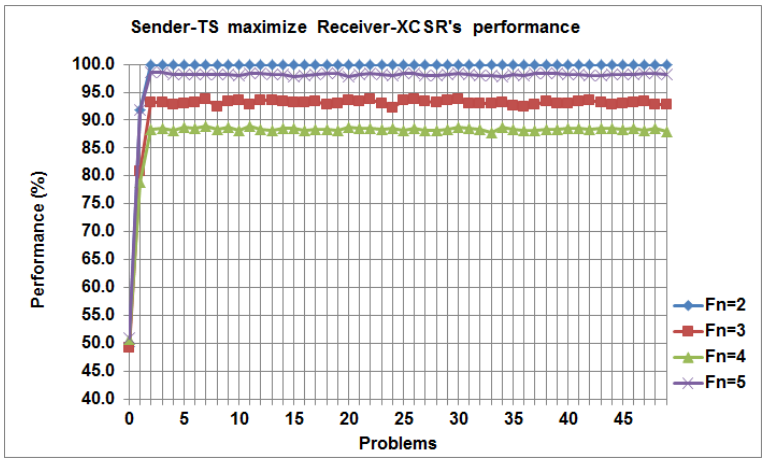


Figure 4.24: Average of R-XCSR's classification performance in the *training mode* to learn a two-class classification problem in the four problem domains over 30 runs for 50 problems. TS is used in S for adjusting the difficulty levels (i.e. from 'hard' to 'easy').

Table 4.10 describes the changes of features F in the problem (i.e. from initial problem to the next problem), when TS is applied to S (i.e. S-TS) to *maximize* R-XCSR's performance, where TS iterations is set to 100. S-TS starts with the initial problem (the second column), where R-XCSR's classification performance is shown in the third column. Next, S-TS tunes the initial problem to the next problem (the fourth column) to be 'easier' which *increases* R-XCSR's performance (the fifth column). The last column shows R-XCSR's execution time to address an individual problem for a single run. Here, S-TS uses TS to discover the best combinations of features F in the problem that can maximize R-XCSR's performance (i.e. selects the best combinations of features F from 100 iterations with highest classification performance), where R-XCSR's performance is from the training mode and R-XCSR's parameters setting are fixed for each problem.

Table 4.10: Changes of features F when S-TS *maximize* R-XCSR's performance.

Problem Domain	**Initial Problem**	**R (%)**	**Next problem**	**R (%)**	**Time Taken**
Fn=2	[2, 1, 0, 0, 0, 50, 50, 50, 25]	54	[2, 1, 0, 0, 0, 1, 0, 3, 2]	100	23 sec
Fn=3	[3, 1, 0, 0, 1, 50, 50, 50, 25]	47	[3, 1, 0, 1, 0, 0, 3, 4, 0]	100	43 sec
Fn=4	[4, 1, 0, 0, 1, 50, 50, 50, 25]	54	[4, 1, 0, 0, 1, 0, 3, 3, 3]	99	1 min 36 sec
Fn=5	[5, 1, 0, 0, 1, 50, 50, 50, 25]	49	[5, 1, 0, 0, 0, 0, 2, 4, 4]	97	2 min 45 sec

S-TS was able to adjust the difficulty levels by varying the features `F` in the problem to maximize R-XCSR's performance, where S-TS can make the problem 'easier' for R-XCSR to learn. For instance, when `Fn=2`, TS changes the initial problem `[2 1 0 0 0 50 50 50 25]` to the next problem `[2 1 0 0 0 1 0 3 2]`, which increases R-XCSR's performance from 54% to 100% (see Table 4.10). The results suggest that applying TS to S is suitable for helping S-TS to discover the best combination of features `F` in the problem that alter R-XCSR's performance (i.e. S-TS can autonomously determine the effect of the individual problem feature towards R-XCSR's performance).

Figure 4.25 shows the average of R-XCSR's classification performance when TS is applied to S (i.e. S-TS) to search for the best combination of features `F` (i.e. `[Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd]`) for a two-class classification problem, with the aim to *minimize* R-XCSR's performance. S-TS was initialised with a predefined problem, where `[Fc=1 Fan=5 Fcn=5 Fcbl=50 Fcbd=5]` and was likely to be an 'easy' problem. TS was used to vary the features `F` in the problem except for `Fn`.

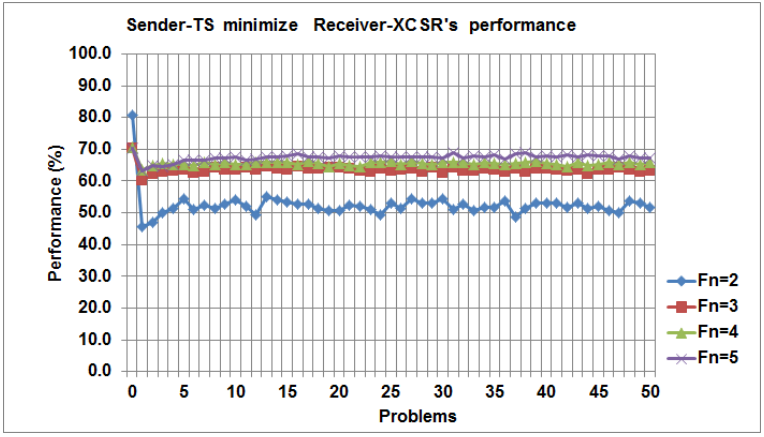


Figure 4.25: Average of R-XCSR's classification performance in the *training mode* to learn a two-class classification problem in the four problem domains over 30 runs for 50 problems. TS is used in S for adjusting the difficulty levels (i.e. from 'easy' to 'hard').

Table 4.11 describes the changes of features `F` in the problem (i.e. from the initial problem to the next problem) when TS is applied to S to *minimize* R-XCSR's performance, and TS iterations is set to 100. S-TS starts with the initial problem (the second column), and R-XCSR's classification performance is shown in the third column. Next, S-TS tunes the initial problem to the next problem (the fourth column) to be 'harder' which *decreases* R-XCSR's performance (the fifth column). The last column shows R-XCSR's execution time to address an individual problem for a single run.

Table 4.11: Changes of features `F` when S-TS *minimize* R's performance.

Problem Domain	**Initial Problem**	**R (%)**	**Next problem**	**R (%)**	**Time Taken**
Fn=2	[2, 1, 0, 0, 0, 5, 5, 50, 5]	94	[2, 1, 0, 0, 1, 5, 5, 50, 5]	82	24 sec
Fn=3	[3, 1, 0, 0, 1, 5, 5, 50, 5]	86	[3, 1, 0, 1, 1, 5, 3, 50, 5]	84	45 sec
Fn=4	[4, 1, 0, 0, 1, 5, 5, 50, 5]	85	[4, 1, 0, 1, 1, 3, 3, 50, 4]	84	1 min 20 sec
Fn=5	[5, 1, 0, 0, 1, 5, 5, 50, 5]	88	[5, 1, 0, 1, 1, 5, 5, 50, 5]	79	2 min 28sec

Here, a single parameter tended to be the focus when applying TS to S, which resulted in the approach becoming stuck in local optimum. For example, when `Fn=2`, TS changes the initial problem `[2 1 0 0 0 5 5 50 5]` to the next problem `[2 1 0 0 1 5 5 50 5]` which decreases R-XCSR's performance from 94% to 82% (see Table 4.11). Subsequently there is no significant difference in R-XCSR's performance. Therefore, in the next set of experiments, Pittsburgh-style LCSs, A-PLUS, was implemented in S-TS to overcome this problem.

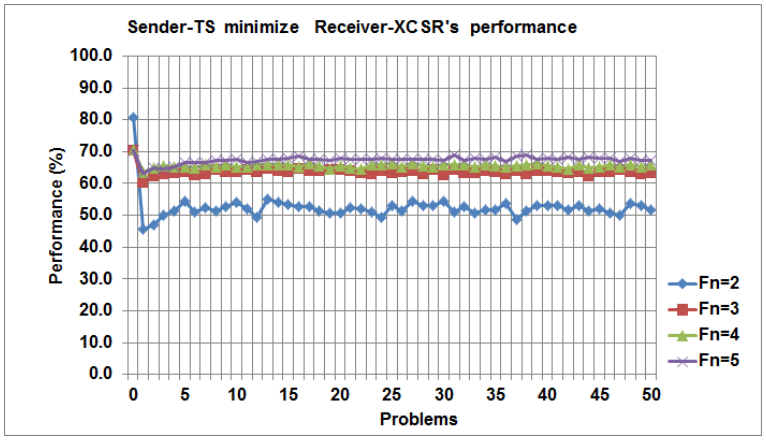
The **second set of experiments** was conducted in order to investigate S's ability when two different systems were applied to S (i.e. either the plain Sender with TS referred as S-TS or the Sender that applied Pittsburgh-style LCS, A-PLUS, with TS referred as S-APLUS-TS). Both of the methods were compared in order to find an effective system that autonomously adjusts the problem's difficulty based on R-XCSR's ability to learn (i.e. either to maximize or minimize R's performance).

Figure 4.26 show the average of R-XCSR's classification performance when TS is applied to S (i.e. either the plain Sender or the Sender-APLUS). S's objective is to *minimize* R-XCSR's performance where S is initialised with a predefined 'hard' problem (see Table 4.12 and Table 4.13).

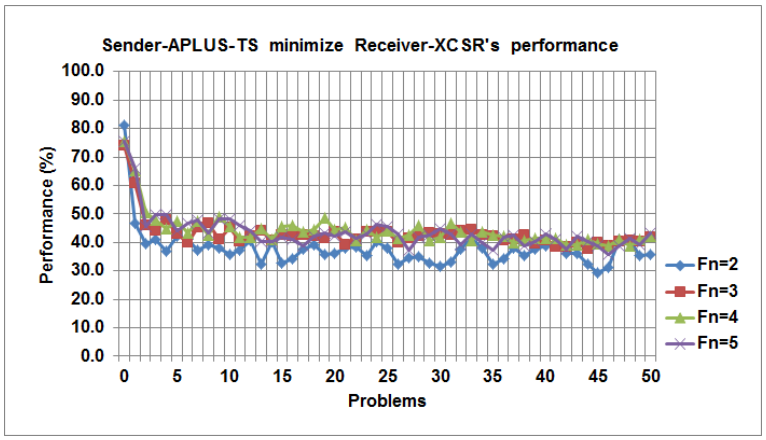
Table 4.12 describes how *S-TS* changes the initial problem (the second column) to the next problem (the fourth column) and *decreases* R-XCSR's performance (the fifth column) within a period of time (the sixth column).

Table 4.12: Changes of features `F` when S-TS *minimize* R-XCSR's performance.

Problem Domain	**Initial Problem**	**R (%)**	**Next problem**	**R (%)**	**Time Taken**
Fn=2	[2, 1, 0, 0, 0, 5, 5, 50, 5]	92	[2, 1, 0, 0, 1, 5, 5, 50, 5]	82	17 sec
Fn=3	[3, 1, 0, 0, 1, 5, 5, 50, 5]	86	[3, 1, 0, 1, 1, 5, 3, 50, 5]	84	24 sec
Fn=4	[4, 1, 0, 0, 1, 5, 5, 50, 5]	85	[4, 1, 0, 1, 1, 3, 3, 50, 4]	84	51 sec
Fn=5	[5, 1, 0, 0, 1, 5, 5, 50, 5]	88	[5, 1, 0, 1, 1, 5, 5, 50, 5]	79	1 min 40 sec



(a) S-TS minimize R-XCSR's performance.



(b) S-APLUS-TS minimize R-XCSR's performance.

Figure 4.26: Average of R-XCSR's classification performance in the *training mode* to learn a two-class classification problem in the four problem domains over 30 runs for 50 problems.

Table 4.13 describes how *S-APLUS-TS* changes the initial problem (the second column) to the next problem (the fourth column) and *decreases* R-XCSR's performance (the fifth column) within a period of time (the sixth column).

Table 4.13: Changes of features `F` when S-APLUS-TS *minimize* R-XCSR's performance.

Problem Domain	**Initial Problem**	**R (%)**	**Next problem**	**R (%)**	**Time Taken**
Fn=2	[2, 1, 0, 0, 0, 5, 5, 50, 5]	78	[2, 0, 1, 1, 1, 47, 23, 93, 19]	44	24 sec
Fn=3	[3, 1, 0, 0, 1, 5, 5, 50, 5]	70	[3, 0, 1, 1, 2, 38, 20, 50, 33]	43	1 min 2 sec
Fn=4	[4, 1, 0, 0, 1, 5, 5, 50, 5]	66	[4, 0, 1, 1, 2, 26, 28, 20, 4]	51	2 min 43 sec
Fn=5	[5, 1, 0, 0, 1, 5, 5, 50, 5]	75	[5, 0, 1, 1, 1, 13, 18, 32, 16]	73	4 min 50 sec

Note, R-XCSR's performance in Table 4.13 is different from Table 4.12 because the results are recorded from two independent experiments with different random seeds where each instance are created 'on-the-fly' (the data is created on-line).

Table 4.14 describes how *S-APLUS-TS* changes the initial problem (the second column) to the next problem (the fifth column) until up to 10 problems which *decreases* R-XCSR's performance (the third and sixth column) when `Fn=2`.

Table 4.14: Changes of features `F` when S-APLUS-TS *minimize* R-XCSR's performance (`Fn=2`).

Problem Num.	**Problem**	**R (%)**	**Problem Num.**	**Problem**	**R (%)**
0	[2, 1, 0, 0, 0, 5, 5, 50, 5]	78	1	[2, 0, 1, 1, 1, 47, 23, 93, 19]	44
2	[2, 0, 1, 1, 1, 33, 22, 79, 31]	38	3	[2, 0, 1, 1, 1, 26, 19, 70, 26]	47
4	[2, 0, 1, 1, 1, 38, 34, 79, 27]	49	5	[2, 0, 1, 1, 1, 26, 30, 52, 20]	58
6	[2, 0, 1, 1, 1, 47, 4, 44, 17]	51	7	[2, 0, 1, 1, 1, 36, 22, 88, 23]	50
8	[2, 0, 1, 1, 1, 23, 38, 61, 15]	58	9	[2, 0, 1, 1, 1, 47, 4, 43, 28]	54

Table 4.15 shows the results of the statistical test (i.e. student *t*-test) in order to verify that S-APLUS-TS effectively *minimizes* R-XCSR's performance compared to S-TS. The data (sample) for this test is based on the results from Figure 4.26. The p-value for each problem domain (i.e. `Fn=2` to `Fn=5`) is far less than 0.05. The results indicate that there is a statistically significant difference between the two systems. S-APLUS-TS performs better than S-TS, where applying the Pittsburgh-style LCS, A-PLUS with TS in S improved S's rules and effectively *minimize* R-XCSR's performance.

Table 4.15: Results of the statistical test (i.e. student *t*-test) between two systems (i.e. S-APLUS-TS and S-TS) to *minimize* the problem's difficulty.

Problem Domain	**p-value**
Fn=2	2.531E-20
Fn=3	1.754E-31
Fn=4	1.116E-29
Fn=5	6.919E-31

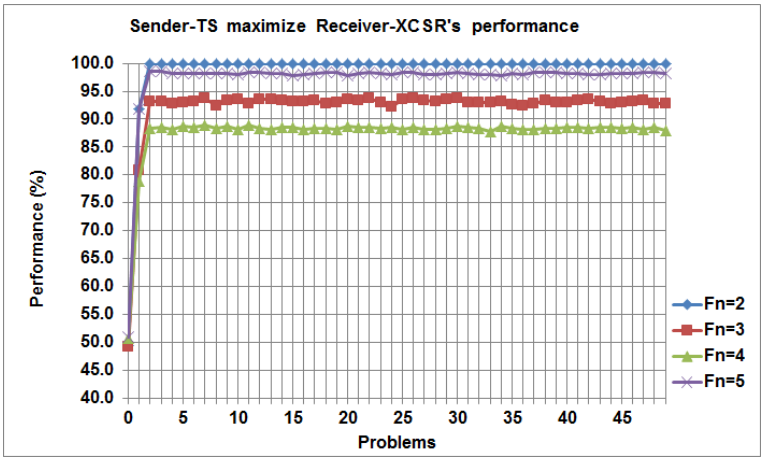
The results indicate that S-TS struggles to minimize R-XCSR's performance using TS alone in its methods (see Figure 4.26). However, the results suggest that using the Pittsburgh-style LCS, A-PLUS, with TS in S (i.e. S-APLUS-TS), not only facilitates S to discover the best combination of features `F` in the problem, but it also improves S's rule which effectively minimize R-XCSR's performance (see Figure 4.26 and Table 4.15). On the other hand, both of the systems can minimize R-XCSR's performance with a small difference in the total execution time for the two problem domains (i.e. `Fn=2` and `Fn=3`) except for `Fn=4` and `Fn=5`.

Figure 4.27 compares the average of R-XCSR's classification performance when S-TS and S-APLUS-TS are used to search for the best combination of features `F` in the four problem domains. S's objective is to *maximize* R-XCSR's performance and S is initialised with an 'easy' predefined problem (see Table 4.16 and Table 4.17).

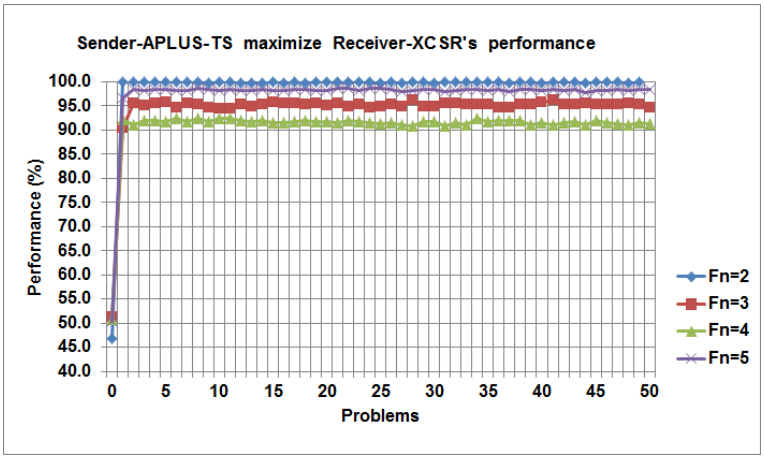
Table 4.16 describes how S-TS changes the initial problem (the second column) to the next problem (the fourth column) and increases R-XCSR's performance (the fifth column) within a period of time (the sixth column).

Table 4.16: Changes of features `F` when S-TS *maximize* R-XCSR's performance.

Problem Domain	**Initial Problem**	**R (%)**	**Next problem**	**R (%)**	**Time Taken**
Fn=2	[2, 1, 0, 0, 0, 50, 50, 50, 25]	54	[2, 1, 0, 0, 0, 1, 0, 3, 2]	100	15 sec
Fn=3	[3, 1, 0, 0, 1, 50, 50, 50, 25]	47	[3, 1, 0, 1, 0, 0, 3, 4, 0]	100	27 sec
Fn=4	[4, 1, 0, 0, 1, 50, 50, 50, 25]	54	[4, 1, 0, 0, 1, 0, 3, 3, 3]	100	55 sec
Fn=5	[5, 1, 0, 0, 1, 50, 50, 50, 25]	49	[5, 1, 0, 0, 0, 0, 2, 4, 4]	100	1 min 55 sec



(a) S-TS maximize R-XCSR's performance.



(b) S-APLUS-TS maximize R-XCSR's performance.

Figure 4.27: Average of R-XCSR's classification performance in the *training mode* to learn a two-class classification problem in the four problem domains over 30 runs for 50 problems.

Table 4.17 describes how S-APLUS-TS changes the initial problem (the second column) to the next problem (the fourth column) and increases R-XCSR's performance (the fifth column) within a period of time (the sixth column).

