Pattern and Function Transfer for Learning Classifier Systems

by

Trung Bao Nguyen

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
2021

Abstract

A key goal of Artificial Intelligence (AI) is to replicate different aspects of biological intelligence. Human intelligence can accumulate progressively complicated knowledge by reusing simpler concepts/tasks to represent more complex concepts and solve more difficult tasks. Also, humans and animals with biological intelligence have the autonomy that helps sustain them over a long period.

Young humans need a long period to obtain simple concepts and master basic skills. However, these learnt basic concepts and skills are important to construct foundation knowledge, which is highly reusable and thereby efficiently exploited to learn new knowledge. By relating unseen tasks to learnt knowledge, humans can learn new knowledge or solve new problems effectively. Thus, AI researchers aim to mimic human performance with the same ability to reuse learnt knowledge when solving a novel task in a continual manner.

Initial attempts to implement this knowledge-transfer ability have been through *layered learning* and *multitask learning*. Layered learning aims to learn a complex target task by learning a sequence of easier tasks that provide supportive knowledge prior to learning the target task. This learning paradigm requires human knowledge that may be biased, costly, or not available in a particular domain. Multitask learning generally uses multiple related tasks with individual goals to be learnt together with the hope that they can provide externally supportive signals to each other. However, multitask learning is commonly applied to optimisation tasks that are required to start simultaneously.

In this thesis, using the transfer of building blocks of learnt knowledge is of interest to solve complex problems. A complex problem is one where the solution cannot be simply enumerated in the time and computation available, often because there are multiple interacting patterns of input features or high dimensions in the data. A strategy for solving complex problems is to discover the high-level patterns in the data. The high-level patterns are ones with complex combinations of original input features (the underlying building blocks) to describe the desired output. However, as the complexity of building blocks grows along with the problem complexity, the size of the search space for solutions and the optimal building blocks also increases in complexity. This poses a challenge in discovering optimal building blocks.

Learning Classifier Systems (LCSs) are evolutionary rule-based algorithms inspired by cognitive science. LCSs are of interest as their niching nature enables solving problems heterogeneously and learning them progressively from simpler subproblems to more complex (sub)problems. LCSs also encourage transferring subproblem building blocks among tasks. Recent work has extended LCSs with various flexible representations. Among them, Code Fragments (CFs), Genetic Programming (GP)-like trees, are a rich form that can encode complex patterns in a small and concise format. CF-based LCSs are particularly suitable for addressing complex problems. For example, XCSCF*, which was based on Wilson's XCS (an accuracy-based online learning LCS), can learn a generalised solution to the n-bit Multiplexer problem. The above techniques provided remarkable improvements to the scalability of CF-based LCSs. However, there are certain limits in such systems compared with human intelligence, such as their limited autonomy, e.g. the requirement of an appropriate learning order (e.g. layered learning) to enable learning progress. Humans can learn multiple tasks in a parallel ad hoc manner, whereas AI cannot do this autonomously.

The proposed thesis is that systems of parallel learning agents can solve multiple problems concurrently enabling multitask learning and eventually the ability to learn continually. Here, each agent is a CF-based XCS where the problems are Boolean in nature to aid interpretability. The overall goal of this thesis is to develop novel CF-based XCSs that enable learning continually with the least human support.

The contributions of this thesis are three specific systems that provide a pathway to continual learning. By reducing the requirements of human guidance without degrading the learning performance. (1) The *evolution of CFs* is nested and interactive with the evolution of rules. The fitness of CFs called CF-fitness is introduced to guide this process. The evolution of CFs enables growing the complexity of CFs without a depth limit to address hierarchical features. The system is the first XCS with CFs in rule conditions that can learn complex problems that used to be intractable without transfer learning. The introduction of CF evolution allows appropriate latent building blocks that address subproblems to be grouped together and flexibly reused. (2) A new system of multitask learning is developed based on the estimation of the relatedness among tasks. A new dynamic parameter helps automate feature transfer among multiple tasks, which enables improved learning performance in supportive tasks and reduced negative influence between unrelated tasks. (3) A system of parallel learning agents, where each is an XCS with CF-actions, is developed to remove the requirement of a human-biased learning order. The system can provide a clear learning order and a highly interpretable network of knowledge. This network of knowledge enables the system to accumulate knowledge hierarchically and focus on only the novel aspects of any new task.

The research work has shown that CF-based LCSs can solve hierarchical and large-scale problems autonomously without (extensive) human guidance. The learnt knowledge represented by CFs is highly interpretable. This work is also a foundation for the systems that can learn continu-

ally. Ultimately, this thesis is a step towards general learners and problem solvers.

Publications

- **T. B. Nguyen**, W. N. Browne, and M. Zhang, ″Online feature-generation of code fragments for XCS to guide feature construction,″ in 2019 IEEE Congress on Evolutionary Computation (CEC). IEEE, 2019, pp. 3308–3315.
- **T. B. Nguyen**, W. N. Browne, and M. Zhang, ″Improvement of code fragment fitness to guide feature construction in XCS,″ in Proceedings of the Genetic and Evolutionary Computation Conference, ser. GECCO 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 428–436.
- **T. B. Nguyen**, W. N. Browne, and M. Zhang. ″Relatedness measures to aid the transfer of building blocks among multiple tasks,″ in Proceedings of the Genetic and Evolutionary Computation Conference. New York, NY, USA: Association for Computing Machinery, 2020.
- **T. B. Nguyen**, W. N. Browne, and M. Zhang. ″Constructing complexity-efficient features in XCS with tree-based rule conditions,″ in 2021 IEEE Congress on Evolutionary Computation (CEC). IEEE, 2021, accepted.
- **I. Alvarez, T. B. Nguyen**, W. N. Browne, and M. Zhang, ″A Layered Learning Approach to Scaling in Learning Classifier Systems for Boolean Problems,″ submitted to Evolutionary Computation (MIT Press). 2020, under revision.

- **T. B. Nguyen**, W. N. Browne, and M. Zhang, "ConCS: A Continual Classifier System for Continual Learning of Multiple Boolean Problems," submitted to IEEE Transactions on Evolutionary Computation. 2020.

Acknowledgements

I would like to express my gratitude to those who have helped me in numerous ways to complete this thesis.

First and foremost, I am thankful to my supervisors, Associate Professor Will Brown and Professor Mengjie Zhang, for their encouragement, and patient guidance throughout this research. During my PhD journey, they steered me in the right directions and were always available for any discussion or request at any time. I am incredibly grateful to them for revising my papers and thesis within a very tight schedule. They gave me very detailed and constructive comments. Their corrections and suggestions contributed to the quality of my publications.

I have been amazingly fortunate to have Associate Professor Will Browne as my primary supervisor. His patience, openness, and positive mindset at different stages of my research had helped me focused my ideas and accomplished this research. I also learnt a lot of research, professional and supervisory skills from him over this journey, which will help me in my future career in academia. Importantly, Will has been a role model of being positive and kind to everyone that gradually shapes me into a better person. Thank you for all you have done, Will!

My gratitude also goes to my co-supervisor, Professor Mengjie Zhang. His expertise, insightful comments, and quick response assisted me greatly in this research. I was strongly inspired by his enthusiasm, motivation in

working, and kindness in leadership. I will not forget the way he supported me.

Special thanks to other professors, lecturers, administrators from School of Engineering and Computer Science for their kind assistance, and friendly environment. A special note must be made to my AM409+ friends (Abubakar, Bach, Tony, Yi, Abhi, and Roshni) and other ECRG members for their pleasant company. Sharing and chatting with them always motivated me to improve myself.

This research was funded and strongly supported by Victoria University Doctoral Research Scholarship, and for which I am very grateful.

I am indebted to my Mom, my parents in law and my extended family for their love, understanding, and continuous encouragement in all my pursuits.

I wish to thank my wife, An, whose support for me has been boundless. From the bottom of my heart, I am grateful for her encouragement that helps me overcome all the stress in my PhD time. She also sacrificed her time to helped me through the hardest moments of my study. Her study in Auckland has been compromised because of staying here in Wellington with me. As a logical person, her voluntary sacrifices have taught me a true lesson of giving in a relationship.

Last but not least, I would like to thank everyone who had contributed to this research in any way either directly or indirectly. Their support, assistance and contribution are much appreciated.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Scope and Context

Human intelligence is the best-known example of general intelligence. The long-term goal of Artificial Intelligence (AI) is to replicate different aspects of biological intelligence, especially human intelligence:

> "The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990 [76])

An agent of biological intelligence can accumulate increasingly complicated knowledge by reusing simpler concepts/tasks to represent more complex concepts and solve more difficult tasks. For example, children take months to learn basic skills like grasping [42]. Still, they can reuse them to develop variants of basic skills quickly [122].

Machine Learning (ML) is an aspect of AI that imitates humans' learning abilities through experience, e.g. usually training instances in training stages, without being explicitly programmed [13]. Unlike biological intelligence, traditional ML algorithms commonly serve only one specific task. An ML system usually learns its task from scratch without the ability to reuse useful knowledge learnt from other tasks. This necessitates many trials and associated errors for a system to learn a hierarchical and/or large-scale problem[1]. This limitation also results in inefficient consumption of time and computation resource for large-scale and hierarchical problems.

Human beings are altricial so take years to correctly learn to recognize patterns of signals from our senses, e.g. sounds and images, and to understand highly abstract concepts. However, we eventually master the ability to tackle unseen problems and situations by taking advantage of knowledge learnt in the past from other similar or related problems. Specifically, humans can relate new problems to learnt knowledge and solve them with very limited numbers of trials by reusing experienced knowledge. Generally speaking, AI/ML researchers desire to build intelligent systems with human-like abilities. In this case, ML researchers are attempting to mimic human performance with the same ability to reuse/transfer learnt knowledge when solving a novel task. In general, this ability is an important aspect of biological intelligence. For example, it benefits autonomous agents (including biological intelligence) in changing behaviours flexibly among tasks and environments [117]. Because of its efficiency in the learning process of humans, it is a desirable feature that ML researchers are still investigating how best to integrate into machines.

Knowledge transfer was implemented in transfer learning [101]. This learning paradigm concerns the ability to reuse knowledge from a source task to improve the performance in a target task. There are other learning paradigms related to transfer learning as they take advantage of knowledge transferred from other tasks to bootstrap the performance on a target task. Layered learning aims to learn a complex target task by learning an ordered sequence of easier tasks that provide supportive or even

[1]A hierarchical problem is a combination of multiple levels of simpler patterns

prerequisite knowledge prior to learning the target task [119]. Layered learning is highly correlated with continual learning, which is a continual process where learning occurs over time [125]. Continual learning closely relates to how humans learn as a continual-learning agent's experiences occur sequentially without a final task. The learning process is incremental and hierarchical. Knowledge from one task can be reused later to solve a completely different task or set a basis for hierarchical behaviours/concepts.

Conversely, multitask learning generally uses multiple related tasks to be learnt together with the hope that they can provide externally supportive signals to each other [29]. Unlike transfer learning, tasks in multitask learning have no priority. An early study of multitask learning applied it on multiple predictive supervised learning [28].

In this thesis, using the transfer of building blocks is of interest to solve complex problems, which create large search spaces of solutions. The search space can be large when the target problem is hierarchical and/or large in scale because the solutions are either complex or appear in a high-dimensional search space.

Solving complex problems requires the ability to discover the high-level underlying patterns in the data. The high-level patterns require high-level building blocks to reproduce the output pattern. In complex problems, high-level building blocks can be made of lower-level building blocks (simple combinations of environment data), so can be are built in a bottom-up manner. However, as the building block level grows along with the problem complexity or scale, the size of the solution search space also increases rapidly. This poses a challenge in finding optimal building blocks and reducing the interpretability of learnt solutions. For example, one of the possible negative influences by increasing the complexity limit of building blocks is the increasing presence of bloat in tree-based programs [74], where inefficient information is encoded into a solution. Bloat could lead to local optima and prevent the interpretability.

Evolutionary Computation (EC) is a branch of AI that mimics the Darwinian evolution principle to solve problems [54]. Moreover, artificial evolution techniques are believed to be capable of designing more complex behaviours than ones designed by hand (Chapter 8, [107]). An EC algorithm generally solves a problem by evolving a population of solution candidates. The principles Darwinian evolution theories, such as natural selection and reproduction, are applied in the selection, mutation and crossover operators of EC algorithms. These operators iteratively drive the population to discover fitter candidates regarding a pre-defined fitness evaluation.

EC techniques represent individuals through building blocks, such as *schema* [54]. This encourages the reuse of learned knowledge across problems. Multitask learning has been adopted in EC to mainly solve optimisation tasks [47, 48]. However, this thesis is focused on classification tasks because this category offers latent patterns that can be identified in a reusable form. Furthermore, optimisation problems often do not contain separable patterns to be reused, while regression problems are inherently included in this thesis as subproblems in classification tasks.

Learning Classifier Systems (LCSs) are the EC rule-based algorithms of interest as this group of techniques has the unique niching property [129, 130]. This feature helps divide a problem into simpler subproblems, which are subsets of instances that share some common characteristics, to be solved efficiently [54, 129]. This is an efficient approach to decompose hierarchical problems. Additionally, LCSs have pressures to suppress bloat implicitly through the niche-based fitness sharing [27].

An LCS usually learns using evolutionary operators to solve a specific task, which is often a reinforcement learning problem [121]. The rules in traditional LCSs are in the form of "if *condition* then *action*". Among various implemented of LCSs, XCSs are a popularly adopted branch of LCSs that uses accuracy-based fitness and simplifies the concept of LCSs for ML and robotics problems [141].

Recent work has integrated various representations into classifier conditions of LCSs. Among those representations, Code Fragments (CFs) [63], which are GP-like tree-based programs, are a rich and flexible form that can represent complex programs. Using CFs enables rules to encode complex patterns in a small and concise format. Thus, CF-based LCSs are particularly suitable for addressing complex problems. Also, LCSs' ability to divide-and-conquer is useful in solving complex problems. A learning system with this ability can generalise by constructing high-level building blocks in a bottom-up manner from low-level building blocks.

Iqbal et al. introduced XCSCFC as the first XCS that uses CFs in rule conditions to target 135-bit Multiplexer problem[2] through a layered learning approach [59]. Because XCSCFC grows at most two layers to existing CFs in a learning stage, the learning system is believed to construct CFs with controllable amounts of bloats. XCSCF* used CFs in rule actions and transferred CF-based ruleset functions to address n-bit Multiplexer problem [6, 7]. Compared with XCSCFC, XCSCF* can transfer knowledge from one problem domain to other related ones. This was achieved by providing supporting simpler problems with a strict order, i.e. the curriculum, through layered learning. This learning order (curricular order) needs to be done manually by humans to build up the complexity of supportive knowledge efficiently before learning the most complex logic. However, the order requires prerequisite knowledge on the target problem, which is not always available. XCSCF* can learn a generalised solution to the Multiplexer problems, which can solve n-bit Multiplexer problem (any scale). Discovering the generalised patterns of a problem domain is a major desirable property of ML [95].

[2]Multiplexer problems are Boolean problems describing Multiplexer circuits in electronics. They are interesting as they have variable interactions and are highly non-linear.

The above techniques provided remarkable improvements to the scalability of CF-based XCSs on the tested problem domains. However, there are certain limitations in such algorithms compared with human intelligence, such as their limited autonomy. This is against the self-sufficiency, one of the design principles of AI systems that concerns the ability of the AI system to sustain itself with full autonomy over time [107].

1.2 Motivations

The Need for External Guidance for Layered Learning and Transfer Learning

Knowledge transfer in layered learning can bootstrap the learning performance on later tasks in the curriculum. However, layered learning poses a constraint when it requires external human guidance. The human guidance is generally unwanted as it reduces the flexibility of the layered learning and is sometimes unavailable. Additionally, the transferring criteria require prerequisite knowledge about the target and the source tasks, such as statistical information in domain adaptation [99, 128]. The statistical information is not available in the case of online learning as this paradigm only allows accessing the streaming data. For instance, XCSCFC, a layered learning approach, has been shown to be efficient in specific cases where the reusing criteria are tailored for the problems in the learning layers [63]. Another requirement for these learning paradigms is the learning order, especially for layered learning [6]. The learning performance may be disrupted (i.e. slowly or not converged) without a proper learning curriculum/order. In summary, the autonomy of the learning systems in these two learning paradigms is far from human intelligence.

Alvarez et al. managed to scale the problems by layered learning in XCSCF* [6]. XCSCF* avoids the intractable size of the search space of solutions by dividing the target problem into smaller problems, where each covers an aspect of the target problem. The main limitation of XCSCF* is the need for human intervention in the guidance for the curricular order as well as prior knowledge of how the problem should be decomposed. The decomposition task is done by a human instead of a machine in an automatic manner. With an appropriate curriculum, Alvarez's work suggested that CF-based XCSs can solve large-scale and hierarchical problems given an empirical curriculum for transferring and accumulating knowledge. However, curricula are generally not available in unseen tasks.

All existing CF-based XCSs rely on layered learning to scale up the CF-features and solve large-scale/complex problems. However, the autonomy is also a goal of AI for the self-sufficiency of general intelligence [107].

Large Volume of Available Knowledge

Learning complex problems can be efficiently addressed by constructing high-level features. However, the search space of constructing high-level features is large and potentially intractable, such as hierarchical problems [25], Even-parity, or Carry-one problems [63]. It is even more problematic in the case of multitask learning, where multiple unrelated problems are solved in parallel. The volume of learnt knowledge shared among systems grows much faster in size and diversity than in single-task learning.

The learning process of one task may generate irrelevant knowledge in addition to only applicable and especially generalised knowledge for other tasks in transfer learning, layered learning, and multitask learning. If irrelevant knowledge is transferred, it will create extra noisy information that reduces the learning performance on the target task instead of improving it. Thus, it is necessary to have a gate to control the transferring criteria. Nevertheless, the general transferring criteria are not available beforehand in online learning since there is no prerequisite information on the tasks. Also, along with level/depth growth in features, the transferring criteria should be adapted appropriately.

In another viewpoint, connections among neurons in the form of axon-synapse-dendrite are the key factor for the biological intelligence as they help neurons communicate [106]. The neural communication enables human intelligence to link relevant knowledge, e.g. concepts and actions, given perceived signal(s) [43, 68]. This is a critical advantage of human brains compared with existing AI systems because, for an unending learning process, there is much more irrelevant knowledge than the relevant one in all accumulated knowledge. Consequently, if a continual learner has to do reasoning given a large amount of irrelevant knowledge, its performance could be unresponsive or not competitive. Therefore, an intelligent system should be able to provide relevant connections among its learnt knowledge.

Both these issues lead to the motivation to achieve intelligent systems with more flexibility (less constraints). The flexibility enables CF-based XCSs to be generalised to address a broader range of problems. Nevertheless, this objective have not been achieved since the existing CF-based XCSs rely on layered learning.

1.3 Thesis Statement

Overall, this thesis is that systems of parallel learning systems, where each system is a CF-based XCS, can solve multiple Boolean problems concurrently and, therefore, enable both multitask learning and continual learning for CF-based XCSs.

The system will be capable of constructing CFs that capture underlying patterns of the data in complex problems. Capturing the underlying patterns in some CFs means that the systems using such CFs can abstract away low-level details of the data patterns. Therefore, the learning system can address all aspects of the problem with a minimum number of rules. Searching the underlying patterns of a hierarchical problem using tree-based programs (CFs) is possible [3] but also challenging. Without customised human guidance, the search space of the underlying patterns for hierarchical problems is usually intractable.

For learning multiple problems in parallel, a challenge is to find appropriate relationships among problems. This is because no knowledge about the target problems is available in advance. These relationships will provide connections among knowledge produced by learning the problems so that learning new problems can refer to only relevant knowledge. This reduces the search space and makes learning tractable.

1.4 Research Goals

The overall goal of this thesis is to improve the learning capabilities of evolutionary machine learning without the need of layered learning and, thus, to enable flexibility and autonomy in using these systems to address (multiple) classification problems. This includes introducing multitask learning and continual learning to existing CF-based XCSs. This goal is divided into three sub-goals, as follows:

1. To provide XCS using CF-conditions with the ability to discover useful high-level CFs without a human-customised sequence of learning stages and transferring criteria (layered learning). By being useful, CFs are supposed to capture the underlying data patterns. High-level CFs are only required to describe complex patterns, which are not efficiently described by low-level CFs or the traditional ternary representation $(0, 1, \#)$ in standard XCS [141]. The following research objective has been established to achieve this sub-goal:

    - To develop a novel evolution of CFs that is nested within the rule evolution of XCS. These two evolutionary processes interact and support each other. The evolution of CFs is to start from the lowest-level CFs and grow higher-level CFs. This principle is based on the assumption that the combination of the most useful CFs usually has a higher chance of creating more useful CFs compared with combining random CFs. The evolution of CFs will retain the niching property of XCS.

- To introduce a subset of CFs that captures the data patterns the best among all generated CFs as this subset has the most relevant available information about an online-learning task. Through this subset, the new CF-based XCS is expected to dynamically harvest the most up-to-date information of the problem in the form of tree patterns at any learning stage. This subset is the central base to grow higher-level tree patterns. Furthermore, this opens the possibilities of collaborating the learning processes of multiple online-learning tasks without prerequisite knowledge about the tasks, which was done in the second sub-goal.

  Both of these two sub-goals rely on the introduction of a measure for estimating the applicability of CFs. This quantity will be encoded in a new parameter called CF-fitness. Also, these goals require establishing a sub-group of CFs containing CFs with the highest CF-fitness.

- To optimise the structural efficiency of the constructed CF-based patterns. The structural efficiency is based on the generality of CFs, which enables classifiers using them to match accurately with the most instances, and the structural complexity of CFs. This efficiency eliminates bloat, which can prevent the learning system from advancing the learning process, and improves the interpretability of the extracted solutions. Also, when feature reusabilities stack many times, this efficiency will result in much more readable solutions.

2. To develop a novel system of multiple XCS-based agents for multi-task learning. Each agent is a variant of the system introduced in the preceding sub-goal. The following research objectives have been designated to achieve this sub-goal:

    - To develop a novel method of estimating the relatedness (relationship) among learning tasks dynamically. The measurement of this quantity is to be encoded in a parameter called "relatedness". This parameter represents the relationships/connections among target tasks. The measured relationships among tasks are expected to provide relevant connections among knowledge, which enable appropriate reusabilities of learnt knowledge.

    - To utilise the relationships among tasks to enable the ability to automatically transfer knowledge among related tasks/systems. This ability aims to allow the system to not only improve the performance of agents when the tasks are mutually supportive but also reduces the harmful transfer among unrelated tasks.

3. To develop a system of multiple XCS classifiers that is designed with the ability to accumulate progressively more complex knowledge through learning multiple problems with minimal external guidance. This system learns multiple tasks continually and simultaneously. This system is composed of multiple agents that interact indirectly with each other through a dynamic knowledge pool. The following research objectives have been established to achieve this sub-goal:

    - To reduce the search space of rule actions (CFs in this case) by dividing it based on the input and output types of available functions. This property will enable generating CFs that are verifiable.

    - To bootstrap the system with a general loop skill that can integrate a given sub-function to iteratively process a string input.

This skill is an axiomatic building block that is designed to scan the whole input signal or a latent signal for patterns defined by a sub-function. This skill will be essential for solving hierarchical Boolean problems as it will enable the intelligent system to shift a kernel (a sub-function) throughout the (latent) signal to scan for patterns.

- To develop a simple stochastic method of selecting the task to work on iteratively. This process is to allocate the computing resource with the most promising tasks.

By targeting the ability to construct complex useful building blocks (CFs) from simple ones without or with little human interventions, The above goals include the ability to harness compositionality to rapidly acquire and generalise knowledge to new tasks. Thus, this thesis will be an intermediate step towards human-like machine intelligence using the LCS framework [77].

Interpretable and explainable AI are an important issue as AI is used in real-world applications [91]. Although this is not a direct objective of this work, LCS has the nature to encapsulate knowledge in a transparent and interpretable manner. The ability to access learnt knowledge in the learnt curricula as a knowledge hierarchy will enhance interpretability.

The developed systems will be evaluated using various hierarchical and large-scale Boolean problems, as well as real-world datasets. The hierarchical problems provide high-level patterns while the large-scale problems require the learning system to choose the right building blocks in large search spaces. Real-world datasets will be used for the initial investigations of the potential applications of the developed systems. These systems are compared with standard XCS [141] and existing CF-based XCSs.

1.5 Major Contributions

CF-based XCSs have more flexibility yet are still able to learn large-scale and hierarchical problems. The outputs of this thesis include the following major contributions to the fields of Evolutionary ML (EML), especially XCSs using tree-based programs:

1. An XCS with CF-conditions, named XOF, was created that can learn large-scale and hierarchical problems without layered learning. XOF can learn such problems using less learning experience than XCS. The new learning operations do not necessarily cause the system to run slowly, even compared with XCS, which uses only ternary representation. Parts of this contribution have been published in:

   T. B. Nguyen, W. N. Browne, and M. Zhang, "Online feature-generation of code fragments for XCS to guide feature construction," in 2019 IEEE Congress on Evolutionary Computation (CEC). IEEE, 2019, pp. 3308–3315.

   T. B. Nguyen, W. N. Browne, and M. Zhang, "Improvement of code fragment fitness to guide feature construction in XCS," in Proceedings of the Genetic and Evolutionary Computation Conference, ser. GECCO 19. Association for Computing Machinery, 2019, p. 428–436.

2. The structural efficiency in the introduced XOF has been substantially improved by introducing the niching method to the evolution of CFs in XOF. The efficiency allows XOF to construct CFs that encapsulate the underlying data patterns efficiently. The niching method also enables XOF to conserve the niching property in the whole system, which was the strength of XCS previously.

   Parts of this contribution have been reported in:

   T. B. Nguyen, W. N. Browne, and M. Zhang. "Constructing complexity-efficient features in XCS with tree-based rule conditions," in 2021

IEEE Congress on Evolutionary Computation (CEC). IEEE, 2021, accepted.

3. The ability to automatically estimate the appropriate probabilities of building block (feature) sharing among Boolean classification problems is introduced in a multitask learning system using multiple XOFs (mXOF). This system can yield relatedness/connections among tasks based on the overlapped features. The relatedness is dynamic and enables mXOF to automatically adjust feature sharing among tasks in multitask (online) learning.

   Parts of this contribution have been published in:

   T. B. Nguyen, W. N. Browne, and M. Zhang. "Relatedness measures to aid the transfer of building blocks among multiple tasks," in Proceedings of the Genetic and Evolutionary Computation Conference. Association for Computing Machinery, 2020.

4. ConCS, a system of multiple type-fitting XCSCFA agents, can learn continually multiple problems, which may be presented to the system at a random order or at the same time. The system can solve increasingly more complex problems, such as n-bit Multiplexer, n-bit Carry-one, and multiple n-bit Hierarchical Boolean problems, as soon as it obtains all necessary building blocks through learning easier and more general problems. ConCS is the first system that can learn continually without the need of human guidance on providing curricular orders. In contrast, the results of ConCS can provide the connections among learnt knowledge, which can infer the curricular order.

   This contribution has been written and submitted to:

   T. B. Nguyen, W. N. Browne, and M. Zhang, "ConCS: A Continual Classifier System for Continual Learning of Multiple Boolean Problems," submitted to IEEE Transactions on Evolutionary Computation. 2021.

1.6 Organisation of Thesis

The remainder of this thesis is organized as follows. Chapter 2 presents the literature review of related works. The common experimental design is referred to in Chapter 3. Chapter 4 to chapter 6 present the major contributions to fulfil the established research goals. Chapter 7 concludes this thesis.

Chapter 2 provides the necessary background for this thesis. The background reviews LCSs, related evolutionary ML, and learning paradigms with knowledge transfer. The review of LCSs includes a detailed description of XCS, a variety of encoding methods used in LCSs, and especially numerous variants of XCS using tree-based programs (code fragments). This representation will be used as the encoding method for XCS-based systems in a later chapter as it encourages the interpretability of extracted solutions.

Chapter 3 introduces the experimental designs for the experiments in contribution chapters. Also, it gives brief introductions of the benchmark problems used in this thesis.

In Chapter 4, several learning processes and corresponding methods are introduced to improve the autonomy of XCS with CF-conditions. The autonomy is to eliminate the need for the layered learning approach, which includes the design of a sequence of learning stages and transfer criteria. By introducing an evolution of CFs nested within the rule evolution of XCS and a new parameter called CF-fitness, this approach can learn large-scale and especially hierarchical problems in the traditional independent learning.

Chapter 5 presents a multitask learning system that utilises the output of

Chapter 4 to introduce an ability to adapt the feature transfer automatically during the learning process. In a learning system that grows tree-based features, this ability is essential as one-time transfer criteria are only appropriate at one phase of the feature growth. This system is expected to improve the learning performance and reduce the bloat in tree features when multiple tasks are highly related. Another target of the system is to avoid negative transfer in case of low-related tasks.

In Chapter 6, a system with the ability to learn continually is introduced. This chapter reduces substantially the dependency on external guidance in layered-learning systems, which is to provide a curriculum. The system is designed to accumulate increasingly more complex knowledge through learning more and more problems.

Chapter 7 concludes the thesis with the achieved goals. This chapter presents the main conclusions from contribution chapters and the promising future directions that stems from this thesis.

Chapter 2

Literature Review

This chapter provides an overview and details of the essential aspects of Artificial Intelligence and Machine Learning. Hence, the background of subfields of Machine Learning relevant to this work, including Transfer Learning, Layered Learning and Multitask Learning, will be referenced. After obtaining a broad picture of the field, an overview of Evolutionary Computation is introduced. Then a detailed introduction and extensions of Learning Classifier Systems, the baseline algorithms of this research, follows. Lastly, this chapter will review and discuss existing versions of Code Fragment-based XCSs, that will be utilised as the base frameworks in this research.

2.1 Artificial Intelligence and Machine Learning

Artificial Intelligence (AI) is the study of creating artificial agents that can imitate or mimic biological intelligence (i.e. human intelligence) [30, 76]. These agents may take the form of machine learning algorithms, like neural networks and statistical analysis, or robots integrated with machine learning algorithms [109]. According to Bellman [10], AI is:

> ”[The automation of] activities that we associate with human thinking, activities such as decision-making, problem-solving, learning...”

One of the key topics in AI is how to achieve human-like intelligence, which is considered a long-term goal of AI. AI researchers have been investigating many aspects of human intelligence to understand and integrate them into AI.

Machine Learning is an aspect of AI that provides machines with the ability to learn and improve through the experience with the learning environment(s) or data without hard-coded reasoning and decision-making [2, 95]. A Machine Learning algorithm is capable of building its model mainly itself through its exposure to examples. Machine Learning can be divided into three categories based on the different types of given input data and the expected output [113]:

- Supervised learning: the outputs are provided correspondingly with the inputs when the agent learn the examples. The goal of supervised learning is to learn the function mapping the inputs to the outputs.
- Unsupervised learning: the provided learning examples are provided with input data only, and the goal is to learn the patterns within the received data. The learning results could be used for grouping similar unseen examples, initializing a supervised learning system, etc.
- Reinforcement learning (RL): the learning agent will directly interact with its environment by effecting actions on the environment given the perceived data from the environment and learn by adapting through corresponding environment feedbacks (rewards or punishments) [121]. The goal of the agent is to learn the policies that choose the appropriate actions regarding perceived data from the environment to maximize the future rewards it receives.

An RL entity operates in an environment and receives streaming sensed signals from the environment. Thus, RL is more appropriate to the work in this thesis than supervised learning as it is closer to human-like learning than being restricted to learning a labelled training set (supervised learning). Collecting data and splitting the dataset into a training set and a testing set already reduces the autonomy of the AI system. Therefore, Sutton stated that the lexicon of RL is appropriate for describing the problems faced by such cognitive systems like mobile creatures in a complex, stochastic environment [121].

The frameworks anticipated to be used in this thesis are extended versions of XCS, integrated with an RL mechanism. Therefore, this thesis will be focused on learning with environment feedbacks instead of instance outputs.

2.1.1 Transfer Learning

Transfer learning (TL) is a process that applies knowledge learnt from one or more source problems on a related target problem to improve learning performance on the target problem [101, 126]. To understand different techniques of TL, this thesis firstly goes through the understanding of a "domain" and a "task", the components of an independent problem learning, according to Pan and Yang [101]. A domain is composed of feature space and its marginal probability distribution, while a task includes a label space (space of outputs) and an objective predictive function $f(.)$. Therefore, TL can be defined as techniques to help improve the learning of target predictive function in the target domain using the knowledge in the source domain and source task. In TL, the source problem and target problem must be different in either domain or task; otherwise, it becomes traditional a machine learning method [101, 126].

TL can be divided into three different settings regarding the availability of labelled data in the source domain and target domain: inductive transfer learning, transductive transfer learning, and unsupervised transfer learning [101]. The latter two settings are not relevant and thus are not included in this thesis. Inductive transfer learning requires labelled data (the ground-truth of output) for both the source domain and target domain. In this thesis, XCS and XCS-based systems, which are reinforcement learning algorithms, will be used as the main frameworks for single-step problems. Because these systems learn in reinforcement learning without the ground-truth, the labelled data in the inductive transfer learning setting will be converted to rewards from the environment.

On the other hand, TL can also be categorised into four approaches according to the transfer medium [101]:

- Transfer instances: to increase the amount of data for training the target task. The transfer strategy is to re-weight some labelled data in the source domain and reuse them in the target domain [35, 66, 84, 94, 147].
- Transfer feature representation: to find a good feature representation to reduce the difference between the source and target domains with the final aim to minimize domain divergence and error in the target task [14, 40, 67, 88, 137].
- Transfer parameters: to discover shared parameters and priors between the source and target models [44, 114]. Most models transferring parameters include a regularization framework and a hierarchical Bayesian framework, where prior distributions or model parameters can be shared under certain assumptions.
- Transfer relational knowledge: to map relational knowledge, which is the relationship among variables (data input), between the source and target domains [38, 71, 92].

The transfer medium that is the most relevant to this thesis is feature representation as this approach is encouraged by the feature construction in

XCSs with tree-based programs. This approach is also applicable in the feature construction of the multitask-learning systems in this thesis.

Another transfer medium that has not been addressed the literature of transfer learning was the functions extracted from the final solutions of solved problems. This approach was firstly implemented in the field of learning classifier systems by Alvarez et al. [3, 4, 6, 7]. The extracted knowledge is encoded in the form of functions, which are available to be reused in other tasks in layered learning.

Layered Learning

Layered learning is a hierarchical Machine Learning paradigm applied to problems where learning a mapping from inputs to outputs is intractable using currently available algorithms [118]. A hierarchical problem was decomposed in a bottom-up manner into a sequence of subtasks where each requires a learning session. For each subtask, an appropriate machine learning algorithm was used and might differ from ones for other subtasks. The extracted knowledge at each subtask was directly fed to the learning of the next subtask at a higher layer. The subtasks are supposed to be correctly determined as the aspects of specific domains. The hierarchical decomposition was done by a human to learn several problems in [118]. This work applied layered learning to decompose the complexity of the activity of robotic soccer. The authors showed that their proposed layered learning allowed the program to evolve goal-scoring behaviours with quality analogous to the evolved outputs of standard GP, in a shorter time.

Recently, layered learning was implemented to allow a CF-based XCS (see section 2.3.3), XCSCF* [6, 7], to generate a generalised solution to multiplexer problems. A 6-bit multiplexer problem was decomposed into a number of consequent basic subtasks, e.g. $NAND$ ($NotAND$), OR, $ValueAt$, $PowerOf$. All the aspects of multiplexer problems were addressed in subtasks, and only basic general operations and skills were provided with predefined functions. The subtasks were given with ideal data sets created with prior knowledge about the functions they should learn. The final result of the system on multiplexer problems was claimed to be optimal and general to all multiplexer problems and therefore was not scale limited.

Multitask Learning

In addition to TL, there are other subfields of Machine Learning, such as multitask learning and continual learning, that also transfer knowledge among tasks and domains. However, they are slightly different from TL in several aspects. In multitask learning, more than one tasks are learnt simultaneously, where there is no difference in priority like source and target as in TL [29].

Because these two subfields also address the problem of transferring knowledge, they would share very similar aspects and challenges. For example, it is possible to extend a transfer learning method to apply for multitask learning, and continual learning. In the case of this thesis work in the network of functionalities and patterns, the learning paradigm of the multi-learner system is a mix of multitask learning and continual. While layered learning was investigated in XCSCFC [60] and XCSCF* [6], multitask learning has not been applied in any version of CF-based XCSs.

In this thesis, the target tasks in multitask learning may differ in decision-boundary patterns. Additionally, they may have different domains because the input variables for each problem might change in size as well as marginal distribution. When it comes to involving transferring knowledge between binary and real-valued problems, both the problem domains and tasks might be related but are of different categories. For example, in some scenarios, the task providing transferred knowledge can be a regression problem, and the task receiving the transfer is a classification problem in

[6]. Also, one domain might be a regression problem with an input length of one variable only while a closely related domain might have a variable input length.

Continual Learning

In the literature of AI, there has been research that focused on learning continually based on reusing learnt knowledge. Continual learning is the constant development of increasingly complex behaviours; the process of building more complicated skills on top of those already developed [112]. This concept is highly related to the definition of layered learning. Continual learning, however, is focused on the continuity and autonomy of the learning process without a curriculum and a final task.

Another term that is mostly identical to continual learning is lifelong learning [125]. This is a machine learning approach that addresses situations where a learner faces a stream of learning tasks. The readers interested in continual/lifelong learning are advised to consult [125] to understand this approach.

2.2 Evolutionary Computation

Evolutionary computation (EC) is a family of population-based problem-solving techniques, where each individual represents a candidate solution or a part of a whole solution to the targeted problem [74]. EC techniques mimic the principles of Darwinian evolution theories, such as natural selection and reproduction, and biological principles. Every EC technique maintains and evolves a population of individuals. The population is evolved using some forms of selection, mutation and crossover operators to search for the fittest candidates regarding a pre-defined fitness evaluation.

Specific EC techniques are used in traditional Learning Classifier Systems as well as CF-based LCSs, e.g. Genetic Algorithms (GA) and Genetic Programming (GP). The reproduction technique crossover within GA could be maintained, while the GP-like tree-based programs, which are CFs, are the main representation form of the rule conditions of the framework used in this thesis. The two common EC techniques, GA and GP, are briefly introduced in the following sections.

2.2.1 Genetic Algorithms

GA is an evolutionary computation technique used to generate high-quality solutions to optimisation and search problems and is commonly implemented in the discovery components of LCSs [129, 130], the main framework of this thesis. GA integrates bio-inspired operators such as mutation, crossover and selection. Holland introduced schema theory to explain how evolutionary processes in GA works [54]. $Schema$ is a template for describing a set of positional, finite and fixed-length strings defined using a finite set of alphabets. For example, the alphabets for representing bit string are $0, 1, *$, in which '*' is for "wild card" and is not actual value, then a schema "$01 * 1$" represents both '0101' and '0111'. A wild card or a "don't care" bit means its actual value could be any actual value in the set of representing alphabets.

The normal process of GA starts with the selection of fittest individuals from a population by a selection operator. They are combined using crossover and mutation operators to produce offspring which are expected to have inherent characteristics of the parents and will be added to the next generation. This process is believed to create offspring fitter than parents and therefore have a better chance at surviving. The algorithm keeps on iterating until a generation with the fittest individuals will be found. Crossover, the operation of combining parents in reproduction, is illustrated in Figure 2.1.

Goldberg proposed the *building block hypothesis* that analyses how GAs works because they can find good building blocks [46]. Building blocks are just schemata with short defining lengths[1] which consist of bits that work well together. Goldberg hypothesized that, in GA, the short (defining length), low-order[2], and highly fit schemata are sampled, recombined, and resampled to form strings with potentially higher fitness. However, Goldberg discovered a major weakness of GA that in order for the building blocks to form, there must be low epistasis among the genes, which are interaction among the positional bits. Therefore, Estimation of Distribution Algorithms with explicit "linkage learning" schemes, as a research branch of EC, have been developed to deal with the high epistasis problems [50, 102–105].

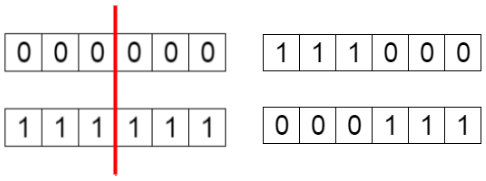


Figure 2.1: Illustration of crossover. On the left, selected parents are combined using a crossover point. The two individuals on the right are offspring.

2.2.2 Genetic Programming

Genetic Programming (GP) is also an EC technique that works on optimisation and learning problems [74]. Each individual in the population is a potential candidate for a complete solution for the tackling problem. To develop population each generation, GP also reproduces by bio-inspired operators such as selection, crossover and mutation. The process of evolution in the GP population follows a similar order of steps as GA.

The distinction of GP from other EC techniques is that each individual in

[1]The defining length is the distance between the outermost non-wildcard symbols
[2]The number of non-wildcard symbols

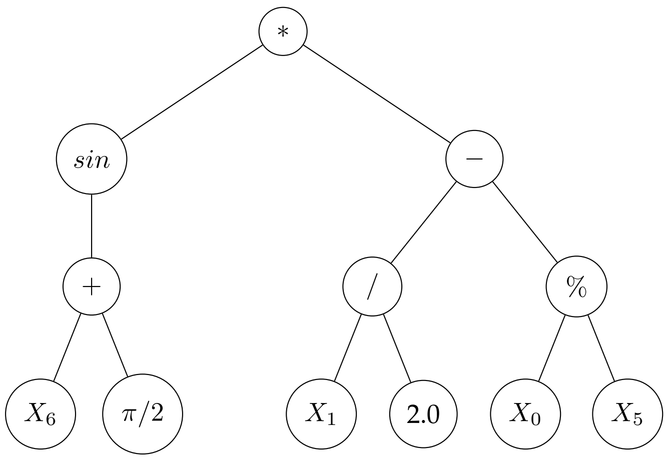


Figure 2.2: Example of a GP tree for the following program $(sin(X_6 + \pi) \times ((X_1/2.0) - (X_0\%X_5))$

the population is a computer program represented in the form of graph-based programs. In addition to standard GP using (parse) tree programs, there have been other variants of GP, such as cartesian GP [93], linear genetic programming [16], and gene expression programming [41]. This thesis will use the form of tree programs for the developed systems. The solution to the solving task is expected to be expressible by a predefined and primitive set of operators, a.k.a. the function set, and a set of operands, also called the terminal set. The program trees are composed of functions in the internal nodes and terminals in the leaves. See Figure 2.2 for an example of a tree program in tree-based GP.

A tree-based GP program with rich alphabet composed of the terminals and the functions allows high complexity and flexibility of solution representations. However, its undesirable effect is a high probability of producing bloat, which are non-relevant structures of code without corresponding increases in fitness [74]. Bloats not only consume considerable resources but also interfere with finding better solutions, since the code manipulation by evolutionary operators may occur in the non-relevant regions. They might also prevent the solutions from generalising [133, 136].

