Learning Classifier Systems for Multi-Objective Reinforcement Learning Problems

by

Xiu Cheng

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
2022

Abstract

The real world is full of problems with multiple conflicting objectives. However, Reinforcement Learning (RL) traditionally deals with only a single learning objective. Recently, several Multi-Objective Reinforcement Learning (MORL) algorithms have been proposed. Nevertheless, many of these algorithms rely on tabular representations of the value function, which are only suitable for solving small-scale problems. In addition, although some existing MORL techniques can learn the Pareto optimal solutions, they can only be applied to simple multi-step problems in a discrete Markov environment. However, many real-world problems involve partially-observable continuous environments, where continuous environments are often discretise before the systems learn the best policies from a starting to a goal state. Therefore, effective MORL techniques are needed to address these problems.

Learning Classifier Systems (LCSs) have been widely used to tackle RL problems as they have a good generalization ability and provide a simple, understandable rule-based solution. The rule-based solution has good scalability to solve large-scale problems. In addition, the rule-based solution is explainable. Thus the user is not only able to know what the solution is but understand why the solution is suited to the task and when to trust the solution. Moreover, the accuracy-based LCS, XCS, has been adopted successfully for single-objective RL problems. Thus, LCSs, especially XCS, have huge potential to solve MORL problems.

In this thesis, LCSs are enabled to learn the Pareto optimal policies for discrete, partially-observable, discrete and continuous MORL problems. The objectives are to develop XCS-based algorithms to learn the Pareto

optimal policies for MORL problems, improve the generalization ability of the multi-objective XCS-based algorithm where possible, evaluate the generalization ability of developed multi-objective XCS-based algorithms for solving MORL problems, and apply multi-objective XCS-based algorithms to solve large-scale partially-observed MORL problems. The following tasks have been completed in this thesis.

A new multi-objective LCS-based algorithm MO-XCS has been developed based on XCS for multi-objective learning. This algorithm is designed to learn a group of Pareto optimal solutions through a single learning process. For this purpose, four technical issues in XCS have been identified and addressed. Experimental studies on three bi-objective maze problems further demonstrate the effectiveness of MO-XCS.

A new multi-objective LCS-based algorithm MOXCS, different from MO-XCS, has been developed based on XCS and MOEA/D. It employs a decomposition strategy based on MOEA/D in XCS to approximate complex Pareto Fronts. The experimental results show that on complex bi-objective maze problems, MOXCS can learn a group of Pareto optimal solutions for MORL problems without requiring huge storage. Analysis of the learned policies shows successful trade-offs between the distance to the reward and the amount of reward itself. With integer inputs, MOXCS can address the partially observable Markov problem Deep Sea Treasure. Lastly, the generalization ability of MOXCS is tested in the Multi-Maze and Multi-Maze Connection environments. However, it is shown that the Multi-Maze Connection domain is not a proper benchmark for testing the generalization ability of the MORL algorithm.

CoinRun is a continuous domain, unlike the previous Maze and Deep Sea Treasure, which are discrete. Two bi-objective CoinRun environments are developed in this thesis for testing the generalization ability of the MORL algorithm. Several changes in the developed bi-objective CoinRun environments have been made. First, they have been changed from continuous environments to the discretized environments by the technique

of discretizing continuous inputs. Second, those environments have been changed from the non-Markov environments to Markov environments by adding extra characters. Third, in order to enable the agent to sense the environment a sub-actions technique has been developed; when the agent takes action several following actions will be taken automatically. With these changes and techniques, the developed bi-objective CoinRun problems are solved by MOXCS. The generalization ability of MOXCS will be evaluated by exchange those two bi-objective CoinRun environments as the training and testing environment. The evidence shows MOXCS has the generalization ability for solving the MORL problem in an unseen environment with proper training in a similar environment.

The single-objective Mountain Car is extended as a bi-objective MORL problem by adding another objective for the mountain car to take more action, 'zero-throttles'. MOXCS is implemented to resolve the multi-objective mountain car problem. The result shows that MOXCS has good potential to resolve this large-scale Markov problem.

This research work has shown that the LCS-based algorithms can learn Pareto optimal policies for the discrete, partially-observed, and adapted continuous MORL problems and have the potential to scale to complex, large-scale MORL problems.

Acknowledgments

Thanks for the great help from my supervisor Will Browne for supervising me and supporting me. During the past several years, Will gives me guidance on academic research to let me have a deep understanding of my research area, Multi-Objective Reinforcement Learning, step by step. I have done lots of successful research with him, and gain lots of valuable experience. At the same time, from him, I have learned what research is and what is the value of research. Valuable research is not always proof the algorithm is better than others, but it also could be research that tells why it does not work well so far; thus, it can benefit future research in this area. This attitude makes me do not afraid of failures when I am exploring new areas. Thanks for his patience; when I am less productive or have difficulties focusing on my research, he always supports me and gives me huge encouragement. Without it, I could not persists in my study and survive a difficult situation. Thanks to his kindness, when I make mistakes or feel frustrated, he always gives me help and support to fix the errors positively without any pressure. There are lots of ups and downs in my Ph.D. joinery. Without his trust and support, I could not get here. So I have learned what trust is and how important it is.

Thanks for the great help from my supervisor Mengjie Zhang. Meng is like a parent and a role model for me. As a parent, he always takes care of me and supports me when I am facing difficulties, and he will give me a kind reminder when I need guidance. In addition, he is always reliable and makes me comfortable to sort to help. As a role model, Meng

is very self-organized and working hard. From him, I've learned success is instead a milestone, but it needs persistence and long-term efforts.

Thanks to my husband Martin Yijing Xie and my daughter Grace Ziqi Xie. Martin is the crucial reason and motivation I came to beautiful New Zealand. We loved, love, and will love each other like what describes in the marriage vows. For better for worse, for richer for poorer, in sickness and in health, to love, honor, and cherish each other. My daughter Grace is the best gift I have ever got. I am trying to find the best word in the world for her, but it is still not enough. We spend one year close with each other, only with each other most of the time, and that is the best time in my life. Then you grow up, and you rarely make any trouble for your dad and me. But my precious daughter, as a Mum, I have not given you enough time and attention. There is always a reason, but I still have to apologize to you. I lost lots of important moments with you, and it is impossible to come back.

Thanks to my parents Shengchao Cheng and Wei Su. I am so lucky to be your child, and I know you are proud of me too. I have received lots of unconditional love and time for accompaniment from you two, and that gives me the courage and confidence to face any difficulties. My father, Shengchao, taught me lots of important lessons since I was young. I learned the importance of independence. I know that I have to earn everything by myself. I learned the responsibility to the family. I learned a positive attitude from you as well. No matter what you experienced, you are always happy and keep a positive attitude. My mother Su, you are the best mum in the world. You gave me lots of time, love, and patient. No matter what mistakes I have made, you always forgive me and are never angry with me. You always gave me all you have. You are more caring about me than yourself. You let me understand what unconditional love is. I still remember and appreciate the happy moments we had.

Thanks to my parents-in-law Kuixin Zhang and Bingquan Xie. Thanks for your support, help, and understanding. As a daughter-in-law, I have

not done my best to take care of Yijing and Grace. However, rather than blaming me, you still support and encourage me. I am so lucky to have you be my parents.

Thanks to the staff from the ECRG group and VUW. You kindly remind me when I make mistakes. I know you are always there with me, even I am in trouble. That makes me comfortable and confident to deal with any difficulties. Thanks for the help from those parents and teachers from my daughter's daycare. There are lots of happy moments in my memory. I can tell you your kindness and assistance. Thanks for the help from lots of warm Kiwis from Hutt Valley, Wellington, and somewhere else in New Zealand. With the help you gave me, I can see, I can feel, and I do appreciate the help. That help is essential to keep me survive from the journey.

Thanks to my friends Lingli Cao, Lin Gong, Tony Zhang, Kelly Hou, Hayley Filipo, Ashley Asli Erginbbas, Lynley Jan Sheppard. Thanks to my colleagues from Hewlett-Packard, Bank of New Zealand, and Tertiary Education Commission. I still remember the happy moments with you and appreciate the time with you. Thanks Priscilla Belcher from Hope Centre Lower Hutt, I appreciate that you open a door of reality for me.

Thanks the beautiful Wellington, at here, I have seen empty streets at the early evening, I have seen the busy traffic at Sunday afternoon. I like the mild sunshine here, even if I don't need sunglasses, I don't feel dazzling. I experienced the Covid here, because of everyone's patience and cooperation, everything here is tightly organized. But I still hope that everything will end soon, because you don't see other people's real expressions when you put on the mask. At here, I learned how important the trust, love, and positivity is. Even I've made mistakes in a very specially condition by my kindness and fear, there is no excuse for it. If I've made mistakes accidentally or innocently, if it hurts something invisible but precious in other people's heart, there is no excuse for it. I paid off for what I should pay. I deserve everything I had or going to have.

Contents

Chapter 1

Introduction

1.1 Scope

Reinforcement Learning (RL) is a machine learning technique that learns the correct behavior to maximize reward from the interaction with the environment by itself. The main difference between RL and other machine learning methods, such as supervised learning, is that there is no existing teacher for training the RL agent. The agent has a specific goal and aims to learn the correct behavior to achieve that goal or maximize its reward by the "trial and error" method. For example, if the agent wants to open the door on its right, it should make sequential decisions to take a set of actions such as turn right, go straight, open the door, to achieve the goal. When taking an action, the agent will receive an immediate reward, which could be earned if the ultimate goal is not reached. Thus, the goal of RL is to find the optimal policy that maximizes the expected cumulative reward.

RL is applied to many real-world tasks such as robot control [49], game playing [79] and control of elevators [18]. The RL problem not only has the practical applications above but also benefits many other research areas, such as operations research [84], optimal control theory [31], and game theory [20]. In the operations research, RL methods are used to solve challenging benchmark problems, such as job-shop scheduling problems [29].

In the optimal control theory, RL methods are applied to solve the adaptive optimal control problems, for example, the adaptive control of the walking machine [42]. In economics and game theory, RL may be used to explain how equilibrium may arise under bounded rationality [20].

RL has a long history, which started from three research directions. One direction concerns the optimal control problem. The research on the "optimal control problem" started in the 1950s, which is a mathematical optimization method for deriving control policies. Dynamic programming (Bellman, 1957a) [77] and policy iteration (Ron Howard 1960) [77] are very important solutions to the optimal control problem. The other direction concerns learning by "trial and error", which comes from the animal learning psychology [77], which is also be called "Law of Effect" [Thorndike, 1911, p. 244]. The last direction concerns temporal-difference methods, which were proposed by Arthur Samuel in 1959 [77]. This has played a particularly important role in the field of RL. In the 1980s, these three threads joined together and formed the modern field of RL [77]. There are many methods developed to resolve RL problems, where the value function approaches [77] and direct policy search [23] are two major RL methods. Value function approaches attempt to find a policy that maximizes the long-term payoff by maintaining and updating a set of expected long-term payoffs for current policies. Another method is to search the optimal policies in the policy space directly.

When a problem can be defined as a Markov Decision Process (MDP), it can be addressed by RL algorithms [74]. In the MDP, the probability of the next state can be determined by the current state and action. A partially observable Markov decision process (POMDP) is a generalization of a Markov decision process (MDP). A POMDP models an agent decision process in which it is assumed that the system dynamics are determined by an MDP, but the agent cannot directly observe the underlying state.

Though there are many techniques to solve RL problems, however, the "black box" solution, for example, deep neural network [36] is hard to

meet the requirement when a user wants to understand not only what is the solution but also why is the solution. Explainable AI (XAI) is artificial intelligence (AI) in which the results of the solution can be understood by humans. It contrasts with the concept of the ”black box” in machine learning where even its designers cannot explain why an AI arrived at a specific decision. Explainable AI (XAI) is artificial intelligence (AI) in which the results of the solution can be understood by humans. It contrasts with the concept of the ”black box” in machine learning where even its designers cannot explain why an AI arrived at a specific decision. LCSs are the machine learning algorithms that can generate the rule-based solution, where the rule-based solution is easily understandable by the user. In this case, the user can see the rule-based solution but also can understand why a specific solution is suggested by the LCSs in that situation.

In practice, there are many problems with multiple conflicting objectives. For example, while making scheduling decisions on a shop floor, the objectives are minimizing not only the makespan but also the total tardiness and the energy consumption [47]. To find a control policy for releasing water from a dam requires balancing multiple uses of the reservoir, including hydroelectric production and flood mitigation, where RL has been used to balance the two conflicting objectives: economic benefit and environmental pollution [87]. MORL has an important value in practice as well as in theory. It is widely used in many applications. For example, in the process of the chemical reaction [85], the control system is designed to accelerate the chemical reaction, economically and energy saving by adjusting the higher temperature or adding the catalyst. But a higher temperature needs to cost more electricity and the catalyst is expensive. Therefore, the two objectives are saving electricity and saving the expense of catalysts. On the other hand, it is a sequential decision-making problem, as the situations are based on the decision made. In this case, MORL will provide a solution for these multiple objective sequential decision-making problems that balance these two objectives above to

accelerate the chemical reaction. Also, MORL algorithms provided a solution for the sequential decision-making problem with multiple objectives simultaneously. Hence, MORL becomes more and more popular recently.

When an RL problem needs to deal with multiple conflicting objectives at the same time, it is called a Multi-Objective RL (MORL) problem. There are many differences between the single-objective RL and MORL problems. In the MORL problem, the reward is a vector instead of a scalar, as each element in the vector is used to evaluate each objective. Besides that, the goal of MORL is, instead of optimizing one single objective, to obtain the optimal policies that optimize multiple criteria simultaneously. In a single-objective RL problem, where the reward is a scalar, the expected long-term payoff is a scalar as well. Thus, the optimal policy is the policy that maximizes the expected long-term payoff. However, considering the MORL problem, the reward is a vector, the expected long-term payoff is a vector as well. Therefore, it is impossible to employ the optimal criteria in single-objective RL that simply determines the policy with the maximum total reward as an optimal policy. In this case, the concept of Pareto Dominance [86] is employed to learn the multiple Pareto optimal policies [86]. More details will be discussed in Chapter 2, which will show the increasing popularity of MORL approaches.

However, MORL research is still limited. Currently, the MORL approaches can be categorized into two groups: single-policy approaches and multi-policy approaches. Single-policy approaches aim to find one optimal policy of a weighted sum of objectives that maximizes the total rewards according to the known preference in a single run. In the single-policy methods, the multiple objectives need to be aggregated together to produce a single objective for learning. Therefore, the MORL problem can be transformed into the single-objective RL problem, then solved by the classic single-objective RL approaches. However, as highlighted in [65], for many problems, it is very difficult to determine the relative importance of each objective without knowing the actual solution (i.e. before apply-

ing the RL algorithms). Instead, each objective has to be treated separately during learning. In this case, multi-policy approaches, whose aim is to find a set of Pareto optimal policies, can be applied. Apparently, for this kind of MORL problem, a different learning technique is highly desirable as it does not need any prior knowledge to learn all the optimal policies and the user can choose the optimal policy according to their requirement later.

Many existing multi-policy algorithms including Pareto Q-Learning (PQL) follow the tradition of adopting a tabular representation of the value function [6][69][86]. Consequently, they may face difficulty when tackling large-scale problems (scale in terms of the number of states and actions). To achieve better scalability, rule-based representation has been developed in XCS for scalable RL [15]. However, only Stanley and Bull have attempted to solve MORL problems by using XCS [75]. Their research considers those problems where one or more objectives can be conveniently treated as part of the state input, thereby leaving only a single objective as the learning target. In this thesis, a different type of problem is considered where no objective can be treated as part of the state input through an extended state input. Therefore, rather than learning a single optimal policy, the goal of this thesis is to learn a group of policies that jointly form the Pareto Front (PF), The PF consists of a set of non-dominated solutions, where no objective can be improved without sacrificing at least one other objective. In addition, the neural network-based representation used previously in [24] provides a more compact and flexible solution to Multi-Objective Problem (MOP) [91]. In this thesis, how to improve the existing network-based solutions method [91] to make it more efficient for MORL problems will be considered as well.

The performance assessment method of a MORL system is different from the single-policy approaches and multi-policy approaches as the number of solutions in each run are different. In single-policy approaches, the performance can be accessed by evaluating the performance multiple

times. At each run, the performance is evaluated according to one given preference metric. In the multi-policy approaches, the performance of the MORL system will be evaluated by comparing the approximate Pareto front with the true Pareto front. Several benchmarks with the known Pareto front are presented in [83], such as Bi-Objective Maze, Deep Sea Treasure will be used to evaluate MORL algorithms in this thesis. Several single objective benchmarks, CoinRun and Mountain Car, will be considered to extend as multi-objective benchmarks for evaluating the performance as well.

1.2 Motivation

Though the research of RL is promising, the research into MORL especially the MORL solutions on multi-policy approaches is still limited. Therefore, in this part, first, the challenges of existing Multi-Objective RL approaches are discussed. Afterward, the potential solution: XCS-based algorithms are proposed, together with the reasons why the algorithms are considered. Finally, the challenges of developing the XCS-based algorithms for the MORL problem are discussed.

1.2.1 Challenges of Existing Multi-Objective RL approaches

As discussed above, MORL is an important research area, but developing the MORL algorithms is challenging because:

1. The tabular representation of the solutions of MORL has poor scalability. Despite the successful developments in MORL theory and algorithms, the tabular-based solution lacks scalability, as it cannot address the 'curse of dimensionality problem. In a reinforcement learning problem, the curse of dimensionality means that the algorithm's computational requirements grow exponentially with the number of state variables [77]. In addition, the tabular representation is not

straightforwardly applicable. For example, the tabular representation is not easy to apply to a real system because it needs a large computational space to store the solution to a large problem. Consequently, they may face difficulty when tackling large-scale problems (scale in terms of the number of states and actions).

2. The Non-Markov environment is the environment that the current state could not be determined with the condition. Research in the area of the Non-Markov MORL environment is very limited [86]. To date, most RL and MORL work has focused on learning tasks that can be described as Markov Decision Processes (MDP). If a problem can be defined as an MDP, it can be solved by RL algorithms [74]. However, while those algorithms are useful for modeling a wide range of sequential-decision problems, there are important tasks that are inherently Non-Markov. A small-scale Non-Markov problem can be solved by encoding the historical environment information into the states. However, it is impossible to record all the historical information and embed it into the states in a large-scale MORL environment. For example, when a robot system has a limited set of sensors that do not always provide adequate information about the current state of the environment [90]. In this case, the existing MORL cannot handle them. Therefore, developing compact solutions with good scalability to resolve the large-scale Non-Markov MORL problems efficiently is highly required.

3. Research in the area of the continuous MORL environment is very limited [63]. Though some existing MORL techniques can learn the Pareto optimal solutions, they can only be applied to the solution to a discrete environment [86]. However, most robotic applications of reinforcement learning require continuous state spaces defined using continuous variables. For example, when driving a car, the wheel angle of the car is a continuous signal. The usual approach has been to

discretize the continuous variables, which quickly leads to the combinatorial explosion and the well-known "curse of dimensionality" (The computational requirements grow exponentially with the number of state variables [77]). In addition, in the continuous environment, the solution space is much bigger than that in the discrete environment as the solution to the continuous environment is infinite. Hence, the number of policies will be much larger than that in the discrete space, therefore how to express the solutions more compactly is also a challenge. Therefore, developing compact solutions with good scalability to resolve the continuous MORL problems efficiently is required.

4. Research in qualifying the generalization ability of MORL algorithm is very limited [17]. Though MORL algorithms can solve complex tasks, they struggle to measure the generalization ability of a MORL algorithm. This is because the existing MORL benchmarks still train and test the MORL algorithms on the same sample space in an environment. However, in the real world, it would be quite common that the testing environment is a different data sample from the training environment. In this case, existing MORL benchmarks cannot measure the generalization ability of MORL algorithms. Therefore, developing a benchmark that can measure the generalization ability of the MORL algorithm is required.

1.2.2 Why Evolution Computation Algorithms for MORL

As discussed above, the existing MORL algorithms based on tabular representation can not scale well. Therefore, the traditional MORL can only be applied to some small problems. To improve the scalability of methods to learn the Pareto optimal policies. This thesis, a new engine-rated approach is requited considers that Evolutionary Algorithms (EAs), especially the symbolic approach of Learning Classification Systems (LCSs),

are suited to address MORL due to the following reasons.

1. Evolution Computation Algorithms are less sensitive to the shape or continuity of the Pareto front than other techniques, such as Neural Network, and can easily deal with the complex Pareto front. This is because Evolution Computation Algorithms are search-based optimization algorithms used to find optimal or near-optimal solutions for search problems and optimization problems. On the other hand, a neural network is a mathematical model that is capable of solving and modeling complex data patterns and prediction problems. In MORL problems, the Pareto Front shape changes according to the specific problems. Especially, in a continuous or large-scale environment, the Pareto front might be very complex [26]. In this case, other methods, such as using a linear function to approximate the Pareto front can not be used [6].

2. Due to their population-based nature, Evolution Computation Algorithms can explore the solution space to a MOP in different directions in a single run rather than focusing on only one direction. Thus evolutionary algorithms can approximate the whole Pareto front of a MOP [98]. The benefit of learning the Pareto Front is identifying the trade-off between objectives, thus Pareto Optimization has become the dominant method for multi-objective nonlinear optimization problems. Many algorithms such as gradient descent can only explore the solution space of a problem in one direction at a time. If the solution they discover is not optimal, they will abandon this solution and start over to search for another solution. However, since Evolution Computation Algorithms are population-based methods, they can explore the solution space in multiple directions at once. If one solution turns out to be not good, they can easily eliminate it and continue work on other individuals with better potential, thus giving them a greater chance of finding good solutions on the Pareto

front.

3. Evolution Computation Algorithms are particularly well-suited to solving problems where the space of all potential solutions is huge [98]. In Evolution Computation Algorithms, the mutation is a divergence operation. It is intended to potentially explore and discover a new or better solution in a possibly huge solution space. On the other hand, crossover with selection is a convergence operation that is intended to pull the population towards a good solution. Since the end goal is to bring the population to convergence, selection and crossover happen more frequently. Mutation, being a divergence operation, should happen less frequently, and typically only affects a few members of a population in any given generation. In this case, Evolution Computation Algorithms can search in the large solution space to find the optimal solutions.

4. Evolution Computation Algorithms support different solution representations, for example, the rule-based solution and the neural network-based solution. The EA method XCS can develop a rule-based solution, where a rule-based solution to multi-objective problems is one of the solution types that will be developed in this work. This is because it has good scalability as the condition of a rule is dynamic rather than fixed. In addition, the rule-based solution is easier to understand than the tabular-based solutions and is explainable to the user.

5. There are some Multi-Objective Evolution Computation Algorithms to solve Multi-Objective Problems (MOPs) [96], thus, they have the potential to solve MORL problems. For example, MOEA/D is a Multi-objective Evolutionary Algorithm Based on Decomposition. It decomposes a multi-objective optimization problem into several scalar optimization sub-problems and optimizes them simultaneously. Each sub-problem is optimized by only using information from its

several neighboring sub-problems. The result has been shown that MOEA/D can generate a set of very evenly distributed solutions for some benchmarks of MOPs.

1.2.3 Challenges to Developing EC-based Algorithms for MORL

In this thesis, rule-based EC-based Algorithms are considered to solve the challenges of existing MORL problems. In this section, the challenges of developing the rule-based solution for MORL will be discussed.

1. In most of the existing MORL algorithms [86], they assume a tabular representation of the value function, thus their applicability is limited to small applications. To tackle this issue, XCS is considered a potential solution that can extend the MORL algorithms for large applications and provide a rule-based solution. XCS's fitness bias prefers general but accurate rules. Combined with GA algorithms that work as the rule discovery component, where the rule discovery component can help generate new rules and tend to delete the rules those are not useful, XCS aims to find accurate, general rules [15]. However, this thesis is aiming to seek a set of Pareto optimal policies, in which the solution space is huge. Therefore, how to use a collection of rules to cover multiple Pareto-optimal policies to approximate the whole Pareto front is a big challenge.

2. When a multi-objective sequential problem can be defined as a Markov Decision Process (MDP), with proper training by the RL algorithm, the agent can make the correct decision at different states. However, when an environment is partially observable, for instance, a robot with a limited sensor, it cannot always know what is the current state according to the information from the sensor. Thus, it cannot make the correct decision in the current state. Although there is some exist-

ing research in solving the Non-Markov problem, for example, Anticipatory Classifier Systems (ACS) [11], it provides the solution to the small-scale single objective Non-Markov problem. ACS can be considered as a potential solution to multi-objective PO-MDP, but it is still challenging. First, the strategy of ACS to a PO-MDP problem is to backup one historical state in the memory, thus the algorithm can tell what in the current state is different from the information of the previous state. However, in a large-scale PO-MDP environment, sometimes, with the information of just one previous historical state, the LCS still cannot tell what is the best action in the current state. In addition, multi-objective PO-MDP needs a large memory to back up the previous state information for different objectives, which is another technical challenge.

3. The generalization ability of MORL algorithms is uncertain as the data sample is used for training and test. However, it is common that the testing environment is different from the training environment in the real world. However, there are limited benchmarks for measuring the practical generalization ability of MORL algorithms. There are existing benchmarks for MORL algorithms, for example, Deep Sea Treasure and Multi-Objective Mazes [83]. However, they use the same environment for both training and testing, thus it could not measure the practical generalization ability of MORL algorithms. In this case, a new benchmark needs to be built for evaluating generalization by creating training and testing data sets with separate samples. The criteria to evaluate this benchmark should be 1) the training and testing environment can be resolved by the algorithm separately 2) know how much differences between the training and testing environment 3) the solutions to the testing environment can be learned from the training environment.

4. In the continuous MORL problem, as the continuous states and ac-

tion need to be handled, both the search space and solution space are even larger than that in the discrete MORL environment. In this case, the need for generalization of the solution is pronounced, and the generalizing is not just over the states but also actions. Therefore, it is necessary to use the generalization capabilities of Learning Classifier Systems to learn the Pareto Optimal solutions on the continuous Pareto front. However, as the number of Pareto optimal policies in the continuous environment is infinite, it is not possible to learn all of them. In this case, how to learn the limited Pareto optimal solutions that give a well-distributed approximation of the Pareto front effectively and efficiently (using limited solutions to approximate) is a huge challenge.

1.3 Thesis Goals

The overall goal of this thesis is to determine compact and effective solutions to the MORL problems by developing LCS-based algorithms. The specific research objectives of this thesis are listed as follows.

1. Develop XCS-based algorithms to learn the Pareto optimal policies for MORL problems.

   (a) Adding Pareto Dominance method to XCS to learn the Pareto optimal policies for MORL problems.

   There are many technical challenges when dealing with the MORL problem with XCS. First, how to update the prediction with the observation is an issue. As the prediction becomes a set of vectors in MORL problems, the prediction cannot be updated recursively with the observation as in a single-objective RL problem. To solve this issue, using the idea of Pareto Q-Learning in the multi-objective XCS will be considered, which uses the observation to replace the prediction. Second, how to calculate the

error estimation (the difference between the prediction and the observation) in multi-objective XCS is another challenge. The error estimation, which is used to calculate the fitness of the classifier, becomes the distance of two sets of vectors. There are many methods to calculate the distance between two sets of vectors, however, different methods may lead to different results. In this case, the results of the distance between two sets of vectors infect the values for the error estimation. Therefore, the cooperation of different methods to find one of the best methods for error estimation needs to be investigated. In this work, an XCS-based algorithm to learn the Pareto optimal policies will be developed to cope with those issues above. To our knowledge, this work will be the first multi-objective XCS-based algorithm to learn the Pareto optimal policies.

(b) Improve the accuracy of the predictions learned through Pareto Q-Learning in multi-objective XCS.
In XCS, when the agent reaches a new state, an action is selected to be performed in the current state according to the fitness-weighted average of the predictions of all classifiers matching the current state. When the predictions in XCS are poor, the action suggested by the XCS could be wrong. Therefore, the prediction is important for the XCS. In the traditional XCS, the prediction of the general classifier is expected to reflect the average prediction of all the states it matches. However, in the preliminary work of the multi-objective XCS, rather than reflecting the average prediction, the prediction of the general classifier has only reflected the prediction of one state. Therefore, improving the accuracy of the prediction for the general classifiers is another technical challenge in multi-objective XCS. To resolve this issue, an adaptive control strategy will be performed on the prediction calculation to make the prediction not only for one state.

In this case, it will improve the accuracy of the multi-objective XCS to make it more effective.

(c) Investigate new methods to trace a specific optimal policy from a set of learned Pareto optimal policies.
In the preliminary work, once one policy is selected, it must be followed until achieving the final state, otherwise, the agent will be trapped. For example, suppose there are two Pareto optimal policies p_1 and p_2, which have opposite action directions on two connected states s_1 and s_2 (s_1 is on the left of s_2). p_1 suggests the agent go to s_2 from s_1, whereas p_2 suggests the agent go to s_1 from s_2. In this case, the agent will oscillate between s_1 and s_2. To solve this issue to follow one optimal policy, the long-term payoff in each state generated by this optimal policy is traced. But this strategy is not reliable when one vector on the path is not exactly learned, it will fail to get to the final state sometimes. Therefore, it would also be interesting to investigate approaches for policy trace.

2. Developing a new multi-objective XCS algorithm to discover multiple optimal policies simultaneously that generalize where possible, without large storage requirements.

(a) Employ a decomposition strategy based on MOEA/D in XCS to approximate complex Pareto Fronts.
It is hypothesized that the decomposition of MOEA/D for multi-objective tasks can sufficiently approximate the complicated PF shapes to enable such policy learning within XCS. MOEA/D first employs a decomposition approach to transform the MORL problem into N single objective RL sub-problems. Then XCS can be used to solve each single objective RL problem simultaneously, such that relevant learned knowledge can be shared amongst XCS rules.

(b) Update XCS to learn and maintain multi-objective predictions.
First, to enable XCS to maintain multiple objectives, the classifier structure, e.g. prediction, error, and fitness needs to be extended from a number to a vector for storing those values for multiple objectives. Secondly, according to the changes in the classifier structure, how XCS updates classifier parameters need to be amended from updating a number to a vector. Third, correspondingly, the Genetic Algorithm (GA) process needs to be updated to discover multi-objective optimal rules.

3. Evaluating the generalization ability of developed multi-objective XCS-based algorithms for solving MORL problems.

   (a) Developing an environment for evaluating the generalization ability of MORL algorithms.
   Among the most common benchmarks in Reinforcement Learning, it is common to use the same environments for both training and testing. OpenAI introduced a new environment called CoinRun, designed as a benchmark for generalization in RL [17]. To test the generalization ability of MORL, the CoinRun environment will be updated to a multi-objective reinforcement learning environment by adding another objective.

   (b) Evaluating generalization ability of multi-objective XCS-based algorithm on multi-objective CoinRun problems.
   The CoinRun environment is designed as a continuous RL problem, thus the multi-objective CoinRun environment is designed in the continuous space as well. However, the XCS-based algorithm is designed for solving the discrete MORL problems, therefore, the continuous inputs of the states and actions from the CoinRun environment need to be discretized.

4. Developing XCS-based algorithm to address large-scale MORL problems.

The mountain car problem is commonly applied as a benchmark for the RL algorithm because it requires an agent to learn two continuous variables: position and velocity. Although there is existing research to discretize the continuous environment to discrete environment, the mountain car problem cannot be solved manually as it is a large-scale Markov problem and the optimal policy contains over a hundred steps. In this research, the mountain car problem will be extended as a MORL problem. First, the continuous environment will be discretized as integers. Second, to handle the Markov MORL problem, the XCS-based MORL algorithm needs to be updated to consume integer inputs from the environment. In addition, the value function will be plotted for analyzing the effectiveness of RL algorithms to find what is the strength of the success algorithm.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 presents essential background and reviews the related works in some existing Multi-Objective Reinforcement Learning algorithms. Chapter 3 presents an LCS-based algorithm MO-XCS, which adding Pareto to XCS to solve Multi-Objective Reinforcement Learning Problems. Chapter 4 presents a Multi-Objective Reinforcement Learning algorithm MOXCS, which is a decomposition-based Multi-Objective evolutionary algorithm in XCS for solving Multi-Objective Reinforcement Learning problems. Chapter 5 is quantifying the generalization ability of MOXCS in Multi-Objective Reinforcement Learning problems. Chapter 6 is using MOXCS to solve large-scale Non-Markov MORL problems. Chapter 7 concludes this work.

Chapter 2 provides an overview of reinforcement learning, multi-objective reinforcement learning, algorithms for solving reinforcement learning problems. First, a reinforcement learning problem definition is described as a Markov Decision Process or a Partially Observable Markov Decision Pro-

cess. Secondly, it reviews the multi-objective reinforcement learning problem. Third, traditional reinforcement learning algorithms are reviewed. Fourth, it reviews the evolutionary computation algorithms for RL. Fifth, the evolutionary computation algorithms for MORL are reviewed.

Chapter 3 presents the methodology of developing the MORL algorithm MO-XCS, which is adding Pareto Dominance to XCS to solve Multi-Objective Reinforcement Learning Problems. MO-XCS is tested on three bi-objective mazes to show the effectiveness of solving MORL problems.

In Chapter 4, a Multi-Objective Reinforcement Learning algorithm MOXCS is proposed, which is a decomposition-based Multi-Objective evolutionary algorithm in XCS for solving Multi-Objective Reinforcement Learning problems. MOXCS will solve the three bi-objective mazes. With integer inputs, MOXCS can solve the Non-Markov problem Deep Sea Treasure. Lastly, the generalization ability of MOXCS is tested in the introduced Multi-Maze and Multi-Maze Connection environment, but the Multi-Maze Connection is not a proper benchmark for testing the generalization ability of the MORL algorithm.

In Chapter 5, CoinRun Env1 and Env5 are introduced and solved by XCS, and the generalization ability of XCS is tested and demonstrated by crossover Env1 and Env5 in each experiment as the training and testing environment. Then the CoinRun environment is extended as a bi-objective CoinRun environment and solved by MOXCS and tested the generalization ability of MOXCS by crossover the training and testing environment. After that, the Pareto Front learned by MOXCS in a bi-objective CoinRun environment is discussed.

Chapter 6 is MOXCS is used to solve the continuous MORL problem in a large sparse domain. First, XCS and Q-Learning are implemented to resolve the mountain car problem. The result shows XCS converges better than Q-Learning. Then, the mountain car problem is extended as a bi-objective MORL problem. MOXCS is implemented to resolve the multi-objective mountain car problem. It shows MOXCS has the potential to

resolve the multi-objective mountain car problem.

Chapter 7 presents a summary of work from each contribution chapter, contributions, future work, and the thesis summary.

Chapter 8 list some algorithms we may use in this thesis.

Chapter 2

Literature Review

This chapter covers essential concepts of RL and MORL for a better understanding of RL and MORL. The related works are divided into four parts, namely, Explainable Artificial Intelligence (XAI), traditional RL, MORL approaches, and the evolutionary computation (EC) methods for RL. For XAI, the difference between traditional AI and XAI is introduced. For traditional RL, important classic algorithms are introduced, such as value function-based approaches, and policy search approaches. For the MORL approaches, the single-policy and multi-policy approaches are reviewed. However, few MORL algorithms can learn the Pareto optimal policies [86] and solutions with tabular representation lack scalability to large problems. Though some methods can cope with the scalability issues, such as Policy Search [64] and EC [91], we consider EC as the solution in this work as it is more suitable for the multi-objective problems. Therefore, the EC methods for RL are reviewed in this chapter, especially the rule-based and neural network-based EC algorithms. The EC methods have been much researched for the single-objective RL, but there are limited works on MORL. With its good scalability and the evidence from the literature, we can see the potential of EC for handling MORL problems, which is this proposed work aims to achieve.

2.1 Explainable Artificial Intelligence

Nowadays, Artificial Intelligence is widely used in many industries and our daily life. For example, it provides personalized recommendations to people based on their previous searches and purchases or other online behavior; it predicts the sales of the products according to the sales histories and other factors; it can even give financial advice to people according to their financial situation.

However, the explainability of the algorithm is critical for AI applications. To accept and use the AI prescriptions, human needs to trust the AI algorithms, and explainability of the algorithm results are critical to buildup the trust [32]. Another reason for seeking explainability of AI algorithm results is because AI systems sometimes learn undesirable tricks that do an optimal job of satisfying explicit goals on the training data, but that do not reflect the complicated implicit desires of the human system designers. For example, a 2017 system tasked with image recognition learned to "cheat" by looking for a copyright tag that happened to be associated with horse pictures, rather than learning how to tell if a horse was pictured [32].

In this case, with Explainable Artificial Intelligence (XAI), users should be able to understand the AI's decision and should be able to determine when to trust the AI and when the AI should be distrusted [39]. Explainable Artificial Intelligence (XAI) is Artificial Intelligence (AI) in which the results of the solution can be understood by humans. It contrasts with the concept of the "black box" in machine learning where even the designer of the algorithm cannot explain why an AI arrives at a specific decision [32]. XAI is a "white box", which is not only able to tell the user what is this solution but also enable the user to understand why it is this solution when its application will lead to success, and when to trust its output and why it erred if errors occur.

For example, although a deep neural network can yield highly effective

results according to the data set size and data set quality, it is inherently a dark black box by nature. A network is embedded in thousands of neurons, which are stacked and learned through multiple layers. Each of the neurons in the first layer receives input and then performs a calculation before sending a new signal as an output and the input to the next layer, and this process continues until a final output at the last layer is produced. However, as a user, you could not see and understand what is the input and output from every single layer of a deep neural network as some of the outputs in the middle layers. The middle layers do support the final output, so they are meaningful in this sense. Middle layers output is often not in a realistic space, i.e. There does not exist an equal in real life, instead, they operate in latent space. On the other hand, Learning Classifier Systems work as a "white box". The output of Learning Classifier Systems is a rule-based solution, each rule is associated with the long-term payoff of an action under a condition. Under a specific condition, the system will suggest the action with the maximum long-term payoff at that condition. Users can observe what is the long-term payoff of an action at that condition, thus understand why the system suggests that action at the condition. When the suggested action has the maximum long-term payoff compared to other actions under the same condition, the result is reliable. In this case, the user can know when to trust the solution and why the solution erred.