Table 4.17: Changes of features `F` when S-APLUS-TS *maximize* R-XCSR's performance.

Problem Domain	**Initial Problem**	**R (%)**	**Next problem**	**R (%)**	**Time Taken**
Fn=2	[2, 1, 0, 0, 0, 50, 50, 50, 25]	53	[2, 0, 1, 0, 1, 1, 3, 3, 2]	99	31 sec
Fn=3	[3, 1, 0, 0, 1, 50, 50, 50, 25]	44	[3, 0, 1, 1, 2, 0, 0, 0, 1]	100	44 sec
Fn=4	[4, 1, 0, 0, 1, 50, 50, 50, 25]	56	[4, 0, 1, 0, 3, 0, 0, 2, 0]	100	1 min 40 sec
Fn=5	[5, 1, 0, 0, 1, 50, 50, 50, 25]	57	[5, 0, 1, 0, 4, 4, 0, 0, 2]	100	4 min 20 sec

However, in this case the results suggest that using the Pittsburgh-style LCS, A-PLUS, with TS helps S to discover the best combination of features `F` in the problem which maximizes R-XCSR's classification performance, but there is no significant change in S's rule from what has been discovered by TS alone in S. Here, both systems successfully maximize R-XCSR's performance. Further, both of the systems can maximize R-XCSR's classification performance with a small difference of the total execution time for the three problem domains (i.e. `Fn=2`, `Fn=3` and `Fn=4`) except for `Fn=5`.

4.2.2.2 Discussions and Findings

The overall aim of this section is to develop the *Two-Cornered Coevolution System*, where the generation agent (i.e the Sender) is to be able to tune the problem's difficulty autonomously depending on the classification agent's (i.e the Receiver) ability to learn (i.e to make the problem either 'harder'

or 'easier'). The goal has been achieved as S was able to autonomously adjust and tune the difficulty of the problem (i.e. artificial dataset) based on R's ability to learn in the sense of strictly maximizing or minimizing R's performance. This is why the Three-Cornered Coevolution System is preferred over the Two-Cornered Coevolution System. In the Three-Cornered Coevolution System both the rate of convergence to a classification performance level and the maximum achievable performance level itself are not known a priori.

In the first set of experiments, TS was adopted in S to search for the best combination of features `F` in the problem that affected R's classification performance. Applying TS in S, helped S to discover the best combination of features `F` in the problem that altered R's classification performance (i.e. S can autonomously determine the effect of individual problem features regarding R's classification performance). The results suggest that TS was suitable for implementation in S, where S was able to adjust and tune the problem's difficulty either to make the problem 'harder' or 'easier' for R to learn. The results also indicated that TS was able to adjust the problem's difficulty which could either 'maximize' or 'minimize' R's performance. The results also showed that R was able to address the problem for classification as expected.

Here, a single parameter tended to be the focus when applying TS to S (see Table 4.11), resulting in the approach becoming stuck in local optima. In order to overcome this problem, the second set of experiments was performed to evaluate the two different systems (i.e. the plain Sender and the Sender that applied Pittsburgh-style LCSs, A-PLUS) that adopted TS so that S could generate the next problem at the appropriate level of difficulty. The results showed that applying the Pittsburgh-style LCSs, APLUS, in S had improved S's rule, and TS was able to adjust the difficulty levels more effectively by varying the features `F` in the problem specially to minimize R's classification performance, where S could make the problems 'harder' for R to learn.

4.2.2.3 Summary and Way Forward

Generating artificial data through specifying the problem's features such as `[Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd]` rather than image-based data has led to a system that can tune the datasets to adjust the performance of LCSs in a desired manner. An enumerative analysis of the potential datasets identified the performance gradients, but the 'on-line' learner (i.e. the Sender) identified useful gradients more efficiently. Important features, which control the ease of learning within the problem domains were identified using TS (e.g. extreme levels of noise decreased the classification agents's performance). The classification agent (i.e. the Receiver) was able to learn the generated datasets successfully, while the generation agent (i.e. the Sender) effectively tuned the problem's difficulty based on the classification agent's performance. Applying the Pittsburgh-style LCSs, A-PLUS, facilitated S to evolve and optimize the rules more effectively in order to generate the next problem at the appropriate levels of difficulty. The adaption of A-PLUS to an 'on-line' system was successful.

However, the Two-Cornered Coevolution System was implemented as a *coadaptive evolution* rather than a *real coevolution* such as proposed in Wilson's Three-Cornered Coevolution Framework. Thus, in the next phase the main focus is to ensure that all the agents cooperate with each other through a coevolutionary process within the system, so that the agents will trigger evolution when necessary. In Phase 3, the Three-Cornered Coevolution System consists of three different agents that communicate and work cooperatively to adapt with the changes of the problem. The implementation of this framework will be explored in the next section. In Phase 3, the generation agent (i.e. the Sender) needs to tune and adjust the problem's difficulty based on the two different classification agents' ability to learn (i.e. the Receiver (R) and the Interceptor (I) which will use different techniques of learning) [124]. I is required to direct S to change the problem's difficulty when R becomes stagnated through the coevolutionary process.

4.3 Phase 3: Three-Cornered Coevolution System

The overall aim of Phase 3 is to develop the *Three-Cornered Coevolution System* where three different agents evolve to adapt to and drive the changes of the problem. In Phase 3, a new classification agent, termed the Interceptor is introduced to trigger the coevolutionary process within the system (see Section 3.3). Here, both of the classification agents (i.e. the Receiver (R) and the Interceptor (I)) evolve and compete to learn various classification problems (i.e. artificial data) using different types of learning techniques (i.e. supervised learning or reinforcement learning), while the generation agent (i.e. the Sender (S)) evolves to tune and adjust the problem's difficulty based on the classification agents' ability to learn. A set of experiments was performed to investigate the generation agent's ability and the classification agents' performance in this problem domain[8].

All the agents work in a *coevolutionary manner* (i.e. coadaptive evolution) so that all the agents evolve and adapt to the changes of the problem. R and I were only applied to the artificial data, while S tuned and adjusted the problem's difficulty based on R's and I's performance. By introducing a similar third agent (i.e. the Interceptor), the difference in classification performance can be used to trigger the change in problem difficulty when necessary. The third agent can also assist in determining whether the problem should be made 'harder' or 'easier' when the agents' performance have stagnated.

[8]Note: *problem domain* refers to various datasets with different number of data features in each dataset depending on value *Fn* (i.e. `Fn=2` to `Fn=5`), *instance* refers to an instance in each dataset, *problem* refers to a problem from any problem domain that contains a problem-specific parameters (i.e. `[Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd]`.

4.3.1 Interceptor's Performance

In this section, the results of the new classification agent (i.e. I's classification performance) in the Three-Cornered Coevolution System are presented. A set of experiments was performed in order to investigate the ability of: 1) I to learn various classification problems (i.e. artificial data)[9], 2) S for maximizing and minimizing I's classification performance[10]. I was applied to the four problem domains (i.e. `Fn=2` to `Fn=5`), while the difficulty levels were tuned by S either to maximize or minimize I's classification performance.

The **first set of experiments** was performed in order to investigate the ability of: 1) I to learn various classification problems (i.e. artificial data), 2) S to *maximize* I's classification performance by tuning and adjusting the features `F` in the problem.

Figure 4.28 shows the average of I's classification performance, where S-TS is able to search for the best combination of features `F` in the problem, in the four problem domains and *maximizes* I's classification performance. S is initialised with a predefined problem (selected to be a 'hard' problem) (see Table 4.18).

Figure 4.29 shows the average of I-UCS's classification performance, where S-APLUS-TS successfully searches for the best combination of features `F` in the problem to address the same problem (see Table 4.19), and *maximizes* I-UCS's classification performance.

[9] I is initially set to be a supervised learning system, i.e. UCS

[10] either 'S-TS' where Tabu Search is applied to S and no learning systems involved or 'S-APLUS-TS' where Tabu Search is applied to S and S used Pittsburgh-style LCSs, A-PLUS

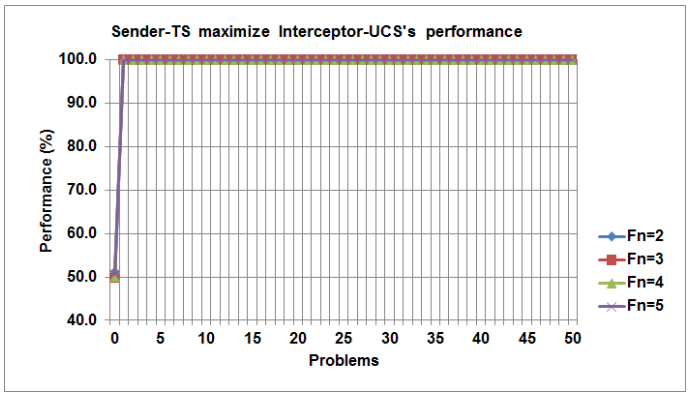


Figure 4.28: Average of I-UCS's classification performance in the *training mode* to learn a two-class classification problem in the four problem domains over 30 runs for 50 problems. TS is used in S for adjusting the difficulty levels (i.e. from 'hard' to 'easy').

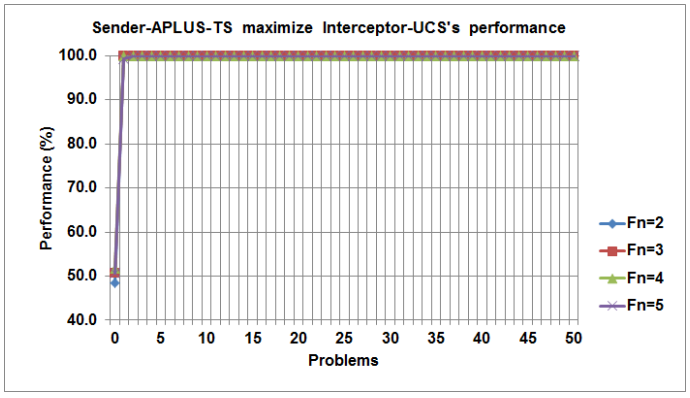


Figure 4.29: Average of I-UCS's classification performance in the *training mode* to learn a two-class classification problem in the four problem domains over 30 runs for 50 problems. S-APLUS together with TS is used for adjusting the difficulty levels (i.e. from 'hard' to 'easy').

Table 4.18 describes how S-TS changes the initial problem (the second column) to the next problem (the fourth column) and *increases* I-UCS's performance (the fifth column) within a period of time (the sixth column). The results show that S-TS is able to tune the features in the problem and adjust the problem's difficulty in order to maximize I-UCS's performance.

Table 4.18: Changes of features `F` when S-TS *maximize* I-UCS's performance.

Problem Domain	**Initial Problem**	**I (%)**	**Next problem**	**I (%)**	**Time Taken**
Fn=2	[2, 1, 0, 0, 0, 50, 50, 50, 25]	54	[2, 0, 0, 0, 1, 3, 0, 4, 3]	100	15sec
Fn=3	[3, 1, 0, 0, 1, 50, 50, 50, 25]	54	[3, 0, 0, 1, 1, 1, 4, 2, 4]	100	27sec
Fn=4	[4, 1, 0, 0, 1, 50, 50, 50, 25]	51	[4, 0, 0, 0, 1, 0, 2, 3, 3]	100	55sec
Fn=5	[5, 1, 0, 0, 1, 50, 50, 50, 25]	39	[5, 0, 0, 1, 0, 1, 0, 1, 0]	100	1min 55sec

Table 4.19 describes how S-APLUS-TS changes the initial problem (the second column) to the next problem (the fourth column) and *increases* I-UCS's performance (the fifth column) within a period of time (the sixth column). The results also show that S-APLUS-TS successfully tunes the features in the problem and adjusts the problem's difficulty to maximize I-UCS's performance.

Table 4.19: Changes of features F when S-APLUS-TS *maximize* I-UCS's performance.

Problem Domain	**Initial Problem**	**I (%)**	**Next problem**	**I (%)**	**Time Taken**
Fn=2	[2, 1, 0, 0, 0, 50, 50, 50, 25]	55	[2, 0, 1, 0, 1, 0, 4, 0, 1]	100	29sec
Fn=3	[3, 1, 0, 0, 1, 50, 50, 50, 25]	50	[3, 0, 1, 1, 1, 1, 4, 0, 4]	100	44sec
Fn=4	[4, 1, 0, 0, 1, 50, 50, 50, 25]	39	[4, 0, 2, 0, 0, 0, 4, 3, 2]	100	1min 39sec
Fn=5	[5, 1, 0, 0, 1, 50, 50, 50, 25]	51	[5, 0, 1, 0, 2, 1, 0, 0, 3]	100	2min 20sec

In this case, the results confirmed that applying TS in S (i.e. either S-TS or S-APLUS-TS) facilitated S to discover the best combination of features F in the problem to maximize I's performance. There were no significant changes in S-APLUS-TS's rules compared to S-TS's rules. Both systems successfully maximized I's performance, though S-APLUS-TS required a longer time compared to S-TS.

The **second set of experiments** was performed in order to investigate the ability of: 1) I to learn various classification problems (i.e. artificial data), 2) S to *minimize* I's classification performance by tuning and adjusting the features F in the problem.

Figure 4.30 shows the average of I-UCS's classification performance, where S-TS is unable to *minimize* I-UCS's classification performance effectively on the four problem domains. S is initialised with a predefined problem (an 'easy' problem) (see Table 4.20) to *minimize* I-UCS's classification performance. It is not an easy task for S-TS to search for the best combination of features F in this set-up as S-TS needs to search multiple features simultaneously and becomes trap in the local optima.

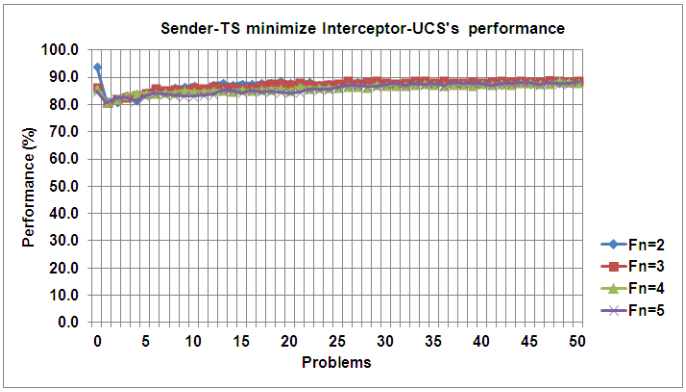


Figure 4.30: Average of I-UCS's classification performance in the *training mode* to learn a two-class classification problem in the four problem domains over 30 runs for 50 problems. TS is used in S for adjusting the difficulty levels (i.e. from 'easy' to 'hard').

Figure 4.31 shows the average of I-UCS's classification performance, where S-APLUS-TS successfully *minimizes* I-UCS's classification performance. S-APLUS-TS is able to search for the best combination of features `F` in the problem to address the same problem (see Table 4.21) effectively and *minimizes* I-UCS's classification performance.

Table 4.20 describes how S-TS changes the initial problem (the second column) to the next problem (the fourth column) and *decreases* I-UCS's performance (the fifth column) within a period of time (the sixth column). The results show that S-TS is unable to tune the features F in the problem and adjust the problem's difficulty effectively in order to minimize I-UCS's performance.

Table 4.21 describes how S-APLUS-TS changes the initial problem (the second column) to the next problem (the fourth column) and *decreases* I-UCS's performance (the fifth column) within a period of time (the sixth column). The results show that S-APLUS-TS successfully tunes the features in the problem and adjusts the problem's difficulty to *minimize* I-UCS's performance.

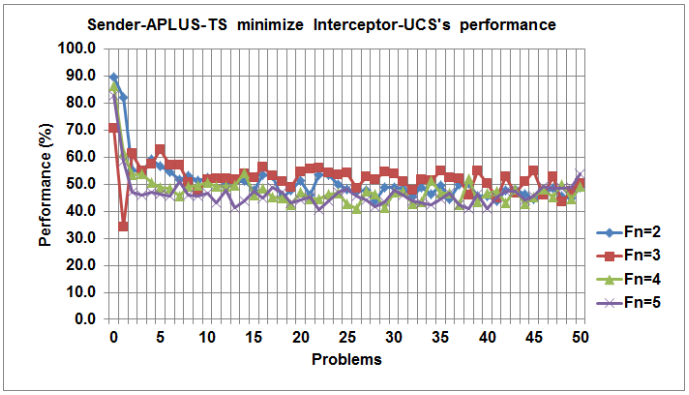


Figure 4.31: Average of I-UCS's classification performance in the *training mode* to learn a two-class classification problem in the four problem domains over 30 runs for 50 problems. A-PLUS with TS is used in S for adjusting the difficulty levels (i.e. from 'easy' to 'hard').

Table 4.20: Changes of features `F` when S-TS *minimize* I-UCS's performance.

Problem Domain	**Initial Problem**	**I (%)**	**Next problem**	**I (%)**	**Time Taken**
Fn=2	[2, 1, 0, 0, 0, 5, 5, 50, 5]	92	[2, 1, 0, 0, 1, 4, 2, 50, 5]	82	17 sec
Fn=3	[3, 1, 0, 0, 1, 5, 5, 50, 5]	89	[3, 1, 0, 1, 1, 3, 0, 50, 4]	57	24 sec
Fn=4	[4, 1, 0, 0, 1, 5, 5, 50, 5]	75	[4, 1, 0, 0, 1, 5, 5, 50, 5]	75	51 sec
Fn=5	[5, 1, 0, 0, 1, 5, 5, 50, 5]	88	[5, 1, 0, 1, 1, 5, 4, 50, 3]	81	1 min 40 sec

Table 4.21: Changes of features F when S-APLUS-TS *minimize* I-UCS's performance.

Problem Domain	**Initial Problem**	**I (%)**	**Next problem**	**I (%)**	**Time Taken**
Fn=2	[2, 1, 0, 0, 0, 5, 5, 50, 5]	93	[2, 0, 1, 1, 1, 19, 21, 34, 32]	58	29 sec
Fn=3	[3, 1, 0, 0, 1, 5, 5, 50, 5]	84	[3, 0, 1, 0, 0, 38, 0, 1, 1]	51	1 min 19 sec
Fn=4	[4, 1, 0, 0, 1, 5, 5, 50, 5]	66	[4, 0, 1, 1, 1, 4, 3, 52, 31]	47	1 min 52 sec
Fn=5	[5, 1, 0, 0, 1, 5, 5, 50, 5]	75	[5, 0, 1, 1, 0, 2, 1, 86, 0]	43	3 min 38 sec

The results showed that S is unable to minimize I-UCS's classification performance effectively, using TS alone in its methods. In this case, TS was focused on individual parameter tuning compared to A-PLUS that was able to perform holistic multiple simultaneous parameter tuning. However, the results suggest that applying TS to S was suitable for facilitating S to discover the optimal combination of features F in the problem that alters I-UCS's performance. Further, adapting the Pittsburgh-style LCSs, A-PLUS, with TS in S, not only facilitated S to search for the combinations of features in the problem, but it also improved S's rules which effectively minimized I-UCS's classification performance. However, the total execution time increased when the system used S-APLUS-TS, but I-UCS's classification performance decreased significantly. In this case, S-APLUS-TS was able to effectively minimize I-UCS's classification performance, whereas previously I-UCS was able to solve the simpler problems for classification as expected.

4.3.2 Classification Agents' Performance

In this section, the results of the classification agents (i.e. the frequency of peak performance) and the generation agent (i.e. changes of features `F` in the problem) in the Three-Cornered Coevolution System are presented. The *frequency of peak performance* is defined as the number of maximum classification performance of the classification agent (i.e. a repetitive point of the maximum classification performance on the graph, where the classification performance of R and I is greater than 60% and the difference in performance between R and I is less than the threshold value) for solving a number of problems.