In addressing the problem, the usual approaches are to compromise between flexible representations and simpler optimal structures. The growth of tree-based programs can be constrained by limiting the maximal allowed depth of individual program or adding tree size punishment to fitness measurement [87]. The other approaches to avoid bloat are to simplify individual trees using numerical simplification method [70], algebraic simplification method [148], or using multi-objective GP system to promote population diversity and control bloat [39].

As an ML algorithm, GP is capable of generalising, which is one of the most desirable properties machine learning [95]. The generalisation ability is closely related to the model (tree) complexity in GP [31]. Chen et al. [36] proposed a novel complexity measure based on the Rademacher complexity and integrated into a new GP method to improve generalization. Next, Uy et al. [132] considered semantic control as an approach to promote the generalising ability of GP for the first time. This work extended the semantically driven crossover method in [9] to improve the semantic crossover of real-valued tree-based programs. They defined the approximate semantic output of a subtree as a vector where each element is the output of a point from the domain. Semantic similarity crossover allows crossover when the distance of semantic vectors of two subtrees of the offspring is not greater than a user-defined threshold. The impact of semantic control was compared against using validation sets on the generalisation. The results on a set of real-world symbolic regression problems showed improved performances with smaller sizes of evolved solutions by applying semantic control. The result by using semantic control is significantly better than using validation sets on enhancing generalisation ability. The application of semantic behaviours in reproduction is extended in [75, 97] by geometric semantic operators. Geometric semantic operators use geometric transformations to search directly on semantic space. The generalization ability of GP was enhanced by geometric semantic operators in [32–34, 134]. The semantic or geometric semantic behaviours will be investigated in this thesis work as an approach to compare the difference and mapping between problems and solutions. Instead of modelling problems and solutions by pre-defined distributions with learnt parameters, it would be more efficient to use semantic output vectors to represent problems and solutions.

As a form of representing conditions of rules in CF-based XCS, GP is the carrier of knowledge to be transferred among learning systems. It is also an integral part of mapping between target problems and extracted solutions.

2.3 Learning Classifier Systems

Learning Classifier Systems (LCSs) are a family of ML algorithms following a concept of systems inspired by Cognitive Science, Computer Science and Biology [129]. The concept of LCSs was firstly formalised by Holland and Rietman [55] as a cognitive system which was to replicate the processes of cognition [20]. However, due to the complication and impractical of original LCSs, the later research in the field has focused on solving interesting problems. The field of LCSs has also been subsumed into the wider field of evolutionary computation. LCSs can be applied to both supervised and reinforcement learning tasks, including, classification, data mining, regression, function approximation, behaviour modelling, adaptive control, and more [19, 129].

All LCS algorithms follow a number of common principles and steps, as illustrated in 2.3 [23]. Being considered as a part of evolutionary computation, each LCS also maintains and evolves a population of classifiers, also called as rules, where all with cooperation are a solution or each is a candidate solution. When receiving the environment state, a subset of the population, called a match-set, having the conditions satisfying the received state is selected. Then, an action is chosen to be effected on the environment. The subset of the match set that votes the chosen action forms an action set. If the match set is empty, then a process called covering is activated to create classifiers for the match set in a random manner. The action set is then updated regarding the corresponding reward returned from the environment. An LCS algorithm can have its own evolutionary crossover as its reproduction mechanism. All the processes are executed for each received environment state to form an iteration. The evolution of the population is finished when the performance of the cooperative population on the environment reaches its maximum, or the population experiences a pre-defined number of iterations.

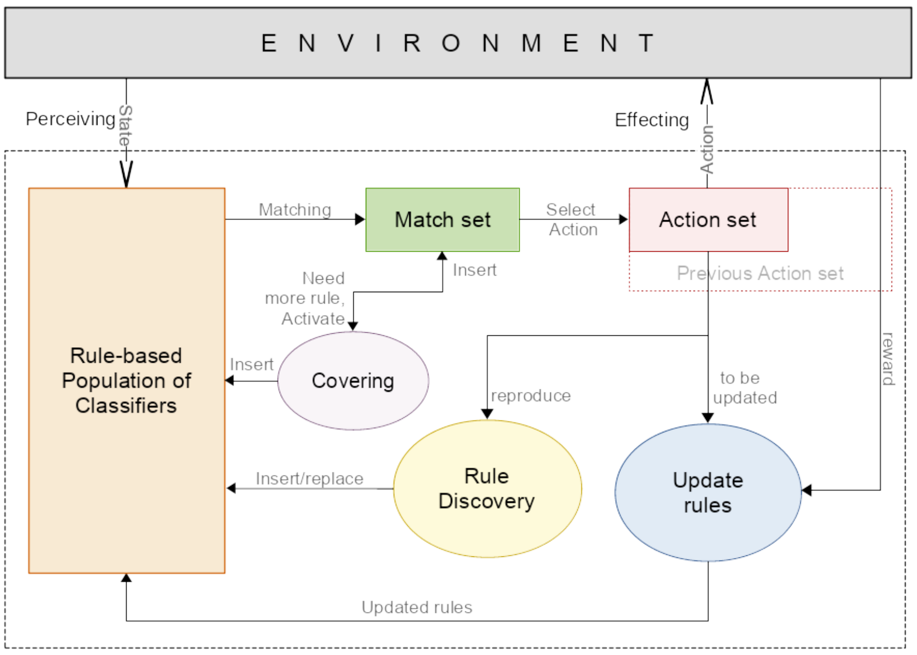


Figure 2.3: General components of a Learning Classifier System

LCSs are developed with two main directions: the Pittsburgh [116] and Michigan [15] approaches. The design of Pittsburgh classifier systems is mainly compatible with offline learning as they produce a population of rulesets, while Michigan classifier systems are well designed for both offline learning and online learning as they evolve a single set of rules. In the scope of this thesis, the branch of Michigan style XCSs is of interest because of their advantages in reinforcement learning.

There are two main types of fitness functions in Michigan classifier systems: strength-based and accuracy-based functions. The strength-based fitness functions, such as in ZCS [140], measure fitness of a rule using the magnitude of its predicted payoff. On the other hand, the accuracy-based functions determine the fitness based on the accuracy of predicted payoff. With accuracy-based fitness functions, if a classifier is consistently correct at predicting a reward signal, it is considered fit. According to Urbanowiz and Browne [129], accuracy not only helps in predicting the payoff from the environment but also guides rule discovery to consistent areas of the search space, which empirical evidence suggests are the most profitable to breed. XCS [26, 141] is a widely adopted algorithm of Michigan approach using an accuracy-based fitness function. XCS will be the base framework of this thesis and one of the baselines to be compared with the resulting systems. As a Michigan-style approach, its scalability in online learning as it does not require any prior training set.

2.3.1 Accuracy-based Classifier Systems

XCS, an accuracy-based classifier system, is a Michigan-style LCS that uses accuracy-based fitness function to form an evolutionary and adaptive learning agent with a set of mutually cooperative classifiers [26, 138, 141]. Based around ZCS [140], an LCS using strength-based fitness, XCS was first proposed by Wilson [141]. Butz and Wilson then refined the updating processes and a few parameters of XCS in [26] to improve its stability. There are a number of key distinction of XCS from other LCS algorithms: 1. fitnesses of classifiers are measured using the accuracy of predicted rewards, which allows forming a complete mapping of state-actions to predicted rewards [72]; 2. XCS removes the message list from the design of the original LCS and therefore it is only applicable to learning Markov environments [23, 26]. XCS has been widely adopted in the field of LCSs and investigated to integrate many rich representations into its rules.

As a Michigan-style LCS, XCS is an adaptive agent which learns by interacting with the environment. XCS follows the common steps and principles of general LCSs and Michigan-style LCSs. Specifically, the learning agent initializes an empty population in the beginning and inserting first classifiers to the population by covering. XCS also follows the workflow of general LCSs, where XCS stores an action set of previous iteration $[A]_{\ 1}$ and updates its classifiers in the reinforcement learning manner. Each classifier, in the form of an expressive rule "if *condition* then *action*", contains two parts: condition and action parts. Standard XCS proposed for binary problems represents the condition part of its classifiers by a fixed-length bitstring defined over the ternary alphabet $\{0, 1, \#\}$, where $\#$ is called wild-card or don't care and is equivalent to the wild-card $^0{*}^0$ in schema, and the action part by binary constants. Each classifier is measured by three main parameters: prediction or predicted payoff p; prediction error ϵ, which is an estimate of error between predicted payoff and corresponding actual (average) reward from the environment; and fitness F measuring how well a classifier fits in an environment. Additionally, a classifier has an experience exp counting the number of times it is in the action set $[A]$, and a numerosity n referring to the number of its copies. Traditionally, the population is evolved using GA operators on the positional bitstrings.

For a complete and detailed description as well as explanations of terms, the interested reader is directed to the original papers of XCS by Wilson [141, 142] and the refined XCS by Butz and Wilson [26]. The operations of the workflow of a standard XCS are illustrated in Figure 2.3 with an iteration example in Figure 2.4. These operations are described briefly in the following:

- Sensing the environment: receives the current environment state s. The current state s is usually in the form of positional fixed-length bitstring, which is a string of binary values $\{0, 1\}$. A positional string means the positions of variables in the string are fixed.

- Matching: all classifiers in the population are queried for matching to form a match set $[M]$ for the current iteration. A classifier is matched with a given state if the non-wildcard variables in the condition are equal to ones in environment state, respectively.

- Covering: is activated to create new classifiers for the match set $[M]$ and population when any action $a_i \in A$, the set of all possible actions, is missing from the match set $[M]$. In covering, a random classifier is generated by generalizing with probability $p_{\#}$ to not only match the current state and has the missing action a_i as well as small default main parameters.

- Action voting: calculates a prediction array PA, which predicts rewards for all possible actions when they are executed on the environment. In PA, a predicted reward for an action a_i is computed by averaging the fitness-weighted predictions of all classifiers advocating a_i in the match set $[M]$.

- Action selection and execution: action selection can be done by selecting the action with highest predicted reward, in *exploit* mode, or by random manner, in *explore* mode. In *explore* mode, the agent is allowed to discover information about the environment, especially the actions giving lower predictions and therefore build a complete mapping of state-actions to rewards. On the contrary, in the *exploit* mode, the agent attempts to obtain as high rewards as possible. Hence, this mode is used to test the performance of the agent as the outcome of the learning process. The selected action is executed on the environment, and a reward r is returned from the environment.

- Updating all classifiers in action set $[A]\ _1$: updates the parameters of all the classifiers using reinforcement learning manner. This part plays as the local search for classifiers that are being updated.

  On receiving the environment reward r, if the environment is a single-step problem[3], or the agent reaches the end of a trial[4], the classifiers in the current action set $[A]$ is also updated. Firstly, the estimated actual reward is computed: $P\ _1 := r\ _1 + \gamma \times max(PA)$ for $[A]\ _1$, where $r\ _1$ is previous reward and γ is the discount factor; and $P := r$ for $[A]$. The update process is summarised in Algorithm 2.1. In the up-

Algorithm 2.1 Updating parameters of classifiers i given an estimate of the actual reward P

```
1: classifier experience $exp_i := exp_i + 1$
2: **if** $exp_i > 1/\beta$ enough experience ($\beta$ is the learning rate) **then**
3:     update **prediction error** $\epsilon_i := \epsilon_i + \beta(|P - p_i| - \epsilon_i)$
4:     update **prediction** $p_i := p_i + \beta \times (P - p_i)$
5: **else**
6:     average **prediction error** $\epsilon_i := [\epsilon_i \times (exp_i - 1) + |P - p_i|]/exp_i$
7:     average **prediction** $p_i := [p_i \times (exp_i - 1) + P]/exp_i$
8: **if** $\epsilon_i \geq \epsilon_0$ error is not small enough **then**
9:     calculate **accuracy** $k_i := \alpha \times (\epsilon_i/\epsilon_0)\ ^{\nu}$, where $\nu > 0$, $0 < \alpha < 1$ and
   $\epsilon_0$ is the threshold error ($\epsilon_0 > 0$)
10: **else**
11:     maximum accuracy when error is small enough $k_i := 1$
12: **relative accuracy** $k_i^0 := k_i \times n_i / \sum_{j2being\_updated[A]}(k_j \times n_j)$.
13: **fitness**: $F_i := F_i + \beta \times (k_i^0 - F_i)$
```

dating process, the threshold $1/\beta$ for classifier's experience exp is to

[3]The problem that returns the result to every iteration. New state of the learning agent in next iteration is not related to the state and action in current iteration.

[4]When the agent reaches a goal and the problem restarts with a new trial, where the state of the agent is not related to the current state and selected action.

follow a technique called "MAM" [135]. The two hyper-parameters ν and α in the above step 9 are used to control the degree of difference in accuracy between classifiers with respect to ϵ. Besides, ϵ_0 is for considering all classifiers having $\epsilon_i < \epsilon_0$ accurate and therefore their absolute accuracy is at maximum, which is 1. For more analysis of those hyper-parameters, the reader is advised to read [23]. The relative accuracy in step 12 is the accuracy of a classifier compared with other classifiers in the currently updated action set. Consequently, in the later-described deletion process, a classifier only competes to survive with other classifiers in the same niche. This is to guarantee that the fitness summations for all action sets are consistent and approximately equal to each other. Therefore, XCS can assure equal resources allocation among niches, which is an advantage in dealing with imbalanced data.

- Reproduction: produces new classifiers, called offspring, to the population. This step usually applies GA operators, including natural selection, crossover and mutation, in the updated action set [A] or $[A]_{\;1}$ when the average experience counting from last reproduced iteration of the classifiers of the set passes a threshold. In the swapped part in crossover operation, a bit is only swapped with the corresponding bit in the other parent classifier when they are not the same, i.e. both wildcard bits or both non-wildcard bits. When a bit is switched or swapped in crossover or mutation operations, a non-wildcard bit will be changed to a wildcard bit and vice versa.

- Deletion: assures that to the total number of classifiers, including their copies, does not surpass the population size N. The computation of deletion vote not only remove low-fitness classifiers but also puts pressure towards equal action set size among niches.

- Subsumption: is a step of replacing overly specific classifiers by a more general and accurate classifier with enough experience $exp >$

θ_{sub}. By replacing, a.k.a. subsuming, the numerosity of more general and accurate classifier accumulates the numerosity of the overly specific classifier and the overly specific classifier is removed from the population. The subsumption is activated whenever a new classifier is added to the population, except in covering because new classifier is the only classifier matched with the current state. Hence, a standard XCS has two subsumption processes: GA subsumption and action set subsumption. Subsumption contributes to biasing the search towards more general, but still, accurate classifiers [27].

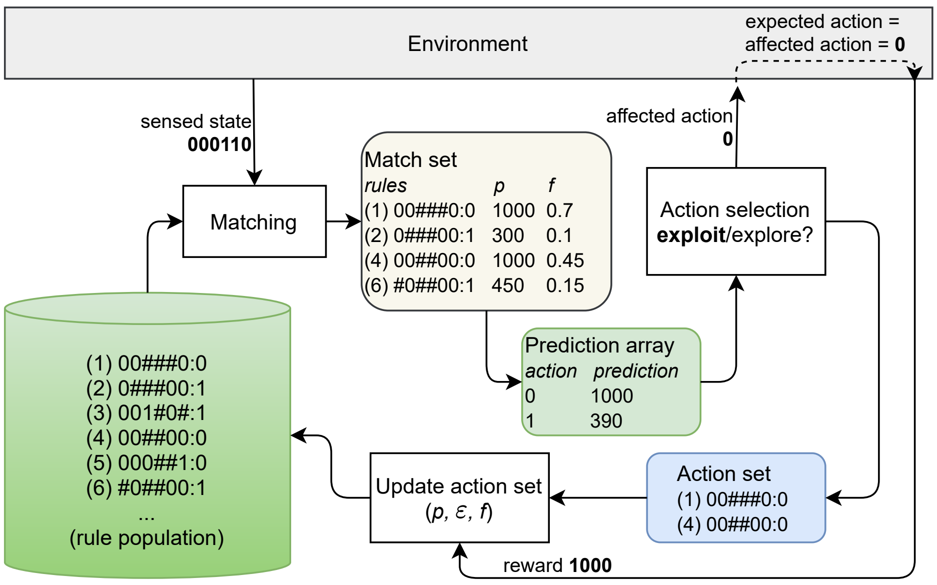


Figure 2.4: An example of XCS workflow in an iteration. There is no rule discovery or covering in this workflow.

As mentioned in section 2.2.1, GA struggles to find relevant building blocks in problems with high epistasis, such as Multiplexer and Hierarchical Boolean problems [23]. A theoretical study has shown that XCS with optimal parameter settings can learn large-scale Multiplexer problems (70 and 135 bits) very fast [98]. However, finding optimal parameter values requires statistical data on the target problem, which is not always available. Therefore, Butz et al. [22] applied Estimation Distribution Algorithms (EDAs) in XCS to capture the interactions among input variables. EDAs were shown to improve the performance of XCS in hierarchical problems. Butz et al. used two variants of EDAs, i.e. ECGA [50] and BOA [102, 103] to build a global probabilistic model of interaction among the bits from the qualified classifiers and assign the model with the statistics of local classifiers chosen from the action set to sample new classifiers. Replacing traditional GA operators, EDAs were demonstrated to be efficient in preserving building blocks [22]. This improved the performances of XCS in hierarchical problems, such as the 15-bit parity-count ones. However, this work based on traditional ternary alphabet representation in XCS, which is lack of richness and flexibility.

To date, a large number of different representations have been integrated into classifiers of LCS algorithms, especially XCS, to improve the scalability of LCSs. These important representations will be introduced and discussed in the next subsection.

2.3.2 Notable Representations Integrated in LCSs

A classifier in LCS in general and XCS, in particular, is composed of two components: condition and action parts. Traditionally, in a binary problem, the condition part is represented by the ternary alphabet, which is similar to *schema* in GA, while the action part is binary-valued. Throughout the development of LCS, various rich encoding schemes have been integrated into the condition and action parts to improve the flexibility and complexity of classifiers, which results in solving larger scale and more complex problems.

2.3.2.1 Real-valued Conditions and Actions

Wilson extended standard XCS to cope with continuous input data in XCSR [143]. The representation of classifier condition was replaced by interval predicates for each input variables respectively. For example, an interval predicate for variable X_i is $int_i = (c_i, s_i)$, where c_i and s_i are real. The interval predicates indicate that the classifier matches an instance $x = x_0, x_1, ..., x_{n\ 1}$ if and only if for all variable X_i with value x_i, $c_i - s_i \leq x_i < c_i + s_i$. Wilson later developed an integer-based version, called as XCSI, where the interval predicate is represented by $int_i = (l_i, u_i)$, where l_i is the lower bound and c_i is the upper bound. XCSR and XCSI have been used as the base for many later extensions of XCS for problems with real-valued inputs, such as real-valued classification problems.

Wilson later introduced XCSF [144] as one of the first initial XCS algorithms that address the category of problems with continuous action, such as regression problems or navigation in a continuous environment. The classifier conditions used the interval predicates of XCSI. Wilson introduces the idea of computed prediction into XCSF, where the action is omitted to be dummy and prediction is an approximation to a function. The predictions of classifiers, which is used to approximate the target function, was represented by either "piecewise-constant" functions or "piecewise-linear" functions with a weight vector. The results showed that XCSF successfully learnt "2-line" piecewise-linear function, squared function and $sine$ function. Tran et al. [127] extended XCSF to have computed continuous actions instead of dummy actions, named as XCSFCA [127]. XCSFCA was similar to XCSF but instead of using linear functions for predictions, it applied the linear combination of input on the action parts of classifiers. XCSFCA was shown to approximate the frog problems very well with less than 1% error in frog 1 problem and around 1% error in problem frog 2. Lanzi and Loiacono introduced another extension of XCSF combined with sUpervised Classifier System (UCS) [12], named as XCSCA, to be used for problems with binary inputs and a large number of discrete actions [81]. The action part of a classifier in XCSCA was represented by a parameterized function, where the parameters were learnt in a supervised fashion. The results showed that XCSCA surpassed XCS in 20-bit multiplexer problem and evolved accurate representations of actions that were difficult challenges for standard XCS.

2.3.2.2 Neural Networks

Another trend is to integrate Neural Networks (NNs) into LCSs to combine the readability of LCSs and the learning power of NN variants [18, 21, 37, 45, 52, 56, 69, 80, 89, 90, 111, 115]. Bull and O'Hara [21] introduced neuro and neuro-fuzzy into XCS in X-NCS and X-NFCS, where they replace the condition-action parts of a classifier by a small feedforward NN. The action value of a classifier is then computed as the output of its NN with environment state input. For a classifier, another output of the NN is used to signify the membership of the classifier in a match set. Experimental results indicate that NN based XCS are capable of solving single-step and multi-step binary problems with competitive performances. X-NFCS, which extended X-NCS to problems with continuous actions, was proved to be able to solve regression problems, such as root-mean-square [21].

Representing classifiers with NNs has been also investigated in other research. PANIC was the first LCS using NNs for rules, but it was applied on a Pittsburgh-style Classifier System [45]. Lanzi and Loiacono [80] extended XCSF to use multilayer NN for approximating the predictive function in XCSFNN, which resulted in a better performance compared to XCSF. Dam et al. [37] integrated a NN into each classifier action of UCS to build an algorithm named NLCS. Howard et al. [56] managed to extend further XCSF with Spiking NNs [64], the third generation of NN, for approximation function used in prediction. Experimental results indicated that Spiking NN offers an interesting alternative to Multilayer Perceptron representation, in terms of solution sizes and search stability. Matsumoto et al. [89] used encoder output from Deep Autoencoders [53] to feed into XCSR. This system achieved 99.9% accuracy on Connectionist

Bench. This approach was extended in DCAXCSR [90] to enable classifying image data, i.e. MNIST [82], with interpretable rules. Siddique et al. [115] used UCS to create readable rules for deep features extracted from Deep Neural Networks [83]. Through classifying between "dog" and "cat", the system was shown to improve the robustness against adversarial attacks in deep learning.

2.3.2.3 Tree-based Programs

A branch of representations using tree-based programs is of interest because they provide flexibility and readability. Various research has been dedicated to integrating tree-based programs into LCS [1, 3, 6, 57, 60, 63, 78, 79, 131, 140**?**]. Ahluwalia and Bull [1] applied a GP-based rich encoding in ZCS [140] to form an algorithm named GP-CS. GP-CS represented the classifier actions by S-expressions. This system does not perform classification itself but generates filters for extracting features as the inputs to classification algorithms like k-nearest neighbour.

LISP S-expressions was introduced into the classifier condition for the first time by Lanzi and Perrucci in XCSL [79]. XCSL successfully learnt both single-step and multi-step problems, e.g. the small scale multiplexer problems and $Woods1$ problem. Next, Lanzi proposed an XCS with stack-based Genetic Programming [78], where the tree-based programs are in the form of mathematical expression using Reverse Polish Notation. The algorithm allowed created conditions to be syntactically incorrect and, therefore, the search space was undesirably large. This limited the system from learning large-scale multiplexer problems. Uwano et al. [131] recently combined Random Forest [17] and XCS to propose a high-dimensional data mining technique called Random Forest-based XCS. Random Forest was used to generate branch nodes, which were considered as attributes in XCS. This system surpassed XCS in multiple Multiplexer problems.

Ioannides and Browne [57] investigated the scaling of abstracted LCS using combinations of ternary alphabet and S-expressions in classifier conditions. This addressed an important challenge in using flexible representations which were that the generation of good building blocks of information is infrequent. By comparing S_XCS using basic operators and S_XCS1 provided domain-relevant functions for the target problem domain, the authors showed that the tailored functions improved learning of multiplexers both in the sense of faster learning and generalization, but the improvements were not significant. It was also demonstrated that generalised solutions can be discovered through abstraction, which was through learning on the generalized rules learnt in a ternary alphabet. Later Wilson [145] represented the classifier conditions of XCSF with genetic expression programming (GEP), which was *expression trees* of input data. GEP provides greater insight into the regularities of the target environment/problem compared to the traditional interval predicates, although the learning process was slower and the evolved populations were not compact. In 2012, GP-like tree-based programs were introduced into rule conditions of XCS in the form of code fragments by Iqbal et al. [60]. Transfer learning and layered learning were then applied for the first time in code fragments-based XCSs to reuse learnt building blocks [3, 6, 60, 63]. The code fragments-based XCSs will be the base for this thesis work and therefore will be discussed in detail in the next subsections.

2.3.3 Code Fragment-based XCSs

The CF-based XCSs are XSCs that either integrate tree-based programs (CFs) in rule conditions or actions. CFs enable rich and flexible representation for XCS which can technically encode any function. The limitations of existing CF-based XCSs are parts of the motivations of this thesis.

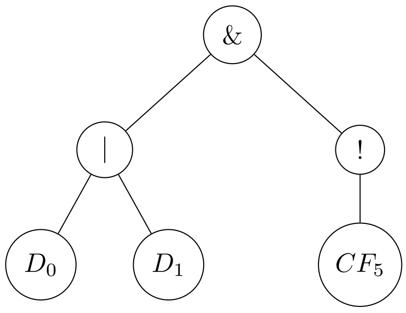


Figure 2.5: Example of a CF for $((D_0|D_1)\&(!CF_5))$

Code Fragments

Code Fragments (CFs) were originally defined as binary trees of depth up to two, which was set to limit tree size and therefore bloating. A binary tree of depth two can have a maximum of seven nodes, where each could be an internal node or terminal node, see Figure 2.5. The internal nodes are dedicated for functions, whereas the terminal nodes are assigned with bits/variables from environment states or other CFs. The functions in internal nodes are chosen from a pre-determined set of functions. This set usually contains basic operators $\{AND, OR, NOT, NAND...\}$ for the domain of binary problems and $\{+, -, \times, /...\}$ for symbolic regression problems. By replacing terminal nodes with other CFs, newly created CFs grow deeper (higher-level) and have the potential to address more complex data patterns. Integrating XCSs with CFs provides numerous solutions to problems that used to be intractable for standard XCS, such as the 135-bit multiplexer [60, 61]. Moreover, the use of CFs in terminal nodes has enabled the reusability of learnt knowledge, which was done in transfer learning [3, 60] and layered learning [6].

In addition to using a pre-determined set of functions in tree-based programs for LCS, Alvarez et al. [3] introduced XCSCF^2 which extended the concept of CFs to use constructed functions in the internal nodes of CFs rather than only pre-determined functions. Constructed functions were inspired by the use of Automatically Defined Functions in GP [74]. A constructed function is represented by a rule-set extracted from the final population of an XCS learning a problem. The ruleset is evaluated in the same way with XCS in exploit mode. The rule-set function is expected to produce a mapping from environment states to actions that solve the problem. Therefore, CF-based learnt functions allow any number of arguments (not only 2) depending on the number of input variables of the problems used to learn the functions. This extends the concept of CFs to a general concept of tree-based programs, which accept any number of terminals and theoretically can encode any function. This extension was believed to reduce the search space of XCS and hence improve the learning process. However, XCSCF^2 was only superior to XCSCFC in small scale multiplexer problems [3].

XCSCFC - An XCS with Code-Fragment Conditions

XCSCFC [60, 63] utilize CFs in classifier conditions as a rich representation replacing the traditional ternary alphabet. The function set is constant during learning and provided prior to the learning process. The set usually contains basic operators as mentioned in the preceding subsection about CFs. The use of CFs in classifier conditions results in an inherent benefit of decoupling between a CF and a position within the condition. Although Iqbal set a constant number of CFs in each classifier condition, the number of "meaningful" CFs (non *don't care* ones) varies across rules since the number of *don't care* CFs varies. The number of CFs in rule conditions can be any number without the use of *don't care* CFs. To embed CFs within classifier conditions, XCSCFC extended standard XCS [26] in the following components: the classifier matching procedure, the covering operation, the rule discovery operation, the subsumption mechanism, and the checking equality of two classifiers. There are two implementations of XCSCFC, one keeps all CFs in a separate population and another simplified one maintains CFs within classifier conditions. The two versions differ from each other in a few minor operations. The following descriptions will be given for the later implementation.

Matching: a classifier cl from the population $[P]$ is said to match an environment state s if all the CFs in its condition output 1. A CF is evaluated by loading the symbols in the terminal nodes with corresponding values from environment state s. A classifier condition in XCSCFC has a fixed number n of CFs. Iqbal defined a constant *don't care* CF, which is equivalent to the "don't care" (or wildcard) bit in standard XCS. The *don't care* (wildcard) CF is set as $(D_0|(!D_0))$ to always output 1 with any environment state.

Covering: is activated in the same way with XCS when an action is missing in the match set $[M]$. This operation creates a random classifier which is matched with the current environment state s and has the missing action. $P_{don'tCare}$, a similar parameter used in XCS, defines the ratio of *don't care* CFs in a classifier condition created in covering operation. The workflow of the covering operation is described in Algorithm 2.2.

Algorithm 2.2 XCSCFC: Covering Operation

```
1:  initialize a $cl$: assign action $a_i$, initialize condition $cl.cond$
2:  for i=1 to n do
3:      if $random[0,1) < P_{don'tCare}$ then
4:          $cl.cond[i] :=$ don't care CF
5:      else
6:          $val := 0$
7:          while $val \neq 1$ do
8:              $cf :=$ randomly create a CF
9:              $val :=$ evaluate $cf$
10:         $cl.cond[i] := cf$
```

Rule Discovery: produces two offspring using GA operation in the action set $[A]$. The crossover operation in XCSCFC is almost identical to that in

XCS, except that the swapping of CFs between two selected classifiers is done without checking whether the CFs are wildcard or not. This is because the two CFs to be swapped might be both non-wildcard but still different from each other. The mutation operation in XCSCFC differs from the one in XCS in the way of changing a wildcard CF into a non-wildcard CF. This process is done in the same way of creating a new CF in the previously described covering operation. That is, each CF is randomly created and evaluated until the output of the created CF is 1, then the process exits by returning the last created CF.

Comparing the equality of two CFs: is done syntactically by matching character by character in each node of the two CFs. The semantic comparison of two CFs requires evaluation of all possible values of the symbols in terminal nodes. Since XCSCFC allows multi-level construction of CFs, this could lead to recursive queries of terminal nodes until reaching the input variables/bits. Therefore, even though the semantic comparison is accurate, it is too expensive and not implemented in XCSCFC.

Comparing the equality of two classifiers: is based on comparing conditions and comparing actions. While comparing actions is straightforward, comparing two conditions of two classifiers is based on syntactic comparison and not expected to be precise for the same reason causing difficulty in semantic comparison of two CFs. Therefore, the comparison of two classifiers is simplified by comparing whether the two sets of non-wildcard CFs in two classifiers are identical. The set comparison was based on a comparison of two CFs in the above manner.

Subsumption: a classifier cl_1 can subsume another classifier cl_2 if they share a same action and the cl_1 is accurate, sufficiently experienced and more general than cl_2. Checking whether a classifier cl_1 is more general than cl_2 is not done semantically because the flexibility of the functions in two CFs does not provide a direct comparison of the sets of instances matched by the CFs. Therefore, a classifier cl_1 is more general than cl_2

if the set of non-wildcard CFs in cl_1 is a subset of the corresponding set of cl_2. This is based on comparing equality of two CFs described above. The subsumption in CF-based XCSs is very infrequent because multiple different genotypes of CFs can actually represent a common phenotype of a program.

XCSCFC [60] was experimentally slower than standard XCS in independent learning of binary problems, except the overlapping niche-based problems. In independent learning, XCSCFC was slower than standard XCS since the flexibility of CFs causes a much larger search space compared to that in standard XCS using the ternary alphabet. The only approach to tackle this challenge was to limit the level of CFs to 2. The generation of CFs was based on the random creation of CFs which were guaranteed to match only the current environment state. This method has a small probability to produce generalised CFs in complex problems with a large search space.

XCSCFA, XCSRCFA - XCS with Code-Fragment Actions

XCSCFA [62] and XCSRCFA [61] are two versions of CF-based XCSs that extended XCS and XCSR respectively to use CFs to replace traditional action representation. The condition parts in XCSCFA and XCSRCFA are identical with the corresponding parts in XCS and XCSR respectively, which are ternary alphabet [141] and interval predicates [143]. Therefore, the operations of XCS related to classifier conditions were not changed. The rule discovery needs to apply GP-based crossover and GP-based mutation on the action parts of parent classifiers. The comparison of CF actions also follows syntactical comparison, similar to comparing classifier conditions in XCSCFC. Internal nodes in CFs representing actions will apply the set of basic binary operators $\{AND, OR, NOT, NAND...\}$ for binary classification problems (XCSCFA) and the set of numeric calculation operators $\{+, -, \times, /...\}$ for symbolic regression problems (XCSRCFA).

Experimental results showed that XCSCFA successfully solved different complex Boolean problems, especially the overlapping and niche imbalance problems. The reasons behind its power against such problems was the inconsistent actions and the redundancy provided by the CF representation of actions. The diverse genotypes of CF-based actions implicitly disabled most subsumption operations.

XCSRCFA is a combination of XCSR and GP (CFs), but it was more powerful than each of them. XCSRCFA is superior to GP in piece-wise function approximation problems because of its niche-based property. On the other hand, it showed an advantage over XCSR because of its powerful representation of actions. However, the diversity of program genotype in CFs also implicitly disabled subsumption operation and therefore caused undesirably large search space. Thus, the final population contains redundant and inefficiently large classifier rules. XCSRCFA also could not scale very well due to that reason.

Extending the concept of Code Fragments with XCSCF^2, XCSCF^3

Alvarez et al. [3] extended CFs in XCSCFC to use constructed rule-set functions in the internal nodes in XCSCF^2. The new concept of CFs theoretically allows encoding of any functions with any number of arguments. This greatly increases the flexibility of CFs and also undesirably increases the search space, which consequently limits the scalability. Therefore, Alvarez et al. [5] later proposed an approach to compact rules of final learnt population, named Distilled Rules, in XCSCF^3. It was an effort to transform the various genotypes of a program to the same rule-set with traditional ternary alphabet representation. This allows subsumption to be possible in CF-based XCSs. Moreover, it also constrains the growth of the search space as the problem scale increases and improves the scalability of the proposed system. Experimental results showed that XCSCF^3 can scale to 70-bit multiplexer problems. Its scalability was not as good as an older

CF-based XCS, which was XCSCFC [60].

Reuse of Learnt Knowledge in CF-based XCSs

By reusing learnt CFs in the manner of transfer learning, Iqbal et al. [60] improved the learning performance of XCSCFC significantly, especially in large-scale multiplexer problems.

The fitter CFs from smaller problems were used to create CFs in a larger-scale problem of the same domain. The fitness of a CF was estimated by the accuracy and fitness of sufficiently experienced classifiers containing it. CFs from smaller-scale problems were transferred and used as terminal/leaf nodes to generate CFs for higher-scale problems. The problem domain used to test this approach was multiplexer. However, the application of this approach is limited to problem domains where at least parts of all the interactions among bits/variables of the domains at different scales are maintained. For example, in the case of multiplexer domain, the patterns (CFs) working with address bits in a smaller scale multiplexer can be useful for a larger scale multiplexer. For transferring between different problem domains or in the same domain of other problems, transfer learning with CF-based XCSs would require the ability to transfer learnt functions.

Reuse of learnt knowledge was also implemented in layered learning with XCSCF* [6, 7]. XCSCF* has the same components and workflow as XCSCFA [62] in each learning stage of layered learning. Additionally, XCSCF* can transfer learnt functions in rule-set forms with the limitation of a strict learning order. The details were given in the introduction of layered learning in subsection 2.1.1. This approach requires human guidance in prior knowledge of preset functions and prerequisite sub-tasks as well as the ordering of the problem sequence.

2.4 Chapter Summary

This chapter provided essential concepts of Artificial Intelligence and Machine Learning, as well as brief introductions of transfer learning and multitask learning. As a branch of AI, EC was introduced along with its common techniques, GA and GP. LCSs in general, and XCS in particular, as the selected base framework, were defined and explained with details about workflows and components. A brief review of important representations used in the prediction parameter, condition parts and action parts of XCS classifiers was also provided.

Since CFs will be used as the representation in classifiers of XCS, the mainstream of CF-based XCSs, e.g. XCSCFC, XCSCF^2, XCSCF^3, and layered learning using XCSCF* were discussed with detailed explanations. Although the generalised building blocks addressing high-level patterns within problems were addressed as a desirable feature for generalisation, no current approaches have tackled this challenge. The limitations of the existing works that form parts of the motivation of this research were also discussed. A summary of these limitations are presented as follows:

- XCSCFC in traditional independent learning was slower than standard XCS due to the fact that the flexibility of CFs causes a much larger search space compared to the ternary alphabet in standard XCS.

- The generation of CFs was based on random creation of CFs that guaranteed to match only current environment state. This method was not likely to produce generalised CFs.

- Layered learning using XCSCFC was only applicable in specific problems, such as Multiplexer, because the CFs integrating the address bits are reusable. The ability to find appropriate functions to solve a sub-task in another problem would be a desirable feature to reuse knowledge in CF-based XCSs.

- Layered learning with XCSCF* needs human guidance to produce general solutions. This is not practical in Machine Learning.

This thesis work will address these limitations.

Chapter 3

Experimental Design

This chapter provides an overview of the problem domains and the common design used in the experiments of the contribution chapters.

3.1 General Experimental Setup

Unless stated otherwise, XCS is configured with its common settings in the literature [26], which generally provides good performances across many problem domains: the learning rate $\beta = 0.2$; two-point crossover with probability $\chi = 0.8$; the mutation probability $\mu = 0.04$; the experience thresholds for subsumption and deletion $\theta_{sub} = \theta_{del} = 20$; threshold for GA occurrence in the action set $\theta_{GA} = 25$; the initial fitness of covered classifiers $F_{init} = 0.01$ and the initial prediction $P_{init} = 10$; the probability of specificness $p_{spec} = 0.67$, except for Even-Parity domain $p_{spec} = 1.0$; offset prediction error $\epsilon_0 = 10$; fitness exponent $\nu = 5$; reduction of the fitness in offspring $fitnessReduction = 0.1$; max reward 1000 for correct actions and min reward 0 for incorrect actions; the threshold number of actions in the match set that activates covering $\theta_{mna} = 2$; and tournament selection with a tournament size ratio 0.4 in genetic operations. Both GA subsumption and action set subsumption are enabled. Exploration and exploitation are alternated to ensure 50% exploration rate.

XCSCFC is also used in experiments to compare the developed systems. The parameters of XCSCFC are configured the same as the original implementation [63]. Specifically, the XCS processes and parameters of XCSCFC are configured the same with XCS. The maximum depth added in each learning stage is 2.

All stochastic algorithms experimented are independently run 30 times using 30 random seeds. The learning performances of online-learning algorithms in Boolean problems are plotted in figures, where the X-axis describes the number of explored instances (or evaluations/generations); and the Y-axis refers to the accuracies in exploitation. The results in these experiments are averaged among 30 runs. In addition to the strategy of selecting actions, the difference between the exploration and exploitation in XCSs is the presence of genetic operations that creates new individuals based on competitive individuals [141].

3.2 The Problems

The first benchmarks used in the experiments in this thesis are benchmark Boolean problems because they have a measurable (countable) search space and identifiable building blocks. Therefore, it is easy to analyse the results and investigate the final solutions in these problems. The complexity of tested Boolean domains is sufficient to test the learning capacity of the developed classification systems.

The second set of benchmarks in this thesis are real-world datasets. These are datasets from the UCI repository and are dedicated to the classification task. The data size and the number of attributes of these datasets vary from small dataset like UCI Zoo to relatively big one like UCI Phishing. This set of experiments demonstrates the capability of LCSs to learn real-world problems.

3.2.1 Boolean Problems

The following Boolean problem domains will be used in the experiments: the Multiplexer, Carry-one, Even-parity, and Majority-on domain, along with their Hierarchical combinations. Also, the subproblems supporting those problems are introduced in Chapter 6. These problem domains are diversified in characteristics and tested in varied scales. The diversity of problem domains enables the experiments to demonstrate different aspects of the tested systems.

3.2.1.1 The Multiplexer Domain

The Multiplexer problem domain is inspired by the multiplexer logic circuit in electronics. In Multiplexer domain, the bitstring input contains data bits and address bits. The address bits are equivalent to the selector in multiplexer circuits, where their values determine which data bit (data channel) is connected to the single output. Thus, if there are k address bits, their 2^k possible values correspond to 2^k data bits. Specifically, the address-bit value is equal to the index of the connected data bit. The total length of the input is $n = k + 2^k$. An example of the 6-bit Multiplexer problem is depicted in Figure 3.1.

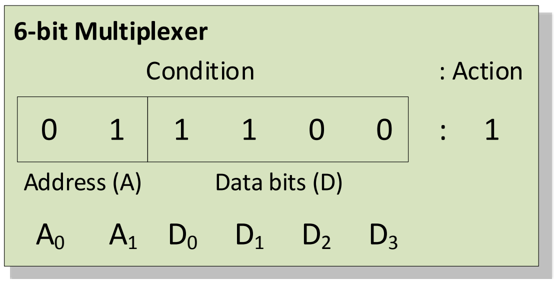


Figure 3.1: 6-bit Multiplexer problem showing the address bits (A_0, A_1) and the data bits (D_0, D_1, D_2, D_3) of the condition, this distinction is not provided to the learning system.

The Multiplexer domain is challenging and interesting because it has epistasis and is highly non-linear. The search space of the problem is also adequate enough to show the benefits of the proposed work. For example, the search space of the 135-bit Multiplexer problem consists of 2^{135} combinations, which is immensely beyond enumerated search [73].

3.2.1.2 The Carry-one Domain

The Carry-one domain is the set of problems that checks whether the addition of two numbers, in the form of binary numbers, carries one in the addition of the highest-level bits of the two numbers. Binary numbers are represented by bitstrings. The input of Carry-one problems is a bitstring resulted from concatenating the two bitstrings representing the two binary numbers to be added. Figure 3.2 illustrates how a bitstring is calculated to output for Carry-one problem. The division point of the two numbers is not known to the learning system.

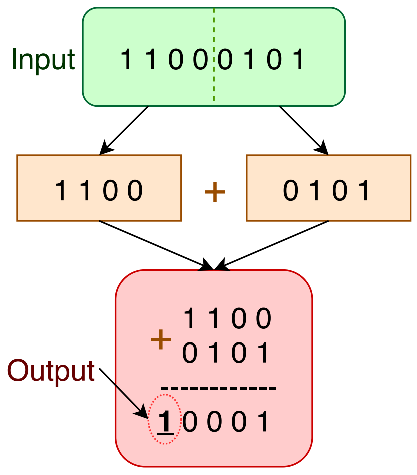


Figure 3.2: A sample of the 8-bit Carry-one problem.