2.2 Machine Learning

Machine Learning is the study of computer algorithms that improve automatically through experience and by the use of data. It is seen as a part of artificial intelligence. Machine learning algorithms learn the pattern from data and create a machine learning model. With the learned model, new data set can be predicted according to the pattern. There are many machine learning algorithms.There are several types of Machine Learning al-

gorithms, supervised learning, unsupervised learning and reinforcement learning. Supervised Learning algorithm is to train the model with a set of labeled data. With the label, a model can learn from it. Labeled data set means, for each data set given, an answer or solution to it is given as well. This would help the model in learning and hence providing the result of the problem easily. The supervised learning task can be classification and regression. The major difference between supervised and unsupervised learning is that there is no complete and clean labeled data set in unsupervised learning. The machine learning algorithm is trained with a set of unlabeled data without any guidance and learn the pattern in it. The unsupervised learning task can be aggression. Whereas reinforcement learning is when a machine or an agent interacts with its environment, performs actions, and learns by a trial-and-error method. Generalization is a term that used to describe a model's ability to react to new data. As an user always expect the learned model can predict the unknown data accurately. Generalization is a common challenge in Machine Learning.

2.3 Reinforcement Learning

RL is an important AI technology, which lets the agent learn the correct behavior from the interaction with the environment by the "trial and error" method [78]. The main difference between RL algorithms and other machine learning methods, like supervised learning, is that there are no existing optimal solutions provided to the learner [78]. Instead, the learner must learn solutions that maximize the long-term payoff from the interaction with the environment. When taking an action, the agent will receive a reward or punishment from the environment. Finally, the agent learns how to take action in an environment to maximize the cumulative total reward, namely, the long-term payoff. To calculate the long-term payoff, it not only considers the immediate reward but also the rewards received by the agent. RL has been widely used in many real-world ap-

plications, for example, a cleaning and maintenance robot has been deployed to complete coverage path planning using reinforcement learning [51]. With proper training, the model is minimizing the transformation and rotational actions and time to generate the path with lower cost than tiling methods and is able to determine the optimal solution for each environment with different obstacle settings.

2.3.1 Markov Decision Process

An RL problem can be formulated as a Markov Decision Process (MDP) which is described through a 5-tuple $\{S, A, P, r, \gamma\}$ [58]. Here, S is the set of states in the learning environment while A represents the set of actions; a transition function $P(s'|a, s)$ gives the probability of reaching state s' upon taking action a in state s; $r(s, a)$ stands for a scalar reward provided by the environment after performing action a. γ is a *discount factor* between 0 and 1, which represents the difference in importance between current rewards and future rewards. Such a RL problem can be solved by identifying a *deterministic policy* π that maps every state $s \in S$ to a specific action $\pi(s) \in A$. Or it can also be solved by identifying a *stochastic policy* π that maps every state $s \in S$ and action $a \in A$ to a probability $\pi(s, a) \in [0, 1]$. Given a starting state s_0, the *long-term payoff* of following policy π is defined as the *expected cumulative reward* in the long run is as follows:

$$J^{\pi} = E^{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\} \tag{2.1}$$

where, t represents at any random time step, and E represents the expectation under the policy π. Based on (2.1), the goal of RL is therefore to learn the optimal policy π^* that maximizes the long-term payoff. For the single-objective RL problem introduced in this subsection, the reward signals r are always measured by a scalar. Therefore, the long-term payoff to be maximized through RL is also a scalar.

In order to explain the RL algorithms, it is necessary to introduce the

concept of Value Functions first. The value function-based RL algorithms utilise value functions to estimate the long-term payoff of an action that the agent performs in a given state. The value functions include functions of states or state-action pairs, both of the functions are defined with respect to particular policies. The value of a state s under a policy π, denoted $V^\pi(s)$, is the long-term payoff when starting in s and following π:

$$V^\pi(s) = E^\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\} \tag{2.2}$$

The function V^π is called the state-value function for policy π. Similarly, the value of taking an action a in state s under a policy π, denoted as $Q^\pi(s, a)$, is the expected long term payoff when starting in s, taking the action a and following π:

$$Q^\pi(s, a) = E^\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\} \tag{2.3}$$

The function Q^π is called the action-value function for policy π.

On-Policy and Off-Policy Learning are two learning methods in RL. The difference between them is how they obtain the policy that they are trying to learn. In the On-Policy method, the agent selects a policy, evaluates it, and moves on to better policies, whereas in the Off-Policy method, the agent collects all available information so can update its policy assuming it between greedily in future steps. The distinction disappears if the current policy is a greedy policy.

2.3.2 Partially Observable Markov Decision Process

A partially observable Markov decision process (POMDP) is a generalization of a Markov decision process (MDP). In POMDP, same with MDP, we have a set of states S, a set of actions A, transitions P, immediate rewards r and a discount factor γ we also have observations. The actions' effects on the state in a POMDP are exactly the same as in an MDP as well. The

only difference is in whether or not we can observe the current state of the process. In a POMDP, we add a set of observations O to the model. So instead of directly observing the current state, the state gives us an observation, which provides a hint about what state it is in. In this case, since we have no direct access to the current state, it must maintain a probability distribution over the set of possible states, based on a set of observations and observation probabilities.

2.3.3 Benchmarks

The RL field has been explored through the use of benchmarks for testing the associated algorithms. The maze problem is widely used for testing RL algorithms [35], as it is suitable for beginners. Recently, more feature-rich domains have been introduced as challenging and widely-used benchmarking domains, such as the Atari Learning Environment [7]. These benchmarks enable fair and easy comparison of RL methods, which is useful to properly evaluate progress and ideas assume the reader will know this.

However, the current benchmark reinforcement learning (RL) tasks are in many ways poorly suited to evaluate the generalization ability of RL algorithms. By testing RL algorithms on the same environments as training, these RL algorithms are often poor at generalizing beyond the very specific domain. To combat this, there are several benchmark RL domains proposed that separate the train and test environments. In order to investigate the problem of overfitting in deep reinforcement learning, Karl Cobbe, et al. [17] generated environments to construct distinct training and test sets. They introduced a new environment called CoinRun, designed as a benchmark for generalization in RL forward are to where it is processed. As testing increasingly complex RL algorithms on low-complexity simulation environments, we often end up with brittle RL policies that generalize poorly beyond the very specific domain. Zhang, et al. [1] proposed

natural environment benchmarks of RL domains that contain some of the complexity of the natural world. They proposed three new families of benchmark RL domains that contain some of the complexity of the natural world, while still supporting fast and extensive data acquisition. The proposed domains also permit characterization of generalization through fair train and test separation, and easy comparison and replication of results. This work challenges the RL research community to develop more robust algorithms that meet high standards of evaluation as future work.

2.4 Traditional RL Algorithms

In this section, some traditional RL algorithms are investigated. The value function-based approaches are reviewed first as they are quite straightforward and basic approaches. However, as value function-based approaches have some limitations, such as they cannot learn the stochastic rules and the convergence assurances issue, the policy search-based approaches that are assumed to solve those problems are reviewed afterward [64]. Finally, the MORL algorithms including the single-policy approaches and multi-policy approaches are discussed.

2.4.1 RL Algorithms based on Value Function

In this section, several RL Algorithms based on Value Function will be reviewed. It starts from TD-learning as it is the most basic value function-based algorithm [80]. After that, Q-Learning and SARSA will be reviewed [89] [66], which is respectively the offline TD-learning and the online TD-learning method.

TD-learning

Before introducing the TD-learning algorithm, let us consider the following issue. If the value functions were calculated without estimation, the

state values have to be updated after the final reward is received. Once the final reward is received, the trajectory to reach the final state would need to be traced back and each value updated accordingly. In this case, as the agent will learn only when the final reward is received, it will take too long for the agent to learn.

The problem solving by **TD-learning** is estimating these value functions for a particular policy. With TD methods, an estimate of the expected final reward is calculated at each state. Thus, the state value can be updated for every step. This can be expressed formally as:

$$V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{2.4}$$

where r_{t+1} is the observed reward at time $t + 1$.

The key idea of the TD method is: the value is updated partly using an existing estimation and not a final reward.

Many other algorithms are based on the idea of TD-learning, such as Q-learning and SARSA as following.

Q-learning

Q-Learning is an Off-Policy algorithm for Temporal Difference learning [78]. The reason that Q-learning is off-policy is that it updates its Q-values using the Q-value of the next state s' with the greedy action a' rather than necessarily the action is taken. The Q-learning algorithm works by estimating the values of state-action pairs rather than the values of states. The purpose of Q-learning is to generate the Q-table, Q(s,a), which uses state-action pairs to index a Q-value of that pair. The update rule for setting values in the table is as below:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_{a+1} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.5}$$

The parameter used in the Q-value update process is the learning rate, α, which is between 0 and 1. If it is set to 0, the Q-values will be never

updated, hence, nothing is learned. In the contrast, if it is set to a high value, such as 0.9, the learning can occur quickly but the error will increase. The discount factor γ is also set between 0 and 1. When γ is close to 1, Q-Learning considers the future reward more than the immediate reward. $Q(s_{t+1}, a_{t+1})$ is the Q-value when taking action a_{t+1} in the next state s_{t+1}, and the $\max_{a+1}$ means a_{t+1} maximizes the expected long-term payoff when it is performed in the next state S_{t+1}.

SARSA

The **SARSA** algorithm is an On-Policy algorithm for TD-Learning. The reason that SARSA is an On-Policy method is that it updates its Q-values using the Q-value of the next state s' and the current policy's action a' in that state. It estimates the return for state-action pairs assuming the current policy continues to be followed. The name SARSA actually comes from the fact that the updates are done using the quintuple Q(s, a, r, s', a'). Where: s, a are the current state and action, r is the observed reward in the following state s and s', a' are the next state-action pair. The major difference between SARSA and Q-Learning is that the maximum reward for the next state is not necessarily used for updating the Q-values. Instead of computing the difference between Q(s,a) and the maximum action value in Q-Leaning, SARSA computes the difference between Q(s,a) and the value of the actual action that was taken. The updating form of the SARSA algorithm is comparable to that of Q-Learning:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_t) - Q(s_t, a_t)] \tag{2.6}$$

Similar to Q-Learning, α is the learning rate and γ is the discount factor.

In this case, the advantage of Q-Learning is able to learn solutions fast, whereas, SARSA is able to learn the solution more steadily.

2.4.2 RL Algorithms based on Policy Gradient

As discussed, the value function approach is very important in the RL field. However, there are several limitations. First, it is oriented toward seeking deterministic policies, but sometimes the policy is stochastic [67]. Second, during the learning process of the value function approach, a small change may result in an action to be or not to be selected. To solve the issues above, another method to find a good policy is proposed, namely, search the optimal policy directly in the policy space. In this proposal, we will review Policy Gradient Methods as they are highly related to our work.

The policy can be represented as a parameterized function, where input is a set of states, the output is the probabilities of action selection. Let θ denotes the policy parameters, J^π denotes the performance of the policy as following.

$$J^\pi = E[\sum_{t=1}^{\infty} \gamma^{t-1} r_t | s_0, \pi] \tag{2.7}$$

Then, in the policy gradient method, the $principle$ of policy parameters updating work like below,

$$\Delta\theta \approx \alpha \frac{\partial J^\pi}{\partial \theta} \tag{2.8}$$

where α is the learning rate, the $\frac{\partial J^\pi}{\partial \theta}$ is the policy gradient. After the learning process in Algorithm 1, θ will help to maximize J^π, thus obtaining local optimal policy. Therefore, the key task of policy gradient method is to approximate the value of $\frac{\partial J^\pi}{\partial \theta}$.

There are many ways to approximate the $\frac{\partial J^\pi}{\partial \theta}$ [64] [93]. One method is proposed by Sutton [64] as following.

$$\frac{\partial J^\pi}{\partial \theta} = \sum_s d^\pi(s) \sum_a \frac{\partial \pi(s,a)}{\partial \theta} Q^\pi(s,a) \tag{2.9}$$

where the $d^\pi(s)$ is the stationary distribution of states under π. Based on (2.9), several policy gradient methods are proposed [93] [64] as following.

Algorithm 1 Step-Based Policy Gradient

procedure STEP-BASED POLICY GRADIENT

Initialize the policy parameter θ

REPEAT:

Take action a *at current state* s *following the policy with parameter* θ

Evaluate the policy performance of current policy with parameter θ

Adjust θ *with equation (2.8)*

$\Theta \leftarrow \Theta + \alpha\Delta\Theta$

$s \leftarrow s^{'}$, $a \leftarrow a^{'}$

UNTIL:

Until Policy converges

end procedure

Algorithm 2 Episode-Based Policy Gradient

procedure EPISODE-BASED POLICY GRADIENT

Initialize the policy parameter θ

REPEAT:

Sample the trajectory following the policy with parameter θ

Evaluate the policy performance of current policy with parameter θ

Adjust θ *with equation (2.8)*

UNTIL:

Until Policy converges

end procedure

REINFORCE

REINFORCE is proposed by Williams [93]. When calculating the policy gradient in equation (2.9), $Q^{\pi}(s,a)$ is approximated by the actual returns J, which can be calculated as below,

$$J = E[\sum_{k=1}^{\infty} \gamma^{k-1} r_k] \tag{2.10}$$

This leads to Williams's episodic policy gradient method. However, the issue with this method is that they may get stuck in local optima.

Actor-Critic with Policy Gradient

There are algorithms that mix value function and the policy search method, such as **Actor-Critic method** [64]. The Actor-Critic method combines the strengths of both value function and policy search methods. It is divided into two parts. The policy structure is known as the actor, which is used to select actions. The other value function structure is known as the critic, which is used to criticize the actions made by the actor. When adding the policy gradient method into Actor-Critic, there are two parameters added. One parameter θ in π is used to adjust the policy, the other parameter w with a state feature function $\Phi(s,a)$ is used to approximate the state-value function $Q(s,a)$ as below.

$$Q_w(s,a) \approx \overrightarrow{w}^T \cdot \overrightarrow{\Phi(s,a)} \tag{2.11}$$

In each step, it first calculates the TD error, which is the difference δ between the observation long-term payoff and the current long-term payoff.

$$\delta = r + \gamma \cdot Q_w(s',a') - Q_w(s,a) \tag{2.12}$$

In order to minimize the TD error, we calculate the gradient descent of the mean squared error between the approximated Q function, and update w following the direction as follows.

$$\overrightarrow{w} = \overrightarrow{w} + \beta \cdot \delta \cdot \overrightarrow{\Phi(s,a)} \tag{2.13}$$

Finally, the parameters θ in policy π need to be updated as follows.

$$\vec{\theta} = \vec{\theta} + \alpha \cdot \bigtriangledown_{\vec{\theta}} log \pi_{\vec{\theta}} \cdot \delta \tag{2.14}$$

From (2.11), (2.12) and (2.13), we can see that the TD error δ affects w, w affects $Q_w(s, a)$, and $Q_w(s, a)$ affects θ in a policy. In this case, according to the trend of the TD error (positive or negative), there is a tendency to strengthen or weaken the chance of selecting the action. When the proper value of θ is achieved, J^{π} will be maximized so as to get an optimal policy.

2.5 Multi-Objective Reinforcement Learning

The approaches to MORL can be divided into two groups: the single-policy and multi-policy approaches according to the different solutions they are trying to learn. The single-policy approaches are interested in the policies that maximize the weighted long-term payoff over multiple objectives, whereas the multi-policy approaches focus on the policies that better balance different objectives.

Similar to a classical RL problem, a MORL problem is defined as a MDP, where the inputs are the 5-tuple $\{S, A, P, \vec{r}, \gamma\}$, and the outputs are a bunch of optimal policies mapping from states to actions according to different objectives. S, A, P, γ bear the same meanings as in the single-objective problem. However, in a MORL problem, the reward signal $\vec{r}$ is no longer a scalar but a m-dimensional vector, i.e.,

$$\overrightarrow{r}(s, a) = \begin{bmatrix} r_1(s, a) \\ \vdots \\ r_m(s, a) \end{bmatrix} \tag{2.15}$$

Hence the problem involves m different and potentially conflicting objectives. Each objective corresponds to a separate dimension of $\vec{r}$ in (2.15). Given a policy π and starting from state s_0, the long term payoff of π is

further defined as below.

$$\vec{J}^{\pi} = E^{\pi}[\sum_{k=0}^{\infty} \gamma^k \vec{r}_{t+k+1} | s_t = s_0] \tag{2.16}$$

In a single-objective problem, with respect to an arbitrary pair of policies π_1 and π_2, π_1 is considered better than π_2 whenever $J^{\pi_1} > J^{\pi_2}$. However, in a multi-objective problem, since the long-term payoff (2.16) becomes a vector, the Pareto dominance concept defined below can be employed as the optimality criterion for MORL. To simplify our definition, let's denote the long-term payoff of objective i for any policy π as $\vec{J}^{\pi}[i]$.

Definition 1. *Given two policies π_1 and π_2, π_1* **Pareto dominates** *π_2 (denoted as $\pi_1 \succ \pi_2$) subject to two conditions:*

1. $\forall i \in \{0, 1, \ldots, m\},\ \vec{J}^{\pi_1}[i] \geq \vec{J}^{\pi_2}[i]$;
2. $\exists k \in \{0, 1, \ldots, m\},$ *such that* $\vec{J}^{\pi_1}[k] > \vec{J}^{\pi_2}[k]$.

Based on Pareto dominance, we can further define the Pareto optimal policy, which is the outputs of the MORL algorithm. Any policy π_i is called a **Pareto optimal policy** if it is not Pareto dominated by any other policies of a MORL problem. The set of all Pareto optimal policies is jointly called the **Pareto front**.

So far, there are many techniques used in the MORL area. There are two common approaches in the reinforcement learning area when dealing with the MO problem, namely, the single-policy approach and the multi-policy approach [86]. In single-policy approaches, they usually transform a multi-objective problem to a single objective problem so as to resolve it by a single objective reinforcement learning method, e.g. a Multi-Objective Deep Reinforcement Learning Framework**??**. In contrast, instead of searching for a single optimal policy for the problem, the multi-policy approach searches for a set of optimal policies during a single run.

Single-Policy MORL Algorithms

In a MORL problem, if the preferences for the multiple objectives can be quantified, the MORL problem can be transformed into a single objective problem, where the single-policy MORL algorithms are applied to seek one optimal policy that can maximize the total reward. The main difference in single-policy approaches is the representations of preferences (different methods that assign the weights for the multiple objectives when transferring the multiple objectives to a single objective), where several approaches have been developed.

To transfer the multiple objectives to a single objective, the weighted sum approach assigns a weight for each objective [69] [41], therefore, the long-term payoff is the sum of the weighted long-term payoff for each objective. However, it has a natural drawback, as the long-term payoff is the linear weighted sum of each objective, any Pareto optimal policy not on the convex hull may not be learned by the algorithm. Thus, it cannot approximate well the Pareto front.

The W-learning approach is a single-policy MORL approach [41] following the idea above. Based on the winner-take-all methods, it ensures that the selected action is optimal for at least one objective. At each step, the Q-value $Q_i(s)$ is computed for each objective i, and it selects the action with the highest Q-value $Q_i(s)$. As the Q-value depends on the reward function, it is quite sensitive to the reward function setting.

The ranking approach is known as a sequential approach or the threshold approach [9], which aims to solve the problem by ordering multiple criteria and constraints on one or more objectives. One example in [30] optimizes one main objective while putting constraints on other objectives. In this case, it guarantees the optimization of the main objective and satisfies the constraints on other objectives. When choosing the action, it compares the actions for each objective in turn and selects the action that maximizes the main objective and satisfies the constraints on other objectives. However, the design of an appropriate ordering and the constraints

on other objectives still requires some prior knowledge of the problem domain.

The Dynamic Preferences algorithm [60] is another single-policy algorithm, which has been proposed by Natarajan and Tadepalli. The difference between the Dynamic Preferences algorithm and other single-policy algorithms [69] [41] is that the aim of the Dynamic Preferences algorithm is to learn multiple optimal policies for different preferences, whereas other single-policy algorithms only learn one optimal policy for one specific preference. The basic idea of this algorithm is to perform the single-objective algorithm for multiple runs with different weights. Finally, it learns a set of policies that are optimal for different weights respectively.

Multi-Policy MORL Algorithms

In contrast to the single-policy approaches, instead of searching for a single optimal policy of the problem, the multi-policy approaches search for a set of optimal policies during a single run. There are many multi-policy approaches that have been developed as well. Different multi-policy approaches, also have different aims.

Some multi-policy approaches learn non-Pareto policies as they are trying to learn multiple optimal policies where each optimal policy is expected to maximize the long-term payoff for one preference. Barrett and Narayanan proposed a multi-policy approach: Convex Hull Algorithm (CHVI) to MORL problems [6], which can learn optimal policies at once for all linear preferences over multiple objectives. In the single-objective RL algorithm, the algorithm is backing up the maximum expected rewards repeatedly rather the propagating the maximum expected rewards to other states immediately. Here, the CHVI can be viewed as an extension to the standard RL algorithm. When the preferences can be quantified as a weight vector, they back up a set of expected rewards that are maximal for some set of linear preferences on the convex hulls, so as to find the optimal policy for any linear preference function.

Instead of learning the optimal policies on the convex hull, it is highly desirable to develop an algorithm that can learn a set of Pareto-optimal policies (the concept of Pareto optimality was introduced in Section 2.5). One algorithm that meets this requirement and is closely related to our research is the Pareto Q-Learning (PQL) algorithm [86]. Kristof and Ann claimed in [86] that PQL can learn all Pareto optimal policies provided that every possible state-action pair is sufficiently sampled. This claim is only valid if the state transition is deterministic and the rewards of performing any action never change over time.

The PQL is a TD-based multi-policy algorithm. The long-term payoff in the MORL is a vector, called Q-vector. PQL will initialize an empty Q-set. During the run time, the Q-set is a set of non-dominated Q-vectors, which converge to a set of non-dominated policies. After the Pareto optimal policies are learned, the user can trace the Pareto optimal policies of Q-set by applying a preference function. Because it is hard to identify the correspondence during the bootstrapping process, especially when there are new non-dominated Q-vectors that appear in the next state, the key idea is to update a set of non-dominated Q-vectors and the average reward separately. The Q_{set} can be calculated by performing

$$Q_{set}(s,a) = \overline{R}(s,a) \oplus \gamma ND_t(s,a) \tag{2.17}$$

The average immediate reward vector $\overline{R}(s,a)$ is updated fairly straightforward, as shown below.

$$\overline{R}(s,a) \leftarrow \overline{R}(s,a) + (r - \overline{R}(s,a))/n(s,a) \tag{2.18}$$

The $ND_t(s,a)$ is updated using the non-dominated $\widehat{Q} - vectors$ in the next state $s^{'}$. where n is the number of times action a is taken at state s. Once state $s^{'}$ is reached, the algorithm can collect the predictions from all the actions $a^{'}$. Using them, the non-dominated Q-vectors in state $s^{'}$ can be obtained and used to update $ND_t(s,a)$. This updating method is described in (2.19).

$$ND_t(s,a) \leftarrow ND\left(\bigcup_{a'} \widehat{Q}_{set}(s^{'}, a^{'})\right) \tag{2.19}$$

where $\widehat{Q}_{set}(s', a')$ is all the $Q - vectors$ from s' and $ND()$ stands for a function that identifies all non-dominated Q-vectors from a given set of vectors.

Single-Policy vs Multi-Policy MORL Algorithms

Single-policy methods aim to learn one optimal policy based on quantified preferences over multiple objectives. For example, there is one research using XCS to solve MORL problem**??**, which considers treating one or more objectives as part of the state input, thereby leaving only a single objective as the learning target. But in practice, very often preferences over different objectives may not be easily quantified into weights before or during the learning process. Thus, the single-policy methods are not considered in this thesis. On the other hand, multi-policy methods seek to find a set of compromising policies without quantifying the preferences in advance. However, as reviewed in the literature, many existing multi-policy algorithms follow the tradition of adopting a tabular representation of the value function, which lacks generalization ability. In this case, the LCS-based algorithms are considered as a multi-policy approach to learn compact and effective solutions for solving the MORL problems in this work as they have good generalization ability.

2.6 Evolutionary Computation for RL

The basic concept of Evolutionary Computation (EC) and its strengths for RL will be introduced first. After that, popular EC techniques for RL will be reviewed, such as the approaches with rule-based solutions and neural network-based solutions. Moreover, the aim of this work is to find

the Pareto optimal policies for MORL problems. As we discussed in Section 1.2.2, EC techniques not only have huge potential to solve the deterministic MORL problem, but also the continuous and stochastic MORL problems. Therefore, the EC techniques with rule-based solutions and the neural network-based solutions will be the starting points of this MORL work.

2.6.1 What is Evolutionary Computation

Evolutionary Computation (EC) is a popular field of artificial intelligence [37] [68] not only because EC is applicable over a wide range of problem categories, including classification, regression, clustering, design, optimization, planning, and generating computer programs. Moreover, EC presents many important benefits over popular deep learning methods [68]. The EC algorithms are based on adopting Darwinian principles [22]. EC is the optimization method that simulates the process of natural selection to find the best solutions for a given problem. One important feature of all these algorithms is their population-based search strategy. Namely, all the solutions developed through generated new species via selecting, crossover, mutation, then reproducing the good species to adapt to the environment. In detail, after each round of the simulation, the idea is to delete the n worst design solutions and to breed n new ones from the best design solutions. Each solution, therefore, needs to be awarded a merit, termed fitness, to indicate how close it came to meeting the final goal, which is generated by applying the fitness function to the test the results obtained from that solution.

EC can be mainly categorized into three groups: evolutionary algorithms, swarm intelligence, and others (as shown in Figure 2.1). They have been successfully applied to many problems, such as job shop scheduling [47], computer vision, and image processing [27].

EC techniques are also used in dealing with RL problems [91]. In the

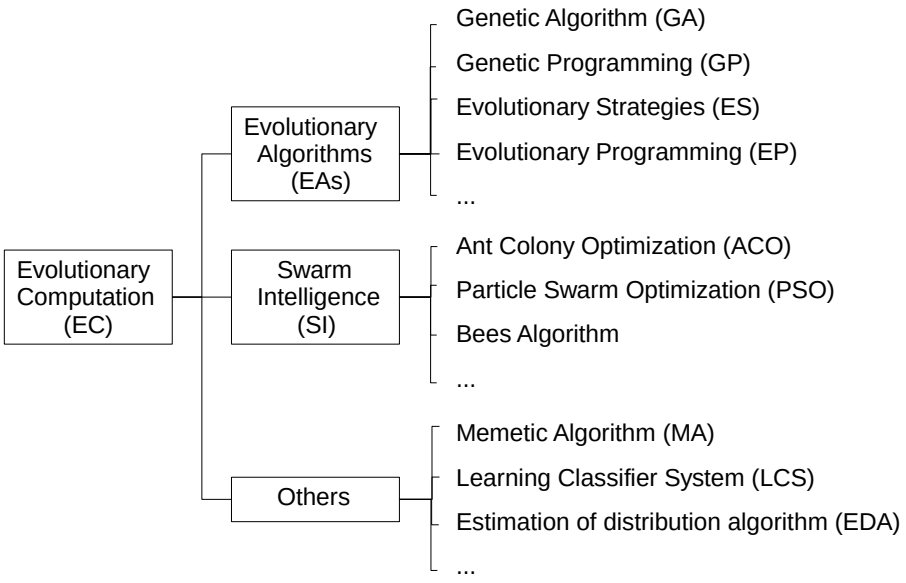


Figure 2.1: The categories of EC Algorithms.

RL tasks, the inputs are the 5-tuple $\{S, A, P, r, \gamma\}$ in MDP, and the outputs are the solution to this RL problem, namely the optimal policies mapping from states to actions. In this case, the aim of EC is to search for policies that can maximize the expected cumulative reward. As EC has been successfully applied to many RL problems [91] and with the strengths for multi-objective problems [2] we mentioned in Section 1.2.2, we consider EC as a proper method for solving MORL problems in my thesis.

2.6.2 Selection Strategy in EC

As discussed in Section 2.6.1, in EC, an initial set of candidate solutions is generated and iteratively updated. Each new generation is produced by stochastically removing less desired solutions, and introducing small random changes. In the EC process, the population of solutions is subjected to environmental selection, crossover, and mutation. As a result,

the population will gradually evolve to increase in fitness. The selection strategies will have a huge impact on the efficiency of the results of the EC Algorithms. This section focuses on the selection and discusses different search types, such as objective-based search, novelty search, and go-explore. They are suits for different search spaces and optimal solution distributions. When search pace is small, objective-based search method is best as the objective is specific; when search pace is large, and the optimal solutions are evenly distributed, the novelty is the best strategy as it will not stick with one solution; when the search space is huge but the optimal solutions are sparse, the go-explore policy is the best strategy as it will enable the system to store the most promising solution and fully explore later.

Objective-based Search (Fitness Function-based Search)

Objective-based search in EC models is driven by an objective function, such as the fitness function. It uses the fitness function to measure progress towards an objective in the search space, so drives the selection based on the measurement of the fitness function. The fitness function simply defined is a function that takes a candidate solution to the problem as input and produces as output how good the solution is with respect to the problem in consideration. For example, in an RL case, agents are rewarded according to the Euclidean distance between their final position and the goal point. Thus, a proper fitness function plays a key role in the success of this type of objective-based EC algorithm.

Go-Explore

Reinforcement learning is attempting to solve sequential-decision problems by specifying a highly informative reward function. Unfortunately, the reward can be 'sparse' when it requires long string actions to achieve the goal and 'deceptive' when the reward function leads to a dead end.

The research of Lehman [24] shows that sufficient exploration of the state space enables discovering sparse rewards and avoiding deceptive local optima. However, there are two major issues of exploration, namely detachment, and derailment. Detachment means the algorithm stops returning to certain areas of the search space, even those areas that are promising for an optimal solution. Derailment is where the exploratory mechanisms prevent the agent from returning to previously visited states, preventing exploration directly and/or minimizing exploratory mechanisms so minimal that effective exploration does not occur.

For solving the problems of detachment and derailment, The Go-Explore family of algorithms are proposed [25]. To avoid detachment, Go-Explore 'archives' different visited states in the environment, thus ensuring that states cannot be forgotten. The 'archive' process is starting from an archive containing only the initial state, then archives iteratively: 1) the algorithm selects a state from the archive probabilistically. 2) goes to that selected state. 3) explores from the selected state. 4) maps encountered states to cells 5) updates the archive with all novel encountered states. By first returning before exploring, Go-Explore avoids derailment by first returning before exploring, which can focus purely on exploration after. To maximize the performance, if a state is reached that is associated with a higher-performing solution, the better solution will replace the stored solution of that state in 'archive'.

Go-Explore solves many previously unsolved Atari games and surpasses the state-of-the-art on all hard-exploration games [25], e.g. it shows the improvements on the grand challenges of Montezuma's Revenge and Pitfall. The practical potential of Go-Explore on a sparse-reward pick-and-place robotics task is also demonstrated. This work presents the simple decomposition of remembering previously found states, returning to them, and then exploring from them appears to be especially powerful, suggesting it may be a fundamental feature of learning in general. With the advantage of Go-Explore, it is able to explore and archive all the potential

solutions and finally returns to fully explore the most promising solution.

2.6.3 Neural Network-based Solution

Neural networks [36] has provided a good way to present complex functions. This is because a neural network consists of a number of 'neurons'. The first type of neuron is called perceptron, with the weighted sum and activation function, it is able to approximate a linear function and deal with bias with a threshold. The second type of neuron is called the sigmoid neuron, with the sigmoid function, it is able to obtain the small changes in the output and making small changes in weights or bias, thus it can approximate the complex non-linear function. In this case, artificial neural networks can be used to solve RL problems as the solution, which are the learned policies to RL problems, as they often can be approximated as complex functions. Thus, Neural network-based solutios have good scalability to present the solutions for RL problems. In addition, at each stage, the difference of the expected reward with different actions is often small. With the advantages of EC, for example, it requires fewer prior assumptions regarding the problem and it is able to search the solutions in parallel, thus it can help Neural networks to adjust the parameters. In this case, the basic idea of Neuroevolution is to involve evolutionary algorithms to train artificial neural networks.

The basic steps of a neuroevolutionary algorithm work as following in Figure 2.2. In each generation, each network in the population is evaluated in the learning task. The networks with the best performance are selected. The selected networks are then bred via crossover and mutation with the new added to the population. Then the whole process is repeated during the training process.

When using the artificial neural network to solve the RL problem, the state feature usually is the input for each input node in a network, so the value of the inputs together describes the agent's current state. The value

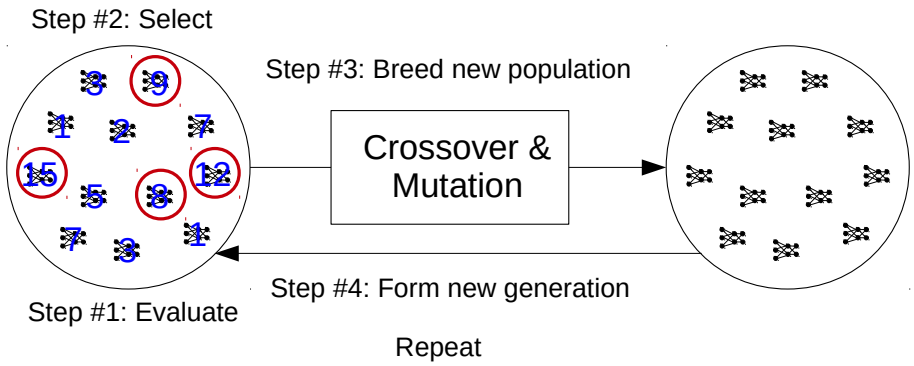


Figure 2.2: The basic steps of neuroevolution.

of the outputs could describe the action to be selected for the given state. For example, each output node denotes one action, and the action with the highest output value will be selected for the given state.

Both the weights and architectures, such as weights and nodes, can be involved in the evolution process. The evolution of weights has shown its strength in some problems like RL where gradient-based algorithms often experience great difficulties, such as gradient disappearance. The evolution of architectures enables Artificial Neural Networks (ANN)'s to adapt their topologies to different tasks without human intervention and thus provides an approach to automatic ANN design as both ANN connection weights and structures can be evolved. In this section, both evolutions of weights and architectures will be surveyed.

Though a Neural Network-based solution is powerful, especially, it supposes to perform better than other techniques when has a large amount of data, it is extremely expensive to train due to complex data models. For example, a neural network-based solution requires expensive GPUs and hundreds of machines. These increases cost to the users. More important, a Neural Network-based solution is not an explainable AI solution. Therefore, in this thesis, the neural network-based solution is not considered as

the ideal solution to MORL problems.

Evolving Weight Artificial Neural Networks

Evolution Algorithms can apply to various tasks to train Artificial Neural Networks, for example, evolving ANNs connection weights, architectures, learning rules, and input features [95].

for For the issues related to the evolution of Connection Weights, there is a great interest to find a way to automate the discovery of good representations of the neural network. Evolutionary methods provide a solution to this challenge. Especially, since neuroevolution already directly searches the space of network weights, it can also simultaneously search the space of network topologies. Methods that do so are sometimes called topology and weight evolving artificial neural networks (TWEANNs) [70]. In order to develop a successful TWEANN, three issues need to be addressed. First, the competing conventions problem in neuroevolution, which means having more than one way to express a solution with a neural network. It may happen when using the crossover process to breed new networks. When two parents represent different solutions are combined to generate an offspring, the offspring may lose some functions from both parents. It also could be two networks have the same solution but different representations. Second, the system must protect topological innovations long enough to optimize the associated weights. For instance, when a new network topology is generated, it may get a low fitness even if the structure of the topology is suitable for representing the solution. Because the network is new, there is not enough time for training the weights of the network. Third, how to keep the size of NN minimal and effective. Some TWEANNs methods may initialize a population of random topologies. If it qualifies some too complex topologies, this may lead to unnecessarily searching in the solution space, and can lead to over-fitting.

The neuroevolution of augmenting topologies (NEAT) is one of the most popular TWEANNs [70]. In this section, we briefly describe NEAT

and how it copes with the three challenges in TWEANNs mentioned above. In fact, NEAT differs from traditional neuroevolution only in the way that it breeds. To represent networks of varying topologies, NEAT employs a flexible genetic encoding. Each network is described by a list of edge genes, each of which describes an edge between two node genes. Each edge gene specifies the in-node, the out-node, and the weight of the edge. During mutation, a new structure can be introduced to a network via special mutation operators that add the new node or edge genes to the network as shown in Figures 2.3 and 2.4.

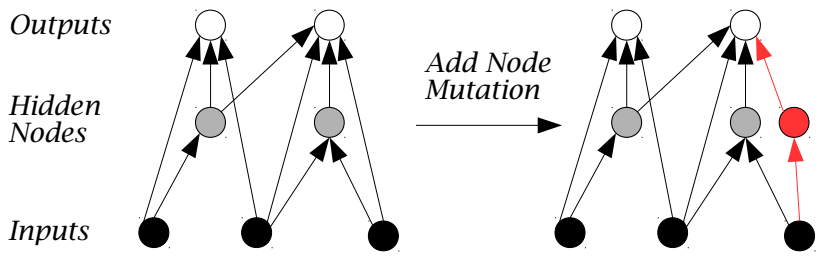


Figure 2.3: Structural mutation operators in NEAT. A new node is added by splitting an existing edge in two.

There are three main contributions of NEAT to cope with the issues in TWEANNs.