A set of experiments was performed in order to investigate the ability of I[11] to trigger S[12] to change the problem's difficulty (i.e. either to make it 'harder' or 'easier'). S tunes the problem's difficulty to be 'harder', when the difference in performance between R and I is less than the threshold value. S tunes the problem's difficulty to be 'easier', when the difference in performance between R and I is greater than the threshold value. R and I were applied to the four problem domains (i.e. `Fn=2` to `Fn=5`), where S tuned the difficulty levels either to increase or decrease the problem's difficulty.

It is a standard practice to take the average of 30 independent runs in assessing the results of EC experiments in order to avoid outliers and assist in statistical analysis. However, this makes the assumption that each problem remains the same for 30 independent runs, which is not the case here, as S varies the problem to R and I stochastically. Although the patterns of performance may be similar between all the runs, the phase between performance improvements or decreases may (often) differ, which render taking the average at any given instance meaningless. Therefore, the re-

[11] either 'I-UCS' that used the supervised learning system (i.e. UCS) or 'I-XCSR' that used the reinforcement learning system (i.e. XCSR)

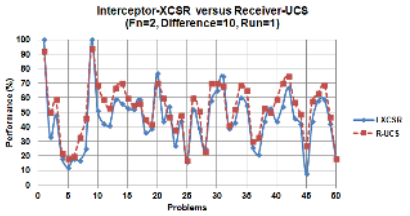
[12] 'S-APLUS-TS' where Tabu Search is applied to S and S used Pittsburgh-style LCSs, A-PLUS

sults only illustrate R's and I's classification performance from a single run out of 30 runs conducted to verify consistency in performance for learning a number of problems. The classification performance is recorded from the average of every 100 Exploit trials.

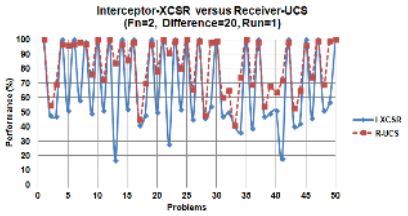
The **first set of experiments** was performed in order to investigate the ability of I-XCSR[13] to trigger S to change the problem's difficulty when the threshold values were set to: 1) 10%, and 2) 20%.

Figure 4.32 shows the frequency of peak performance of R-UCS and I-XCSR when I-XCSR *that used reinforcement learning* is set to trigger S to change the problem's difficulty on the four problem domains (i.e. `Fn=2` to `Fn=5`) for a single run. The frequency of peak performance is higher for all the problem domains except for `Fn=2` when the threshold value is set to 10% compared to 20%.

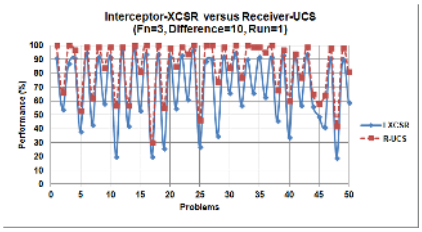
[13]Here, I-XCSR is the triggering agent when the difference in performance is calculated as I-XCSR's performance - R-UCS's performance



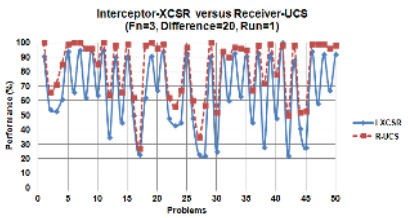
(a) threshold=10, Fn=2.



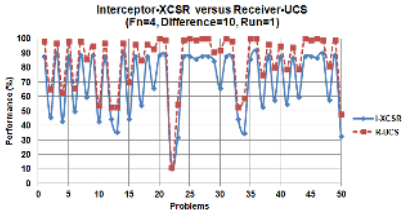
(b) threshold=20, Fn=2.



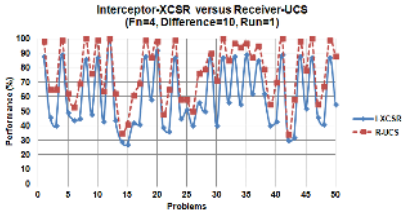
(a) threshold=10, Fn=3.



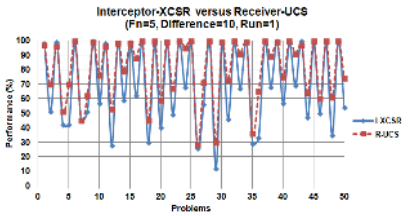
(b) threshold=20, Fn=3.



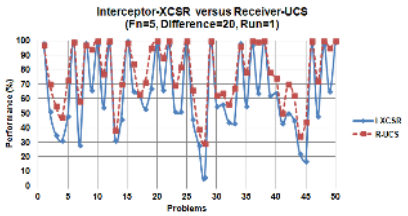
(a) threshold=10, Fn=4.



(b) threshold=20, Fn=4.



(a) threshold=10, Fn=5.



(b) threshold=20, Fn=5.

Figure 4.32: Frequency of peak performance of R-UCS and I-XCSR when I-XCSR triggers S to change the difficulty levels (threshold=10,20).

Table 4.22 provides the frequency of peak performance of R-UCS and I-XCSR (the second and third column) and the percentage of peak performance (i.e. the frequency of peak performance divided by the total problems) (the fourth and fifth column) when I-XCSR triggers S to change the problem's difficulty on the four problem domains (i.e. `Fn=2` to `Fn=5`). The threshold values are set to 10% and 20%.

Table 4.22: Frequency of peak performance of R-UCS and I-XCSR when I-XCSR triggers S to change the difficulty levels (threshold=10,20) for 50 problems from a single run.

Problem Domain	**peak performance (threshold=10)**	**peak performance (threshold=20)**	**% of peak performance (threshold=10)**	**% of peak performance (threshold=20)**
Fn=2	10	20	20	40
Fn=3	24	21	48	42
Fn=4	18	17	36	34
Fn=5	23	16	46	32

Table 4.23 describes how S changes the initial problem to the next problem, which either increases or decreases I-XCSR's and R-UCS's performance when the *threshold value is set to 10%*. For example, when `Fn=2`, S starts with the initial problem that needs to be learned by I-XCSR and R-UCS (the second column), where the classification performance is shown in the third and fourth column. S tunes the initial problem to the next problem (the fifth column) to be 'harder' when the difference in performance is less than *threshold=10*, which *decreases* I-XCSR's and R-UCS's performance (the sixth and seventh column). S tunes the second problem (the second column) to the third problem (the fifth column) to be 'easier' when the difference in performance is greater than *threshold=10*, which *increases* I-XCSR's and R-UCS's performance (the sixth and seventh column). Conversely, when `Fn=3` and `Fn=4`, S tunes the second problem (the second

column) to the third problem (the fifth column) to be 'harder' when the difference in performance is less than *threshold=10*, which *decreases* I-XCSR's and R-UCS's performance (the sixth and seventh column).

Table 4.23: Changes of features F when I-XCSR triggers S to change the problem's difficulty (threshold=10).

Problem Domain	**Initial Problem**	**I (%)**	**R (%)**	**Problem=2**	**I (%)**	**R (%)**
Fn=2	[2, 1, 0, 0, 0, 50, 50, 50, 25]	100	100	[2, 0, 1, 1, 1, 21, 14, 18, 4]	59	77
Fn=3	[3, 1, 0, 0, 1, 50, 50, 50, 25]	88	100	[3, 0, 1, 0, 0, 1, 7, 0, 31]	91	100
Fn=4	[4, 1, 0, 0, 1, 50, 50, 50, 25]	89	100	[4, 0, 1, 1, 0, 2, 3, 4, 3]	90	98
Fn=5	[5, 1, 0, 0, 1, 50, 50, 50, 25]	98	97	[5, 0, 1, 0, 0, 47, 3, 3, 0]	40	61
Problem Domain	**Problem=2**	**I (%)**	**R (%)**	**Problem=3**	**I (%)**	**R (%)**
Fn=2	[2, 0, 1, 1, 1, 21, 14, 18, 4]	59	77	[2, 2, 0, 1, 1, 1, 0, 0, 0, 3]	100	100
Fn=3	[3, 0, 1, 0, 0, 1, 7, 0, 31]	91	100	[3, 0, 1, 1, 1, 1, 12, 63, 27]	43	59
Fn=4	[4, 0, 1, 1, 0, 2, 3, 4, 3]	90	98	[4, 0, 1, 1, 1, 16, 2, 4, 3]	53	86
Fn=5	[5, 0, 1, 0, 0, 47, 3, 3, 0]	40	61	[5, 0, 1, 0, 0, 1, 1, 3, 4]	97	100

Table 4.24 describes how S changes the initial problem to the next problem, which either increases or decreases I-XCSR's and R-UCS's performance when the *threshold value is set to 20%*. The sequence is similar to that described earlier. S tunes the initial problem to the next problem (the fifth column) to be 'harder' when the difference in performance is less than *threshold=20*, which *decreases* I-XCSR's and R-UCS's performance (the sixth and seventh column). Next, S tunes the second problem (the second column) to the third problem (the fifth column) again to be 'harder' when the difference in performance is less than *threshold=20*, which *decreases* I-XCSR's and R-UCS's performance (the sixth and seventh column).

Table 4.24: Changes of features F when I-XCSR triggers S to change the problem's difficulty (threshold=20).

Problem Domain	**Initial Problem**	**I (%)**	**R (%)**	**Problem=2**	**I (%)**	**R (%)**
Fn=2	[2, 1, 0, 0, 0, 50, 50, 50, 25]	100	100	[2, 0, 1, 1, 1, 21, 14, 18, 4]	59	77
Fn=3	[3, 1, 0, 0, 1, 50, 50, 50, 25]	88	100	[3, 0, 1, 0, 1, 1, 3, 48, 31]	37	66
Fn=4	[4, 1, 0, 0, 1, 50, 50, 50, 25]	89	100	[4, 0, 1, 1, 1, 31, 4, 4, 3]	50	73
Fn=5	[5, 1, 0, 0, 1, 50, 50, 50, 25]	98	97	[5, 0, 1, 0, 0, 47, 3, 3, 0]	40	61
Problem Domain	**Problem=2**	**I (%)**	**R (%)**	**Problem=3**	**I (%)**	**R (%)**
Fn=2	[2, 0, 1, 1, 1, 21, 14, 18, 4]	59	77	[2, 0, 1, 1, 0, 38, 1, 4, 1]	44	60
Fn=3	[3, 0, 1, 0, 1, 1, 3, 48, 31]	37	66	[3, 0, 1, 1, 1, 2, 0, 0, 4]	91	96
Fn=4	[4, 0, 1, 1, 1, 31, 4, 4, 3]	50	73	[4, 0, 1, 1, 0, 0, 1, 2, 2]	89	100
Fn=5	[5, 0, 1, 0, 0, 47, 3, 3, 0]	40	61	[5, 0, 1, 0, 0, 1, 1, 3, 4]	97	100

In this set-up, I-XCSR was able to trigger S to change the problem's difficulty for a given threshold. S tuned the problem's difficulty to be 'harder', when the difference in performance between R and I was less than the threshold value, which decreased R's and I's performance. S adjusted the problem's difficulty to be 'easier', when the difference in performance between R and I was greater than the threshold value which increased R's and I's performance.

The **second set of experiments** was performed in order to investigate the ability of I-UCS[14] to trigger S to change the problem's difficulty when the threshold values were set to: 1) 10%, 2) 20%.

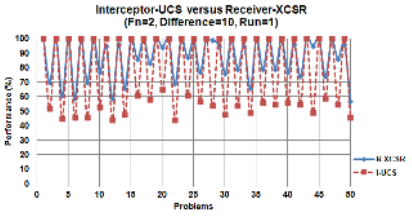
Figure 4.33 shows the frequency of peak performance of R-XCSR and I-UCS when I-UCS, *that used supervised learning*, is able to trigger S to change the problem's difficulty on the four problem domains (i.e. `Fn=2` to `Fn=5`) for a single run. The frequency of peak performance is higher for all the problem domains when the threshold value is 10%, instead of 20%.

Table 4.25 provides the frequency of peak performance and the percentage of peak performance when I-UCS is used to trigger S to change the problem's difficulty. The frequency of peak performance again is higher for all the problem domains when the threshold value is set to 10% rather than 20%.

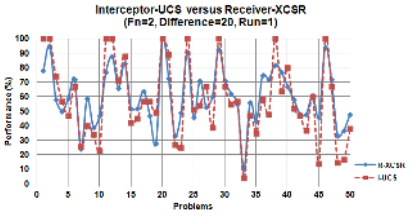
Table 4.25: Frequency of peak performance of R-XCSR and I-UCS when I-UCS triggers S to change the difficulty levels (threshold=10,20) for 50 problems from a single run.

Problem Domain	**peak performance (threshold=10)**	**peak performance (threshold=20)**	**% of peak performance (threshold=10)**	**% of peak performance (threshold=20)**
Fn=2	25	9	50	18
Fn=3	24	7	48	14
Fn=4	25	11	50	22
Fn=5	25	10	50	20

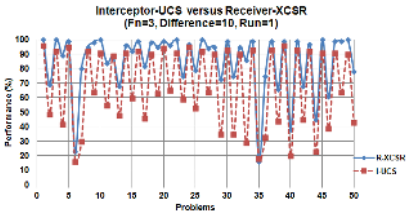
[14]Here, I-UCS is the triggering agent when the difference in performance is calculated as I-UCS's performance - R-XCSR's performance



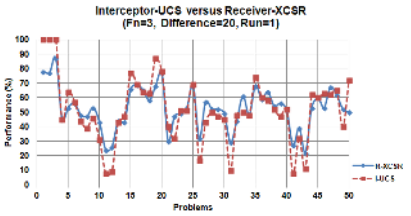
(a) threshold=10, Fn=2.



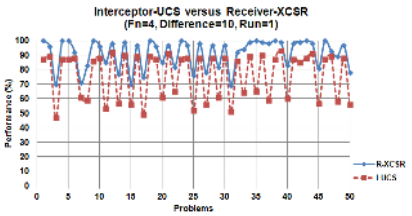
(b) threshold=20, Fn=2.



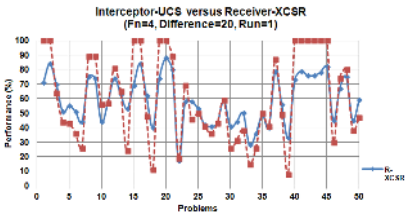
(a) threshold=10, Fn=3.



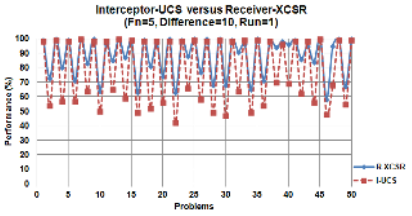
(b) threshold=20, Fn=3.



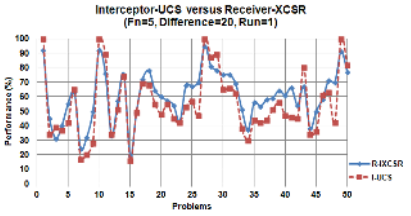
(a) threshold=10, Fn=4.



(b) threshold=20, Fn=4.



(a) threshold=10, Fn=5.



(b) threshold=20, Fn=5.

Figure 4.33: Frequency of peak performance of R-XCSR and I-UCS when I-UCS triggers S to change the difficulty levels (threshold=10,20).

Table 4.26 describes how S changes the initial problem to the next problem, which either increases or decreases I's and R's performance when the *threshold value is set to 10*. Table 4.27 describes how S changes the initial problem to the next problem, which either increases or decreases I's and R's performance when the *threshold value is set to 20*. In this case, I-UCS was able to trigger S to change the problem's difficulty for a given threshold. The process of S changing the problems either to be 'harder' or 'easier' was similar to that described earlier.

Table 4.26: Changes of features `F` when I-UCS triggers S to change the problem's difficulty (threshold=10).

Problem Domain	**Initial Problem**	**I (%)**	**R (%)**	**Problem=2**	**I (%)**	**R (%)**
Fn=2	[2, 1, 0, 0, 0, 50, 50, 50, 25]	100	58	[2, 0, 1, 0, 1, 1, 1, 0, 4]	100	90
Fn=3	[3, 1, 0, 0, 1, 50, 50, 50, 25]	100	80	[3, 0, 1, 0, 1, 1, 2, 1, 2]	100	68
Fn=4	[4, 1, 0, 0, 1, 50, 50, 50, 25]	100	81	[4, 0, 1, 0, 1, 3, 2, 0, 2]	100	83
Fn=5	[5, 1, 0, 0, 1, 50, 50, 50, 25]	100	73	[5, 0, 1, 0, 0, 1, 3, 2, 3]	100	89
Problem Domain	**Problem=2**	**I (%)**	**R (%)**	**Problem=3**	**I (%)**	**R (%)**
Fn=2	[2, 0, 1, 0, 1, 1, 1, 0, 4]	100	90	[2, 0, 1, 1, 1, 0, 8, 51, 36]	48	52
Fn=3	[3, 0, 1, 0, 1, 1, 2, 1, 2]	100	68	[3, 0, 1, 1, 1, 1, 12, 63, 27]	100	80
Fn=4	[4, 0, 1, 0, 1, 3, 2, 0, 2]	100	83	[4, 0, 3, 1, 1, 1, 1, 3, 2]	100	77
Fn=5	[5, 0, 1, 0, 0, 1, 3, 2, 3]	100	89	[5, 0, 1, 1, 0, 1, 4, 4, 2]	100	86

Table 4.27: Changes of features F when I-UCS triggers S to change the problem's difficulty (threshold=20).

Problem Domain	**Initial Problem**	**I (%)**	**R (%)**	**Problem=2**	**I (%)**	**R (%)**
Fn=2	[2, 1, 0, 0, 0, 50, 50, 50, 25]	100	58	[2, 0, 1, 0, 1, 1, 1, 0, 4]	100	90
Fn=3	[3, 1, 0, 0, 1, 50, 50, 50, 25]	100	80	[3, 0, 1, 0, 0, 26, 1, 2, 3]	55	61
Fn=4	[4, 1, 0, 0, 1, 50, 50, 50, 25]	100	81	[4, 0, 1, 0, 1, 31, 3, 3, 4]	57	61
Fn=5	[5, 1, 0, 0, 1, 50, 50, 50, 25]	100	73	[5, 0, 1, 0, 0, 1, 3, 2, 3]	100	89
Problem Domain	**Problem=2**	**I (%)**	**R (%)**	**Problem=3**	**I (%)**	**R (%)**
Fn=2	[2, 0, 1, 0, 1, 1, 1, 0, 4]	100	90	[2, 0, 1, 1, 1, 0, 8, 51, 36]	48	52
Fn=3	[3, 0, 1, 0, 0, 26, 1, 2, 3]	55	61	[3, 0, 1, 1, 1, 29, 3, 59, 10]	50	59
Fn=4	[4, 0, 1, 0, 1, 31, 3, 3, 4]	57	61	[4, 0, 1, 1, 0, 46, 1, 2, 4]	36	56
Fn=5	[5, 0, 1, 0, 0, 1, 3, 2, 3]	100	89	[5, 0, 1, 1, 1, 1, 4, 84, 4]	19	36

The results suggest that both classification agents (i.e. R and I) achieved different levels of classification performance, where in most problems the classification agent that used the supervised learning system (i.e. UCS) achieved a better performance as expected (see Table 4.26 and 4.27). These results are related to the way the learning system is designed. UCS is specifically designed for the classification tasks and benefits directly from the known label (provided class). In contrast, XCSR receives an immediate reward from the environment upon predicting an action (class) for each input, where XCSR works, based on a 'trial-an-error' basis. For all of the experiments, both systems were given 2,000 instances for classifying the instance either belonging to 'Class 1' or 'Class 0'. The results suggest that XCSR requires a larger number of instances than UCS in order to achieve its best performance.