Carry-one problems are problems with overlapped niches. This poses a challenge for XCS to discriminate between optimal rules and over-general ones, especially in large-scale problems. Moreover, the niches of this problem domain are highly variable in size. The ability to balance between niches is essential in this problem domain.

3.2.1.3 The Even-parity and Majority-on Domains

The Even-parity problem outputs $True$ when the number of 1 in the input bitstring is even, and $False$ otherwise. For traditional XCS with ternary alphabet representation, Even-parity is a hard problem as it is against all the generality pressures in XCS. A correct solution of XCS for an Even-parity problem must contain a full map of instance space, which is highly inefficient. This is because the niches of Even-parity problems are fragmented into instance level.

The Majority-on problem is a Boolean problem that checks whether the majority of bits in the input bitstring is 1 or not. The expected output is $True$ when the summation of all bits of the input is greater than the half of the input length, and $False$ otherwise. The Majority-on domain has highly overlapped niches. An over-general rule can easily dominate and replace optimal rules in many overlapped subset niches.

3.2.1.4 The Hierarchical Boolean Domains

A Hierarchical problem is a two-layer combination of 3-bit Even-parity problems in the low layer and a Boolean problem in the high layer [23]. Figure 3.3 demonstrates the 15-bit Hierarchical Majority-on problem and how the output is computed. The 15 bits are divided into 5 non-overlapped chunks of 3 bits. Each chunk, as a 3-bit Even-parity problem, produces a latent feature. These latent features are concatenated into a 5-bit hierarchical input, which is passed to a 5-bit Majority-on problem. The output of this layer is the final output of the 15-bit Hierarchical Majority-on problem. Other Hierarchical problem domains are created in the same way to result in two-layer problems.

The search spaces of the Hierarchical problem domains are fragmented

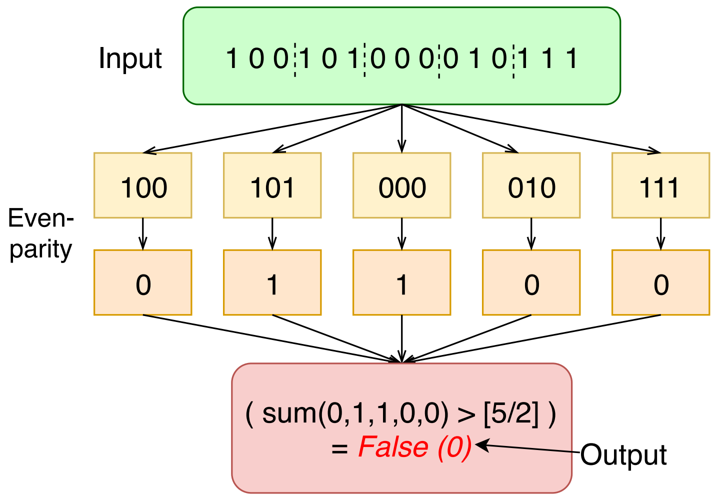


Figure 3.3: An example of the 15-bit Hierarchical Majority-on problem.

because of the low-layer Even-parity problem. Capturing the underlying patterns of a Hierarchical problem requires discovering two different sets of patterns nested together. Therefore, Hierarchical Boolean problems are generally more complex and challenging compared with previously mentioned Boolean domains at the same scales.

3.2.2 Real-world Datasets

The real-world datasets used in the experiments are UCI Zoo, UCI SPECT, and UCI Phishing datasets. The purpose of using these datasets is to evaluate the epistasis and interactions of variables in learnt solutions. This enables analysing the knowledge connections among solutions within one task and across tasks. These datasets have binary and nominal attributes. As the systems in this thesis are dedicated to the classification task with binary attributes, nominal attributes are converted to binary attributes using one-hot encoding. Table 3.1 summaries these three datasets. The detailed description of these datasets can be found on the website of UCI machine learning repository.

Table 3.1: Real-world datasets used for testing. Attributes that are not binary/Boolean are either nominal or numeric with limited values.

Datasets	#Input Attributes	#Boolean (binary) Attributes	#Instances	#Classes
UCI Zoo	16	15	101	7
UCI SPECT	22	22	267	2
UCI Phishing	30	22	2456	2

3.3 Chapter Summary

Boolean problems are chosen as the primary benchmark problems in this thesis because they are scalable, and their solutions are verifiable. Real-world datasets are mostly designed for single-task problems while our thesis is focused on learning continually. It is desirable to create real-world datasets for continual learning to investigate the future research of the work in this thesis.

Chapter 4

Online Feature-Generation of Code Fragments for XCS to Guide Feature Construction

In complex classification problems, constructed features with rich discriminative information can simplify decision boundaries. For example, deep features enables classifying visual objects efficiently with only a shallow fully connected network [83]. This effect was due to the fact that rich features can enable classifying with compact and reliable decision boundaries in such problem domains. CFs produce Genetic Programming-like (GP-like) tree features that can represent decision boundaries effectively in XCS. However, the trade-off for the richness and flexibility results in an undesirable increase of the search space of useful CFs. This is due to that the number of possible tree combinations is exponentially proportional to the tree depth, which is correlated with the representation flexibility.

XCSCFC used CFs in rule conditions of XCS and layered learning to transfer CFs from small-scale problems to large-scale problems [63]. By solving problems with progressively increasing scales, XCSCFC was the first

LCS that can solve 135-bit Multiplexer problem accurately. However, using transfer learning in a sequence requires investigating the criteria for selecting transferred CFs along with constructing learning stages.

Therefore, this chapter introduces a novel model extension, called Online Feature-generation (OF), that allows automatically constructing high-level useful CFs in problems with large search spaces. The OF module enables evolving features (CFs) through a dynamic *Observed List* (OL) of CFs. This extension enables a method for estimating the worth of CFs to identify the patterns in the problem. Therefore, it improves the construction of applicable high-level features. An XCS with OF, called XOF, aims to solve the tested large-scale and hierarchical Boolean problems within fewer instances compared with XCS and XCSCFC in non-transfer learning scenarios. A main contribution of this work is to formalise a system that can autonomously discover complex underlying patterns of hierarchical Boolean problems. This system creates an internal evolution of CFs that, together with the evolution of rules, mutually support each other through a link by a novel CF parameter, the CF-fitness. The OF extension can be a framework to grow the complexity of any representation other than CFs.

The following sections describe in detail two implementations of XOF. The first initial implementation proposes to use the OL to contain the highest CF-fitness CFs. The second version of XOF develops the niching property for CFs. These sections also discuss several different methods of estimating the CF-fitness, the parameter to represent the applicability of CFs. Each implementation will be validated through comparison experiments with XCS, XCSCFC, and other machine learning algorithms in real-world problems.

4.1 XOF - The Initial Implementation

4.1.1 Introduction

Feature selection, extraction and construction are essential parts of classification. The decision boundaries may easily become complex and overfitted when using raw or low-level features in hierarchical and large-scale problems, such as the Hierarchical Even-parity problem [23]. Another approach is to construct high-level features with discriminatory information in order to assist in finding generalised and accurate decision boundaries [83].

Using the ternary alphabet $\{0, 1, \#\}$ to represent its classifier conditions, XCS initially can only have the conjunctive "AND" to connect the original input features in rule conditions. This conjunctive cannot provide a rich set of patterns for the conditions that can simplify the decision boundaries in the latent feature space and thus limits the scalability. Therefore, research has sought to extend the condition representations in XCS to expand its learning ability.

Code Fragments (CFs) provide rich and flexible features because they can produce a wide range of possible patterns linking features to classes. Hence, CFs can be used to find applicable high-level features, which describe the hidden underlying patterns within the problem domain. Useful high-level features may result in compact and robust decision boundaries, which have fewer rules in the population of XCS, in classification tasks. For instance, "XOR" operation can combine several rules using one CF as shown in Figure 4.1.

However, CFs also pose a challenge when constructing and selecting high-level features. The number of possible combinations of nodes to generate CFs increases exponentially with the depth of CF trees. Thus, search spaces can become intractable in complex problems that require high-level

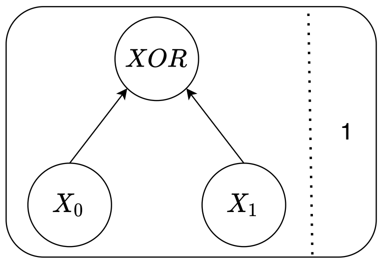


Figure 4.1: An example of a rule using the XOR logic to capture two ternary rules: $\{10 : 1\}$ and $\{01 : 1\}$.

features for compact and accurate decision boundaries. However, the existing methods to generate or transfer CFs between problems are only guided by the current environment state [63]. This guidance has limited probabilities of producing useful high-level features across a problem set.

This work aims to provide XCS using CF-conditions the capability to efficiently discover useful high-level CFs without a customised sequence of layered learning or sacrificing computation time. The ultimate goal is to make XCS using CF-conditions a capable online learning system. First, it is important to define the applicability of CFs. As CFs constitute rule conditions, the applicability of CFs can be defined as their possibility to construct high-fitness classifiers. Based on this sense of CF applicability, another evolution of CFs that interacts with the rule evolution is proposed. The evolution of CFs selects and grows CFs with the highest applicability. The sub-goal is that the evolution of CFs will rely on the evolution of classifiers and also support it to accelerate the learning process of XCS. The idea of CF evolution is implemented through two key components: an "*Observed List*" (OL) that dynamically collects a set of the most applicable CFs and a new parameter, called CF-fitness, to estimate the applicability of CFs. This implementation is grouped in an extension to XCS with CF-conditions, called Online Feature-generation (OF). XCS with CF-conditions that use the OF module is termed as XOF.

The objectives of this work are as follows:

- To optimise/limit the search space of tree features without losing the necessary building blocks by growing CFs locally around the OL. This is based on the premise that higher-level features should encapsulate the highly relevant lower-level features. The search space of CFs is guided from the initial lowest-level CFs to grow higher-level CFs based on the OL.

- To introduce a measure to estimate the applicability of CFs, a new parameter called CF-fitness or the fitness of CFs. This parameter and (rule) fitness are the link that connects the evolution of classifiers and the evolution of CFs. By its definition, CF-fitness can guide the learning system to prioritise applicable CFs for building rule conditions. On the contrary, updated information from classifier fitness can be forwarded to evaluate CF-fitness since CFs interact with the environment through classifiers containing them.

- To investigate whether focusing the search around the OL (depth search), instead of the whole search space (breadth search), can improve the evolution of classifiers without increasing the tendency to be trapped in local optima.

XOF will be experimented on multiple Boolean problems, including Multiplexer, Carry-one, Majority-on, and Even Parity problems. These experiments evaluate the pattern discovery and retention abilities of XOF. XOF will be compared with both XCS and XCSCFC in the independent learning paradigm, without any support of transfer learning, to investigate the learning performance of XOF as an independent classification system.

4.1.2 The Novel XCS with the Online Feature-generation for CF-Conditions

The Online Feature-generation (OF) module introduces evolutionary methods to construct high-level CF features for XCS. A pool of CFs is evolved in the OF, where the rules are equivalent to CFs' environment. It can be interpreted as this evolution is nested within the evolution of classifiers as the CFs interacts directly with classifiers. The evolving population in OF is a CF population storing all in-use CFs. The central component of the CF population is the Observed List (OL), which is a subset with the most preferable CFs. XOF directs the search of applicable CFs and CF construction locally within the OL. The processes and components of the Online Feature-generation are shown in Figure 4.2.

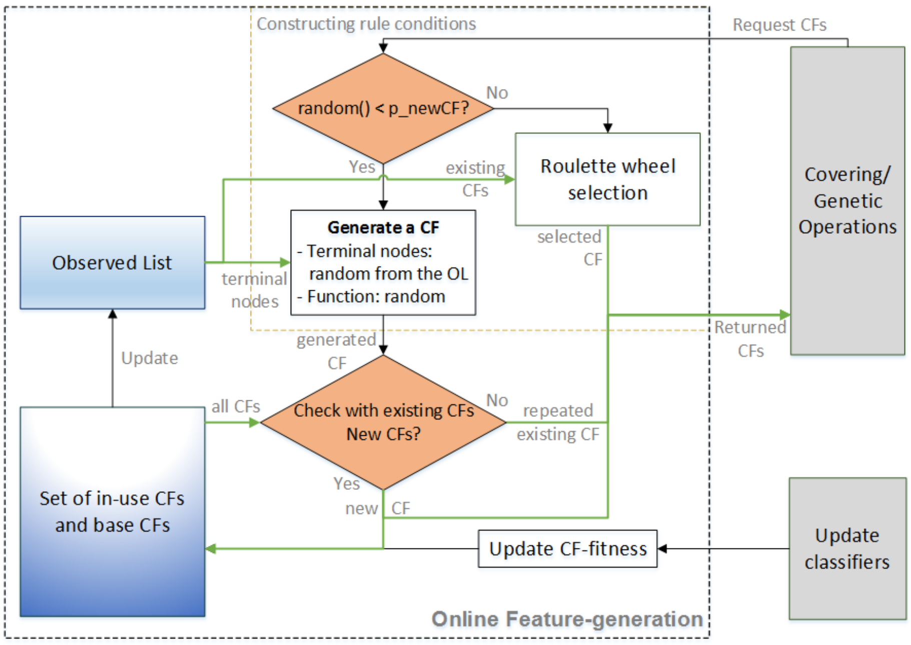


Figure 4.2: Online Feature-generation.

The OF includes the evolution of CFs and interacts with classifiers/rules in two processes. First, the OF returns a CF when covering or mutation requests a new CF for a rule condition. Second, the CFs were updated when the classifiers' fitness was updated. While information flows from the CF evolution to the rule evolution in the first interaction, the second interaction is to forward information oppositely.

4.1.2.1 CF-fitness: The Applicability of CFs

A new parameter, called CF-fitness (the fitness of CF)s, is the key to connect between the CF evolution and the rule evolution. It is designed to describe the applicability of CFs. The applicability of a CF is its ability to produce high-fitness classifiers by constituting their conditions. A high-fitness classifier in XCS learning paradigm is also an accurate and general rule, which can cover a big niche (with many instances) [24]. Thus, CF-fitness can describe the ability of CFs to produce accurate and generalised classifiers. With this assertion, it can bootstrap covering and reproduction processes, and updating the OL, see Section 4.1.2.2. The CFs with high CF-fitness will be prioritised to be selected in classifier conditions in covering and mutation.

In this work, a method for estimating the applicability of CFs is developed to define CF-fitness based on the fitness of classifier(s) containing the CF. CF-fitness can be estimated indirectly from the environment response through the classifier fitness as follows:

$$cf.cf\_fitness := \frac{cl.fitness}{\text{length of } cl.condition}, \tag{4.1}$$

where cf is a CF in $cl.condition$, and the length of a rule condition is the number of CFs within it.

There are several theoretical reasons for this estimation. To begin with, being correlated with the classifier fitness means CF-fitness correlates with the accuracy, the numerosity and size of the niche covered by the classifier [23]. Therefore, this estimation could provide insight into the applicability of CFs. Furthermore, this estimation favours CFs constructing classifiers with shorter condition lengths. As a result, accurate classifiers with short rule conditions that can cover large niche are supposed to contain highly discriminative features, or CFs with high CF-fitness. This is desirable in our systems as the estimation creates an innate tendency to shorten classifiers and growing towards higher-level features that are more applicable to the target problem.

A useful high-level CF can not always guarantee that all classifiers containing it have high fitness because the CF might be misused by randomly combining with other incompatible CFs in rule conditions. Therefore, two different ways of updating CF-fitness are proposed and discussed in the following subsections. Both methods update the CF-fitness using the Widrow-Hoff learning rule [120] with a learning rate β_{cf}. β_{cf} is smaller than the learning rate β for other parameters as the CF-fitness is more frequently updated. The initial CF-fitness is 0.01 for base CFs. The base CFs need non-zero CF-fitness in the beginning since they are initially assumed to have common small applicability. Also, this is required for Roulette Wheel selection (Figure 4.2) to advance in the beginning of the learning process. Meanwhile, the initial CF-fitness of constructed CFs simply starts from the value of 0. Because constructed CFs appear in at least one rule condition as they are created in covering and mutation, their CF-fitness will be lifted from 0 as soon as the classifier containing it updates.

Update CF-fitness by Promising Classifier's Fitness

This approach allows all "promising" classifiers containing a CF to update the CF-fitness of the CF. As the update mechanism relies on Promising-classifiers' Fitness (PF), the system using it is abbreviated as XOF-PF. A promising classifier means that it is "nearly" an accurate classifier. Specifically, its error is no greater than $\epsilon_0 = 10$ (max reward is 1000) and its experience is at least than $\theta_{cf} = 10$. These values are more relaxed than corresponding ones for subsumers [26], which enable picking up promising classifiers earlier. Each time a classifier that satisfies the "promising criteria" updates its fitness, it will also update the CF-fitness of all CFs contained in the condition of the classifier.

Update CF-fitness by Best Classifier's Fitness

The CF-fitness can be updated using just the containing classifier with the highest fitness. This is designed to estimate the CF-fitness at the best known performance of the CF. As it uses the fitness of the classifier with the highest fitness to update CF-fitness, it is called Best-classifier's Fitness (BF) and the whole system is abbreviated as XOF-BF. This strategy is theoretically expected to preserve CFs that are only good in small niches. The update method of classifier fitness always checks whether the fitness of a classifier has become greater than the fitness of best classifiers of CFs contained in the classifier.

Criteria are required for a classifier to be a possible source of updating CF-fitness because bad combinations of useful CFs may occur before good classifiers are created. However, as only the best classifier is used, the criteria can be relaxed to bootstrap the process of updating CFs. Experiments show that following thresholds, $\epsilon_0 = 10$ and $\theta_{cf} = 100$, result in good performances in most problems. These values are used in the experiments in Section 4.1.3.

4.1.2.2 The Observed List of CFs

The Observed List (OL) is a subset of the CF population (more details in 4.1.2.3) and is designed to contain the most applicable CFs based on their CF-fitness. The main contribution of the OL is to focus the search for useful high-level CFs. Specifically, the system grows high-level CFs by combining CFs from the OL and add newly constructed CFs to the CF population. The update mechanism of the OL is expected to add high CF-fitness CFs to the OL and remove low CF-fitness CFs from it. The OL is limited in size with the original purpose of avoiding a too large search space of new CFs. However, the growth of the OL is slow enough not to have adverse influence on the learning performance. Hence, the size limit (N_{OL}) is ranged from 3 to 10 times the length of data inputs as these are sufficient to avoid removal of CFs from the OL before reaching 100%.

Updating the OL

The OL is updated every $\theta_{OL} = 500$ iterations by the Algorithm 4.1. θ_{OL} is chosen empirically large to avoid too frequent changes in the search space of useful CFs. In each update, only one CF can be added to the OL. If the size of the OL exceeds N_{OL}, the lowest CF-fitness CF is removed from the OL before adding a new one. A tournament selection method will choose a CF based on the CF-fitness to add to the OL. The OL starts with the base CFs of all the original data features. Though they initially have no appearance in promising classifiers, the starting OL enables the learning process to advance in the beginning. Thus, the CF-fitness of all CFs is padded with 1 to allow roulette wheel selection to function.

4.1.2.3 Managing CFs

As described in Section 4.1.2, a population of CFs is maintained to track all in-use CFs without the root node function NOT and the base CFs representing the features of environment states. Every negated version of a CF, which is the CF added with a root function NOT, and the CF itself are linked with each other and share the same parameters. The base CFs with depth zero are the initial elements of the CF population and the OL at the beginning of the learning process. They are kept throughout the learning process to allow the learning system to recover from local optima although the evolution of rules might eliminate all rules containing (some of) them. Conversely, an online created CF can be entirely deleted from the CF pop-

Algorithm 4.1 Updating the observed list. The $unobserved\_cfs$ are the CFs in the CF population that is not in the OL.

```
List all positive CF-fitness CFs not contained in the OL
Rank them by descending order of CF-fitness in an $unobserved\_cfs$
Rank all the remaining CFs in the OL in descending order of CF-fitness in $ranked\_ol$
**if** $size(OL) >= N_{OL}$ **then**
    Remove last item of $ranked\_ol$ from the OL and add it to $unobserved\_cfs$
**if** $size(unobserved\_cfs) > 0$ **then**
    Select 1 CF $cf_{newOL}$ from $unobserved\_cfs$ using tournament selection based on the CF-fitness
    **for** $cf$ in the OL **do**
        **if** $cf$ can subsume $cf_{newOL}$ **then**
            Add $cf_{newOL}$ to $cf.subsumees$
        **else if** $cf_{newOL}$ subsumes $cf$ **then**
            Add $cf$ to $cf_{newOL}.subsumees$
    add $cf_{newOL}$ to the OL
```

ulation and the OL once neither itself nor its negated CF is used in any rule. Each newly created CF will be compared with all existing CFs in the population. This, combined with the methods in comparing and generating CFs, can mostly avoid adding CFs with the same structure, including swapped inputs, more than once to the CF population.

Avoiding repeated CFs makes computing CFs and updating CF-fitness of CFs more efficient. All in-use CFs are computed once with the results saved afterwards. The results can be reused for computing CFs that have the computed CFs as branches. This saves the system from recomputing CFs in different rules or processes of the learning cycle. Also, estimating CF-fitness of a CF is more robust and centralised on the CF as almost all of its appearances in rule conditions are used to update the CF-fitness.

Comparison between CFs

There are two types of comparisons between CFs. One is the comparison between two CFs that have already been added to the CF population. This case happens in crossover and classifier subsumption. The other is to compare two CFs where at least one of them is newly generated and thus has not been added to the CF population. In both cases, OF enables highly accurate comparisons between two CFs with little computation effort (see exception outlined below). For the first case, checking identity (i.e. a Python reference in this implementation) of CFs is sufficient since all CFs added to the CF population are checked for equality. In the second case, the comparison is to check only the top layer of two CFs. Comparison returns equal if two CFs have the same root function and the same set of inputs. This becomes trivial since the OF allows adding only one layer on top of existing CFs in generating new CFs. There is a chance that the second case can return "not equal" while two CFs are actually the same. It can happen when an existing CF (cf_{old}) is already in the CF population, and a CF representing a branch of the cf_{old} was deleted in advance. The lower-level or base features might again grow to replicate the deleted branch, thus duplicating cf_{old}. The efficiency of these comparisons is verified in our experiments as there are hardly any repeated CFs in the final CF population.

Subsumption between two CFs

The possibility of subsumption is a benefit of using the XOF. It is used to remove redundant CFs in covering and genetic operations. The subsumption between two CFs is defined in the sense of comparing the sets of matched instances. A CF cf_a can subsume another CF cf_b if the set of instances matched cf_a is a subset of cf_b. This means that when cf_a outputs

1 ($True$), the cf_b will always output 1. Therefore, in a rule condition, if cf_a exists, cf_b is redundant and can be safely removed from the rule condition to avoid inefficient structures.

To check for subsumption, XOF utilises the OL as the equivalence to the data attributes in XCS [26]. Since, on request, the OL will either select an existing CF or generate a new CF with only one layer added, almost all CFs in the CF population, except for the incompetent ones, can be represented by converted ternary rules based on the OL. These converted rules are produced by Algorithm 4.2. Based on the CF rules, instead of iterating through all instances to compare the sets of matched instances, the subsumption of two CFs becomes comparing two sets of converted ternary rules. Also, comparing converted rules simply resembles to comparing classifiers in the classifier subsumption.

Algorithm 4.2 Generating ternary rules based on the OL for cf_i

```
1: if $cf_i.function = NOT$ then
2:     Back to step 1 to find rules for $cf_i.terminals[0]$, toggle all output of rules found
3: if $cf_i$ in the OL then
4:     $id$ := index of $cf_i$ in the OL
5:     return rule: $(...\#1\#...|1)$, condition 1 at $id$
6: else
7:     if both terminals of $cf_i$ in the OL then
8:         create rules based on the ruleset of $cf_i.function$
9:     else
10:        return no rule (can't subsume)
```

Constructing New CFs

XOF requests CFs in covering and mutation. In this implementation, the CF population can return an existing CF from the OL by roulette wheel selection based on the CF-fitness or generate a new CF with probability

$p_{newCF} = 0.1$. The small probability 0.1 for generating new CFs is empirically selected to stabilise the search space of CFs because this probability can balance the learning of existing CFs and creating new ones in most problems. Creating too many new CFs and using them instead of existing CFs may prevent discovering the best combination of existing CFs, which result in a poor evolution of CFs. In contrast, if this probability is too small, the growth of constructed CFs will be too slow.

The proposed systems exploit CF-fitness with roulette wheel selection of two sub-nodes cf_0 and cf_1 in the first step of generating new CFs. The generating process is described in Algorithm 4.3. A new CF is composed of a function randomly selected from a fixed function set including only three binary operators $S_f = \{AND, OR, XOR\}$ and two CFs selected from the OL. Depths of CFs are accumulated through layer growth. When traversing from the root node to leaf nodes of a CF, each function except for NOT will increase the depth of the CF by 1.

The random-negation method generates CFs by combining the strategies of using the simplified f_s and randomly adding function NOT. These strategies work in harmony to cover all possible structures of CFs, including ones equivalent to using $\{\text{NAND}, \text{NOR}, \text{NOT-XOR}\}$. Furthermore, this method enables more efficient generating of CFs. Firstly, the use of the CF population without the root node function NOT saves the system from creating CFs with repeated NOT functions, a source of redundancy in existing CF-based XCSs [3, 60]. Also, toggling output by adding NOT to satisfy the current environment state prevents the process from having to repeat until getting output 1 [60]. Lastly, since the function set f_s has only three functions, the search space of CFs was divided into 2 coupled spaces, one having CFs without root function NOT saved in the CF population and one containing CFs with root function NOT. Moreover, a CF and its negated CF share the same parameters. As a result, the method explicitly reduces the search space size of CFs by a half.

Algorithm 4.3 Generating a new CF given state s and existing CFs in the condition $cl.condition$ using random negation.

```
1: roulette wheel selection of $cf_0$ and $cf_1$ from the OL
2: if $cf_0.depth = max\_depth$ or $cf_1.level = max\_depth$ then
3:     Back to step 1
4: if $random[0,1) < 0.5$ then
5:     $cf_0 := !cf_0$
6: if $random[0,1) < 0.5$ then
7:     $cf_1 := !cf_1$
8: Randomly select a function $f_i$ from the function set $S_f$
9: Create a CF $cf_{new}$ by the selected function and CFs
10: $cf_{new}.depth = max(cf_0.depth, cf_1.depth) + f_i.level$
11: Compute output value $val$ with given state $s$
12: if $val = 0$ then
13:     $cf_{new} := !cf_{new}$
14: for $cf_{existing}$ in the $cl.condition$ do
15:     if $cf_{new}$ = $cf_{existing}$ then
16:         Back to step 1
17: for $cf_{existing}$ in the $cl.condition$ do
18:     if $cf_{new}$ subsumes $cf_{existing}$ then
19:         remove $cf_{existing}$ from $cl.condition$
20:     else if $cf_{existing}$ subsumes $cf_{new}$ then
21:         return no CF
22: return $cf_{new}$
```

4.1.2.4 Genetic Operations

Since the representation of CFs decouples the positions of CFs in classifier condition, the fixed length classifier condition and positional processes like GA operators [60, 63] are not necessarily appropriate. XOF no longer needs built-in "don't care" CFs as it allows rule conditions to have varied lengths. The probability of specificity p_{spec} is used similarly with standard XCS in covering. If the random number is smaller than p_{spec}, the system requests a CF. Otherwise, no CF is added. A maximum length of classifiers is generally set to be twice the number of input data bits to encourage the flexibility in condition length.

The crossover and mutation operators [63, 141] in existing CF-based XCSs are not applicable for variable-length conditions. Instead, the proposed crossover swaps two randomly selected CFs from parent conditions. The offspring are then mutated by a mutation process inspired by mutation operator of GP, depicted in Algorithm 4.4. Our experiments suggested that a high probability of mutation $\mu = 0.9$ and low crossover rate $\chi = [0.1, 0.3]$ can accelerate the performance convergence of the system. This is anticipated as mutation can exploit the useful data from CF-fitness. Using high mutation rate is expected as this operation promotes the evolution of CFs. Also, crossover by swapping CFs does not contribute much to the learning process because the CF-base rule conditions do not preserve the attribute positions as in ternary alphabet representation. The final step of genetic operations is to remove redundant CFs by checking subsumption between CFs within each offspring condition.

Algorithm 4.4 Mutation operation on one offspring cl.

```
1: if $random[0,1) < \mu$ then
2:     randomly select a CF $cf_i$ at $p_{mutate} = [random[0,1) \times length(cl.condition)]$
3:     if $random[0,1) < 0.5$ then
4:         remove $cf_i$ from condition of this $cl$
5:     if $random[0,1) < 0.5$ then
6:         add another CF to this $cl.condition$ at $p_{mutate}$
7:     if $random[0,1) < 0.5$ then
8:         toggle $cl.action$
```

4.1.3 Experimental Setups and Results

XOF-PF and XOF-BF are compared with XCS and XCSCFC on five Boolean problem domains to compare their performances with corresponding results of XCSCFC as well as standard XCS: Multiplexer, Majority-on, Even-Parity, Hierarchical Multiplexer and Hierarchical Majority-on. The performances on the 70-bit Multiplexer problem are averaged from 10 runs only due to time consumption. All the tested approaches were implemented in Python except for XCSCFC in C++. Therefore, time consumption can only be direct comparisons among XCS and XOF systems, although it is reported that XCSCFC consumes more time than XCS for the same problem and settings when they were both implemented in C++ [4]. XOF-BF is also tested with several real-world datasets, i.e. UCI Zoo, UCI SPECT heart and UCI Phishing Websites datasets. These datasets are chosen as they have binary or nominal attributes, with a range of number of features, classes and instances. The details of datasets can be seen from Chapter 3.

4.1.3.1 Experimental Design

XCS is configured with subsumption in GA only, except for the Majority-on problems. In this domain, all subsumption is disabled to reduce over-general rules enable higher performance. XOFs (XOF-PF and XOF-BF) use almost the same settings for the XCS part, except for the crossover rate $\chi = 0.2$ and the mutation rate $\mu = 0.9$ (see Section 4.1.2.4 for justification). XOFs also share many other settings. The learning rate of CFs $\beta_{cf} = 0.001$ for XOF-PF and XOF-BF is chosen empirically to balance the evolution of rules and CFs in most problems. If this rate is higher, the dynamics of constructing CFs becomes faster, which could result in unstable CF-fitness and hurt the performance of rules. Maximum condition length is twice the number of data attributes and is only equal to that number in 70-bit Multiplexer problem. Maximum depth for CFs is five layers. The only subsumption in the genetic operations is enabled with very strict criteria:

$\theta_{sub} = 50$ and $\epsilon_{sub} = 0.0001$ (error threshold for subsumption) as the effect of subsumption is not investigated in this work. The maximum sizes of the OL in all experiments are varied from 3 to 10 times the number of data attributes as dependent on the problem domain. The criteria for classifiers to be a source for CF-fitness of CFs within the classifier conditions use the same $\theta_{sub} = 10$ and different ϵ_{cf} with 10 for XOF-PF and 100 for XOF-BF. The population sizes are equal for all approaches in each experiment.

The experiments for real-world problems are executed in supervised learning. XCS-based systems are online-learning algorithms and not truly designed for supervised learning. To investigate the performance in supervised learning, XOF can only access the instances from the training set during training with enabled exploration. Performance is reported on the result of XOF, which are configured to disable exploration and rule updates, on the separated testing set. For Zoo and SPECT heart datasets, the performance of all tested algorithms is determined by 10-fold cross-validation. UCI Zoo and UCI SPECT heart are both experimented using 10-fold cross validation for all approaches. XOF-BF will be compared with XCS and other popular machine learning algorithms. Among them, results of Random Forest are reported with the batch size of 200, which is generally the best result among several batch sizes tested.

4.1.3.2 Results on Multiplexer Problems

The Multiplexer domain requires learning systems to prioritise features related to address bits over data bits because CFs for address attributes generally can be reused in different niches. XOF-PF and XOF-BF learn much faster than other approaches in both problems. In the 37-bit Multiplexer problem, the numbers of iterations needed for the two approaches to reach 100% accuracy in most runs are less than $400{,}000$ iterations (Figure 4.3a). Also, all runs solve the problem in $1{,}000{,}000$ iterations. In 70-bit Multiplexer problem, XOF-BF is slightly better than XOF-PF since all runs of XOF-BF reach 100% as shown in Figure 4.3b. Both the final accuracies of XOF-PF and XOF-BF are higher than others and statistically significant based on the Wilcoxon signed rank test with $p{-}value < 0.05$ (see Table 4.1).

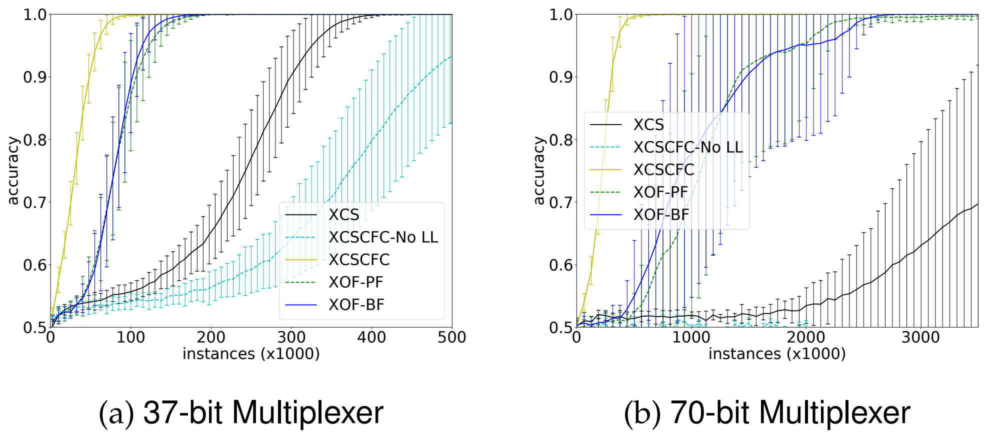
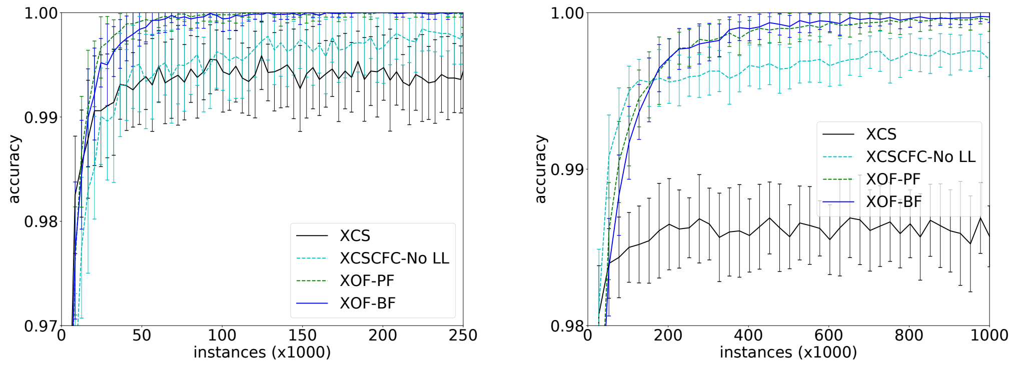


Figure 4.3: Performances on Multiplexer problems.

4.1.3.3 Results on Majority-on Problems

In this problem domain, useful CFs in one niche can have a high chance of being useful in other niches. Therefore, using CF-fitness can be beneficial in the learning performance. Figures 4.4 show the learning performance of the tested systems. XCS in general settings is discussed to have limited performances on overlapped problems and thereby has lowest performances in Majority-on domain [58]. XCSCFC is better than XCS because of its inherent redundancy. Without such redundancy, however, all the XOFs have relatively equivalent performances and end up at higher accuracies than XCS and XCSCFC.



(a) 9-bit Majority-on (b) 11-bit Majority-on

Figure 4.4: Performances on Majority-on problems.

4.1.3.4 Results on Even-Parity Problems

The Even-Parity domain can be solved by only one straight decision boundary, i.e. two tree-based correct XCS′ classifiers (prediction 1000), if any one of the optimal high-level features is found (see Figure 4.9). It is noted that generalisation is not possible in the ternary alphabet. This means that multiple niches are gradually replaced by more general accurate-rules during the evolution of CFs. When finding the optimal high-level feature, CFs can be reused among niches. XCS cannot learn these problems even though $p_{spec} = 1.0$ because XCS needs large population sizes and the mutation will flip random specified bits to create over-general rules. This problem domain also returns intractable search spaces to XCSCFC when learning without transfer learning.

In both problems, XOF-PF lags behind XOF-BF in the later learning phase of 11-bit problem because one run did not progress as it fluctuated around 50% accuracy (see Figure 4.5). The accuracy of XOF-BF on the 13-bit problem is significantly higher than others (see Table 4.1).

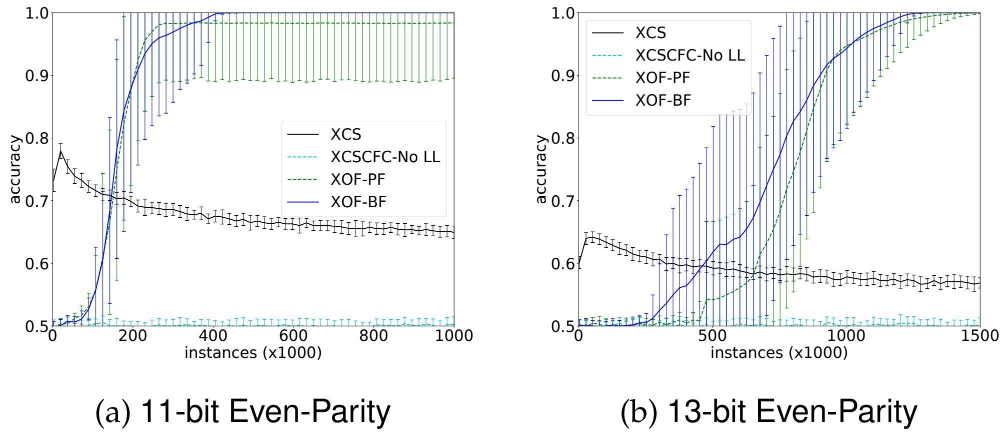


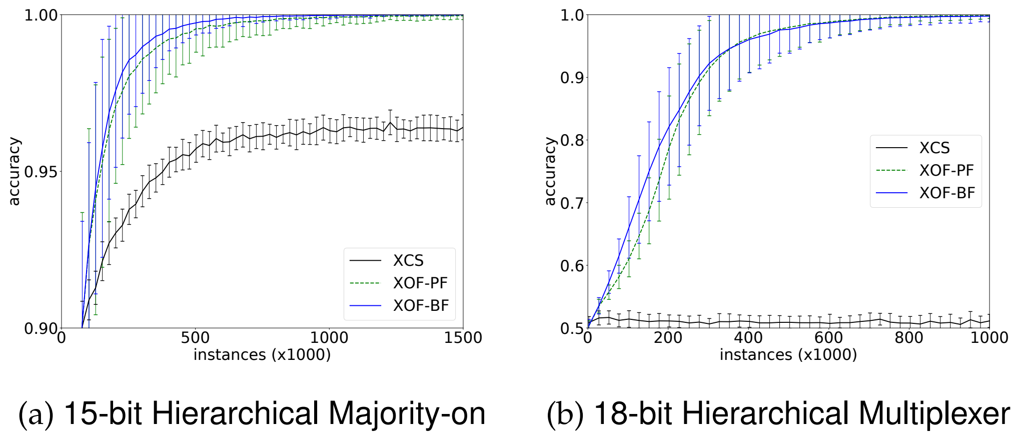
Figure 4.5: Performances on Even-Parity problems.

4.1.3.5 Results on Hierarchical Problems

Tested problems are 15-bit Hierarchical Majority-on and 18-bit Hierarchical Multiplexer problems. When dealing with hierarchical problems, the approaches should be able to extract useful high-level CFs to achieve simple and tractable decision boundaries. XOF-PF and XOF-BF are shown in Figure 4.6 to be efficient in generating not only high-level but also applicable CFs. Most runs of XOF-BF reach 100% accuracy for 15-bit Hierarchical Majority-on and 18-bit Hierarchical Multiplexer problems ($29/30$ and $24/30$ respectively). The two systems reach the highest accuracies that are statistical significance compared with XCS and XOF (see Table 4.1). Optimal features representing lower-level Even-Parity bit-chunks are generally discovered among runs of both the approaches, as the first two CFs shown in Figure 4.7.

4.1.3.6 Time Consumptions by XOFs

XOF-PF and XOF-BF both consumed remarkably less time to finish all the enabled iterations than XCS in many problems, except for 18-bit Hierarchical Multiplexer and 13-bit Even-Parity problems, see Table 4.2. This is be-



(a) 15-bit Hierarchical Majority-on (b) 18-bit Hierarchical Multiplexer

Figure 4.6: Performances on Hierarchical problems.

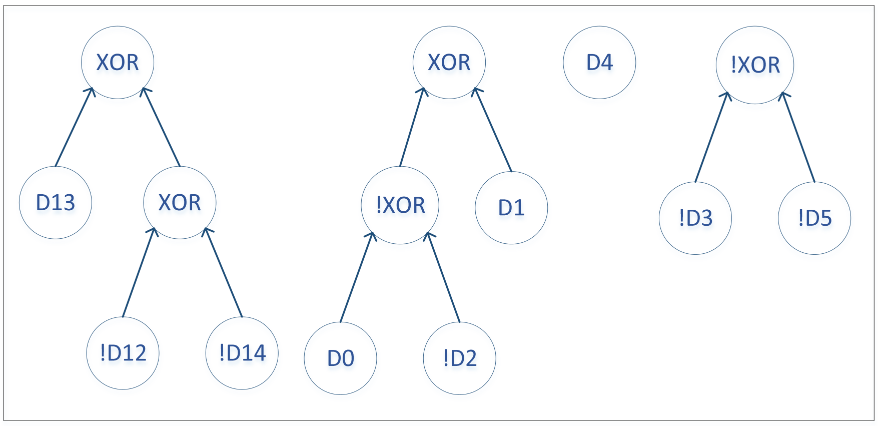


Figure 4.7: A sample classifier with highest numerosity (349) in one run of XOF-BF for 18-bit Hierarchical Multiplexer: action 0, prediction 1000, error 0, fitness 0.462.

yond our expectation as these two algorithms use tree-based programs for feature construction. In terms of time for reaching 100% accuracy, XOF-PF and XOF-BF further eclipse XCS. Time consumption for XOF also demonstrates that XOF can run more quickly compared with XCS in the majority of experiments. In conclusion, the extension OF alone can greatly optimise

Table 4.1: Accuracy comparisons on the Boolean problems. The results of XOF-PF and XOF-BF are bold if they have statistically significant differences from other approaches (not from each other) based on The Wilcoxon signed rank test with $p{-}value < 0.05$. $p{-}value$ is corrected using Bonferroni correction.