- Avoid the competing conventions problem in neuroevolution. NEAT deals with the catastrophic crossover by using an innovation number to track the historical origin of each gene. Whenever a new gene appears, it is assigned a unique innovation number and the genes having the same innovation number must represent the same structure. In this way, every gene in the system can be tracked. During crossover, innovation numbers are used to determine which genes belong to which parents. Genes that do not match are either disjoint

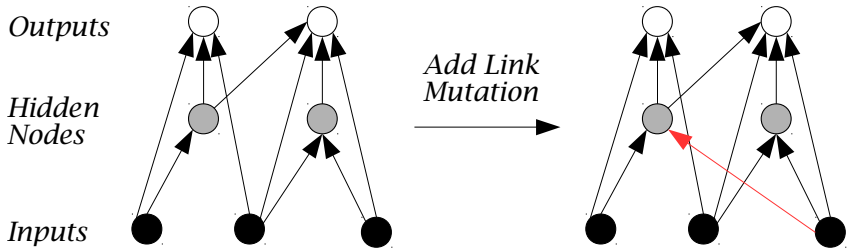


Figure 2.4: Structural mutation operators in NEAT. A new link (edge) is added between two existing nodes.

or excess, depending on whether they do not match in the middle or at the end. When crossing over, genes with the same innovation number are inherited as one gene. Genes that disjoint or excess are inherited randomly. Therefore, NEAT avoids the competing conventions problem.

- Protecting Innovation through Speciation. NEAT speciates the population-based on topological similarity by the innovation number. It measures the distance δ of each speciation of networks according to the equation as follows,

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W} \tag{2.20}$$

where E and D is the number of excess and disjoint genes, N is the population size, $\overline{W}$ is the average weight differences of matching genes, c_1, c_2, c_3 are the parameters that adjust the importances of the three factors. If the distance of networks is greater than δ_t, they are different species. Explicit fitness sharing in the same species is employed to protect innovative species.

- Minimizing solutions. TWEANNs start with a uniform population with random topologies without hidden nodes (i.e., all inputs connect directly to outputs). In this way, it starts from minimal dimensional space, and the new structure is introduced incrementally as structural mutations occur. During the evolution, the structures are evaluated by fitness, where only the ones with high fitness can survive. Since NEAT starts from a minimal structure and uses fitness to control the mutation, minimal solutions are favored.

Hybrids

Many researchers have investigated hybrid methods to combine evolution with other methods like supervised or unsupervised learning. During the learning process, the fitness of evolved individuals is changed over time by learning from the interaction with the environment. In this section, very important evolutionary function approximation methods will be investigated.

Evolutionary function approximation is a way to combine evolutionary and temporal-difference methods into a single method. It can select function approximator representations that enable efficient individual learning. The main idea behind this is to use temporal-difference methods to update the value functions during each fitness evaluation rather than just evolve the action selectors. In this way, the function approximators can be learned via temporal-difference methods. These hybrid methods have been applied to reinforcement-learning algorithms [91].

One evolutionary function approximation technique is NEAT+Q [92], which is the combination of NEAT and Q-learning with neural-network function approximation. The aim of NEAT+Q is to optimize value functions instead of action selectors, as the inputs are still the same, the change here would be the reinterpretation of its output values. As the structure of neural-network action selectors is identical to the Q-learning function approximators, the only method to amend is the updated weights of the

networks. In this case, value functions instead of action selectors will be effectively evolved. Therefore, the outputs of the network are the long-term payoff of the associated state-action pairs rather than arbitrary values. Instead of only selecting the most desirable action, it also updates the estimates of other state-action pairs. NEAT+Q combines the advantages of TD methods with those of evolution. Especially, it uses the ability of NEAT to discover effective representations of the neural network value-function approximation. Unlike traditional neural network function approximators that just adjust the weights for a fixed topology to represent the value function, NEAT+Q explores the space of weights and topology to increase the chance to find a proper representation. Therefore, on certain tasks such as Mountain Car and Server Job Scheduling, this approach has been shown to significantly outperform both TD methods and neuroevolution [92]. Because NEAT+Q not only selects function approximators for the NNs as NEAT does, it also trains the function approximators with Q-learning.

Coevolution

Coevolution in EC refers to the interactions between two individuals that are evolving simultaneously. It can be grouped into two categories, i.e. cooperative and competitive coevolution. The cooperative coevolution category includes the relationship between humans and the bacteria in our digestive systems. The competitive coevolution includes the relationship between wolves and caribou. In this section, both of those two methods in EC will be investigated.

Though Neural Network-based solutions are powerful, especially, they are supposed a better perform than other techniques in domains with a large number of features in the data, they are extremely expensive to train due to complex data models. For example, the neural network-based solution often requires expensive GPUs and hundreds of machines [73]. These increases cost to the users. More important, Neural Network-based so-

lutions are not explainable AI solutions. This is because, without the explanation of why use the optimal solution under a specific condition, the Neural Network-based solutions can 'cheat' on the reasons and make errors when applying the learned optimal solution. Therefore, in this thesis, neural network-based solutions are not considered as the ideal solution to MORL problems.

2.6.4 Rule-based Evolutionary Computation

There are many EC paradigms can generate rules. A Rule-based solution is the mapping of inputs to outputs, which can be expressed as a set of IF-THEN rules. These rules, in many cases, allow a straightforward encoding the knowledge as condition-action pairs, where the IF part of the rule specifies aspects of a condition leading to one or more actions as described in the THEN part. Rule-based solutions have good scalability because the condition of the rule is dynamic rather than fixed. Furthermore, one rule can provide solutions for multiple situations, as the condition of the rule can match multiple situations. In this section, the LCSs algorithms will be reviewed as they generate the rule-based solutions, which are explainable to the user. In this case, a user is able to know what is the solution, why is the solution, and when to trust the solution.

LCSs can be divided into two groups, namely the Pittsburgh and Michigan styles. First, the differences will be reviewed. Then, Michigan style has become more popular for RL, we will focus on this approach. Therefore, such algorithms such as XCS [15], ACS [11] and XCSG [12] will be investigated afterward.

LCSs were first invented by Holland [38] as a "cognitive system based on adaptive mechanism" [81]. As a "modeling tool", the classifier system will evolve rules online, then generate the rules that provide a model for an unknown dynamic system. In contrast, Kenneth De Jong and his students developed a classifier system as an offline optimization process

rather than the online adaptation process by Holland [46]. In this Pittsburgh classifier system, the genetic algorithm is applied to a population of individuals, which represents a complete set of rules. A performance measure happens at each cycle, then a genetic algorithm is used to guide the exploration of the solution space. These two contrasting perspectives on classifier systems are online adaptive systems and offline optimization systems form two different types of LCSs.

The models of LCSs that have been inspired by the work of Holland [38] at the University of Michigan are usually called Michigan Classifier Systems, whereas the ones that have been inspired by the work of Smith and De Jong at the University of Pittsburgh are usually termed Pittsburgh Classifier Systems. There are several differences between the Michigan and Pittsburgh classifier systems in an individual structure, problem-solution structure, individual competition/cooperation, and Online vs. Offline learning.

For the individual structure, each individual in Michigan approaches is one classifier, whereas each individual in Pittsburgh is a set of classifiers. In the Pittsburgh approaches, the individuals are competing with each other for reproductive opportunities. In the Michigan approach, the classifiers cooperate with each other to solve the problem, while competing with each other for reproductive opportunities. Therefore, the solution from the Pittsburgh system is the best set of classifiers, whereas the solution from Michigan approaches is provided by the whole population.

The Michigan approaches follow an online learning strategy, whereas the Pittsburgh approaches follow an offline learning strategy. This is because, during the training process, the population is continuously evaluated and evolved by GA in Michigan approaches. In Pittsburgh, the evaluation does not happen until the next generation of populations is generated, as it needs to calculate the fitness of the whole population. A consequence is that the different competition mechanisms in Michigan and Pittsburgh classifier systems produce different final solutions. Michigan

evolves a large number of rules, Pittsburgh only evolves a few rule sets in a population. Therefore, Pittsburgh systems are more suitable when a compact solution with few rules works, while the Michigan systems are suitable for distributed solutions.

Comparing Pittsburgh with Michigan systems, we will consider Michigan systems as the potential solution. As the Pittsburgh system evolves multiple rule sets simultaneously, it suffers from heavy computational requirements. In this case, the Michigan systems can be applied to the larger, more complex tasks [82]. Moreover, we will focus on XCS [15]. Because, among all learning classifier systems, XCS has been most popularly used for tackling reinforcement learning problems as it is able to general the general and accurate solution[53].

XCS

XCS is widely demonstrated in the literature as an effective algorithm for single-objective RL [53]. This algorithm contains several important learning components, such as the *performance component*, *reinforcement component* and *discovery component* [15]. As an evolutionary algorithm for machine learning, XCS maintains and evolves a population of classifiers. Each classifier cl consists of a condition $cl.c$ that matches any given state input, an action $cl.a$ to be performed in matched states, and the prediction $cl.P$ that estimates the long-term payoff of the following classifier cl.

At any time t, upon reaching a new state s_t, the *performance component* will be utilized first to determine an action a_t to be performed in s_t. For this purpose, a *match set* $[M]_t$ is formed to include all classifiers whose conditions match the current state input s_t. If classifiers in $[M]_t$ do not cover all possible actions that a learning agent can perform, a *covering* mechanism will be activated to generate new classifiers to be further included in $[M]_t$. Based on $[M]_t$, the estimated payoff of performing every action $a \in A$, i.e. $PA(s_t, a)$, will be calculated as the fitness-weighted average of the predictions of all classifiers in $[M]_t$ advocating the same action a

[15]. Subsequently, an action a_t is selected and performed by using the ϵ-greedy strategy [15]. The environment then provides its reward $r(s_t, a_t)$ which will be further used in the reinforcement component to update all classifiers in the *action set* $[A]_t$. $[A]_t$ stands for the set of classifiers in $[M]_t$ that advocate the same action a_t.

In the *reinforcement component*, several parameters of each classifier $cl \in [A]_t$ will be updated. The details for updating other parameters except the prediction and error can be found in [15]. We would like to highlight particularly the updating rules for the prediction $cl.P$ and the error $cl.\epsilon$ respectively in (2.21) and (2.22).

$$cl.P \leftarrow cl.P + \beta \cdot \Big(r(s_t, a_t) + \gamma \max_{a \in A} P_t(a) - cl.P \Big) \tag{2.21}$$

$$cl.\varepsilon \leftarrow cl.\varepsilon + \beta \cdot \Big(\Big| r(s_t, a_t) + \gamma \max_{a \in A} P_t(a) - cl.P \Big| - cl.\varepsilon \Big) \tag{2.22}$$

The β is the learning rate, and the γ is the discount rate. Clearly, with the prediction of each classifier updated as a scalar, XCS is designed to solve merely single-objective problems. In the preliminary work, some important changes will be introduced in our algorithms to enable XCS to simultaneously pursue multiple objectives.

ACS and ACS2

ACS is another well-known Michigan system. In the partially observed Markov environments, some of the distinct states of the environment will be considered identical. In this case, the LCS may not be able to decide the best action in those situations. Therefore, to deal with this issue, Stolzmann presented Anticipatory Classifier System (ACS) [11], then Martin presented Anticipatory Classifier System2 (ACS2) [10].

ACS consists of three essential components: input interface, output interface, and classifier list. Input interface acts as a sensor for the system;

Output interface is like an actuator for the system; Classifier list is learned during the training and represented by a population of rules. The structure of a classifier includes Condition C is used to match the current state; Action A denotes the action to execute if this classifier is selected; Effect E is the anticipation of the effect of action; Mark M indicates unsuccessful situations; Quality q measures quality of anticipations and Reward prediction r is the prediction of payoff.

Most of the Anticipatory Learning Process (ALP) in ACS works as the same as XCS, such as the action selection strategy, classifier deletion. At any time step t, it receives a string $\sigma(t)$ matching the actual situation perceived by the environment, and a payoff r. The system selects an action a at time t and executes it in the environment to reach the current state. After that, it will compare the expectation state with the current state. If a classifier anticipates correctly, there is no new rule generated; otherwise, the ALP generates a new classifier that specifies all changing attributes. In the latter case, the original classifier is marked by the current situation, and its quality q decreases. Suppose a classifier anticipates correctly, the quality q of the classifier increases. If the classifier has a wild card, then the ALP generates a new classifier with a more specific condition. In the condition part of the new classifier, the attributes are specialized where the current situation differs from situations in which the classifier did not anticipate correctly.

ACS2 is quite similar to ACS, and the significant difference between them is ACS2 has the genetic generalization process, but ACS does not have the genetic generalization process.

In this case, ACS will be considered in future work to solve PO-MDP problems, especially the large-scale PO-MDP. When the good classifier is deleted in a potential optimal policy, especially at the early state in that policy, it is hard for the system to learn it back.

XCSG

As the applications of XCS in multi-step problems were restricted to very small problems [50], Butz presented an algorithm, which is XCS with the prediction updated based on gradient descent as XCSG in 2005 [12]. In Q-learning, the value of state-action pair is updated according to the equation 2.5. In Q-learning with a function approximator parametrized by a weight matrix W, when using gradient descent, each weight w changes by Δw at each time step t

$$\Delta w = \beta(r + \gamma \max_{a \in A} Q(s_t, a) - Q(s_{t-1}, a_{t-1})) \frac{\partial Q(s_{t-1}, a_{t-1})}{\partial w} \tag{2.23}$$

The key idea in XCSG is to add gradient descent to XCS. As the prediction $P(a_{t-1})$, which is corresponding to $Q(s_{t-1}, a_{t-1})$ can be computed as follows:

$$Q(s_{t-1}, a_{t-1}) = P(a_{t-1}) = \frac{\sum_{cl_j \in [A]_{-1}} p_j \times F_j}{\sum_{cl_j \in [A]_{-1}} F_j} \tag{2.24}$$

where p_j and F_j are, respectively, the prediction and the fitness of classifier cl_j.

When the parameters of classifiers in $[A]_{-1}$ are updated, the sum $F_{[A]_{-1}}$ of classifiers' fitness in $[A]_{-1}$ is computed as

$$F_{[A]_{-1}} = \sum_{cl_j \in [A]_{-1}} F_j \tag{2.25}$$

Then, for each classifiers $cl_j \in [A]_{-1}$, the prediction p_k is updated as

$$p_k \leftarrow p_k + \beta(r + \gamma \max_{a \in A} P(a) - p_k) \frac{F_k}{F_{[A]_{-1}}} \tag{2.26}$$

Though XCSG is able to approximate a complex function to represent the learned solution, however, the gradient descent method may encounter the gradient descent disappearance problem. Thus, XCSG is the optimal method to solve the problems in this thesis.

ZCS

As Zeroth-level Classifier System (ZCS)[94] is quite similar to its successor XCS, the following is a high-level description of ZCS, focusing on the difference between XCS and ZCS and components necessary for our research.

Same with XCS, a ZCS follows a cycle of performance component, reinforcement component, and discovery component [94]. It employs a population of classifiers, each encoding the environment of the problem domain, and thus, finally collectively forming the overall solution to the target problem. Same again, at each time step t, ZCS is able to receive a binary encoded input s_t, determines an appropriate action a_t based on the match set $[M]_t$. After the agent takes a_t, the reinforcement component receives a Reward $r(s_t, a_t)$, which will be used to update the classifiers in action set $[A]_t$.

However, there are differences between XCS and ZCS. First, in XCS, $PA(s_t, a)$, which is the estimated payoff of performing every action $a \in A$, will be calculated as the fitness-weighted average of the predictions of all classifiers in $[M]_t$ advocating the same action a [15]. In ZCS, $PA(s_t, a)$ will be calculated as the average of the predictions of all classifiers in $[M]_t$ advocating the same action a [15]. Second, in the reinforcement component, it only needs to update the classifier's prediction with the equation in (2.21).

We can see that ZCS is also designed to solve merely single-objective problems. Again, in our work, some important changes will be introduced in our algorithm to enable ZCS to simultaneously pursue multiple objectives.

UCS

LCS is able to adapt to supervised learning. sUpervised Classifier System (UCS) is a type of LCSs designed for supervised learning problems [43].

As described in section 2.6.4, the reinforcement process updates the XCS fitness based on the mapping of states and actions to rewards. The difference between UCS with XCS is that UCS's fitness is calculated from a supervised learning perspective. However, as supervising is hard to involve in multi-step problems, UCS can be only applied to single-step classification tasks. In this case, to address multi-step problems in thesis, XCS will be a better solution than UCS.

NXCS

LCSs are rule-based machine learning technologies designed to learn optimal decision-making policies in the form of a compact set of maximally general and accurate rules. Most of the existing LCSs focused on learning deterministic policies in a Markov environment. However, when the environment is partially observable, a desirable policy may often be stochastic, which leads to Stochastic Decision Making.

To fill this gap, a new XCS-based algorithm Natural XCS (NXCS) was developed [3]. NXCS and stochastic decision making in LCS to enable learning of stochastic policies by utilizing a natural gradient learning technology under a policy gradient framework. The results show that NXCS can achieve competitive performance in both deterministic and stochastic multi-step problems. NXCS added stochastic decision making into XCS, thus it is able to solve the PO-MDP problem with relatively high accuracy. In this case, NXCS will be considered to be used to solve the PO-MDP problem in future work.

Other LCS-based algorithms

There are many learning classifier systems-based algorithms developed [21] [13] [54], which will be reviewed as well as they are related to our future work, especially the Neural-LCS [21]. One of the issues of XCS is that it may require a large number of rules to cover the input space.

In order to solve this issue, Dam developed NLCS [21], which combined Artificial neural networks with classifiers and used back-propagation to train the neural network, in this case, to obtain more compact and also accurate rules.

Butz proposed Function Approximation with XCS (XCSF) [13], in which conditions are extended for numerical inputs, classifiers are extended with a vector of weights $\overrightarrow{w}$ to compute prediction. XCSF just updates the weights rather than the prediction. With the improvements, population size drops dramatically. Lanzi [54] proposed XCSμ, which is an extension to the XCS for stochastic environments. In order to distinguish the inaccuracy caused by over-general classifiers and uncertainty in the environment, it added a new parameter to estimate the degree of uncertainty that the classifier experiences so as to deal with the uncertainty in the stochastic environment.

2.7 EC for Multi-Objective Reinforcement Learning

2.7.1 Multi-Objective Algorithm

Many real-world optimization problems involve multiple conflicting problems are called multiobjective optimization problems (MOPs). As the objectives are conflicting with each other, when optimizing one of them may lead to the deterioration of another. In this case, a set of Pareto optimal solutions are desired. The Evolutionary algorithms (EAs), with their population-based nature, are able to search for the Pareto set or the Pareto front in a single run, therefore, they have huge potential to solve MOP, and these EAs are called multiobjective evolutionary algorithms (MOEAs) [101]. In this section, several popular MOEAs, such as NSGA-II [52], SPEA [44], SPEA2 [100] will be surveyed, which will help for the non-dominated individuals' selection process in future work.

MOEA/D

A Multi-Objective Evolutionary Algorithm based on decomposition is presented to solve the MOP problem (MOEA/D) [96]. Decomposition is a basic strategy in traditional multi-objective optimization. It decomposes a multi-objective optimization problem into a number of evenly distributed subproblems, then optimizes them simultaneously.

There are several approaches for converting the problem of approximation of the PF into a number of scalar optimization problems. In this thesis, the weighted sum approach in MOEA/D method will be integrated with XCS to solve the MORL problem. In this approach, a weight vector $\lambda = (\lambda_1, ..., \lambda_m)$ is evenly initialized, where $\lambda_i > 0$ for all $i = 1, ...m$ and $\sum_{i=1}^{m} \lambda_i = 1$. A neighborhood of weight for $\lambda_i > 0$ for all $i = 1, ...n$, where $n < m$. An initial population $x^1, ..., x^t$ is generated as well. An ideal solution f^* is initialized as well. Then the optimal solution to the following scalar optimization problem is maximize $g^{ws}(x|\lambda) = \sum_{i=1}^{m} \lambda_i f_i(x)$. Then the general framework for seeking for seeking the optimal solution is as follows. For each λ_i, it will randomly select two optimal solutions from the neighborhood and generate a new solution x_t, $f_i(x)$ will replace f^* only when $f_i(x)$ is better than the best solution f^*. For the neighbourhood solution, x_i will pass to its neighbourhood, only when $f_i(x)$ is better than their neighbourhood's solutions.

NSGA-II

Multiobjective evolutionary algorithms (EAs) that use non-dominated sorting and sharing have been criticized mainly for three main reasons: 1). High computational complexity (O(MN3)) of non-dominated sorting, where M is the number of objectives and N is the population size. 2). Lack of elitism can speed up the performance of algorithms, and prevent the loss of good solutions once they are found. 3). Need for specifying the sharing parameter to protect the diversity in the population. Deb presented a

nondominated sorting-based multi-objective EA (MOEA) [52], called non-dominated sorting genetic algorithm II (NSGA-II), which alleviates all the above three difficulties. The computational complexity is decreased to O(MN2). Also, a selection operator is presented that creates a mating pool by combining the parent and offspring populations and selecting the best (with respect to fitness and spread) solutions. The non-dominated sorting technique in NSGA-II is highly related to the work of this thesis, as we will consider it as a potential strategy to select the non-dominated solutions in an algorithm.

SPEA

Strength Pareto Evolutionary Algorithm (SPEA) [44] is similar to other MOEAs in the following ways. First, it stores the Pareto-optimal solutions found so far externally (archiving). Second, it uses Pareto dominance to assign fitness values to individuals. Third, it performs clustering to reduce the number of nondominated solutions stored without destroying the characteristics of the Pareto optimal front. SPEA has some unique aspects as well. The fitness of an individual is determined from the archive of nondominated solutions; whether members of the population dominate each other or not is irrelevant. All solutions in the archive participate in the selection. A niching method is used that does not require any fitness sharing parameters. SPEA has been a very successful algorithm and has resulted in numerous improved algorithms, such as SPEA2 [100], SPEA2+ [48].

SPEA2

SPEA2 [100] differs from SPEA and NSGA-II only in how it does fitness assignment and selection (this is a common theme among the most popular MOEAs.) The improvements of SPEA2 over SPEA are as following. First, an improved fitness assignment scheme is used, which takes into ac-

count, for each individual, how many individuals it dominates and it is dominated by. Second, the nearest neighbor density estimation technique is incorporated, which allows more precise guidance of the search process (increased diversity). Third, a new archive truncation method guarantees the preservation of boundary solutions. The diversity issue exists for solving the MORL problem. Therefore, the method nearest neighbor density estimation technique in SPEA2 can be a potential method for solving the diversity issue.

Pareto Evolutionary Neural Networks

There are some algorithms that using Evolutionary Neural Networks to learn the Pareto front [28] [4] [19]. Those algorithms construct a methodology for implementing multi-objective optimization within the evolutionary neural network (ENN) domain. Fieldsend [28] developed a method called the Pareto evolutionary neural network (Pareto-ENN). The Pareto-ENN evolves a population of models that may be heterogeneous in their topologies inputs and degree of connectivity and maintains a set of the Pareto optimal ENNs that it discovers. Finally, experimental evidence is presented in this study demonstrating the general application potential of the framework by generating populations of ENNs for forecasting 37 different international stock indices. Abbass [4] presents an Evolutionary Artificial Neural Networks (EANN) approach based on Pareto multi-objective optimization and differential evolution augmented with local search, which is called the approach Memetic Pareto Artificial Neural Networks (MPANN). He shows that MPANN is capable of overcoming the slow training of traditional EANN with equivalent or better generalization.

Though those Pareto-ENN algorithms have shown evidence that the ENN algorithms can learn the Pareto front effectively. However, there are still some shortages that need to be addressed. First, the diversity issues on the Pareto front. Abbass tried to solve this issue by setting a threshold

of the number of the non-dominated NNs (3 non-dominated NNs in his work) [4]. However, how to set the threshold setting needs prior knowledge, and it cannot guarantee diversity on the Pareto front (e.g. when those non-dominated NNs are crowded). Second, the Pareto front convergence efficiency issue. The ENN method used mutation to generate new individuals and focuses on how to select the non-dominated NNs but it is not effective in training the non-dominated NNs.

2.7.2 EC for MORL algorithms

EC has been applied in many areas related to MORL, such as the multi-objective control [99] and multi-objective job shop scheduling problems [61]. Haigen applied the EA on the Multi-Objective Control Optimization for Greenhouse Environment [40]. In this research, they investigated using NSGA-II to tune the PID controller parameters in the greenhouse climate control system. Multi-Objective is needed to balance two objectives: good static-dynamic performance specifications and the smooth process of control. The results show that by tuning the parameters the controllers can achieve good control performance. It can be applied to complex control systems, which have strong interactions among variables, nonlinear, and conflicting performance criteria. Su investigated a multi-objective genetic programming-based hyperheuristic (MO-GPHH) method on handling multiple conflicting objectives in dynamic job shop scheduling problem [61]. MO-GPHH is used to evolve and seek a Pareto Front of non-dominated dispatching rules that help the decision-maker to trade off the different objectives. The experimental results suggest that the evolved Pareto front contains a very wide range of effective and robust rules. From the literature, we can see the huge potential of EC to deal with the MORL problem.

Besides that, a search of the literature shows that only Studley and Bull have attempted to solve MORL problems by using XCS [75]. Their re-

search considers those problems where one or more objectives can be conveniently treated as part of the state input, thereby leaving only a single objective as the learning target. In this work, we will consider a different type of problem, where no objectives can be covered through extended state input. Therefore, rather than learning a single optimal policy, our goal is to learn group policies that jointly form the Pareto front.

2.8 Chapter Summary

This chapter reviewed the main concepts of Reinforcement Learning and Multi-Objective Reinforcement Learning. The traditional MORL techniques and EC techniques for solving MORL were also reviewed in this chapter. The limitations of the existing work that form the motivations of this research were also discussed, which can be summarized as follows.

1. Existing approaches for MORL problems can be grouped into two branches: single-policy and multi-policy approaches. The single-policy needs prior knowledge for the user's preference to transform a multi-objective problem to a single objective problem for solving it with a single objective reinforcement learning method. In contrast, the multi-policy approaches search a set of Pareto optimal policies. However, limited multi-policy approaches have been investigated. In addition, to our best knowledge, the solution of the existing multi-policy approaches have a tabular-based representation, which lacks scalability. Therefore, a MORL solution with good scalability for learning the Pareto optimal policies is required.

2. Limited approaches have been developed for solving continuous and stochastic MORL problems. Therefore, research needs to be conducted to propose new solutions for learning the Pareto optimal policies for continuous and stochastic MORL algorithms.

3. As discussed in the literature, XCS can generate rule-based solutions with good scalability and are easier to understand than tabular-based solutions.

4. There are some algorithms solving the MORL problems with Pareto non-dominated NNs, thus providing a more compact and flexible solution. NEAT has a good performance on the single-objective RL problem. However, if using NEAT to solve the MORL problem, there are some issues like the diversity on the Pareto front, the efficiency for finding the Pareto front. Our research will not focus on coping with those issues above.

Chapter 3

MO-XCS: Adding Pareto Dominance to XCS to Address Multi-Objective Reinforcement Learning Problems

3.1 Introduction

Existing studies of LCSs for Multi-Objective Reinforcement Learning (MORL) are very limited, see Subsection 2.7.2, page 63. In particular, no research has been proposed using XCS or other LCSs to learn multiple non-dominated policies that together define the Pareto front of a MORL problem since XCS is not able to maintain multiple predictions for different objectives. However, there are full of MORL problems in practice. Given this lack of support for MORL in the LCS community, the goal of developing a new XCS based algorithm for MORL is set in this chapter.

3.1.1 Chapter Goals

Since XCS was originally designed for single objective learning, it is not a trivial task to enable it to learn multiple objectives. Four major technical issues have been identified and addressed to achieve this goal in this chapter as follows.

(1) The classifier representation in XCS is extended to maintain all non-dominated long-term payoffs of the policy of the affected classifiers.

(2) A new mechanism for controlling errors in general classifiers has been proposed as well.

(3) The efficacy of two different methods for selecting promising actions during learning is studied, where the action's PA has the largest number of non-dominated Q-vectors and has the largest hypervolume.

(4) A separate process is developed for building the Pareto front once learning is completed. In association with these technical improvements, the new algorithm is experimentally evaluated on several benchmark maze problems.

3.1.2 Chapter Organisation

The rest of this chapter is organized as follows. The new MORL algorithm MO-XCS is proposed in Section 3.2, and the experiment for evaluating this algorithm is designed in Section 3.3. The algorithm is experimentally evaluated, where the results are discussed in Section 3.4. Finally, Section 3.5 concludes this chapter and highlights some potential future research directions.

3.2 Methodology

MO-XCS is designed to solve MORL problems by learning the Pareto front through a single learning process. This goal is achieved by addressing four important issues in XCS. These issues, together with their solutions, will be discussed in Subsections 3.2.1, 3.2.2, and 3.2.3.

3.2.1 Multi-Dimensional Reward Handling in XCS

Several modifications of XCS are necessary to handle multiple learning objectives (i.e. multi-dimensional rewards). Specifically, a new formula should be developed to facilitate the accurate update of the prediction of each classifier. Different from XCS, the classifier prediction in MO-XCS is represented as a set of non-dominated Q-vectors (or long-term payoff vectors) which will be further used as the basis to derive the Pareto front. When a classifier's prediction becomes a set of vectors, the way of updating the classifier's error must also be revised. To cope with these issues, new learning techniques will be developed in this subsection based on Pareto Dominance.

Prediction Calculation in MORL

In a MORL problem, the reward $\vec{r}(s_t, a_t)$ of performing action a_t at state s_t at any time t is a m-dimensional vector. Based on the concept of Pareto dominance and the Pareto Q-Learning (PQL) algorithm, see Section 2.5, page 37, the prediction of a classifier in $[A]_t$ should be updated as a set of non-dominated Q-vectors. Each Q-vector represents the current estimate of the expected long-term payoff of following a Pareto optimal policy supported by the classifier. Because the prediction becomes a set, it is difficult to update every individual in the set by using the standard updating rule of the Q-Learning algorithm. This is because Q-Learning updates Q-value based on the maximum observed Q-value but Q-vector needs to be up-

dated based on the Pareto dominance rule. Instead, inspired by PQL, the prediction of a classifier cl is replaced by another two parameters, i.e. (1) the *reward* $cl.\vec{R}$ that measures the average immediate reward obtainable by performing action $cl.a$ and (2) the *non-dominated next-step payoff* $cl.ND$ gives the non-dominated Q-vectors of the next state after performing $cl.a$. By using the two, the prediction of classifier cl can be determined instantly as

$$cl.P = cl.\vec{R} \oplus \gamma \times cl.ND \tag{3.1}$$

Here, $\oplus$ refers to an operation that adds vector $cl.\vec{R}$ to every non-dominated Q-vector in $cl.ND$ and γ refers to the discount rate in the learning process. The result is another set of non-dominated Q-vectors. Because of (3.1), the prediction $cl.P$ does not need to be explicitly maintained in classifier cl.

At time t, for any classifier $cl \in [A]_t$, the updating rule for $cl.R$ is straightforward, as shown below.

$$cl.\vec{R} \leftarrow cl.\vec{R} + (\vec{r}(s_t, a_t) - cl.\vec{R})/n \tag{3.2}$$

where n is the number of times for classifier cl to appear in any action set. Once the state s_{t+1} is reached, the predictions from all *high-fitness classifiers* in the match set $[M]_{t+1}$ could be collected. Using them, the non-dominated Q-vectors in state s_{t+1} can be obtained and used to update $cl.ND$. This updating method is described in (3.3) below.

$$cl.ND \leftarrow ND\left(\bigcup_{cl' \in [M]_{t+1}} cl'.P\right) \tag{3.3}$$

where $cl'.P$ is determined from (3.1), and $ND()$ stands for a function that identifies all non-dominated Q-vectors from a given set of vectors, see in Section 2.5, page 37. The two conditions presented in Definition 1 explain the dominance relationship between any two Q-vectors. The rules for updating $cl.\vec{R}$ and $cl.ND$, i.e. (3.2) and (3.3), are similar to those used in PQL. However, different from PQL, changes have been introduced in MO-XCS to tackle the subtleties involved in learning classifiers (not present in a

tabular representation used by PQL). Specifically, $\bigcup$ in (3.3) applies only to high-fitness classifiers in $[M]_{t+1}$. A classifier is considered of high fitness if its fitness is higher than all other classifiers in $[M]_{t+1}$ that advocate the same action.

Classifier Error Estimation in MO-XCS

For any classifier $cl \in [A]_t$, its error $cl.\varepsilon$ measures the average distance between the estimated prediction $cl.P$ (Ref. (3.1)) and the *observed prediction* (OP) calculated from (3.4) below

$$cl.OP = \vec{r}(s_t, a_t) \oplus \gamma \times ND\left(\bigcup_{cl' \in [M]_{t+1}} cl'.\vec{P}\right) \tag{3.4}$$

Since $cl.P$ and $cl.OP$ are two sets of vectors, the distance between them is the distance between two sets. Several mathematical methods can be utilized to calculate this distance [59]. In consideration of learning efficiency, we will specifically consider four efficient and frequently used distance measures, i.e. the single-link (SL) measure [59], the complete-link (CL) measure [59], the core distance (CO) [59], and the Jaccard distance (JA) [59]. Given that the distance between $cl.P$ and $cl.OP$ is measured as $\|cl.P, cl.OP\|$, the error of classifier cl will be updated according to the rule below.

$$cl.\varepsilon \leftarrow cl.\varepsilon + \beta \cdot (\|cl.P, cl.OP\| - cl.\varepsilon) \tag{3.5}$$

It is interesting to note that the updating rule in (3.5) is new to the existing LCSs methods. PQL does not use (3.5) since it operates directly on value functions represented in tabular form.

Error Control in General Classifiers

As an accuracy-based LCS, XCS aims at learning both general and accurate classifiers. However, whenever the accuracy of a general classifier

becomes too poor, the accuracy of other classifiers might be negatively affected. This may subsequently lead to sudden degradation of learning performance. Therefore, the inherent error of each evolved classifier in MO-XCS has to be properly controlled.

To demonstrate this issue in further detail, let us consider a maze problem in Figure 3.1. As depicted in this figure, there are 9 open locations in the maze. Each location is indicated as a separate state (i.e. $s_1, \ldots, s_9$) in the learning environment. Besides that, the maze also contains two final locations/states, denoted respectively as F_1 and F_2. Starting from an arbitrary open location, a learning agent must eventually move to one final state to obtain food. For this purpose, in any state s_t at time t, the agent can perform one of four alternative actions: i.e. $North$, $East$, $South$, and $West$. By doing so, it can move to an adjacent location in the corresponding direction whenever possible. Its location may also remain unchanged if its way is blocked by an obstacle indicated as T in Figure 3.1.

T	T	T	T	T	T	T
T	s_1	s_2	T	s_3	F_1	T
T	s_4	s_5	T	s_6	T	T
T	F_2	s_7	s_8	s_9	T	T
T	T	T	T	T	T	T

Figure 3.1: An example bi-objective maze problem.

Different from many single-objective benchmark maze problems studied previously [57], the agent has *two objectives* to pursue in our maze. The first objective is to reduce the total number of actions to be performed to reach a final state. The second objective is to get as much food as possible. In association with the two objectives, there are three different rewards that the agent can receive after performing an action at any time t:

(1) $\vec{r}_t = \{-1, 0\}$ if the agent does not reach any final states at time $t+1$, i.e. $s_{t+1} \neq F_1$ or $s_{t+1} \neq F_2$

(2) $\vec{r}_t = \{-1, 1\}$ if $s_{t+1} = F_1$

(3) $\vec{r}_t = \{-1, 10\}$ if $s_{t+1} = F_2$

The agent can get more food (i.e. 10) when it finds the final state F_2. However, this might require it to perform more actions (depending on its current location). Since every action produces -1 in the first dimension of the corresponding reward, an agent must obtain a good balance between the two objectives by learning the Pareto front.

Suppose that MO-XCS learned a classifier cl_1 whose condition $cl_1.c$ matches both states s_1 and s_3 (notice that an agent has identical local observations in the two states). Moreover, $cl_1.a = East$. When the agent follows the classifier cl_1 and performs the action $West$ in state s_3, it will immediately reach final state F_1. Consequently, according to (3.3) and (3.1), the prediction $cl_1.P$ of cl_1 will become a set that contains only a single non-dominated Q-vector, i.e. $\{-1, 1\}$. Clearly, the classifier cl_1 has poor accuracy after this update since its prediction completely ignores another non-dominated Q-vector, i.e. $\{-3, 10\}$. As can be verified from Figure 3.1, this missing Q-vector gives the long-term payoff (assuming that $\gamma = 1$) obtainable by the agent if it follows classifier cl_1 in state s_1. Such inaccuracy may prevent us from learning some Pareto optimal policies. Furthermore, assume that there is another learned classifier cl_2 whose condition $cl_2.c$ matches state s_2. When the agent follows cl_2 in s_2 and performs action $cl_2.a = West$, it will then reach state s_1 matched by cl_1. If cl_1 is a high-fitness classifier in state s_1, the prediction $cl_2.P$ of cl_2 will start to include Q-vector $\{-2, 1\}$ (see (3.3) and (3.1)). Apparently, the agent can never obtain the long-term payoff of $\{-2, 1\}$ by following cl_2 in state s_2. Classifier cl_2 is hence updated wrongly and loses its accuracy. This error can propagate quickly from one classifier to another, resulting in a significant drop in learning performance.

Based on the example described above, it was found that the real cause of the problem is because the parameter $cl.ND$ of any classifier cl is not incrementally updated. Instead, in (3.3) it is immediately replaced by the prediction of the next state, which is possibly matched by low-accuracy classifiers. Errors can easily transfer between classifiers in this way. To solve this problem, the parameter $cl.ND$ of every classifier cl in MO-XCS is extended to a more complicated *history* parameter, i.e. $cl.H$. The history $cl.H$ keeps track of all the observed payoffs $cl.ND$ in the past and continuously measures their accuracy. At any time, the most accurate $cl.ND$ in the history $cl.H$ will be utilized to determine the classifier's prediction $cl.P$ in (3.1). By doing so, it can prevent any inaccurate $cl.ND$ completely replacing an existing $cl.ND$ which might be more accurate. In other words, any accurate $cl.ND$ in classifier cl can be eventually recovered from the history $cl.H$.