Varying the threshold values (i.e. 10% and 20%) either increased or decreased the frequency of peak performance of R and I. In this case, both

systems (either 'I-XCSR' (the reinforcement learning system) or 'I-UCS' (the supervised learning system)) successfully triggered S to change the problem's difficulty. When the threshold value was set to 10%, the frequency of peak performance was higher than when set to 20%, for almost all the problem domains. The results also showed I (either I-XCSR or I-UCS) was able to trigger S to tune the problem's difficulty either to make the problem 'harder' or 'easier'. If the difference in performance between R and I was less than the threshold value, S tuned the problem's difficulty to be 'harder'. Conversely, if the difference in performance between R and I was greater than the threshold value, S tuned the problem's difficulty to be 'easier'. In this case, the ideal threshold value was 10%, where I successfully triggered S to change the difficulty levels of the problem and increased the frequency of peak performance.

4.3.3 Triggering Agent's and Learning Agent's Performance

In this section, the results of the classification agents (i.e. the frequency of peak performance and the data analysis) and the generation agent in the Three-Cornered Coevolution System are presented. A set of experiments was performed in order to investigate the ability of I[15] either as being suitable to be a *learning agent* or a *triggering agent* when the difference in performance between I and R was less than the threshold value: 1) 10%, and 2) 20%.

The classification agent that is set to trigger S to change the problem's difficulty is termed the *triggering agent*. The difference in performance between the triggering agent and the learning agent is defined as follows:

$$T = TriggeringAgentperformance - LearningAgentperformance \quad (4.1)$$

[15]either 'I-UCS' that used the supervised learning system (i.e. UCS) or 'I-XCSR' that used the reinforcement learning system (i.e. XCSR)

If I is set as the triggering agent, T is defined as follows:

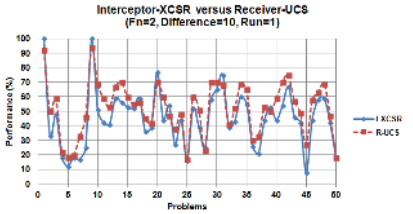
$$T = Iperformance - Rperformance \tag{4.2}$$

where I is the triggering agent, and R is the learning agent.

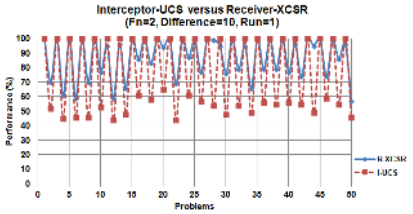
Initially, I was the triggering agent when I triggered S to change the problem's difficulty, while R was the learning agent. R and I were applied to the four problem domains (i.e. `Fn=2` to `Fn=5`), where S tuned the difficulty levels based on the difference in performance between I and R.

The **first and second sets of experiments** were performed in order to investigate the ability of I-XCSR and I-UCS to be suitable either as a triggering agent or learning agent.

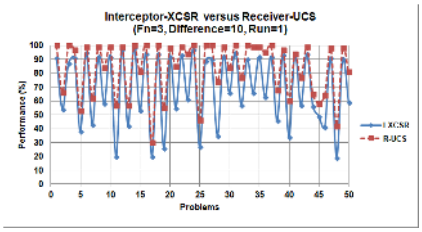
Figure 4.34 and Figure 4.35 show the frequency of peak performance between I-XCSR and I-UCS to trigger S to change the problem's difficulty, when the threshold value is set to 10% and to 20% respectively, for a single run. The frequency of peak performance is higher for almost all the problem domains when the threshold value is set to 10% compared to 20%, either I-XCSR or I-UCS is used to trigger S to change the problem's difficulty.



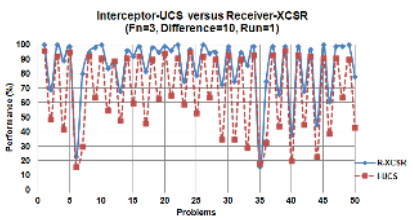
(a) I-XCSR trigger S, Fn=2.



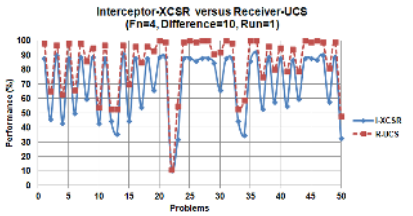
(b) I-UCS trigger S, Fn=2.



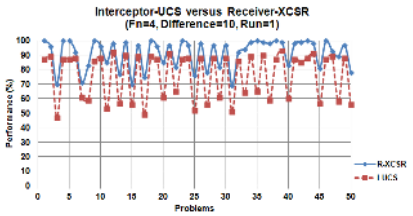
(a) I-XCSR trigger S, Fn=3.



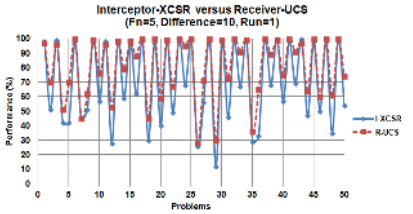
(b) I-UCS trigger S, Fn=3.



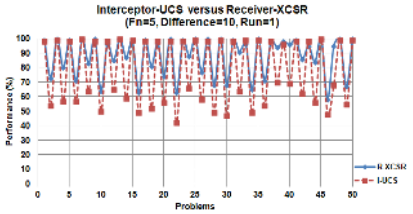
(a) I-XCSR trigger S, Fn=4.



(b) I-UCS trigger S, Fn=4.

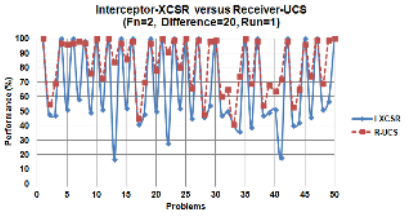


(a) I-XCSR trigger S, Fn=5.

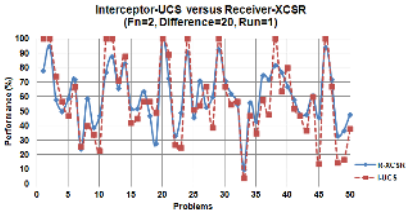


(b) I-UCS trigger S, Fn=5.

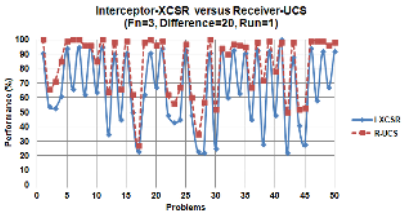
Figure 4.34: I-XCSR versus I-UCS triggers S to change the difficulty levels (threshold=10).



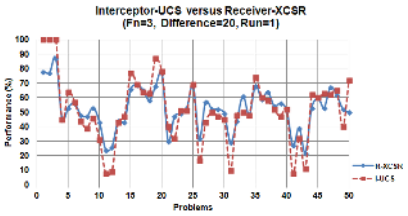
(a) I-XCSR trigger S, Fn=2.



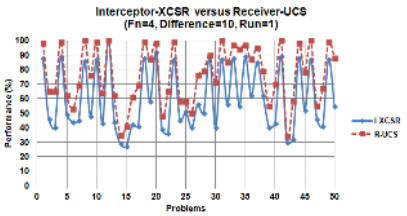
(b) I-UCS trigger S, Fn=2.



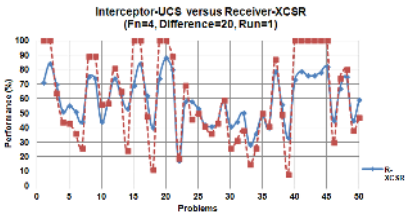
(a) I-XCSR trigger S, Fn=3.



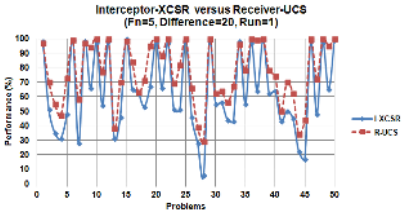
(b) I-UCS trigger S, Fn=3.



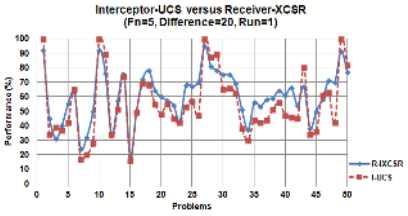
(a) I-XCSR trigger S, Fn=4.



(b) I-UCS trigger S, Fn=4.



(a) I-XCSR trigger S, Fn=5.



(b) I-UCS trigger S, Fn=5.

Figure 4.35: I-XCSR versus I-UCS trigger S to change difficulty levels (threshold=20).

Table 4.28 and Table 4.29 compare the percentage of peak performance between I-XCSR and I-UCS to trigger S to change the problem's difficulty, when the threshold value is set to 10% and to 20% respectively, for a single run.

Table 4.28: Percentage of peak performance between I-XCSR and I-UCS (threshold=10) for 50 problems.

Problem Domain	**I-XCSR trigger S (% peak performance)**	**I-UCS trigger S (%peak performance)**
Fn=2	20	50
Fn=3	48	48
Fn=4	36	50
Fn=5	46	50

Table 4.29: Percentage of peak performance between I-XCSR and I-UCS (threshold=20) for 50 problems.

Problem Domain	**I-XCSR trigger S (% peak performance)**	**I-UCS trigger S (%peak performance)**
Fn=2	40	18
Fn=3	42	14
Fn=4	34	22
Fn=5	32	20

The percentage of peak frequency is higher for all of the problem domains, if I-UCS is used to trigger S to change the problem's difficulty and the threshold value is set to 10%. Conversely, the frequency of peak performance is higher for all of the problem domains, if I-XCSR is used to trigger S to change the problem's difficulty and the threshold value is set to 20%.

Table 4.30 and Table 4.31 present the analysis data when I-XCSR is used to trigger S to change the problem's difficulty where the threshold value is set to 10 and 20 respectively. Table 4.32 and Table 4.33 present the analysis data when I-UCS is used to trigger S to change the problem's difficulty, where the threshold value is set to 10% and 20% respectively. The second column indicates R's and I's best performance and the third column indicates R's and I's worst performance. The next column gives R's and I's average performance over 50 problems. The fifth column gives the standard deviation of R's and I's performance (e.g. the data distribution about the mean value). The last column gives the standard error of R's and I's performance (e.g. how the mean varies with different experiments measuring the same quantity). Results are varied when different learning techniques (i.e. UCS and XCSR) are used in I to trigger S to change the problem's difficulty.

Table 4.30: Analysis data, when I-XCSR triggers S to change the difficulty levels (threshold=10) for 50 problems.

Problem Domain	Agent	Maximum Performance	Minimum Performance	Average Performance	Std	Std Error
Fn=2	I-XCSR	100.0	8.0	45.2	19.69	2.78
	R-UCS	94.0	17.0	52.3	18.05	2.55
Fn=3	I-XCSR	98.0	19.0	69.2	25.66	3.63
	R-UCS	100.0	2.0	83.8	19.37	2.74
Fn=4	I-XCSR	92.0	11.0	71.1	21.89	3.10
	R-UCS	100.0	11.0	84.8	19.62	2.77
Fn=5	I-XCSR	100.0	12.0	70.8	27.75	3.92
	R-UCS	100.0	28.0	80.9	21.29	3.01

Table 4.31: Analysis data, when I-XCSR triggers S to change the difficulty levels (threshold=20) for 50 problems.

Problem Domain	Agent	Maximum Performance	Minimum Performance	Average Performance	Std	Std Error
Fn=2	I-XCSR	100.0	17.0	66.96	27.86	3.94
	R-UCS	100.0	41.0	82.22	17.90	2.53
Fn=3	I-XCSR	100.0	22.0	66.08	25.48	3.60
	R-UCS	100.0	27.0	82.40	20.16	2.85
Fn=4	I-XCSR	100.0	27.0	59.86	21.91	3.10
	R-UCS	100.0	34.0	76.60	19.54	2.76
Fn=5	I-XCSR	100.0	6.0	63.76	26.65	3.77
	R-UCS	100.0	29.0	77.20	20.96	2.96

Table 4.32: Analysis data, when I-UCS triggers S to change the difficulty levels (threshold=10) for 50 problems.

Problem Domain	Agent	Maximum Performance	Minimum Performance	Average Performance	Std	Std Error
Fn=2	I-UCS	100.0	44.0	76.32	24.03	3.40
	R-XCSR	100.0	57.0	87.40	13.96	1.97
Fn=3	I-UCS	96.0	16.0	67.08	26.49	3.75
	R-XCSR	100.0	16.0	85.12	19.85	2.81
Fn=4	I-UCS	100.0	21.0	71.48	24.30	3.44
	R-XCSR	100.0	45.0	87.64	15.15	2.14
Fn=5	I-UCS	100.0	42.0	77.64	21.51	3.04
	R-XCSR	100.0	58.0	88.08	13.38	1.89

Table 4.33: Analysis data, when I-UCS triggers S to change the difficulty levels (threshold=20) for 50 problems.

Problem Domain	Agent	Maximum Performance	Minimum Performance	Average Performance	Std	Std Error
n=2	I-UCS	100.0	4.0	57.82	26.61	3.76
	R-XCSR	100.0	10.0	60.44	19.52	2.76
Fn=3	I-UCS	100.0	8.0	51.80	22.16	3.13
	R-XCSR	87.0	22.0	52.72	14.37	2.03
Fn=4	I-UCS	100.0	8.0	60.50	29.48	4.17
	R-XCSR	88.0	17.0	58.90	16.71	2.36
Fn=5	I-UCS	100.0	16.0	54.88	21.70	3.07
	R-XCSR	95.0	19.0	60.40	17.42	2.46

When I-XCSR was used to trigger S to change the problem's difficulty and the threshold value was set to 20% compared with 10%, for all the problem domains, R achieved 100% classification performance, and R-UCS's worst classification performance also increased (see Table 4.31). R-UCS's average classification performance was above 75% and the standard error of R was below than 3.0 for all the problem domains (see Table 4.31), again when I-XCSR was used to trigger S to change the problem's difficulty and the threshold value was set to 20% rather than 10%.

When I-UCS was used to trigger S to change the problem's difficulty and the threshold value was set to 10% rather than 20%, R-XCSR achieved 100% classification performance for all the problem domains except for `Fn=3`, and R-XCSR's worst classification performance also increased except for Fn=3 (see Table 4.32). R-XCSR's average classification performance was above 85% and the standard error of R was below 3.0 for all the problem domains again when I-UCS was used to trigger S to change the problem's difficulty and the threshold value was set to 10% rather than 20% (see Table 4.32).

It is noted that in certain problem domains R's and I's classification performance are below 50% for solving the binary classification. This is because the results are recorded from a single run instead of the average of 30 runs for learning a number of problems, where the classification performance is traced from every 100 exploit trial.

Table 4.34 and Table 4.35 show the mean square error (MSE) between R-UCS and R-XCSR for the first 10 problems when either I-XCSR or I-UCS is used to trigger S to change the problem's difficulty, and the threshold value is set to 10% and 20% respectively. The first column is R-UCS's MSE when I-XCSR is a triggering agent. The second column is R-XCSR's MSE when I-UCS is a triggering agent.

Table 4.34: Mean square error (MSE) between R-XCSR and R-UCS (threshold=10).

Problem Domain	**MSE R-UCS (triggering agent: I-XCSR)**	**MSE R-XCSR (triggering agent: I-UCS)**	**Result**
Fn=2	158.7	190.1	R-UCS $<$ R-XCSR
Fn=3	180.6	650.0	R-UCS $<$ R-XCSR
Fn=4	218.0	463.0	R-UCS $<$ R-XCSR
Fn=5	172.3	165.0	R-UCS $>$ R-XCSR

Table 4.35: Mean square error (MSE) between R-XCSR and R-UCS (threshold=20).

Problem Domain	**MSE R-UCS (triggering agent: I-XCSR)**	**MSE R-XCSR (triggering agent: I-UCS)**	**Result of MSE**
Fn=2	490.6	194.9	R-UCS $>$ R-XCSR
Fn=3	392.9	157.7	R-UCS $>$ R-XCSR
Fn=4	313.6	224.0	R-UCS $>$ R-XCSR
Fn=5	333.7	117.2	R-UCS $>$ R-XCSR

Table 4.34 shows that the mean square error of R-UCS is small on all the problem domains except for `Fn=5`, when I-XCSR is the triggering agent when the threshold value is set to 10%. The results indicate that R-UCS is suitable to be the learning agent when MSE R-UCS is smaller than R-XCSR. Table 4.35 show R-XCSR's mean square error is small on all the problem domains, when I-UCS is the triggering agent and the threshold value is set to 20%. Here, the results indicate that R-XCSR is suitable to be the learning agent when MSE R-XCSR is smaller than R-UCS.

Therefore, the results suggest that XCSR was suitable to be the *triggering agent*, while UCS should be the learning agent when the threshold value was set to 10%. Conversely, the results suggest that XCSR was more suitable to be the *learning agent* rather than the triggering agent when the threshold value was set to 20%.

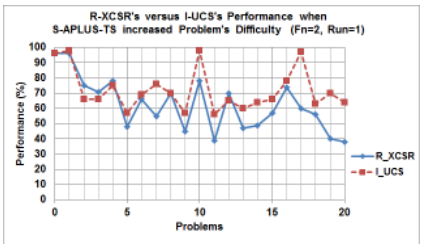
4.3.4 Classification Agents' Performance when Problem's Difficulty Increased

In this section, the results of the classification agents (i.e. the classification performance) and the generation agent (i.e. changes of features `F` in the problem) in the Three-Cornered Coevolution System are presented. In the previous section S tunes the problem's difficulty to be either 'harder' or 'easier' (see Section 4.3.3). Here, S consecutively tunes the problem's difficulty to be 'harder' and generates various 'difficult' problems for classification. A set of experiments was performed in order to investigate the capability of R and I using different types of learning systems[16] to learn, while S consistently increased the problem's difficulty. R and I were applied to the four problem domains (i.e. `Fn=2` to `Fn=5`), where S tuned the difficulty levels to be 'harder' to increase either R's or I's classification performance at a time. The threshold value was set to 10%.

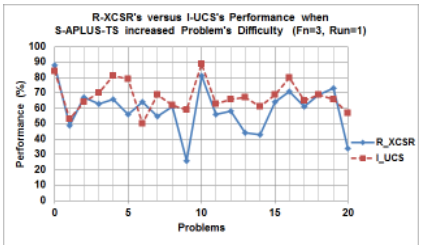
[16] either the supervised learning system (i.e. UCS) or the reinforcement learning system (i.e. XCSR)

In **the first set of experiments**, the capability of R-XCSR and I-UCS is investigated when S increases the difficulty levels for the four problem domains. S is tasked to *minimize I-UCS's performance*.

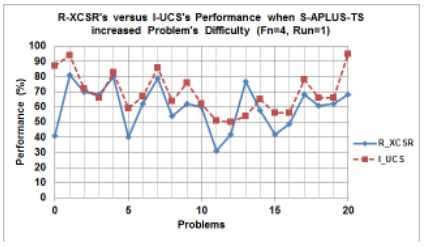
Figure 4.36 depicts the classification performance between R-XCSR and I-UCS on the four problem domains (i.e. `Fn=2` to `Fn=5`) when S increases the problem's difficulty. The plots show that for most of the problems, UCS outperforms XCSR, as expected, due to the increased of domain information. Next, these results are discussed in more detail (see Table 4.36).