Problems	XCS	XCSCFC	XOF-PF	XOF-BF
37-bit Multiplexer	100%	$93.0\pm$ 10.6%	100%	100%
70-bit Multiplexer	$70.0\pm$ 21.9%	n/a	$\mathbf{99.72\pm0.44\%}$	$\mathbf{100\%}$
9-bit Majority-on	$99.4\pm$ 0.18%	$99.8\pm$ 0.15%	$99.99\pm0.04\%$	$99.997\pm0.018\%$
11-bit Majority-on	$98.6\pm$ 0.14%	$99.7\pm$ 0.08%	$99.96\pm0.04\%$	$99.98\pm0.03\%$
11-bit Even-Parity	$65.0\pm$ 0.6%	$49.9\pm$ 0.5%	$\mathbf{98.31\pm9.26\%}$	$\mathbf{100\%}$
13-bit Even-Parity	$56.9\pm$ 0.5%	$50.1\pm$ 0.5%	$\mathbf{99.97\pm0.14\%}$	$\mathbf{100\%}$
15-bit H.Majority-on	$96.39\pm$ 0.25%	n/a	$\mathbf{99.97\pm0.10\%}$	$\mathbf{99.998\pm0.010\%}$
18-bit H.Multiplexer	$50.9\pm$ 0.7%	n/a	$\mathbf{99.86\pm0.52\%}$	$\mathbf{99.76\pm0.90\%}$

the computation time in XOFs as discussed below.

XOF-BF had more stable performance across 30 runs of all experiments. Therefore, only XOF-BF was used in later experiments on real-world datasets.

4.1.3.7 Results on Real-world Problems

XOF-BF is compared with well-known approaches in three benchmarks to evaluate the ability of XOF-BF in real-world classification tasks. It performs well on small data set like UCI Zoo but falls behind others on UCI

Table 4.2: Average time consumptions in minutes of the tested approaches. Each approach is measured with running time (RT) and solving time (ST). Solving time is the duration when all runs can reach 100% accuracy within the maximum iterations. XCSCFC is not listed as it is implemented in C++ while others are implemented in Python.

Problems	**XCS**		**XOF-PF**		**XOF-BF**	
	RT	ST	RT	ST	RT	ST
37-bit Multiplexer	34.52	26.90	34.35	18.67	33.46	17.66
70-bit Multiplexer	4821.	n/a	3683.	n/a	3552.	2932.
9-bit Majority-on	11.57	n/a	11.18	n/a	10.67	n/a
11-bit Majority-on	143.3	n/a	96.38	n/a	91.95	n/a
11-bit Even-Parity	71.88	n/a	58.2	n/a	59.8	45.0
13-bit Even-Parity	156.9	n/a	340.8	n/a	354.4	329.6
15-bit Hierarchical Majority-on	422.7	n/a	224.3	n/a	209.2	n/a
18-bit Hierarchical Multiplexer	98.98	n/a	259.5	n/a	259.0	n/a

Table 4.3: Results on real-world problems in supervised learning mode.

Algorithms	**Zoo**	**SPECT**	**Phishing**
XCS	$96.8 \pm 1.3\%$	$80.15 \pm 0.90\%$	$95.42 \pm .21\%$
Naïve Bayes	95.05%	76.0%	94.06%
SVM	92.08%	83.1%	95.50%
MLP	$95.91 \pm 0.42\%$	$80.1 \pm 1.16\%$	$98.17 \pm 0.09\%$
C4.5	92.08%	82.4%	97.2%
Random Forest	$96.07 \pm 0.65\%$	$83.12 \pm 0.53\%$	$98.43 \pm 0.26\%$
XOF-BF	$97.72 \pm 0.70\%$	$81.65 \pm 1.45\%$	$95.14 \pm 0.22\%$

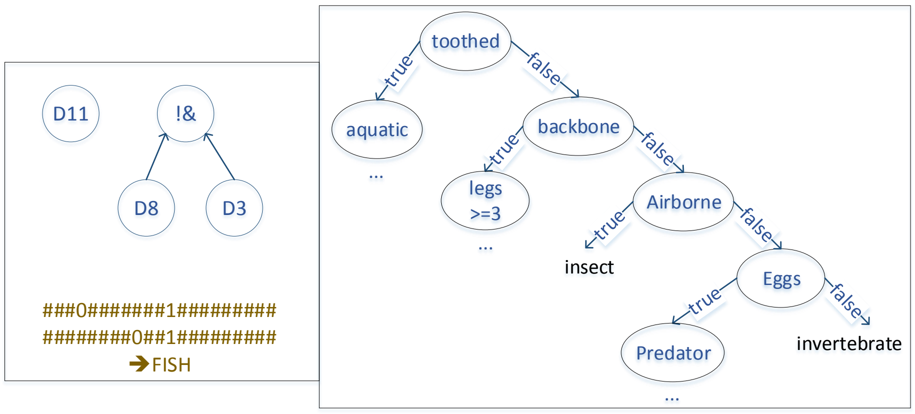


Figure 4.8: A sample rule learned by XOF-BF (left) and a sample tree by Random Forests on UCI Zoo dataset (right). D_{11} is fins, D_3 is milk, D_8 is breathes.

SPECT and UCI Phishing websites data sets, especially Random Forests, see Table 4.3. This could be due to the fact that XCS-based systems were designed for online learning instead of supervised learning, where algorithms like Random Forests can look at a batch of many instances or the whole training set at once to learn about the statistics of training data. However, the learned knowledge by Random Forests is harder to read as they are larger in tree size and require scanning all trees to determine the whole rule. On the contrary, the knowledge provided by XOF-BF is generally more straightforward to understand. This is due to that XOF-BF's rules only cover their own niches instead of being used for any instance as in Random Forests. This enables solving problems with a limited maximum tree depth of five layers. Compared with genetic programming trees, rules produced by XOF-BF are also simpler (Figure 4.8) for the same reason.

4.1.4 Discussion

The OF and the new CF-fitness contribute to the computation reduction of the two XOFs. First, XOF can reuse the computed outputs of lower-level CFs to compute higher-level CFs that contain the computed CFs as branches. Second, growing CFs around the OL enables subsumption of classifiers, subsumption of CFs, the comparison between CFs and updating CF-fitness to be low cost in computation. Lastly, new CF-fitness improves the convergence of accuracy, and thereby reduces the time consumed by XOFs. In general, XOF-BF has slightly better performances than XOF-PF. The CF-fitness estimated by the fitness of the best-containing classifier would reflect better the applicability of CFs. XOF-BF also consumes a little less time than XOF-PF since the update of CF-fitness only occurs with the best-containing classifier.

The results of XCSCFC in the experiments were not good as the support from transfer learning or layered learning is mandatory for XCSCFC to achieve a good performance in difficult problems.

4.1.5 Summary of the Initial Implementations of XOF

This work introduces the Online Feature-generation module for XCS with CF-conditions that enables growing high-level CFs without the need of a sequence of learning stages. The new system (XOF) evolves a CF population in addition to the rule population. The two evolutionary processes support each other. The rule evolution translates the environmental feedback into the feedback for CFs in the form of rule fitness. On the contrary, CF-fitness improves the rule evolution by assisting rule construction in prioritising CFs with high CF-fitness. XOF successfully discovered high-level patterns in the tested Boolean problems as well as real-world problems despite of the large search spaces of CFs. High-level patterns are constructed by growing from CFs in the OL. This strategy in XOF-BF does not lose the necessary building blocks, which results in higher accuracy than XCS and other approaches in tested problems.

Two methods of estimating the CF-fitness were developed to conquer the large search spaces of tree-based features. The proposed systems, XOF-PF and XOF-BF, learn large-scale and hierarchical problems in fewer iterations than XCS and XCSCFC, which search in the search spaces of original data features and tree-based features respectively using equivalent GA search techniques. The new CF-fitness successfully combine local search of applicable features and global search of the rule-based population.

Even though using tree-based features, both implementations of XOF consume less time than XCS to solve problems in most benchmarks. They direct the learning process effectively to reach 100% accuracy more quickly. Overall, XOF-BF is the most efficient approach. It runs the most efficiently in terms of both time and algorithmic iterations in most tested problems, but still achieves the best performances. In conclusion, XOF-PF and XOF-BF succeed in assigning the new CF-fitness with the applicability of CF features to provide rules with rich discriminative information.

XOF can also be considered as a framework to grow high-level features, which can be in any flexible representation instead of tree-based programs. Although the combination operation to create higher-level features might need an adaptation to the representation, other process and especially the fitness to estimate the applicability of constructed features could be generalised with ease. This opens the opportunity to work with other problems using appropriate representations.

In the next section, constructed CFs are analysed in terms of their efficiency in encapsulating the underlying patterns of the problem with minimal structural complexity. This goal motivated the addition of a niching property to the evolution of CFs. This property was not covered in this section, although this niching is a unique feature of XCS that distinguishes it from other machine learning approaches.

4.2 Niching Method for CFs and CF-fitness for Structural Efficiency

A major goal of machine learning is to create techniques that abstract away irrelevant information. The generalisation property of standard Learning Classifier Systems (LCSs) removes such information at the feature level but not at the feature interaction level. CFs introduced feature manipulation to discover important interactions, but they often contain irrelevant information, which causes structural inefficiency. XOF, introduced in the previous section, uses CFs to encode building blocks of knowledge about feature interaction. This section aims to optimise the structural efficiency of CFs in XOF. Two measures to improve constructing CFs are proposed to achieve this goal. Firstly, a new CF-fitness update estimates the applicability of CFs to the problem while also considering the structural complexity. The second measure is a niche-based method for generating CFs.

These approaches were tested on Even-parity and Hierarchical problems as they require highly complex combinations of input features to capture the data patterns. The results show that the proposed methods significantly increase the structural efficiency of CFs, the ability to capture generalised patterns using minimal tree structures. This results in faster learning performance in the Hierarchical Majority-on problem. Furthermore, a user-set depth limit for CF generation is not needed as the learning system will not adopt higher-level CFs once optimal CFs are constructed.

4.2.1 Introduction

It is considered that abstracting away irrelevant information improves the explainability of complex learned knowledge. A popular representation for encoding knowledge that encourages the explainability of learned knowledge is tree-based programs, such as Genetic Programming trees [74]. However, the problem of bloat in learned trees inhibits their explainability and hurts the performance of the system [87]. It can be inefficient computationally and disrupts rule discovery by including poor building blocks of knowledge in recombination operators. In case of continual learning [125] or layered learning [118], the accumulated knowledge can suffer from exponentially increasing inefficiency in complex trees as the learning system continues to deal with more and more complex problems.

An XOF is a system that grows high-level CFs based on a set of the most applicable CFs that are included in an Observed List (OL). XOF can learn hierarchical and large-scale problems by capturing the data patterns in CFs. However, its constructed trees also contain bloat due to the panmictic crossover of CFs in the OL. The general learning process of an XOF, when addressing a hierarchical problem, is to generalise from small niches, i.e. some specific cases, to larger niches by combining the building blocks from the small niches. Figure 4.9 illustrates the relationship between the lower-level CFs in less generalised rules and the higher-level CFs in more generalised rules that can replace all the more specific rules. The higher-level CFs here are shown to combine the lower-level CFs to create CFs that can capture a superset niche. This heuristic suggests that high-level (abstract) features can be generated by combining local features from one of more specific cases (smaller niches). Hence, a niching method for CFs can be beneficial for the generalising process of XOF.

Niching of classifiers is a unique advantage of LCSs. Previous implementations of XOF have not included any niching property for constructing CFs. All CFs in the OL and in the CF population are grouped together without any discrimination among niches. If the information that a CF in the OL performs the best in the current niche or another niche is available, the learning system can avoid combining CFs from unrelated niches, which was the likely cause of the non-optimal trees with bloat. It is hypothesised that prioritising the most applicable CFs in a niche in covering and genetic operations can be beneficial to the learning performance as

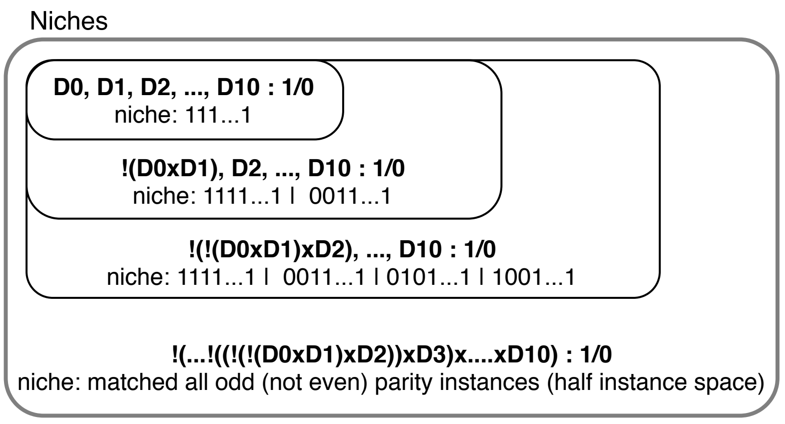

Niches
D0, D1, D2, ..., D10 : 1/0
niche: 111...1
!(D0xD1), D2, ..., D10 : 1/0
niche: 1111...1 | 0011...1
!(!(D0xD1)xD2), ..., D10 : 1/0
niche: 1111...1 | 0011...1 | 0101...1 | 1001...1
!(...!((!(!(D0xD1)xD2))xD3)x....xD10) : 1/0
niche: matched all odd (not even) parity instances (half instance space)

Figure 4.9: An example of generalising from a smallest niche to the largest one in a hierarchical problem (11-bit Even-parity problem) by XOF. $\times$ is the abbreviation for XOR. ','s separate CFs in a condition:action binary classifier effecting 1 or 0. ! is NOT. The generalising process benefits from growing from lowest-level CFs ($D_0, D_1, ..., D_{10}$) to more complex ones gradually, i.e. $(D_0 \times D_1)$, $(!(D_0 \times D_1) \times D_2)$, ..., $(!(...!(!(!(D_0 \times D_1) \times D_2) \times D_3) \times ... \times D_{10}))$.

well avoiding bloat in constructed CFs.

This work proposes two novel methods to combat bloat in CFs. Accordingly, the objectives of this section are as follows:

1. To develop a new generalised CF-fitness measure, which is to estimate the applicability of CFs in creating high-fitness classifiers, to emphasise the efficiency of CF structures. This objective includes investigating the criteria based on the structural efficiency of CFs to select the most applicable CFs to the task accordingly.

2. To introduce a niche-based method for adjusting CF-fitness of CFs in the OL.

3. To introduce a new niche-based method for updating the OL that can collect necessary building blocks (CFs) from all niches.

4. To investigate the influence of the new CF-fitness and the two niche-based methods for CFs on the structural efficiency of constructed CFs and the learning performances of XOF.

5. To analyse further the influence of the CF evolution on the learning performance of XOF.

The system will be tested on complex problems that require building hierarchical features to capture the patterns of data. Benchmark problems include Even-parity, Hierarchical Multiplexer, and Hierarchical Majority-on problems [23] because these Boolean problems require accurate hierarchical combinations of input attributes. These combinations must match with the data patterns of these problems to carry the maximal discriminative information of the problems. As a result, constructing such combinations can reduce the search space of rules in XOF. However, finding accurate complex combinations in CFs is challenging due to the large search space.

4.2.2 Niche-based CF-fitness using Rule-Fitness Rate

The term "complexity" of CFs, which estimates the structural complexity of CFs, is introduced to improve the structural efficiency of CFs. Because CFs here are binary trees (without considering negation, i.e. function NOT, as a separated node and adding complexity), the number of function nodes (internal nodes) is always 1 less than the number of leaf nodes. Thus, the complexity of a CF is defined as the number of leaf nodes, which are the amount of input information involved in evaluating the CF. Accordingly, the complexity of a rule is the accumulated complexity of all CFs in its condition.

According to section 4.1 (the initial implementation), the CF-based XCS targets to generate highly applicable tree-based features. Therefore, the structural efficiency of a CF is its capability to construct high-fitness rules with the least rule complexity. The rule complexity is defined as the total complexity of all CFs in its condition $cl.complexity = \sum_{cf 2 cl.condition} cf.complexity$, where $cf.complexity$ is the total leaf nodes of cf. A new term, called the "fitness rate" of a classifier, is defined as the fitness per unit of classifier complexity:

$$cl.f\_rate = \frac{cl.f}{cl.complexity}, \tag{4.2}$$

to estimate the CF-fitness of all CFs within the classifier. The CF-fitness of a CF based on the rule-fitness rate of the classifier (containing the CF) rewards higher CF-fitness on the CF that can construct accurate and more generalised rules using the least input information. Thus, this CF-fitness is termed as Generalised CF-fitness (GCFF). At this point, this CF-fitness has similar goals with the rule fitness of traditional XCS. The additional benefit is that CF-based conditions enable more complex patterns with the same input attributes compared with XCS, which can result in larger niches.

Accordingly, the OL gathers CFs from the conditions of the classifiers with the highest fitness per complexity unit in the action set of XCS (see Section 4.2.4). The updates of CF-fitness also follow the Widrow-Hoff learning rule [120] based on the classifier with the highest fitness per complexity unit containing the CF:

$$cf.f \mathrel{+}= \beta_{cf} \times (\max_{cl j cf 2 cl.condition} \frac{cl.f}{cl.complexity} - cf.f), \tag{4.3}$$

where β_{cf} is the learning rate of CF-fitness. This CF-fitness represents the highest fitness per complexity unit of a rule among rules having the CF, therefore called rule-fitness rate. In short, the OF module evaluates generated CFs based on their efficiency of CFs in using binary functions to combine the input attributes to produce accurate and generalised classifiers.

4.2.3 Niche-based Calibration of CF-fitness

The part of the OF module that associates with the learning processes of XCS, i.e. covering and genetic operations, is the CF generation by either selecting an existing CF or constructing a new one. As this process relies on CF-fitness, a niching method for CFs needs to be implemented for at least the CF-fitness. A niching method is developed to calibrate the CF-fitness of a CF based on the performance of the CF on the current niche. This method is designed to create boundaries between niches to prevent continual undesirable sharing of CFs. While sharing knowledge among niches is generally beneficial in many problems, undesirable transfers of CFs between niches can hold back the discovery of optimal building blocks for each niche.

The niching method calibrates the CF-fitness in three cases to estimate a local CF-fitness for the CF. First, if a CF has its best classifier matched in the current action set (the definition of the best classifier of a CF is discussed later in this paragraph), this CF is known to perform the best in this niche. In this case, the OF uses its CF-fitness directly. The second case is when a CF never appears in any classifier in the current action set. The system obviously has no data on its actual performance in this niche. This niching method estimates the local CF-fitness of this CF naïvely with a constant rate of 0.1 of its global CF-fitness. The third case lies between the first two edge cases when a CF does appear in at least one classifier in this local niche, but its best classifier is not the best overall. The estimated local CF-fitness of this CF is as follows:

$$cf.f_{\text{local}} = cf.f \times \frac{cf.local\_best\_classifier.f}{cf.best\_classifier.f}, \tag{4.4}$$

where the $cf.local\_best\_classifier$ is the "best" classifier containing the CF in the current action set $[A]$, and the $cf.best\_classifier$ is its global "best" classifier in the whole rule population $[P]$. It is noted that the definition of classifier being the "best" for a CF varies according to the CF-fitness. In XOF-BF, it is the highest-fitness classifier containing the CF. Because this niching method will be tested in the system that stacks this method with generalised CF-fitness, the quality of classifier is based on this new CF-fitness. Specifically, classifiers selected for Equation 4.4 are the ones with the highest rule-fitness rate:

$$cf.local\_best\_classifier = \underset{cl2[A]jcf2cl.condition}{\arg\max} \; cl.f/cl.complexity, \tag{4.5}$$

$$cf.best\_classifier = \underset{cl2[P]jcf2cl.condition}{\arg\max} \; cl.f/cl.complexity, \tag{4.6}$$

4.2.4 The Niche-based Observed List Update

This section describes a new niche-based OL update that simplifies XOF's processes and thereby eliminates a number of hyper-parameters. Also, the remaining hyper-parameters can still control the pace of the evolution of CFs, such as the learning rate for CF-fitness β_{cf}. Instead of periodical updates, the system updates the OL in every exploiting iteration with two processes. The first process is to collect the CFs in the conditions of the classifiers that best represent the action sets. For example, when using the CF-fitness in Section 4.2.2, the classifiers satisfying the following criteria will be used to collect the CFs for the OL:

$$cl.f/cl.complexity \geqslant 0.9 \times \max_{cl2[A]} cl.f/cl.complexity, \tag{4.7}$$

where 0.9 represents the selectivity of the OL. This value is empirically chosen among high values to compress the OL size. The second process is to remove CFs in outdated classifiers in the current niche that do not satisfy Eq. 4.7. This step could remove necessary building blocks of other niches in case of the problems with overlapping niches. However, as we place the OL update before genetic operations, the removed necessary CFs in other niches are always added back. This method is to collect all necessary building blocks in all niches.

4.2.4.1 Using XOF in Layered Learning

Although XOF is developed to be a learning system that can learn independently without any form of transfer learning, there is no restriction that prevents using XOF in transfer learning or layered learning as XCSCFC. XOF can even facilitate the transfer of CFs further with CF-fitness.

Transferring Criteria

The transferring criteria consider numerous aspects. First, only CFs in experienced and accurate classifiers, i.e. $cl.experience >= 25$ and $cl.error < \theta_0$, are the candidates for possibly being transferred as irrelevant CFs are not expected to have high applicability in the destined problem. Secondly, CFs are selected by their scores based on the accumulated structural efficiencies in classifiers to choose the highest-ranked CFs for transferring:

$$cf.score = \sum_{cf2cl|cl.experience \quad \theta_{GA} \hat{} cl.error<\theta_0} cl.fitness/cl.complexity \tag{4.8}$$

Since $cl.fitness$ already includes the numerosity information, the score does not need to take into account the numerosity. However, the more appearances of a CF in filtered classifiers, the more score it gets allocated. Lastly, the transferred CFs should not create noisy and irrelevant patterns on the target problem.

Based on the above principles, only a predefined maximum number of CFs with the highest scores are transferred. The maximum numbers of transferred CFs in each transferring step were chosen as the problem scale of the pre-transfer learning stage. This is because the Multiplexer domain does not require latent tree features and, thus, transferring more than original features is not necessary. Also, only CFs with substantial scores compared with the highest score are selected: $cf.score >= \arg\max_{cf} cf.score/10$ (10 is arbitrarily chosen). When transferring between problems at scales from 20 bits, the later criterion is normally met first.

Initialised CF-fitness of Transferred CFs

XOF can take advantage of the scores of the transferred CFs in CF-fitness. Initialising CF-fitness appropriately can accelerate the learning performance of XOF. The CF-fitness of CFs is initialised to be correlated with their scores and to maintain the ratio between the scores of the transferred CFs and the non-transferred base CFs, which will be created again in the transferred problem. As the initial CF-fitness of the non-transferred base CFs is 0.01, initialised CF-fitness of transferred CFs was transformed linearly using the rate $score_{base}/0.01$, where $score_{base}$ was the average of the scores of the non-transferred base CFs. Also, it is important to avoid being stuck in early learning phase by assigning too high scores on transferred CFs. We limited the maximum initial CF-fitness of transferred CFs to a naïve upper-bound 1.0. Hence, the initialised CF-fitness of transferred CFs is:

$$cf.fitness = \frac{(score_{max} - score_{base}) \times range_{limited}}{range_{score}} + 0.01, \tag{4.9}$$

$$\text{where } range_{score} = max(\{cf.score | cf \in \text{transferred CFs}\}) - score_{base}, \tag{4.10}$$

$$\text{and } range_{limited} = min(0.99, range_{score} \times 0.01/score_{base}) \tag{4.11}$$

4.2.5 Experiments

4.2.5.1 Generality Rate to Estimate the Structural Efficiency of CFs

To test the ability of XOF to generate complexity-efficient CFs, the structural efficiency of the CFs in the highest-fitness classifiers are tracked and evaluated. Also, the evaluation should be the least niche-biased to assess the ability of solutions to cover the whole problem. Thus, the classifiers for collecting CFs for tracking the structural efficiency are gathered from at most one classifier per action set. These classifiers also need to be accurate and experienced to avoid irrelevant estimation of performance, such as the high structural efficiency of an inaccurate general rule $cl.experience \geq \theta_{GA}$ and $cl.error \leq \epsilon_0$ [26].

This estimation of structural efficiency is still somewhat niche-biased because any niche with no experienced and accurate classifiers has no contribution to the estimated structural efficiency. This case is common when the classification accuracy has not reached 100%, but does not occur otherwise. Even after achieving 100% accuracy, the estimation of the CF-structural efficiency can still be niche-biased if the estimation is not weighted by niche size. However, precise measurement requires that niche sizes are known, which leads to deep knowledge of the problem. Due to being naïve about the tested problems, the evaluation will approximate the evolution of structural efficiency of the highest-fitness classifiers by averaging them among niches where experienced and accurate classifiers are available.

Having the representative classifiers to collect the most applicable CFs of the tested problem, a method to estimate the structural efficiency of these CFs is needed. Since these CFs are from experienced and accurate classifiers, the other aspect of efficiency is only the generality. Therefore, the structural efficiency should involve the generality and complexity. The "generality rate" of these classifiers to evaluate the structural efficiency of a classifier is tracked:

$$cl.generality = \frac{cl.matches}{cl.matches + cl.no\_matches}, \text{ thus} \tag{4.12}$$

$$cl.generality\_rate = \frac{cl.generality}{cl.complexity}, \tag{4.13}$$

where $cl.matches$ and $cl.no\_matches$ respectively track the numbers of times classifier cl matches and does not match all instances since it was created, and therefore the part $cl.matches/(cl.matches + cl.no\_matches)$ provides the generality of classifier cl as it tracks the probability that classifier cl matches any instance. The generality rate of classifiers estimates their efficiency in using $cl.complexity$ complexity units (CF structures) to produce accurate classifiers with the highest generality.

4.2.5.2 Experimental Design

This section includes a comparison of XOF and the two newly implemented features with XOF itself and XOFs with one of the two features. There are two criteria for comparisons: the learning performance as well as the discovery of complexity-efficient CFs. In this work, the existing CF-fitness focusing on shortening rule conditions is abbreviated as Shortening CF-fitness (SCFF) and the niching method for CFs as Niching CFs (NCF). Thus, in addition to the existing version named XOF-BF, experiments will include three new other approaches abbreviated as XOF-SCFF, XOF-GCFF, and XOF-GCFF-NCF within this work. All of these new systems use the simplified OL update in Section 4.2.4.

All parameters of all tested versions of XOF are configured equally except for the rule population size and stopping iteration. A general configuration of XOF is used in these experiments: the learning rate for rule parameters $\beta = 0.2$ and the learning rate for CFs $\beta_{cf} = 0.001$; the crossover rate is $\chi = 0.2$; the mutation rate is $\mu = 0.9$; the experience thresholds for deletion is $\theta_{del} = 20$; the initial fitness of covered classifiers are $F_{init} = 0.01$; the probability of specificness $p_{spec} = 0.25$ with maximum rule-condition length set at twice the number of original input attributes; and the experience thresholds for subsumption $\theta_{sub} = 50$. All three newly implemented versions of XOF have no limit on the OL size and a redundantly high limit on the CF depth, i.e. the maximum depth is 20.

4.2.5.3 Results on 11-bit Even-parity Problem

The population sizes for all systems on this problem were equal to 8000 classifiers, which enabled XOF-BF to converge. The learning graphs of tested approaches in this experiment were not substantially different except for the convergence phase, see Figure 4.10a. XOF-SCFF and XOF-BF had a slight advantage in the early phase. In 30 runs of the three new systems in this work, there was always one or two runs that were stuck at 50% accuracy. The reason for being stuck was that the evolution of CFs creates an extra force to push the evolution of rules further to the local optima. Also, this Even-parity problem already poses a high probability of local optima for XCS as the probability of finding correct rules in XCS is very low. All inaccurate rules have the same accuracy of 50%, including the simplest rules and the rules with genotypes near the accurate ones. These stuck runs always ended up with the domination of a few very short rules (with only one CF in its conditions). The high rule-numerosity and short rule conditions (few CFs) caused the CFs in these rules to achieve high CF-fitness and thereafter pushed these rules to earn more numerosity through genetic operations.

All three systems discovered CFs with significantly more efficient structures than ones by XOF-BF. The system with the niching method for CFs evolved the most optimal CFs (Figure 4.10b), although there were two stuck runs in most of the tested experience ($1{,}000{,}000$ instances), which had very low generality rate. The evolved CFs of this system reached near the optimal generality rate for this problem. The optimal generality rate is equivalent to the generality rate of the most generalised classifier in Figure 4.9. This classifier has the generality of 0.5 as it matches half of all instances. Also, its complexity is 11 for 11 leaf nodes. Hence, its generality rate is $0.5/11 = 0.04545$, which is the optimal generality rate here.

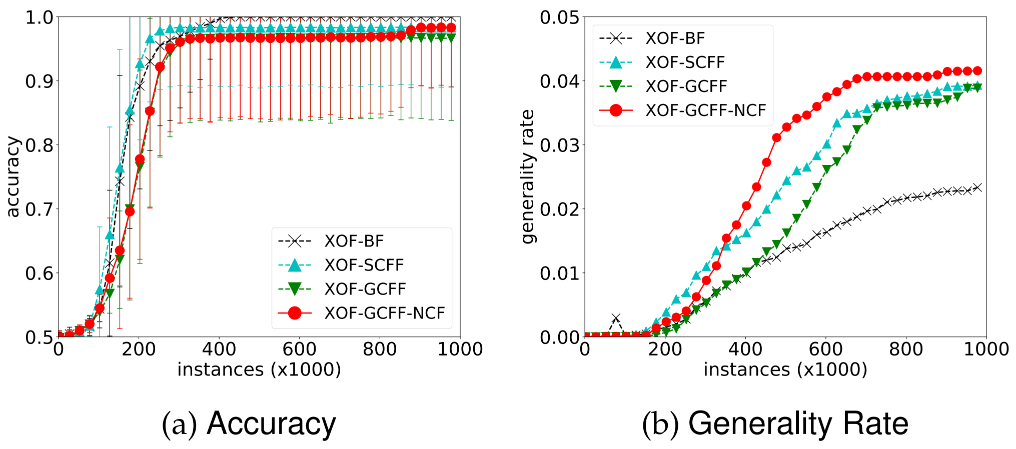
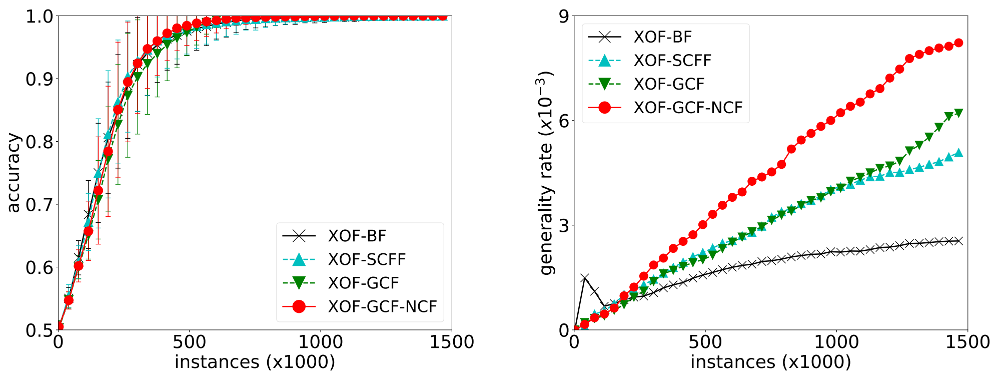


Figure 4.10: Results on 11-bit Even-parity problem.

4.2.5.4 Results on Hierarchical Problems

The 18-bit Hierarchical Multiplexer and 18-bit Hierarchical Majority-on problems are used to evaluate the generality rate of the tested approaches. These two problems pose relatively large search spaces as their hierarchy adds complexity to the data patterns. To capture these complex patterns, constructed CFs need to cover all 6 non-overlapped three-successive-bit chunks with 3-bit Even-parity problems. An optimal CF that can cover a chunk, say (D_3, D_4, D_5), has to use XOR in all function nodes except for any arbitrary negation, such as $(!((!D_3) \times D_4)) \times D_5)$. Such CFs can match half of all possibilities for the three bits and the other half by its negated version, which is not counted as a distinct construction in XOF because each CF has one corresponding negated version.

All systems in these two experiments had the same population size of $20{,}000$. The learning performances of all approaches are no substantially different from one another in the 18-bit Hierarchical Multiplexer problem, see Figure 4.11a. In the 18-bit Hierarchical Majority-on problem, two XOFs with the SCFF, XOF-SCFF and XOF-BF, did not converge to 100% accuracy,

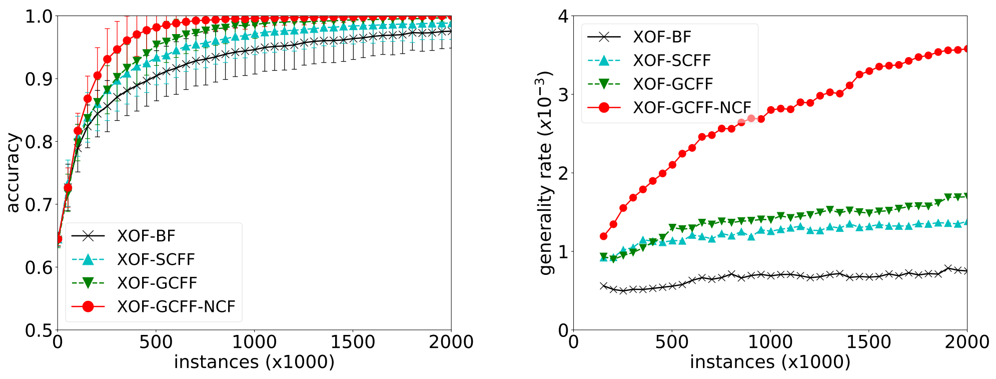


(a) Accuracy (b) Generality Rate

Figure 4.11: Results on 18-bit Hierarchical Multiplexer problem.

while the two XOFs using the GCFF, XOF-GCFF and XOF-GCFF-NCF, did (see Figure 4.12a). The niching CFs also improves the learning performances of XOF in both experiments.

In these two experiments, XOF with niching CFs (XOF-GCFF-NCF) yielded the most optimal CFs among the tested versions of XOF (Figures 4.11b and 4.12b). The evolution of CFs in XOF-BF again was stuck, plus it contained the most bloated CFs. Table 4.4 illustrates a few samples of CFs in the OLs of three tested systems. This table includes learnt CFs related to chunks without optimal CFs to demonstrate the difference of the influence by the two CF-fitness. SCFF, the CF-fitness focusing on shortening rule conditions, rated the two non-optimal CFs $(!((!D_{14}) \times D_{13})) \wedge D_{12}$ and $(!((!D_{14}) \times D_{13})) \vee D_{12}$ much higher CF-fitness than the lower-level CFs $!((!D_{14}) \times D_{13})$ and D_{12}. The two latter CFs were the ones that can be combined to construct optimal CFs for the 3-bit Even-parity problem (on three bits (D_{12}, D_{13}, D_{14})). Although the patterns of the two former CFs did not generalise more than the lower-level ones on the 3-bit Even-parity problem, these higher-level CFs still had higher CF-fitness because they can produce rules with shorter conditions and the same patterns. This pro-



(a) Accuracy (b) Generality Rate

Figure 4.12: Results on 18-bit Hierarchical Majority-on problem. The beginning parts of generality rates are omitted because over-general (inaccurate) rules that are temporarily accurate and experienced can create unreliable estimation.

cess even hindered discovering the optimal CFs for this 3-bit Even-parity chunk because the higher-level CFs have greater probability to be selected in constructing CFs. Meanwhile, GCFF, the generalised CF-fitness, did not face this problem because such non-optimal combinations do not achieve higher CF-fitness than the lower-level CFs.

4.2.5.5 Layered Learning in the Multiplexer Domain

The learning stages designed for XOF started from the 20-bit Multiplexer problem to progressively larger scales problems. The learning stages of XCSCFC were the same as the stages used in the original implementation of XCSCFC [63], which included all scales of the Multiplexer domain from 6 bits to 70 bits. The population sizes of both systems were set equal in corresponding stages (at the same scales): $N = 2000$ for the 20-bit scale, $N = 5000$ for the 37-bit scale, and $N = 20000$ for the 70-bit scale.

Figure 4.13 shows performance comparisons between XOF and XCSCFC

Table 4.4: A few sample CFs with their CF-fitness in the OLs of XOF-SCFF, XOF-GCFF, and XOF-GCFF-NCF after $150{,}000$ instances of learning the 18-bit Hierarchical Multiplexer problem. These samples were CFs related to a chunk that optimal CFs have not yet constructed. To be as fair as possible, these samples were chosen from runs with "generality rates" in the small range from 0.0075 to 0.0080.

XOF-SCFF	**XOF-GCFF**	**XOF-GCFF-NCF**
$(!((!D_{14}) \times D_{13})) \wedge D_{12}$ (0.157)	$(!D_9) \wedge (!D_{11})$ (0.058)	$(!D_8) \times (!D_7)$ (0.071)
$(!((!D_{14}) \times D_{13})) \vee D_{12}$ (0.157)	$(D_9 \wedge D_{11}) \times (!D_{10})$ (0.056)	$(!D_8) \times D_7$ (0.061)
$!((!D_{14}) \times D_{13})$ (0.121)	D_{10} (0.057)	D_6 (0.071)
D_{12} (0.121)	D_{11} (0.054)	n/a
n/a	D_9 (0.054)	n/a

with layered learning in the Multiplexer domain at the 37-bit, and 70-bit scales. XCSCFC performed better than XOF in traditional learning without layers. With layered learning, XOF learned slightly faster than XCSCFC in the 37-bit scale, and remarkably faster than XCSCFC in the 70-bit scale. The learning performances of XOF and XCSCFC are both improved with the layered learning approach.

The layered learning approach bootstrapped both XCSCFC and XOF with starting points closer to the underlying patterns of the Multiplexer domain. The transferring criteria in layered learning filtered out some or all of the base CFs representing the data bits. The transferred data bits were also initialised with a lower starting CF-fitness. The reason was that the contributions of data bits in large-scales Multiplexer problems were generally less than 10% of the contributions of the address bits. These results describe correctly the characteristics, including epistasis where data bits must be linked to address bits, of the Multiplexer domains.

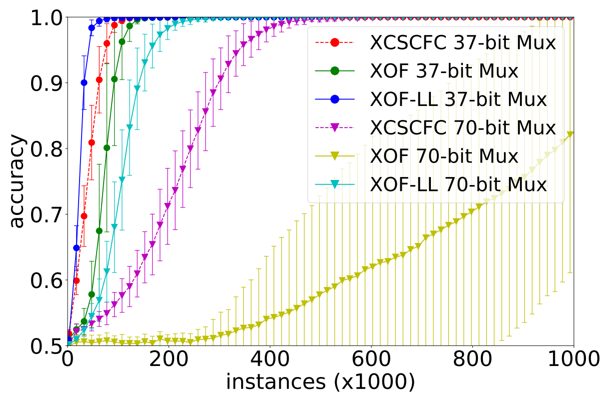


Figure 4.13: Layered Learning in the Multiplexer domain.

4.2.5.6 Analysis of the Evolution of CFs

This section investigates the importance of the CF evolution along with its influence when the evolving pace is varied. Two benchmarks on 15-bit Hierarchical Majority-on and 37-bit Multiplexer were used for this analysis as they are either hierarchical or relatively large in scale. In these two problems, different learning rates of CF-fitness $\beta_{cf} \in \{0.1, 0.01, 0.001, 0.0001\}$ were used to demonstrate the varied paces of the CF evolution. Additionally, another experiment of XOF without constructing new CFs was executed to investigate the importance of the process in XOF. Figure 4.14 showed the results with $\beta_{cf} = 0.001$ having the best performance in both problems.

In 15-bit Hierarchical Majority-on, the construction of new CFs has an important role in this problem as the performance of XOF without it was much worse than standard XOF. This is explainable as this problem can be solved with much fewer rules when complex CFs are created to capture the underlying Even-parity patterns of the lower level. The construction of new CFs had a much less contribution to the learning performance of XOF in 37-bit Multiplexer as XOF without it learned slightly slower than XOF

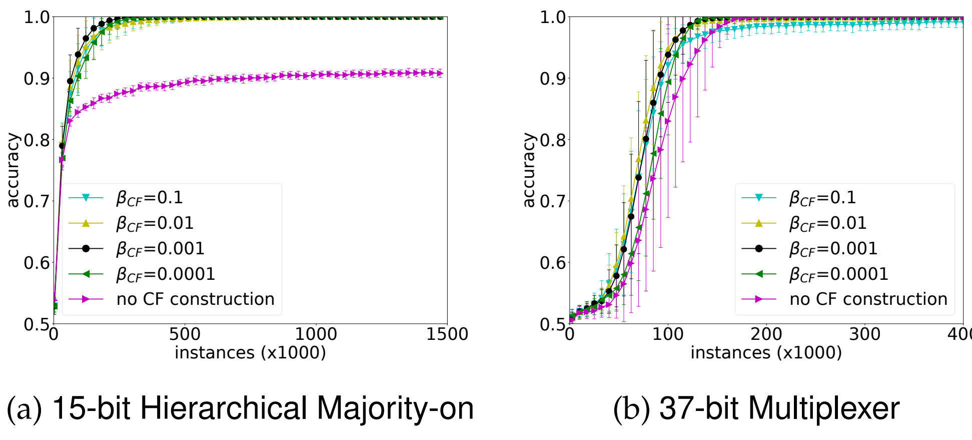


Figure 4.14: Analysis on the evolution of CFs.

with $\beta_{cf} = 0.001$. This result also shows that the construction of new CFs did contribute to the learning process of the Multiplexer domain, although this domain does not require complex patterns. The possible reason was that the creation of new CF based on the CFs in the OL encouraged creating more CFs involving address bits, which are more frequently required to solve the Multiplexer domain.

All tested values of β_{cf} yielded relatively equal performance in the 15-bit Hierarchical Majority-on. In the 37-bit Multiplexer problem, the performances were more varying as XOF with higher β_{cf} struggled to converge in the final learning phase. The high β_{cf} pushed XOF to create more new CFs that confused the rule evolution of this problem domain, which does not require complex CFs. This also happened in the 15-bit Hierarchical Majority-on problem but not substantially noticeable. XOF with the lowest β_{cf} learned only slightly slower than the XOF with $\beta_{cf} = 0.001$.