In line with the maze problem in Figure 3.1, suppose that classifier cl_2 accepts Q-vector $\{-2, 1\}$ in its prediction, i.e. Q-vector $\{-1, 1\}$ has been recorded in the history $cl_2.H$. Initially, this Q-vector $\{-1, 1\}$ has an average error of 0. However, it will not replace any existing Q-vectors in history. Instead, classifier cl_2 will continue to measure its accuracy whenever cl_2 is used again by the learning agent to perform an action. Since the agent will never experience $\{-1, 1\}$ as its real next-step payoff at state s_2, over time the average error of the Q-vector $\{-1, 1\}$ will increase. Eventually, classifier cl_2 will learn to recover from its error by not using this Q-vector to determine its prediction $cl_2.P$. Experiments, to be reported in Section 3.3, show that this error control mechanism allows classifiers in MO-XCS to learn their accurate predictions. This method helps the subsequent construction of the Pareto front, as described in Subsection 3.2.3.

3.2.2 Exploitation and Exploration

Similar to XCS, the ε-greedy strategy will be employed in MO-XCS for action selection during learning, to achieve a good balance between exploration and exploitation. Particularly, at any time t, with a probability of $\varepsilon, 0 \leq \varepsilon \leq 1$, a randomly selected action a_t will be performed in state s_t for exploring the learning environment. On the other hand, for a proportion $1 - \varepsilon$ of the time, the *most promising action* will be selected for exploitation. In XCS, such an action a_t is associated with the highest estimated long-term payoff, i.e. $PA(s_t, a_t)$. In MO-XCS, because the prediction of any classifier is a set of non-dominated Q-vectors, the long-term payoff of performing any action a also becomes a set of non-dominated Q-vectors. Specifically, suppose $[A]_t$ contains all classifiers that match state input s_t at time t and advocate the same action a, $\mathbf{PA}(s_t, a)$ in MO-XCS is determined as

$$\mathbf{PA}(s_t, a) = cl.P, \quad \text{where } cl \in [A]_t \text{ and } \forall cl' \in [A]_t, cl.\varepsilon \leq cl'.\varepsilon \tag{3.6}$$

Given (3.2) and (3.7), it is clear to see that the ways of identifying the most promising action in MO-XCS are not unique. In this chapter, two commonly used approaches [86] are explicitly studied, as briefly summarized below.

A1 The action a whose $\mathbf{PA}(s_t, a)$ has the largest number of non-dominated Q-vectors is the most promising action.

A2 The action a whose $\mathbf{PA}(s_t, a)$ has the largest *hypervolume*, i.e. $HV(\mathbf{PA}(s_t, a))$, is considered the most promising action. Here $HV()$ stands for the function that calculates the hypervolume of a given set of non-dominated Q-vectors. Whiteson [88] presents a good account of hypervolume as an important performance measure for multi-objective optimization problems.

In Section 3.3, the usefulness of both approaches A1 and A2 are experimentally studied during learning. It is important to note that, by following any of the two approaches, there is no guarantee that a Pareto optimal policy can be obtained eventually. Instead, after the learning process is completed, a separate procedure as described in the next subsection will be performed to determine Pareto optimal policies and the Pareto front.

3.2.3 Construction of the Pareto Front

In MO-XCS, the predictions of all classifiers learned from a MORL problem can be jointly used to identify the Pareto front. To achieve this goal, first of all, a starting state s_0 is fixed that can be either selected arbitrarily among all possible states S of the problem or determined by human users according to their practical interests. Next, based on the match set $[M]_0$, the collection of actions A_0, each of which follows a separate Pareto optimal policy, can be obtained. For any action that belongs to the collection A_0, its predicted long-term payoff in (3.7) must contain at least one Q-vector that is not Pareto dominated by the payoffs of other actions. If there are several such actions, to construct the whole Pareto front, the algorithm has to iterate through all these actions and perform each of them at state s_0 to create each different Pareto optimal policy.

To construct a Pareto optimal policy π^* and assuming that $\pi^*(s_0) = a_0$, a weight vector $\vec{\omega} = \{\omega_1, \ldots, \omega_m\}$ can be automatically created at state s_0 such that performing action a_0 in s_0 will produce the highest weighted sum of the long-term payoff. After performing action a_0, state s_1 is reached at time 1. The match set $[M]_1$ and action collection A_1 is built up. This time, not every action in A_1 is eligible for subsequent construction of policy π^*. Instead, the action in A_1 that yields the highest aggregated payoff (based on the same weight vector $\vec{\omega}$) will be selected and effected. This process will continue until a final state is reached. Through this way, the algorithm can eventually find a complete state-action mapping for policy π^*. The

performance of this policy can also be evaluated accordingly.

In [86], an interesting *tracing mechanism* was designed for PQL to construct Pareto optimal policies after learning. However, this tracing mechanism only works when the rewards of performing any action consistently. In addition, as the tracing mechanism is developed by tracing the value of $cl.p$, in this case, only one of the classifiers from a state will be used for constructing Pareto optimal policies. In this work, without requiring stationary rewards, a different approach based on weighted sums will be used. It is worthwhile to note that the learning process in MO-XCS does not depend on any weight vectors, which will only be used for building Pareto optimal policies from learned classifiers.

3.3 Experiment Design and Evaluation Strategy

3.3.1 Experiment Design

As the classic MDP problem, the single-objective maze problem was frequently used as the benchmark for RL algorithms and used in the literature to study the performance of LCSs [8]. However, there are not many benchmarks for evaluating MORL algorithms. To evaluate the learning effectiveness, three bi-objective maze problems, i.e. bi-objective Maze4, bi-objective Maze5, and bi-objective Maze6 are introduced and MO-XCS has experimented on these bi-objective maze problems. These three problems have been depicted respectively in Figure 3.2, Figure 3.3, and Figure 3.4. In Subsection 3.2.1, the basic settings of the bi-objective versions of the maze problems are introduced. In this experiment, these settings will be followed exactly in all the experiments. In each bi-objective maze, there are two final states, F1 and F2. The agent has two objectives to pursue in the bi-objective maze. The first objective is to reduce the total number of actions to be performed to reach a final state. The second objective is to get as much food as possible. In this case, how to choose the learned optimal

policy is depending on weighting steps and food.

T	T	T	T	T	T	T	T
T	s_1	s_2	T	s_3	s_4	F_1	T
T	T	s_5	s_6	T	s_7	s_8	T
T	T	s_9	T	s_{10}	s_{11}	T	T
T	s_{12}	s_{13}	s_{14}	s_{15}	s_{16}	s_{17}	T
T	T	s_{18}	T	s_{19}	s_{20}	s_{21}	T
T	F_2	s_{22}	s_{23}	s_{24}	T	s_{25}	T
T	T	T	T	T	T	T	T

Figure 3.2: Bi-objective Maze4

T	T	T	T	T	T	T	T	T
T	s_1	s_2	s_3	s_4	s_5	s_6	F_1	T
T	s_7	s_8	T	s_9	T	T	s_{10}	T
T	s_{11}	T	s_{12}	s_{13}	s_{14}	s_{15}	s_{16}	T
T	s_{17}	s_{18}	s_{19}	T	T	s_{20}	s_{21}	T
T	s_{22}	T	s_{23}	T	s_{24}	s_{25}	T	T
T	s_{26}	T	s_{27}	s_{28}	T	s_{29}	s_{30}	T
T	F_2	s_{31}	s_{32}	s_{33}	s_{34}	T	s_{35}	T
T	T	T	T	T	T	T	T	T

Figure 3.3: Bi-objective Maze 5

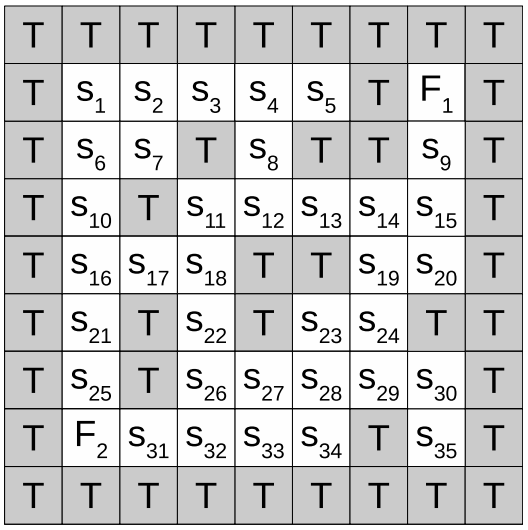


Figure 3.4: Bi-objective Maze 6

3.3.2 Performance Evaluation

In this chapter, two different performance measures will be utilized to jointly examine the effectiveness of MO-XCS. To obtain these measures, each time, after a certain number of problem instances have been learned, the performance of the learned classifiers at every open location (or non-final state) in the maze environment will be evaluated.

First, the hypervolume of the learned Pareto Optimal policies by MO-XCS can be compared with the hypervolume of the true Pareto front. For example, for an open state s_i, the optimal policies for each weight $\overrightarrow{\lambda^o}$ ($o = 1, ...n$) of a set of initialized weights is calculated by enumeration, which is a costly procedure. Each optimal policy will achieve a different long-term payoff as $\{\vec{J}_1, \ldots, \vec{J}_m\}$. Note, if 10 weights (i.e., $\overrightarrow{\lambda^1}, \ldots, \overrightarrow{\lambda^{10}}$) are initialized, they are associated with 10 long-term payoffs (for example, $\vec{J}_1, \ldots, \vec{J}_{10}$). Based on these values, the procedure to calculate the *hypervolume* in state s_1 is as follows:

$$HV(s_1) = HV\left(\{\vec{J}_1, \ldots, \vec{J}_m\}\right) \tag{3.7}$$

With (3.7), the *total hypervolume* over all states of the maze can be cal-

culated:

$$THV = \sum_{i=1}^{L} HV(s_i) \tag{3.8}$$

where L is the number of open states in the maze. The higher the THV, the better the XCS performance on the learning problem.

Second, the match rate of the learned optimal policies and the theoretical optimal policies is measured. For this purpose, from every open state s_i, the learned optimal policies will be compared against the theoretical optimal policies for the selected weight $\overrightarrow{\lambda^n}$ ($n = 0, ...k$). Only when they match, will it be counted in the policy match rate. Given this, the *percentage of true Pareto optimal policies* $\%OP$ is calculated in all the open states of the maze. Note, in these experiments, the number of weights in Subsection 4.2.1 is set as 11, so 11 evenly distributed weights between $(0, 1)$ and $(1, 0)$ are initialized. In result analysis, the result of weight $\overrightarrow{\lambda^1} = (0.1, 0.9)$ is discussed as the trend is similar between different weights.

3.4 Results and Discussions

In this section, the results of the experiments are presented. To evaluate the learning effectiveness of MO-XCS, the experiments on multi-objective mazes will be implemented to test the performance of MO-XCS on those MORL problems.

To determine the effectiveness of MO-XCS, the performance of the learned classifiers will be evaluated for 5000 problem instances at every open state in the maze environment. Both THV and $\%OP$ will be calculated by averaging the results of 30 independent runs of MO-XCS. In all the experiments in this chapter, typical parameter settings recommended in [14] have been followed. Particularly, the crossover probability $\chi = 0.8$, the mutation probability $\mu = 0.001$, the error threshold $\varepsilon_0 = 0.05$, the classifier deleting threshold $\theta_{del} = 200$, and the subsumption threshold $\theta_{sub} = 20$. The GA subsumption is set to $false$. Meanwhile, the threshold for per-

forming the niche GA is set to 500, which is much higher than 25 in single-step problems, as the reward is encountered infrequently and policy learning is slow to converge. Finally, the population size N in all maze problems is set to 6000.

The first experiment is conducted on the bi-objective Maze4 problem. As can be seen from Figure 3.5, a total of 8 different combinations over the four alternative distance measures (introduced in Subsection 3.2.1) and the two action selection approaches (introduced in Subsection 3.2.2) have been considered. Each combination has been tested separately in our experiments.

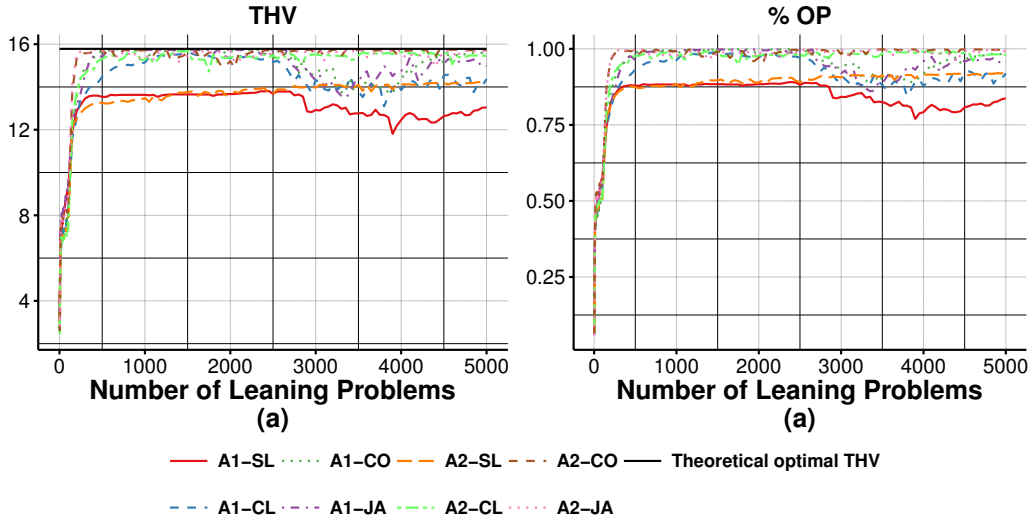


Figure 3.5: Learning performance, i.e. THV and $\%OP$, on the bi-objective maze4 problem. A1 and A2 denote the two action selection approaches in Subsection 3.2.2. SL, CL, CO and JA represent the four distance measures introduced in Subsection 3.2.1.

As clearly evidenced in Figure 3.5, depending on the distance measure and action selection method used, MO-XCS can manage to achieve theoretical optimal THV after learning through a small number of problem instances. Particularly, among all the 8 combinations in this figure, the

combinations A2-CO and A2-CL appear to be the most effective and stable. For example, when the combination A2-CO was used, after 450 problem instances, the highest average THV of 15.78 was witnessed, identical to the theoretical optimal value (indicated as a flat black line in Figure 3.5). On the other hand, although the combinations A1-CL, A2-JA, and A2-SL managed to reach a high average THV initially, large fluctuations of THV were observed afterward, suggesting unstable learning behavior in MO-XCS. This indicates with the combinations A1-CL, A2-JA, and A2-SL, the accurate classifiers have a high chance to be deleted than other combinations when the population size exceeds the maximum population size.

Meanwhile, when certain combinations, such as A1-SL and A2-SL, were used, MO-XCS can never achieve close-to-optimal THV. The observed performance differences have also been verified statistically. In particular, the one-way ANOVA test over all the results in Figure 3.5(a) produces a p-value of 0.0013. The follow-up Tukey's post hoc analysis at the standard confidence level of 0.05 further confirmed that the performance differences between the three combination pairs, i.e. A1-CO and A2-SL, A2-SL and A2-JA, A2-SL and A2-CO, are of statistical significance.

Besides THV, the change of $\%OP$ during the whole learning process has also been depicted in Figure 3.5(b). As can be easily verified, after less than 500 problem instances, several combinations including A2-C and A2-CL, achieved an average $\%OP$ close to 100%, indicating that the MO-XCS can successfully learn Pareto optimal policies in every state of the maze. This observation is consistent with the THV results. Again ANOVA test was performed, resulting in a p-value of 0.018. Tukey's post hoc analysis further showed that A2-JA and A1-SL performed significantly differently from other combinations in terms of $\%OP$ (i.e. A2-JA achieved better-than-average performance and A1-SL achieved lower-than-average performance).

The bi-objective Maze5 problem is more challenging than the Maze4 problem, as there are ten more open states in bi-objective Maze5 than that

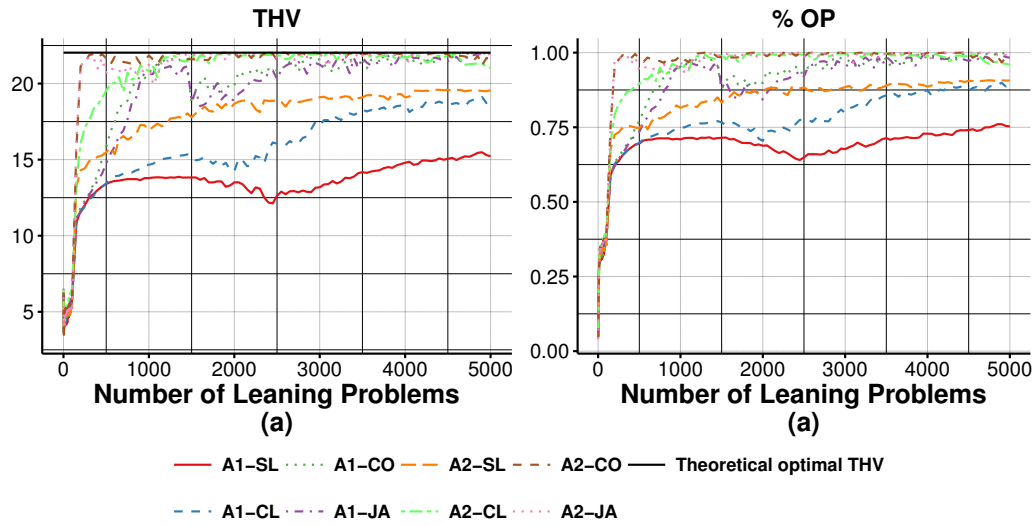


Figure 3.6: Learning performance, i.e. THV and $\%OP$, on the bi-objective maze5 problem.

in Maze4. However, as evidenced in Figure 3.6(a), MO-XCS can still manage to successfully solve this problem under certain combinations of distance measures and action selection approaches. In particular, after 2500 problem instances, close-to-optimal average THV of 22.03, 20.89, 20.89, and 22.00 has been obtained for A2-CO, A1-CO, A2-JA, and A2-CL respectively. In the meantime, A2-SL, A1-CL, and A1-SL failed to produce desirable learning performance in Figure 3.6(a). Besides THV, it is also easy to verify that, for certain combinations (i.e. A2-CO, A2-CL, and A2-JA), the $\%OP$ measure was very close to 100% after about 1000 problem instances, further confirming that MO-XCS can effectively learn all Pareto optimal policies on Maze5. Statistical analysis (both ANOVA and Tukey's post hoc analysis) has been conducted, indicating that A2-CO achieved the best performance amongst other combinations.

The experiment results on the Maze6 problem have been summarized in Figure 3.7. As shown clearly, the theoretical optimal THV of 21.17 (i.e. the black flat line in Figure 3.7(a)) has been successfully reached by MO-

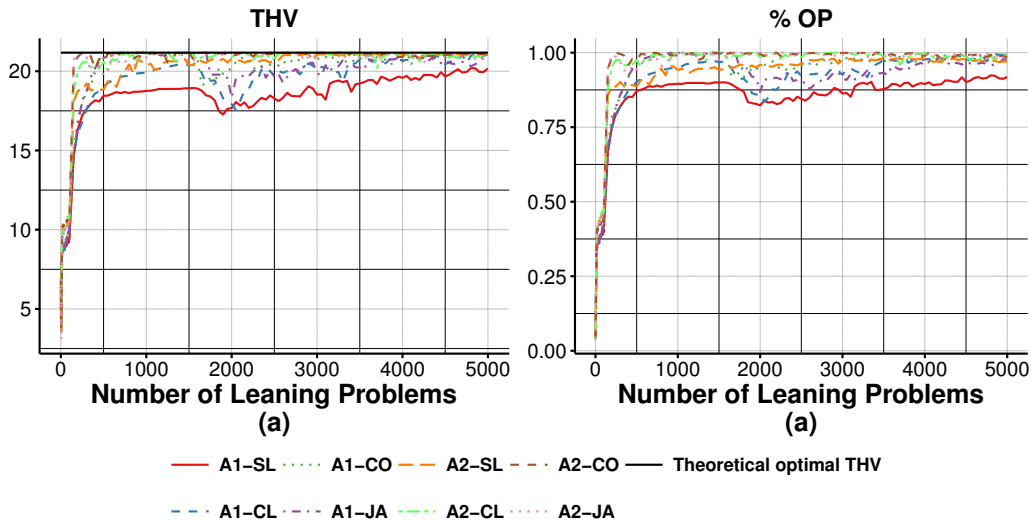


Figure 3.7: Learning performance, i.e. THV and $\%OP$, on the bi-objective Maze6 problem.

XCS after about 1000 problem instances under 5 different combinations, i.e. A1-CO, A1-JA, A2-CL, A2-CO, and A2-JA. The results of other combinations did not appear to as good as these 5 combinations. Meanwhile, the performance results in terms of $\%OP$ follow the same pattern as witnessed on the maze5 problem. In particular, the most effective combinations are A2-CO and A2-CL. Combinations that are not so successful are A1-SL and A1-CL. The above observations have been verified statistically through ANOVA tests and Tukey's post hoc analysis.

3.5 Chapter Summary

In this chapter, motivated by the practical importance of multi-objective reinforcement learning (MORL), a new reinforcement learning algorithm based on XCS has been successfully developed. Different from many recently proposed learning algorithms that rely on tabular representations of the value function, the algorithm MO-XCS facilitates a more scalable

representation in the form of a population of classifiers. It was also designed to learn multiple Pareto optimal policies concurrently through a single learning process. According to the literature review, MO-XCS is the first XCS-based algorithm for this type of learning task.

The newly developed algorithm has been experimented on three benchmark maze problems, each having two separate objectives. In the experiments in this chapter, the learning effectiveness of four alternative distance measures and two separate approaches for action selection have been specifically examined. From the results, it can be seen that the action selection method has more influence on learning performance than the distance measure. Specifically, hypervolume-based action selection (i.e. Approach A2 in Subsection 3.2.2) helps explore promising regions of the learning space more effectively and therefore helps to achieve more effective learning.

The developed multi-objective XCS algorithm can learn a group of Pareto optimal solutions through a single learning process. Here the classifiers must store the whole history of predictions for the states the LCS visited, and select one such stored prediction as the prediction of the classifier. However, this work exhibits several limitations. First, the stored strategies may consume a large amount of storage. Second, the tracing mechanism employed for constructing Pareto optimal policies will only choose one of the classifiers in the next state, thus removing the generalization ability of XCS. In the next chapter, a new multi-objective XCS (MOXCS) will be developed to discover multiple optimal policies simultaneously that generalize where possible, without large storage requirements.

Chapter 4

MOXCS: Decomposition based Multi-Objective Evolutionary Algorithm in XCS for Multi-Objective Reinforcement Learning

4.1 Introduction

In Chapter 3, a multi-objective XCS (MO-XCS) has been developed by learning a group of Pareto optimal solutions through a unique learning process. Here, the classifier must store the complete prediction history for the visited states and select one of these backup predictions as the classifier's prediction. However, this work has several limitations. First, the backup strategy can consume a large amount of storage. Secondly, the tracing mechanism for constructing the Pareto optimal policies has removed the generalization capability of XCS.

4.1.1 Chapter Goals

In this chapter, a new multi-objective XCS algorithm, MOXCS is developed to discover several optimal policies simultaneously that are generalized as much as possible, without large storage requirements. As MOXCS supersedes MO-XCS, the name is deliberately similar. It is assumed that the decomposition of MOEA/D for multipurpose tasks can sufficiently approach the complicated forms of Pareto Front to allow for such policy learning within XCS. Five major technical issues have been identified and addressed to achieve this goal in this chapter as follows.

(1) It decompose a MORL problem to N single objective RL sub-problems by different weights with MOEA/D.

(2) The classifier structure is updated to maintain multiple objectives.

(3) The process of updating classifier parameters is updated as it will update classifier parameters with multiple objectives.

(4) The Genetic Algorithm (GA) process, classifiers with similar weights will be used to search for the optimal solutions.

(5) The condition of the classifier is updated to consume the integer input to address the environment with integer input.

4.1.2 Chapter Organisation

The rest of this chapter are organized as follows. Section 4.2 uses a simple bi-objective corridor problem to explore the effects of MORL problems. Section 4.3 described the experiment settings and evaluation. Three MORL experiments results from testing on bi-objective maze, Deep Sea Treasure, and Multi-maze are presented in Section 4.4, 4.5 and 4.6. Finally, the Section 4.7 presents our conclusions and highlights potential future work.

4.2 Methodology

This Chapter aims to employ a decomposition strategy based on MOEA/D in XCS to approximate complex Pareto Fronts. To achieve multi-objective learning, new MORL algorithms have been developed in this chapter based on XCS and MOEA/D as MOXCS. The methodologies of MOXCS are described in Section 4.2.1. In order to handle the integer input in MOXCS, the Min-Max Representation (MMR)[72] in Section 4.2.2 has been used.

4.2.1 Adding MOEA/D to XCS

In this section, the Multi-Objective XCS (MOXCS) algorithm is proposed, which combines MOEA/D into XCS to enable learning the Pareto optimal policies for MORL problems. A schematic view of the MOXCS general architecture is presented in Figure 4.2. MOEA/D is employed to decompose the multi-objective problem into multiple single objectives by initializing a set of evenly distributed weights. For XCS, the classifier structure is modified to maintain multiple objectives. The parameters updating process in classifier reinforcement and GA in rule discovery is implemented by the weights.

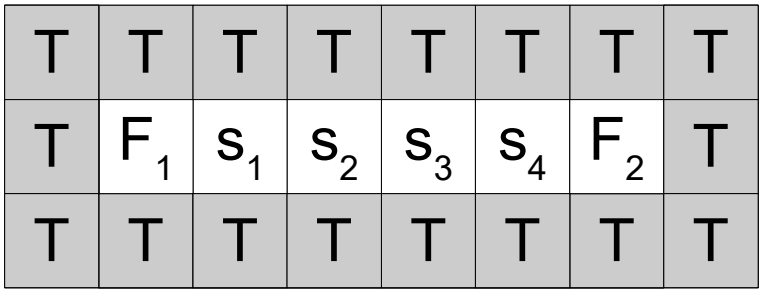


Figure 4.1: Bi-objective Corridor, where block T is impassable.

A simple bi-objective corridor problem in Figure 4.1 is used to illustrate the functionality of the novel MOXCS algorithm. There are four open states ($s_1 ... s_4$); two goal states (F_1, F_2); and only two possible actions that

an agent can take $(left, right)$. An episode can start from an open state and finishes once the agent enters a goal state. When the agent reaches a goal state it immediately receives a reinforcement from 'gold' reward (to differentiate it from the "reward" signal in single objective RL updates). There are two objectives, to minimize the steps and maximize the gold reward. Therefore, the immediate reward signal is a two-dimensional vector $\vec{r}$, which is received for each step. The first element is always -1 as the cost per step, the second element is the amount of the reward that is received at the current state. For example, $\vec{r} = \{-1, 0\}$ at an open state; $\vec{r} = \{-1, 100\}$ at state F_1; $\vec{r} = \{-1, 900\}$ at state F_2.

Initialization of weights and neighbour size

The main idea of MOXCS is to decompose a MORL problem to N single objective RL sub-problems by different weights $\overrightarrow{\lambda^i}(i = 1, ..., N), \lambda^i_k \geq 0$ for all $k = 1, ..., m$ and $\sum_{k=1}^{m} \lambda^i_k = 1$. Then use XCS to solve each single objective RL problem simultaneously. This enables knowledge learnt from one combination of weights to improve another, which is not possible in the weighted sum approach. For each weight $\overrightarrow{\lambda^i}$, it will use the m closest neighbours information to address the current sub-problem.

Following the MOEA/D methodology, the number of weights N, number of neighbours T and a reference point for the best performance for the prediction of each classifier on each dimension will be initialized. In addition, classifier cl_i with related weight $\overrightarrow{\lambda^i}$ will also be also initialized. For example, suppose $N = 6$ and $T = 3$, MOXCS will initialize 6 classifiers $cl_1, cl_2, cl_3, cl_4, cl_5, cl_6$ with 6 weights $\overrightarrow{\lambda^1} = (0.0, 1.0)$, $\overrightarrow{\lambda^2} = (0.2, 0.8)$, $\overrightarrow{\lambda^3} = (0.4, 0.6)$, $\overrightarrow{\lambda^2} = (0.6, 0.4)$, $\overrightarrow{\lambda^2} = (0.8, 0.2)$ and $\overrightarrow{\lambda^2} = (1.0, 0.0)$. Then for each $\overrightarrow{\lambda}$, for example $\overrightarrow{\lambda^1}$, it will calculate the distance between $\overrightarrow{\lambda^1}$ and other weights. When updating the prediction of cl_1, if one of the prediction from cl_2, cl_3 and cl_4 is better than the prediction of cl_1, the prediction of cl_1 will be updated. More details will be introduced in the next subsection.

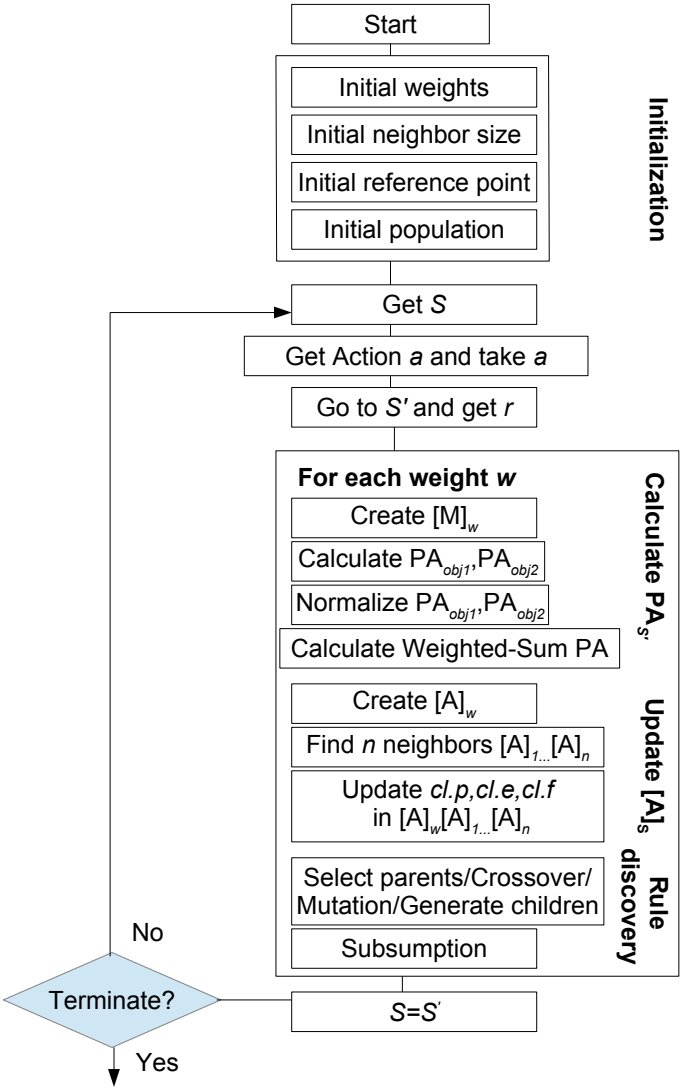


Figure 4.2: General architecture of MOXCS.

Classifier Structure

For the classifier structure, each classifier cl_i will be extended with a weight $\overrightarrow{\lambda^i}$, and used to solve the specific single problem with $\overrightarrow{\lambda^i}(i = 0, ...N)$, where

N is the number of initialized weights. In standard XCS, the prediction ($cl.p$), error ($cl.\varepsilon$) and fitness ($cl.f$) of classifier are scalars. However, in our work, to solve an m $objective$ MORL problem, the prediction $\overrightarrow{cl.p}$, error $\overrightarrow{cl.\varepsilon}$ and fitness $\overrightarrow{cl.f}$ are m dimensional vectors. In order to maintain the predictions of m objectives, each classifier has m predictions, errors and finesses. Following the example in Figure 4.1, classifier cl_1 should have a weight $\overrightarrow{\lambda^1} = (0.0, 1.0)$, parameters $cl.p_1$, $cl.\varepsilon_1$ and $cl.f_1$ for objective 1, and parameters $cl.p_2$, $cl.\varepsilon_2$ and $cl.f_2$ for objective 2.

Note, as the predictions for objective 1 and objective 2 may have huge differences, when updating the parameters, the error of a classifier $\overrightarrow{cl.\varepsilon}$ needs to be normalized between 0 and 1. In this case, the fitness of classifier $\overrightarrow{cl.f}$ is calculated according to the normalized error.

Updating Classifier Parameters

When the agent moves to a new state, the reinforcement component process that updates the classifier parameters will go through for each weight $\overrightarrow{\lambda^i}$. Note, in the prediction calculation process, the prediction array for each objective is calculated separately first, and then a normalization process will be implemented to ensure that the value of each prediction is within 0 and 1 inclusively. So the goal reward and the step can be considered without bias.

Compared with XCS, there are two changes in updating the $\max(PA)$ in formula for calculating $cl.p$ in standard XCS [14]. 1) The $\max(PA)$ is formed by the classifiers that match not only the current state but also the current weight $\overrightarrow{\lambda^i}$. 2) The following formulas are employed to calculate the weighted-sum prediction $PA_{\overrightarrow{\lambda^i}}[a]_{a\in A}$ for the current weight $\overrightarrow{\lambda^i}$:

$$PA_{\overrightarrow{\lambda^i}}[a]_{a\in A} = \sum_{o=1}^{m} \overrightarrow{\lambda^i}_o * PA_o[A] \tag{4.1}$$

where the $PA_i[a]$ is the prediction for the objective i for action a, and m is the number of objectives. Following formula (4.1), the largest action

prediction array PA can be calculated as follows:

$$\max_{a \in A}(PA) \leftarrow \max_{a \in A}(PA_{\overrightarrow{\lambda^i}}) \tag{4.2}$$

where the $max(PA_{\overrightarrow{\lambda^i}})$ is the maximum weighted-sum action prediction for $\overrightarrow{\lambda^i}$ in the prediction array $PA_{\overrightarrow{\lambda^i}}$.

Suppose the agent moves from s_3 to s_4, and the weight is $\overrightarrow{\lambda^i}$, where $\lambda_0^i = 0, \lambda_1^i = 1$. The PA for each objective can be calculated separately using the same method as in XCS [14]. Suppose the prediction array for objective 1 at s_4 for $left, right$ is $PA1_{s_4} = (4, 1)$, and the prediction array for objective 2 $PA2_{s_4} = (65, 810)$. Following formula (4.1), as $\lambda_0^i = 0$, the weighted-sum $PA_{\overrightarrow{\lambda^i}}$ should equal $PA2_{s_4}$, thus $PA_{\overrightarrow{\lambda^i}} = (65, 810)$. With formula (4.2), for $\overrightarrow{\lambda^i}$, largest weighted-sum action prediction $max(PA_{\overrightarrow{\lambda^i}}) = 810$ is selected, so the agent takes the second action to go right.

Rule Discovery

Inspired by MOEA/D, the sub-problems with similar weight will have similar solutions. Therefore, in the Genetic Algorithm (GA) process, classifiers with similar weights will be used to search for the optimal solutions. The GA process will also loop by weight as follows.

(1) For $\overrightarrow{\lambda^i}$, compute the $Euclidean\ distances$ between any two weight vectors and then get the T closest weight vectors to $\overrightarrow{\lambda^i}$. Group the classifiers with weights $\overrightarrow{\lambda^i}$ or T closest weight vectors to $\overrightarrow{\lambda^i}$ as $[T]$.

(2) Select two parents classifiers from $[T]$ based on fitness and generate two new child classifiers.

(3) Update reference point $\overrightarrow{z^*}$. Suppose it is a maximize multi-objective problem, for each cl_i, if $z_j < cl_i.p_j$, then set $z_j = cl_i.p_j$.

(4) Update of neighboring solutions. Suppose it is a maximize multi-objective problem, for classifier cl_l in $[T]$, if $g^{te}(cl_l.p_j|\lambda_i, z_i^*) < g^{te}(cl_i.p_j|\lambda_i, z_i^*)$,

then set $cl_l.p_j = cl_i.p_j$. where g^{te} is the distance between these two prediction value of these two classifiers.

Following the example bi-objective corridor in Figure 4.1, suppose the agent moves from s_3 to s_4, now MOXCS updates the classifiers with weight $\overrightarrow{\lambda^i} = (0, 1)$ and $cl.a = right$ step by step. That is:

(1) Calculate the Euclidean distance Dis between $\overrightarrow{\lambda^1}$ and each other weight $\overrightarrow{\lambda^m}$. For example, the distance between $\overrightarrow{\lambda^1}$ and $\overrightarrow{\lambda^2}$ is $Dis_{12} = \sqrt[2]{(0-0.2)^2 + (1-0.8)^2} = 0.283$, similarly, other distance can be calculated $Dis_{13} = 0.566$, $Dis_{14} = 0.849$, $Dis_{15} = 1.131$, $Dis_{15} = 1.414$. As $T = 3$, the classifiers with $\overrightarrow{\lambda^1}$ or $\overrightarrow{\lambda^2}$ or $\overrightarrow{\lambda^3}$ will be collected in $[T]$.

(2) Suppose there are 3 classifiers in $[T]$, where $cl_1.f = 0.9, cl_1.p_1 = 1.75, cl_1.p_2 = 810$, $cl_2.f = 0.7, cl_2.p_1 = 1.25, cl_2.p_2 = 710$ and $cl_3.f = 0.3, cl_3.p_1 = 1, cl_3.p_2 = 300$. The GA selects cl_1 and cl_2 as two parents. Following the GA process in XCS, two child classifiers will be generated through crossover and mutation. For example, child classifiers cl_3 and cl_4 have the same fitness and prediction, for example, $cl_3.f = cl_4.f = 0.8, cl_3.p_1 = cl_4.p_1 = 1.50, cl_3.p_2 = cl_4.p_1 = 760$, but may have different conditions $cl.c$.