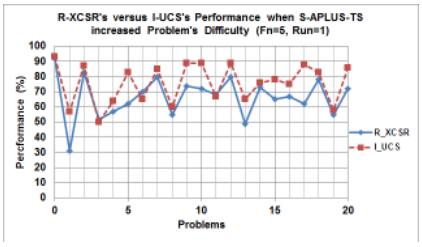
(a) Fn=2.



(b) Fn=3.



(a) Fn=4.



(b) Fn=5.

Figure 4.36: R-XCSR's versus I-UCS's performance when S consistently tunes the problem's difficulty to be 'harder' while minimizing I-UCS's performance.

Table 4.36 describes the changes of features F in the problem (i.e. examples of problem, from the initial problem up to 10 problems) when S increases the problem's difficulty while *minimizing I-UCS's classification performance* for Fn=2. S starts with the initial problem (the first row), where R-XCSR's and I-UCS's classification performances are shown in the last two columns. Next, the initial problem is adjusted to the next problem (the second row) by S to be 'harder' which affects R-XCSR's and I-UCS's classification performance (the last two columns), and so on. It is noted that when Fn=2 and S is tasked to make the problem 'harder', S sets Fd to be 1 (i.e. from Fd=0 to Fd=1). The results indicate that I-UCS achieves better results compared to R-XCSR, specifically for addressing the class imbalance problems (i.e. problem *1, 5, 6, 9* and *10*, when the class balance (i.e. Fcbl) decreases. Here, one class (i.e. 'Class 0') is represented by a larger number of instances than the other class (i.e. 'Class 1'). In the noisy problem (i.e. problem *4, 7* and *10*), where the noise that applies to the action attribute (class) of the instance (i.e. Fan) is higher, I-UCS is more robust than R-XCSR. I-UCS also outperforms R-XCSR in the problem where the decision boundary (i.e. Fcbd) is small (i.e. problem *1, 5* and *6*). The results from the other problem domains (i.e. Fn=3, Fn=4 and Fn=5) show the same pattern which is not shown here.

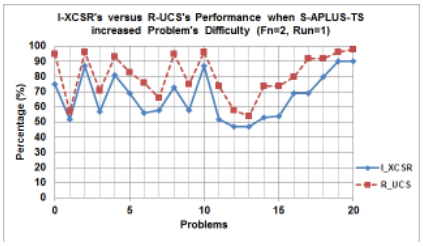
R-XCSR performed better than I-UCS when the class balance (i.e. Fcbl) was set in the range of 50-60%, while the noise that applied to the action (i.e. Fan) and condition (i.e. Fcn) was low (i.e. problem *2, 3* and *4*). Conversely, when the values of the class balance (i.e. Fcbl), the noise that applied to the action (i.e. Fan) and condition (i.e. Fcn), and the class boundary (i.e. Fcbd) were increased, I-UCS achieved better performance than R-XCSR (i.e. problem *5* to *10*).

Table 4.36: Changes of features `F` when S increases problem's difficulty while *minimizing I-UCS's performance* (`Fn=2`).

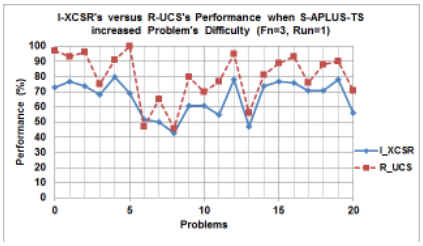
Problem Number	**Fc**	**Fd**	**Fi**	**Fr**	**Fan**	**Fcn**	**Fcbl**	**Fcbd**	**R-XCSR (%)**	**I-UCS (%)**
0	0	0	0	1	5	5	50	5	96	96
1	0	1	1	1	3	5	26	3	96	98
2	0	1	1	1	2	1	60	0	75	66
3	0	1	1	1	0	4	65	4	71	66
4	0	1	1	1	5	38	48	16	78	75
5	0	1	0	1	3	1	78	0	48	57
6	0	1	0	1	1	4	76	2	66	69
7	0	1	1	1	4	33	56	6	55	76
8	0	1	1	1	1	0	78	2	70	70
9	0	1	1	1	3	3	45	36	45	57
10	0	1	1	1	4	26	25	4	78	98

In **the second set of experiments**, the capability of R-UCS and I-XCSR is investigated when S increases the difficulty levels for the four problem domains. S is tasked to *minimize I-XCSR's performance* instead of R-UCS's performance.

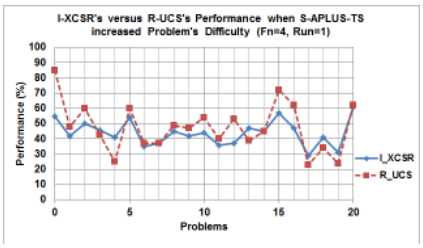
Figure 4.37 depicts R-UCS's and I-XCSR's classification performance on the four problem domains (i.e. `Fn=2` to `Fn=5`) when S increases the problem's difficulty. The plots illustrate that for most of the problems, R-UCS achieves better performance compared to I-XCSR. Again UCS outperforms XCSR for almost all the problems, where these results are further described in Table 4.37.



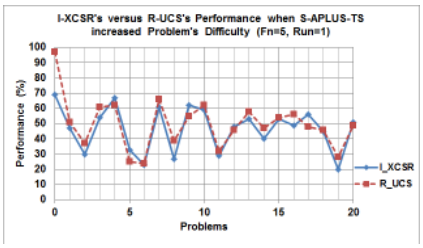
(a) Fn=2.



(b) Fn=3.



(a) Fn=4.



(b) Fn=5.

Figure 4.37: R-UCS's versus I-XCSR's performance when S consistently tunes the problem's difficulty to be 'harder' while minimizing I-XCSR's performance.

Table 4.37 describes the changes of features F in the problem (i.e. examples of problem, from the initial problem up to 10 problems) when S increases the problem's difficulty while *minimizing I-XCSR's classification performance* for Fn=2. S starts with the initial problem (the first row), where R-UCS's and I-XCSR's classification performance are shown in the last two columns. Next, the initial problem is adjusted to the next problem (the second row) by S to be 'harder' which affect R-UCS's and I-XCSR's classification performance (the last two columns), and so on. The results indicate R-UCS outperforms I-XCSR for addressing the class imbalance problems (i.e. problem *2, 3, 5, 7, 8, 9* and *10*), when the class balance (i.e. Fcbl) decreases. However, in the noisy problem (i.e. problem *7* and *10*), where the noise that applies to the condition attribute of the instance (i.e. Fcn) is higher, there is a significant difference between R-UCS's performance and I-XCSR's performance. In the case of a smaller decision boundary (i.e. problem *6*), R-UCS also outperforms I-XCSR. The results from the other problem domains (i.e. Fn=3, Fn=4 and Fn=5) show the same pattern which is not shown here for the sake of brevity.

R-UCS achieved better performance compared to I-XCSR even though S increased the problem's difficulty by increasing the value of the class balance (i.e. Fcbl) and the noise that applied to the action (i.e. Fan) and condition (i.e. Fcn) for all problems except for problem 1. I-XCSR performed better than R-UCS when the class balance (i.e. Fcbl) was set to 100% (all instances were set to 'Class 1' or 'Class 0'), while the noise that applied to the action (i.e. Fan) was nearly 50% and the noise that applied to the condition (i.e. Fcn) was 0% (i.e. problem 4).

Table 4.37: Changes of features `F` when S increases problem's difficulty while *maximizing I-XCSR's performance* (`Fn=2`).

Problem Number	**Fc**	**Fd**	**Fi**	**Fr**	**Fan**	**Fcn**	**Fcbl**	**Fcbd**	**I-XCSR (%)**	**R-UCS (%)**
0	0	0	0	1	5	5	50	5	75	95
1	0	1	1	1	39	2	0	3	58	61
2	0	1	1	0	1	0	1	1	66	99
3	0	1	1	1	4	4	1	3	66	97
4	0	1	1	1	46	0	0	0	58	57
5	0	1	1	1	2	4	1	3	66	100
6	0	1	0	0	4	1	84	0	50	57
7	0	1	0	1	3	2	3	3	62	94
8	0	1	0	0	2	4	3	4	64	96
9	0	1	0	0	1	3	2	2	70	96
10	0	1	1	0	3	0	3	2	69	97

4.3.5 Discussions and Findings

The overall aim of this work is to develop a *Three-Cornered Coevolution System*, where three different agents work cooperatively and adapt to the changes of the problem. The goal has been achieved as the system is capable of triggering (activating) coevolution between the participating agents in the system when necessary without human involvement. S was able to tune and adjust the difficulty levels based on the difference in performance between R and I, while both R and I learned the problem using different techniques of learning.

In Section 4.3.1, the new classification agent (i.e. the Interceptor) that used supervised learning LCSs, I-UCS was applied to the four problem domains (i.e. `Fn=2` to `Fn=5`), where the difficulty levels were tuned and adjusted by S. The results confirmed that S-APLUS-TS was successful and effectively tuned and adjusted the problem's difficulty by varying features in the problem compared to S-TS, to either maximize or minimize I's classification performance. Moreover, the results also suggested I-UCS was able to solve the problems as expected.

In Section 4.3.2, the classification agents' performance (i.e. R's and I's frequency of peak performance, where R and I used two different learning systems) in the coevolutionary system was investigated. In this case, the frequency of peak performance was higher on the four problem domains when the threshold value was set to 10% compared to when the threshold value was set to 20%, where I-UCS was able to trigger S to change the problem's difficulty and improved R's performance. In addition, S was able to change features in the problem to either make the problem 'harder' or 'easier'. If the difference in performance between R and I was greater than the threshold value, S was able to change the problem's difficulty to become 'easier' which decreased the gap between R and I. Conversely, if the difference in performance between R and I was smaller than the threshold value, S was able to change the problem's difficulty to 'harder' which increased the gap between R and I and decreased R's and I's performance.

In Section 4.3.3 a set of experiments was performed to investigate the effectiveness of using two different learning systems in I in order to determine whether UCS or XCSR was more suitable to be the learning agent or the triggering agent. In this case, when the threshold value was set to 10% rather than 20%, the results suggested that XCSR was suitable to be the triggering agent, while UCS being the learning agent, even though I-UCS was able to improve R's performance. However, the results suggest that XCSR was more suitable to be the learning agent rather than the triggering agent, even though I-XCSR was able to improve R's performance, when the threshold value was set to 20% instead of 10%.

In Section 4.3.4, the capability of the classification agents (i.e. R and I) that used different types of learning systems (i.e. either UCS or XCSR) on 'hard' problems was investigated, where the threshold value was set to 10. For almost all the problems, UCS outperformed XCSR, even though S was tasked to maximize XCSR's performance. In comparison, UCS was specifically designed for supervised learning, while XCS was more general and learned only from the feedback about action consequences [87]. It was expected that UCS would perform well in all tested problems as UCS was provided with the known label. The results indicated that UCS was more robust than XCSR for addressing the class imbalance problems and UCS was less sensitive to parameter settings compared to XCSR.

It was found that both systems (i.e. UCS and XCSR) were not achieved 100% performance for solving a certain 'hard' problems created by S due to the same parameter setting being used for XCSR and UCS for all of the experiments. In [84, 85], the paper provides a few resampling techniques that allow UCS to deal with the class imbalance problems more effectively. However, these techniques were not applied for this work. In [89, 90], the author reports there are two critical parameters in XCS for optimal performance (i.e. the learning rate and the GA triggering threshold θ_{GA}). Again, the same learning rate and the GA triggering threshold θ_{GA} was used in XCSR. In future, if XCSR is configured appropriately (e.g. adaptive

tuning on the two critical parameters), it is hypothesised that a significant improvement of XCSR's performance can be achieved.

4.4 Summary and Way Forward

In the *Three-Cornered Coevolution System*, I was the main agent that triggered the coevolutionary process, while S was the main agent that controlled the coevolutionary process between the participating agents when necessary. S was able to tune and adjust the problem's difficulty autonomously based on the classification agents' ability to learn within the given threshold, either to make the problem 'harder' or 'easier'. Further, the behaviour of two different learning systems (i.e. UCS and XCSR) depending on the type and the percentage of noise, class imbalance and the characteristics of the data was investigated. As S increased the problem's difficulty, R's and I's performance were varied, where features in the problem (i.e. `Fan`, `Fcn`, `Fcbl` and `Fcbd`) that affected either UCS's or XCSR's performance were apparent. Therefore, if both learning systems are configured appropriately (i.e. to self-adapt during learning), both systems become competitive in solving complex classification problems.

The core principle of the research to develop a novel *Three-Cornered Coevolution System*, where three different agents work cooperatively in a coevolutionary system has been achieved. The generation agent was able to determine the effect on the triggering of the coevolutionary process between the classification agents, to successfully tune and adjust the problem's difficulty. In future, the generation should be further enhanced to let the classification agents continuously improve, especially when the problem's difficulty becomes 'harder' and 'complex'. Thus, the cooperative coevolution between all the agents can be performed in such a way that each agent attempts to evolve and achieve its best performance consistently. Instead, a cooperative coevolution can be further implemented in the Three-Cornered Coevolution System.

Chapter 5

Discussion

A theoretical framework of *Three-Cornered Coevolution* was proposed by Wilson in the paper 'Coevolution of Pattern Generators and Recognizers' [124]. The author proposed an automatic system for creating a pattern generator and two pattern recognizers that might provide new and human-independent insight into the pattern recognition problem [124]. However, this theoretical framework had not yet been implemented and tested. It was not known how the coevolutionary process would actually work between the pattern generator and the pattern recognizers within the system.

Several issues were identified in Wilson's paper. First, is the coevolutionary framework relevant to the way natural patterns form? Secondly, if the framework functions as a pattern recognizer, will the system evolve similar methods to human saccades? Thirdly, how well will the framework drive the coevolution in the system, given that one of the pattern recognizers is provided with prior information? Finally, if the framework works, will the results have wider relevance than image classification?

In order to address the above issues, this work has dealt with the first and the third issues specifically. In answering the first issue, two problem domains have been developed: image-based data and artificial data. The results showed that the artificial data was more appropriate for artificial systems compared to the image-based data in order to develop the coevo-

lutionary system. In this domain, a variety of datasets can be built with different characteristics defined by a set of difficulty factors (i.e. dimensionality, noise, class balance, decision complexity) for generating various classification problems at different levels of difficulty. Further, different types of difficulty factors for the classification tasks can be introduced to test the learner's ability, which can be assessed using different types of performance measure (i.e. robustness, scalability, and predictive accuracy). Within this domain, when the coevolutionary process is triggered, the pattern generator is able to coevolve and adapt with the changes (i.e. to tune and adjust the problem's difficulty appropriately based on the pattern recognizers' classification performance).

As the focus was now on the artificial data task, the second issue on human saccades evolution was not considered in the work for this thesis. However, this issue can be considered in future work, where it can be handled using image-based data.

In addressing the third issue, this work applied different techniques of learning (i.e. supervised learning and reinforcement learning) in the pattern recognizers, whereas the pattern generator was provided with useful information regarding the problem domain. Here, the pattern generator learned for the benefit of the group of learning systems [97] (i.e. to tune and adjust the problem's difficulty appropriately based on the pattern recognizers' classification performance).

Learnt information regarding the problem domain was therefore useful in order for the pattern generator to generate the next problem at appropriate levels of difficulty. On the other hand, in order for the pattern recognizers to address the on-line arrival of problem and coordination knowledge more flexibly, the pattern recognizers should depend less on shared-information [104] so that it can adapt to changing environments. Here, the pattern recognizers are given their own learning goal (i.e. to solve the classification problems) independent from the pattern generator's goal.

The proposed framework for a human-independent pattern recogni-

tion system by Wilson provided a vast opportunity for research. The framework offers the potential for autonomous learning, extending to complex image-based data (pattern) and insight into coevolution learning over standard studies of pattern recognition that can be explored further.

In the next section, related issues and highlights of overall findings for each phase are discussed. Insightful discussions are provided to answer the above issues based on the experimental results from the previous chapter.

5.1 Phase 1: An Evolvable Problem Generator

In order to develop a *human-independent system* for addressing classification problems, the system should be capable of creating an appropriate problem domain where it could be changed and adjusted in meaningful ways. Thus, Phase 1 was necessary to establish an appropriate problem domain for classification tasks that could be evolved and tuned automatically as a basis of the Three-Cornered Coevolution System. In Phase 1, two different problem domains for classification tasks that can be tuned automatically (i.e. image-based data and artificial data) have been established.

For simplicity, a method to generate image-based data with different dimensionality as an initial problem domain was developed. However, the image-based data were not suitable for application to the Three-Cornered Coevolution System. This was because important features in the problem that altered the classification performance of the pattern recognizers were not transparent.

In this image-based data task, the underlying feature relationships that control the ease of learning within the problem domain were not easily separated. Findings showed that the generated image-based data were not well separable: one pattern may be labeled to more than one class, which leads to data ambiguity and class imbalance. It has been discovered that there was no underlying relation between the resulting image-based data

(pattern) and the features in the problem in order to distinguish the class clearly. The problem formulation was not intrinsically separable, thus the set of features was not sufficient to describe the underlying concept.

In addition, the generation agent randomly generated image-based data without having any mechanism that could control certain features in the problem, such as data sparsity, noise and class balance. Therefore, the interpretation of the image-based data (pattern) created for each problem and mapped to corresponding difficulty levels were not clear. Moreover, a small number of certain patterns was rare, which introduced sparsity to the problems. This caused the system to wrongly generalize all such 'problems' as 'difficult'. However, many more samples would have resulted in the patterns being trivial to learn, but this would have been time consuming.

A further challenge for pattern recognition research was to create problems with large sets of examples that could be learned from [124]. When the number of instances and the dimensionality increased and the problem's difficulty became complex, the classification agent consumed a large amount of memory and demanded a much longer training time than would be expected. Therefore, the classification agent was only tested within a limited bound rather than the whole search space of the problem domain in order to investigate features in the problem that altered the classification agent's performance.

Instead of using the image-based data, it was now considered better to use the artificial data where various datasets for the classification tasks could be generated based on a list of problem-specific parameters. Findings strongly suggested that the method of using artificial datasets for classification was more relevant and suitable for several reasons. Various datasets for the classification task could be built with different characteristics defined by a set of factors controlling difficulty (i.e. dimensionality, noise, class balance, decision complexity) as suggested in [39, 103, 83]. Furthermore, a wide range of the complexity factors in data mining could be

introduced to test the classifier's ability, which could be measured based on different types of performance measure such as robustness, scalability, and predictive accuracy as described in [72, 73]. It was considered better to generate the datasets by specifying features rather than patterns. The problem generation of artificial datasets for pattern extraction tasks has led to a system that can tune and adjust the problem's difficulty based on the learner's ability to learn.

5.1.1 Tuning of Pattern Recognition Task versus Interacting with a Learning System

It was important to ensure that the problem domain was flexible in order to be tuned and adjusted, e.g. the difficulty levels of the problem could either be increased or decreased. It was hypothesised that by tuning a certain feature(s) in the problem the pattern generator could make the problem either 'harder' or 'easier' to learn. The findings indicated that the new problem domain of artificial data for classification can be tuned and adjusted in meaningful ways so that the gradient of performance related to the difficulty levels exist (see Figure 4.14). Within this domain, the classification agent's ability can be investigated under a known scenario; where the artificial data can be generated according to a particular known feature of the problem. Thus, issues of problem difficulty such as noise, class balance, decision complexity and many others can be introduced in a controlled way.

Based on the enumerative analysis of the potential datasets (see Figure 4.14), the gradient of performance related to the difficulty levels were identified and became apparent. The surface landscape of the classification agent's performance showed the trade-off surface for each problem domain that altered the classification agent's performance when a certain feature in the problem was adjusted. The results indicated that a gradient in difficulty existed in relation to features in the problem.