4.2.6 Further Discussions

XOF is equivalent to genetic programming [74] in terms of evolving tree-based programs to address problem patterns. The evolution of CFs has an essential contribution to the learning performance of XOF in hierarchical problems as it progressively produces complex CFs to capture the complex patterns of such problems. Unlike genetic programming, XOF has an intermediate level of symbolic rules between trees and the target task. The inclusion of trees in rules enables evaluating trees even though trees have not been evolved to the point they can be complete solutions.

Although transfer learning and layered learning are not required to achieve a good performance on complex problems, these approaches can help XOF to learn better. However, designing transferring criteria and layers of learning is dependent on source and target tasks. This task is challenging for both XOF and XCSCFC.

On the 11-bit Even-parity problem, the pure pressure on combining CFs and shortening rule conditions of SCFF has slightly better learning performance than the generalising pressure of GCFF. This can be explained as this CF-fitness estimation awards higher CF-fitness on combined CFs, which pushes the generalisation process faster. However, GCFF has a better performance on the Hierarchical Majority-on problem because it does not push the system towards rules with shorter conditions, which can easily become over-general rules in problems with overlapping niches.

The niching method for CFs improves the structural efficiency of CFs in XOF significantly, as shown by its superior average generality rate. Niching CFs guides combining optimal CFs to generalise existing patterns. Therefore, the evolution of CFs with niching CFs is accelerated without adding the likelihood of being trapped in local optima. This also results in slightly faster learning performances on the tested Hierarchical problems.

4.3 Chapter Summary

This chapter introduces XOF that grows high-level CFs to address complex (high-level) patterns that are relevant to the target problem. This extension enables CF-based XCS to solve large-scale and complex problems without the need of a customised sequence of layered learning. An extra evolution of CFs is connected with the rule evolution by developing new measures of estimating a parameter called CF-fitness. CF-fitness enables CF evolution to interact with the environment indirectly through the rule evolution. Although transfer learning and layered learning are not required to achieve a good performance on complex problems, these approaches can help XOF to learn better.

This chapter has also introduced a new method for estimating CF-fitness, called generalised CF-fitness estimation, that focuses on generalised patterns and avoids naïvely combining existing CFs. Accordingly, other processes of XOF have been adjusted to select CFs following the new criteria. The generalised CF-fitness slows down the growth of CF depth but adds more reliability to the CF construction. Although the structural efficiency of generated CFs has not been improved, it enables integrating a newly developed niching method for CFs, which results in accelerating the evolution of CFs without being trapped in local optima.

The niching method for CFs introduces the niching property to CF construction in XCS with the OF module, i.e. XOF. The niching property enables appropriate combinations of CFs to grow optimally complex CFs for hierarchical problems. This property accelerates the generalisation of XOF rules. With this new feature, XOF, as an extension of XCS, has the niching property in both the evolutions of rules and CFs. Regarding the niching property, a new method for updating the OL that enables collecting the most applicable CFs from all niches has been developed.

The next chapter considers extending XOF with the new features to a multi-XOF system to target multitask learning, unlike the traditional single-task learning in this chapter. The availability of the OL as the representative patterns can facilitate comparing the relatedness between problems and, as a result, dynamically improving the transfer of features among the multiple tasks.

Chapter 5

Relatedness Measures to Facilitate Automating the Transfer of Building Blocks among Multiple Tasks

Multitask Learning (MTL) is a learning paradigm that deals with multiple different tasks in parallel and transfers knowledge among them. While much MTL research in evolutionary computation is dealing with multiple optimisation tasks, this work is focused on multiple classification tasks. The previous chapter introduces XOF as a Learning Classifier System using tree-based programs to encode building blocks (meta-features). XOF constructs and collects features with the most discriminative information for classification tasks in an Observed List (OL).

This chapter seeks to facilitate the automation of feature transferring in MTL by utilising the OL. This list contains the patterns that are validated to perform the best among all tree features. It is considered that the features with the most discriminative information of a classification task carry the characteristics of the task. Therefore, the relatedness between any two tasks can be estimated to be correlated with the portion of common patterns between their OLs.

This chapter introduces a multiple-XOF system, called mXOF, that can dynamically adapt feature transfer among XOFs. mXOF utilises the OL of each individual system to estimate the task relatedness. This method enables the automation of transferring features. Experiments of mXOF include various scenarios, e.g. representative Hierarchical Boolean problems, classification of distinct classes in the UCI Zoo dataset, and unrelated tasks that are not supportive of each other. These experiments are to analyse mXOF's abilities of automatic knowledge-transfer and estimating task relatedness. Results show that mXOF can estimate the relatedness dynamically between multiple tasks to aid the learning performance with the dynamic feature transferring.

5.1 Introduction

The ability to reuse knowledge among similar tasks allows humans to comprehend skills and concepts through a few examples for each new problem. This motivates the advent of Multitask Learning (MTL), a learning paradigm where the learning system addresses multiple related tasks simultaneously with equal task priority [28, 101]. MTL aims to improve the learning performance of each task by transferring useful knowledge among related tasks.

However, existing MTL work are generally restricted to related tasks where the contribution of common knowledge significantly dominates the adverse effect of the harmful signal coming from the unrelatedness of other tasks. Otherwise, the learning performances can become worse than those in separated traditional single-task learning. This limitation requires external knowledge on the tasks selected for MTL. On the contrary, humans' ability to reuse knowledge is not bounded to related tasks. Human intelligence can choose to relate a target task with learnt tasks that are statistically more relevant than randomly choose any learnt tasks[1]. The hypothesis is that relatedness of any two problems, estimated based on the overlap of their best-described patterns, can be a guidance to transferring knowledge among tasks.

Evolutionary Computation learns optimisation tasks through building blocks, which can be transferred among tasks. Evolutionary MTL has been investigated with a variety of EC algorithms. Among them, the series of Multifactorial Evolutionary Algorithms [8, 48] also offers the ability to prevent harmful interactions between distinct optimisation tasks using a matrix of random mating probability. However, MTL in optimisation tasks seeks to guide search trajectories, while here classification tasks with the transfer of useful building blocks between systems are considered.

The previous chapter introduces XOF that grows high-level CFs in LCS's rule conditions through the concept of the Observed List (OL). This list includes the most applicable CFs, which contain the most discriminative information for the target task. In other words, the OL contains the harvested information about the task in the form of tree features (CFs). In the case of online learning, this is also the only information from the task as a pre-built training set is not available as in supervised learning. In this work, two tasks are considered highly related if their common underlying patterns contribute largely to each of the total sets of their patterns. Therefore, it is hypothesised that leveraging OLs to estimate the task relatedness can assist in sharing knowledge among tasks within XOF.

In this chapter, a system of multiple XOFs, called mXOF, that can solve different problems simultaneously is proposed. It needs to automatically detect the common characteristics of the problems to facilitate transferring

[1]The amount of existing learnt knowledge is just too large to be queried randomly for reuse

features among XOFs. Each XOF in mXOF learns one problem. The comparison of two OLs is used to estimate the relatedness of two problems. The objectives of this chapter are:

1. To introduce an MTL system that automatically shares learnt features among related tasks by estimating the asymmetric relatedness of tasks. The relatedness $RelSS_{a,b}$ of a task a to another task b directly influences the probability of sharing CFs from task a to task b.

2. To facilitate estimating the relatedness between tasks by using the similarity of their OLs. A similarity metric among OLs is the key factor of this chapter as it will influence the results of all three objectives.

3. To investigate the ability of mXOF to handle arbitrary multiple tasks, including related and unrelated tasks. This can further validate the automation of mXOF in sharing features among tasks. If the system can estimate a low relatedness between two unrelated problems, the learning signals of these two problems can be prevented from interfering with the learning processes of each other. The ability to handle arbitrary tasks contributes to general AI systems with continual learning [51, 124].

The specific context of this work is that a learner (robot or computer), is learning to recognise different objects (classes) in parallel using signals from the same sensor as the input data. In the very beginning, recognising all objects starts with the original data features from the sensor, named as $(D_0, D_1, D_2, ...)$. At this stage, there is no divergence of patterns among the tasks except for the expected output $(1/0)$ because the learner sees all objects as the raw input signal. The progressive learning grows high-level building blocks from the original input by feature construction inherent in CFs. Recognition tasks that have different sets of latent patterns will require different latent features. The goal of mXOF is to track the relatedness among the recognition tasks to automate the transfer of building blocks. The automated transfer improves the learning process of each task and link related tasks together.

The system will be tested on multiple scenarios to validate its capability to automate sharing features among tasks. First, multiple hierarchical problems [23], which share the equivalent base-level patterns, will validate the ability to transfer CFs among highly related tasks. Also, a scenario of two relatively unrelated tasks, i.e. 11-bit Even Parity problem and 10-bit Carry problem, will test the ability of mXOF to transfer features among unrelated tasks selectively. Finally, a practical multi-class classification problem will be used to evaluate mXOF as a multi-class classifier.

Learning benchmark datasets have been constructed for independent (non-transfer) tasks. They are often unrelated and recorded in different settings. Thus, the UCI Zoo dataset has been repurposed by converting to multiple binary classifications to test whether mXOF can discover possible relatedness among seven classes in this dataset. This also test the abilities of mXOF without a priori knowledge, to autonomously discover the relatedness among classes.

5.2 Multitask Learning with mXOF

This section provides detailed description of an mXOF system where each XOF learns one of the multiple tasks. Figure 5.1 illustrates a case of mXOF with three systems and three tasks. The sharing of CFs and the relatedness measurement among systems mutually support each other during the learning processes of the multiple systems. In this chapter, the identification for a system is the same with its corresponding task because each system works on one task.

All systems utilise a common CF population. The CF population here serves similar purposes as the CF population does in a single-population system of XOF. The common population enables tracking generated CFs

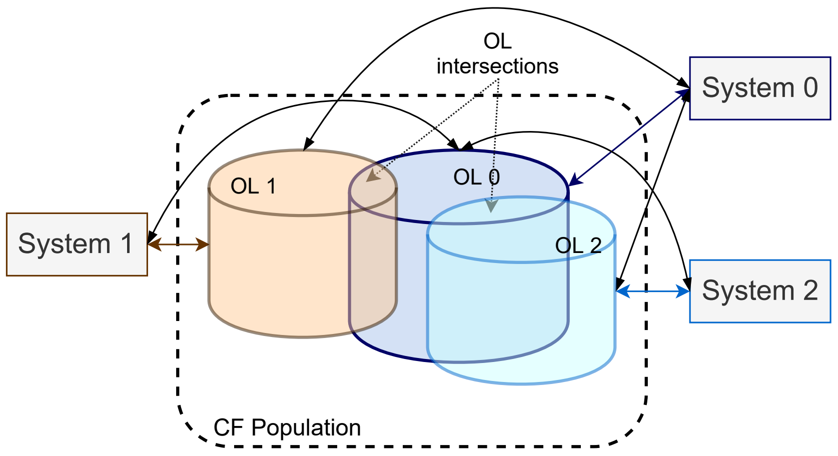


Figure 5.1: An illustration of mXOF solving three tasks. A large intersection between two OLs indicates a high relatedness between two tasks, which is the guidance used for automatic CF transfer.

among systems. The population links equal CFs (same genotype) among single-XOF systems, which enables comparing the OLs and thereby the estimation of task relatedness among systems.

5.2.1 Asymmetric Fitness-weighted Relatedness

The commonality of CFs between the OLs of two tasks is hypothesised to indicate the relatedness between the tasks. Specifically, CFs in the OL of a system are the most discriminative patterns among generated features, which are useful in constructing accurate and generalised rules, i.e. high-fitness rules. Therefore, two tasks with common discriminative patterns in the intersection of two OLs are considered related tasks. Intuitively, two OLs with more similarity correspond to systems/tasks that are more related.

In accordance with the above intuition, the relatedness of a system a to another system b is to estimate the general applicability of CFs produced in the system a to system b (see Chapter 4 for the applicability of CFs). The fitness-weighted relatedness of system a to system b corresponds to the CF-fitness portion of sharing CFs between a and b in the total CF-fitness of CFs in the OL of system a:

$$RelSS(a,b) = \frac{\sum_{cf_i 2 OL_a \wedge OL_b} cf_i.f(a)}{\sum_{cf_j 2 OL_a} cf_j.f(a)}, \tag{5.1}$$

where OL_a and OL_b are two OLs of system a and b respectively, and $cf.f(a)$ is the CF-fitness of cf in system a. This asymmetric definition of task relatedness is a statistical estimation of the applicability of any CF from the OL of system a to system b. The maximum relatedness is 1 when all CFs in the OL of the source task are applicable to the target task, and is 0 when no CFs in the OL of the source task is relevant to the target task. The asymmetric property is desirable because the applicability of features between two tasks is generally not symmetric. For example, when a task can be a subset of another task, almost all the features from one task is applicable to the other task while the applicability in the opposite direction can be minimal.

5.2.2 Automatic Transfer of CFs

The CF transfer among tasks is automatically driven by their relatedness. This mechanism benefits the learning processes of the involved systems in multiple ways. First, there is no need for human intervention as in transfer learning or layered learning, such as selecting a sequence of highly related tasks and criteria of features for transferring. Second, on the contrary, this enables the dynamic criteria of transferring features among tasks. This property is desirable because, when the learning process of XOF evolves CFs, the applicability of CFs from one task to another and the relatedness of tasks usually change dynamically. For example, two tasks may only share common patterns at certain levels, which are only present at some points of evolving CFs. Lastly, the transfer of features provides individual systems during evolving with external knowledge that can be a force to escape local optima.

In mXOF, a system uses transferred CFs as well as existing CFs in its OL when selecting CFs for covering new rules or mutating a rule condition. Algorithm 5.1 illustrates how the selection procedure works. Whenever a target system a queries an existing CF, it will select from its OL plus at most one external CF from other systems.

The external CF is selected from a set S_{ecf} of all CFs in the OLs of other systems that satisfy two relatedness criteria. The first criterion is that the source system providing external CFs must be more related to the target system a than a threshold r_thres. This threshold is drawn from a uniform distribution (step 2, discussed later). Second, these CFs themselves have to be potentially applicable to the target task a. This quantity is estimated by a statistical expectation of relatedness from external CFs to a target task. The expected relatedness of an external CF cf_j is actually the relatedness of the source task containing cf_j adjusted by the rate between CF-fitness of cf_j and the average CF-fitness of the common (shared) CFs (between the source task i and the target task a) on the source task i:

$$RelCfS(cf_j, a) = \frac{cf_j.f(i)}{\sum_{cf 2 C(i,a)} cf.f(i)/len(C(i,a)} \times RelSS(i,a), \tag{5.2}$$

where $C(i,a) = OL_i \wedge OL_a$. This check can also be interpreted that the performance of the candidate cf_j should be comparable with the performances (on the source task) of shared CFs between the two tasks. If the relatedness of the external CF satisfies the threshold r_thres, S_{ecf} will append this CF with its adjusted vote (applicability) shown in Step 10. Finally, a Roulette Wheel selection chooses a CF from S_{ecf} using adjusted votes of external CFs.

The relatedness threshold r_thres, drawn from a uniform distribution, represents the stochastic selectivity of the learner in sharing CFs among tasks.

Algorithm 5.1 Transferring an external CF cf_e from other systems to reuse it in target system a. cf_e with its adjusted vote $vote_{adj}(cf_e, a)$ will be another candidate for system a when selecting existing CFs to construct rules.

```
1: Collection of external CFs from other systems' OLs $S_{ecf} = \emptyset$
2: Draw a relatedness threshold from a uniform distribution $r\_thres = max(0.1, uniform(0, 1))$
3: for system $i \neq a$ do
4:     if $RelSS(i, a) >= r\_thres$ then
5:         Common CFs in the $OL_i$ and $OL_a$: $C(i, a) = OL_i \wedge OL_a$
6:         for $cf_j \in OL_i$, the OL of $i$ do
7:             if $cf_j \notin OL_a$ then
8:                 Relatedness of $cf_j$ to system $a$:
                   $RelCfS(cf_j, a) = RelSS(i, a) \times (cf_j.f(i)/$
                   $(\sum_{cf2OL_i\hat{}OL_a} cf.f(a)/len(C(i, a))))$
9:                 if $RelCfS(cf_j, a) >= r\_thres$ then
10:                    Compute adjusted vote of $cf_j$: $vote_{adj}(cf_j, a) = cf_j.f(i) \times$
                       $(\frac{\sum_{cf \in (OL_i \wedge OL_a} cf.f(a)}{\sum_{cf \in (OL_i \wedge OL_a} cf.f(i)} \times RelSS(i, a))$
11:                    Add $cf_j$ to external selections $S_{ecf}.add(cf_j)$ with its adjusted vote
12: Select an external CF from $S_{ecf}$ using roulette wheel selection and their adjusted votes $cf_e = RW(S_{ecf})$
```

To simplify the hyper-parameters, one threshold value is used as a common filter for both external tasks and CFs. Using the uniform distribution for the sharing selectivity might not filter out all negative transfer. However, when there are more and more tasks, the most related tasks always have the highest probability to share with one another.

The vote (applicability) of an external CF, see step 10 of Algorithm 5.1 is adjusted regarding the performance of the shared CFs between the two

OLs on the source and the target tasks. The adjustment is based on the rate between the total CF-fitness of shared CFs in the target task and the corresponding amount in the source task. The better performance of shared CFs on the target task, the higher adjusted vote the external CF can get. This adjusted applicability is also used for the external CF when competing with existing CFs in the OL of the target task for constructing rules.

5.2.3 mXOF for Multi-class Classification

Multi-class classification can be converted into multiple binary classification problems, where each class corresponds to a binary classifier, i.e. an XOF. Each binary classifier is responsible for recognising its associated class (one versus other classes). Subsequently, all the binary classifiers can work synchronously on each same instance with rewards converted from the ground truth. That is, while each system receives the same environment state (instance input) each iteration, only one that works on the true class of the instance should output 1 to receive maximum reward (e.g. 1000) and other systems should output 0 for the maximum reward. mXOF can be considered a multi-class classification system, where each XOF recognises one class.

When evaluating an instance on the test set, each binary classifier produces a recognition probability, or a confidence level, that this instance belongs to the class corresponding to the classifier. The probability is necessary in case multiple binary classifiers output $True$ on an instance in multi-class classification tasks. The chosen class is simply the class of the system with the highest probability.

The implementation of the recognition probability is based on the prediction array of XOF. Specifically, the probability of action 1 ($True$), which means the corresponding class is detected, is:

$$P(1) = \frac{\sum_{cl2[A]jcl.action=1} cl.prediction \times cl.f}{\sum_{cl2[A]} cl.prediction \times cl.f}. \tag{5.3}$$

This is the rate of the total fitness-weighted prediction of action 1. Thus, the probability of action 0 is $P(0) = 1 - P(1)$. When more than one binary classifiers produce the same highest probability, mXOF randomly selects a class from the classes of such binary classifiers.

The possible downside of this strategy of converting multi-class classification into multiple binary classification problems is that one balanced dataset can become multiple imbalanced data. However, it is expected that this problem does not have much influence on the result because XCS can manage to balance its niches very well [129]. Conventional approaches to dividing a multi-class problem into multiple binary problems also suffer as building blocks that describe patterns for more than one class have to be relearnt in each new binary problem. On the contrary, using mXOF for multi-class classification provide relatedness among classes. The class relatedness provides an insight knowledge on the relationships among the associated classes.

5.3 Experimental Results

The experiments illustrate the benefit of automatic transfer by comparing mXOF that simultaneously addresses several problems with individual XOF on the same but separated problems. There are two criteria for comparisons: the learning performance as well as the discovery of complexity-efficient CFs.

All parameters of each system in mXOF were the same as the corresponding ones in single XOF for the same problems. Except for the rule population size and stopping iteration, a general configuration of XOF for other parameters was used in these experiments: the learning rate $\beta = 0.2$; the crossover rate $\chi = 0.2$; the mutation rate $\mu = 0.9$; the experience threshold for deletion $\theta_{del} = 20$; the initial fitness of covered classifiers $F_{init} = 0.01$; the probability of specificness $p_{spec} = 0.25$ with maximum rule-condition length set at twice the number of original input attributes; and the experience thresholds for subsumption $\theta_{sub} = 50$. The learning performances in these experiments were the number of exploration trials (instances).

The implementation is in multi-threaded Python with a progress synchroniser to assure all systems in mXOF experience the same number of iterations during their learning processes. The purpose of this progress synchronisation is to produce a more stable and reliable evaluation.

To track and evaluate approximately the generality rate of the most complexity-efficient CFs in an XOF, the most efficient classifiers in action sets of exploitation, i.e. highest fitness per complexity, that satisfy:

$$cl.f/cl.complexity >= 0.9 \times \max_{cl2[A]} cl.f/cl.complexity. \tag{5.4}$$

were collected. These classifiers are called exploit classifiers regarding how they were collected. Exploit classifiers are also the classifiers selected for collecting the OL. The tracked generality rate is the average generality rate of the most efficient and accurate classifiers with $cl.error = 0$. Experiments showed that the threshold of 0.9 in the above equation can be changed in the range of $[0.8, 0.95]$ without substantial impact on the learning performance and feature construction in most tested problems. The generality rates of classifiers are estimated as follows:

$$cl.generality = \frac{cl.matches}{cl.matches + cl.no\_matches}, \text{ thus} \tag{5.5}$$

$$cl.generality\_rate = \frac{cl.generality}{cl.complexity}, \tag{5.6}$$

where $cl.matches$ and $cl.no\_matches$ respectively track the numbers of times classifier cl matches and does not match all instances since it was created, and therefore the part $cl.matches/(cl.matches + cl.no\_matches)$ provides the generality of classifier cl as it tracks the probability that classifier cl matches any instance.

5.3.1 MTL with Hierarchical Boolean Problems

Hierarchical Boolean problems are problems that combine a Boolean problem at the top-level with successive sub-problems (e.g. 3-bit Even-parity problems) at the bottom-level (see Chapter 3). Because of this combination, hierarchical problems have highly complex underlying patterns. Even though these problems can be small in scale (number of input bits), the search spaces in solving them are much higher than other Boolean problems, such as Multiplexer, at the same scale because they require complex combination of attributes. Discovery of these patterns can simplify the search of decision boundaries, i.e. the rules of XOF. These experiments can be considered to involve related problems because at some stage of learning, XOF can construct CFs covering the bottom-level 3-bit Even-parity problem, which are common among problems.

mXOF is first evaluated on two set of multiple hierarchical problems. The first set includes 12-bit Hierarchical Carry-one, 9-bit Hierarchical Multiplexer, and 9-bit Hierarchical Majority-on problems. All individual systems of mXOFs and single XOFs used the same population size of $N = 2000$ (Figure 5.2). The second experiment has two larger-scale problems including 18-bit Hierarchical Carry-one and 18-bit Hierarchical Multiplexer problems (Figure 5.3). In this experiment, individual systems of mXOF and single-system XOFs all had the same population size of $N = 20000$. These experiments also included mXOF, XCSCFC (with layered learning) and XCS with the same higher population size of $N = 50000$. These problems require constructing relevant and complex patterns to simplify decision boundaries.

In both experiments, the performances of mXOF for each of these prob-

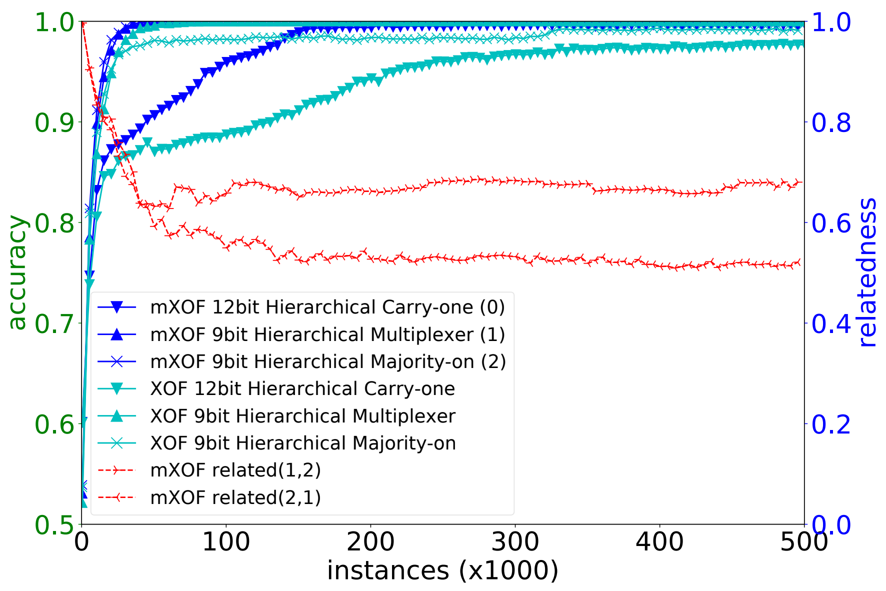


Figure 5.2: Learning performance on multiple small-scale hierarchical problems. The red lines show the average relatedness among tasks. Note: 0 is Hierarchical Carry-one, 1 is Hierarchical Multiplexer, and 2 is Hierarchical Majority-on.

lems are superior to those of single-system XOFs on corresponding problems. The MTL system also achieves higher accuracies compared with the single-system method except for XCSCFC. This system has a similar performance with mXOF on the 18-bit Hierarchical Multiplexer problem, but it has a higher population size and also requires layered learning to achieve such performance [63].

The average values of the relatedness parameters between 18-bit Hierarchical Carry-one problem and 18-bit Hierarchical Multiplexer problem are illustrated in Figure 5.3. These parameters were averaged across 30 runs. The relatedness (between two 9-bit problems) values start at high values (1.0) as both tasks start with the same base CFs. The relatedness param-

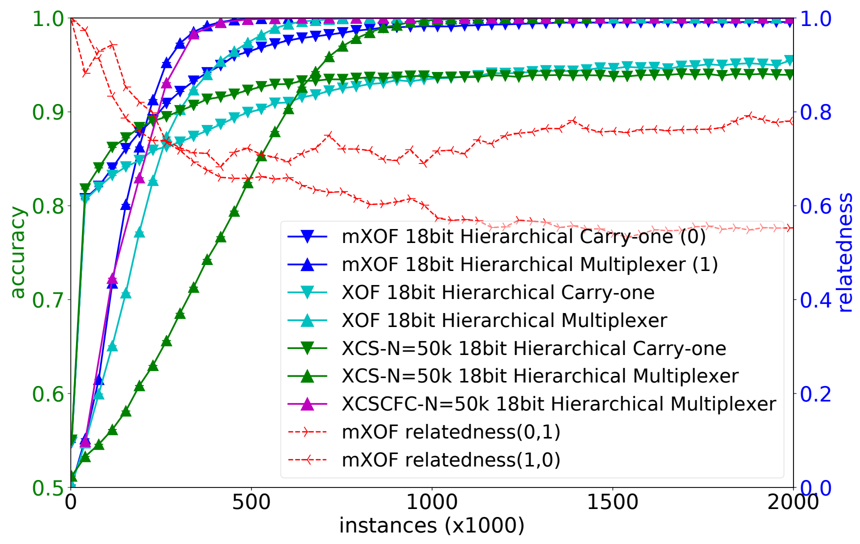
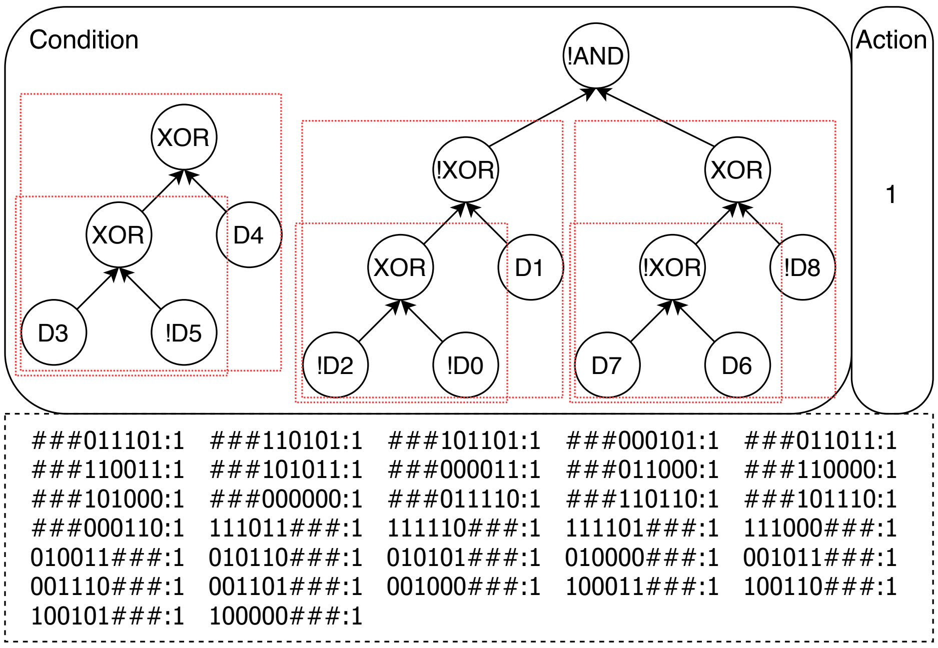


Figure 5.3: Learning performance on multiple 18-bit hierarchical problems. The red lines records the relatedness between two tasks. Note: 0 is Hierarchical Carry-one, 1 is Hierarchical Multiplexer.

eters declines quickly when the accuracies of these problems progress to the accuracy of 100% as the OLs of these tasks diverge with their own patterns (encoded in CFs). After reaching the maximum accuracy, the average relatedness maintains its value at around 0.8 for the relatedness of Hierarchical Carry-one to Hierarchical Multiplexer and 0.6 for the opposite relatedness. The difference in these converged values is due to that the Carry-one layer enables growing CFs further than ones addressing the lower Even-parity layer (e.g. the CFs in red rectangles in Figure 5.4) while the Multiplexer layer does not. Thus, the XOF system for 18-bit Hierarchical Multiplexer problem normally stops growing CFs after discovering the patterns for the Even-parity layer, which are also useful in 18-bit Hierarchical Carry-one problem. In contrast, the higher-level CFs customised for the higher Carry-one layer in 18-bit Hierarchical Carry-one problem are not useful for the other problem.



```
###011101:1   ###110101:1   ###101101:1   ###000101:1   ###011011:1
###110011:1   ###101011:1   ###000011:1   ###011000:1   ###110000:1
###101000:1   ###000000:1   ###011110:1   ###110110:1   ###101110:1
###000110:1   111011###:1   111110###:1   111101###:1   111000###:1
010011###:1   010110###:1   010101###:1   010000###:1   001011###:1
001110###:1   001101###:1   001000###:1   100011###:1   100110###:1
100101###:1   100000###:1
```

Figure 5.4: An optimal rule (prediction 1000) of the 9-bit Hierarchical Multiplexer problem with its equivalent ternary rules. With the generality of 0.375 and the complexity of 9, its generality rate is $1/24$. The CFs in red boxes are reusable grown-patterns for any Hierarchical Boolean problem with parity in the lower layer (addressing the chunks of $\{D_0, D_1, D_2\}$, $\{D_3, D_4, D_5\}$, and $\{D_6, D_7, D_8\}$).

The progress of discovering complexity-efficient CFs for both mXOF and XOF is shown in Figure 5.5. The generality rate of exploit classifiers evolves faster in mXOF since the discovery of complex and efficient CFs in all systems supports each other. Specifically, the optimal set of complexity-efficient CFs required to solve the 9-bit Hierarchical Multiplexer problem must include at least three CFs covering the three non-overlapped 3-successive-bit chunks (see Figure 5.4), which correspond to three underlying bottom-level 3-bit Even-parity problems, or the combinations of these CFs. Such optimal CFs that can cover a 3-bit Even-parity problem must combine 3-bit input using XOR operator (with an arbitrary amount of the NOT function). These CFs are the reusable grown-patterns among all Hierarchical Boolean problems that involve parity. The sample rule in Figure 5.4 is optimal in terms of the complexity efficiency for the 9-bit Hierarchical Multiplexer problem. This rule contains various constructed patterns (in red dashed boxes) that are transferable across the hierarchical problems. These patterns include the CFs covering the bottom-layer problems and the CFs constructed intermediately (in three smaller red boxes).

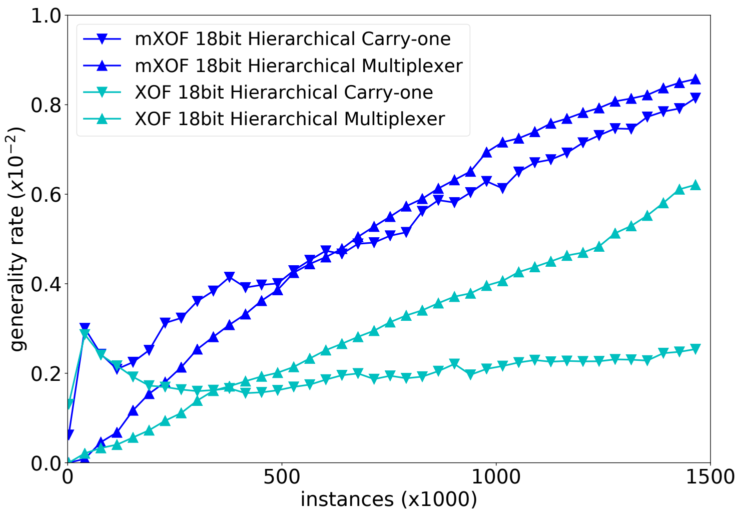


Figure 5.5: Generality rates on 18-bit hierarchical problems.

These shared patterns enable rules using them to generalise and so cover a larger set of instances. The sample rule in Figure 5.4 is equivalent to 32 ternary rules. Because the niche of each ternary rule overlaps partly with another, the coverage of these 32 rules is equivalent to 24 ternary rules. Therefore, the sample rule in Figure 5.4 covers $24/2^6 = 3/8$ of the instance space, where 6 is the number of specified bits in these rules. The practical value of the optimal generality rate of 9-bit Hierarchical Multiplexer problem is $3/8 \times 1/9 = 1/24 \approx 0.0416$.

5.3.2 MTL with Low Relatedness

These experiments was to evaluate mXOF in its ability to prevent negative interactions among unrelated or slightly related tasks. These experiment included two sets of Boolean problems: (1) 37-bit Multiplexer and 11-bit Even-parity problems; and (2) 10-bit Carry-one and 11-bit Even-parity problems. While 37-bit Multiplexer problem does not require complex patterns, 10-bit Carry-one problem and especially 11-bit Even-parity problem can benefit from constructing complex CFs. In the second experiment, some hierarchical patterns from 10-bit Carry-one problem can be useful for 11-bit Even-parity problem, e.g. $D_0 \times (!D_5)$ and $D_7 \times (!D_2)$. However, most of the hierarchical patterns from 11-bit Even-parity problem contain no discriminative information for 10-bit Carry-on problem.

Figure 5.6 shows the learning performances of mXOF on 37-bit Multiplexer and 11-bit Even-parity problems together, and XOF on these two problems separately. The learning curves of mXOF and XOF on corresponding problems have no substantial difference. The learning process of mXOF on 11-bit Even-parity problem can even converge to 100% faster. The relatedness of 37-bit Multiplexer task to 11-bit Even-parity starts low because their OLs start with base CFs encoding original data attributes. Specifically, the starting OL of 11-bit Even-parity system is $(D_0, ..., D_{10})$, while $(D_0, ..., D_{36})$ is the one in 37-bit Multiplexer system. Hence, only the part $(D_0, ..., D_{10})$ in the OL of 37-bit Multiplexer system can be applicable to the other system. It then increases a little before declining together to the relatedness of around 0.2.

Two of the 30 runs of the single system on the 11-bit Even-parity problem were stuck at the accuracy of 50%. In this run, one classifier containing only one base CF in its condition dominated the rule population with very high numerosity and fitness. This is because this problem poses a high chance of local optima for XOF because of the influence of CF-fitness in genetic operations. The diversity from an external task, the 37-bit Multiplexer problem, helps XOF escape the local optima because solving the 37-bit Multiplexer problem only requires XOF to use base CFs, which cover all base CFs that the 11-bit Even-parity problem needs to balance before generalising.

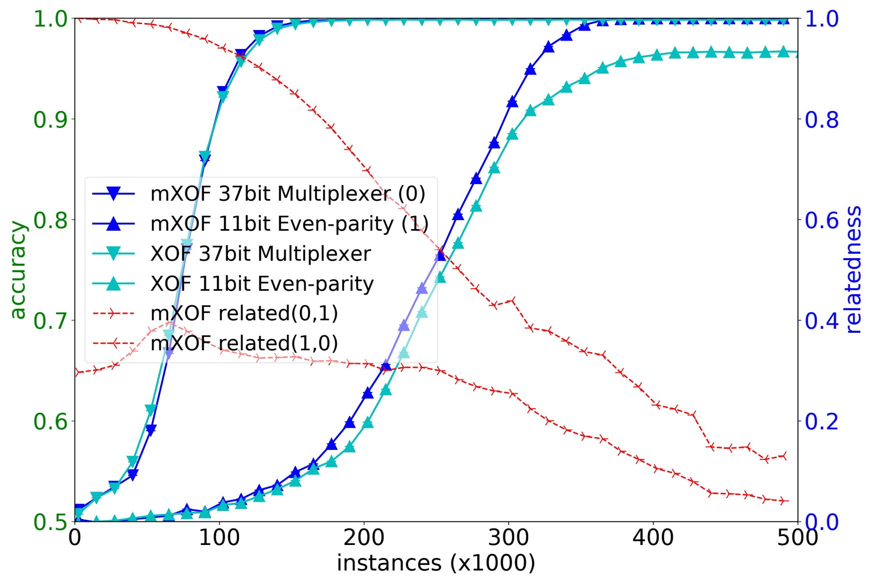


Figure 5.6: Learning performance on 37-bit Multiplexer and 11-bit Even-parity problems. It is noted that the relatedness between the two problems are asymmetric.

Similar trends occur when learning multiple tasks with 10-bit Carry-one and 11-bit Even-parity problems (see Figure 5.7). The learning performances of both tasks in mXOF are not substantially different from learning them separately with XOF. The relatedness on both two directions stays relatively high in the beginning but then declines to the values of near 0.5 when 11-bit Even-parity system starts progressing. The fact that these relatedness parameters stays high in the early phase of both cases in this section seems to suggest that this relatedness estimation does not reflect the relatedness of these tasks. This will be discussed further in Discussion Section.

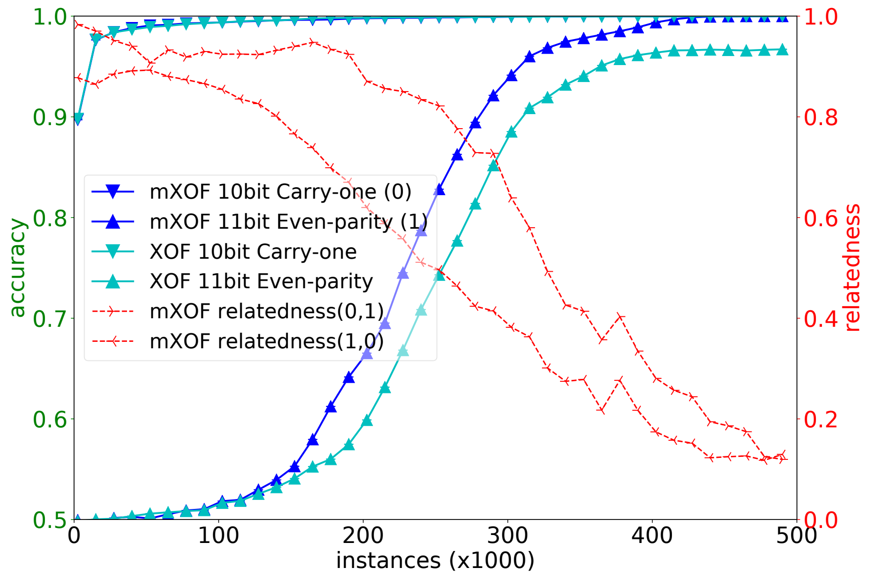


Figure 5.7: Learning performance on 10-bit Carry-one and 11-bit Even-parity problems.

5.3.3 Multi-class Classification using Multiple Binary Classifiers

This section will provide an initial investigation on the performance of mXOF compared with several popular machine learning algorithms on the UCI Zoo dataset. All results were evaluated using 10-fold cross-validation in supervised learning. The baseline algorithms were tested using Weka [49] mostly with default settings except for Multi-layer Perceptron (MLP) and Random Forest. Random Forest was set with the batch size of 200, which was the best result among several tested batch sizes. MLP used two hidden layers of sizes 15 and 10. mXOF scores 95.83% and 96.25% on average with the population size of $N = 500$ and $N = 1000$ for each system respectively (see Table 5.1). These population sizes are considered relatively small for XCSs. These results are competitive compared with other popular machine learning algorithms in this experiment even though the differences are not statistically significant based on the Wilcoxon Signed-Ranks test with $p\text{-}value < 0.05$.

Table 5.1: Results on the UCI Zoo dataset in supervised learning.

Problem	UCI Zoo
Naïve Bayes	95.05%
SVM	92.08%
MLP	$95.91 \pm 0.42\%$
C4.5	92.08%
Random Forests	$96.07 \pm 0.65\%$
mXOF ($N = 500 \times 7$)	$95.83 \pm 1.09\%$
mXOF ($N = 1000 \times 7$)	$96.25 \pm 1.30\%$

Figure 5.8 illustrates the average relatedness of class "reptile" to other classes. The initial class relatedness starts at near 1.0 because all binary classifiers start with the OLs of all original data attributes. All the relatedness values fell to the converged values equivalent to the final relatedness in Figure 5.9 as the systems generalise their rules with fewer discriminative features. The class "reptile" has the most interactive relatedness with four other classes because the optimal rules for "reptile" include the largest number of data attributes. Therefore, it has a high chance to share common building blocks with other tasks. On the contrary, class "bird" needs only attribute "feathers" to be recognised from other classes. Also, this attribute is only needed for class "bird". These two factors result in no link between "bird" and other classes.

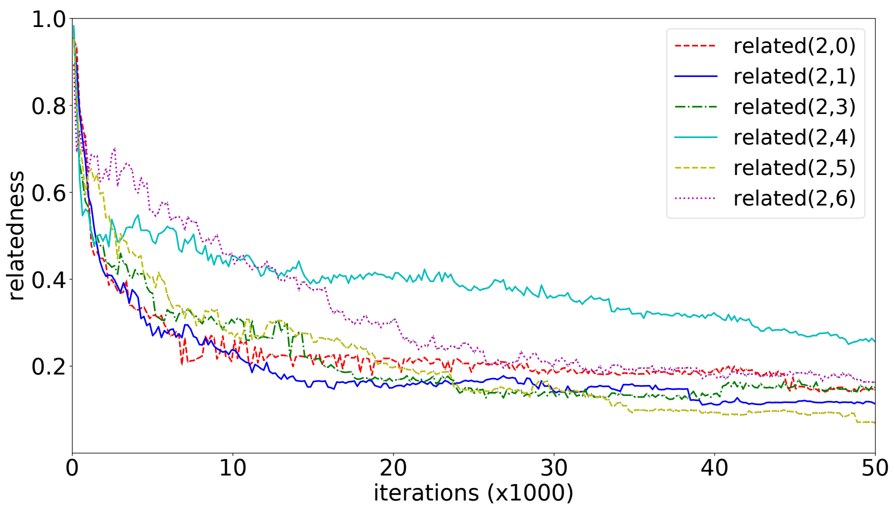


Figure 5.8: The dynamic flow of relatedness of task for class 2 to other tasks/classes (UCI Zoo dataset). Classes numbers correspond to actual classes as follows: 0: mammal, 1: bird, 2: reptile, 3: fish, 4: amphibian, 5: insect, and 6: invertebrate.