(3) Update the reference point $\overrightarrow{z^*}$ if the prediction of cl_3 or cl_4 is better than $\overrightarrow{z^*}$. For example, when $cl_3.p_1$ and $cl_4.p_1$ is less than z_1^*, or $cl_3.p_2$ and $cl_4.p_2$ is larger than z_1^*, as this algorithm is aiming to minimize the first objective and maximize the second objective. Suppose $z_1^* = 3.50, z_2^* = 780$, the new value of $\overrightarrow{z^*}$ should be $z_1^* = 1.50, z_2^* = 780$.

(4) In this work, the Tchebycheff method [97] is employed to update the prediction of classifiers in $[T]$. Considering $cl_3.p_1$ and $cl_4.p_1$ are better than $cl_1.p_1$, $cl_3.p_2$ and $cl_4.p_2$ is better than $cl_2.p_2$ and $cl_3.p_2$, the updated prediction for cl_1, cl_2, cl_3 should be $cl_1.f = 0.9, cl_1.p_1 = 1.50, cl_1.p_2 =$

810, $cl_2.f = 0.7, cl_2.p_1 = 1.25, cl_2.p_2 = 760$ and $cl_3.f = 0.3, cl_3.p_1 = 1, cl_3.p_2 = 760$.

Deterministic Result

In this section, the bi-objective corridor is manually explored in Figure 4.1 to get the deterministic result, aiming to demonstrate how gold reward settings affect the multi-objective optimal policies. The total gold reward for both F_1 and F_2 is 1000, the reward ratio for F_1 and F_2 starts from 0 to 1, and increases 0.1 by each time (for example, 0 vs 1000, 100 vs 900...1000 vs 0). The discount rate is 0.9. Two episodes are implemented for each reward setting. Suppose the first episode starts from s_1, and the agent only takes action right until it arrives at F_2; whereas, the second episode starts from s_4, and the agent only takes action right until it arrives at F_1. In this case, for each state, the perdition of long-term payoff for action left and right can be calculated. For example, if gold setting is 100 vs 900, the left prediction of gold for s_1 is $100 \times 0.9 = 90$, the right prediction of gold for s_1 is $900 \times (0.5)^3 = 590.49$. Next, the difference of the prediction for right and left ΔR can be calculated. In this case, for s_1, $\Delta R_s 1 = 590.49 - 90 = 500.49$. Next, the maximum reward for each state is calculated, for example, the maximum reward of s_1 is 590.49. The ΔR and maximum reward for all the open states and reward settings plots in Figure 4.3.

When the $\Delta R < 0$, the agent will go right; otherwise, go left. Two things should be noticed. First, if the states are symmetrical about the final states (for example, s_2 and s_3, s_1 and s_4), they will have the symmetrical reward settings affection. Second, the policy of a state that is far from both final states (for example, s_2 and s_3) will be more influenced by the change of reward setting than the states closer to one of the final states (for example, s_1 and s_4). As s_2 and s_3 are affected by the reward changes of both final states, but s_1 is less affected by the reward of F_2, also s_4 will be less affected by the reward of F_1.

Figure 4.3: Bi-objective corridor deterministic results showing maximum predicted reachable reward for each state compared with the differences in the two predicted rewards as the actual difference ΔR varies.

4.2.2 Integer Input

To enable MOXCS and MOZCS to learn Pareto policies with integer-valued inputs, a method similar to Min-Max Representation (MMR) [72] is introduced. In this method, each attribute in the condition of the classifier is represented as an interval predicate in the form of (l_i, u_i) where l_i, u_i are the minimum and maximum bounds of the interval respectively. The genetic operators such as covering, mutation, crossover, and subsumption also need to be changed.

(1) The covering technique generates a new classifier into the system when the population does not contain any classifier to match the

current input. The covering process creates a classifier containing intervals (l_i, u_i) given by $l_i = x_i - R[0, s0)$; $u_i = x_i + R[0, s_0)$ where s_0 is a constant number, $R[0, s_0)$ is a random number between 0 and x_i, and x_i is an input value.

(2) In the mutation process, the l_i or u_i might be added or minus $R[0, s_0)$ by chance.

(3) In the crossover process, two classifiers will be selected, l_i or u_i of them might be switched by chance.

(4) It follows the subsumption process in the MOXCS and MOZCS, but when $l_i > l_m$ and $u_i < u_m$ means classifier i is more general than classifier m.

4.3 Experiment Settings

In order to evaluate the learning effectiveness of MOXCS, the experiment will be implemented on two multi-objective Markov problems: multi-objective maze, deep sea treasure corridor and one multi-objective PO-MDP problem: deep sea treasure to test the performance of MOXCS on those MORL problems.

In order to determine the effectiveness of MOXCS, the performance of the learned classifiers will be evaluated for 3500 problem instances at every open state in the maze environment. In this work, two different performance measures *total hypervolume over all states* (THV) and *the match rate of the learned optimal policies and the theoretical optimal policies* ($\%OP$) as described in subsection 3.3.2, page 79 will be used to evaluate the learned Pareto Front and the learned optimal policies.

For this purpose, from every open state s_i, the learned optimal policies will be compared against the theoretical optimal policies for the selected weight $\overrightarrow{\lambda^n}$ ($n = 0, ...k$). Only when they match, will it be counted in the

policy match rate. Given this, the *percentage of true Pareto optimal policies* $\%OP$ is calculated in all the open states of the maze. Note, in the experiment, the number of weights in 4.2.1 is set as 11, so 11 evenly distributed weights between $(0, 1)$ and $(1, 0)$ are initialized. In result analysis, only the result of weight $\overrightarrow{\lambda^1} = (0.1, 0.9)$ is demonstrated as the trend is similar between different weights.

Both THV and $\%OP$ will be calculated by averaging the results of 30 independent runs of MOXCS. In all the experiments, typical parameter settings recommended in [14] has been followed. Particularly, the crossover probability $\chi = 0.8$, the mutation probability $\mu = 0.001$, the error threshold $\varepsilon_0 = 0.05$, the classifier deleting threshold $\theta_{del} = 200$, and the subsumption threshold $\theta_{sub} = 20$. The GA subsumption is set to $false$. Meanwhile, the threshold for performing the niche GA is set to 500 which is much higher than 25 in single step problems, as the reward is encountered infrequently and policy learning is slow to converge. Finally, the population size N in all maze problems is set to 6000.

4.3.1 Benchmarks

The test problems, tested algorithms, baselines, and why they have been chosen are described as follows: The effectiveness and efficiency of MOXCS have been tested in bi-objective mazes, and more testing for discussion of Pareto Front follows in subsection 4.4. MO-XCS, which was tested in Chapter 3, has been chosen as a baseline to evaluate the effectiveness and efficiency of MOXCS as MOXCS is designed to improve the limitations of MO-XCS (eg. consuming a large amount of storage, lack of generalization ability). In subsection 4.5, the following experiments are implemented. MOXCS, MOZCS, and PQL have been tested in the Deep Sea Treasure Corridor problem. PQL functions as a baseline because it is an existing MORL algorithm for DST, thus it should solve DSTC. MOXCS-integer and PQL have been tested in the Deep Sea Treasure problem for

evaluating the effectiveness and efficiency of the PO-MDP problem. PQL works as a baseline because it is an existing MORL algorithm for the PO-MDP problem DST. MOXCS and XCS have been tested in the Deep Sea Treasure Corridor (DSTC) for comparing the number of solutions on the PF learned by the algorithms. MOXCS is the baseline, and it compares with XCS to show whether MOXCS can learn more optimal solutions to DSTC than XCS. Note, though PQL can solve DSTC as well, it lacks generalization ability thus it is not tested here and for further problems.

MOXCS and MOZCS have been tested in the DSTC problem, which analyzes the relationship between learning more solutions on a PF and learning problem itself. From the previous testing problems, MOXCS has demonstrated its effectiveness, efficiency, and generalization ability, thus MOXCS is the algorithm that is taken forward and works as a baseline for this test. The result shows MOXCS is more efficient (faster to learn) on learning solutions on the PF than MOZCS. MOXCS has been tested in the DSTC problem for analyzing the relationship between more solutions on a PF and different weights and exploring PF shapes. This is because MOXCS is more efficient in learning solutions on PF than MOZCS on DSTC, so it had been adopted for testing to learn solutions with specified weights, and testing with different shapes of PF to see if the Pareto optimal policies can be learned to address different shapes of PF.

The Novel MOXCS and MOXCS have been tested in a small Multi-Objective maze for evaluating the search strategy. The baseline is MOXCS for comparing the performance of MOXCS with novelty search strategy and without novelty search strategy on MORL problems, however, the experiment result shows novelty search cannot help to improve the capability of MOXCS to learn more multi-objective optimal solutions. In subsection 4.6, MOXCS, MOZCS, and PQL have been tested in Multi-Maze. PQL works as a baseline for comparing the generalization ability of the other two LCS-based algorithms, where PQL cannot generalize the environment while MOXCS and MOZCS can generalize the environment by

encoding the inputs. Then, MOXCS has been planned to be tested in the Multi-Maze connection problem for evaluating its generalization ability of it. However, during the testing, it has been found that the environment does not work for testing the MORL algorithm.

4.4 Results: Bi-Objective Maze

To evaluate the learning effectiveness of MOXCS, experiments are conducted on three complex bi-objective maze problems (bi-objective maze4, bi-objective maze5, and bi-objective maze6) in Figure 3.2, Figure 3.3, and Figure 3.4 respectively. The results will be demonstrated and discussed in this section. Here 'food' is reached with 'gold' to align the terminology with Deep Sea Treasure problem description.

4.4.1 Bi-Objective Maze4

As all the learning curves for the different gold reward settings have a similar trend, Figure 4.4 only shows the performance of gold reward for F_1 and F_2 as 30 vs 100 with 4 weights ($\overrightarrow{\lambda^1}...\overrightarrow{\lambda^4}$) for MOXCS on bi-objective maze4. The result is divided into two parts, the Total Hypervolume (THV) of each open location and the Optimal Policy match rate ($\%OP$). In Figure 4.4, there are three lines in the same color used to represent the performance for each weight. The middle line is the average value of the results from 30 runs for THV and $\%OP$, whereas the other two lines demonstrate the variance of the performance. Note, in the first 1000 problem instances, the variance is large, but not much difference after 1000 problem instances, which means MOXCS becomes much more stable after 1000 problem instances. It shows that MOXCS can manage to achieve the theoretical optimal THV after learning 800 problem instances. It is also shown that $\%OP$ has the almost same trend as THV, but $\%OP$ converges faster than THV. $\%OP$ achieves 100% after only 400 problem instances, as small changes in

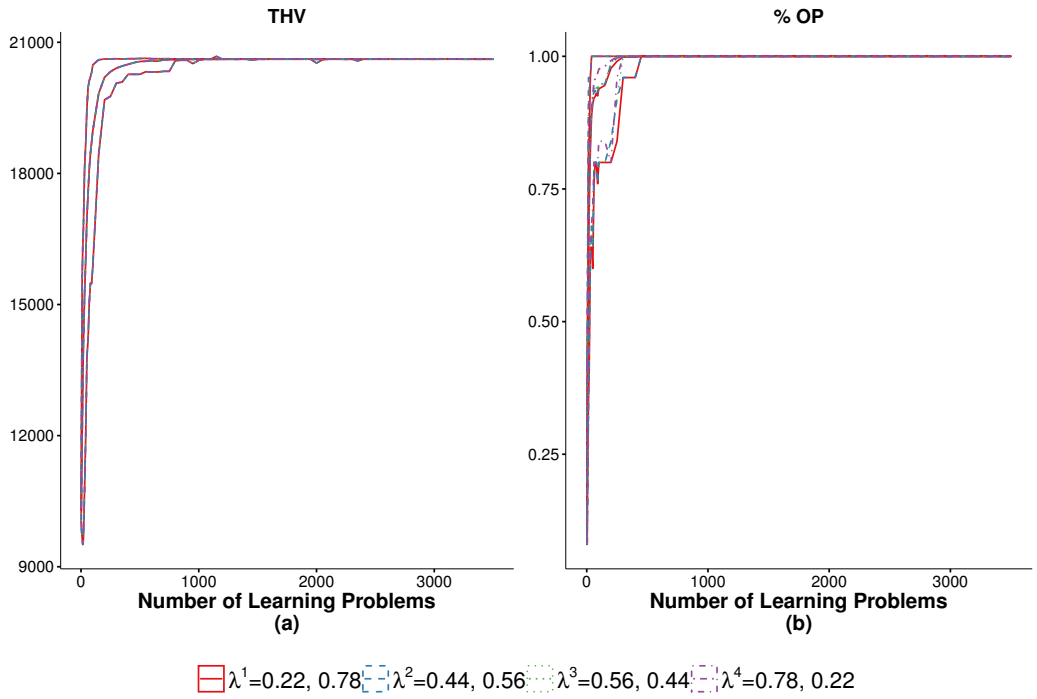


Figure 4.4: Learning performance, i.e. The average value and variance of THV and $\%OP$, on the bi-objective maze4 problem.

THV sometimes do not affect the action selection.

The full results of different weights and gold reward settings for MOXCS on bi-objective maze4 are shown in Table 4.1, where the $R1$ is the reward for 'F1' in Figures 3.2, and THV^* is the optimal total hypervolume over all states for the reward settings in $R1$.

It can be seen that MOXCS reaches the optimal performance for THV and $\%OP$ for all the gold reward settings after 1200 and 550 learning problem respectively. Note, when there is a huge difference between the reward settings on 'F1' and 'F2', THV^* is much high than that when the reward settings on 'F1' and 'F2' is similar. In this case, the learning problems increase a bit, but it is reasonable in order to achieve a better performance.

Table 4.1: Learning performance on the bi-objective maze4

R1	20	30	40	50	60	65	70	80	90	100	110
THV*	22396	20615	19225	18015	17016	16727	16695	16932	17606	19552	21217
THV	1200	800	650	650	550	450	450	600	700	800	800
%OP	300	400	450	550	350	350	250	300	300	550	55

Table 4.2: Learning performance on the bi-objective maze5

R1	20	30	40	50	60	65	70	80	90	100	110
THV*	28754	26842	25368	24185	23537	23336	23318	23530	24275	25314	26930
THV	1400	1300	1100	900	900	750	650	1100	1200	1500	1450
%OP	600	550	850	650	900	750	650	900	900	900	950

4.4.2 Bi-Objective Maze5

Similar trends of maze4 were observed in the performance of maze5. The optimal performance for THV and $\%OP$ for bi-objective maze5 is achieved after 1400 and 950 problem instance respectively, see Table 4.2.

4.4.3 Bi-Objective Maze6

The experiment on maze6 has the same trend with maze4 and maze5. The performance of gold reward for F_1 set from 20 to 10 on bi-objective maze6 is shown in Table 4.3.

Table 4.3: Learning performance on the bi-objective maze6

R1	20	30	40	50	60	65	70	80	90	100	110
THV*	26373	24589	23036	22051	22024	22171	22654	24090	25962	28119	30314
THV	1250	1250	900	950	750	1000	750	900	1000	1350	1050
%OP	500	550	600	850	750	650	550	750	850	900	900

4.4.4 Discussion of Pareto Front

In Subsection 4.2.1, the deterministic result of the influence of the reward settings on PFs was demonstrated. In this section, the result of varying the reward settings at the final state F_1 and F_2, and weights settings on bi-objective maze policy learning problems will be tested. Certain states are selected to compare the influence of varying gold rewards and weights, as the influence is not only affected by the current weight and gold reward settings but also highly related to the position of states in the maze. All analysis data was generated only for one single run to analyze the behavior. Results were collected after 3500 training problem instances in each experiment because the learned policies are converged and stable after 3500 training problem instances.

Reward Settings

In order to demonstrate how the reward settings affect the action selection for states having two objectives, the weights are fixed as $\overrightarrow{\lambda} = (0.0, 1.0)$, which means the agent only favors the larger reward. The total reward for both F_1 and F_2 is 130, the reward for F_1 starts from 20, and increase 10 by each time, until it reaches 110, whereas, the reward for F_2 starts from 110, and decreases to 20 finally. There are 10 reward settings that are tested for each bi-objective maze problem.

In Figure 4.5, the axis is the difference of reward from F_1 and F_2, i.e. the value of average prediction of action following policy to F_1 subtracted by the average prediction of action following policy to F_2. If the Difference of PA for F_1 and F_2 is positive ($\Delta PA > 0$), the action following policy to F_1 will be selected, otherwise the action following policy to F_2 will be selected. In this experiment, the influence of reward settings is studied on the states $s_{10}, s_{14}, s_{22}, s_8$ in bi-objective maze4. A few points from Figure 4.5 should be noticed. First, s_8 has less affect than s_{10} (same with s_{22} compared with s_{14}), as s_8 is further from F_2, so the affect of the reward from

Figure 4.5: Influence of Gold Settings on Action Selection in Maze4

F_2 is discounted by the discount rate γ in equation. 2.21. In addition, the trend of action selection is symmetrical according to its position. For example, s_{10} and s_{22} have an opposite trend on action selection policies when changing the reward ratio of F_1 and F_2. Similarly, these two findings are also observed in bi-objective maze5 and maze6.

Weights and Rewards

Figure 4.6, Figure 4.7 and Figure 4.8 show how the weights of objectives and ratio of gold setting in the second objective (gold collection) affect the policy for action selection for three open positions s_{10}, s_3, s_{13} in bi-objective maze 4, 5, 6 respectively. In each figure, each line with different shape shows the influence of the gold reward varies as in Subsection 4.4.4. Different lines represent different weights settings. The figures show that

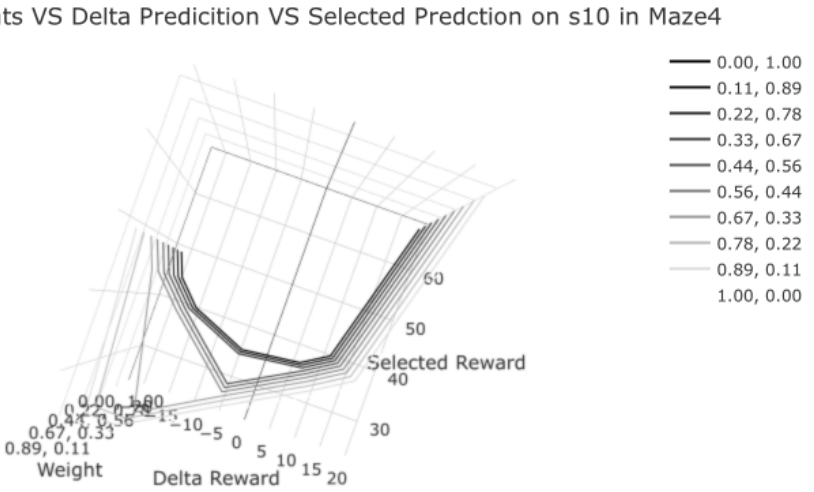


Figure 4.6: Reward and Weights Influence on Action Selection in Maze4

when placing more weight on gold, the action selection policy will be similar to the policy only considering the gold setting on the second objective. On the other hand, when the weights vary, the learned optimal policy will change as well.

4.5 PO-MDP Environment

As the results of Bi-Objective Maze problems in Section 4.4, MOXCS can solve the MORL problems in Markov environments. In this section, MOXCS is modified by two approaches to address a Multi-Objective Reinforcement Learning problem, Deep Sea Treasure problem. The first approach is adding extra characters in the DST environment to transform DST as an Non-Markov environment, which called Deep Sea Treasure Corridor (DSTC). The second approach is to edit the condition of the classifiers as integers to cope with the DST environment.

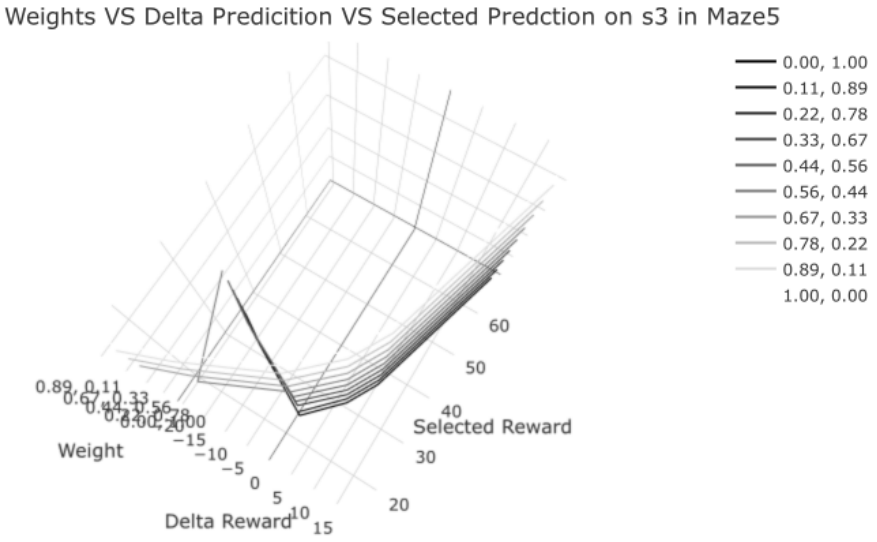


Figure 4.7: Reward and Weights Influence on Action Selection in Maze5

4.5.1 Deep Sea Treasure Corridor

The original Deep Sea Treasure (DST) is one of the benchmarks for testing MORL algorithms [86]. In Figure 4.9, the DST environment is a grid of 11 rows and 10 columns. A submarine is moving in the grid and trying to find one of the treasures at the bottom of the sea. The submarine can perform four different moves ($up, down, right, left$). In case the action applied takes the ship off the grid or into the sea floor, the submarine's position does not change.

When applying the traditional MORL algorithm to solve the DST problem, the inputs are the x-axis and y-axis index of the current position [86]. However, MOXCS receives a binary encoded input from the observation of the current state. In this case, there will have lots of aliased inputs from the states in the environment. For example, for states with the same color in Figure 4.9, they have the same binary encoded inputs. To cope with the non-Markov MORL problem in this DST environment, here, some walls

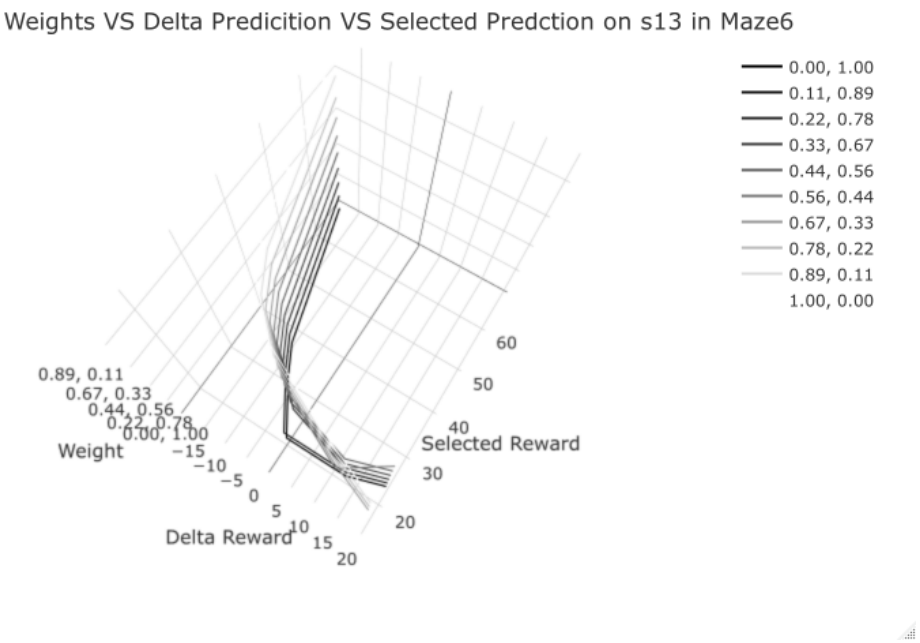


Figure 4.8: Reward and Weights Influence on Action Selection in Maze6

'T' and 'N' will be added into the top right part of DST environment to make it as an observable Markov problem. As shown in Figure 4.10, the open states are the only a set of states near those treasures (final states with reward points. eg. 3000), which looks like a corridor, which is why it is called "Deep Sea Treasure Corridor (DSTC)".

In Figure 4.10, the grey squares represent the treasures and the numbers on the squares are their reward values. The black squares are the sea floor and the white ones are the open states that the submarine can go through freely. In the training process, each trail starts at any open state of the grid and ends when the submarine picks one of the treasures. In the testing process, each trail starts from the top left state of the grid and ends when picking one of the treasures. This problem has two objectives: to minimize the number of moves performed while maximizing the value of the treasure found. The immediate reward signal is a two-dimensional vector $\vec{r}$, which is received for each step. The first element is always -1

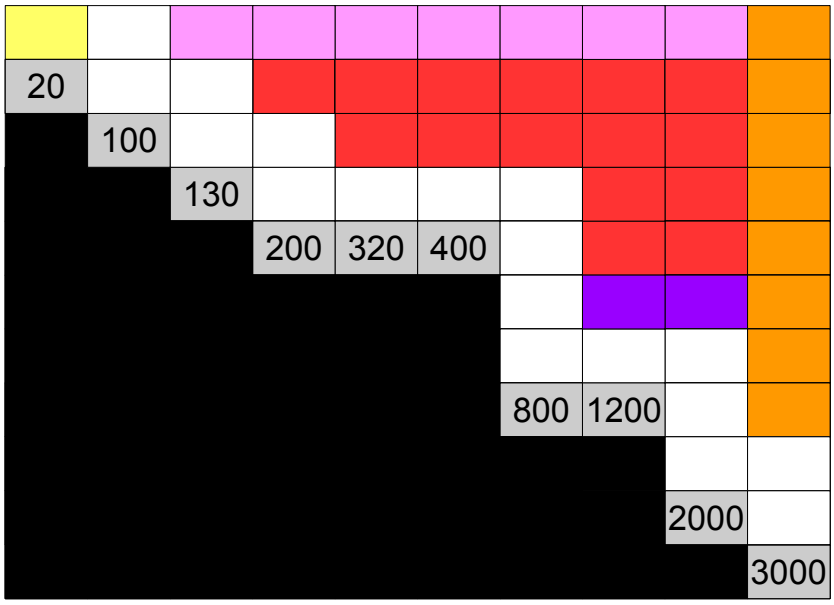


Figure 4.9: Deep Sea Treasure

as the cost per step, the second element is the amount of reward that is received at the current state. For example, 0 is received at any open state; when the agent moves into a treasure location, the value indicated in Figure 4.10 is received. Note, the treasure value settings in the experiment are different from the standard DST problem, as when the discounted value from different treasures to the start state is higher, it is easier for MOXCS to collect those treasures.

Similar to bi-objective maze problems, the result is divided into two parts, the Optimal Policy match rate ($\%OP$) and the Total Hypervolume (THV), but the THV is only calculated on the start position (S_1). In Figure 4.11, we can see that $\%OP$ achieves 100% after only 50 problem instances for MOXCS and 200 for MOZCS. To treat bi-objective maze problems as benchmark, the PQL is tested on the problem as well, PQL achieves 100%

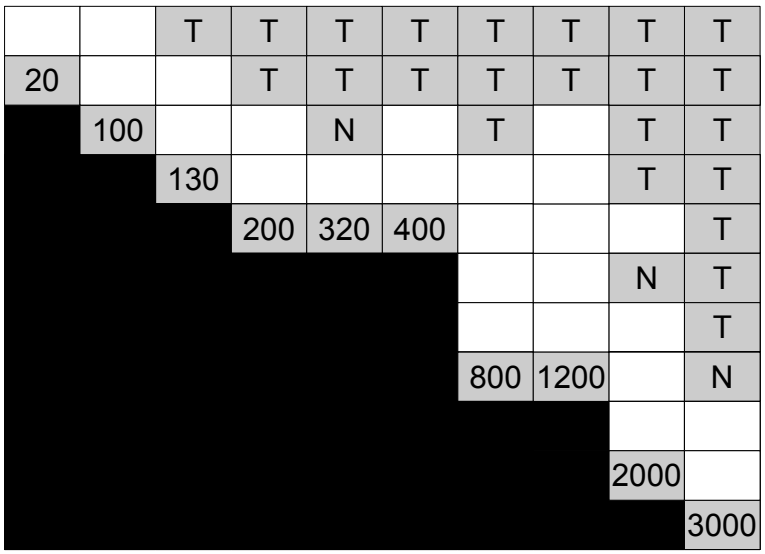


Figure 4.10: Deep Sea Treasure Corridor

after 500 learning problems. On the other hand, MOXCS and MOZCS can manage to achieve the theoretical optimal THV after learning 2500 problem instances, but PQL achieves the theoretical optimal THV after 500 learning problems. Clearly, PQL can learn more optimal policies with the Pareto strategy as it stored all the potential solutions. For MOXCS and MOZCS, the diversity is maintained with GA, enable them to find the some of the optimal policies earlier. As the result shown in Figure 4.11, the MOXCS converges faster than MOZCS on $\%OP$, and they have a similar performance on THV. In this case, MOZCS is not considered as a solution to the other similar MORL problems.

4.5.2 Deep Sea Treasure

As mentioned in Section 4.5.1, the deep sea treasure is a 11 $\times$ 10 grid in Figure 4.9 with 10 final states. It looks similar to the conventional DST problem. For the traditional LCSs, the inputs of the current position are

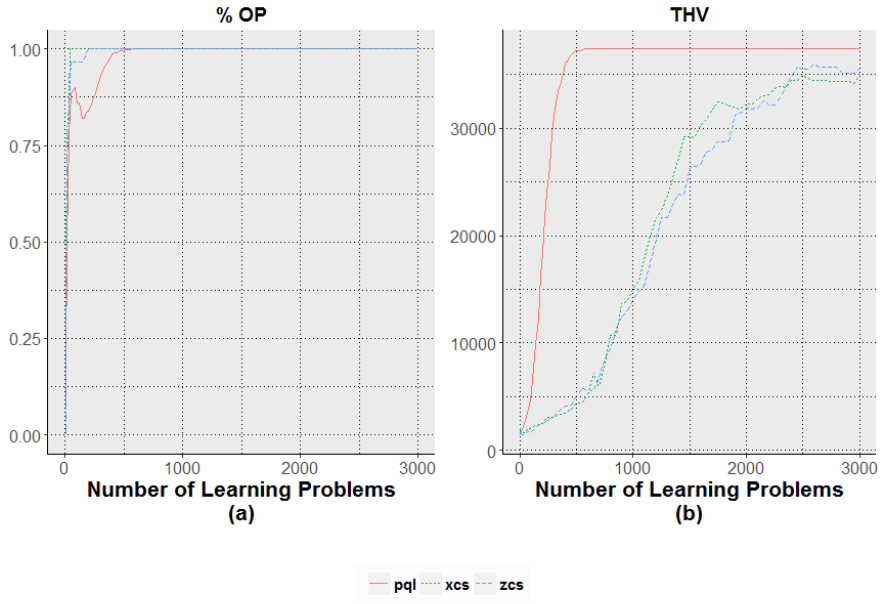


Figure 4.11: Learning performance, i.e. THV and $\%OP$, on the Deep Sea Treasure Corridor problem.

the status of the 8 squares around the current position. In this case, the DST becomes a PO-MDP problem, where there are some states that have the same inputs to make this problem more difficult to solve by LCSs. XCS has no standard mechanism to disambiguate these states.

The two integers of the current coordinate are employed as the immediate inputs. For example, the inputs of the left top corner are $(1, 1)$; the inputs of the right top corner is $(10, 1)$. In this section, Min-Max Representation (MMR) [72] is employed to handle integer inputs. In the MMR, the interval predicate is represented as (l_i, u_i) where l_i, u_i are the minimum and maximum bounds of the interval. For crossover in the run GA process, it will crossover l_i or u_i by chance; for mutation, it will randomly add or minus a small amount of l_i or u_i.

Similar to the DSTC problem, the result is divided into two parts, the Total Hypervolume (THV) of each open location and the Optimal Policy

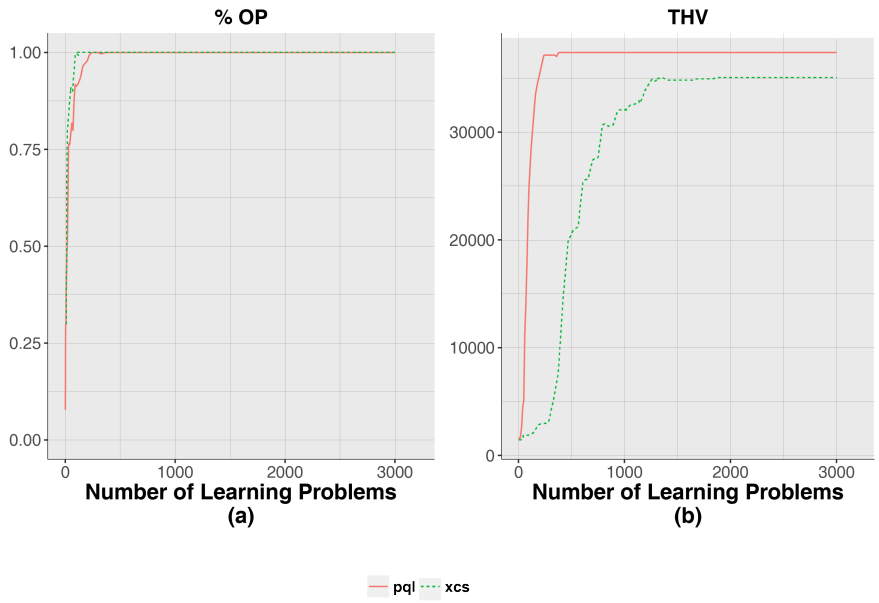


Figure 4.12: Learning performance, i.e. THV and $\%OP$, on deep sea treasure problem.

match rate ($\%OP$). In Figure 4.12, we can see that MOXCS can manage to achieve the theoretical optimal THV after learning 1200 problem instances, whereas PQL achieves it after about 400 problem instances. The THV of PQL is a little higher than MOXCS as PQL can learn more pareto optimal solutions. $\%OP$ achieves 100% by MOXCS and PQL after only 50 and 100 problem instances respectively, but both of them include many trails that only go from state (1,1) to (1,2).

4.5.3 Discussion of Pareto Front

MOXCS Versus XCS on Deep Sea Treasure Corridor

In order to compare the performance of MOXCS and XCS on solving MORL problems, both of them will be run on DSTC problem for 30 times for 3000 iterations to see what treasure can be collected by these two algorithms. One example of policies learned by XCS and MOXCS are shown in Fig-

ures 4.13 and 4.14.

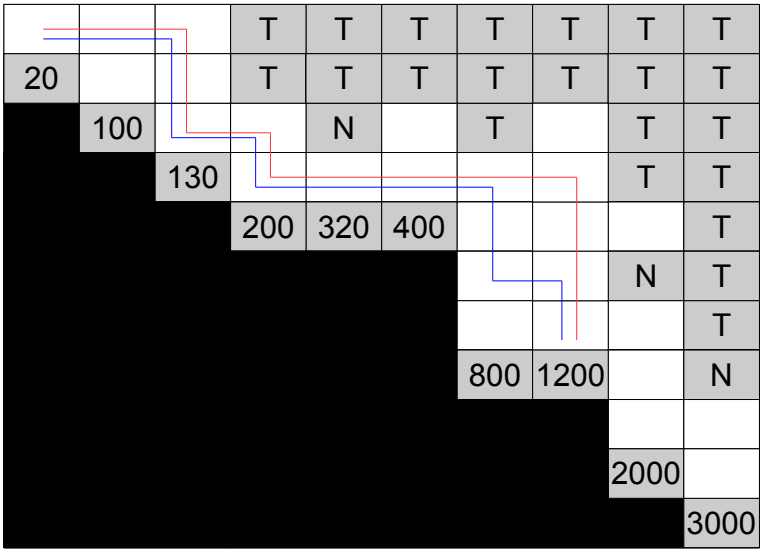


Figure 4.13: Policies learned by XCS. Two different rules shown blue and red in Deep Sea Treasure Corridor

From Figure 4.14, we can see that MOXCS can learn different policies for different weights to collect treasures: 20, 100, and 3000. Moreover, XCS learns one policy every single time and only collects treasure: 1200. In this case, it is clear that MOXCS has better performance than XCS on searching multi-policy in MORL problems.

How to Get More Treasure Locations

In Subsection 4.5.1, it is demonstrated that the Deep Sea Treasure Corridor problem can be solved by MOXCS. However, there are 10 treasures (final states) in the DSCT problem, so there are some treasures that may not be found by the agent sometimes. To improve the performance, how to find more treasure locations will be studied.

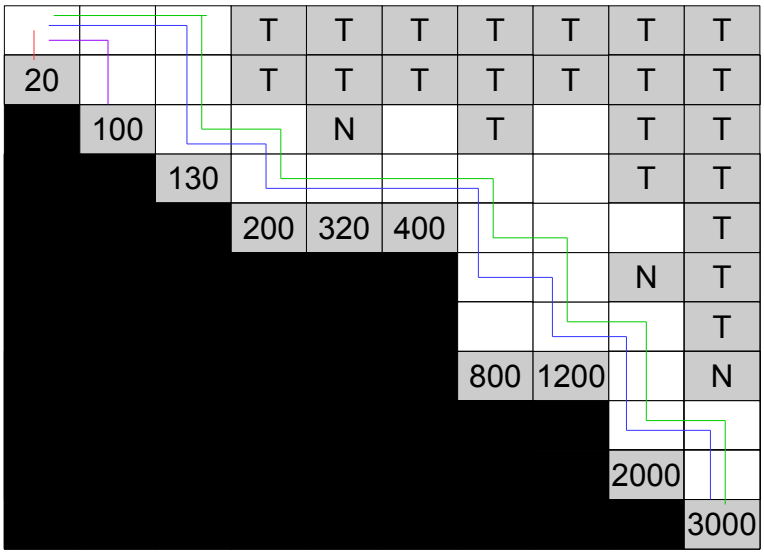


Figure 4.14: Policies learned by MOXCS. Four rules shown with different weights: red, purple, blue, green in Deep Sea Treasure Corridor

First, stopping training at the proper time may help get more treasures. In Table 4.4, it shows the treasures were collected in DSTC problem by MOXCS and MOZCS from 1000 to 10000 Learning Problems (LPs) in a single run. From the results in Table 4.4, it is clear that the treasures collected by the agent may vary with the number of learning problems. For example, the agent collects the treasures with MOXCS with value 20, 100, 1200, and 3000 at 1000 number-of-learning problems, and collects the treasures with value 20, 100, 3000 at 2000 problems.