Next, the method of the automated evolvable problem generator was introduced for generating various artificial data. Here, EC algorithms, specifically GA in an LCS, were applied to the automated problem generator. The GA was commonly used for rule discovery. GA was used to create various artificial data for the classification tasks in order to determine the classification agent's behaviour and test the classification agent's accuracy under various scenarios. The results showed that GA efficiently created the evolutionary processes within the system compared to the enumerative method. Millions of generations of the artificial data could be executed and repeated on a computer under various circumstances. Further, GA enabled the system to automatically deploy solutions for more complex problems with less time.

The introduced method of the automated evolvable problem generator enabled the production of scalable and evolvable artificial data for classification with various levels of difficulty. In addition, the problem generator allowed detailed analysis of the results to high precision to be performed in comparison with the enumerative method. Details of the analysis, specifically on the important areas of the search space that contains useful information to the problem generator, could be conducted.

The conceptual distinction between the new method and the enumerative method is that, in the enumerative method, each and every point of the search space (i.e. the list of problem-specific parameters) needs to be evaluated in order to achieve the optimal solution, which is very time consuming. The enumerative method may fail to search the space of any problem with moderate size and complexity because it may become simply impractical to search all the points in the space [8]. The problem generator has a list of problem-specific parameters that can generate random instances for each parameter value. Further, the characteristics of the problem can be tuned by adjusting the problem-specific parameters.

This method enabled a systematic investigation of the learner's ability on the most important sub-set of the problem domain (i.e. the hardest

parameter range of the problem) to be performed. In particular, when there were many combinations of features in the problem, the problems would be hard to learn. Thus, results relating the problem's characteristics to learner performance could be extracted (see Figure 4.14) and further investigation on the interaction of problem-specific parameters could be performed. For instance, important features which control the ease of learning within the problem domain for the classification system were identified e.g. extreme levels of noise.

5.1.2 Selection of Agent

In this work, the *coevolution* approach required an interaction between the agents, where each agent attempted to adapt with one another's changes in an *on-line* manner. Therefore, both of the agents (i.e. the generation agent and the classification agent) were designed as on-line systems. Theoretically, the on-line system has the advantage over the off-line system in the following ways. First, in order to learn and continue to adapt to the changes of the environment (i.e. the coevolutionary approach), the on-line system is necessary. Secondly, the training data (i.e. image-based data or artificial data created *on-the-fly*) can be generated easily, based on particular problem-specific parameters during the system's operational phase. This method allows a systematic investigation of the learner's ability by relating the problem characteristics to the learner's performance to be performed. This includes determining the learner's optimal performance at different stages of the iterations. Further, various types of problems can be generated instantly to test the learner's performance and ability based on a certain performance measure. Thirdly, the on-line system is potentially more robust because errors or omissions, while creating the training set, can be corrected during the system's operational phase. In contrast, the *off-line* system is presented with the whole training set in order to solve the problem at hand. As a consequence, it is a waste of computational re-

sources and time to track the learner's performance at each iteration until the end of the process, especially when the learner does not reach its highest performance. By the time the process is finished, it is too late to amend and make any correction.

In this work, Learning Classifier Systems (LCSs) are mainly applied as the learning systems. LCSs were chosen to address the classification problems based on the potential characteristics described as follows:

1. *Interpretability* [2]: The system produced solutions in a form of human-readable 'condition-action' rules. The rules were easily interpreted and readable, which suggested interesting dependencies of attributes in the problem under study. For example, the classification agent 'condition-action' rules were retrieved from file(s) for further analysis (see Section 3.1.3.2 and Section 3.1.4.2). A study on the associated information regarding the problem domain was performed successfully in order to understand the classification agent's behaviour for a particular problem. The interpretability of the 'condition-action' rules offered a potential tool either for explanatory data analysis or predictive modeling analysis.

2. *Generalisation* [46]: The system has the capability to generalise over the input (state) in order to develop a compact description of the input-output map that has been learned. The introduction of this generality allowed the system to sample parts of the state space at different levels of abstraction [74]. The system was therefore able to represent learnt information in a compact form of rules and applied the learned knowledge to unseen input. For instance, each time the classification agent sensed inputs (i.e. image-based data or artificial data) a population of classifiers that matched the inputs were created. Based on its generalisation capability, the classification agent developed a compact description of the input-output map to successively classify each input in the correct class as suggested by its

classifiers (see Section 3.1.3.2 and Section 3.1.4.2).

3. *Variations in representation* [65]: The representation of knowledge in LCSs is flexible. The 'condition-action' rules referred as a classifier can be represented using different kinds of representations (e.g. binary, real-valued, messy, GP-like, symbolic expressions). For example, the classification agent's 'condition' was represented by the 'ternary' alphabet to specify the patterns for the image-based data (see Section 3.1.3.2). Further, the classification agent's 'condition' was represented by 'real-value' numbers for specifying each instance in the dataset for the artificial data (see Section 3.1.4.2). The classification agent's 'action' was encoded by an 'integer' value where the action could be either '1' for 'Class 1', otherwise '0' for 'Class 0'. Variations in representation enabled the system to be applied to many research areas for addressing different types of problems e.g. pattern recognition and data mining, where each classifier component was tailored to fit the need of a particular application without modifying the main structure of the system [65].

5.1.3 Problem Generator

Traditionally, research in pattern recognition involves choosing a domain, creating a source of exemplars, and trying out learning algorithms that seem likely to work in that domain [124]. Thus, an automatic problem generator (i.e. the generation agent) would be valuable to provide several advantages over the traditional method. The introduced method in this thesis was different from the traditional method and other methods mentioned in [71] in several ways.

1. The problem generator has a list of problem-specific parameters (i.e. `[Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd]`) that can generate a random instance for each combination of parameter values, which can

cover a wide range of the complexity factors (i.e. feature space dimensionality, noise, class balance, boundary complexity, sample sparsity, irrelevant and redundant attributes, and many other parameters) compared to the method proposed in [71]. The advantage of the introduced method was that the characteristics of the complexity factors can be included in the problem to generate various datasets at different levels of difficulty in order to investigate the learner's ability in a known scenario.

2. The problem generator was able to autonomously adjust and tune features in the problem for the generation of various datasets in order to test the learner's ability. Based on a certain problem-specific parameter setting, different types of performance measures can be adopted to assess the learner's performance. However, the work only focused on the classification agent's performance (i.e. classification accuracy), being a standard measure in the area. Nevertheless, the performance measure was not limited only to the classification accuracy, which can be further extended to such matters as robustness, scalability and relevance. For example, the learner's scalability could be tested by varying the number of features and the number of instances in the problem's set-up.

3. The problem generator enabled the generation of a large set of examples (i.e. creating a library of problems) autonomously. The system was completely self-contained [74] (i.e. no human in the loop). The problem generator was able to generate various datasets with a varying number of problems depending on the problem-specific parameter which was set-up autonomously. Random instances for each problem-specific parameter (i.e. `[Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd]`) could therefore be created automatically, and this changed the composition of the dataset at each run. A large set of examples for each class that were diverse and numerous could be produced

according to the setting of the number of instances in the datasets `Fn` coupled with other associated parameters. The introduced method enabled an investigation of the learner's ability to solve different types of problems to be performed.

4. Most importantly, problems of encoding and decoding for generating various problems could be handled efficiently with minimal error by the problem generator. The problem generation task could be conducted with less human intervention for setting and tuning the problem characteristics for generating variants classification problems at different levels of difficulty (see Table 4.10, Table 4.11, Table 4.17 and Table 4.13). Instead, the task could now be performed by the problem generator autonomously (i.e. adjusting and tuning certain features in the problem to generate various datasets). When thousands or millions of instances in the datasets need to be produced for each problem by changing and tuning certain parameters manually (i.e. either to increase or decrease the problem's difficulty), humans might make a mistake.

5. The artificial data could be used as a test bed on various problem domains to help in empirically testing the learning bounds of the classifiers. The list of problem-specific parameters (i.e. `[Fn Fc Fd Fi Fr Fan Fcn Fcbl Fcbd]`) can take any value up to a certain range. However, there was a limit for each feature value of the problem that would affect the classification performance and the learning bound of the learner. For example, when the problem set-up was set to [`Fn=2 Fc=1 Fd=0 Fi=0 Fr=0`], and the noise level that applied to the action was greater than 30% (`Fan`$\geq$30), the classification performance started to decrease to less than 60% (see Figure 4.14). This meant that this particular classification agent's performance would never achieve 100% performance if the noise level that was applied to the action was greater than 30% in this problem set-

up. Thus, instead of testing the classification agent on various problems, the task can be refined to identify how the classification agent can perform better with the problem (i.e. increasing or decreasing the classification agent's learning rate or the classification agent's error tolerance rate).

In summary, Phase 1 was necessary to establish an appropriate problem domain for classification that can be evolved and tuned automatically. Although the problem of automatic test generation (i.e. artificial data) has been considered in the literature from different perspectives, the generation of test-problems by coevolution for LCSs is quite new. In Phase 1, a novel system for an evolvable problem generator that can create various problems for classification has been established; a new human-independent system for the creation of various classification problems. Moreover, the problem domain can be evolved using LCSs and manipulated autonomously to generate a scalable and evolvable problem (i.e. image-based data or artificial data).

5.2 Phase 2: Two-Cornered Coevolution System

In order to develop a *coevolution* system for addressing the classification tasks, the system should be able to tune the problem appropriately (i.e. find an appropriate level of the problem's difficulty) based on the classification agent's performance. The classification agent can then be aligned to solve the classification task. Thus, Phase 2 was required in order to investigate the coevolutionary approach within the system, where two different agents (i.e. the classification agent and the generation agent) interact with each other to adapt to the changes within the problem.

Usually, the problem domain is created and controlled by humans. *Humans* set up and tune the problem domain, such as determining the problem's difficulty. Different problems are often required to be configured

with different parameters or strategies in order to improve the performance. For instance, one can perform trial-and-error tuning through extensive experiments, while others use their past knowledge or experience to tune the parameters differently, guided by some rules-of-thumb [36]. In the latter case, there is no formal tool or statistical methodology involved. The approaches are extremely tedious, error prone, unproductive and expensive [10]. Optimizing parameter values, is however, a non-trivial problem which is beyond the limits of human problem solving [81]. The problem of parameter tuning is hard because for any given application there is a large number of options, but only little knowledge about the effect of the parameters values on the learner's performance is provided [81]. In addressing this problem, a local search method (i.e. Tabu Search technique) was used to tune the parameter values (i.e. the problem's features) so that the generation agent could commence from the previous problem without requiring repetition of the same problem. The results (see Section 4.2.2.1) showed that the generation agent was able to find an appropriate level of the problem's difficulty autonomously (i.e. either increased or decreased the problem's difficulty based on the classification agent's performance), which was very crucial in this phase. Moreover, Phase 2 was a baseline for Three-Cornered Coevolution System.

Alternatively, automatic configuration and tuning can be considered as an integral part of the system development [10]. This method can reduce the effort needed for tuning and setting the parameter values in order to analyze the learner's ability (e.g. study how the learner's performance depends on its parameter values and how to choose parameter values that optimize the learner's performance). Instead of spending time to search for the appropriate parameter values blindly, the same effort can be spent to autonomously control the parameter values in order to find an optimal solution for a given task. In order to achieve this adjustable control over blind tuning, a suitable control mechanism is required.

There are two approaches in choosing and setting the parameter values

in the field of evolutionary computation [21]: 1) parameter tuning and 2) parameter control. In the case of *parameter tuning*, the parameter values are established before the run (i.e. fixed in the initialization stage) and do not change while the algorithm is running. However, in the case of *parameter control* the parameter values are given an initial value when starting the algorithm and the parameter values are change during the algorithm run.

In this phase, the *parameter control* scheme is selected to control the parameter value of features `F` in the problem. Using this scheme, the parameter values are initialized with an initial value when starting the algorithm and changed during run time. TS was applied to the generation agent to change the parameter values. First, the generation agent is initialised with an instance (set-up) of the problem. Secondly, TS changed the parameter values during run time for a number of iterations. Here, TS is used to define a new problem at an appropriate level of difficulty. Applying TS to the generation agent helped the agent to discover the best combinations of features in the problem that altered the classification agent's performance. The generation agent was able to search for the best combination of features in the problem (i.e. with the objective either to 'increase' or 'decrease' the classification agent's performance) by varying the difficulty levels (i.e. either to make the problem 'easier' or 'harder' to learn) (see Figure 4.24 and Figure 4.25). This method helped the generation agent to: 1) define the new problem at the appropriate levels of difficulty based on the classification agent's performance, 2) find the types of problems that were commensurate in the domain of competence of the classification agent, and 3) identify adversarial problems for classification tasks. However, the generation agent that used TS alone in its methods did not effectively minimize the classification agent's performance (see Figure 4.25). The findings showed that at a certain number of iterations, TS was likely to become stuck in local-optima when there was not much improvement in the features of the problem for the new problem. The cause was that the generation agent created the new problem within a similar parameter

setting, which did not effectively reduce the classification agent's performance to make the problem 'harder'.

Considering that Pittsburgh-style LCSs are very close to the essence of EC techniques, this approach was chosen in order to handle the problem of the parameter setting (i.e. when TS is likely to become stuck in local-optima). Pittsburgh-style LCSs (i.e. A-PLUS) was applied to the generation agent to evolve the generation agent's rules once the new problem was identified by TS.

The drawback of Pittsburgh-style LCSs is that the system explores a much larger search space compared to Michigan-style LCSs. Conversely, Pittsburgh-style LCSs usually evolve more compact populations involving only a few rules in a population compared to Michigan-style LCSs. For this work, Pittsburgh-style LCSs were more suitable when compact solutions with few rules were expected to solve the problem. Since Pittsburgh-style LCSs are often more computationally expensive as they need to evolve multiple rule-sets and require longer evaluation time to assess the whole population of multiple rule-sets, A-PLUS (Accuracy-based Pittsburgh Learner Using Subsumption) was selected. A-PLUS was capable of addressing the bloat phenomenon, which refers to increasing any variable-sized set of rules [1]. The results confirmed that applying A-PLUS in the generation agent had improved its rules where the generation agent was able to adjust the difficulty levels more effectively. Using this approach, the generation agent effectively adjusted the difficulty levels by varying the features in the problem specially to minimize the classification agent's performance, where the generation agent could make the problems 'harder' for the classification agent to learn (see Figure 4.26).

5.3 Phase 3: Three-Cornered Coevolution System

The Three-Cornered Coevolution System is the final research goal and the core principle of this research. The system is a new coevolution LCS where three different agents evolve to adapt with and drive the changes of the problem. There are a few distinctions between the new approach compared to the original framework proposed by Wilson described as follows.

5.3.1 Naming of the Agents

In Wilson's framework, the system was conceived as a pattern generator and two pattern recognizers. The pattern generator was referred as the Sender (S), while the pattern recognizers were referred as the Friend (F) and the Enemy (E). S's objective was to increase F's performance instead of E's performance. Therefore, F was provided with prior information regarding the problem. Most unlikely both of the pattern recognizers (i.e. F and E) used the same learning systems to address a similar problem. In this framework, the same learning system received different information to address the similar problem.

The Three-Cornered Coevolution System consisted of a generation agent and two classification agents. The generation agent was referred as the Sender (S) and the classification agents were referred as the Receiver (R) and the Interceptor (I). The classification agents were termed the Receiver and the Interceptor, because both of the classification agents competed with each other to learn a similar problem. However, the classification agents used different types of learning techniques (i.e. R learned through reinforcement learning, while I learned through guided learning). S's objective was to increase both R's and I's performance. In this system, two different learning systems received the same information to address a similar problem.

5.3.2 Input and Output Data

In Wilson's framework, F and E received variants of input images. However, F received an input image with prior information regarding its archetype. Next, both F and E (i.e. the pattern recognizers) attempted to transform the input image back into the archetype. The performance (i.e. the classification performance) of the pattern recognizers was computed as correct recognition against the images. In this framework, the learning systems were used to solve and address the image-based data for the classification tasks.

In the Three-Cornered Coevolution System both R and I (i.e. the classification agents) received an identical individual problem (i.e. artificial data) at a time and attempted to classify all instances in the dataset to the correct class. However, R and I used different types of learning techniques. R learned through reinforcement learning, while I learned through supervised learning. It was expected that most of the time I outperformed R, but not for all the problems. The performance (i.e. the classification performance) of the classification agents was computed as correct classification against a total number of instances. In the Three-Cornered Coevolution System, the learning systems were used to solve and address the artificial data for classification problems.

5.3.3 Coevolutionary Approach

The Three-Cornered Coevolution System offers several advantages over Wilson's framework as it provides a greater flexibility in problem creation and problem solving, especially with the work now focusing on the artificial data for classification. In this system, S (i.e. the generation agent) autonomously generated variants of problems at different levels of difficulty for classification, where S was able to adjust the difficulty levels effectively by varying the features in the problem that could either make the problems 'harder' or 'easier' based on the classification agents' abil-

ity to learn (see Figure 4.34). To formulate this approach, S first learned the 'meta-problem' by generating many problems, while both R and I (i.e. the classification agents) learned an individual problem at a time, through many instances of the problem. Next, based on R's and I's ability to learn (i.e. when the difference in performance between R and I was below a certain threshold), S coevolved to tune and adjust the problem's difficulty (i.e. either to increase or decrease the problem's difficulty).

In the Three-Cornered Coevolution System, I triggered the coevolutionary process within the system (i.e. at the lower-level of the system; the classifier), while S controlled the coevolutionary process within the system (i.e. at the top-level of the system's structure; the problem domain). However, in Wilson's framework, E (i.e. the pattern recognizer) controlled and triggered the coevolutionary process within the system. It was expected that at a certain degree, E's success would be necessary to improve S's and F's performance.

In Wilson's framework, as the problem's difficulty increased, the overall system implemented the Three-Cornered Coevolution in which each agent (S, F and E) attempted to evolve the best program to consistently achieve its objectives. In the Three-Cornered Coevolution System when S increased the problem's difficulty and generated various 'hard' problems for classification, the overall system successfully executed the Three-Cornered Coevolution. S consecutively increased the problem's difficulty, whereas R and I attempted to solve the classification tasks where the classification performance were varied (see Figure 4.36 and Figure 4.37). The findings showed that the classification performance was related to the type and percentage of the noise, the class imbalance ratio and the characteristics of the artificial data. It was found that I (i.e. the UCS system) was less sensitive to parameter tuning and more robust to noise and class imbalanced problem compared to R (i.e. the XCS system). In the XCS system, two parameters became critical for optimal performance: the learning rate and the GA triggering threshold θ_{GA} [89, 90]. Therefore, if both

R and I were configured appropriately (i.e. to self-adapt during learning [88, 89, 90]), both R and I were competitive to solve complex classification problems and evolve consistently in order to maximize the classification performance.

To this end, the version of *coevolution* in the Three-Cornered Coevolution System was implemented as *coadaptive evolution*. The system was conducted in such a way that the agents work cooperatively in a co-adaptive manner, where three different agents interacted with one another to adapt with one another's changes. In this way, S was able to identify features in the problem that altered R's and I's performance where a number of readable rules for each problem could be interpreted effectively when required. Consequently, the real cooperative coevolution process between the agents could be further improved, so that the rules in each communicating agent could be evolved in parallel within a sub-population. Later, the rules can be combined with the rules in the other sub-populations to create a comprehensive set of rules for addressing the problem.

5.3.4 Summary

The Three-Cornered Coevolution could be addressed in two different ways as discussed earlier, although this assumed that both of the methods were applicable to the system. In the Three-Cornered Coevolution System, the effect to trigger the coevolutionary process between R and I was known by S (i.e. the difference in performance between R and I). S was able to identify the features in the problem that altered the classification performance. Therefore, S was capable to autonomously generate various problems at the appropriate levels of difficulty whilst lowering human involvement, and produced variants of datasets for each problem to be learned by R and I respectively.