5.3.4 Discussions

The above results show that the relatedness among tasks can guide the sharing of constructed knowledge among tasks in mXOF to improve the learning performances of related tasks. The learning processes of each task can benefit from useful CFs found in other tasks. The estimated relatedness among tasks can also lower negative interference among relatively unrelated tasks that could reduce the performance of each task. As a result, the learning performances of these tasks remain unchanged compared with separate learning. On the other hand, the feature sharing among tasks also reinforces the relatedness when useful CFs become common among tasks and thereby dynamically change the relatedness parameters.

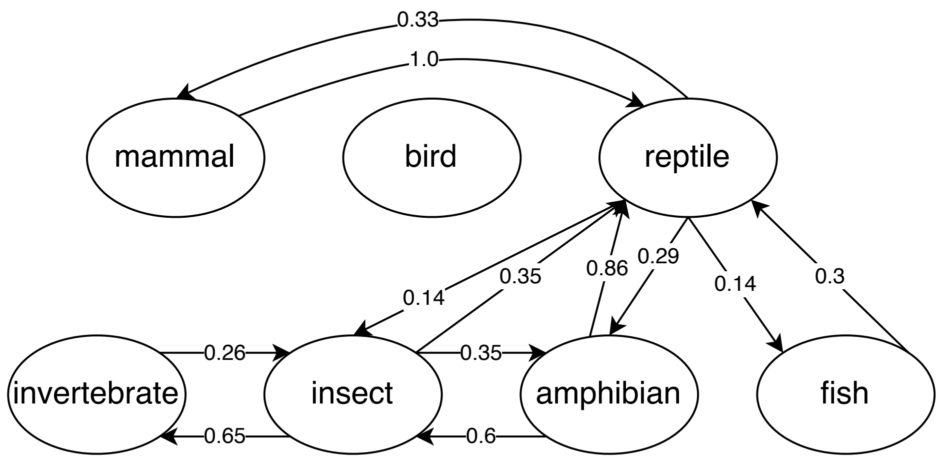


Figure 5.9: The final relatedness of 7 classes in the UCI Zoo dataset. This result in from one of 30 runs.

The high values of relatedness in the early phase of the two experiments in Section 5.3.2 do not show that these tasks are highly related. However, they do share common CFs in the early phase because, in this phase, each system has not grown complex CFs other than base CFs, the common raw input signal. This situation is similar to the beginning phase of human learning to recognise different objects using signals from the same sense. The decreases of these values happen quickly when the 11-bit Even-parity system starts generalising by building up complex patterns to replace original data attributes. This explains why two relatively unrelated tasks have such high relatedness in the early learning phase. Similarly, the relatedness of the 18-bit Hierarchical Carry-one task to the 18-bit Hierarchical Multiplexer task declines quickly to the relatedness values of around 0.6 because these systems build up complex patterns with a limited divergence. They start to diverge with their own complex-pattern discovery including patterns specialised for their problems. The contradiction between the system divergence and feature sharing in mXOF causes the relatedness to balance at around 0.6. In short, the relatedness parameter in our experiments updates itself according to the actual dynamics of constructing tree-based features.

The transferred CFs also provide each system in mXOF external diversity, which can be valuable in escaping local optima. The learning process of mXOF on 11-bit Even-parity problem when learning with 37-bit Multiplexer problem can even converge to 100% faster than XOF because of the small external influence from the task solving 37-bit Multiplexer problem. In the case of highly related tasks (multiple hierarchical problems), this influence results in easier accuracy convergence of mXOF on 18-bit hierarchical problems.

mXOF can also work as a multi-class classifier with competitive results on the UCI Zoo dataset. The supervised-learning accuracy of mXOF tends to increase with larger population size. However, the optimal rules for this dataset mainly use (AND, OR, NOT) logics with the original data attributes. The transferable building blocks are only the original attributes. Therefore, the benefits of growing and sharing complex features among tasks are not applicable. However, the experiments of mXOF on this dataset are an initial investigation to show its potential in solving multi-class classification and demonstrating the links among classes. Theoretically, mXOF could benefit the tasks that require latent features where the improvement of feature construction can result in higher accuracy.

The idea of automatically adjusting the transfer among tasks in mXOF fits well for online learning, especially when tasks share common input signals (original features). For offline tasks with big data, pre-learning investigation of transfer learning with statistical approaches could be more efficient.

5.4 Chapter Summary

This chapter has introduced a multitask online-learning system using multiple XOFs with the ability to automate the feature sharing among tasks automatically. By crafting internal parameters of the hypothesised task relatedness to guide the automatic feature transfer, mXOF can improve the learning performances of individual tasks when they are related. It also reduces transferring harmful signals from other tasks when they are not supportive of a target task. The relatedness parameter based on the OL of XOF estimates reasonably the dynamic commonality of patterns among tasks. The dynamic update of relatedness is essentially useful for learning systems with feature construction, e.g. mXOF, because the benefit of sharing features among tasks may only occur at some specific stages of feature-complexity growth. However, further development and investigation on mXOF is necessary to explore its abilities on a broader range of problems.

Having the problem relatedness measurements can help create network links among target objects of the binary classifiers in mXOF. Therefore, learning more objects/tasks builds up this knowledge network. This network enables links of only specific knowledge that could be useful for a target task. In an AI system with a high volume of accumulated knowledge, this ability is essential to avoid intractable search spaces when querying all knowledge.

Future research can consider mXOF for the context of continual and multitask learning. The reason is that in mXOF, learning a new class only requires spawning a new system without remarkable negative impacts on existing tasks given a proper estimation of relatedness. Learning a new class could take advantage of the bias of previously learned knowledge to acquire relevant knowledge within fewer examples. This is equivalent to human/robot learning to recognise multiple objects using signals from the same sense/sensor.

Because XCS and the OF module can be considered frameworks to be integrated with different representations (for its rules), mXOF is not bound to using only tree-based programs (CFs). Future research could consider integrating mXOF with neural networks to learn real-valued data. This integration could enable combining global optimisation and local optimisation. Furthermore, it could grow complexity-efficient network structures to address meaningful aspects of target tasks instead of black-box networks.

The next chapter introduces a further parallel system that uses multiple CF-based XCSs to learn multiple problems simultaneously and continuously. Instead of transferring features in mXOF, this system accumulates complex knowledge by reusing ruleset functions [3]. This system is designed to solve hard problems by discovering complex decision boundaries instead of complex features. This work develops a parallel system based on layered-learning, XCSCF* [6], to solve Boolean problems at any scale.

Chapter 6

ConCS: A Continual Classifier System for Continual Learning of Multiple Boolean Problems

Human intelligence can simultaneously process many tasks with the abilities to accumulate and reuse knowledge among tasks. These abilities enable solving more complex problems progressively. The recent approach of Layered Learning provides Artificial Intelligence with such abilities but requires human guidance for the order of tasks or only targets a specific problem.

In the previous chapter, a multi-XOF system (mXOF) automates the transfer of CFs and solves multiple problems concurrently. However, mXOF was suitable for multitask learning with classification tasks only as it is focused on the relationships among classes.

This chapter aims to develop a system that solves multiple prediction problems concurrently such that once one is solved, it can contribute to solving others. The hypothesis is that the Evolutionary Computation approach of Learning Classifier Systems, due to the nature of its cooperative

rules, is suitable for this concurrent learning. A system of parallel classifier systems that can solve multiple tasks cooperatively and continually is proposed. Each agent utilises Code Fragments (CFs) to represent knowledge plus reuses CF-based knowledge from other agents to solve its task. Distinct Boolean classification problems that are either dependent or independent are used to test the novel system. Results show that by combining knowledge from simple problems, complex problems including intractable ones for single-agent approaches can be solved at any scale. Not only is human guidance unnecessary for the learning order, but the system also produces the curricula for autonomous learning.

6.1 Introduction

Accumulating and transferring/reusing knowledge are inherent abilities of humans. A human accumulates knowledge through his/her lifetime from the most intuitive concepts and simple skills to increasingly abstract and complex knowledge [108]. This increasing order of knowledge difficulty is important for fast learning progress, but humans do not need strict orders of problems, skills, and lessons to progressively acquire knowledge.

The abilities of knowledge accumulation and reusability are also desirable features for Artificial Intelligence (AI) systems [123, 125] as learning complex problems from scratch faces the challenge of intractable search spaces. Layered Learning (LL) is a sequential learning paradigm that can achieve such abilities [119]. LL enables learning complex knowledge, plus functions to manipulate this knowledge, by incrementally learning a series of sub-tasks and associated component knowledge, where previous knowledge, i.e. functions and skills[1] can bootstrap later tasks. Similarly, continual learning is an AI concept that encapsulates the continuity of

[1]A skill is a function without a returned value, e.g. the loop skill

the sequential learning process with knowledge reusability [112]. These mechanisms are analogous to how humans accumulate knowledge, as mentioned above. LL normally refers to sequential LL by default, which assimilates knowledge components sequentially. Sequential learning requires human guidance to specify a learning order that allows each learning stage to obtain its target knowledge. However, this guidance is not always achievable if the knowledge is new to humans. Also, it limits the autonomy of the AI system. Automatic discovery of the learning order is also a desirable ability for an AI system because it provides understandings on the dependencies among knowledge components. This feature leads to concurrent LL, which is learning all stages of sequential LL in parallel without the requirement of the learning order [110, 139]. However, a concurrent LL system targets only a specific problem. This limits the potential of such systems to be within specific domains.

This work aims to utilise knowledge reusability in a system of multiple distributed learning agents[2] to accumulate knowledge and solve multiple problems for the following benefits. First, the intractable search space of a complex problem can be divided amongst agents to ease the task. By solving more problems, the system can accumulate more knowledge and maximise its problem-solving capability. Second, having multiple agents can minimise the complexity of each agent, which arguably encourages the generality of each agent. Accordingly, the system can flexibly adapt its complexity during its operation by adding or removing agents. Lastly, this system can work as a continuous AI system [85] designed for problem-solving, especially pattern discovery, and state-action-reward/state-class predictions.

Evolutionary Computation (EC) algorithms can learn optimisation problems through building blocks that can facilitate sharing knowledge [3, 63]. There are many attempts to exploit this feature of EC to develop EC algo-

[2]In this chapter, an agent is a complete EC system.

rithms with the ability to reuse knowledge across multiple tasks.

One of the research directions in EC focused on this ability is evolutionary multitasking [100]. This research direction is related to this work in terms of the distribution property but such algorithms are mainly designed for optimisation, rather than classification tasks here.

The use of Code Fragments (CFs), a form of tree-based programs, in LCSs has extended their scalability, particularly by enabling knowledge reusability. This ability makes CF-based LCSs promising algorithms for imitating human-learning abilities. Alvarez et al. showed that an LCS-produced ruleset could be treated as a function, which is reusable in future tasks [3]. As well as transferring the learnt skill or manipulating knowledge, relevant building blocks of knowledge can also be transferred in the form of CFs. He then extended the reusability of ruleset functions to produce XCSCF* [6] with LL. This system was able to discover the logic for the Multiplexer problem domain, so can solve the problem at any given scale. However, these attempts at implementing the ability to reuse knowledge were limited to sequential learning. This required human interventions to specify the numbers of transferred CFs and design a curriculum. Designing a curriculum is a non-trivial task because it requires deep understanding of target problems in advance [11]. Therefore, these systems were strictly limited in flexibility and not appropriate for an autonomous continual learning system.

In this study, a novel continual AI system of multiple XCS-based agents, termed Continual Classifier System (ConCS), is proposed. ConCS is targeted to learn multiple tasks in parallel and continuously. ConCS solves problems with the ability to accumulate knowledge in a knowledge pool and reuse it in novel tasks. The research objectives for ConCS are as follows:

- To develop continual learning in LCSs, which will be tested in complex Boolean problems that are intractable to independent learning.

- To learn curricula through automatically determining the next problem that is most appropriate to address, and thereby removing the requirement of human intervention in layered learning [6, 7].
- To discover knowledge dependencies among problems through learnt solutions. The representation of CFs in solutions should enable a clear understanding of learnt problems.

One may relate the learning paradigm of ConCS with Multitask Learning (MTL) [28]. The general concept of MTL is to learn multiple different tasks together to improve the learning performance of each task. Technically the proposed ConCS also learns multiple tasks together, but it differs from standard MTL in its purpose. ConCS not only improves the performances of increasingly more complex tasks (the complexity of tasks is set *a priori*, and remains fixed in standard MTL) but also serves as a general Boolean problem-solving system. This includes being able to handle unrelatedness among the presented tasks. Furthermore, a general problem-solving system should also be able to solve problems arriving at any arbitrary time, unlike standard MTL where tasks (typically only two) are presented simultaneously. Furthermore, MTL considers optimisation problems, whereas ConCS is specialised on classification problems. For these reasons, the learning paradigm of ConCS incorporates core aspects of LL, continual learning [112], and multitask learning, where the system learns multiple problems/tasks incrementally and continually.

There is an important paradigm shift in designing test problems when switching from learning systems that consider a single problem to continual learning systems. As a human cannot learn integration without first knowing addition, the building blocks (both knowledge and skills to manipulate the knowledge) must be made available. In Genetic Programming (GP), the researcher defines the function and terminal set, while, in ConCS, it is the problems themselves and the skills/functions they provide that are paramount, e.g. once the Boolean "AND" problem is learnt, the “AND” function can be reused along with its component knowledge.

ConCS will be tested with 19 problem types, including regression and classification problems, to demonstrate its learning capabilities. Boolean problems are an appropriate testbed as known solutions exist for comparison, they can be set at increasingly complex scales and are likely to make varying dependencies (i.e. certain problems are unrelated, whereas others have known relationships). The final target problems in this set are four hierarchical problems: Hierarchical Carry-one, Hierarchical Even-parity, Hierarchical Majority-on, and Hierarchical Multiplexer. They are two-layer problems with Carry-one, Even-parity, Majority-on, and Multiplexer problems at the upper layer and a 3-bit Even-parity problem at the bottom layer. These problems have high epistasis, overlapping niches, and benefit from a hierarchy of knowledge being learnt (see Chapter 3). Experiments will run multiple problems at the same time to see whether the system can learn the logic behind all problems without guidance, pseudo simultaneously, i.e. all problems are presented at once where ConCS is to determine the best sequence to solve them. The results are evaluated using the hierarchical problems at scales which no other system has been able to solve without a learning curricula/sequence order provided by humans *a priori*.

6.2 ConCS: Continual Classifier System

The section provides description of the Continual Classifier System (ConCS). ConCS is composed of multiple agents, where each is dedicated to a task (see Figure 6.1). The target of the whole ConCS is to accumulate knowledge from its agents. This global target is expected to support the problem-solving capability of the agents of ConCS. Note the global target is not needed a priori, plus a new one can be given once an old one has been reached without the need for retraining. The proposed system spawns a new agent for each task. The objectives of each agent are not only solving its problem, but also accumulating knowledge from solving its problem into the knowledge pool of ConCS, which is the common goal of the whole ConCS. The problem-solving capability is validated with its ability to scale and comprehend harder problems with increasing complexity.

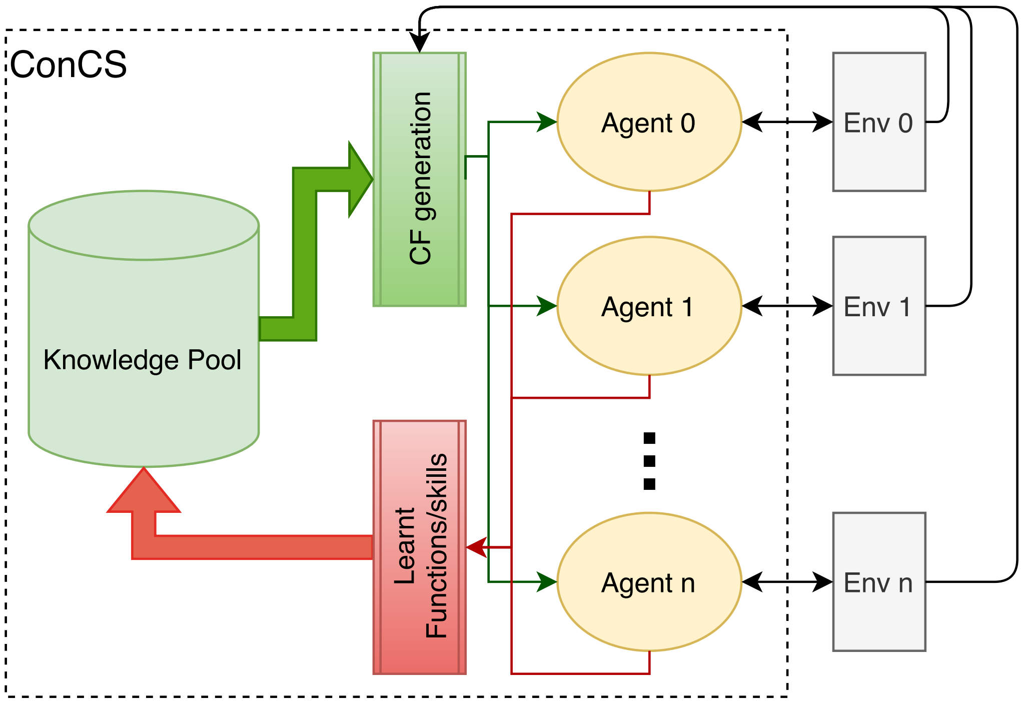


Figure 6.1: A simplified illustration of ConCS. Each environment (Env) represents a task.

Communication among agents in ConCS is indirect and limited to interactions through the knowledge pool. An agent can extract skills and functions from the knowledge pool to reuse after it has learnt a problem, it can append its completed skill/function to the pool. That is, a skill/function maps from input to output, which can be learnt through rules mapping conditions (subsets of input) to actions (output).

Continual multitask learning produces challenges for ConCS. First, ConCS without the curricula might pick up a complex task by chance and get stuck there. The intuitive resolution is that ConCS should focus on the easiest problems, which usually feedback positively with the least effort. This leads to a need for a priority of agents for access to the CPU. Ultimately, ConCS will be configured for truly parallel GPU-based cloud computing but initial development is kept simple to avoid confounding factors. Second, the search space becomes larger as more knowledge is available. To add limits to the search space, the type-fitting XCSCFA introduced in XCSCF* [7] is also used as the learning agent in ConCS to learn both subproblems and target problems. XCSCFA with the type-fitting property can divide the search space into smaller sections by compatible types [7]. This enables accumulating more knowledge without making the search space of each agent intractable. Therefore, there are three key components of ConCS to address these issues: stochastic agent preference, type-fitting XCSCFA, and knowledge management. These components will be described in the following subsections. This chapter also provides a brief explanation of all subproblems designed to bootstrap the system with low-level knowledge.

6.2.1 Type-fitting XCSCFA

A new implementation of XCSCFA with a type-fitting property in generating CFs is employed as the common algorithm for the agents of ConCS. The type-fitting property is a novel CF-generation method (described below in Section 6.2.1.1). This version of XCSCFA generates verifiable CFs as it enables the type-fitting property in generated CFs. This property guarantees connected nodes within generated CFs to be compatible with one another and the CFs′ input and output to be compatible with the problem (environment). Although ConCS is not limited to using only XCSCFA, this algorithm is suitable to learn the high-level logic behind the tested problems as it can address both regression and classification problems. ConCS must be able to spawn an agent with a suitable algorithm (or multiple agents) if no prior experience regarding a new problem is detected.

The term “base CFs” from XOF is used to indicate pre-provided CFs, which are all candidates for leaf nodes of CFs. Base CFs contain information for both input attributes and the constant L to store the length of an input instance in XCSCFA. However, providing a constant L is infeasible because many problems have variable sizes. Therefore, in addition to base CFs for attributes, XCSCFA provides a base CF listing all attributes $attlst$ in the order provided by the problem. It is assumed that this is also a more general way of providing inborn knowledge.

6.2.1.1 Type-fitting Code Fragments

Inspired by Strongly-Typed GP [96], the type-fitting method for generating CFs, called Typed-CFs (T-CFs), was introduced to reduce the search space by fitting each node with only compatible inputs and outputs. T-CFs are designed to create eligible and meaningful CFs. Being meaningful refers to the compatibility of each function node, where the output type of a function in the node cf_i must be compatible with the input types of the function in the node cf_j that takes cf_i as input. Being eligible includes two conditions: the output type of root node function must be compatible with at least one of the action types of the problem, and the leaf nodes are CFs from the set of base CFs. Ultimately, type-fitting method keeps learning agents from generating unworkable CFs.

Accordingly, generating T-CFs applies a top-down recursive process of generating tree nodes, i.e. the function *genNode* illustrated in Algorithm 6.1. In ConCS, the depth limit of CFs is kept as 2 as in the original definition of CFs [63]. The tree depth is the highest count among all possible accumulated levels of all function nodes when traversing from root node to leaf nodes. Most functions have an equal level of 1 including axiomatic and learnt functions. Only function constant, which output an unchanged value, is considered to not adding any depth to CFs.

Generating a new CF needs to match with the action types of the problem and available output types from the base CFs. First, the top node of a T-CF must employ a function with output type(s) compatible with the action type(s) of the problem. Then the process recursively builds lower-level nodes that satisfy the type-fitting property. At any point when generating nodes, there is also a fixed probability of 0.5 for generating a leaf node from base CFs, which stops the CF from going any deeper.

There are four possible types in this implementation: Boolean, integer, float (real numbers), and list. While Boolean variables are compatible with integers and floats, and integers are compatible with floats, the compatibility does not follow the opposite way. Lists are not compatible with other types. Several improvements for XCSCFA [62] are proposed to process redundant genotypes of functions (function versions). The following section will describe one problem related to function genotypes. Another case of processing genotypes is in compacting functions, which will be introduced in Section 6.2.2.1.

6.2.1.2 CF Equality

Two CFs are equal if they have the same genotype. However, because there can be distinct versions of the same function, the definition of having the same genotype varies case by case. For example, in a problem, if two CFs have the same genotypes except that two corresponding nodes have different genotypes of the same reused function, they are considered unequal in general learning processes. This enables reusing the diversity caused by function genotypes. This diversity is rational because two genotypes of a function can behave unevenly in problems other than the one that produces the function. On the contrary, the inequality caused by function genotypes is ignored when checking the equality of classifiers during extracting functions (see Section 6.2.2.1). Even though they have different versions of functions f, they do not demonstrate any distinction of knowledge. Also, when two classifiers with only such differences are always

Algorithm 6.1 T-CFs are generated based on a recursive function for generating nodes. The function is given the set of action types T_a, the type set of base CFs T_b, the expected output types T_o, the expected input types T_i, the intermediate level l (starting from level 2 at the root node with the maximum depth), and a clustered set of all functions S_f. Each function has a level ($f.level$) that adds up the depth of CFs when traversing from root node to leaf nodes.

```
1: procedure GENNODE($T_o, T_i, l$)
2:     Set an empty set of compatible input types for recursively generating the input nodes of the returned node (in this function) $T_{i'} = \phi$
3:     if $l = 2$ then
4:         Output types $T_o = T_a$
5:     if $l = 1$ then
6:         Output types $T_{i'} = T_b \cup \{integer\}$
7:     Filter function set $S_{filtered}$ from $S_f$ by required output types $T_o$ and input types $T_i$
8:     Function $f = randomSelect(S_{filtered})$
9:     for input at position (index) $i$ of all required inputs of $f$ do
10:        if $l - f.level > 0$ and $random[0, 1) < 0.5$ then
11:            input $i$ of $f$=GENNODE(types of input $i$ of $f$,$T_{i'}, l - f.level$)
12:        else
13:            Set of compatible base CFs $S_{bCF} = \phi$
14:            if type integer $\in$ types of input $i$ of $f$ then
15:                $c = randomSelect([1, ..., len(Atts)])$
16:            for $cf_{base}$ in all base CFs do
17:                if $cf_{base}.out\_types \& types of input$ i$of$ f$\neq \phi$
    then
18:                    Add $cf_{base}$ to $S_{bCF}$
19:            $input$ i $of$ f= $randomSelect(S_{bCF})$
```

correct in a target problem, they should behave equally throughout the course of learning the problem.

Based on the above principles, the comparison of any pair of CFs is a recursive process as illustrated in Algorithm 6.2. Step 3 is the key step as the comparison of two corresponding function nodes depends on the flag $ignore\_func\_ver$. Two nodes are only considered equal if they use the same function with the same version when the flag $ignore\_func\_ver$ is $False$, which counts the difference of function versions. Otherwise, they just need to use the same function to be considered equal. Once the two current nodes are considered unequal, a comparison value of $False$ will be recursively returned to the outermost layer of the comparison process. In the case where the current nodes are considered equal, the process will ignore them and traverse towards their leaf nodes until finding a returned $False$ or no unequal result on all corresponding nodes of two CFs, i.e. equal CFs.

6.2.2 Knowledge Management

The knowledge pool is the collection of built-in and obtained skills/functions/CFs. In other words, it is the function set listing all available knowledge (how features through functions/skills are related to higher order features (CFs) and ultimately to actions). Agents search for solutions by combining functions from this function set and their base CFs. At the beginning, the knowledge pool has prerequisite knowledge, termed built-in axioms (see Table 6.1). These built-in axioms must include the building blocks required to construct solutions for the target problems (see Section 6.2.3), which is common practice in GP algorithms [74]. In addition to the necessary building blocks, the knowledge pool also provides general functions for Boolean problems that might or might not be useful, with the assumption that ConCS should be able to choose the appropriate functions. While the majority of these functions are general knowledge that can be reused in

Algorithm 6.2 Comparing two CFs, cf_i and cf_j. The parameter $ignore\_func\_ver$ defines whether the difference of function versions in corresponding nodes makes the comparison return $False$. $cf.function$ is the function in the root node of cf. $cf_j.func\_version$ is the version of the function in the root node of cf. $len()$ is a function returning the number of elements in the input.

```
1: procedure CF_EQUALS($cf_i, cf_j, ignore\_func\_ver$)
2:     if $cf_i = cf_j$ (check reference/pointer) or both CFs
           are constant CFs with equal constants then return $True$
3:     if $cf_i.function \neq cf_j.function \vee (\neg ignore\_func\_ver \wedge$
           $cf_i.func\_version \neq cf_j.func\_version)$ then return $False$
4:     if $len(cf_i.branches) \neq len(cf_j.branches)$ then return $False$
5:     if the branches of $cf_i.function$ are exchangeable in order then
6:         for root-node branch $sub\_cf_i$ of $cf_i$ do
7:             a match not found yet $matched = False$
8:             duplicate a list of root-node branches of $cf_j$ in $branches_j$
9:             for root-node branch $sub\_cf_j$ in $branches_j$ do
10:                if CF_EQUALS($sub\_cf_i, sub\_cf_j, ignore\_func\_ver$) then
11:                    found a match $matched = True$
12:                    remove $sub\_cf_j$ from $branches_j$
13:                    break out of the most recent loop
14:            if $\neg matched$ then return $False$
15:    else
16:        for $i \in [0, ..., len(cf_i.branches) - 1]$ do
17:            if $\neg$EQUALS($cf_i.condition[i], cf_j.condition[j], ignore\_func\_ver$)
                   then return $False$
18:    return $True$
```

many other problems, some of them are tailored for these problems and may not be generally applied.

Table 6.1: Axiomatic skills and functions. The operator $x[a, b]$ is to extract a list of items of the list x starting from the index a to, but not including, the end index b. Binary numbers are in the form of list $(0, 1)$. $\oplus$ and $\ominus$ are binary addition and subtraction respectively.

Skills & Functions (tag)	No. Inputs (inputs)	Input types	Output type	Operation
AT (@)	2 (x_0, x_1)	list, integer	any type	$x_0[x_1]$
LENGTH (len)	1 (x_0)	list	integer	$len(x_0)$
AND ($\wedge$)	2 (x_0, x_1)	Boolean	Boolean	$x_0 \wedge x_1$
OR ($\vee$)	2 (x_0, x_1)	Boolean	Boolean	$x_0 \vee x_1$
XOR (x)	2 (x_0, x_1)	Boolean	Boolean	x_0 xor x_1
NOT ($\neg$)	1 (x_0)	Boolean	Boolean	$\neg x_0$
FLOOR (floor)	1 (x_0)	integer, float	integer	$\lfloor x_0 \rfloor$
CEIL (ceil)	1 (x_0)	integer, float	integer	$\lceil x_0 \rceil = \lfloor x_0 + 1 \rfloor$
SUMMATION of list items (sum)	1 (x_0)	list	integer, float	$sum(x_0)$
ADD (+)	2 (x_0, x_1)	integer, float	integer, float	$x_0 + x_1$
SUBTRACT (subt)	2 (x_0, x_1)	integer, float	integer, float	$x_0 - x_1$
MULTIPLY (mul)	2 (x_0, x_1)	integer, float	integer, float	$x_0 \times x_1$
DIVIDE (div)	2 (x_0, x_1)	integer, float	integer, float	x_0 / x_1
GREATER (isGreater)	2 (x_0, x_1)	integer, float	Boolean	$x_0 > x_1?$
HEAD (head)	2 (x_0, x_1)	list, integer	list	$x_0[0 : x_1]$
TAIL (tail)	2 (x_0, x_1)	list, integer	list	$x_0[x_1 : len(x_0)]$
GENERAL LOOP (loop)	2 (x_0, x_1, x_2)	function, list, integer	list	convolve x_1 by function x_0 with problems size x_2
LOG2 (log2)	1 (x_0)	integer, float	integer, float	$log_2(x_0)$
BINADD (binadd)	2 (x_0, x_1)	list (of Boolean)	list (of Boolean)	$x_0 \oplus x_1$
BINSUB (binsub)	2 (x_0, x_1)	list (of Boolean)	list (of Boolean)	$x_0 \ominus x_1$
BIN2DEC (bin2dec)	1 (x_0)	list (of Boolean)	integer	convert Binary number to decimal
MODULO (mod)	1 (x_0)	integer	integer	$x_0 \% 2$
CONSTANT (c)	0 ()	N/A	integer	return a constant number

The *general loop* is a general skill that requires a core process, i.e. a function, given by the input x_0 to become a function. *General loop* iteratively applies function x_0 on input list x_1 with a moving starting-point s_{it} (Figure 6.2). In the first iteration, function x_0 processes x_1 from the first item. At each iteration, the starting point on x_1 moves x_2 steps from the preceding iteration to extract the input for x_0. The loop ends when the starting point moves beyond the end of x_1. The output of a *loop* is the concatenated list of outputs of x_0 in all its iterations.

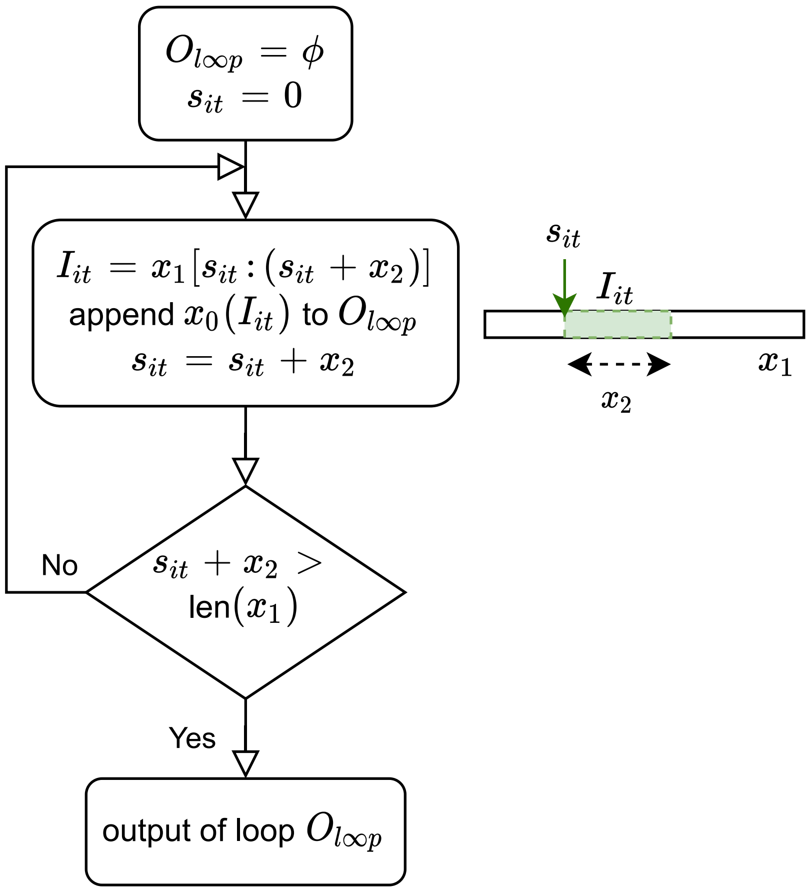

$O_{loop} = \phi$
$s_{it} = 0$
$I_{it} = x_1[s_{it} : (s_{it} + x_2)]$
append $x_0(I_{it})$ to O_{loop}
$s_{it} = s_{it} + x_2$
s_{it}
I_{it}
x_1
x_2
No
$s_{it} + x_2 >$ len(x_1)
Yes
output of loop O_{loop}

Figure 6.2: Flowchart of a function with the loop skill. This function iteratively applies another function x_0 on x_1 with a step size of x_2 and concatenates the iterative output in O_{loop}.

The initial subproblems are not divided any further although it was demonstrated that CF-base XCSs could learn such functions from even smaller subproblems [3]. Future work with ConCS will explore the intellectually interesting question of "what are smallest axioms that can initialise learning?"

6.2.2.1 Function Post-processing

When an agent solves a problem successfully for more than 500 instances, it will try to post-process and extract a solution from its population to form a function for its trained problem. The goal is to provide compact and readable rulesets for extracted functions that can be efficiently compared with existing functions and future learnt functions. The compaction step here is not strict as it allows any two CFs with different genotypes and the same behaviour (logic) to co-exist [5].

The function extraction is shown in Algorithm 6.3 with four main post-processing steps. Firstly, the agent selects only experienced ($exp \geq \theta_{GA}$) and accurate classifiers ($err < \theta_0$) from its population. Then, it finds the highest fitness f_{max} in its classifier population. If the classifier with the highest fitness ($f = f_{max}$) is a completely general rule (no specified attribute in its condition), then all classifiers having low fitness (i.e. not comparable to f_{max} or $f < 0.5 \times f_{max}$) are filtered out. Otherwise, all the classifiers after the first step are kept. The third processing step is to merge all classifiers with the same condition parts and the same action tree-structures except for having different versions in corresponding function nodes (same functions with different versions at the same positions of equal tree structures). The comparison of two classifiers involves comparing two trees, where the difference of function versions are ignored (see Section 6.2.1.2 for the justification). Next, a subsumption step on the remaining classifiers of $[P]$ is executed to remove all over-specific classifiers (step 15).

Algorithm 6.3 Post-processing the rule population $[P]$ to add new functions to the knowledge pool.

```
1: remove inexperienced or inaccurate classifiers with $cl.exp \leq \theta_{GA}$ or
   $cl.err > \theta_0$ or $cl.prediction < 1000$ from $[P]$
2: set $f_{max}$ as the highest fitness of the remaining classifiers
3: **if** $f_{max}$-classifier is general (its condition is empty) **then**
4:     finding a subset of general classifiers $general\_pop = \emptyset$
5:     **for** each $cl$ in $[P]$ **do**
6:         **if** $len(cl.condition) == 0 \wedge cl.fitness \geq 0.5 \times f_{max}$ **then**
7:             $general\_pop.add(cl)$
8:     **if** $general\_pop \neq \emptyset$ **then**
9:         use the subset of general classifiers instead $[P] = general\_pop$
10:    duplicate $mirror\_pop = [P]$
11: **for** $cl_i$ in $mirror\_pop$ **do**
12:     **for** $cl_j \neq cl_i$ in $mirror\_pop$ **do**
13:         **if** $cl_i.equals(cl_j, ignore\_func\_ver = True)$ without considering
            function versions in the function nodes **then**
14:             keep one classifier in $\{cl_i, cl_j\}$ with less computation cost
                and remove the other from $[P]$
15: do subsumption on $[P]$
```

ConCS needs to check the equality of ruleset-functions to avoid adding the same function more than once to the knowledge pool, which undesirably enlarges the search space of all agents. In this case, two functions are considered equal if they contain the same ruleset. ConCS confirms the equality of two ruleset-functions by matching all rules of the ruleset of one function to all rules in the rulesets of other ruleset-functions.

6.2.3 Target Problems

Although real-world domains contain reusable patterns, existing benchmark datasets are often separate in the patterns they contain, e.g. UCI Zoo, Wisconsin Breast Cancer, and Sonar datasets, where a discovered sample distribution is not present in another dataset. Hence, this study needs a set of problems with feature patterns that are constructed from sub-patterns to test the scaling capability of ConCS in related problems. There is also a need of separate problems as continual learning might face independent problems with distinct patterns. Boolean domains satisfy these criteria because they have known solutions and are interrogable. Therefore, target problems are four hierarchical problems and 15 subproblems to be solved continually and simultaneously with the hierarchical problems. These subproblems provide knowledge that is potentially prerequisite for the target problems. In the Multiplexer domain, most of the subproblems are identical to the subproblems used in XCSCF* [7].

The target problems, including subproblems, were selected with variable scales, i.e. lengths of input bits. This requires successful solutions, if any, to be scale invariant. The initial experiments show that, when learning fixed-scale problems, constants (in the form of CFs) (see Section 6.2.1.1) can contribute to generating solutions that are only valid at the fixed scale, which inhibits scaling. Being scale invariant means that successful solutions are capable of solving these problems at any scale. The description of functions to be learnt alongside their anticipated operations are given in Table 6.2.

Address Length of Multiplexer given Problem Size is a regression problem that determines the length of address bits k of a Multiplexer given the problem size $(2^k + k)$ as the input x_0. The anticipated operation is $[log_2(len(x_0))]$, where $\lfloor . \rfloor$ is the floor operator (see Table 6.2).

Address Length of Multiplexer given Bitstring also outputs the same

Table 6.2: To be learnt functions/skills. A few of them are highly reusable in general cases. $len()$ is a function to return the length of the input with type $list$. $(x_0, x_1, ...)$ are inputs of these functions.

Id	Functions (abbreviations)	Inputs	Input types	Output type	Anticipated operation to be learnt
0	Address Length given Mux size	x_0	integer	integer	$\lfloor log_2(x_0) \rfloor$
1	Address Length given Mux bitstring	x_0	list	integer	$\lfloor log_2(len(x_0)) \rfloor$
2	Address Bits	x_0	list	list	$x_0[0 : \lfloor log_2(len(x_0)) \rfloor]$
3	Decimal value of Address Bits	x_0	list	integer	$bin2dec(x_0[0 : \lfloor log_2(len(x_0)) \rfloor])$
4	Data Bit Position	x_0	list	integer	$\lfloor log_2(len(x_0)) \rfloor +$ $bin2dec(x_0[0 : \lfloor log_2(len(x_0)) \rfloor])$
5	General Multiplexer (mux)	x_0	list	Boolean	$x_0[\lfloor log_2(len(x_0)) \rfloor +$ $bin2dec(x_0[0 : \lfloor log_2(len(x_0)) \rfloor])]$
6	Half String Size	x_0	list	integer	$len(x_0)/2$
7	First Half	x_0	list	list	$x_0[0 : (len(x_0)/2)]$
8	Second Half	x_0	list	list	$x_0[(len(x_0)/2) : len(x_0)]$
9	Binary Addition of 2 halves	x_0	list	list	$x_0[0 : (len(x_0)/2)] \oplus$ $x_0[(len(x_0)/2) : l]$
10	Length of Binary Addition	x_0	list	integer	$len(x_0[0 : (len(x_0)/2)]$ $\oplus x_0[(len(x_0)/2) : l))$
11	General Carry-one ($carr$)	x_0	list	Boolean	$len(x_0[0 : (len(x_0)/2)] \oplus$ $x_0[(len(x_0)/2) : l]) > len(x_0)/2$
12	Sum Modulo 2	x_0	list	Boolean	$sum(x_0)\%2$
13	General Even-parity ($epar$)	x_0	list	Boolean	$sum(x_0)\%2 = 0?$
14	General Majority-on (maj)	x_0	list	Boolean	$sum(x_0) > len(x_0)/2?$
15	Hierarchical Multiplexer	x_0	list	Boolean	$mux(loop(epar, x_0, 3))$
16	Hierarchical Carry-one	x_0	list	Boolean	$carr(loop(epar, x_0, 3))$
17	Hierarchical Majority-on	x_0	list	Boolean	$maj(loop(epar, x_0, 3))$
18	Hierarchical Even-parity	x_0	list	Boolean	$epar(loop(epar, x_0, 3))$

output with the previous problem but this problem takes in the Multiplexer bitstring as the input instead of the problem size.

Address Bits is an aspect of the Multiplexer domain. The output of this problem is the list of address bits of a Multiplexer given the input bitstring. The output should the first k bits of the input bitstring of size $(2^k + k)$.

Decimal Value of Address is to find the position of the data channel in a Multiplexer and connect the channel to the Multiplexer output. The expected position here is the index in the data channels only without taking into account the address bits. Therefore, the problem output is the decimal value of the address-bit binary number. Thus, this problem considers converting the address bits to a decimal value given the input bitstring containing both the address bits and data channels.

Data Bit Position outputs the channel position, which connects to the output of a Multiplexer, in the input bitstring. Thus, the learning agent needs to take into account the address bits as well. This problem is anticipated to require a combination of knowledge from the "Decimal Value of Address" problem and the "Address Length of Multiplexer given Bitstring" problem.

Variable-size Multiplexer is equivalent to a general Multiplexer problem where the input has a varied length. Specifically, this problem samples environment states from 3-bit, 6-bit, 11-bit, and 20-bit Multiplexer problems. This problem requires the value at the data channel connecting to the output, i.e. the output of a Multiplexer circuit (see more details in Chapter 3). The final solution of this problem is anticipated to solve Multiplexer problems at n-bit scales.