Secondly, adjusting the weights range may help collect more treasures. The hypothesis behind it is that the agent only initializes 26 evenly distributed weights from [0, 1] to [1, 0] in the experiment, and those weights are corresponding to treasures with values: 20, 100, and 3000. For example, the relationship of treasure value and weights found in experiments is shown in Table 4.5.

Table 4.4: Treasures collected on different number of learning problems

Number of LPs	Treasures MOXCS	Treasures MOZCS
1000	20, 100, 1200, 3000	20, 100, 1200
2000	20, 100, 3000	20, 100, 3000
3000	20, 100, 2000, 3000	20, 100, 3000
4000	20, 100, 3000	20, 100, 3000
5000	20, 100, 3000	20, 100, 1200,3000
6000	20, 100, 3000	20, 100, 3000
7000	20, 100, 3000	20, 100, 3000
8000	20, 100, 3000	20, 100,2000, 3000
9000	20, 100, 3000	20, 100, 3000
10000	20, 100, 2000, 3000	20, 100, 3000

Table 4.5: Treasures collected on different weights

Treasures	Weights
3000	λ_1 less than 0.60
100	λ_1 between 0.60 and 0.70
20	λ_1 larger than 0.70

Table 4.6: Treasures collected based on more weights

Treasures	Weights
1200	$\lambda = 0.445$
130	$\lambda = 0.453$
200	$\lambda = 0.493$
800	$\lambda = 0.517$
800	$\lambda = 0.617$
800	$\lambda = 0.645$

In order to find the treasures were not found in Table 4.4, 137 weights are initialized where λ_1 between 0.431 and 0.703, which are supposed to find treasure values between 100 and 3000. The new treasures obtained from 3000 training problem instances is in Table 4.6. From the results, we can see that with proper weight settings, more treasures are possible to be found by the agent under MOXCS algorithm.

Note, the treasure 3000 is found by most weights where λ is less than 0.60. This means even initializing an evenly distributed weight in the problem space, MOXCS cannot guarantee to find a distributed solution in the solution space if the partition is too large.

PF Shapes: Deep Sea Treasure

In this subsection, MOXCS is tested with different shapes of PF to see if the Pareto policies can be learned to address different shapes of PF. The results are shown in Figures 4.15 and 4.16. In those Figures, the solutions on the Pareto front are demonstrated as orange and blue dots, where the solutions are found by MOXCS are in blue, and the solutions not found by MOXCS are in orange.

In Figure 4.15, there are some solutions on the middle range of the PF niche cannot be achieved. This is because the gradient between each treasure is too small. In this case, the agent cannot get enough information

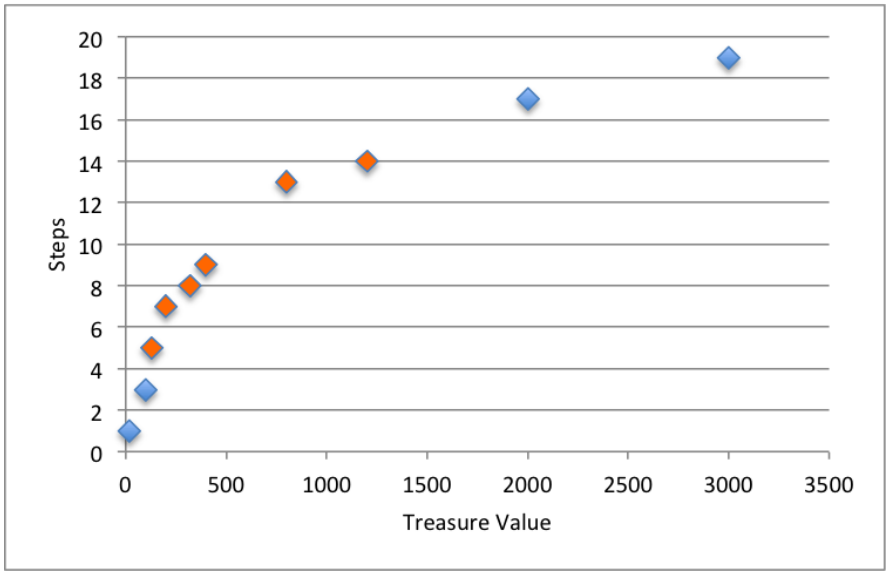


Figure 4.15: Treasures found by MOXCS on Convex PF

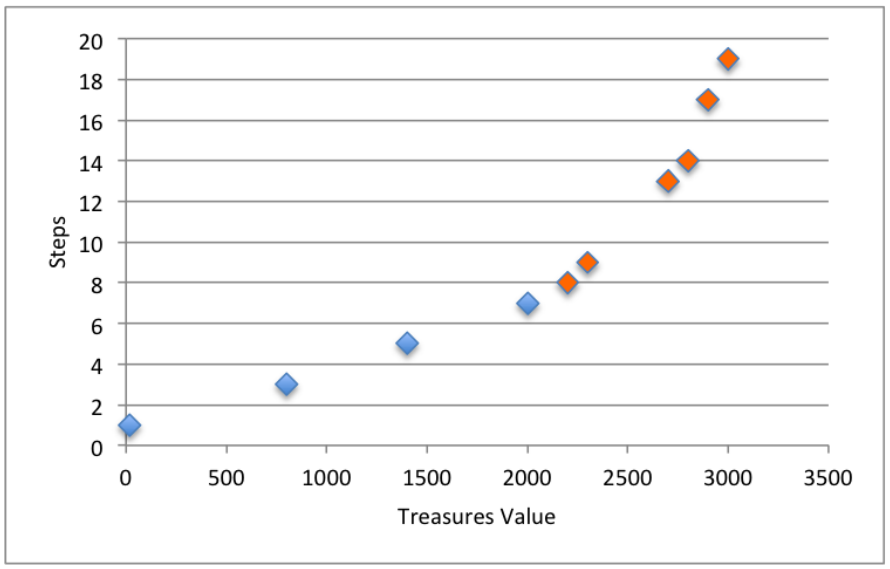


Figure 4.16: Treasures found by MOXCS on Concave PF

from learning, so it cannot find the treasures in the middle of the domain.

In Figure 4.16, we can see if the gradient between two treasures is large enough, which means that with enough information, all treasures to the

left are found by the agent. However, there are solutions to the right of the Concave PF not found, which is because the discounted rewards at the start position of those solutions are smaller than the discounted rewards from other solutions. For example, the discount rate in the experiment is 0.85, so the discounted reward of the fifth treasure at the start position is 599.4, whereas that of the fourth treasure is 641.2.

Novelty Search for PF

The fitness function normally measures the progress towards an objective in the search space as the objective function in Evolutionary Computation. However, it shows that the objective functions themselves may actively misdirect the search towards dead ends, or it might lead to a local optimum. In this case, novelty search in Evolutionary Computation is used to avoid the bias from the fitness function. Novelty search selects for "novel behavior", by some domain-dependent definition of novelty. For example, a novelty in a Maze-solving domain might be "difference of routes explored". Eventually, the networks that take every possible route through the maze will be found, and the use could then select the fastest. This would work far better than a naive "objective", like distance to the goal, which could easily result in local optima that never solves the maze.

In order to compare the performance of MOXCS with novelty search and without novelty search on MORL problems, both of the two strategies are run on a small Multi-Objective maze problem in Figure 4.17. For each algorithm, the experiment runs on three different rewards settings for 500 iterations to see what treasure can be collected by these two algorithms. The results are collected from 10 trials.

As the results in Table 4.7, there is not much difference between novelty MOXCS and without novelty search strategy, except there is 1 of 10 times that MOXCS without novelty search found the treasure of 100 when the reward setting is (20,100,180). This means the novelty search does not work very effectively in the multi-objective maze problem. This is because

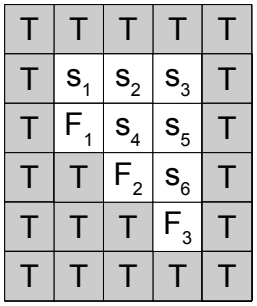


Figure 4.17: A small Multi-Objective maze problem

Table 4.7: Treasures collected by novelty MOXCS and MOXCS

Treasures	Treasures found by novelty MOXCS	Treasures found by MOXCS
20,100,130	20(10),100(10)	20(10),100(10)
20,100,180	20(10),100(0),180(10)	20(10),100(1),180(10)
20,100,150	20(10),100(10),150(10)	20(10),100(10),150(10)

novelty search strategy suits for the large space but the reward distribution is evenly.

Another of note is that the treasure of 130 was never being found when the reward setting is (20,100,130), the treasure of 100 is only found once when the reward setting is (20,100,150), which means the MOXCS algorithm is high sensitive to the reward setting. It can only work on the reward setting when the discounted reward from the further final state is smaller than that of the closer final state.

4.6 Generalization

MOXCS is a rule-based algorithm, which towards evolves accuracy with implicit and explicit pressures encouraging maximal generality. In this chapter, the multi-maze is developed to evaluate the generalization ability

of MOXCS.

4.6.1 Multi-Maze

In Figure 4.18, the multi-maze environment is a maze, which is connected by three same small bi-objective mazes. Each small bi-objective maze has 14 open states and 2 final states, thus multi-maze has 52 open states and 6 final states. The reward for F_1, F_3, F_5 is 20, the reward for F_2, F_4, F_6 is 60. Same with other problems, the agent can perform four different moves ($up, down, right, left$). Each trail starts from any open state and ends at one of the final states.

For testing the generalization ability of MOXCS, the agent is trained in one of the mazes and tested in the three same small bi-objective mazes.

Similarly, in Figure 4.19, there are two different colors used to represent the performance of each algorithm (MOXCS and PQL). Each line is the average value of the results from 30 runs for $\%OP$ and THV.

For the $\%OP$, PQL converges after 100 learning problems, whereas MOXCS reaches the optimal performance for $\%OP$ after about 80 problem instances respectively. This indicates the power of a rule-based solution, because the rule-based solution can be reused if the environment of the current state is the same as another state.

From the THV result, we can see PQL can learn the largest THV (about 94000) after 1200 learning problems. The THV learned by MOXCS converges to a good solution after only 50 learning problems, where it keeps slowly raising. There are some differences on the highest THV learned by PQL and MOXCS, which is due to the limitation of weighted sum optimization strategy of MOXCS.

4.6.2 Multi-Maze Connection

In this environment, three mazes in Figure. 4.18 are connected by adding two states S_{43} and S_{44} for testing the generalization ability of MOXCS.

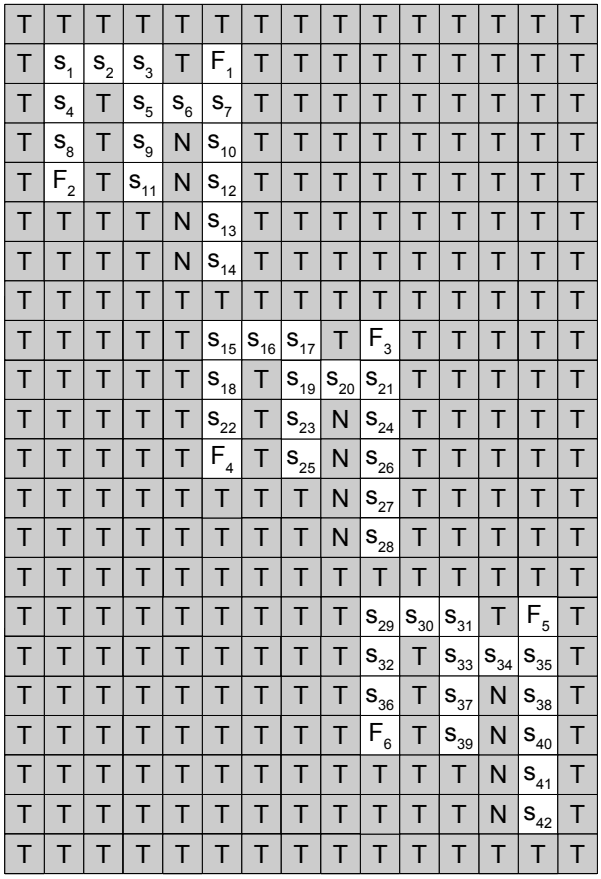


Figure 4.18: Multi-maze environment is a environment which is consisted by three same small bi-objective mazes

Similar to the previous experiment, the agent is trained in one of the mazes in Figure 4.18, but tested in the environment in Figure. 4.20.

However, the environment does not work well for testing two-objective RL algorithm. For example, when training the agent in one of the mazes in Figure 4.18, the agent learns take action up at state S_{14} to get 20 reward from final state F_1. However, when in the connected maze in Figure 4.20,

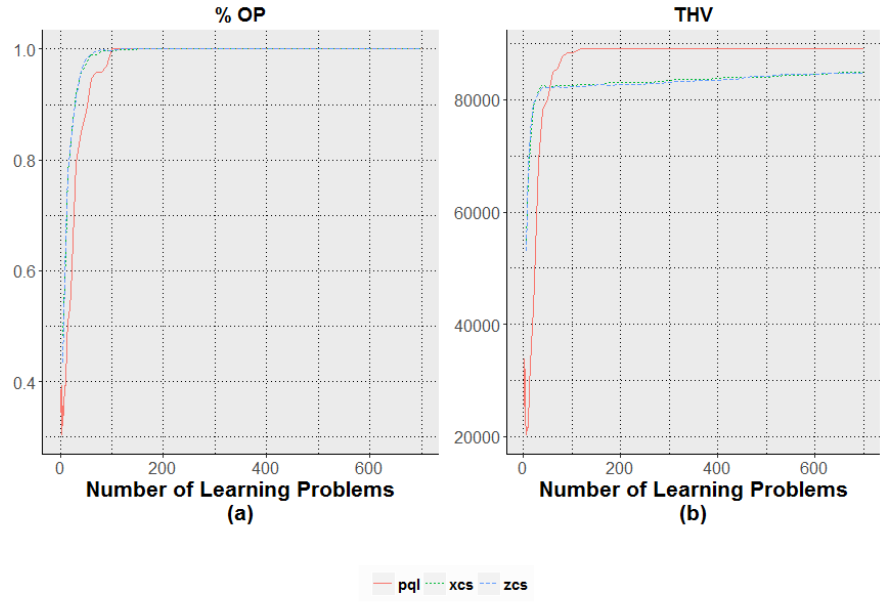


Figure 4.19: Multi-Maze results of PQL and MOXCS

the agent has to take action $down$ to get 60 reward from F_4. In this case, the testing will fail due to the action of the same state not being consistent in the training and testing session.

4.7 Chapter Summary

In this Chapter, a new LCS-based MORL algorithm MOXCS, is developed based on the hypothesis that the decomposition of MOEA/D for multi-objective tasks can sufficiently approximate the complicated PF shapes to enable such policy learning within XCS. According to the literature review, it is the first LCS-based algorithm that can solve the Deep Sea Treasure problem.

In order to test the efficiency of MOXCS for solving MORL problems, MOXCS has been tested on three bi-objective mazes. The performance has been measured by evaluating the hypervolume and policy match rate.

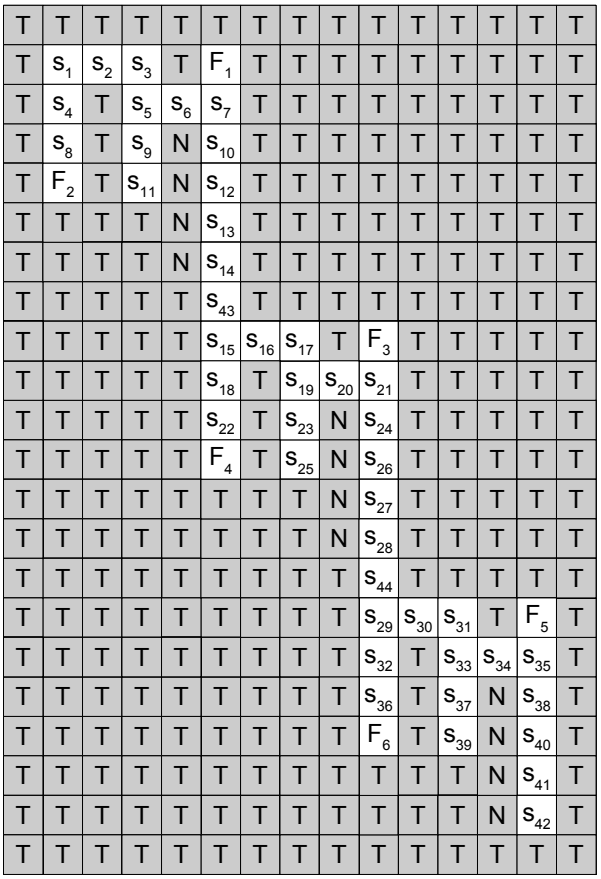


Figure 4.20: Multi-Maze Connection environment is an environment, which consists of three same small bi-objective mazes

From the results, we can see that MOXCS can solve bi-objective maze problems successfully.

MOXCS has been used to solve two MORL benchmarks, Deep Sea Treasure Corridor and Deep Sea Treasure. For the Deep Sea Treasure Corridor problem, the PO-MDP environment is converted to a MDP environment by adding extra characters. For Deep Sea Treasure problem, MOXCS

handles the PO-MDP environment by updating the condition to include the integer input of each state in the environment. The learned Pareto Front by MOXCS is discussed as well. First, it shows MOXCS is capable to learn more optimal solutions on the Pareto Front than XCS. Second, it discussed how to get more solutions on the Pareto Front, for example, stop training at a proper time, or adjust the weights settings in MOXCS. Third, MOXCS is tested to learn the solutions on different shapes of the Pareto Front. It shows MOXCS is able to find Pareto Optimal Policies with both Convex and Concave PF. Lastly, the novelty search strategy is integrated in MOXCS aims to learn more solutions on the Pareto Front. However, the experiment result shows novelty research cannot help to improve the capability of MOXCS to learn more multi-objective optimal solutions.

The generalization ability of MOXCS has been evaluated by solving the Multi-Maze, where the agent trained by MOXCS in one maze, but tested in several same mazes at the same time. The evidence from the results shows MOXCS resolves the Multi-Maze problem successfully. However, the experiment will fail when the maze in the testing environment is slightly different from the maze in the training environment. The next chapter will evaluate the generalization ability of MOXCS in a new MORL environment.

Although MOXCS is able to solve observable MORL problems, DST has limited states for the agent to explore. In chapter 6, MOXCS will be evaluated on a larger scale MORL space.

Chapter 5

Quantifying the Generalization Ability of MOXCS in Multi-Objective Reinforcement Learning problems

5.1 Introduction

In the previous chapter, MOXCS was used to solve a variety of MORL problems, such as multi-objective mazes, Deep Sea Treasure Corridor, and Deep Sea Treasure Corridor. Although the trained agents can solve those complex tasks, it is unknown how well they transfer their experience to new environments. For example, for the existing common benchmarks in the previous chapters, it is common to use the same environments for both training and testing. In this case, it is hard to evaluate the generalization ability of MORL algorithms. However, in the real world, it would be quite common that the testing environment is slightly different from the training environment. For example, when training the robot to explore open case mines, the real work environment is different from the lab environment as

it will be windy and rainy on some days.

5.1.1 Chapter Goals

The goal of this chapter is to evaluate the generalization ability of MOXCS in solving MORL problems. For achieving the goal, CoinRun is introduced, where CoinRun is a training environment that provides a metric for evaluating the agent's ability to transfer its experience to novel situations.

In this chapter, the generalization ability of XCS is tested by CoinRun. Furthermore, the CoinRun environment is updated as a bi-objective reinforcement learning environment for testing the generalization ability of MOXCS.

Three major technical issues have been identified and addressed to achieve this goal in this chapter as follows.

(1) The position of the agent in the original CoinRun game is discrediting into discrete from continuous.

(2) The PO-MDP environment is transformed to MDP by adding extra characters.

(3) A sub-actions strategy is used to enable the movement of agent obvious enough to let the XCS and MOXCS sensor it.

5.1.2 Chapter Organization

This chapter is structured as follows. Section 5.2 introduces CoinRun and multi-objective CoinRun problems as the benchmark for testing the generalization ability of RL and MORL algorithms. Next, Section 5.3 presents the methodology that uses XCS and MOXCS algorithms to solve the single-objective and multi-objective CoinRun problem. The results of using XCS to solve single-objective CoinRun and evaluate the generalization ability

of XCS are given in Section 5.4 and 5.5. While the results of using MOXCS to solve multi-objective CoinRun and evaluate the generalization ability of MOXCS are given in Sections 5.6 and 5.7 respectively. Then, the discussion is given in Section 5.8, and the chapter is finally summarized in Section 5.9.

5.2 Problem Description

5.2.1 CoinRun Problem

CoinRun is a training environment from Open AI, which provides a metric for evaluating reinforcement learning algorithms, especially for the generalization ability of reinforcement learning algorithms [17]. The issue of the most common benchmarks is using the same environment for both training and testing. CoinRun addresses this issue by generating environments to construct distinct training and test sets. Thus, the agent can be trained in the training set and tested in a slightly different environment.

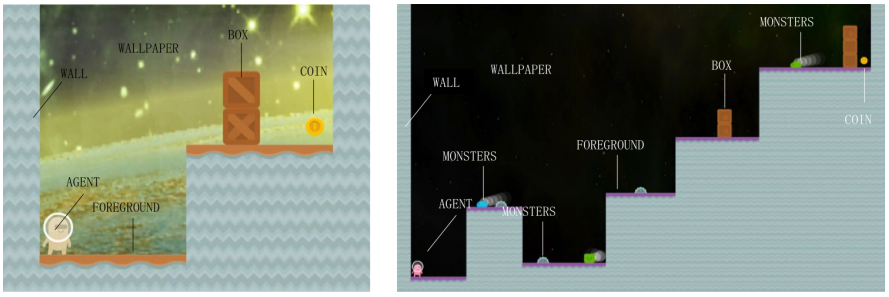


Figure 5.1: The figure is adapted from OpenAI(https://openai.com/blog/quantifying-generalization-in-reinforcement-learning/). Each level of CoinRun has a difficulty setting from 1 to 3. Two levels are displayed above: Difficulty-1 (left) and Difficulty-3 (right)

The goal of each CoinRun level is simple: collect the single coin that

Action Number	Action
a0	do nothing
a1	right
a2	left
a3	jump
a4	right-jump
a5	left-jump
a6	down

Table 5.1: Actions in CoinRun environment.

lies at the end of the level. Several obstacles, both stationary and non-stationary, are laid between the agent and the coin. There are many types of obstacles, like different monsters, that need to be avoided between the agent and the coin. As shown in Table 5.1, the agent can take seven different actions: right, left, jump, right-jump, left jump, and step down in the environment. There are different difficulty levels in CoinRun, as shown in Figure 5.1. Each level of CoinRun has a difficulty setting from 1 to 3. Two levels are displayed above: Difficulty-1 (left) and Difficulty-3 (right). The goal of the agent is to get the coin in the CoinRun environment with minimum steps.

The agent's position x and y in the environment is not only impacted by the action but it is also impacted by $gravity$ and the constraints like $max\ jump$ and $max\ speed$, thus the agent must learn not only the correct direction to the final state but also how to handle the constraints. Here are the technical settings of the CoinRun problem, which are not available to the agent.

Technical Settings

$$gravity = 0.2\ units \tag{5.1}$$

$$max\ jump = 1.5\ units \tag{5.2}$$

$$max\ speed = 0.5\ units \tag{5.3}$$

Shift Calculation

$$v_x = (random(0,1) * 2 - 1) * 0.5 * max\ speed\ units \tag{5.4}$$

$$v_y = (0.8 + 0.2 * random(0,1)) * max\ jump - gravity\ units \tag{5.5}$$

The v_x is the movement on the x-axis, and v_y is the movement on the y-axis.

In each trial of the CoinRun game, the agent starts at the left-hand side and ends when the agent collects the coin, as shown in Figure 5.1. A collision with an obstacle (not including walls, floor, and boxes) results in the agent's immediate death. The level terminates when the agent dies, the coin is collected, or after 1000 time steps. The only reward in the environment is obtained by collecting the coin, and this reward is a fixed positive constant.

In this thesis, to build up an environment for testing the generalization ability of MORL algorithms, there are two restrictions and changes are applied to the CoinRun environment.

Firstly, although the goal of this thesis in this chapter is to build up an environment based on CoinRun Environment to test the generalization ability of MORL algorithms, it is easier to start with a simple environment and explore the complicated environment in future work. In this case, two simple environments from the original CoinRun game developed by OpenAI have been selected for implementing the experiments in this chapter. The original experiments are implemented by OpenAI come across different levels of difficulties. As the implemented agents are moved with an allocentric view usually in deep NNs, which can handle large input spaces, the solution of the deep NNs is robust to different difficulties. However, it needs lots of resources, such as GPU and data to train. For LCS, normally the agent is only equipped with a local egocentric view of its state, but it can learn the solution for a specific situation with limited inputs on the situation, and the solution can generalize to another similar situation. In this case, to test the generalization ability of LCS and avoid its limitation

of limited inputs, the training and testing environments for LCS should be selected properly. For example, in Difficulty-3 environment on the right-hand side in Figure 5.1, there are many moving obstacles, such as monsters, that are relatively fast-moving, if the agent would not sense them before a collision that would end learning. As MOXCS is equipped with a local egocentric view of its state, it is hard for it to sense the moving objects in time. Thus, rather than test the MOXCS algorithm on the Difficulty-3 environment, the Difficulty-1 environment on the left-hand side will be considered. In this case, only two Difficulty-1 CoinRun Environments, Environment 1 (Env1) and CoinRun Environment 5 (Env5), are employed as is shown in Figures 5.2 and 5.3. In these two environments, the agent needs to find the shortcut to get the coin, but there are no monsters in these environments.

Secondly, the actions that are affected need to be operated over a group of time steps to enable the movement of the agent to be larger than one unit, so LCS can sensor the difference of the inputs. In this case, the agent cannot decide an action at each time step, so may not be able to make the most effective action in some states. In this case, instead of terminating after 1000 time steps in the original setting, each trail terminates after 20000-time steps in the training process of each experiment. Therefore, the time step is increased to give the agent more time to learn to solve the problem. However, as observed, if the agent has learned the optimal policy to collect the coin, then the agent can collect the coin within 90-time steps, or if the agent has learned the potential optimal policy to collect the coin, then the agent should be able to collect the coin within 180-time steps. In this case, to evaluate if the agent has learned the optimal policy or a potential optimal policy, each trial terminates after 900-time steps in the testing process.

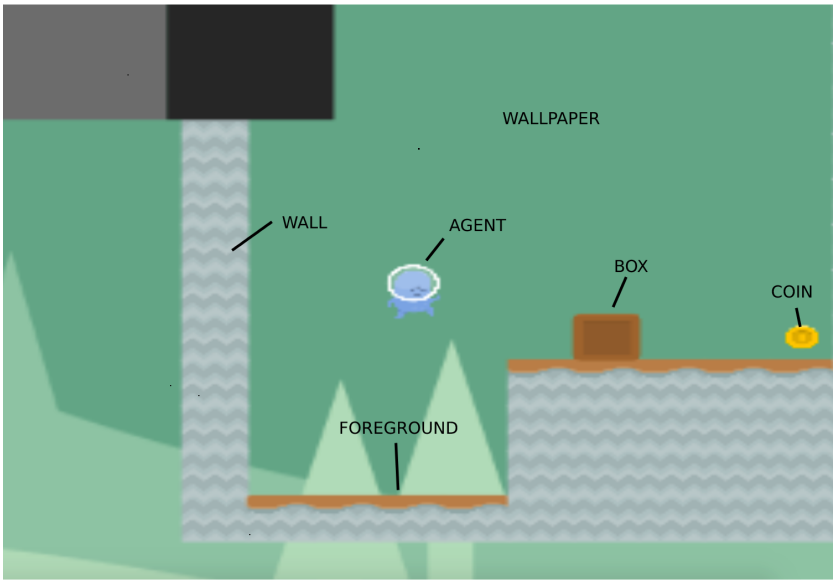


Figure 5.2: CoinRun Environment 1 (Env1).

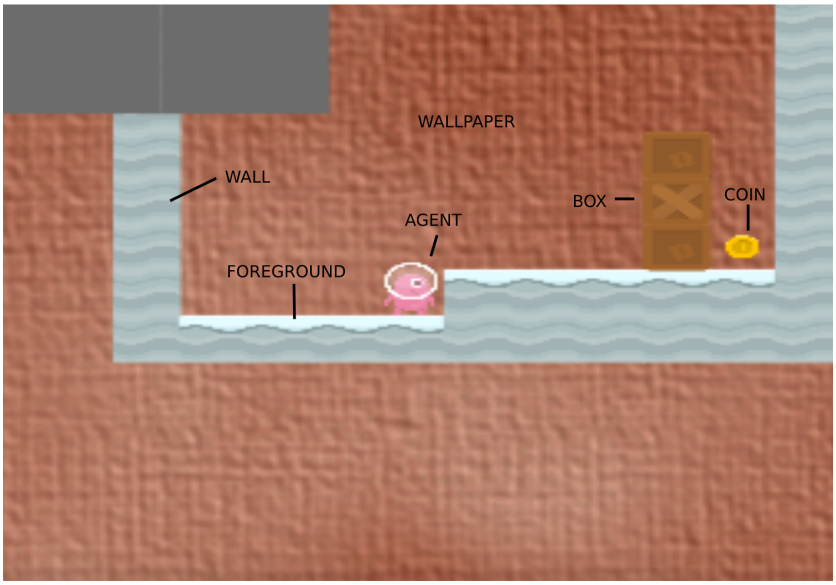


Figure 5.3: CoinRun Environment 5 (Env5).

5.2.2 Multi-Objective CoinRun

For evaluating the MORL algorithm, the second objective is added in CoinRun Env1 and Env5; thus, two bi-objective reinforcement learning experiments, 'CoinRun Action Bias' and 'CoinRun Step VS Reward' are designed.

CoinRun Action Bias

In the CoinRun environment, there are two optimal solutions for the agent to get the coin. The first optimal solution is to get the coin on the right-hand side by taking as much as action *right-jump*. The second optimal solution is to get the coin by maximizing the long-term payoff. In this case, to encourage the agent to choose the first solution, another objective is added to push the agent to get the coin by taking more action right-jump. This experiment is called 'CoinRun Action Bias'. There are two objectives in 'CoinRun Action Bias'. The first objective is encouraging the agent to take as much as the action *right jump* to get the coin, and the second objective is minimizing the steps to get the coin. In this bi-objective environment, it will return a reward of 150 if the agent takes an action *right jump* and returns a reward of 150 when the agent gets the coin for the first objective, and the agent will get a reward of 1000 when it gets the coin for the second objective. For the first objective, both strategies reach a coin, but the policies are conflicting at the last step, where one is to reach the coin as soon as possible, the other account policy takes as many as right jumps. However, it would not affect the result much when the agent is close enough to the coin at the last step, because no matter whether the agent takes action go right or right jump, it will get the coin. The action bias is not only useful in a bi-objective MORL testing environment but also can be applied in practice. For example, it can be used to train a robot to complete a certain task with actions that have a low power consumption.

CoinRun Step VS Reward

Another bi-objective experiment derived from CoinRun is called 'CoinRun Step VS Reward'. In this environment, another coin is added in the CoinRun Environment where the coin is closer to the start state but has less reward. In this case, there are two objectives in 'CoinRun Step VS Reward', the first objective is collecting the coin with a large reward, and the second objective is collecting the coin with fewer steps. The first coin that is closer to the start state has a reward of 30, and the second coin that is away from the start state has a reward of 60. Thus, steps and rewards are conflicting.

5.3 Techniques Used to Solve CoinRun

The CoinRun problem is originally a continuous space for testing agents with allocentric knowledge of the domain. However, XCS and MOXCS can only solve the problem in the discrete space with an egocentric state representation. Thus, several changes need to be made to XCS and MOXCS for solving the CoinRun problem, where the existing technique for discretizing continuous input has been used and the sub-action technique for transforming PD-MDP to MDP is developed.

Note, XCS has been evaluated in solving single-objective CoinRun in Section 5.4 and Single Objective CoinRun Generalization problem in Section 5.5, the reasons are:

(1) Solving single-objective CoinRun works as a stepping stone for developing multi-objective approaches.

(2) Solving Single Objective CoinRun Generalization problem works as a stepping stone for evaluating the generalization ability of multi-objective approaches.

(3) Solving both of these two problems can help investigate issues that

occur during developing multi-objective approaches and evaluating the generalization ability of multi-objective approaches as it provides a baseline of only setting one objective in the MORL problem.

(4) Solving both of these two problems work as benchmarks for comparison purpose: The single-objective problems tested by XCS has been transferred as one of the multi-objective in the MORL problems (Multi-Objective CoinRun and Multi-Objective CoinRun Generalization) for evaluating the effectiveness and generalization ability of MOXCS in the later research. In this case, it is able to identify how the added/new objective in the multi-objective environment affects the performance of the original objective (the single-objective tested by XCS) in the MORL problem.

5.3.1 Discretizing Continuous Input

The original CoinRun environment can be divided into two parts: the background (e.g., walls, boxes, platform) and the agent.

Background

A	.	.	.	.	.	.	.	.	.	A
A	.	.	.	.	.	.	.	.	.	A
A	.	.	.	.	.	%	.	.	1	A
A	.	.	.	.	S	S	S	S	S	A
A	.	.	.	.	A	A	A	A	A	A
A	.	.	.	.	A	A	A	A	A	A
A	S	S	S	S	A	A	A	A	A	A

Figure 5.4: Character Coding of CoinRun Env1.

A	.	.	.	.	.	.	.	.	.	A
A	.	.	.	.	.	.	.	.	.	A
A	.	.	.	.	.	.	.	%	.	A
A	.	.	.	.	.	.	.	$	.	A
A	.	.	.	.	.	.	.	%	1	A
A	.	.	.	.	S	S	S	S	S	A
A	S	S	S	S	A	A	A	A	A	A

Figure 5.5: Character Coding of CoinRun Env5.

The first part is a set of characters that can be used to present the background in CoinRun as shown in Figure 5.4 and Figure 5.5. For instance,

the space is represented by '.', a wall is represented by 'A', the surface is represented by 'S', a box is represented by '$\%$', and the coin is represented by 1. Then the program will render the characters into images. Furthermore, there are colorful CoinRun environments in Env1 and Env5 in Figures 5.2 and 5.3.

Agent

The second part is the agent. The agent position is represented by two floats x and y as described in the shift calculation in Section 5.2.1 that governs how the agent moves. However, XCS and MOXCS are not able to encode the continuous float input, so the discrete float input will be rounded up to the nearest integer. For example, as is shown in Figure. 5.6, if the agent's position is $x = [0, 1)$ and $y = [0, 1)$, it will be considered that the agent is at the state S_1, and the condition of S_1 is consists of the 8 characters surrounding it with an order from the top to bottom by row then left to right within each row, which is '$A..A.ASS$'. Similarly, if the agent's position is $x = [1, 2)$ and $y = [0, 1)$, it will be considered the agent is at the state S_2, and the condition is '$.....SSS$'.

5.3.2 PO-MDP Environment

A	.	.	.	.	.	.	.	.	.	A
A	.	.	.	.	.	.	.	.	.	A
A	.	.	.	.	.	%	.	.	1	A
A	.	.	.	.	S	S	S	S	S	A
A	.	.	.	.	A	A	A	A	A	A
A	S_1	S_2	S_3	S_4	A	A	A	A	A	A
A	S	S	S	S	A	A	A	A	A	A

Figure 5.6: Discrete environment representation in CoinRun Env1.

A	.	.	.	.	.	.	.	.	.	A
A	.	.	.	.	.	.	.	.	.	A
A	.	.	.	.	.	.	.	%	.	A
A	.	.	.	.	.	.	.	$	.	A
A	.	.	.	.	.	.	.	%	1	A
A	S_1	S_2	S_3	S_4	S	S	S	S	S	A
A	S	S	S	S	A	A	A	A	A	A

Figure 5.7: Discrete environment representation in CoinRun Env5.

There are many aliased states in CoinRun Env1 and Env5, where some positions have the same condition to make them a PO-MDP problem. To

demonstrate the problem clearly, the first example will be S_2 and S_3 in CoinRun Env1 and Env5 as shown in Figures 5.6 and 5.7. If the agent is at state S_2 or S_3 in both Figure 5.6 and Figure 5.7, the input string would be '.....SSS' for both state S_2 and S_3. However, at state S_2, the agent has to take action *go right*, whereas at state S_3, the agent has to take action *right-jump*. In this case, a simple Learning Classifier System could not solve the problem, as it could not tell what is the best action for the current state. As introduced in Section 2, such aliased states create a PO-MDP problem.

A	.	.	.	.	.	.	.	.	.	A
A	.	.	.	.	.	.	.	.	.	A
A	.	.	.	T	.	N	.	.	1	A
A	T	.	.	.	S	S	T	S	S	A
A	.	.	.	.	A	A	A	A	A	A
A	S1	S2	S3	S4	A	A	A	A	A	A
A	S	S	T	S	A	A	A	A	A	A

Figure 5.8: Removing Aliasing in CoinRun Environment 1.

To remove aliasing, new characters T and N are added in the environment as shown in Figure 5.8 and Figure 5.9. As a result, the input string for state S_2 is '.....SST', whereas the input string for state S_3 is '.....STS'. Similar to the states S_2 and S_3, other states have a Non-Markov property; in this case, T and N are added in the character CoinRun environment to resolve the non-Markov issue. To be noted, there are still some states that have the same condition in Figure 5.8 and Figure 5.9; for example, the condition for many states on the top of the environment is '.....'. However, the action for those states with input '.....' is consistent, which should be

taking action *right* or *right-jump*; thus, there is no need to distinguish those states.