It is considered that the Three-Cornered Coevolution System has the following benefits. First, both of the problem domain and the solution can

be evolved autonomously. This method can reduce the effort needed for tuning and setting the parameter values in order to analyze the learner's ability. Secondly, a study on either how the learner's performance depends on its parameter values or how to choose parameter values that optimize the learner's performance can be performed through this method (see Section 4.3.1). Thirdly, an investigation on how different agents perform in the coevolutionary system can be conducted. Thus, different aspects of study including the problem domain and the implementation of the framework (e.g. on how the three different agents communicating within the system) can be investigated.

5.4 Summary and Way Forward

In this chapter, several issues relating the original Three-Cornered Coevolution Framework have been identified and addressed appropriately for each phase. Detailed discussions are provided to elaborate the above issues. It is noted that some aspects of the true implementation of cooperative coevolutionary framework are still open to further improvement. Thus, the next chapter will provide a summary of the research and describe future research directions arising from the discussions in this work.

Chapter 6

Conclusions

The overall goal of the work for this thesis is to develop the *Three-Cornered Coevolution System* for classification, where the system is capable of autonomously triggering (or activating) coevolution between the participating agents in the system when necessary, without human involvement. The system is a new coevolution system for generating various classification problems and testing the ability of a learning system autonomously. The goal has been achieved as the system was able to trigger the coevolutionary process automatically. The thesis demonstrated a set of new ideas and methodologies that use the Three-Cornered Coevolution System to modify the problem creation for classification where three different agents interact with each other in a coevolutionary manner.

6.1 Achieved Objectives

The thesis has achieved the following objectives:

1. The thesis introduced an *automated evolvable problem generator* that can create various problems for classification tasks; a new human-independent system for the problem creation of various classification problems (Phase 1). Two new different problem domains have been

created and can be manipulated autonomously (i.e. scalable image-based data, and artificial data). The results showed that the new system can evolve various problems for classification autonomously, which can create variants of image-based data, and artificial data, at different levels of difficulty whilst lowering human involvement.

2. The thesis introduced a new technique, the *Two-Cornered Coevolution System*, for addressing classification task(s). This was a new human-independent adjustable system which was capable of producing various classification problems and testing the learning algorithm (i.e. the classification agent) (Phase 2). The problem domain could be tuned autonomously based on the classification agent's ability to learn. The system was able to generate different types of problems for classification, whereas the problem's difficulty could be tuned and adjusted automatically based on the classification agent's ability to learn (i.e. either increasing or decreasing the problem's difficulty depending on the classification performance). The results suggested that the Two-Cornered Coevolution System had modified the traditional process for classification, as now both the problem and the solution evolve in parallel rather than independently. The detailed analysis shows that the evolved problems can easily be interpreted in order to understand the effect of features in the problem that altered the classification agent's performance. The thesis also demonstrated how two different LCS approaches (i.e. Michigan and Pittsburgh) can be implemented in the system for addressing the classification tasks. The adaption of Pittsburgh-style LCSs, A-PLUS, to an 'on-line' system was successful.

3. The thesis introduced a new technique, the *Three-Cornered Coevolution System*, for addressing classification task(s), which is a new co-evolution system (Phase 3). A set of new ideas and methodologies that use Three-Cornered Coevolution System for the first time to ad-

dress the classification tasks, has been established. This is a new coevolutionary LCS where three different agents evolve to adapt with and drive the changes of the problems. Here, a new classification agent is introduced to trigger the coevolutionary process within the system. Both the classification agents evolve to learn various classification problems, while the generation agent evolves to tune and adjust the problem's difficulty based on the classification agents' ability to learn. The classification agents use different types of learning techniques (i.e. reinforcement learning and supervised learning) to learn the classification tasks. The results showed that, as the classification agents learned and adapted with the changes of the problems, the coevolutionary process (i.e. coadaptive evolutionary) was implemented successfully without human intervention. Moreover, the overall system successfully executed the Three-Cornered Coevolution as the generation agent increased the problem's difficulty and generated various 'hard' problems for classification, where the classification agents' performances were varied.

6.2 Main Conclusions

This section presents the main conclusions for the three research objectives drawn from the three major contributions of the thesis.

6.2.1 Phase 1: An Evolvable Problem Generator

There are two important factors that need to be considered in designing the *automated evolvable problem generator* (Phase 1). First, a well suited problem domain to encompass the classification tasks is required. Secondly, an appropriate knowledge-based representation of the learner (i.e. the classification agent) in order to address the classification tasks effectively, is necessary.

In Phase 1, two different problem domains for classification tasks (i.e. image-based data and artificial data) that can be generated automatically have been established. A novel framework for an automated evolvable problem generator that can create various problems for classification tasks autonomously has been developed. The novelty here is that a new human-independent system for the creation of various classification problems has been established. Moreover, the created problem domain can be tuned and adjusted at various levels of difficulty. The image-based data utilized instances that were directly evolved, whereas for the real-valued artificial data, the underlying dataset properties were evolved, which then created the instances for training the classification agent. The latter abstracted approach was shown be more appropriate, as the underlying feature relationships that control the ease of learning within the problem domain were apparent.

All the agents were designed for an on-line system. The on-line system was required to establish the coevolutionary system where each agent attempted to adapt to each other's changes for an on-line manner. Here, LCSs have been applied to the agents based on the potential of their characteristics, such as interpretability, generalisation capability and variations in representation.

The main idea of the automatic classification problem generation has contributed towards a valuable source of library exemplars over the traditional method of costly human data specification. The system was able to autonomously generate various exemplars at different levels of difficulty for classification. The generated problems could be used to help in empirically testing the learning bounds of the learner (i.e. the classification agent) with less human intervention. The new method has a number of advantages over the traditional method.

1. The problem generator permitted the generation of a large set of examples (i.e. creating library of problems) autonomously.

2. The problem generator has a list of problem-specific parameters that could generate random instances and exemplars for each parameter value, which could cover a wide range of complexity factors in the problems.

3. The problem generator was able to autonomously adjust and tune features in the problem for the generation of various instances and exemplars to test the learner's ability.

4. The problem generator was able to handle problems of encoding and decoding for the generation of various instances and exemplars.

6.2.2 Phase 2: Two-Cornered Coevolution System

The *Two-Cornered Coevolution System* is concerned with the autonomous problem domain creation by the generation agent to autonomously generate various classification problems based on the classification agent's performance. First, the generation agent needs to determine the complexity factors in the problem, which can either 'increase' or 'decrease' the problem's difficulty. Secondly, the generation agent needs to autonomously tune the problem to be either 'harder' or 'easier' for the classification agent to learn based on the classification agent's performance.

In order for the generation agent to adjust the problem's difficulty the associated information in the problem such as the features of the problem should be able to be manipulated and adjusted. This suggests that the created problem domain should be flexible to able to be tuned and adjusted to make the problem either 'harder' or 'easier'. Findings strongly suggested that the method of using artificial data for classification was more relevant and suitable than the image-based data for several reasons, e.g. features in the problem can be tuned and adjusted either to increase or decrease the problem's difficulty (see Section 5.1). The problem generation of artificial data for classification led to a system that can tune and adjust the problem's difficulty based on the learner's ability.

Next, the generation agent needed to determine the complexity factors in the problem that could either 'increase' or 'decrease' the problem's difficulty. Here, Tabu Search is used in the generation agent to determine important features in the problem that alter the classification agent's performance. TS is used to find the best combination of features in the problem by varying the difficulty levels (i.e. either to tune the problem from 'hard' to 'harder' or 'easy' to 'easier' to learn). By implementing TS in the generation agent the difficulty levels can be tuned and adjusted based on the classification agent's performance. However, the generation agent that used TS alone in its component did not effectively minimize the classification agent's performance (i.e. to tune the problem from 'hard' to 'harder') as it became stuck in local optima.

Therefore, the Pittsburgh-style LCSs (i.e. A-PLUS system) is used in the generation agent to generate various classification problems, while Michigan-style LCSs (i.e. XCSR) is used in the classification agent to learn various classification problems. A-PLUS is applied to the generation agent to evolve its rules so that each new problem can be generated effectively at the appropriate levels of difficulty. Both of the systems worked well and showed promising results. The Pittsburgh-style LCSs, A-PLUS system, was successfully executed as an on-line system rather than the off-line system. The on-line system was required in order to establish the co-evolutionary system where each agent attempted to adapt to each other's changes in an on-line manner. Moreover, the on-line system had several advantages over the off-line system as discussed in Section 5.1.

The Two-Cornered Coevolution System is a baseline for coevolution LCSs where two different agents interact with each other to adapt with the changes of the problem. This is the first attempt to develop a new human-independent coevolutionary system for classification using LCSs. The system has modified the traditional process of creating problems for classification. Usually the learning algorithms are tested and adopted on a collection of problems (e.g. from a public repository) with certain con-

straints and data dependence. Here, both of the problem domain and the learner (i.e. the classification agent) evolved to address the classification tasks. Further, the problem domain was tuned autonomously based on the classification agent's ability to learn.

6.2.3 Phase 3: Three-Cornered Coevolution System

In the *Three-Cornered Coevolution System*, a new classification agent, namely the Interceptor was introduced to trigger the coevolutionary process within the system. Two different styles of learning techniques (i.e. reinforcement learning technique and supervised learning technique) were applied to the classification agents (i.e. R and I). I used the supervised learning technique (i.e. UCS system), whereas R used the reinforcement learning technique (i.e. XCS system), or vice versa.

The Three-Cornered Coevolution System is concerned with the coevolutionary process between three different agents within the system so that the coveolutionary process is triggered when necessary (i.e. when the difference in performance between R and I is below a specified threshold value). Both R and I evolve to learn various classification problems, while S coevolves to tune and adjust the problem's difficulty based on R's and I's ability to learn. I is introduced to the system in order to direct S to change the problem's difficulty.

In Phase 3, the difference in the classification agent's performance triggered the coevolutionary process, while the generation agent (i.e. S) controlled the coevolutionary process. The effect to trigger the coevolutionary process between R and I was known by S (i.e. when the difference in performance between R and I is below a specified threshold value). S was able to identify features in the problem that altered the classification performance. S was capable of autonomously generating various problems at the appropriate levels of difficulty (i.e. either to make the problem 'harder' or 'easier') whilst lowering human involvement, and produced

variants of datasets for each problem to be learned by R and I. In addition, the Three-Cornered Coevolution System allowed two different LCSs (i.e. XCS system or UCS system) to be automatically tested to identify which system performed better. The findings showed that the classification performance of the two systems was related to the type and the percentage of the noise, the class imbalance ratio and the characteristics of the artificial data. As the problem's difficulty increased (i.e. when S attempted to generate various 'hard' problems for classification), the classification performance between R and I varied. It was found that the UCS system was less sensitive to the parameters tuning and more robust to noise and class imbalance problem compared to XCS system. However, if both systems were configured appropriately (i.e. to self-adapt during learning), they were competitive to solve complex classification problems.

The Three-Cornered Coevolution System provides a greater flexibility in problem creation and problem solving. This system has the following benefits: 1) both the problem domain and the solution evolve autonomously reducing human intervention, 2) the coevolutionary process in the system is triggered automatically when necessary, and 3) a human-independent system for addressing the classification tasks over standard studies, where the evolutionary approach provides a greater utility and extensibility in problem creation and problem solving.

6.3 Future Work

The Three-Cornered Coevolution System is a novel coevolution LCS where three different agents interact with each other to drive the changes of the problems. The system is developed based on the theoretical model of the Three-Cornered Coevolution proposed by Wilson [124]. Several directions can be further investigated to extend this work. This section highlights potential future work motivated by the study of this thesis.

6.3.1 Problem Domain and Knowledge Representation

Appropriate problem domain and knowledge representation is crucial in designing the automated evolvable problem generator (Phase 1). In order to allow the participating agents to evolve more effectively in a dynamically changing environment, different aspects need to be considered in this direction.

Currently the generation agent is restricted to create image-based data or artificial data, which contain binary and real-valued data. However, in the real world applications, many datasets contain more than just a single type of data. For example, in the biomedical fields, the medical datasets often contain numeric data (e.g. test results), images data (e.g. x-rays), nominal data (e.g. smoker or non-smoker), and acoustic data (e.g. voice recording). Therefore, future work needs to explore this potential area in particular to design a method which integrates data from multiple sources in a single system. A study can be performed in order to investigate the agent's capability within the system either for generating or learning various types of data. In this case, a hybrid representation of numeric, nominal and many others can be introduced to the agent to improve the visualization, representation and condensation of its rules when applied to the real world problem.

6.3.2 EC

Research into enhancing the EC algorithms with other methods plays an important role in various fields and applications in providing a good quality of solutions. In Phase 2, a metaheuristic search method (i.e. Tabu Search) helped the EC algorithms (i.e. LCSs) to perform a search more effectively. Tabu Search has been successfully applied to the generation agent to enhance its ability to adjust the problem's difficulty, though there are still open issues that remain to be explored in this field.

In Phase 2, Tabu Search helped S (i.e. Pittsburgh-style LCSs, A-PLUS)

to search more effectively. However, TS has added extra computational burdens to the system as more parameters were introduced. The system became much more complex and required more time to be executed and evaluated. Thus, a method to determine and control the additional parameters in the system is required. In particular, a method to automatically determine the parameters during the search process (e.g. to control the parameters adaptively and systematically), should be undertaken for further investigation. Therefore, another research direction balancing the performance's improvement and the computational burden caused by the this technique can be explored. A better understanding of how to improve the cooperation between the ML technique and other methods efficiently can be gained.

6.3.3 Evaluation Measure

In order to evaluate the performance of any EC algorithm or ML technique, various evaluation schemes need to be used (e.g. robustness, scalability, and predictive accuracy). For all of the experiments, the learner's performance (i.e. the classification agent) was based on the classification performance (predictive accuracy). However, there are different types of performance measures that can be used to assess the learner's performance from different aspects.

Different types of evaluation schemes can be performed to assess different performance measures when dealing with different types of problems or different sets of problem configuration. For example, noise, missing values, or ambiguity can be added to the problem to test the learner robustness. The learner's scalability would be tested by varying the number of features and the number of instances in the datasets. Thus, the system can test the learner's efficiency in a particular case and comprehend the learner's behaviour for a specific constraint which can be further investigated on various performance measures.

6.3.4 Cooperative Coevolution System

A novel Three-Cornered Coevolution System where three different agents work coadaptively has been achieved. However, the version of the coevolutionary approach in the Three-Cornered Coevolution System was performed in such a way that the agents work cooperatively in a coadaptive evolution manner, where three different agents interacted with one another to adapt with their changes. Thus, the *cooperative coevolution* process between the agents can be implemented further, such that each classifier in all of the communicating agents can be evolved in a subpopulation and then combined with the rules in another subpopulation to create comprehensive rules for addressing the classification tasks.

In many cases, substantial prior-knowledge is required to boost the learning process and enhance the learning experience. The system can therefore be further enhanced to provide all of the agents with substantial prior-knowledge (e.g. previous learnt knowledge or shared knowledge), so that the cooperative coevolution between all participating agents can accelerate the agents' performance in which each agent attempts to evolve their best rules consistently.

6.4 Closing Remarks

The major task of the work for this thesis was to develop a new Three-Cornered Coevolution System for classification, which was a new coevolution LCS. The Three-Cornered Coevolution System was successfully developed and has contributed to the LCS field to provide a new system of implementation choice that can be used in a comparison of existing LCSs in the field. Even though the implementation of the system is different from Wilson's Framework, the system offers a great potential for autonomous learning and provides useful insight into coevolution learning over the standard studies of pattern recognition. Nevertheless, there

are still plenty of areas for improvement, which open new research directions in this domain. For instance, the Three-Cornered Coevolution System can be further enhanced to become a new cooperative coevolution system. The refinement of the system may lead to an autonomous and fully operational cooperative coevolution system in the field and provide another interesting research direction.

Bibliography

[1] J. Bacardit. *Pittsburgh Genetics-based Machine Learning in the Data Mining Era: Representations, Generalization, and Run-time*. PhD thesis, University of Ramon Llull, Spain, 2004.

[2] J. Bacardit, E. Bernado-Mansilla, and M. V. Butz. Learning Classifier Systems: Looking Back and Glimpsing Ahead. In *International Workshop Learning Classifier System (IWLCS'06)*, Seattle MA, USA, July 2006. Springer-Verlag Berlin Heidelberg.

[3] J. Bacardit and M. V. Butz. Data Mining in Learning Classifier Systems: Comparing XCS with GAssist. In *International Workshop Learning Classifier System (IWLCS'05)*. Springer, 2005.

[4] J. Bacardit and N. Krasnogor. Performance and Efficiency of Memetic Pittsburgh Learning Classifier Systems. *Evolutionary Computation*, 17(3):307–342, 2009.

[5] K. Bache and M. Lichman. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml, 2013.

[6] T. Bäck, D. B. Fogel, and Z. Michalewicz. *Evolutionary Computation 1: Basic Algorithms and Operations*. Institute of Physics Publishing, 2000.

[7] T. Bäck, D. B. Fogel, and Z. Michalewicz. *Evolutionary Computation 2: Advanced Algorithms and Operations*. Institute of Physics Publishing, 2000.

[8] S. Bandyopadhyay and S. K. Pal. *Classification and Learning Using Genetic Algorithms*. Springer, 2007.

[9] E. Bernadó-Mansilla and J. M. Garrell-Guiu. Accuracy-Based Learning Classifier Systems: Models, Analysis and Applications to Classification Tasks. *Evolutionary Computation*, 11(3):209–238, 2003.

[10] M. Birattari. *Tuning Metaheuristics - A Machine Learning Perspective*, volume 197 of *Studies in Computational Intelligence*. Springer, 2009.

[11] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, Natural Computing Series, 2006.

[12] L. Bull, E. Bernado-Mansilla, and J. Holmes. Learning Classifier Systems in Data Mining: An Introduction, Studies in Computational Intelligence (SCI). *Evolutionary Computation*, 125:1–15, 2008. Springer-Verlag.

[13] M. V. Butz. XCS Java 1.0: An Implementation of the XCS Classifier System in Java. Technical Report IlliGAL report 2000027, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, 117 Transportation Building, 104 S. Mathews Avenue Urbana, IL 61801, 2000.

[14] M. V. Butz. *Rule-Based Evolutionary Online Learning Systems: A Principal Approach to LCS Analysis and Design*. Springer, Berlin Heidelberg, 2006.

[15] M. V. Butz, P. L. Lanzi, and S. W. Wilson. Function Approximation with XCS: Hyperellipsoidal Conditions, Recursive Least Squares,

and Compaction. *IEEE Transactions on Evolutionary Computation*, 12(3):355–376, 2008. IEEE.

[16] S. Catalin, S. Ruxandra, M. Preuss, and D. Dumitrescu. Coevolution for Classification. Technical Report Technical Report, ISSN 1433-3325, Technical University of Dortmund, Dept. of Computer Science/LS 2, 44221 Dortmund, Germany, 2008.

[17] V. C. Chen. Evaluation of Bayes, ICA, PCA and SVM Methods for Classification. Technical Report RTO-MP-SET-080, Radar Division, US Naval Research Laboratory, 4555 Overlook Avenue, S.W. Washington DC 20375, USA, 2004.

[18] C. Congdon. *A Comparison of Genetic Algorithms and Other Machine Learning Systems on A Complex Classification Task from Common Disease Research*. PhD thesis, University of Michigan, Michigan, USA, 1995.

[19] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification, 2nd Edition*. John Wiley, 2001.