Sum Modulo 2 is to determine whether the total number of bits 1 in the input bitstring is even or odd. Its anticipated output is the summation of the individual bits in the input modulo 2. Variable-size Even-parity requires $True$ if the number of bits 1 in the input bitstring is even and $False$ otherwise. It is variable in size (from 1-bit to 11-bit Even-parity problems) to encourage only general solutions for the Even-parity problem domain. By being general, the solution can solve the problem at any scale. The Even-parity problems with relatively small scales (e.g. from 8 bits onwards) are already intractable to standard XCS using the ternary encoding as XCS cannot generalise to solve the problem but must form a one-to-one mapping of instances to rules.

Half String Size is a regression problem, which requests the learning agent to predict the half-length of the input bitstring. This problem can support both the Carry-one problem domain and the Majority-on problem domain.

First Half of Input requires the learning agent to output the first half of the input bitstring. The output is considered a binary number represented by a list of $0s$ and $1s$.

Second Half of Input is similar to the "First Half of Input" problem but the anticipated output is the latter half of the input bitstring.

Binary Addition of Two Halves requires the learning agent to add the outputs of the two preceding problems, which are base-2 numbers represented in the form of binary lists. Outputs are also binary numbers represented by lists of $0s$ and $1s$.

Length of Binary Sum is based on the output of the preceding problem. The anticipated output of this problem is the length of the binary addition. This is to learn to predict the length of the binary number resulted by adding the binary number of the first half and the binary number of the second half.

Variable-size Carry-on requires the general logic behind the Carry-one problem domain. It is to determine whether 1 is carried at the highest bit when adding the binary number of the first half and the binary number of the second half. The sizes of this problem vary from 2 bits to 12 bits.

Variable-size Majority-on is similar to normal Majority-on problems, which requires $True$ if more than half bits of the input are 1, and $False$ otherwise as the output. Sizes of the input strings vary from 1 bit to 7 bits.

Hierarchical (variable-size) Multiplexer/Carry-one/Even-parity/Majority-on are to train ConCS to learn the logic of the four Hierarchical problems of Multiplexer/Carry-one/Even-parity/Majority-on. However, there are local solutions that are only valid for fixed scales and not generalisable. Hence, these subproblems are designed to be variable in size. All these Hierarchical problems use the same low-layer logic of 3-bit Even-parity to encode the input to the high-layer component. The scale of Hierarchical Multiplexer and Carry-one problems is 18 bits, while the other two Hierarchical problems have the scale of 15 bits.

6.2.4 Stochastic Task Preference

ConCS' preference is to prioritise agents with higher learning progress. A simple repeated Roulette Wheel selections based on agents' learning progress determines which agent to run at each iteration until there are no more operating agents. This selection enables stochasticity, where even agents with temporarily low progress are still run. In this work, the learning progress is defined as a parameter measured by the absolute accuracy and the improvement of the accuracy of the agent:

$$progress = max(0.1 \times accuracy_{adj}, \Delta accuracy), \tag{6.1}$$

where $accuracy_{adj}$ is adjusted to initially start at approximately 0.5 for all agents and 1.0 when the problem is solved. This leads to an initial progress value of approximately 0.05 when there is no increase in accuracy. The adjustment is as follows:

$$accuracy_{adj} = \begin{cases} \frac{accuracy\ jactionsj\ \ 1}{jactionsj\ \ 1} \times 0.5 + 0.5 \text{ if classification,} \\ accuracy/2 + 0.5 \text{ if regression} \end{cases} \tag{6.2}$$

It can be inferred that the frequency of updating agent progress is equal to the frequency of updating agent accuracy, which is set to once every 500 iterations (250 explored instances) for all agents. The progress can be adapted autonomously but this work will investigate a simple estimation.

6.3 Experiments

The proposed ConCS was evaluated by solving 19 different problem types. Each agent is a type-fitting XCSCFA with a common configuration. The experiments are also to examine whether the system (without human-guided customisation) can work on all problems. The population size of all agents is equal to 1000, which is considered small in the literature of XCSs. Additionally, the minimum number of actions in the match set θ_{mna} was set to 4 for both classification and regression problems. This value of θ_{mna} encourages each agent to create more genotypes to increase the chance of obtaining the desired CF-action.

Each experiment was run 30 times with 30 fixed random seeds. The stopping criteria were when the agent consistently maintained 100% accuracy for at least $50{,}000$ instances, or when it reached the maximum learning instances for each agent, i.e. $2{,}000{,}000$ instances, which was chosen to be arbitrarily large.

The first experiment of ConCS was run on 19 problems simultaneously with 19 agents starting at the same time to show the discovery of the network of knowledge (corresponding to 19 problems). Then, the performances of ConCS were compared with those of XCSCF* using type-fitting XCSCFA, on different sets of problems. In these experiments, XCSCF* shared the same configuration and the problem sets with agents of ConCS, except that XCSCF* runs sequentially. ConCS and XCSCF* are also compared with XCS and XCSCFC in solving target problems at specific scales. XCS and XCSCFC are configured with their empirical configurations, which require much larger population sizes. Solutions yielded by these two approaches are limited to solving the tested problems at fixed scales. Additionally, another comparison of ConCS with XCSCF* and traditional GP in terms of accuracy in supervised learning was executed to assess the generalisation of ConCS′ solutions.

Finally, an extra experiment with random arrivals of the 19 problems was performed to show the ability of ConCS to learn continually and multiple tasks in parallel. A problem was randomly chosen as a starting point. The other 18 problems were initialised at random time from the starting point using a uniform random generator within the range of 0 to one hour. The arrival times were generated once and fixed in all 30 runs.

6.3.1 Discovered Knowledge

Experiments on all 19 problems demonstrated the ability of ConCS to achieve 100% performances on all problems in all 30 runs (see Section 6.3.2 for details on learning performances). Table 6.3 shows the learnt solutions from the agents. These acquired solutions are interpreted from the actions of the rules in compacted solutions. It is noted that rule conditions of all these acquired solutions are composed of only *don't care* (all are "#" as XCSCFA also use the ternary alphabet for rule conditions), i.e. matched all possible inputs, which is expected as the learnt CFs in rule actions address all inputs correctly.

For several problems such as the Even-parity problem, solutions acquired for the same problem involve distinct genotypes in the CF-action, where some contain bloat or inefficient solutions (see Table 6.3). However, the bloat in such solutions is limited. It is trivial to verify that diverse solutions for each problem are equal. Also, as acquired solutions are highly interpretable, it is straightforward to confirm that these solutions in Table 6.3 are identical to the logic of the 19 given problems in Table 6.2. How-

Table 6.3: Learned functions/skills. $attlst$ is a based CF representing the list of all inputs. Other inputs are x_0, x_1,etc. i, k refer to arbitrary integers in constants (CFs).

Functions & Skills	Function Name	Learned Solutions
Address Length given Multiplexer size	mux_addr_length	$floor(log2(x_0))$
Address Length given MUX attributes	$mux\_addr\_length2$	$mux\_addr\_length(len(attlst))$
Address Bits	$mux\_addrbits$	$head(attlst, mux\_addr\_length2(attlst))$
Decimal value of Address Bits	mux_gate	$bin2dec(mux\_addrbits(attlst))$
Data Bit Position	$mux\_databit$	$add(mux\_gate(attlst),$ $mux\_addr\_length2(attlst))$
Variable-size Multiplexer	mux	$@(attlst, mux\_databit(attlst))$
Hierarchical Multiplexer	$hpar$	$mux(loop(epar, x_0, 3))$
Sum Modulo 2	$epar\_mod\_2$	$mod(sum(attlst), 2)$
Variable-size Even-parity	$epar$	$\neg(epar\_mod\_2(attlst))$ $greater(c(1), epar\_mod\_2(attlst))$ $greater(div(i, k), epar\_mod\_2(attlst)$ $(i \leq k)$
Hierarchical Even-parity	$hpar$	$epar(loop(epar, x_0, 3))$ $epar(loop(epar, x_0, 1))$
Half String Size	$half\_length$	$div(len(attlst), 2)$ $mul(len(attlst), div(c(i), c(2i)))$
First Half	$car\_headstring$	$head(attlst, half\_length(attlst))$
Second Half	$car\_tailstring$	$tail(attlst, half\_length(attlst))$
Binary Addition of two halves	car_binadd	$binadd(car\_headstring(attlst),$ $car\_tailstring(attlst)))$
Length of Binary Sum	$car\_lenbinadd$	$len(car\_binadd(attlst))$
Variable-size Carry-one	$carr$	$greater(car\_lenbinadd(attlst),$ $half\_length(attlst))$
Hierarchical Carry-one	$hcar$	$carr(loop(epar, x_0, 3))$
Variable-size Majority-on	maj	$greater(sum(x_0), half\_length(attlst))$
Hierarchical Majority-on	$hmaj$	$maj(loop(epar, x_0, 3))$

ever, in certain cases, the evolved solutions deliver new and unexpected insights into the problems.

The final solutions from Table 6.3 on all problems drew a network of knowledge. This network enables learning about the dependencies of one problem on others. Figure 6.3 illustrates the learnt network of knowledge. An arrow directed from a problem A or a pre-provided function f to a problem B means the solution for B uses the solution (i.e. learnt function) discovered from learning problem A or function f. ConCS found that the *Half String problem* is one of the most generally reused as it is used in at least four other problems $(car\_headstring, car\_tailstring, carr, maj)$. For the innately provided skills, general loop $loop$, constant function c, and length len are the three most popularly used functions. The constant function c is the most used one as it creates the based CF $attlst$ for almost all solutions. On the contrary, others functions, such as binary operators $(\wedge, \vee$,x$)$ and binary subtraction, were found redundant as they were never used in any learnt solution.

6.3.1.1 New Understanding of Hierarchical Even-parity Problem

Table 6.3 lists all discovered solutions for the Hierarchical Even-parity $hpar$ problem. In addition to the expected first rule on the Table, ConCS also yields the second strange rule in all 30 runs. This rule proposes a new understanding of the Hierarchical Even-parity problem that was unexpected before experiments. Specifically, the lower layer of Even-parity loop with step 1 (represented by CF $c1$), is the Even-parity problem on each bit of the input. The Even-parity problem on one-bit inputs can be interpreted as the negation of the only bit in the input. Therefore, the second rule proposes that the Hierarchical Even-parity problem is equivalent to a 'flat' Even-parity problem on the bitwise negation of the input bitstring. This rule is validated theoretically below.

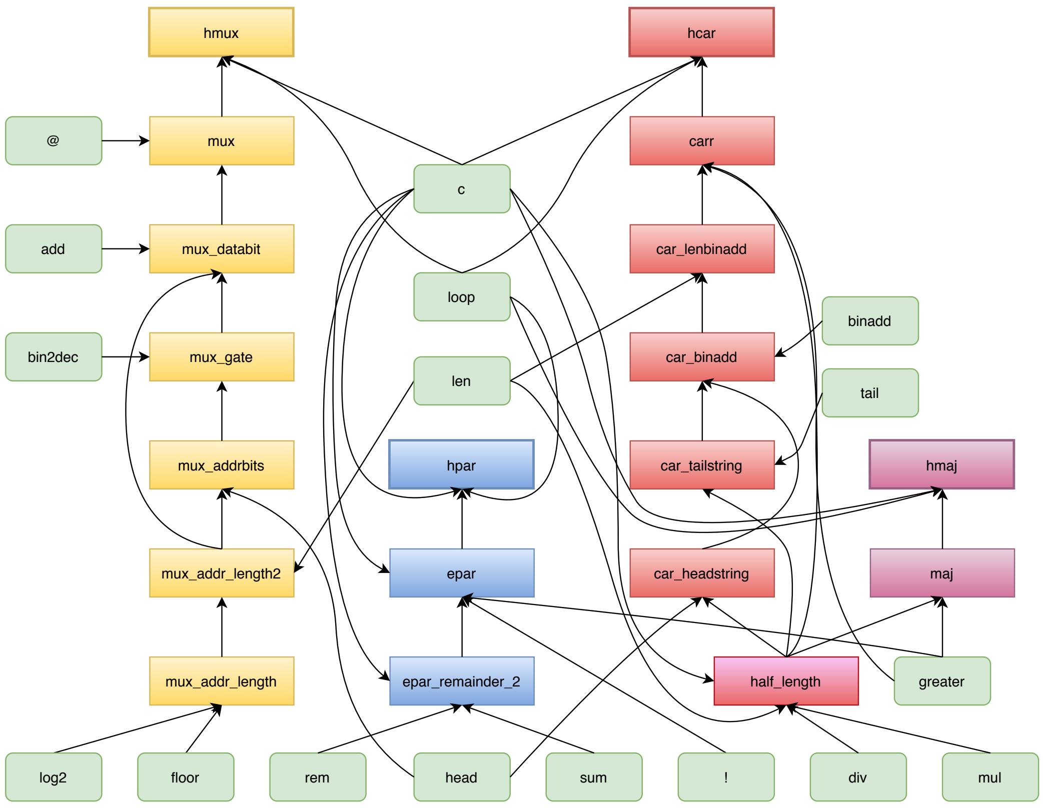


Figure 6.3: The discovered network of knowledge by ConCS. See Table 6.2 and Table 6.3 for the acronyms of the pre-provided functions in green boxes and the learnt functions (all boxes except green ones) respectively. These provide curricular to learn the target (top non-green boxes) functions given initial axiomatic skills (green boxes).

Verifying the new Finding of the Hierarchical Even-parity problem

According to the learnt functions in the experiments, the Hierarchical Even-parity problem with the lower level of 3-bit Even-parity can also be interpreted by an unexpected solution, which is an Even-parity problem of the bitwise negation of the input bitstring. To verify this finding, let's say there is a pair of instance and expected output (X, y) for a Hierarchical Even-parity problem of size $3k$, where k is an integer referring to the number of lower-level 3-bit Even-parity clusters. Input X is a bitstring $X = \{x_0, x_1, x_2, ...\}$ with length $3k$, so the input can be arranged in k non-overlapped clusters: $X = \{Cl_0, Cl_3, , Cl_6, ...\}$, where a cluster contains 3 consecutive bits $Cl_k = \{x_k, x_{k+1}, x_{k+2}\}$. If the expected output is $True$, $y = 1$, X satisfies the Hierarchical Even-parity problem:

$$\sum_{Cl_i 2X} even\_parity(Cl_i)\%2 = 0, \tag{6.3}$$

$$\iff \sum_{Cl_i 2X} even\_parity(Cl_i) = 2 \times l \tag{6.4}$$

$$\iff \begin{cases} \sum_{Cl_i 2X} ((\sum_{x2Cl_i} x)\%2 == 0) = 2 \times l, \\ \sum_{Cl_j 2X} ((\sum_{x2Cl_j} x)\%2 == 1) = k - 2 \times l \end{cases} \tag{6.5}$$

$$\iff \begin{cases} \sum_{Cl_i 2X} ((\sum_{x2Cl_i} \neg x)\%2 == 1) = 2 \times l, \\ \sum_{Cl_j 2X} ((\sum_{x2Cl_j} \neg x)\%2 == 0) = k - 2 \times l \end{cases} \tag{6.6}$$

$$\implies \begin{cases} (\sum_{x22l \text{ clusters}} \neg x)\%2 = 0, \\ (\sum_{x2(k \quad 2l) \text{ clusters}} \neg x)\%2 = 0 \end{cases} \tag{6.7}$$

$$\implies \sum_{x2X} \neg x = 2n, \tag{6.8}$$

where operator "$==$" returns 1 if two elements on both sides are equal and 0 otherwise; l is a non-negative integer. Equation 6.8 infers that the input X has an even number of bit 0. This is also the expected output of the Even-parity problem of the bitwise negation of X.

On the other hand, for any instance created by the unexpected solution, $y = 1$ leads to $\sum_{x2X} \neg x = 2n$ (n is an integer). To see whether the Hierarchical Even-parity rule also yields output 1 or not, let's apply this rule on the input of the instance. The even number of bit 0 can only be arranged in 3-bit clusters as follows: an arbitrary number l_e of clusters having even numbers of bit 0, and an even number l_o of clusters having odd numbers of bit 0. That l_e clusters have even numbers of $0s$ means that each cluster has an odd number of $1s$. The latent features applied to these clusters will yield all 0s. Similarly, the $l_o = 2l$ clusters produce latent features of 1s. Therefore, applying Even-parity on latent features, or, in other words, applying Hierarchical Even-parity rule on the input of the instance of the unexpected rule, the output is also $y = 1$. In the case of $y = 0$, the proofs are analogous. In conclusion, the newly found unexpected rule of the Hierarchical Even-parity problem in the experiment of the study is validated.

6.3.2 Learning Performances

Figure 6.4 illustrates the learning performances of ConCS in the experiment running all 19 problems concurrently. The *average* line depicts the average accuracy of all agents regarding the total learning experience of all agents. Representing the whole system by the line of average accuracy of all problems is also used to compare the performance with XCSCF* [7]. The accuracies on completed problems in XCSCF* are kept at 1.0, while ones on untouched problems are equal to the results of random guessing.

The learning performances of agents are separated and concatenated in the order of increasing indices that follows Table 6.2. The learning process of an agent only requires knowledge from agents with smaller indices. The next graph starts at the average number of instances that all agents on the left side (smaller indices) need to complete solving their problems. This format of illustration is also used in the next subsections for comparisons between ConCS and other approaches.

The performance statistics of the whole ConCS as well as its agents are summarised in Table 6.4. ConCS needs an average of 555,859 instances in all agents to finish learning all problems. The longest run took up to 2,281,000 instances to complete solving all problems, while the fastest run took only 148,250 instances. For individual agents, the slowest one, the

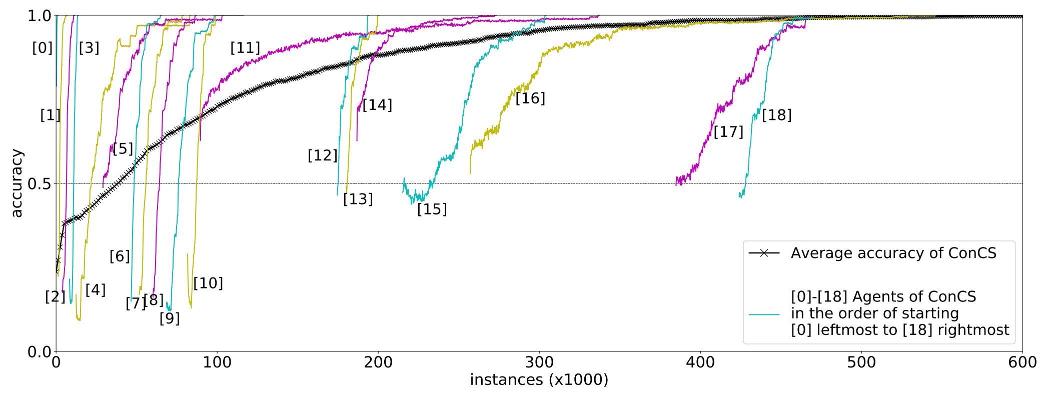


Figure 6.4: Learning performances of ConCS. The average curve depicts the average accuracy of 19 agents regarding the total experience of ConCS. Agent performances are the average learning curves of agents (across 30 runs). The agents' learning curves are ordered with increasing indices (see Table 6.2) from 0 to 18 - partly labelled for clarity. The agent performances are plotted with their separated experience (iterations). Starting points of these curves concatenate with the average completing iterations of all their previous agents (on the left).

general Hierarchical Carry-one problem, needs an average of $167{,}142$ instances to find its optimal solution. According to Figure 6.4, agents with high IDs start learning in their early phases even though they require prerequisite knowledge from other agents to combine in their optimal solutions. This is plausible because agents having better learning progress are prioritised to run.

6.3.2.1 Comparison with XCSCF*, XCS, and XCSCFC

In this section, ConCS will be compared with XCSCF* using type-fitting XCSCFA [7], XCS, and XCSCFC. Because these three approaches were designed to learn a target problem, comparisons with them must be tested on specific problems instead of 19 problems with unrelatedness. The tests on XCSCF* and XCSCFC also follow their designed learning paradigms.

Table 6.4: Performance statistics of ConCS on given problems. All numbers are the numbers of explored instances to reach optimal solutions with 100% accuracy.

Problems	Average	Longest run	Fastest run
All 19 problems	555,859	2,281,000	148,250
Multiplexer	18,884	83,250	6500
Carry-one	147,800	913,000	6750
Hierarchical Multiplexer	21,484	83,250	7250
Hierarchical Carry-one	167,142	1,020,250	11,500
Hierarchical Majority-on	49,275	143,500	5000
Hierarchical Even-parity	28,750	6617	2250

For example, to test on Multiplexer problems, while XCS learns a Multiplexer problem at a specific scale, XCSCFC reuses CFs from Multiplexer problems at lower scales. XCSCF* and ConCS both need a set of components from subproblems before achieving the solutions for Multiplexer problems.

The first five problems from Table 6.2 were selected as these are related to the Multiplexer domain. These problems are used to test ConCS and XCSCF*. Figure 6.5 shows four learning curves of the four approaches and the learning curves of ConCS agents in the same manner that Figure 6.4 separates the component learning performances.

XCS and XCSCFC are tested on the Multiplexer at a specific scale of 135 bits. In this experiment, XCSCFC learns 135-bit Multiplexer problem with the identical configuration of Transfer Learning from 6-, 11-, 20-, 37-, and 70-bit Multiplexer problems in [63]. ConCS and XCSCF* are only compared with XCSCFC in this experiment because it can solve this problem at large scales (70 and 135 bits) while it cannot completely solve problems at relatively large scales of later experiments. The experiment on the Hierarchical Even-parity problem did not include XCS because XCS cannot

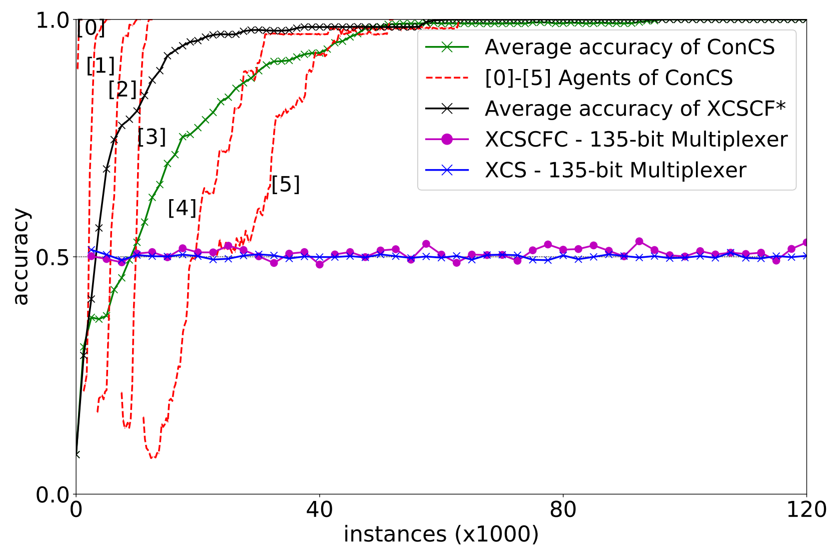


Figure 6.5: Learning performances on the variable-scale Multiplexer problem. Red curves depict the learning performances of agents in ConCS. These curves are arranged in the same manner with the agent performances in Figure 6.4.

scale well to this problem[3].

According to Figure 6.5, XCSCF* solves the variable-size Multiplexer problem most efficiently. ConCS can generate the same solution as XCSCF* within $150{,}000$ instances. XCSCF* and ConCS both outperform XCS and XCSCFC. The pattern on performance differences among the tested systems are analogous to those in other experiments for the sets of general Hierarchical Multiplexer (Figure 6.6), Carry-one (Figure 6.7), Hierarchical Carry-one (Figure 6.8), Hierarchical Even-parity (Figure 6.10), and Hierarchical Majority-on problems (Figure 6.9).

[3]Hierarchical Even-parity problem requires a one-to-one mapping of instances to rules in the solution of standard XCS.

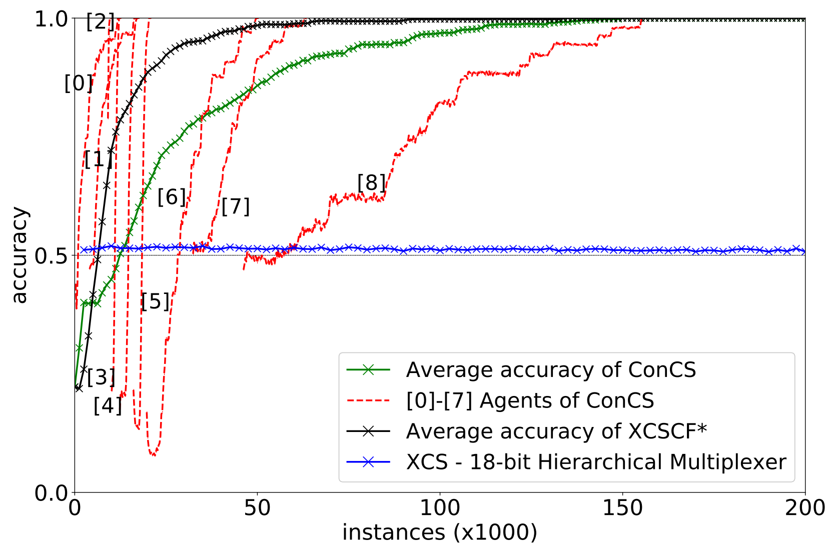


Figure 6.6: Learning performances on the variable-scale Hierarchical Multiplexer problem.

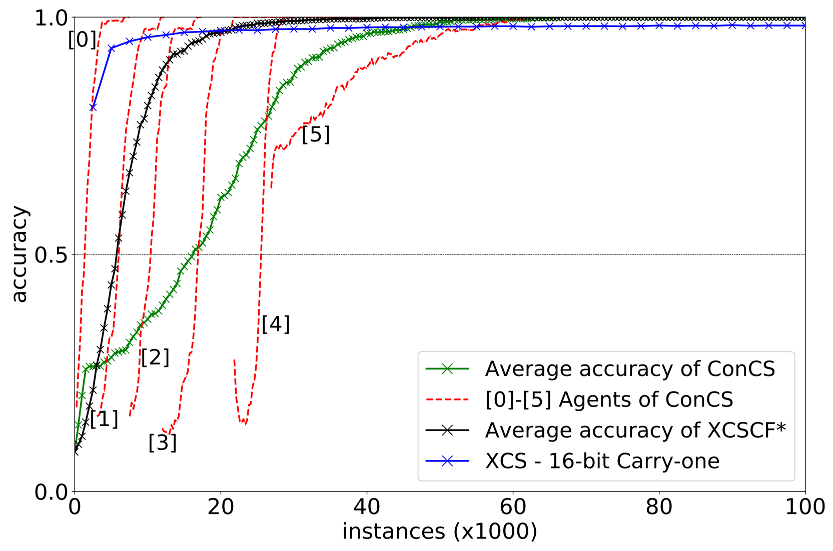


Figure 6.7: Learning performances on the variable-scale Carry-one problem.

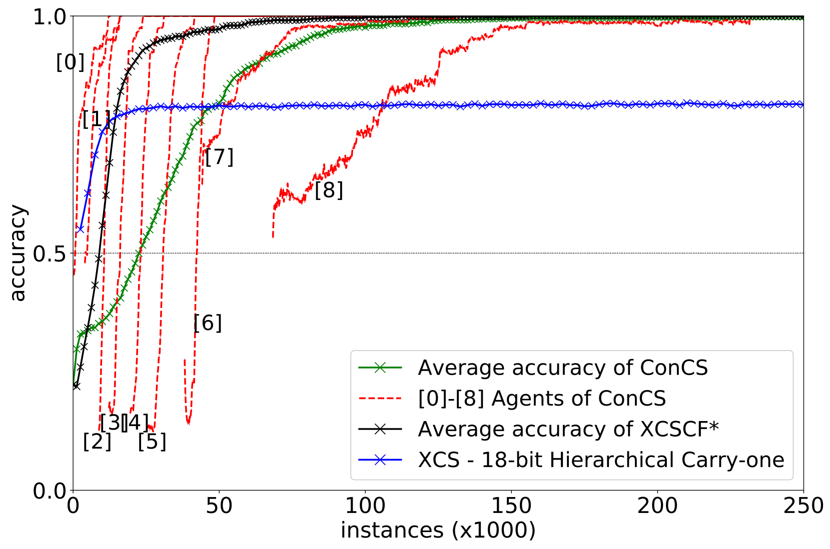


Figure 6.8: Learning performances on the variable-scale Hierarchical Carry-one problem.

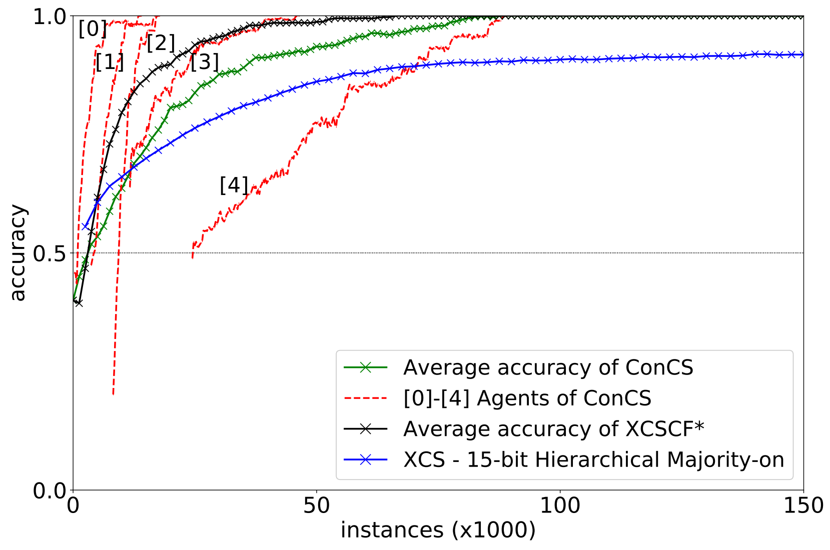


Figure 6.9: Learning performances on the variable-scale Hierarchical Majority-on problem.

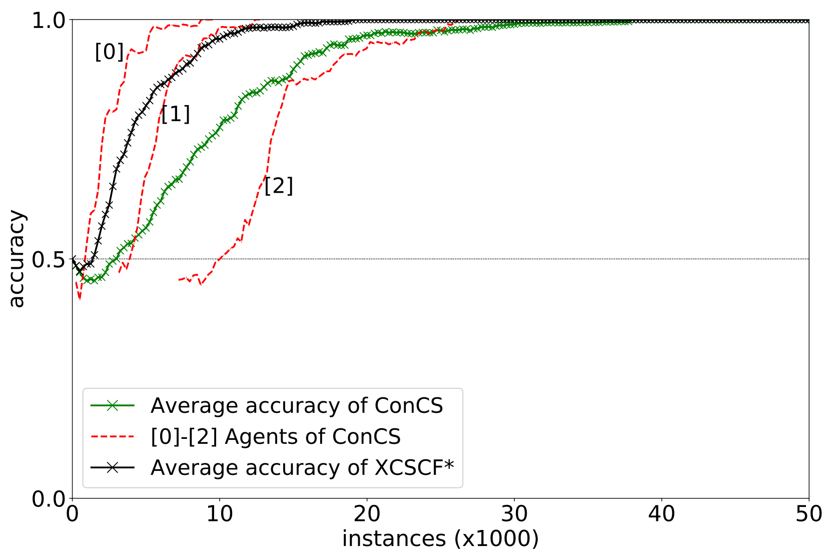


Figure 6.10: Learning performances on the variable-scale Hierarchical Even-parity problem.

6.3.2.2 Comparison with Other Machine Learning Algorithms

In this section, ConCS and XCSCF* are compared with other graph-based machine algorithms, i.e. XGBoost and Random Forest, and standard Genetic Programming (GP) on classifying large-scale and complex problems like Carry-one, Hierarchical Multiplexer, Hierarchical Carry-one, Hierarchical Majority-on, and Hierarchical Even-parity problems. It is noted that, in these experiments, other methods were tasked with solving the problems directly without the provision of sub-problems. The aim of the comparisons is to highlight ConCS performance when it is provided with sub-problems.

Standard GP, XGBoost, and Random Forest are normally used in supervised learning with separate training and testing sets. However, because ConCS can access any instance of the tested problems as its agents are online learning systems, other methods are also experimented with access to all possible instances. Because both ConCS and XCSCF* can solve the

Table 6.5: Accuracy comparisons with other machine learning approaches: standard (naïve) Genetic Programming (GP), XGBoost (XGB), and Random Forest (RF). Results of ConCS and XCSCF* are bold when they are significantly higher than the results of all other tested methods.

Problems	ConCS	XCSCF*	GP	XGB	RF
16-bit Carry-one	(1.0)	**1.0**	0.959	0.998	0.994
18-bit H. Multiplexer	**1.0**	**1.0**	0.805	0.855	0.868
18-bit H. Carry-one	1.0	1.0	0.78	1.0	1.0
15-bit H. Majority-on	1.0	1.0	0.728	0.999	0.992
21-bit H. Majority-on	**1.0**	**1.0**	0.687	0.985	0.923
18-bit H. Even-parity	**1.0**	**1.0**	0.531	0.854	0.859

tested problems within less than $200{,}000$ instances (one instance per iteration), other methods were presented with the same experience of $200{,}000$ instances[4]. Grid search was used to tune hyper-parameters of XGBoost and Random Forest (not standard GP). The results are from the best parameters for each problem.

Results on Table 6.5 show that ConCS and XCSCF* [7] both achieve 100% accuracy in all problems. Because of the ability to solve tested problems at any scale, the accuracies of ConCS and XCSCF* are constantly 100%, which are significantly higher than the average accuracies of other methods in most problems (statistical significance based on Wilcoxon signed rank test with $p{-}value < 0.05$). The differences will be rapidly increased if the scales of the benchmark problems are enlarged.

[4]For any 18-bit or higher scale Boolean problem, this is smaller than the total possible number of instances so overfitting is tested.

6.3.3 Continuous Learning with Randomly Arriving Problems

This experiment shows the capability of ConCS to learn continually when the tasks arrive at different points of time. ConCS consistently solved all 19 problems in all 30 runs. ConCS was able to solve hard problems once the easier problems providing the necessary building blocks were presented and solved. For clarity, Figure 6.11 depicts only the learning curves of all agents related to the Hierarchical Multiplexer problems. Problem 0, which was to find the address length given the Multiplexer problem size (Section 6.2.3), was presented later than most of the other problems. ConCS was gradually able to solve these other problems once the problem 0 was solved as it provides necessary building blocks to enable learning hard problems. This figure shows clearly that ConCS can learn continually by accumulating progressively more complex knowledge.

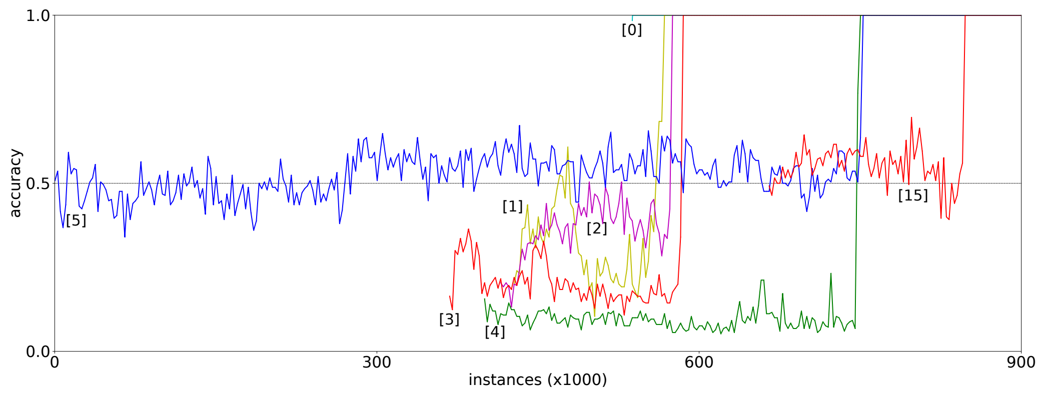


Figure 6.11: The learning performance of agents related to the Hierarchical Multiplexer domain when the problems are presented in a random order. Agents of ConCS were labelled using the "Id" column in Table 6.2. The bigger the number, the more complex the problem is.

6.4 Discussions

ConCS can simultaneously solve a large number (19) of problem types with a mix of regression and classification problems. ConCS can automatically determine the learning curricula, which was provided externally in XCSCF*. Also, the system can eliminate the need for customised configurations. Ultimately, ConCS yields highly interpretable knowledge behind its solved problems by encoding knowledge using tree-based programs.

ConCS has slower performances in benchmark problems compared with XCSCF*. This result is predictable because each learning process in LL only starts learning when all prerequisite building blocks are available. On the contrary, ConCS has to determine itself which agents have all necessary building blocks ready. In separate experiments, XCS and XCSCFC (in Multiplexer domain only) can only solve tested problems at limited scales. In contrast, both XCSCF* and ConCS produce general solutions, which can solve tested problems at infinite scales (any scale within the limit of computation hardware) with 100% accuracy. Therefore, if the experimented problem is at a large enough scale, XCSCF* and ConCS always outperform XCS and XCSCFC because prediction performances of XCSCF* and ConCS are not affected by the problem scale.

Arguably, XCSCF* and ConCS can solve tested problems more effectively than GP does because these two systems are provided with the design of problem components. Nevertheless, it is still a difficult task to combine appropriate building blocks and thereby discover the knowledge relations among problems. The provision of the problem design is analogous to the way young children build up their intelligence with sets of lessons that have been optimised over human civilisation. Optimised lessons can significantly bootstrap the learning progress of young humans. Too many lessons can inhibit the progress, while too few lessons require learners to re-discover a lot of knowledge for him/herself. For ConCS, accumulating too much knowledge may end up with the same effect, although this implementation of XCSCFA is equipped with the type-fitting property to reduce search spaces.

6.5 Chapter Summary

In this chapter, ConCS was developed as the first system that can solve multiple Boolean problems continually without human developed curricula. The minimum human involvement is to provide axiomatic knowledge and useful subproblems, where more than necessary can be provided - the system learns but does not reuse unrelated problems. ConCS is a continuous AI system of learning classifier systems with type-fitting tree-based programs to encode high-level knowledge and a pool storing accumulated knowledge. Type-fitting trees enable ConCS to capture complex knowledge behind target problems. Moreover, through learning continually with parallel tasks and subtasks, ConCS can construct novel knowledge by combining flexibly pre-provided functions and constituent patterns in subtasks.

Thanks to the apparent representation of the learnt trees in ruleset solutions, it is simple to formulate a network of knowledge among problems. The network connections enable effective references to only relevant knowledge. This promising property is important in an AI system with a huge volume of accumulated knowledge, where checking all knowledge is impractical. Moreover, the resulted knowledge network can yield learning curricula, which was previously guided by humans in layered learning [6, 7].

In certain cases, learnt knowledge provided by ConCS delivers an unexpected understanding of target problems which can be surprisingly simple. Even with an increasing volume of knowledge that leads to increasing search spaces, the problem-solving capability of the system keeps being built up by acquiring progressively more complex functions.

Future work will continue the long-term development of a continual learning system through reusing building blocks of knowledge by introducing real-valued domains. How the incorporation of learnt knowledge into the knowledge pool complements to Boolean equivalents and how type-fitting methods incorporate the new types are open questions.

The current implementation of ConCS relies on a single physical computation unit where prioritisation of agents is necessary. Utilising distributed hardware could accelerate the learning process by distributing the computation into multiple physical computing units. Paralleling the computation also enables accumulating a larger volume of knowledge.

In this work, the knowledge pool is limited to store only functions/skills, and thus the interactions among agents via the knowledge pool are limited to transferring skills. Nevertheless, extending the knowledge pool to storing knowledge, such as useful high-level features or building blocks of solving tasks is possible and promising. Using multiple agents for one task can extend interactions among agents to cooperation, coordination, and negotiation [65], which are typical in a multi-agent system [146].

Chapter 7

Conclusions and Future Work

The overall goal of this thesis was to improve the learning capability and, thus, to enable more autonomy to existing evolutionary machine learning systems through the novel developed systems based on multitask learning (and continual learning). This goal was successfully achieved by developing novel learning classifier systems that can solve hierarchical and large-scale problems with reduced human guidance. The learning capabilities of the developed systems were improved by introducing the evolution of CFs through CF-fitness that identifies the best CFs for symbolic rules, an online-adapted relatedness parameter to automate transfer features in multitask learning, and function reuse in continual learning. The results were compared with the existing code fragment-based learning classifier systems to show that a set of linked learning classifier systems can learn efficiently in various challenging learning paradigms.

The achieved ability to grow complex relevant patterns without setting a pre-defined architecture is a key factor in achieving one aspect of human intelligence, i.e. compositionality. Relevant composed building blocks in one task can be reused in other tasks, which benefits multitask learning. Multitask learning with the relatedness measure can be executed with less knowledge of the target tasks, e.g. the potential of task solutions to support one another while being learnt together. The learning curriculum for the multiple tasks is now a subsequent result of the novel continual classifier system (ConCS) instead of being supplied in layered learning. This reduces the need for human knowledge, removes the potential of human bias, and provides insights into the interrelatedness of tasks.

The remainder of this chapter presents the achieved objectives, main conclusions from each contribution chapter, and the promising future directions that can follow up this research work.

7.1 Achieved Objectives

The following research objectives have been fulfilled in support of this thesis:

1. An XCS using CF-condition, i.e. XOF, to solve large-scale and hierarchical problems, as well as real-world datasets without the requirement of a layered learning approach. The learning capability of XOF was achieved by introducing simplified evolution of CFs to grow relevant complex patterns. The evolution of CFs interacts with the environment indirectly via the rule evolution and the new parameter called CF-fitness. These two evolution processes bootstrap the learning process of XCS and enable an efficient search of high-level useful CF-based features for capturing the complex patterns within the data. Different measures of CF-fitness to bridge the two evolutions have been developed and analysed. The conservation of the niching property for the evolution of CFs produces more complexity-efficient tree-based features.

2. The first multitask learning system of XCS-based systems, named mXOF, has the ability to handle a set of arbitrary tasks together and to adjust the transfer of CFs dynamically among tasks. mXOF in-

troduced a method of automatically transferring CF-based features among tasks through a novel parameter called *relatedness*, which measures the similarities of data patterns among tasks. The relatedness enables efficient connections among learnt knowledge as these connections guide feature transfer. This parameter is essential for such multitask learning systems with tree-growing features. The automatic transfer of features improved the learning performances of multiple tasks when they are supportive of each other as well as filtering out negative transfer when the tasks are unrelated.