A	.	.	.	.	.	.	.	.	.	A
A	.	.	.	.	.	.	.	.	.	A
A	.	.	.	T	.	N	.	%	.	A
A	T	.	.	.	.	.	T	$	.	A
A	.	.	.	.	.	.	.	%	1	A
A	S_1	S_2	S_3	S_4	S	S	S	S	S	A
A	S	S	T	S	A	A	A	A	A	A

Figure 5.9: Solving Non-Aliased problem in CoinRun Environment 5.

5.3.3 Sub-action Strategy

Besides the discrete-continuous inputs and aliased environment, another challenge is that the movement of the agent in one step is not obvious enough to be detected by the Learning Classifier System.

For example, the agent is supposed to learn to take action *right-jump* from state S_4 (see Figure 5.6 and 5.7). However, sometimes after the agent took action *right-jump* from state S_4, it still stays at S_4. This is because, in those discrete environments, one character represents a distance of 1 unit, however, according to the technical details in equation 5.2.1 in Section 5.2.1, page 127, the v_x and v_y could be less than 1 unit. In this case, even the agent makes a correct action and moves in the correct direction; it will stay in the same state. This eventually makes the domain Non-Markov as there is a probability distribution for the state transition, which the agent cannot learn as it depends on other previous states or actions.

A sub-action strategy is designed to address this problem. With the sub-action strategy, when taking one action a, the agent will take several *following actions* to ensure that the distance of movement is larger than 1. In the *following actions*, the first n_1 actions are the same with action a so that the action has enough impact on the agent's movement. Then the agent will take another n_2 actions *do nothing*. This will give the agent a window to enable the action a to make a difference to its position in the CoinRun environment and simulate the inertia on the agent for any action. With such a sub-action strategy, the environment is transformed from a Non-Markov problem to a Markov problem.

5.4 Single Objective CoinRun

To test the performance of XCS on solving CoinRun problems, XCS is implemented on CoinRun Env1 in Figure 5.10 and Env5 in Figure 5.11. For example, 1 represents the coin, $S_{\%}$ represents box can walk through, $S_{\$}$ represents a box cannot walk through.

A	S_{29}	S_{30}	S_{31}	S_{32}	S_{33}	S_{34}	S_{35}	S_{36}	S_{37}	A
A	S_{20}	S_{21}	S_{22}	S_{23}	S_{24}	S_{25}	S_{26}	S_{27}	S_{28}	A
A	S_{13}	S_{14}	S_{15}	S_{16}	S_{17}	$S_{\%}$	S_{18}	S_{19}	1	A
A	S_9	S_{10}	S_{11}	S_{12}	S	S	S	S	S	A
A	S_5	S_6	S_7	S_8	A	A	A	A	A	A
A	S_1	S_2	S_3	S_4	A	A	A	A	A	A
A	S	S	S	S	A	A	A	A	A	A

Figure 5.10: States of CoinRun Environment 1(Env1).

A	S37	S38	S39	S40	S41	S42	S43	S44	S45	A
A	S28	S29	S30	S31	S32	S33	S34	S35	S36	A
A	S20	S21	S22	S23	S24	S25	S26	S%2	S27	A
A	S12	S13	S14	S15	S16	S17	S18	S$	S19	A
A	S5	S6	S7	S8	S9	S10	S11	S%1	1	A
A	S1	S2	S3	S4	A	A	A	A	A	A
A	S	S	S	S	A	A	A	A	A	A

Figure 5.11: States of CoinRun Environment 5(Env5).

5.4.1 Experiment Settings

All results in this chapter are calculated by averaging the results of 30 independent runs of XCS for single objective experiments and MOXCS for multi-objective experiments. In the experiments in subsection 5.4, 5.5, 5.6 and 5.7, most of the typical parameter settings recommended in [14] have been followed. Particularly, the number of bits in the state is 24, the number of actions and the minimum number of elements in the match set in this system is 7, the probability of the system exploring the environment is 0.5, α and nu which are used to update the classifier's fitness is 0.1 and 5, the learning rate for prediction, error, and fitness β is 0.2, the classifier deleting threshold $\theta_{del} = 200$, the threshold of average fitness for calculating the deletion vote of a classifier in the system δ is 0.1, the subsumption threshold $\theta_{sub} = 20$, crossover rate in GA is 0.8.

Three parameter settings were added in the experiments in this chapter. First, a parameter ifG is added, which denotes if the system runs in the generalization environments (i.e. different training and test environments), and it is highly related to the deletion process. When it is false,

Table 5.2: Optimal results achieved manually in Env1 and Env5

Env	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	Avg
env1	66	54	53	68	75	55	52	70	65	53	61.1
env5	63	60	60	60	70	59	63	63	61	70	62.9

namely, the system runs in the same environment. In this case, the classifier will be decoded to measure the influence on the population. Thus, it will only delete the classifier if any action set is still larger than one after deleting the classifier. When it is true, as the $P_{\#}$ in the generalization environment is high, it is costly to decode the condition. In this case, it will only reduce the numerosity of choice in the deletion process. Second, a parameter ch_a is added; where if it is set to true, the error and fitness of the child in GA are equal to the initial settings of the error and fitness. Third, a parameter ch_c is added; if it is true, the error and fitness of the child in GA are equal to the initial settings of the error and fitness when the crossover is applied in GA.

In the experiments in this section, the following parameters are set in both CoinRun Env1 and Env5. For example, the threshold for performing the niche GA θ_{ga} is 2000, the payoff decay rate $gamma$ is 0.9, the probability of generating a hash in a condition $P_{\#}$ is 0.18, the mutation rate μ is 0.04, the prediction p, error e and fitness f are 20, 0.001 and 10, as ch_a is true, ch_c would not affect child's error and fitness. In addition, the population size N is 8,000,000 and ifG is false in Env1, but the population size N is 800,000 and ifG is true in Env5.

5.4.2 Optimal Solution

To estimate the optimal solution that minimizes the steps to get the coin in both CoinRun Environment Env1 and Env5 in Figures. 5.2 and 5.3, the game is practiced and then played ten times manually.

The steps that are manually played are counted and shown in Table

5.2, and the average steps that minimize the steps to reach the coin are 61.1 and 62.9 in CoinRun Env1 and Env5, respectively. In this chapter, the solution that minimizes the steps to reach the coin in each experiment by XCS or MOXCS that is similar to or less than 61.1 steps in Env1 and 62.9 steps in Env5 is considered as an optimal solution. As only the main steps are counted in the result of the experiment in this chapter, and one main step consists of 9 steps, the solution with 6 to 7 main steps is considered as an optimal solution that minimizes the steps to reach the coin in CoinRun Env1 and Env5.

5.4.3 Experiment Result

CoinRun Env1.

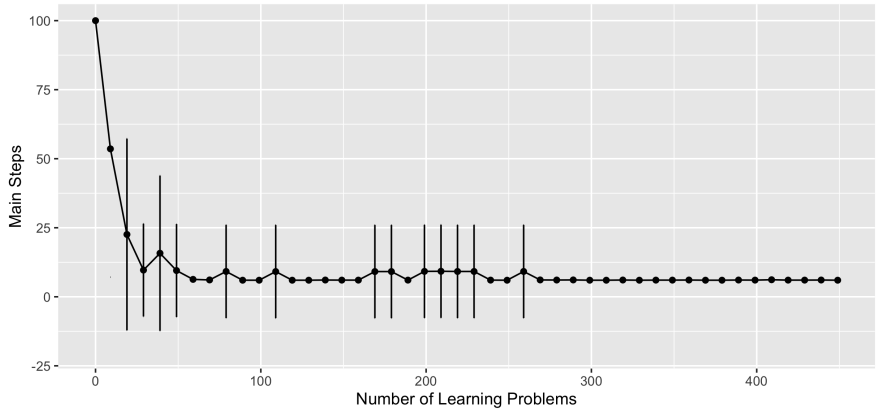


Figure 5.12: Single Objective CoinRun Environment 1(Env1).

The experiment is first conducted on CoinRun Env1 with XCS. As clearly evidenced in Figure 5.12, XCS can solve the problem after learning through a small number of problem instances. Particularly, the number of steps for solving the problem in Env1 drops from 100 to 9.6 main steps within 30 learning problems rapidly and achieves the best performance of 6.0 main steps (54 steps), which can be considered as the optimal solution, as it is

better than the manual result (61.1 steps). The performance fluctuates from 100 to 260 learning problems and keeps stable after that.

CoinRun Env5.

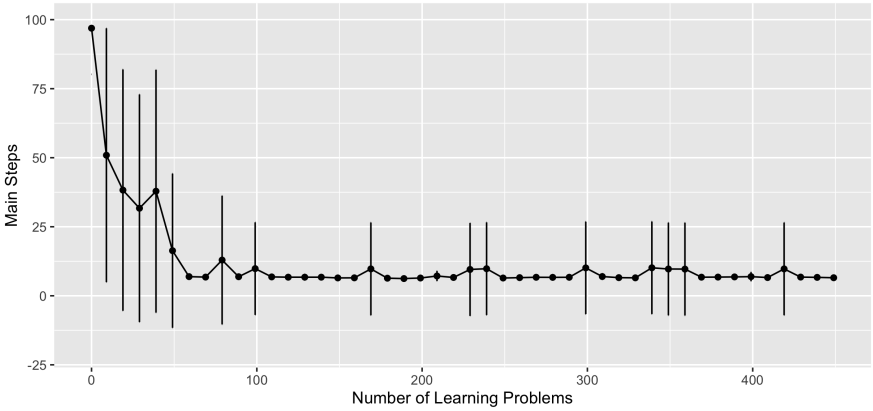


Figure 5.13: Single Objective CoinRun Environment 5(Env5).

The result in Figure 5.13 demonstrates that XCS can solve the RL problem in CoinRun Env5. Similar to Env1, the number of steps for solving the problem drops dramatically within 100 learning problems, then becomes stable. Particularly, the steps for solving the problem in Env5 drop from 96.9 to 6.7 at 70 learning problems. The best performance is 6.23 main steps (equals or less than 57-time steps) at 190 learning problems, which is better than the manual result (62.9 steps). However, the result fluctuates slightly after convergence as classifiers lack training, new classifiers, or over general classifiers with low accuracy affecting the long-term prediction at some states.

5.5 Single Objective CoinRun Generalization

The CoinRun environment can be used to measure how successful algorithms generalize from a given set of training environments to an unseen

set of testing environments. In this section, Env1 and Env5 are used as training and testing environments, and then respectively reversed, namely, uses Env5 as the training environment and use Env1 as the testing environment.

As shown in Figures 5.10 and 5.11, the main difference between training and testing environments is the height on the right-hand side of S_4, where the platform on the right-hand side of S_4 is two layers higher in Env1 than that in Env5. Therefore, the gap between the train and test performance determines if the agent learns to jump over to a platform with different heights.

5.5.1 Experiment Settings

In this section, the experiment settings are mostly the same as the settings in 5.4.1 including the parameters of XCS and the rules for collecting results. Some parameters that different from 5.4.1 are listed. When doing the cross-training and testing in both environments, the population size N is 8000, the threshold for performing the niche GA θ_{ga} is 200000, the payoff decay rate γ is 0.93, and fitness f is 0.0005. The mutation rate μ is 0.05, and the probability of generating a hash in a condition $P_{\#}$ is 0.7 when training in Env1 and testing in Env5. μ is 0.04, and $P_{\#}$ is 0.79 when training in Env5 and testing in Env1.

5.5.2 Experiment Result

Training in Env1 and testing in Env5

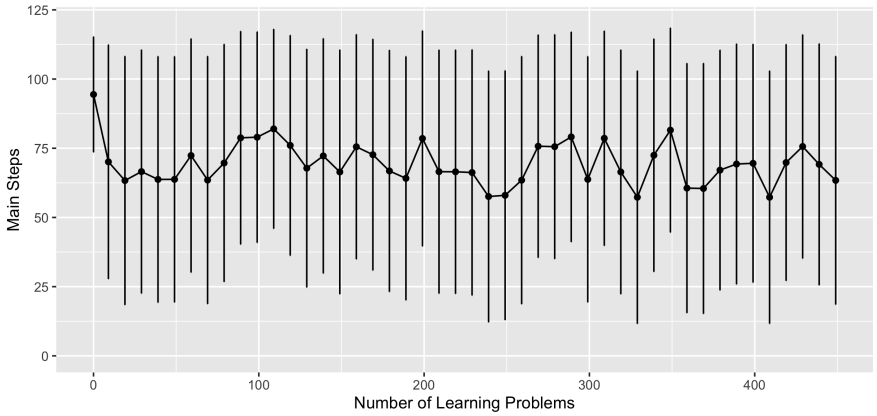


Figure 5.14: Learning performance of training in Environment 1 and testing in Environment 5, i.e. the average steps to the coin.

The agent is trained to play CoinRun in Env1 with XCS, and then it is tested in Env5. From the result in Figure 5.14, it can be seen that when the agent is trained with XCS in Env1, it has the potential to solve the problem in Env5. The main steps for solving the problem in Env5 drop dramatically at the beginning and then fluctuate in a range. More details, within 20 learning problems, which drops from 94.4 to 63.3. However, after that, the main steps fluctuate in a range between 63.3 to 79 most of the time. But it achieves the best performance of 57.3 at 330 and 410 learning problems and jumps to 81.5 at 350 learning problems.

Training in Env5 and testing in Env1

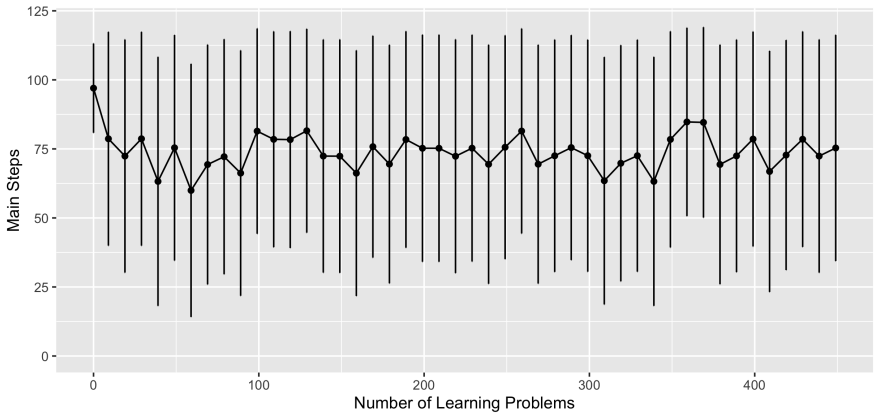


Figure 5.15: Learning performance of training in Env5 and testing on Env1, i.e. the average steps to the coin.

In this section, the agent is trained with XCS in Env5, then tested in Env1. As evidenced in Figure 5.15, it shows that the agent trained with XCS in Env5 has the potential to solve the problem in Env1. Similar to the result in Figure 5.14, the main steps for solving the problem in Env5 drop dramatically at the beginning, then fluctuate in a range. More details, the main steps for solving the problem in Env1 drop dramatically within 60 learning problems from 97 to 59.9 with fluctuation, where 59.9 is the best performance. Then, the main steps fluctuate in a range between 66.2 to 81.5 most of the time. However, it achieves the best performance of 63.2 at 310 learning problems and jumps 84.7 at 360 learning problems.

Compare the results from training and testing in the same environment in Figures 5.12 and 5.13 and in the different environments in Figures 5.14 and 5.15, we can see that XCS can solve the CoinRun problem when training and testing in the same environment, and has the potential to solve the generalization problem. However, there are still some issues that have not been addressed for solving the generalization problem, thus the performance of the generalization is not as good as training and testing in

the same environment. The low performance of generalization may be due to there is lack of a match set for some states or new classifiers and over general classifiers with low accuracy have a negative influence on the long-term prediction.

Although XCS can solve Env1 and Env5 successfully, there are still some test domains that cannot be solved due to the sub-action methodology 5.3.3. For example, there is a specific requirement on the agent's position on the right-hand side of Figure 5.1 as the agent has to jump to the left-hand side corner of the first higher foreground to avoid the monster. However, it will be difficult and sometimes impossible for the agent to achieve this. This is because the movement of v_x and v_y is larger than 1 unit with the sub-action strategy. Thus, the agent may jump over the left-hand side corner of the first higher foreground and collide with the monster. In this case, the experiments in this chapter will focus on the environment without monsters, but the experiments in the environment with monsters will be explored in future work.

5.6 Multi-Objective CoinRun

5.6.1 CoinRun Action Bias

Experiment Design

In the previous experiment in Section 5.4.3, there are different optimal policies to solve the problem in Env1. Those optimal policies have the same total number of steps but with different actions, as shown in Table 5.3.

Policies	a0	a1	a2	a3	a4	a5	a6	Total Steps
1	0	4	0	0	1	1	0	6
2	0	4	0	0	2	0	0	6
3	0	5	0	1	0	0	0	6

Table 5.3: Equal number of total steps can arise from different actions combinations.

From the Table 5.3, if the agent attempts to find the shortest route from state S_1 to get the coin at the final state, it needs to take more actions related to go right and jump, like actions a_1, a_3, a_4, a_5. However, different actions may lead to different optimal policies. In this case, to see how the actions lead to different optimal policies, another objective is added to force the agent to take more action a_4 *right-jump*. Thus, there are two objectives in the CoinRun environment. The first objective is to get the coin with actions a_4 *right-jump* as many as possible, for each action a_4 it will get a reward of 150 and get a reward of 150, at the final state. The second objective is to minimize the steps to get the coin with a reward of 1000.

Experiment Settings

To test the performance of MOXCS on solving MORL problems, MOXCS is implemented on multi-objective CoinRun Env1 and Env5. Both results of these two environments will be calculated by averaging the results of 30 independent runs of MOXCS.

In the experiments of this section, the typical parameter settings [14] have been followed. The experiment settings are mostly the same as the settings in Section 5.4.1 including the parameters of XCS and the rules for collecting results. The following parameters are different: In both training and testing in Env1 and Env5, the population size N is 80,000, the threshold for performing the niche GA θ_{ga} is 20,000, the error, fitness, prediction

for both objective1 and objective2 are 20, 0.001 and 10, and ifG is true. Note, the population size N is higher than normal settings to decrease the chance that a useful classifier is deleted. Finally, the maximum number of steps for the testing process is 500 as the objective is related to taking as much *right-jump* with weights [0.5, 0.5] and [0, 1].

In the 'CoinRun Action Bias' experiments, the agent is trained and tested by MOXCS with three different weights of two objectives in the environment, the first and last weights can test the two objectives directly, while the weight at the middle can test how these two objectives influence the learned policy at the same time. In the figures of results, there are three different colors in Figures 5.16 and 5.17 used to represent the different weights. Each line is the average value of performance, namely, the steps for getting the coin.

Results and Analysis

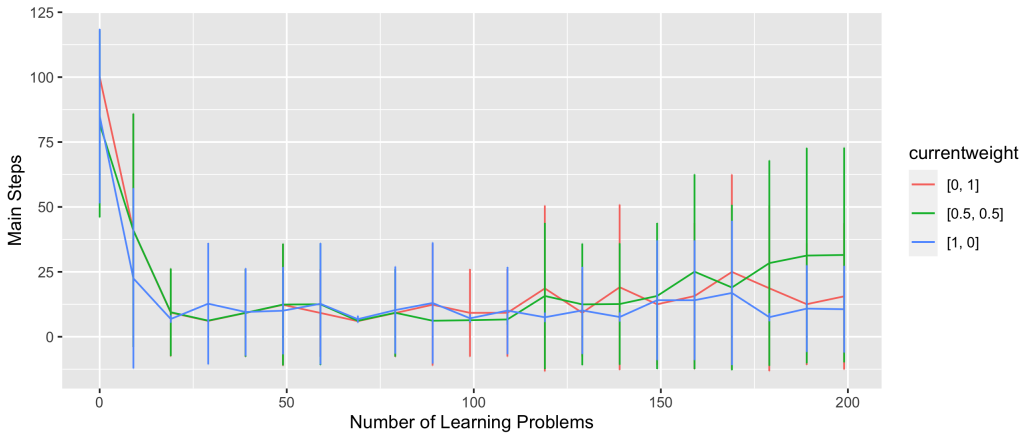


Figure 5.16: Learning performance, i.e. The average steps to get the coin in CoinRun Environment 1 (Env1).

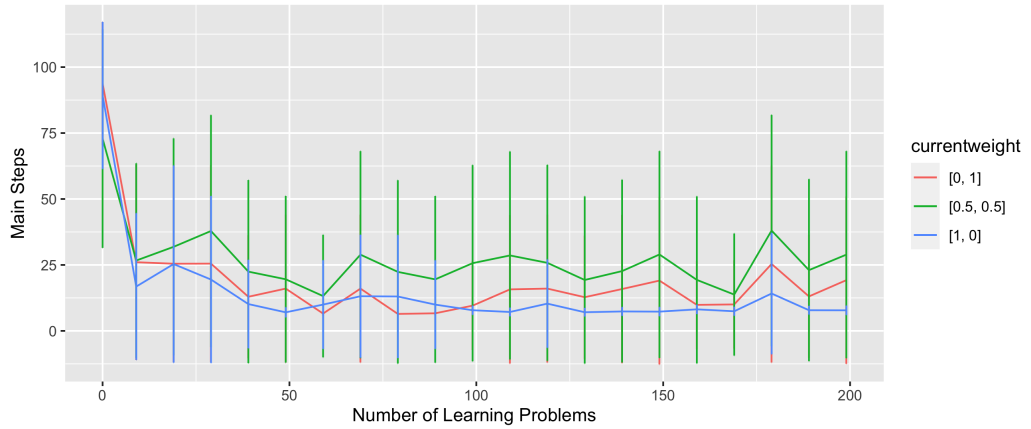


Figure 5.17: Learning performance, i.e. the average steps on CoinRun Environment 5 (Env5).

It can be seen in Figure 5.16, MOXCS is managed to solve the problem with three weights in Env1. The number of main steps for solving the problem is first starting from 100, 82, 85 and drop to 9.3, 9.4, 6.8 at 20 learning problems with weights [0, 1], [0.5, 0.5] and [1, 0] respectively. At 70 learning problems, the performance of those three weights [0, 1], [0.5, 0.5] and [1, 0] are most similar, which are 6.0, 6.1 and 6.7. However, after that, those results increase slightly before 170 learning problems, then there is a drop with weights [0, 1] and [1, 0], but an increase of weight [0.5, 0.5]. Finally, the performance of those three weights [0, 1], [0.5, 0.5] and [1, 0] is 15.5, 31.5 and 10.6 respectively. In this case, we can see weight [1, 0] converges best, while the weight [0.5, 0.5] converges worst than other weights. It may be because weight [0.5, 0.5] was influenced by two weights, thus losing the focus. The best performance to solve problem in Env1 is 6.1, 6.1, 6.8 main steps for weights [0, 1], [0.5, 0.5] and [1, 0] respectively.

It can be seen in Figure 5.17, MOXCS is managed to solve the problem with three weights in Env5. In most cases, the performance of weight [1, 0] is better than [0, 1], and the last is [0.5, 0.5]. With weights [0, 1], [0.5, 0.5] and [1, 0], the number of main steps for solving the problem drops from 93.76 to 6.5, from 72.9 to 13.2, from 89.2 to 9.96 within 60 learning prob-

lems. After that, they fluctuate in a range but are more serious than that in Env1. With weight [0, 1], it fluctuates between 6.4 to 19.2 but reaches 25.3 at 180 learning problems. With weight [0.5, 0.5], it fluctuates between 13.7 to 28.9 but reaches 38 at 180 learning problems. With weight [1, 0], it fluctuates between 7 and 14.1 steps. Similar to Env1, weight [1, 0] converges best, while the weight [0.5, 0.5] converges worst than other weights may due to weight [0.5, 0.5] being influenced by two weights, thus losing the focus. The best performance to solve problem in Env5 is 6.4, 13.2, 7.0 main steps for weights [0, 1], [0.5, 0.5] and [1, 0] respectively.

As discussed in Section 5.6.1, MOXCS can solve the 'CoinRun Action Bias' problems when training and testing the agent in the same environment. The performance difference of those three weights in Env1 is less than that in Env5, especially before 170 learning problems.

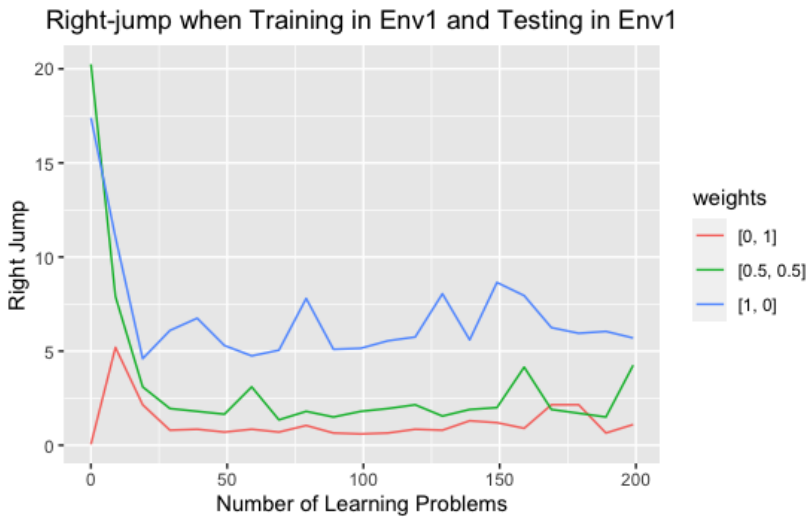


Figure 5.18: The average number of action right-jump in CoinRun Environment 1.

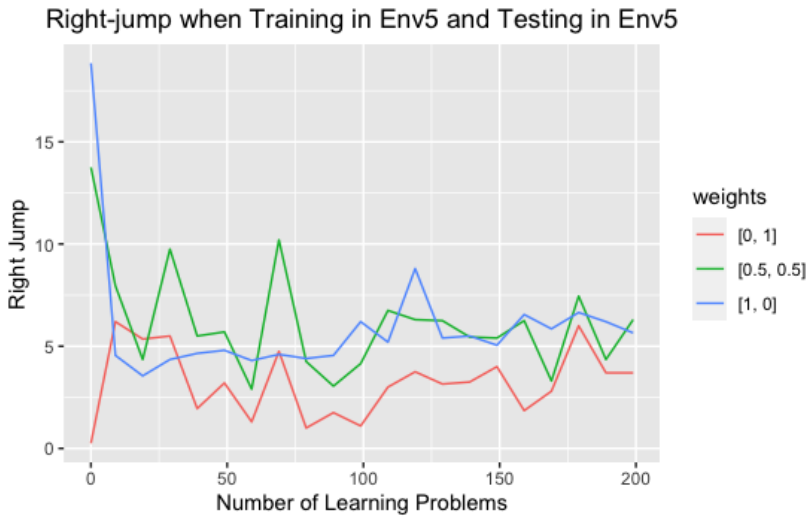


Figure 5.19: The number of right-jump on CoinRun Environment 5.

In Env1, as shown in Figure 5.18, in general, the number of right jumps in weights [1, 0] is higher than [0.5, 0.5], then is [0, 1]. In more detail, for weight [1, 0], the number of right-jumps drops from 18.46 to 4.7 at 20 learning problems, then it fluctuates between 4.73 to 8.9 until 200 learning problems. With weight [0.5, 0.5], it first drops from 20.26 to 2.6 with 30 learning problems, then, it fluctuates between 1.4 and 4.4 until 200 learning problems. For weight [0, 1], the number of right jumps increases to 4.8 within ten learning problems, then drops to 0.93 at 30 learning problems, after that, it fluctuates between 0.76 and 1.1 most of the time.

In Env5, as shown in Figure 5.19, the result is more diverging than that in Env1. In general, the number of right jumps in weights [1, 0] is higher than [0.5, 0.5] and [0, 1]. For weight [1, 0], the number of action *right jump* in the main steps drop from 17.6 to 3.8 within 20 learning problems, then it fluctuates between 4.5 and 6.1 except 7.36 at 120 learning problems. With weight [0.5, 0.5], the total right-jump actions in the main steps drop from 12.76 to 3.7 within 20 learning problems, it then fluctuates between 2.6 to 8.73 until 70 learning problems, after that, it fluctuates between 3.1 and 6.9. With weight [0, 1], it increases from 0.26 to 4.50 within 10 learning

problems and then fluctuates between 1.26 and 4.5.

From the discussion, we can see that, in general, the number of right jumps in weights [1, 0] and [0.5, 0.5] is higher than that of weight [0, 1], as the first objective is to take more actions *right jump* and the second action is to get the coin as soon as possible. In addition, the number of the action *right jump* for each weight has less fluctuation in Env1 than that in Env5 as the performance to solving the problem in Env1 converges better than that in Env5.

Discussion

As shown in Figures 5.16 and 5.17, the results converge better in Env1 than in Env5, especially before 110 learning problems, but they are similar at 200 learning problems. To find the reason behind the different performances in Env1 and Env5, the examples of the learned policy at 70 and 200 learning problems are explored for analysis. Note that not all the states related to the policies learned in Env1 and Env5 are plotted in Figures 5.10 and 5.11 as the space is limited. In this case, the Figures 5.10 and 5.11 are extended as shown in Figures 5.20 and 5.21 to analyze the learned policies in Env1 and Env5.

A	S_{65}	S_{66}	S_{67}	S_{68}	S_{69}	S_{70}	S_{71}	S_{72}	S_{73}	A
A	S_{56}	S_{57}	S_{58}	S_{59}	S_{60}	S_{61}	S_{62}	S_{63}	S_{64}	A
A	S_{47}	S_{48}	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}	A
A	S_{38}	S_{39}	S_{40}	S_{41}	S_{42}	S_{43}	S_{44}	S_{45}	S_{46}	A
A	S_{29}	S_{30}	S_{31}	S_{32}	S_{33}	S_{34}	S_{35}	S_{36}	S_{37}	A
A	S_{20}	S_{21}	S_{22}	S_{23}	S_{24}	S_{25}	S_{26}	S_{27}	S_{28}	A
A	S_{13}	S_{14}	S_{15}	S_{16}	S_{17}	$S_{\%}$	S_{18}	S_{19}	1	A
A	S_{9}	S_{10}	S_{11}	S_{12}	S	S	S	S	S	A
A	S_{5}	S_{6}	S_{7}	S_{8}	A	A	A	A	A	A
A	S_{1}	S_{2}	S_{3}	S_{4}	A	A	A	A	A	A
A	S	S	S	S	A	A	A	A	A	A

Figure 5.20: Extended States of CoinRun Environment 1(Env1).

A	S_{73}	S_{74}	S_{75}	S_{76}	S_{77}	S_{78}	S_{79}	S_{80}	S_{81}	A
A	S_{64}	S_{65}	S_{66}	S_{67}	S_{68}	S_{69}	S_{70}	S_{71}	S_{72}	A
A	S_{55}	S_{56}	S_{57}	S_{58}	S_{59}	S_{60}	S_{61}	S_{62}	S_{63}	A
A	S_{46}	S_{47}	S_{49}	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	A
A	S_{37}	S_{38}	S_{39}	S_{40}	S_{41}	S_{42}	S_{43}	S_{44}	S_{45}	A
A	S_{28}	S_{29}	S_{30}	S_{31}	S_{32}	S_{33}	S_{34}	S_{35}	S_{36}	A
A	S_{20}	S_{21}	S_{22}	S_{23}	S_{24}	S_{25}	S_{26}	$S_{\%2}$	S_{27}	A
A	S_{12}	S_{13}	S_{14}	S_{15}	S_{16}	S_{17}	S_{18}	$S_{\$}$	S_{19}	A
A	S_{5}	S_{6}	S_{7}	S_{8}	S_{9}	S_{10}	S_{11}	$S_{\%1}$	1	A
A	S_{1}	S_{2}	S_{3}	S_{4}	A	A	A	A	A	A
A	S	S	S	S	A	A	A	A	A	A

Figure 5.21: Extended States of CoinRun Environment 5(Env5).

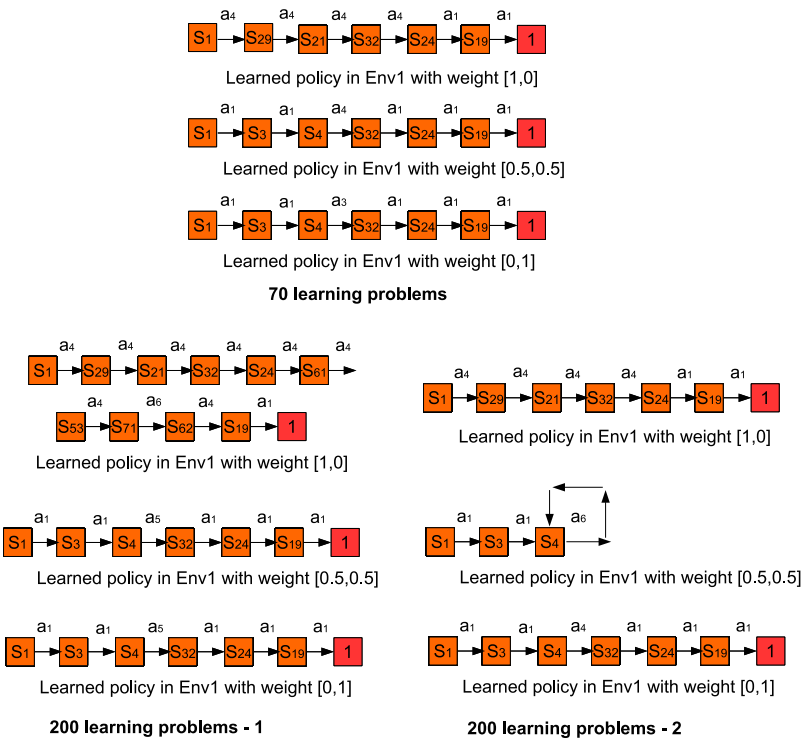


Figure 5.22: Learned policies in 'CoinRun Action Bias' Env1.

As shown in the top of Figure 5.22, all the policy of three weights can solve the problems with 6 main steps at 70 learning problems, where there are 4, 1, and 0 of a_4 with weights [1, 0], [0.5, 0.5], and [0, 1] respectively. In this case, it converges well at 70 learning problems. At 200 learning problems, with weight [0.5, 0.5], the policy cannot converge in some cases (as it shows at the right-hand side of the bottom in Figure 5.22), thus it has the highest average main steps. In this case, at 200 learning problems, weights [1, 0] and [0, 1] performs better than [0.5, 0.5]. For weight [1, 0], it has more steps than [0, 1] is because the optimal policies have more steps and more a_4 (as it shows at the left-hand side bottom in Figure 5.22), which makes sense for the objective of taking more action *right jump*. However, though there are more steps in the optimal solution of weight [1, 0] and [0, 1], weight [1, 0] performs better than [0, 1] may due to the optimum

policy of [1, 0] is easier to learn than that of [0, 1]. This is because the optimum policy of weight [1, 0] just needs to take the same actions right jump at different states, but the optimum policy of weight [0, 1] has to take different actions according to the conditions.

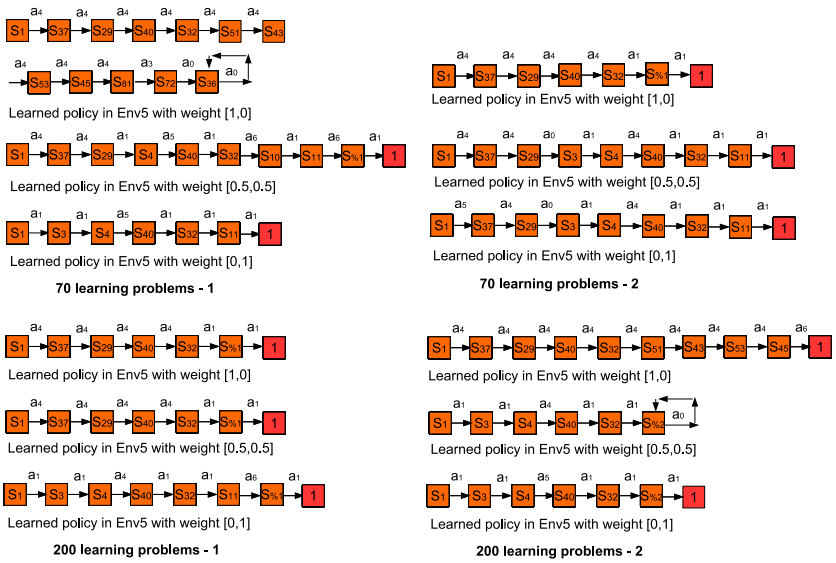


Figure 5.23: Learned policies in 'CoinRun Action Bias' Env5.

As shown in Figure 5.23, at 70 learning problems, with the learned policy, the best policies with weights [1, 0], [0.5, 0.5] and [0, 1] can solve the problem in Env5 with 6, 8, and 6 steps under the best cases (as shown on the top of Figure 5.23). With 200 learning problems, the policies with three weights are possible to converge, but with weight [0.5, 0.5], it may stick at $S_{\%2}$, thus increasing the average steps for solving the problem. In addition, same with Env1, there is more action a_4 in the optimum policy of weight [1, 0] than that of [0, 1] as the difference on the goals of different objectives.

From the discussion, we can see that in both Env1 and Env5 1) with weight [0.5, 0.5], the policy is harder to converge than other weights, thus the performance is slightly less optimum than weights [0, 1] and [1, 0],

especially at 200 learning problems. 2) there are more actions *right jump* in the optimum policy which makes sense for the objective of weight [1, 0]. 3) weight [1, 0] performs better than [0, 1] as the optimum policy of [1, 0] is easier to learn than that of [0, 1].

5.6.2 CoinRun Step vs Reward

In 'CoinRun Action Bias', the agent is trying to learn two objectives: take more action right-jump to reach the coin and minimize the steps to reach the coin, where the first objective is to test if the agent can generalize the action, and the second objective is to test if the agent can maximize the long term payoff with MOXCS. In this subsection, to test if the agent can minimize the steps to reach the coin with MOXCS, another objective is added in the CoinRun environment. Each result obtained in this work is the average of 30 independent runs.