[20] R. P. W. Duin and E. Pekalska. Open Issues in Pattern Recognition. In *International Conference on Computer Recognition Systems (CORES'05)*, pages 27–42. Springer Verlag, 2005.

[21] A. E. Eiben and S. K. Smit. Parameter Tuning for Configuring and Analyzing Evolutionary Algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31, 2011.

[22] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing, First Edition*. Springer, Natural Computing Series, 2003.

[23] A. P. Engelbrecht. *Computational Intelligence, An Introduction, 2nd Edition*. John Wiley, 2005.

[24] L. J. Eshelman, R. Caruana, and J. D. Schaffer. Biases in the Crossover Landscape. In *International Conference on Genetic Algorithms (ICGA)*, pages 10–19. Morgan Kaufmann, 1989.

[25] A. Etham. *Introduction to Machine Learning*. MIT Press, 2010.

[26] D. B. Fogel and L. J. Fogel. An Introduction to Evolutionary Programming. In *Artificial Evolution*, pages 21–33, 1995.

[27] L. J. Fogel, P. J. Angeline, and D. B. Fogel. An Evolutionary Programming Approach to Include Self-adaptation on Finite State. In *Evolutionary Programming*, pages 355–365, 1995.

[28] L. J. Fogel and D. B. Fogel. A Preliminary Investigation on Extending Evolutionary Programming to Include Self-adaptation on Finite State. *Informatica*, 18(4), 1994.

[29] M. Gendreau. *A Tutorial On the Tabu Search*. Department of Computer Science, Montreal University, Canada, 2000.

[30] F. Glover. Tabu Search - Part i. *Informs Journal on Computing*, 1(3):190–206, 1989.

[31] F. Glover. Tabu Search - Part ii. *Informs Journal on Computing*, 2(1):4–32, 1990.

[32] F. Glover. Tabu Search: A Tutorial. *Interfaces*, 20(4):74–94, 1990.

[33] D. E. Goldberg. Genetic Algorithms and Rules Learning in Dynamic System Control. In *International Conference on Genetic Algorithms (ICGA)*, pages 8–15, 1985.

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

[35] D. E. Goldberg and K. Deb. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. *Foundations of Genetic Algorithms (FOGA)*, 1:69–93, 1991.

[36] S. Halim, R. H. C. Yap, and H. C. Lau. Visualization for Analyzing Trajectory-based Metaheuristic Search Algorithms. In *European Conference on Artificial Intelligence (ECAI)*, pages 703–704, 2006.

[37] A. Hamzeh and A. Rahmani. An Evolutionary Function Approximation Approach to Compute Prediction in XCSF. *Lecture Notes in Artificial Intelligence (LNAI)*, 3720:584–592, 2005. Springer-Verlag.

[38] D. S. Himmelstein, C. S. Greene, and J. H. Moore. Evolving Hard Problems: Generating Human Genetics Datasets with a Complex Etiology. *BioData Mining*, 4:21, 2011.

[39] T. K. Ho and M. Basu. Complexity Measure of Supervised Classification Problems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(3):289–300, 2002.

[40] J. H. Holland. Outline for a Logical Theory of Adaptive Systems. *ACM*, 9(3):297–314, 1962.

[41] J. H. Holland. Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, Artificial Intelligence. *University of Michigan Press*, pages 313–329, 1975.

[42] J. H. Holland. Complex Adaptive Systems. *A New Era In Computation*, 121(1):17–30, 1992. The MIT Press.

[43] J. H. Holland. What Is a Learning Classifier System? *Learning Classifier Systems : From Foundations to Applications*, page 3, 2000. Springer.

[44] J. H. Holland and J. S. Reitman. Cognitive Systems based on Adaptive Algorithms. *Pattern-directed Inference Systems*, pages 313–329, 1978.

[45] J. H. Holland and J. S. Reitmann. Cognitive System based on Adaptive Algorithms. *ACM SIGART Bulletin*, 63:49, 1977. ACM.

[46] J. H. Holmes, P. L. Lanzi, W. Stolzmann, and S. W. Wilson. Learning Classifier Systems: New Models, Successful Applications. *Information Processing Letters*, 82(1):23–30, 2002.

[47] D. Howard, L. Bull, and P. L. Lanzi. Continuous Actions in Continuous Space and Time using Self-adaptive Constructivism in Neural XCSF. In *Genetic and Evolutionary Computation Conference (GECCO'08)*, pages 21–24, Georgia, USA, July 2008.

[48] C. Huang and C. Sun. Parameter Adaptation within Co-adaptive Learning Classifier Systems. In *Genetic and Evolutionary Computation Conference (GECCO'04)*, pages 774–784, 2004.

[49] M. Iqbal, W. N. Browne, and M. Zhang. XCSR with Computed Continuous Action. In *Australiasian Joint Conference on Artificial Intelligence 2012*, pages 350–361. Springer, 2012.

[50] A. K. Jain, R. P. W. Duin, and J. Mao. Statistical Pattern Recognition: A Review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):4–35, 2000.

[51] P. Jedrzejowicz. Machine Learning and Agents. In *International Conference Agent and Multi-Agent Systems: Technologies and Applications (KES-AMSTA)*, pages 2–15, 2011.

[52] N. R. Jennings, K. P. Sycara, and M. Wooldridge. A Roadmap of Agent Research and Development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[53] K. A. De Jong, W. M. Spears, and D. F. Gordon. Using Genetic Algorithms for Concept Learning. *Machine Learning*, 13:161–188, 1993.

[54] F. Kharbat, L. Bull, and M. Odeh. Revisiting Genetic Selection in the XCS Learning Classifier System. In *IEEE Congress on Evolutionary Computation*, volume 3, pages 2061–2068, 2005.

[55] F. Klugl, R. Hatko, and M. V. Butz. Agent Learning Instead of Behavior Implementation for Simulations : A Case Study Using Classifier Systems. In *Multiagent System Technologies Conference (MATES)*, pages 111–122, 2008. Springer-Verlag.

[56] T. Kovacs. Strength or Accuracy? Fitness Calculation in Learning Classifier System. In *Learning Classifier Systems*, pages 143–160, 1999.

[57] T. Kovacs. Rule Fitness and Pathology in Learning Classifier Systems. *Evolutionary Computation*, 12(1):99–135, 2004. Massachusetts Institute of Technology.

[58] T. Kovacs. *Strength or Accuracy : Credit Assignment in Learning Classifier System*. PhD thesis, Bristol University, United Kingdom, 2004.

[59] J. R. Koza. Hierarchical Genetic Algorithms Operating on Populations of Computer Programs. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 768–774, 1989.

[60] J. R. Koza. Hierarchical Automatic Function Definition in Genetic Programming. In *Foundations of Genetic Algorithms (FOGA)*, pages 297–318, 1992.

[61] I. Kukenys, W. N. Browne, and M. Zhang. Confusion Matrices for Improving Performance of Feature Pattern Classifier Systems. In *Genetic and Evolutionary Computation Conference (GECCO'11)*, pages 181–182, 2011.

[62] I. Kukenys, W. N. Browne, and M. Zhang. Transparent, Online Image Pattern Classification using a Learning Classier System. In *European Event on Evolutionary Computation in Image Analysis and Signal Processing (EvoApplications'11)*. Springer-Verlag, 2011.

[63] P. Langley. Machine Learning for Intelligent Systems. In *Artificial Intelligence and Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI)*, pages 763–769, 1997.

[64] P. L. Lanzi. Extending XCSF Beyond Linear Approximation. In *Genetic and Evolutionary Computation Conference (GECCO'05)*, Georgia, USA, June 2005.

[65] P. L. Lanzi. Learning Classifier Systems: Then and Now. *Evolutionary Intelligence*, 1(1):63–82, 2008.

[66] P. L. Lanzi, D. Loiacono, S. W. Wilson, and D. E. Goldberg. XCS with Computable Prediction for the Learning of Boolean Functions. Technical report, Illinois Genetic Algorithms Laboratory, 2005.

[67] P. L. Lanzi, D. Loiacono, S. W. Wilson, and D. E. Goldberg. XCS with Computed Prediction in Multistep Environments. In *Congress on Evolutionary Computation (CEC'05)*, pages 2032–2039, 2005.

[68] P. L. Lanzi and R. L. Riolo. A Roadmap to the Last Decade of Learning Classifier System Research (From 1989 to 1999). In *Learning Classifier Systems*, pages 33–62, 1999. Springer-Verlag.

[69] P. L. Lanzi and S. W. Wilson. Optimal Classifier System Performance in Non-Markov Environment. Technical Report Artificial Intelligence and Robotics Project, Department Electronic and Information, Politecnic Milanolanzi, Italy, 1999.

[70] X. Llorà and J. M. Garrell-Guiu. Coevolving Different Knowledge Representations with Fine-grained Parallel Learning Classifier Systems. In *Genetic and Evolutionary Computation Conference (GECCO'02)*, pages 934–941, 2002.

[71] N. Macià, E. Bernadó-Mansilla, and A. Orriols-Puig. Preliminary Approach on Synthetic Data Sets Generation based on Class Sepa-

rability Measure. In *International Conference on Pattern Recognition (ICPR'08)*, pages 1–4. IEEE, 2008.

[72] N. Macià, A. Orriols-Puig, and E. Bernadó-Mansilla. Genetic-based Synthetic Data Sets for the Analysis of Classifiers Behavior. In *International Conference on Hybrid Intelligent Systems (HIS)*, pages 507–512. IEEE, 2008.

[73] N. Macià, A. Orriols-Puig, and E. Bernadó-Mansilla. Beyond Homemade Artificial Data Sets. *Hybrid Artificial Intelligence Systems*, 5572:605–612, 2009.

[74] P. Maes. Modeling Adaptive Autonomous Agents. *Artificial Life*, 1(1-2), 1994.

[75] P. Maes. Artificial Life Meets Entertainment: Lifelike Autonomous Agents. *Communications of the ACM*, 38(11):108–114, 1995.

[76] A. Mani and M. Kirley. CoXCS: A Coevolutionary Learning Classifier Based on Feature Space Partitioning. In *Australasian Joint Conference on Artificial Intelligence 2009*, pages 360–369, 2009.

[77] Z. Michalewicz and D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer-Verlag, 2000.

[78] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[79] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of Machine Learning*. MIT Press, 2012.

[80] Renan C. Moioli, Patrícia Amâncio Vargas, and Fernando J. Von Zuben. Analysing Learning Classifier Systems in Reactive and Non-reactive Robotic Tasks. In *Learning Classifier Systems (IWLCS 2007)*, volume 4998 of *Lecture Notes in Computer Science (LNCS)*, pages 286–305. Springer-Verlag, 2007.

[81] V. Nannen, S. K. Smit, and A. E. Eiben. Costs and Benefits of Tuning Parameters of Evolutionary Algorithms. In *Parallel Problem Solving from Nature (PPSN)*, pages 528–538, 2008.

[82] M. Negnevitsky. *Artificial Intelligence, A Guide to Intelligent Systems, Second Edition*. Addison Wesley, 2002.

[83] D. F. Nettleton, A. Orriols-Puig, and A. Fornells. A Study of the Effect of Different Types of Noise on the Precision of Supervised Learning Techniques. *Artificial Intelligence*, 33(4):275–306, 2010.

[84] A. Orriols-Puig and E. Bernadó-Mansilla. The Class Imbalance Problem in UCS Classifier System: A Preliminary Study. In *International Workshop Learning Classifier System (IWLCS'05)*, pages 161–180, 2005.

[85] A. Orriols-Puig and E. Bernadó-Mansilla. Class Imbalance Problem in UCS Classifier System: Fitness Adaptation. In *Genetic and Evolutionary Computation Conference (GECCO'05)*, pages 604–611, 2005.

[86] A. Orriols-Puig and E. Bernadó-Mansilla. A Further Look at UCS Classifier System. In *Genetic and Evolutionary Computation Conference (GECCO'06)*, pages 1–4, 2006.

[87] A. Orriols-Puig and E. Bernadó-Mansilla. Revisiting UCS: Description, Fitness Sharing, and Comparison with XCS. In *International Workshop Learning Classifier System (IWLCS'07)*, pages 96–116, 2007.

[88] A. Orriols-Puig and E. Bernadó-Mansilla. Mining Imbalanced Data with Learning Classifier Systems. In *Learning Classifier Systems in Data Mining*, pages 123–145. 2008.

[89] A. Orriols-Puig and E. Bernadó-Mansilla. Evolutionary Rule-based Systems for Imbalanced Data Sets. *Soft Computing*, 13(3):213–225, 2009.

[90] A. Orriols-Puig, E. Bernadó-Mansilla, D. E. Goldberg, K. Sastry, and P. L. Lanzi. Facetwise Analysis of XCS for Problems with Class Imbalances. *Evolutionary Computation*, 13(5):1093–1119, 2009.

[91] A. Orriols-Puig, J. Casillas, and E. Bernadó-Mansilla. Genetic-based Machine Learning Systems are Competitive for Pattern Recognition. *Evolutionary Intelligence*, 1:209–232, 2008.

[92] N. P. Padhy. *Artificial Intelligence and Intelligent Systems*. Oxford University Press, 2005.

[93] M. A. Potter and K. A. De Jong. A Cooperative Coevolutionary Approach to Function Optimization. In *Parallel Problem Solving from Nature (PPSN)*, pages 249–257, 1994.

[94] M. A. Potter and K. A. De Jong. Cooperative Coevolution: An Architecture for Evolving Coadapted Subcomponents. *Evolutionary Computation*, 8(1):1–29, 2000.

[95] M. A. Potter, K. A. De Jong, and J. J. Grefenstette. A Coevolutionary Approach to Learning Sequential Decision Rules. In *International Conference on Genetic Algorithms (ICGA)*, pages 366–372, 1995.

[96] I. Rechenberg. *Evolutions Strategie Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution*. PhD thesis, Stuttgart, Germany, 1973.

[97] Z. Ren and C. J. Anumba. Learning in Multi-agent Systems: A Case Study of Construction Claims Negotiation. *Advanced Engineering Informatics*, 16(4):265–275, 2002.

[98] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 3rd Edition*. Prentice Hall, 2009.

[99] B. K. Sarkar and S. S. Sana. A Hybrid Approach to Design Learning Classifier Systems. *Computers and Mathematics with Applications*, 58(1):65–73, 2009.

[100] J. Schürmann. *Pattern Classification: A Unified View of Statistical and Neural Approaches*. John Wiley, 1996.

[101] H. Schwefel. *Evolutions Strategie und Numerische Optimierung*. PhD thesis, Technische Universitat Berlin, Berlin, Germany, 1975.

[102] H. Schwefel and G. Rudolph. Contemporary Evolution Strategies. In *European Conference on Artificial Life (ECAL)*, pages 893–907, 1995.

[103] P. D. Scott and E. Wilkins. Evaluating Data Mining Procedures: Techniques for Generating Artificial Data Sets. *Information and Software Technology*, 41(9):579–587, 1999.

[104] S. Sen, M. Sekaran, and J. Hale. Learning to Coordinate without Sharing Information. In *Innovative Applications of Artificial Intelligence Conference (AAAI)*, pages 426–431, 1994.

[105] K. Shafi. *An Online and Adaptive Signature-based Approach for Intrusion Detection Using Learning Classifier Systems*. PhD thesis, School of Information Technology and Electrical Engineering, University of New South Wales, Australian Defence Force Academy, Australia, 2008.

[106] L. Shi, Y. Gao, L. Wu, and L. Shang. Clustering with XCS on Complex Structure Dataset. In *Australiasian Joint Conference on Artificial Intelligence 2008*, volume 5360, pages 489–499, 2008. Springer-Verlag.

[107] O. Sigaud and S. W. Wilson. Learning Classifier Systems: A Survey. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 11(11):1065–1078, 2007.

[108] S. Smith. *A Learning System Based on Genetic Algorithms*. PhD thesis, University of Pittsburgh, Pittsburgh, PA, USA, 1980.

[109] A. Stacey. An Investigation of Techniques for Improving the Performance of the Pittsburgh LCS. Technical Report UWELCSG04-005, Department Computer Science, University of Bath, United Kingdom, 2004.

[110] C. Stone and L. Bull. Foreign Exchange Trading using a Learning Classifier System. In *Learning Classifier Systems in Data Mining*, volume 125, pages 169–189. 2008. Springer-Verlag.

[111] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Massachusetts, USA, 1998.

[112] J. Tanimoto. Co-evolution Model of Networks and Strategy in a 2 x 2 Game Emerges Cooperation. In *Congress on Evolutionary Computation (CEC'08)*, pages 117–122, 2008.

[113] A. K. Tanwani and M. Farooq. Classification Potential vs Classification Accuracy: A Comprehensive Study of Evolutionary Algorithms with Biomedical Datasets. In *International Workshop Learning Classifier System (IWLCS'09)*. Springer, 2009.

[114] M. Troc and O. Unold. Self-Adaptation of Parameters in A Learning Classifier System Ensemble Machine. *Applied Mathematics and Computer Science*, 20(1):157–174, 2010.

[115] O. Unold. Self-Adaptive Learning Classifier System. *Journal of Circuits, Systems, and Computers*, 19(1):275–296, 2010.

[116] R. J. Urbanowicz and J. H. Moore. Learning Classifier Systems: A Complete Introduction, Review, and Roadmap. *Journal of Artificial Evolution and Applications*, pages 1–25, 2009.

[117] Y. Wen and H. Xu. A Cooperative Coevolution-based Pittsburgh Learning Classifier System embedded with Memtic feature Selection. In *Congress on Evolutionary Computation (CEC'11)*, pages 2415–2422, 2011.

[118] R. P. Wiegand, W. C. Liles, and K. A. De Jong. Modeling Variation in Cooperative Coevolution Using Evolutionary Game Theory. In *Workshop on Foundations of Genetic Algorithms (FOGA)*, pages 203–220, 2002.

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

[120] S. W. Wilson. Get Real! XCS with Continuous-valued Inputs. In *International Workshop Learning Classifier System (IWLCS'99)*, volume 1813 of *Lecture Notes in Computer Science (LNCS)*, pages 209–219. IEEE, Springer Berlin Heidelberg, 2000.

[121] S. W. Wilson. Compact Rulesets from XCSI. In *International Workshop in Learning Classifier Systems (IWLCS'01)*, pages 197–210, San Francisco, California, USA, July 2001.

[122] S. W. Wilson. Mining Oblique Data with XCS. *Lecture Notes in Computer Science (LNCS).*, 1996:158–174, 2001. Springer-Verlag.

[123] S. W. Wilson. Classifiers that Approximate Functions. *Journal of Natural Computing*, pages 211–234, 2002. Springer-Verlag.

[124] S. W. Wilson. Coevolution of Pattern Generators and Recognizers. In *International Workshop Learning Classifier System, IWLCS 2009*, pages 2605–2608. ACM, New York, 2009.

[125] J. Yang, H. Xu, and P. Jia. Effective Search for Pittsburgh Learning Classifier Systems via Estimation of Distribution Algorithms. *Information Sciences*, 2012(198):100–117, 2012.

[126] Z. V. Zatuchna. The Classification of Maze Problems in the Framework of Learning Classifier Systems Research. Technical Report Technical Report CMP-C04-02, School of Computing Sciences, University of East Anglia, UK, 2004.

[127] J. Zhang, Z. Zhan, Y. Lin, N. Chen, Y. Gong, J. Zhong, H. S. Chung, Y. Li, and Y. Shi. Evolutionary Computation Meets Machine Learning: A Survey. *IEEE Computational Intelligence Magazine*, 6(4):68–75, 2011.

[128] F. Zhu and S. U. Guan. Enhanced Cooperative Co-evolution Genetic Algorithm for Rule-based Pattern Classification. In *Hybrid Artificial Intelligence Systems (HAIS)*, pages 113–123, 2008.