3. A system of multiple type-fitting XCSCFAs, called ConCS, is the first XCS-based system that can learn continually and simultaneously (multitask learning). The distributed system of XCSCFAs can accumulate progressively more complex knowledge. ConCS removes the need for providing a learning order in layered learning as the problems can be presented at the same time or in a random sequence. On the contrary, the results of the system yielded a network of knowledge, which provides efficient connections among accumulated concepts and also the learning curriculums. ConCS was shown to be able to solve n-bit Multiplexer, n-bit Carry-one, n-bit Hierarchical Multiplexer, n-bit Hierarchical Carry-one, n-bit Hierarchical Even-parity, and n-bit Hierarchical Majority-on problems by capturing their complex logic in rule actions continually.

The above major achievements improve the learning capabilities of existing CF-based XCSs. As a result, the developed systems are more autonomous as they can solve large-scale and complex problems with less human interventions. The developed systems can directly discover high-level tree features to capture the complex patterns of the tested problems, which previously required layered learning approaches. Controlling bloat and improving the structural efficiency of generated CFs were automatically achieved without a limit of tree depth. The tree features stop growing once constructed CFs cannot encapsulate the data patterns more efficiently. In addition, mXOF and ConCS' results offer efficient connections among learnt knowledge. These connections provide high-level meanings to tree nodes to address complex tasks from basic building blocks, which are equivalent to the role of connections in human brains.

7.2 Main Conclusions

This section presents the main conclusions from the three major experimental chapters (Chapter 4 to Chapter 6).

7.2.1 Online Feature-Generation of Code Fragments

An XCS with Code Fragment (CF) conditions, named XOF, can solve large-scale and hierarchical problems without the need for layered learning or transfer learning. XOF can discover useful high-level CFs to encapsulate the complex data patterns, although the search space of useful high-level CFs is very large. Specifically, the search space of depth-two CFs in a 18-bit problem has more than $20{,}000{,}000$ possible CFs[1]. As XOF enables CFs at unlimited depths, the number of possible combinations could be infinite if the computation power allows. Meanwhile, there are only 48 possible optimal CFs for capturing a chunk of the low-level 3-bit Even-parity problem in Hierarchical problems. Although the search space grows exponentially with the depth, the construction of CFs accelerates the learning process of XOF by the CF-fitness-driven search of complex patterns. The evolution of CFs can be considered as the feature construction and extraction, while the rule evolution learns the decision-making part.

- XOF introduces a simplified evolution of CFs that learns tree features

[1]A depth-one CF can have $L_1 = (18 * 17/2) * 3 * 2 * 2$ possible combinations. A depth-two CF can have $L_2 = (L_1 * (L_1 - 1)/2) * 3 * 2 * 2$ combinations, let alone the CFs connecting base CFs to root nodes.

and interacts with the rule evolution. The rule evolution supports the evolution of CFs through the estimation of CF-fitness based on classifier fitness. On the contrary, the evolution of CFs improves formalising rule condition in covering and mutation through CF-fitness that enhances the selection and construction of CFs. The evolution of CFs was set at a slower pace than the rule evolution to stabilise the growth of CFs and avoid developing bloat.

- CF-fitness is a new fitness parameter to support the evolution of CFs. It enables the CF evolution to interact with the environment through XCS rules. Different measures of CF-fitness have been developed and analysed. One of the measures is the generalising CF-fitness that focuses on more generalised patterns and avoids naïvely constructing more complex CFs from existing CFs. As a result, XOF does not need a tree depth limit as the system will not pick up more complex trees without adding discriminative information. The new CF-fitness slows down the growth of CF depth but adds more reliability to the CF construction. Although the structural efficiency of generated CFs has not been improved, it enables integrating a newly developed niching method for CFs, which results in accelerating the evolution of CFs without being trapped in local optima.

- The collected CFs with the highest CF-fitness in the Observed List (OL) encapsulates the data patterns the best among in-use CFs because these CFs are validated through constructing the highest-fitness classifiers, which are accurate and the most generalised ones. Thus, the CFs of the OL are likely to contain the richest data about the task. The OL can be considered as the harvested information the system has about the task. This information is important as pre-provided data, e.g. the data distribution, about the task is usually not available in online learning.

- The evolution of CFs relies on CF-fitness and the OL. The evolution of CFs grows existing tree features by stemming locally from the CFs in the OL. Since the OL contains the fittest and richest patterns based on their CF-fitness that the system has about the task, the construction of CFs is to search for more complex patterns that can generalise from existing specified accurate patterns (in the OL). Only the patterns that can generalise more efficiently than existing patterns are selected to be put in the OL and grow more complex patterns.

- The niching method for evolving CFs complements the existing niching property of XCS to yield a complete divide-and-conquer basis for XOF. The two evolution processes of rules and CFs are separated by niches, where the interactions inside niches have no restrictions. The niching method in evolving CFs enables improved generalisation with less bloat compared with panmictic approaches. Specifically, it reduces the frequency of irrelevant combinations of CFs. This creates a barrier that restricts CFs from crossing niche boundaries. In the initial implementation of XOF, CFs can be reused among niches with no restriction.

- XOF simplified the rule conditions of XCS by eliminating the use of 'don't care' CFs and the fixed length of rule conditions in XCSCFC [63]. Rule conditions are completely decoupled with the original input features as evolved CFs in rule conditions are usually not related to the original features. Keeping fixed-length rule conditions and maintaining GA-like evolutionary operators are not beneficial as the advantage of these operators with corresponding positions in genotypes is generally not applicable. Additionally, XOF adjusted the crossover and mutation operations of XCS to enable flexible lengths for rule conditions. Therefore, there is no need to keep rule conditions at fixed lengths. Having a flexible length enables rule conditions to shrink to one CF when the CF can generalise by encapsulating the data patterns.

- Although using the flexible representation with high-level patterns that requires more computation cost to evaluate at each environment state, XOF does not necessarily cost more computation and training time. There are a few implementation factors that yield such an efficient computation cost. First, the search for high-level CFs only samples a small portion of the search space among all possible combinations. Second, as the same genotypes of CFs are not scattered, which is required to learn CF-fitness, the evaluation results of subtrees can be reused from branches without re-evaluating.

- Although XOF does not need layered learning, it can facilitate feature transfer in layered learning and transfer learning. Solving the Multiplexer problem domain can benefit from layered learning as the selection of CFs was based on classifier fitness and the appearance frequency of CFs. These selection criteria encouraged the CFs using the address bits to be reused more than the CFs containing the data bits. These criteria fit well with the distribution of the bits in the optimal solutions of Multiplexer problems [86]. Both XOF and XCSCFC can use these selection criteria for transferring features in layered learning. However, layered learning requires crafting transferring criteria as well as an ordered sequence of learning stages.

7.2.2 Automatic Transfer in Multitask Learning with Relatedness

A system of multiple XOFs (mXOF) is introduced as the first multitask-learning XCS-based system. mXOF inherits XOF's ability to solve complex and large-scale problems by constructing useful high-level building blocks (latent features). With constructed latent features, mXOF can automate feature sharing among tasks to improve the feature extraction and construction of each task in multitask learning. The automation of feature sharing is driven by a new relationship measured in relatedness. The relatedness refers to the relationships among the data patterns of multiple tasks. mXOF uses the relatedness to adjust the feature sharing automatically during the learning process. This parameter enables mXOF to handle a set of arbitrary tasks. Specifically, it improves the learning performance of related tasks that share common constructed features. On the contrary, mXOF reduces the negative transfer of low related tasks, which maintain the learning performance of each task when the tasks in multitask learning do not share many high-level features. The relatedness between any two tasks is updated dynamically during the learning process based on the observed lists, which contain the richest information about the tasks.

- The dynamic update of relatedness is essential in multitask learning with tree-feature construction and has been captured in mXOF. mXOF can adapt relatedness among tasks regarding the common patterns as features are constructed and tasks may only be highly related at certain phases of tree construction. A fixed set of transferring criteria is not required.

- mXOF can improve the learning performance of each task in multitask learning as it promotes transferring CFs among related tasks. In addition to learning performance, mXOF improves the generality rate of discovered CFs when multiple tasks are related. The increased generality rate is an indicator of improved interpretability in learnt solutions. mXOF was also experimented to work on multiple unrelated tasks experiments without negatively affecting the learning performance of each task due to the online adaptation of the relatedness among tasks.

- The task relatedness can facilitate creating network links among target objects addressed by all XOFs in mXOF. Learning more objects/tasks builds up this knowledge network. This network enables referring to only specific knowledge that could be relevant for a target task. This is useful for an AI system with a high volume of accumulated knowledge.

- mXOF can be used for multi-class classifications. The system gives a clear indication of the relationships among classes. In multi-class classification, mXOF replaces the direct competitions between any two classes by the competitions between each class with all other classes. However, mXOF is constrained by the online learning scheme, which makes it quickly adaptable to data changes but slow for big datasets.

7.2.3 A Continual Classifier System to Solve Multiple Boolean Problems

ConCS was the first system that can solve multiple Boolean problems continuously and simultaneously without the requirement of an externally designed learning order. The minimal human involvement is to provide axiomatic knowledge and useful subproblems, where more than necessary can be provided - the system learns but does not reuse knowledge provided from unrelated problems and non-supportive axiomatic functions. ConCS was able to solve multiple Boolean problems at any scale that the computation power allows.

- ConCS is a distributed AI system of learning classifier systems with tree-based programs to encode high-level knowledge, agent prioritisation, and a pool storing accumulated knowledge. ConCS represents the high-level logic behind target problems in the form of trees with reused functionalities. Thanks to the apparent representation of the learnt trees in ruleset solutions, it is straightforward to formulate a network of knowledge among problems. As a result, the system can automatically form learning curricula, which was previously required for XCSCF* to initialise the learning process. Moreover, ConCS' network of knowledge also enables efficient connections among learnt knowledge. The connections could enable effective references to only relevant knowledge. This promising property is important in an AI system with a huge volume of accumulated knowledge, where checking all knowledge is impossible.

- In certain cases, such as the Hierarchical Even-parity domain, learnt knowledge provided by ConCS delivers an unexpected understanding of target problems, which can be surprisingly simple. This is the result of the tree flexibility in code fragments that enables the system to search for the solutions of diversified structures.

- Even with an increasing volume of knowledge that leads to increasing search spaces, the problem-solving capability of the system keeps being built up by acquiring progressively more complex functions. To be able to solve a harder related problem, the system can combine learnt knowledge and focus on the novel aspects of the new problem. However, it is highly likely that at some point, the volume of accumulated knowledge may be excessively large with diversified unrelated knowledge. This could prevent the system from becoming a general intelligence as search for relevant knowledge from a lot of fragmented learnt knowledge would be already intractable. In this case, further advanced findings from cognitive science would be essential for the development of the system.

- The current implementation of ConCS relies on a single physical computation unit where prioritisation of agents is necessary. Utilising distributed hardware could accelerate the learning process by distributing the computation into multiple physical computing units. Also, the use of distributed hardware would fit well with the architecture of ConCS as a distributed system. Lastly, paralleling the computation also enables accumulating a larger volume of knowledge.

7.3 Future Work

The achieved objectives show that LCSs are capable of dealing with large-scale and hierarchical problems without external support in traditional independent learning as well as multitask learning. The obtained solutions are readable trees that provide efficient connections among knowledge in parallel learning. This enables opportunities for further research.

- The evolution of CFs is currently based on random combinations of the CFs in the OL. The first future research is to guide the construction of new CFs through Bayesian approaches, such as an adapted Bayesian optimisation algorithm [102]. The growth of CFs could obtain complicated interactions among original features through stacking one-edge interactions among constructed features. Using statistical information could direct the search of complex feature interactions faster, especially in large datasets.

- XOF is shown to be efficient in problems with binary input features. However, the obvious gap is that XOF has not been adapted for real-valued features. The second future research is to develop an XOF-like system for real-valued features. This work could reuse existing approaches in the LCS field, such as using interval encoding [143].

- The third future research is to consider mXOF for the context of continual learning [125] and lifelong learning [85], where the AI system learns to recognise increasingly complex objects. The reason is that in mXOF, learning a new class requires only spawning a new system without remarkable negative impacts on existing tasks given a proper estimation of relatedness. Learning a new class could take advantage of the bias of previously learned knowledge to acquire relevant knowledge within fewer examples. This is equivalent to human/robot learning to recognise multiple objects using signals from the same sense/sensor. mXOF could efficiently connect a perceived signal with learnt (complex) concepts and thereby enable recognising complex objects through their learnt components.

- Because XCS and the online feature-generation module can be considered frameworks to be integrated with flexible representations (for its rules), mXOF is not bound to using only tree-based programs (CFs). Hence, the fourth future research is to consider integrating mXOF with neural networks to learn real-valued data. This combination could also be fruitful in producing arbitrary and complexity-efficient network structures. mXOF would reduce irrelevant and inefficient connections among neurons, which encapsulate learnt concepts.
- In ConCS, each problem is assigned to one learning agent. Using multiple agents for one task can extend interactions among agents to cooperation, coordination, and negotiation [65], which are typical in a multi-agent system [146]. Thus, the fifth future work is to extending the indirect interaction among the learning agents of ConCS to enable diversified communication as in multi-agent systems.
- The sixth future research could consider the selectivity of functions in ConCS for accumulation as the search space might become intractable at certain points. An approach is to consider recent findings in cognitive science to develop a forgetting mechanism. The ability to forget memory is an essential aspect of human intelligence that would be desirable for an AI system. Removing outdated knowledge helps the AI system avoid searching through the knowledge that is no longer useful.
- Although most of the experimented problems are Boolean-input problems, ConCS could be adapted to real-world problems when the XCS-based agents apply appropriate encodings for their rules, such as real-valued intervals and neural networks. Therefore, the last future work is to investigate applying ConCS on a broader range of

problems.

7.4 Closing Remarks

This research work for this thesis has demonstrated that LCSs with great flexibility can produce efficient learning systems to conquer complex problems. The ability to integrate a rich encoding, i.e. code fragments, into rule conditions and actions enables LCSs to capture complex patterns in either feature construction or decision making. The large search space caused by the richness of code fragments can be conquered using hierarchical tree growth approaches. More complex trees are constructed from reliable simpler trees/functions (tree-based ruleset functions) by either assigning simpler trees to the leaf nodes or more primitive functions to the inner nodes. This ability is argued to be a key factor in building human-like machine intelligence [77]. Thus, the developed systems in this thesis could be promising frameworks for autonomous learning systems that imitate aspects of human learning. This work opens a direction to use LCSs with rich encodings for developing continual learning systems.

Bibliography

[1] Mann Ahluwalia and Larry Bull. A genetic programming-based classifier system. In *Proceedings of the genetic and evolutionary computation conference*, volume 1, pages 11–18, 1999.

[2] Ethem Alpaydin. *Introduction to machine learning*. The MIT Press, 2020. ISBN 978-0-2620-4379-3.

[3] Isidro M. Alvarez, Will N. Browne, and Mengjie Zhang. Reusing learned functionality in XCS: Code fragments with constructed functionality and constructed features. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, pages 969–976, New York, NY, USA, 2014. Association for Computing Machinery. doi: 10.1145/2598394.2611383.

[4] Isidro M. Alvarez, Will N. Browne, and Mengjie Zhang. Reusing learned functionality to address complex boolean functions. In *Simulated Evolution and Learning*, Lecture Notes in Computer Science, pages 383–394. Springer International Publishing, December 2014. ISBN 978-3-319-13563-2. doi: 10.1007/978-3-319-13563-2_33.

[5] Isidro M. Alvarez, Will N. Browne, and Mengjie Zhang. Compaction for code fragment based learning classifier systems. In *Australasian Conference on Artificial Life and Computational Intelligence*, pages 41–53. Springer, 2016. doi: 10.1007/978-3-319-28270-1_4.

[6] Isidro M. Alvarez, Will N. Browne, and Mengjie Zhang. Human-inspired scaling in learning classifier systems: Case study on the n-bit multiplexer problem set. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 429–436, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342063. doi: 10.1145/2908812.2908813.

[7] Isidro M. Alvarez, Trung B. Nguyen, Will N. Browne, and Mengjie Zhang. A layered learning approach to scaling in learning classifier systems for boolean problems, 2020.

[8] Kavitesh Kumar Bali, Yew-Soon Ong, Abhishek Gupta, and Puay Siew Tan. Multifactorial evolutionary algorithm with online transfer parameter estimation: MFEA-II. *IEEE Transactions on Evolutionary Computation*, 24(1):69–83, 2019. ISSN 1089-778X. doi: 10.1109/TEVC.2019.2906927.

[9] Lawrence Beadle and Colin G. Johnson. Semantically driven crossover in genetic programming. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 111–116. IEEE, 2008.

[10] Richard Bellman. *An introduction to artificial intelligence: Can computers think?* Boyd & Fraser Publishing Company, 1978. ISBN 978-0-8783-5066-7.

[11] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 41–48, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585161. doi: 10.1145/1553374.1553380.

[12] Ester Bernadó-Mansilla and Josep M. Garrell-Guiu. Accuracy-based learning classifier systems: Models, analysis and applications to

classification tasks. *Evolutionary Computation*, 11(3):209–238, 2003. doi: 10.1162/106365603322365289.

[13] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information science and statistics. Springer, New York, 2006. ISBN 978-0-387-31073-2.

[14] John Blitzer, Ryan McDonald, and Fernando Pereira. Domain adaptation with structural correspondence learning. In *Proceedings of the 2006 conference on empirical methods in natural language processing*, pages 120–128, 2006.

[15] Lashon B. Booker, David E. Goldberg, and John H. Holland. Classifier systems and genetic algorithms. *Artificial Intelligence*, 40(1-3): 235–282, 1989.

[16] Markus Brameier and Wolfgang Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26, 2001. doi: 10.1109/4235.910462.

[17] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[18] Seok-Jun Bu and Sung-Bae Cho. A hybrid system of deep learning and learning classifier system for database intrusion detection. In Francisco Javier Martínez de Pisón, Rubén Urraca, Héctor Quintián, and Emilio Corchado, editors, *Hybrid Artificial Intelligent Systems*, pages 615–625, Cham, 2017. Springer International Publishing. ISBN 978-3-319-59650-1.

[19] Larry Bull. *Applications of learning classifier systems*, volume 150. Springer Science & Business Media, 2004.

[20] Larry Bull. A brief history of learning classifier systems: from CS-1 to XCS and its variants. *Evolutionary Intelligence*, 8(2-3):55–70, 2015. doi: 10.1007/s12065-015-0125-y.

[21] Larry Bull and Toby O'Hara. Accuracy-based neuro and neuro-fuzzy classifier systems. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, GECCO'02, pages 905–911, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc. ISBN 978-1-55860-878-8.

[22] Martin Butz, Martin Pelikan, Xavier Llora, and David E. Goldberg. Effective and reliable online classification combining XCS with EDA mechanisms. In *Scalable Optimization via Probabilistic Modeling*, pages 249–273. Springer, 2006.

[23] Martin V. Butz. *Rule-based evolutionary online learning systems*. Springer, 2006. ISBN 978-3-540-31231-4.

[24] Martin V. Butz. How XCS works: Ensuring effective evolutionary pressures. In *Rule-Based Evolutionary Online Learning Systems: A Principled Approach to LCS Analysis and Design*, pages 65–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. doi: 10.1007/3-540-31231-5_5.

[25] Martin V. Butz. The XCS classifier system. In *Rule-Based Evolutionary Online Learning Systems: A Principled Approach to LCS Analysis and Design*, pages 51–64. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. doi: 10.1007/3-540-31231-5_4.

[26] Martin V. Butz and Stewart W. Wilson. An algorithmic description of XCS. In Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *Advances in Learning Classifier Systems*, pages 253–272, Berlin, Heidelberg, September 2000. Springer Berlin Heidelberg. doi: 10.1007/3-540-44640-0_15.

[27] Martin V Butz, Tim Kovacs, Pier Luca Lanzi, and Stewart W Wilson. Toward a theory of generalization and learning in XCS. *IEEE transactions on evolutionary computation*, 8(1):28–46, 2004.

[28] Rich Caruana. Multitask learning. *Machine Learning*, 28(1):41–75, 1997. doi: 10.1023/A:1007379606734.

[29] Rich Caruana. Multitask learning. In *Learning to Learn*, pages 95–133. Springer, Boston, MA, 1998. doi: 10.1007/978-1-4615-5529-2_5.

[30] Eugene Charniak. *Introduction to artificial intelligence*. Addison-Wesley series in Computer Science. Pearson Education India, 1985. ISBN 978-8-1317-0306-9.

[31] Qi Chen. *Improving the generalisation of genetic programming for symbolic regression*. PhD thesis, 2018.

[32] Qi Chen, Bing Xue, Yi Mei, and Mengjie Zhang. Geometric semantic crossover with an angle-aware mating scheme in genetic programming for symbolic regression. In James McDermott, Mauro Castelli, Lukas Sekanina, Evert Haasdijk, and Pablo García-Sánchez, editors, *Genetic Programming*, pages 229–245, Cham, 2017. Springer International Publishing. ISBN 978-3-319-55696-3.

[33] Qi Chen, Mengjie Zhang, and Bing Xue. New geometric semantic operators in genetic programming: Perpendicular crossover and random segment mutation. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, pages 223–224, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349390. doi: 10.1145/3067695.3076008.

[34] Qi Chen, Bing Xue, and Mengjie Zhang. Improving generalization of genetic programming for symbolic regression with angle-driven geometric semantic operators. *IEEE Transactions on Evolutionary Computation*, 23(3):488–502, 2019. doi: 10.1109/TEVC.2018.2869621.

[35] Qi Chen, Bing Xue, and Mengjie Zhang. Genetic programming for instance transfer learning in symbolic regression. *IEEE Transactions on Cybernetics*, pages 1–14, 2020. doi: 10.1109/TCYB.2020.2969689.

[36] Qi Chen, Bing Xue, and Mengjie Zhang. Rademacher complexity for enhancing the generalization of genetic programming for symbolic regression. *IEEE Transactions on Cybernetics*, pages 1–14, 2020. doi: 10.1109/TCYB.2020.3004361.

[37] Hai H. Dam, Hussein A. Abbass, Chris Lokan, and Xin Yao. Neural-based learning classifier systems. *IEEE Transactions on Knowledge and Data Engineering*, 20(1):26–39, January 2008. ISSN 1041-4347. doi: 10.1109/TKDE.2007.190671.

[38] Jesse Davis and Pedro Domingos. Deep transfer via second-order markov logic. In *Proceedings of the 26th annual international conference on machine learning*, pages 217–224. ACM, 2009.

[39] Edwin D. De Jong and Jordan B. Pollack. Multi-objective methods for tree size control. *Genetic Programming and Evolvable Machines*, 4 (3):211–233, 2003.

[40] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[41] Cândida Ferreira. Gene expression programming in problem solving. In *Soft computing and industry*, pages 635–653. Springer, 2002.

[42] Hans Forssberg, Ann-Christin Eliasson, Hiroshi Kinoshita, Roland S. Johansson, and Göran Westling. Development of human precision grip i: basic coordination of force. *Experimental Brain Research*, 85(2):451–457, 1991. doi: 10.1007/BF00229422.

[43] Jonathan B. Freeman, Nalini Ambady, Katherine J. Midgley, and Phillip J. Holcomb. The real-time link between person perception and action: Brain potential evidence for dynamic continuity. *Social Neuroscience*, 6(2):139–155, 2011. doi: 10.1080/17470919.2010.490674. PMID: 20602284.

[44] Jing Gao, Wei Fan, Jing Jiang, and Jiawei Han. Knowledge transfer via multiple model local structure mapping. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 283–291, New York, NY, USA, 2008. Association for Computing Machinery. doi: 10.1145/1401890.1401928.

[45] Antonella Giani, Fabrizio Baiardi, and Antonina Starita. PANIC: A parallel evolutionary rule based system. In *Evolutionary Programming*, pages 753–771, 1995.

[46] David E. Goldberg. Genetic algorithms in search, optimization, and machine learning. *New York, Addison-Wesley*, 1989.

[47] A. Gupta, Y. Ong, L. Feng, and K. C. Tan. Multiobjective multifactorial optimization in evolutionary multitasking. *IEEE Transactions on Cybernetics*, 47(7):1652–1665, 2017.

[48] Abhishek Gupta, Yew-Soon Ong, and Liang Feng. Multifactorial evolution: toward evolutionary multitasking. *IEEE Transactions on Evolutionary Computation*, 20(3):343–357, 2015.

[49] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[50] Georges Harik. Linkage learning via probabilistic modeling in the ECGA. *Urbana*, 51(61):801, 1999.

[51] Demis Hassabis, Dharshan Kumaran, Christopher Summerfield, and Matthew Botvinick. Neuroscience-inspired artificial intelligence. *Neuron*, 95(2):245–258, 2017.

[52] Tomohiro Hayashida, Ichiro Nishizaki, Shinya Sekizaki, and Yuki Ogasawara. Development of a classifier system for a continuous environment. *Electronics and Communications in Japan*, 102(10):17–25, 2019. doi: 10.1002/ecj.12209.

[53] Geoffrey E. Hinton and Ruslan R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[54] John H. Holland. Adaptation in natural and artificial systems. an introductory analysis with application to biology, control, and artificial intelligence. *Ann Arbor, MI: University of Michigan Press*, pages 439–444, 1975.

[55] John H. Holland. Adaptation. In *Progress in Theoretical Biology*, pages 263–293. Academic Press, 1976. ISBN 978-0-12-543104-0. doi: 10.1016/B978-0-12-543104-0.50012-3.

[56] Gerard Howard, Larry Bull, and Pier-Luca Lanzi. A spiking neural representation for XCSF. In *IEEE Congress on Evolutionary Computation*, pages 1–8, July 2010. doi: 10.1109/CEC.2010.5586035.

[57] Charalambos Ioannides and Will Browne. Investigating scaling of an abstracted LCS utilising ternary and S-expression alphabets. In Jaume Bacardit, Ester Bernadó-Mansilla, Martin V. Butz, Tim Kovacs, Xavier Llorà, and Keiki Takadama, editors, *Learning Classifier Systems*, pages 46–56, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-88138-4.

[58] Charalambos Ioannides, Geoff Barrett, and Kerstin Eder. XCS cannot learn all boolean functions. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1283–1290, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0557-0.

[59] Muhammad Iqbal. *Improving the Scalability of XCS-Based Learning Classifier Systems*. PhD thesis, 2014.

[60] Muhammad Iqbal, Will N. Browne, and Mengjie Zhang. Extracting and using building blocks of knowledge in learning classi-

fier systems. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO '12, pages 863–870, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311779. doi: 10.1145/2330163.2330283.

[61] Muhammad Iqbal, Will N. Browne, and Mengjie Zhang. XCSR with computed continuous action. In *AI 2012: Advances in Artificial Intelligence*, pages 350–361, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi: 10.1007/978-3-642-35101-3_30.

[62] Muhammad Iqbal, Will N. Browne, and Mengjie Zhang. Evolving optimum populations with XCS classifier systems. *Soft Computing*, 17(3):503–518, 2013.

[63] Muhammad Iqbal, Will N. Browne, and Mengjie Zhang. Reusing building blocks of extracted knowledge to solve complex, large-scale boolean problems. *IEEE Transactions on Evolutionary Computation*, 18(4):465–480, 2014.

[64] Eugene M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572, November 2003. doi: 10.1109/TNN.2003.820440.

[65] Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998. doi: 10.1023/A:1010090405266.

[66] Jing Jiang and ChengXiang Zhai. Instance weighting for domain adaptation in NLP. In *Proceedings of the 45th annual meeting of the association of computational linguistics*, pages 264–271, 2007.

[67] Rie Johnson and Tong Zhang. A high-performance semi-supervised learning method for text chunking. In *Proceedings of the 43rd An-*

nual Meeting of the Association for Computational Linguistics (ACL'05), pages 1–9, 2005.

[68] Markus Kiefer, Eun-Jin Sim, Bärbel Herrnberger, Jo Grothe, and Klaus Hoenig. The sound of concepts: Four markers for a link between auditory and conceptual brain systems. *Journal of Neuroscience*, 28(47):12224–12230, 2008. doi: 10.1523/JNEUROSCI.3579-08.2008.

[69] Ji-Yoon Kim and Sung-Bae Cho. Exploiting deep convolutional neural networks for a neural-based learning classifier system. *Neurocomputing*, 354:61 – 70, 2019. doi: 10.1016/j.neucom.2018.05.137. Recent Advancements in Hybrid Artificial Intelligence Systems.

[70] David Kinzett, Mark Johnston, and Mengjie Zhang. Numerical simplification for bloat control and analysis of building blocks in genetic programming. *Evolutionary Intelligence*, 2(4):151, 2009. doi: 10.1007/s12065-009-0029-9.

[71] George Konidaris and Andrew Barto. Autonomous shaping: Knowledge transfer in reinforcement learning. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 489–496. Association for Computing Machinery, 2006. doi: 10.1145/1143844.1143906.

[72] Tim Kovacs. Strength or accuracy? a comparison of two approaches to fitness calculation in learning classifier systems. In *Proceedings of the 1999 Genetic and Evolutionary Computation Conference Workshop Program*, pages 258–265, 1999.

[73] John R. Koza. A hierarchical approach to learning the boolean multiplexer function. 1:171–192, 1991. doi: 10.1016/B978-0-08-050684-5.50014-8.

[74] John R. Koza. *Genetic Programming: On the Programming of Computers*

by Means of Natural Selection. The MIT Press, Cambridge, MA, USA, 1992. ISBN 978-0-262-11170-6.

[75] Krzysztof Krawiec and Pawel Lichocki. Approximating geometric crossover in semantic space. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 987–994, New York, NY, USA, 2009. Association for Computing Machinery. doi: 10.1145/1569901.1570036.

[76] Ray Kurzweil. *The Age of Intelligent Machines*. The MIT Press, 1990. ISBN 978-0-2621-1121-8.

[77] Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. Building machines that learn and think like people. *Behavioral and Brain Sciences*, 40:e253, 2017. doi: 10.1017/S0140525X16001837.

[78] Pier Luca Lanzi. XCS with stack-based genetic programming. In *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, volume 2, pages 1186–1191. IEEE, 2003.

[79] Pier Luca Lanzi and Perrucci Alessandro. Extending the representation of classifier conditions part ii: From messy coding to S-expressions. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 99)*, pages 345–352, 1999.

[80] Pier Luca Lanzi and Daniele Loiacono. XCSF with neural prediction. In *2006 IEEE International Conference on Evolutionary Computation*, pages 2270–2276, 2006. doi: 10.1109/CEC.2006.1688588.

[81] Pier Luca Lanzi and Daniele Loiacono. Classifier systems that compute action mappings. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1822–1829, New York, NY, USA, 2007. Association for Computing Machinery. doi: 10.1145/1276958.1277322.

[82] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

[83] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[84] Zachary C. Lipton, Yu-Xiang Wang, and Alex Smola. Detecting and correcting for label shift with black box predictors, 2018.

[85] Bing Liu. Lifelong machine learning: a paradigm for continuous learning. *Frontiers of Computer Science*, 11(3):359–361, 2017.

[86] Yi Liu, Bing Xue, and Will N. Browne. Visualisation and optimisation of learning classifier systems for multiple domain learning. In Yuhui Shi, Kay Chen Tan, Mengjie Zhang, Ke Tang, Xiaodong Li, Qingfu Zhang, Ying Tan, Martin Middendorf, and Yaochu Jin, editors, *Simulated Evolution and Learning*, pages 448–461, Cham, 2017. Springer International Publishing. ISBN 978-3-319-68759-9.

[87] Sean Luke and Liviu Panait. A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344, 2006. doi: 10.1162/evco.2006.14.3.309.

[88] David Madras, Elliot Creager, Toniann Pitassi, and Richard Zemel. Learning adversarially fair and transferable representations, 2018.

[89] Kazuma Matsumoto, Takato Tatsumi, Hiroyuki Sato, Tim Kovacs, and Keiki Takadama. XCSR learning from compressed data acquired by deep neural network. *Journal of Advanced Computational Intelligence and Intelligent Informatics*, 21(5):856–867, 2017. doi: 10.20965/jaciii.2017.p0856.

[90] Kazuma Matsumoto, Ryo Takano, Takato Tatsumi, Hiroyuki Sato, Tim Kovacs, and Keiki Takadama. XCSR based on compressed input by deep neural network for high dimensional data. In *Proceed-*

ings of the Genetic and Evolutionary Computation Conference Companion, GECCO '18, page 1418–1425, New York, NY, USA, 2018. Association for Computing Machinery. doi: 10.1145/3205651.3208281.

[91] Ryszard S. Michalski. A theory and methodology of inductive learning. In Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors, *Machine Learning*, pages 83–134. Morgan Kaufmann, San Francisco (CA), 1983. ISBN 978-0-08-051054-5.

[92] Lilyana Mihalkova, Tuyen Huynh, and Raymond J Mooney. Mapping and revising markov logic networks for transfer learning. In *AAAI*, volume 7, pages 608–614, 2007.

[93] Julian F. Miller and Peter Thomson. Cartesian genetic programming. In *European Conference on Genetic Programming*, pages 121–132. Springer, 2000.

[94] Alan T. W. Min, Ramon Sagarna, Abhishek Gupta, Yew-Soon Ong, and Chi K. Goh. Knowledge transfer through machine learning in aircraft design. *IEEE Computational Intelligence Magazine*, 12(4):48–60, 2017.

[95] Tom M. Mitchell. *Machine Learning*. McGraw Hill, 1997. ISBN 978-0070428072.

[96] David J. Montana. Strongly Typed Genetic Programming. *Evolutionary Computation*, 3:199–230, June 1995. doi: 10.1162/evco.1995.3.2.199.

[97] Alberto Moraglio, Krzysztof Krawiec, and Colin G Johnson. Geometric semantic genetic programming. In *International Conference on Parallel Problem Solving from Nature*, pages 21–31. Springer, 2012.

[98] Masaya Nakata and Will N. Browne. Learning optimality theory for accuracy-based learning classifier systems. *IEEE Transactions on Evo-*

lutionary Computation, 25(1):61–74, 2021. doi: 10.1109/TEVC.2020.2994314.

[99] Bach H. Nguyen, Bing Xue, Peter Andreae, and Mengjie Zhang. A hybrid evolutionary computation approach to inducing transfer classifiers for domain adaptation. *IEEE Transactions on Cybernetics*, pages 1–14, 2020. doi: 10.1109/TCYB.2020.2980815.

[100] Yew-Soon Ong and Abhishek Gupta. Evolutionary multitasking: a computer science view of cognitive multitasking. *Cognitive Computation*, 8(2):125–142, 2016. doi: 10.1007/s12559-016-9395-7.

[101] Sinno J. Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, October 2010. doi: 10.1109/TKDE.2009.191.

[102] Martin Pelikan and David E. Goldberg. *Bayesian optimization algorithm: From single level to hierarchy*. University of Illinois at Urbana-Champaign Urbana, IL, 2002.

[103] Martin Pelikan, David E. Goldberg, and Erick Cantú-Paz. BOA: The Bayesian optimization algorithm. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1*, GECCO'99, pages 525–532, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 978-1-55860-611-1.

[104] Martin Pelikan, Kumara Sastry, and David E. Goldberg. iBOA: The incremental Bayesian optimization algorithm. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, GECCO '08, pages 455–462, New York, NY, USA, 2008. ACM. doi: 10.1145/1389095.1389177.

[105] Martin Pelikan, Mark W. Hauschild, and Fernando G. Lobo. Introduction to estimation of distribution algorithms. (2012003), 2012.

[106] Alberto E. Pereda. Electrical synapses and their functional inter-

actions with chemical synapses. *Nature Reviews Neuroscience*, 15(4): 250–263, 2014. doi: 10.1038/nrn3708.

[107] Rolf Pfeifer and Christian Scheier. *Understanding intelligence*. The MIT Press, 2001.

[108] Jean Piaget and Margaret Cook. *The Origins of Intelligence in Children*, volume 8. W W Norton & Co, 1952. doi: 10.1037/11494-000.

[109] David Poole, Alan Mackworth, and Randy Goebel. *Computational Intelligence: A Logical Approach*. Oxford University Press, 1998. ISBN 978-0-1951-0270-3.

[110] Duncan Potts and Bernhard Hengst. Concurrent discovery of task hierarchies. In *AAAI Spring Symposium on Knowledge Representation and Ontology for Autonomous Systems*, pages 1–8, 2004.

[111] Richard J. Preen, Stewart W. Wilson, and Larry Bull. Autoencoding with a classifier system, 2020.

[112] Mark B. Ring. *Child: A First Step Towards Continual Learning*, pages 261–292. Springer US, Boston, MA, 1998. doi: 10.1007/978-1-4615-5529-2_11.

[113] Stuart Jonathan Russell and Peter Norvig. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Upper Saddle River, 2003. ISBN 978-0-1379-0395-5.

[114] Anton Schwaighofer, Volker Tresp, and Kai Yu. Learning Gaussian process kernels via hierarchical Bayes. In *Advances in neural information processing systems*, pages 1209–1216. The MIT Press, 2005.

[115] Abubakar Siddique, Will N. Browne, and Gina M. Grimshaw. Lateralized learning for robustness against adversarial attacks in a visual classification system. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, page 395–403, New

York, NY, USA, 2020. Association for Computing Machinery. doi: 10.1145/3377930.3390164.

[116] Stephen Frederick Smith. *A learning system based on genetic adaptive algorithms*. PhD thesis, 1980.

[117] Peter Stone. Learning and multiagent reasoning for autonomous agents. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, pages 12–30. Morgan Kaufmann Publishers Inc., 2007.

[118] Peter Stone and Manuela Veloso. Layered learning. In Ramon López de Mántaras and Enric Plaza, editors, *Machine Learning: ECML 2000*, volume 1810, pages 369–381, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. doi: 10.1007/3-540-45164-1_38.

[119] Peter H. Stone. Layered learning in multi-agent systems. Technical report, Carnegie Mellon University, dec 1998.

[120] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988. doi: 10.1007/BF00115009.

[121] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction*. The MIT Press, Cambridge, MA, 2018. ISBN 978-0-2620-3924-6.

[122] Esther Thelen. Rhythmical stereotypies in normal human infants. *Animal Behaviour*, 27:699 – 715, 1979. doi: 10.1016/0003-3472(79)90006-X.

[123] Sebastian Thrun and Tom M. Mitchell. Learning one more thing. Technical report, Carnegie Mellon University, September 1994.

[124] Sebastian Thrun and Tom M. Mitchell. Lifelong robot learning. *Robotics and autonomous systems*, 15(1-2):25–46, 1995.

[125] Sebastian Thrun and Lorien Pratt. *Learning to learn*. Springer Science & Business Media, 2012. ISBN 978-1-4615-5529-2.

[126] Lisa Torrey and Jude Shavlik. Transfer learning. *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, 1:242–264, 2009. doi: 10.4018/978-1-60566-766-9.ch011.

[127] Hau T. Tran, Cédric Sanza, Yves Duthen, and Thuc Dinh Nguyen. XCSF with computed continuous action. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1861–1869, New York, NY, USA, 2007. Association for Computing Machinery. doi: 10.1145/1276958.1277327.

[128] Eric Tzeng, Judy Hoffman, Kate Saenko, and Trevor Darrell. Adversarial discriminative domain adaptation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[129] Ryan J. Urbanowicz and Will N. Browne. *Introduction to Learning Classifier Systems*. SpringerBriefs in Intelligent Systems. Springer-Verlag, Berlin Heidelberg, 2017. doi: 10.1007/978-3-662-55007-6.

[130] Ryan J. Urbanowicz and Jason H. Moore. Learning classifier systems: A complete introduction, review, and roadmap. *Journal of Artificial Evolution and Applications*, 2009:1, 2009. doi: 10.1155/2009/736398.

[131] Fumito Uwano, Koji Dobashi, Keiki Takadama, and Tim Kovacs. Generalizing rules by random forest-based learning classifier systems for high-dimensional data mining. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '18, page 1465–1472, New York, NY, USA, 2018. Association for Computing Machinery. doi: 10.1145/3205651.3208298.

[132] Nguyen Q. Uy, Nguyen T. Hien, Nguyen X. Hoai, and Michael

O'Neill. Improving the generalisation ability of genetic programming with semantic similarity based crossover. In *Genetic Programming*, pages 184–195, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-12148-7.

[133] Leonardo Vanneschi, Mauro Castelli, and Sara Silva. Measuring bloat, overfitting and functional complexity in genetic programming. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pages 877–884, New York, NY, USA, 2010. Association for Computing Machinery. doi: 10.1145/1830483.1830643.

[134] Leonardo Vanneschi, Sara Silva, Mauro Castelli, and Luca Manzoni. Geometric semantic genetic programming for real life applications. In *Genetic Programming Theory and Practice XI*, pages 191–209. Springer, 2014. doi: 10.1007/978-1-4939-0375-7_11.

[135] Gilles Venturini. *Apprentissage adaptatif et apprentissage supervise par algorithme genetique*. PhD thesis, Paris 11, 1994.

[136] Ekaterina J. Vladislavleva, Guido F. Smits, and Dick Den Hertog. Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. *IEEE Transactions on Evolutionary Computation*, 13(2):333–349, 2009. doi: 10.1109/TEVC.2008.926486.

[137] Chang Wang and Sridhar Mahadevan. Manifold alignment using procrustes analysis. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, pages 1120–1127. Association for Computing Machinery, 2008. doi: 10.1145/1390156.1390297.

[138] Shimon Whiteson. Evolutionary computation for reinforcement learning. In *Reinforcement Learning: State-of-the-Art*, pages 325–355. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-27645-3_10.

[139] Shimon Whiteson and Peter Stone. Concurrent layered learning. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '03, pages 193–200, New York, NY, USA, 2003. Association for Computing Machinery. doi: 10.1145/860575.860607.

[140] Stewart W Wilson. ZCS: A zeroth level classifier system. *Evolutionary computation*, 2(1):1–18, 1994. doi: 10.1162/evco.1994.2.1.1.

[141] Stewart W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, June 1995. doi: 10.1162/evco.1995.3.2.149.

[142] Stewart W. Wilson. Generalization in the XCS classifier system. 1998.

[143] Stewart W. Wilson. Get real! XCS with continuous-valued inputs. In Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *Learning Classifier Systems*, pages 209–219, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-45027-6.

[144] Stewart W. Wilson. Classifiers that approximate functions. *Natural Computing*, 1(2-3):211–234, 2002. doi: 10.1023/A:1016535925043.

[145] Stewart W. Wilson. Classifier conditions using gene expression programming. In Jaume Bacardit, Ester Bernadó-Mansilla, Martin V. Butz, Tim Kovacs, Xavier Llorà, and Keiki Takadama, editors, *Learning Classifier Systems*, pages 206–217. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-88138-4.

[146] Michael Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2009. ISBN 978-0-470-51946-2.

[147] Bianca Zadrozny. Learning and evaluating classifiers under sample selection bias. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, page 114, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138385.

[148] Mengjie Zhang and Phillip Wong. Genetic programming for medical classification: a program simplification approach. *Genetic Programming and Evolvable Machines*, 9(3):229–255, 2008. doi: 10.1007/s10710-008-9059-9.