Experiment Design

Another two-objective CoinRun experiment is developed by adding another coin in Env1 and Env5, which is closer to the start state but with less reward. Thus, as it shows in Figure 5.24, there are two coins in the environment, where if the agent collects the red one, it will get the reward of 30, or if the agent collects the yellow coin, it will get the reward of 60. In this case, the two objectives are to minimize the steps to get the final reward and to maximize the long-term payoff to get the coin with a larger reward with more steps.

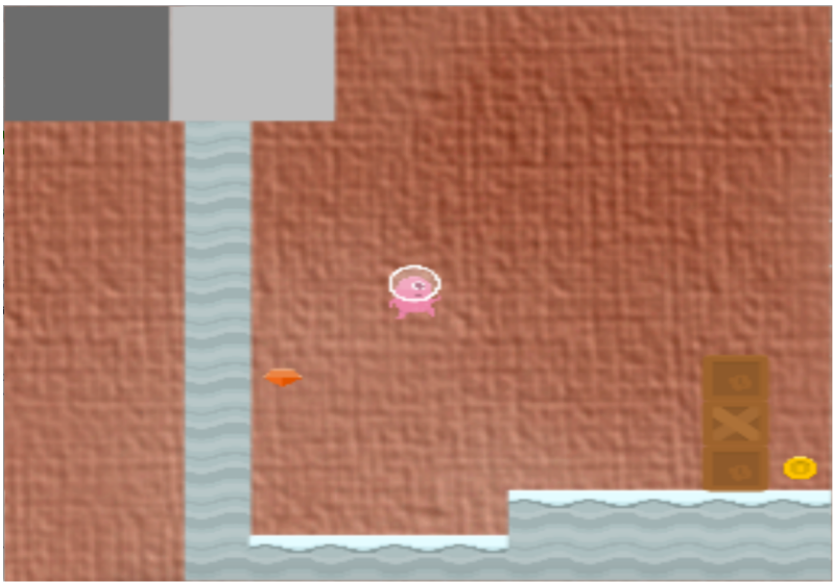


Figure 5.24: Two Objectives in CoinRun Environment 5.

Figure 5.25: Two Objectives in CoinRun Environment 1.

In Section 5.6.2, the experiment is first conducted in Env1 and then Env5, where the results will be shown and analyzed.

Experiment Settings

In the experiments of this section, the typical parameter settings recommended [14] have been followed. In this section, the experiment settings are mostly the same as the settings in Section 5.6.1 including the parameters of MOXCS and the rules for collecting results.

In the experiments of this section, some parameters are set in both 'CoinRun Steps vs. Reward' in both Env1 and Env5. For example, the threshold for performing the niche GA θ_{ga} is 2000, the payoff decay rate $gamma$ is 0.9, the probability of generating a hash in a condition $P_{\#}$ is 0.18, the mutation rate μ is 0.04, ch_a is true, ch_c is false and ifG is true. When training and testing in Env1, N is 8000000, the prediction p, error e and fitness f for the first objective are 0.2, 0.0001 and 10, those for the second objective are 20, 0.001 and 10. When training and testing in Env5, N is 800000, the prediction p, error e and fitness f for the first objective are 0.3, 0.000001 and 10, those for the second objective are 20, 0.001 and 10.

Results and Analysis

The experiment results of MOXCS are collected in training and testing in Env1 and Env5. As the trends of the results are quite similar in those two environments, in this section, the results are demonstrated and discussed together.

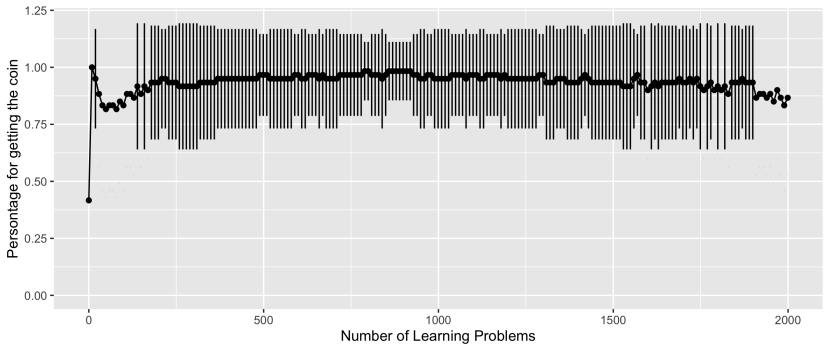


Figure 5.26: Percentage of the agent reaching final reward in Env1.

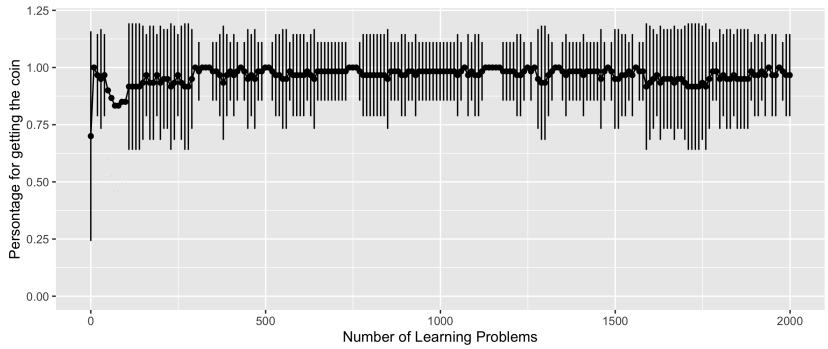


Figure 5.27: Percentage of the agent get to the final reward in Env5.

The experiment results of MOXCS are collected in training and testing in Env1 and Env5. As shown in Figures 5.26 and 5.27, in those two environments, the percentage of the agent to get one of the coins starts from 40% and 70% in Env1 and Env5, but after that, the trends of them are quite similar. The agent can get one of the coins in both Env1 and Env5 every trial with 10 learning problems. At this stage, the agent collects the nearest coin most of the time. Then there is a drop for both of them after that when the agent is learning how to collect the coin further away but with a larger reward in the second objective. For example, it drops to 81.6% at 50 learning problems in Env1 and 83.3% at 70 learning problems in Env5. The percentage of the agent to get one of the coins increases back to over 90% shortly when the agent learns how to collect the coin further away but with a large reward. For example, it increases to 95% with 210 learning problems in Env1 and 96.6% at 190 learning problems. After about 500 learning problems, the performance is relatively stable. Between 500 and 1500 learning problems, the percentage of the agent to get one of the coins keeps between 93.3% and 98.3% in Env1, and between 93.3% and 100% in Env5. But at the final stage, there is a drop in Env1, the percentage drops to 86.6% and 83.3% at 1910 and 1990 learning problems. There is a drop after 1500 learning problems as well. For example, in 1990 learning problems in Env5, the percentage of getting one of the coins drops to 83.3%. This

may be due to a new poor classifier or an over general classifier with high error being added, thus influencing the accuracy of the long-term payoff at some states.

Figure 5.28: Ratio of different coin types obtained by the agent with number of learning problems in Env1.

Figure 5.29: Ratio of different coin types obtained by the agent with number of learning problems in Env5.

In Figures 5.28 and 5.29, the ratio of an agent to get one of the coins are plotted. The results in Env1 and Env5 are quite similar. As the total ratio to get coin0 and coin1 is 100%, in this case, only the percentage to get coin1 is analyzed here.

As shown in Figure 5.28 in Env1, the ratio to get coin1 is 1.7% with 10 learning problems, it increases to 50% at 320 learning problems. The ratio

to get coin1 can go as high as 52.94% at 1960 learning problems and as low as 48.2% at 1700 learning problems. Note, the ideal value of the ratio to get coin1 is 50% as if it is higher than 50%, which means the ratio to get coin0 is under 50%. As shown in Figure 5.29 in Env5, the ratio to get coin1 starts with 8.6% at 40 learning problems, then increases to 50% at 300 learning problems. However, the performance fluctuated afterward, for example, the percentage to get coin1 drops to 45.4% at 1590.

From the discussion, we can see that though there are some fluctuations after they achieve the best performance, however, there is a clear trend that the agent can collect coin0 and coin1 over at least 45% after the performance has converged in both Env1 and Env5. In addition, the performance of collecting different coin types in Env1 performs better than that in Env5.

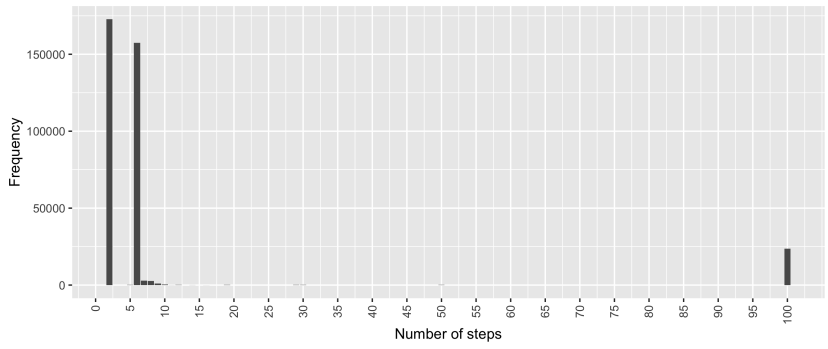


Figure 5.30: Number of steps in result in Env1.

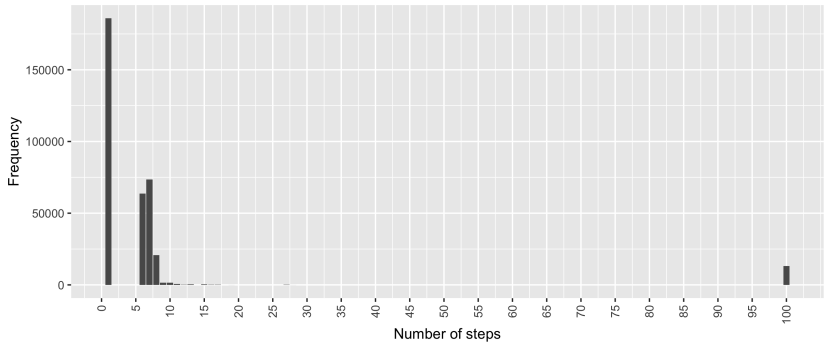


Figure 5.31: Number of steps in result in Env5.

To analyze how many steps that the agent learns to get one of the coins, the total main steps to get one of the coins for each test are counted in both Env1 and Env5. Note, as the goal is to understand the common case of how many steps that the agent requires to get the coins for other analysis later, for example, to define the number of steps in the optimal solution. In this case, the details of the counts for each number of steps in the tests are not our focus. In Figures 5.30 and 5.31, the total steps to get coins are plotted. First, the agent takes one or two steps to get a coin most of the time, i.e., the agent gets coin0. Second, in most common cases, it uses 6 main steps in Env1 and 6,7 or 8 main steps in Env5 to get coin1. Besides the most common cases, when the agent uses less or equal than 15 steps to collect coin1, it shows the potential to learn the optimum policies, thus it can be considered as the sub-optimal policy. Third, some records show the agent occasionally uses 16 to 50 steps to get a coin, but these records could be ignored for two reasons: 1) the number of steps between 16 to 50 to get a coin is really rare. 2) the policy with 16 to 50 to reach a coin is not an optimal policy and does not show any potential of learning optimum policy. Last, when the agent spends 100 steps in the trail, as 100-time steps are the settings of the maximum step in the test; thus, it means the agent has not reached any coin.

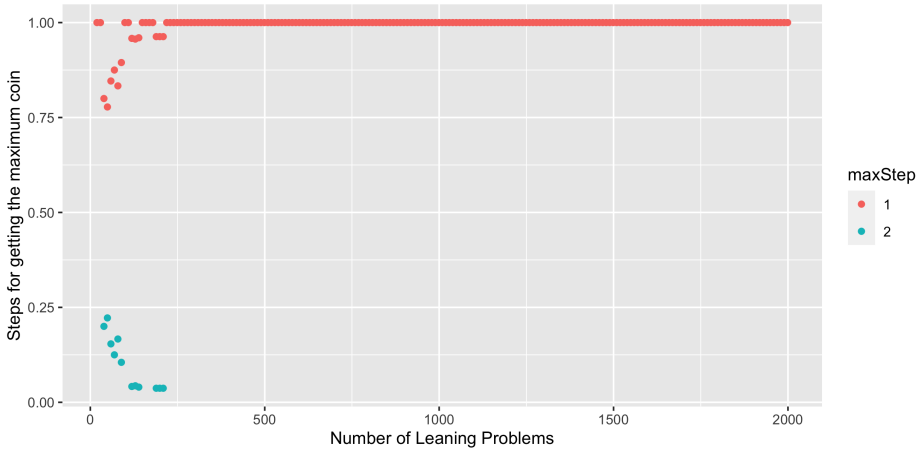


Figure 5.32: Obtaining coin with large reward in Env1.

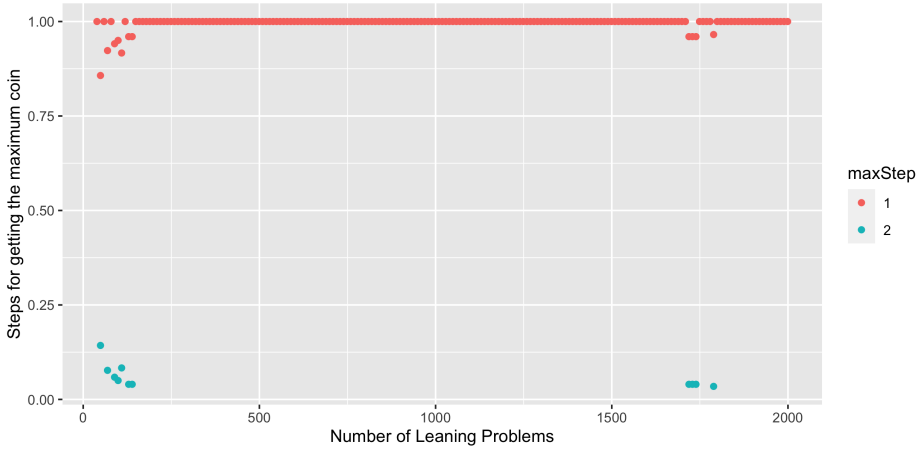


Figure 5.33: Obtaining coin with large reward in Env5.

In Figures 5.32 and 5.33, the results focus on whether the agent obtaining the coin with a large reward in Env1 and Env5 with an optimal policy or sub-optimal policy. Following the distribution of the number of steps in the learning policies, the results are divided into three groups. Group 1 is getting coin1 with less than and equal to 15 steps, which supposes the agent finds the optimal policy or sub-optimal policy to get coin1. Groups 2 and 3 are collecting coin1 with 15 to 50 steps, or over 50 steps to get coin1,

which are considered as the non-efficient policy, or could not find a policy to reach the coin at all. Note, there are only two groups in these figures as there is no data for Group 3.

From the results from Figures 5.32, we can see that the agent learns to use an optimal policy to get coin1 at 100% of the time with 220 learning problems, and the performance is quite stable afterward in Env1. The chance to learn and use the non-optimal policy achieves 0% after 210 learning problems and keeps stable in most cases. From the results of Env5 from Figures 5.33, we can see that the percentage of using an optimum policy to reach coin1 achieves 100% at 150 learning problems, but with a slight drop to 96% after 1700 learning problems. The chance to learn and use the non-optimal policy achieves 0% after 140 learning problems and keeps stable in most cases, but there is a slight chance (0.03 to 0.04) to use the non-optimal policy at over 1700 learning problems.

In addition, from both of the figures, we can see that with the training, the ratio of Group 2 is reduced dramatically from about 20% to 0%. This is means the agent stop using the sub-optimal rules to solve the problem after learning the optimal rules.

5.7 Multi-Objective CoinRun Generalization

Experiment Settings

In the experiments of 'CoinRun Action Bias Generalization', the typical parameter settings recommended [14] have been followed. The experiment settings are mostly the same as the settings in 5.6.1 including the parameters of MOXCS and the rules for collecting results, except the $P_{\#}$. The $P_{\#}$ is 0.6 when training in Env1 and testing in Env5, and it is 0.7 when training in Env5 and testing in Env1.

In the experiments of 'CoinRun Step vs Reward Generalization', the typical parameter settings recommended [14] have been followed as well.

Some parameters are setting for this generalization experiments are listed. For example, the prediction p, error e and fitness f for the first objective are 0.2, 0.00000001 and 10, those for the second objective are 20, 0.0005 and 10, $gamma$ is 0.93, ch_a is true, ch_c is false ,and ifG is true. When training in Env1 and testing in Env5, N is 8000, θ_{ga} is 200000, $P_{\#}$ is 0.7, μ is 0.05. When training in Env5 and testing in Env1, N is 8000000, θ_{ga} is 2000, $P_{\#}$ is 0.74, μ is 0.04.

5.7.1 Result of CoinRun Action Bias

As the evidence shows in Figures 5.16 and 5.17, MOXCS is able to solve the multi-objective reinforcement learning problem ('CoinRun Action Bias') in Env1 and Env5. In this subsection, the agent is trained in one environment and tested in another environment to evaluate the generalization performance.

Results and Analysis

The experimental results of MOXCS are collected by training in Env1 and testing in Env5 (Env15), then training in Env5 and testing in Env1 (Env51), respectively. As the trends of the results are quite similar in those two environments, in this section, the results will be demonstrated and discussed together.

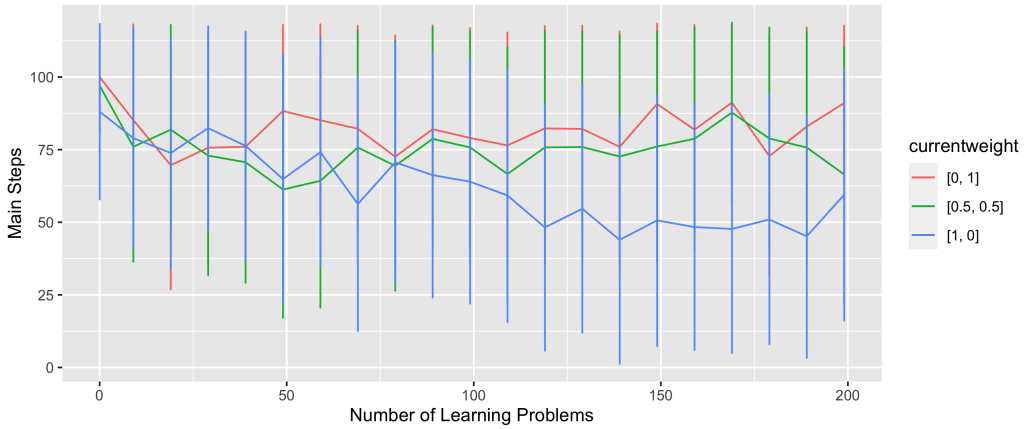


Figure 5.34: Learning performance of training in Env1 and testing in Env5 (Env15).

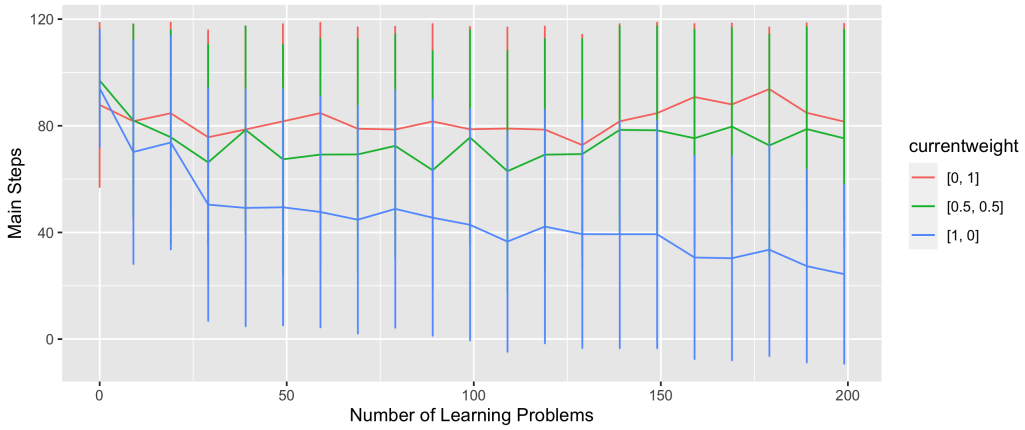


Figure 5.35: Learning performance of training in Env5 and testing in Env1 (Env51).

It can be seen in Figure 5.34, in Env15, only weight [1, 0] is tend to converge. The best performance for weights [1, 0], [0.5, 0.5] and [0, 1] is 43.9 at 140 learning problems, 61.2 at 50 learning problems, and 69.6 at 20 learning problems respectively. Similar to the result in Env15, only weight [1, 0] is tend to converge in Env51 as shown in Figure 5.35. The best performance for weights [1, 0], [0.5, 0.5] and [0, 1] is 24.3 at 200 learning

problems, 62.9 at 110 learning problems, and 72.7 at 130 learning problems respectively.

From the results above, we can see that the results are hard to converge with weights [0, 1] and [0.5, 0.5], but much easier with weight [1, 0] in both Env15 and Env51.

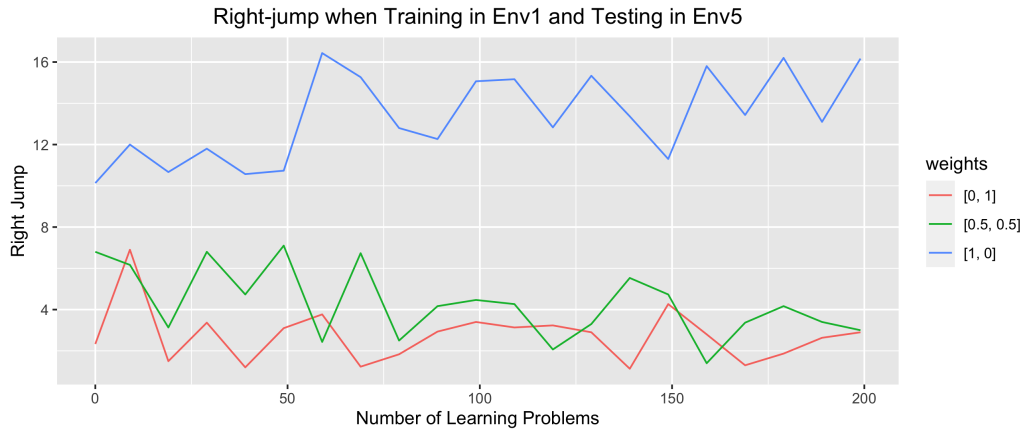


Figure 5.36: The number of right-jumps when training in Env1 and testing in Env5(Env15).

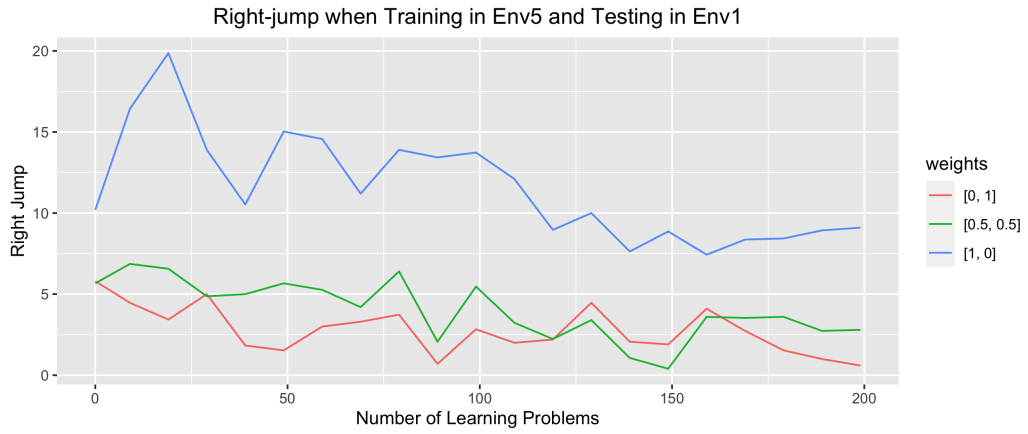


Figure 5.37: The number of right-jump when training in Env5 and testing in Env1(Env51).

As shown in Figure 5.36, in Env15, there are more actions *right jump* in weight [1, 0] than weights [0.5, 0.5] and [0, 1]. With weight [1, 0], the number of right jump in main steps for solving the 'CoinRun Action Bias' problem increases from 10.1 at the beginning to 16.4 at 60 learning problems, then drops to 12.2 at 90 learning problems, then fluctuates between 12.2 and 16.2 until 200 learning problems, except 11.3 at 150 learning problems. With weight [0.5, 0.5], the number of right jumps fluctuates between 2.0 to 7.1 most of the time, except 1.4 at the 160 learning problems. For weight [0, 1], the number of right jumps fluctuates between 1.1 to 3.7, except 6.9 at 10 and 4.2 at 150 learning problems. As shown in Figure 5.37, in Env51, similar to that in Env15, weight [1, 0] has more right jumps than weights [0.5, 0.5] and [0, 1] as well. However, with weight [1, 0], there is an increasing trend in Env15 and a decreasing trend in Env51. In Env51, first, with weight [1, 0], the number of right jumps for solving the 'CoinRun Action Bias' problem increases from 10.2 to 19.8 at 20 learning problems, then drops to 7.4 at 160 problems with huge fluctuation, finally slightly increases to 9.1 at 200 learning problems. With weight [0.5, 0.5], the number of right jumps fluctuates between 1.06 to 6.8 most of the time, except 0.4 at the 150 learning problems. For weight [0, 1], the number of right jumps fluctuates between 0.6 to 5.0, except 5.8 at the beginning.

From the discussion above, we can see that more right jumps can help to resolve the 'CoinRun Action Bias' problem. For example, with weight [1, 0], in both Env15 and Env51, it has more right jumps, thus the result converges better than other weights.

Discussions

As shown in Figures 5.34 and 5.35, as the whole trends of three weights are consistent, where only the performance of weight [1, 0] is keeping improving, but weights [0, 1] and [0.5, 0.5] cannot converge. In this case, to find out the reason why weight [1, 0] converges better than other weights, some of the examples of the learned policy at 200 learning problems are

explored for analysis.

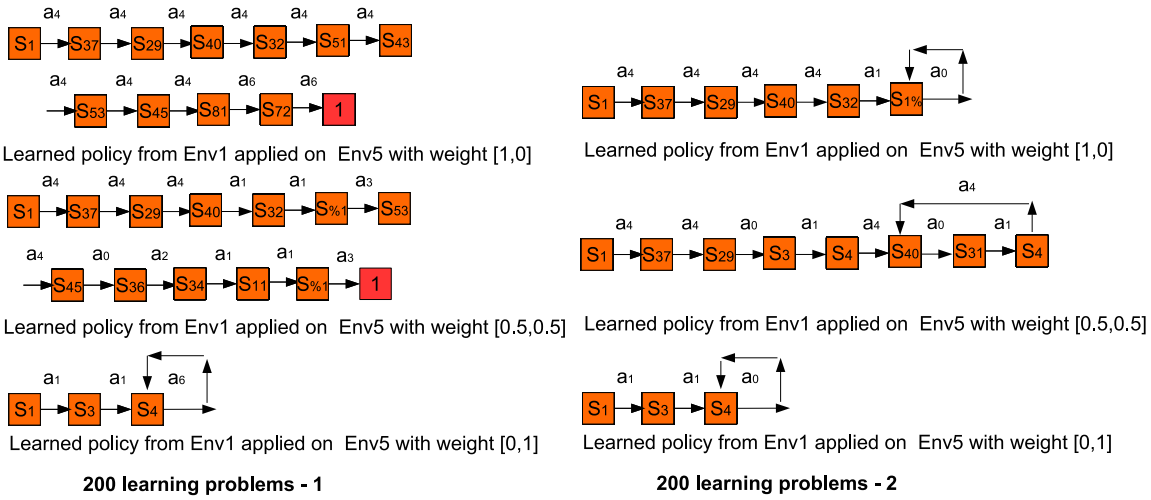


Figure 5.38: Learned policies of training in Env1 and testing in Env5 (Env15).

From the policies learn from Env15 at 200 learning problems in Figure 5.38, we can see that, first, with weights [1, 0] and [0.5, 0.5], there learned policies can solve the problem with 11 and 12 steps, and hard to find the policy to solve the problem with weight [0, 1]. It explains why weight [0, 1] performs worse than other weights, and weight [1, 0] converges slightly better than weight [0.5, 0.5] in Figure 5.34. Second, for both of the successful and unsuccessful policies, there are more actions *right jump* with weight [1, 0] than weight [0.5, 0.5] and [0, 1], which makes sense of the objectives for different weights. Third, some policies cannot reach the final

state and are stuck in some states. There are two reasons for that. First, it cannot find the match set at that state, thus taking action a_0. Second, it takes the wrong action as the new classifier and over general classifier affect the prediction for the actions at that state.

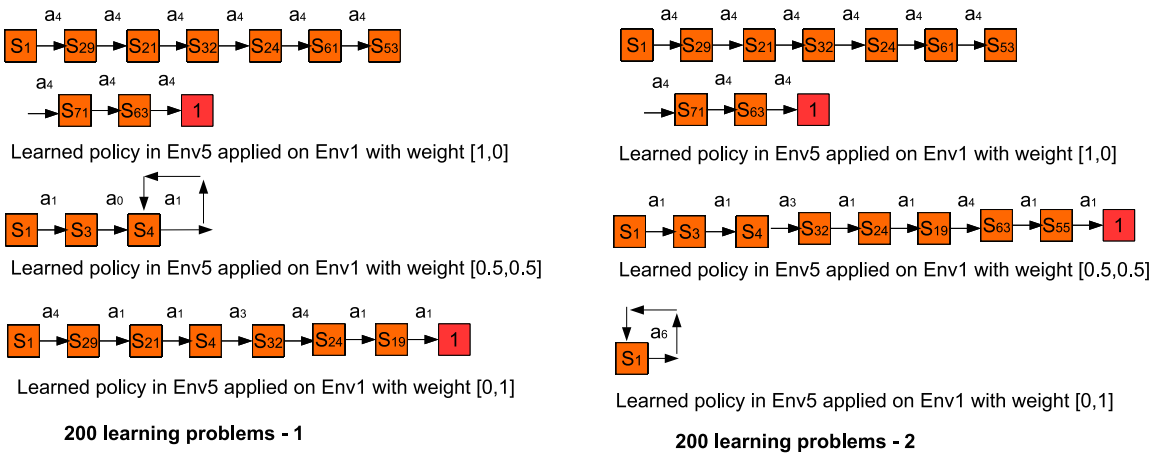


Figure 5.39: Learned policies of training in Env5 and testing in Env1 (Env51).

Figure 5.39 shows the learned policies at 200 learning problems in Env51. First, with those three weights [1,0], [0.5, 0.5], and [0, 1], the problem in Env5 can be solved by training in Env1 with 9, 8, and 7 steps, respectively, and there are more optimum policies with weight [1, 0]. It explains why weight [1, 0] performs better than other weights. Second, weight [1, 0] has more action *right jump* than other weights among all the other optimal

policies. For example, it has 9 action *right jump* in the optimal policy with weight [1, 0], but only 1 and 2 action *right jump* in the optimal policies with weights [0.5, 0.5] and [0, 1].

From what has been discussed above, we can see that in both generalization experiments, with weight [1, 0], the agent is easier to reach the coin, as it only needs to general the action *right jump* in different states, whereas for other weights, it is trickier to general the inputs from different states. This is because when generalizing the inputs from different states if the $P_{\#}$ used for generalization is too low, it is hard to generate enough match set to some state at the testing environment. On the other hand, if the $P_{\#}$ used for generalization is too high, the over general classifier with the high error may have a negative influence on the predictions of some states. Therefore, generalizing action is easier than generalizing the environment.

5.7.2 Result of CoinRun Step vs Reward

For testing the performance of MOXCS to learn the larger reward, another objective is added in Env1 and Env5, as shown in Figures. 5.24 and 5.25. In this section, the agent is trained by MOXCS in one environment and tested in another environment for evaluating the generalization ability of MOXCS.

Results and Analysis

The experimental results of MOXCS are collected by training in Env1 and testing in Env5 (Env15), then training in Env5 and testing in Env1 (Env51), respectively. As the trends of the results are quite similar in those two environments, in this section, the results will be demonstrated and discussed together.

When training and testing in different environments, the performance is not as good as training and testing in the same environment as shown

in Figures 5.40 and 5.41. However, we can still see the potential of MOXCS to solve the generalization problems across different environments. More details, in those two environments, the percentage of the agent to get one of the coin starts from 50% when training in Env1 and testing in Env5(Env15), but after that, the trends of them are similar. In Env15, the performance jumps to 74% first, then the performance reduces to 42% at 120 learning problems and fluctuates in a range of 34% and 56% between 100 to 1000 learning problems, and fluctuates in a range of 28% and 54% between 1000 to 1500 learning problems, and fluctuates in a range of 28% and 50% between 1500 to 2000 learning problems. When training in Env5 and testing in Env1(Env51), the performance starts from 65% then jumps to 81.6% first, then the performance reduces to 61.6% at 50 learning problems and fluctuates in a range of 56.6% to 73.3% until 1500 learning problems except for 75% at 670 learning problems. After 1500 learning problems, it keeps in the range from 61.6% and 73.3% most of the time except 75% and 76.6% at 1740 and 1750 learning problems.

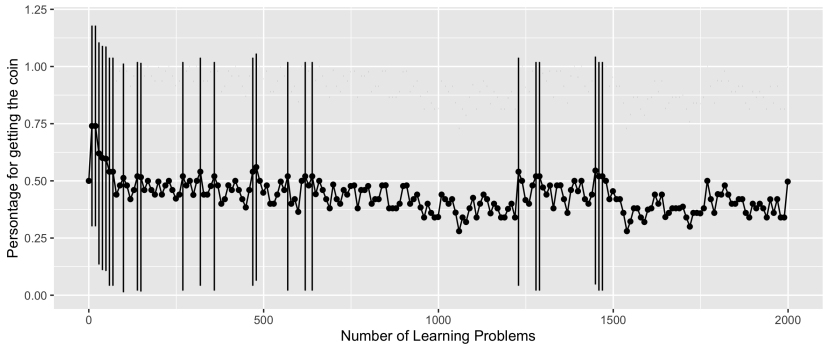


Figure 5.40: How many times the agent achieves the final reward when training in Env1 and testing in Env5.

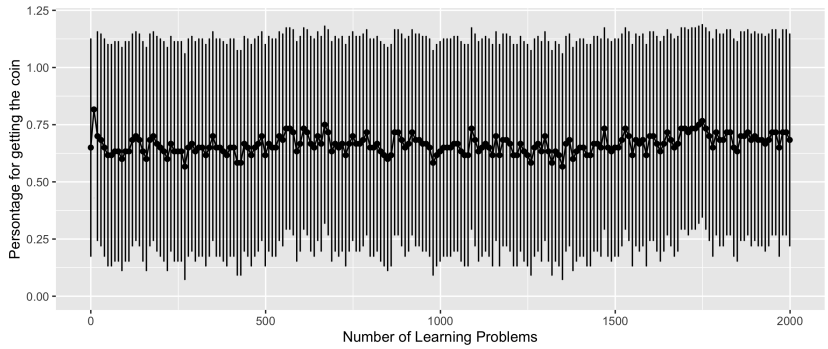


Figure 5.41: How many times the agent achieves to the final reward when training in Env5 and testing in Env1.

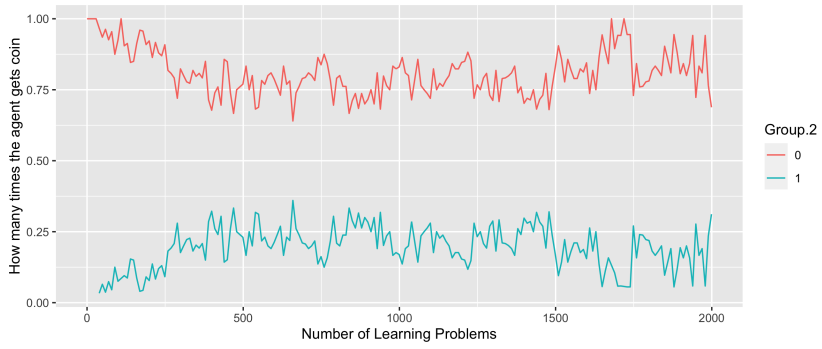


Figure 5.42: Different final reward obtained by the agent with number of learning problems when training in Env1 and testing in Env5.

Figure 5.43: Different final reward obtained by the agent with number of learning problems when training in Env5 and testing in Env1.

In Figures 5.42 and 5.43, the percentages of the agent to get one of the coins are plotted. Same with the analysis in subsection 5.6.2, only the percentage to get coin1 is analyzed here. In Env15 and Env51, the ratio to get coin1 is 0% at the beginning, and the agent starts to learn the policy to collect coin1 shortly. In Env15, the agent starts to get coin1 at 40 learning problems and the ratio to get coin1 is 3.3%, then it increases to 33.3% at 470 learning problems. After that, in Env15, it stays in a range from 9.5% to 36% until 1480 learning problems. Then, the ratio to get coin1 decreases to 5.5% at 1730 learning problems, and jumps back to 27% at 1750 learning problems, and continues to fluctuate 5.5% to 31.1% afterward. In Env51, the agent starts to get coin1 with a ratio of 4.8% at 30 learning problems, and then it increases to 31.8% at 560 learning problems. After that, it stays in a range from 8.5% to 30.2% until 1250 learning problems. Then, the ratio to get coin1 drops to 1.1% at 1350 learning problems, and increases to 34.7% a with fluctuation at 1750 learning problems and drops to 19.5% at 1800 learning problems. Then achieves 26.8% at 2000 learning problems with fluctuation. From the discussion above, we can see with training by MOXCS in one environment, the agent has the potential to get coin1 in a slightly different environment. The agent learned to collect coin1 after 500 learning problems in both Env15 and Env51. But after 1500 learning prob-

lems, the performance is decreasing in Env15 and increasing in Env51.

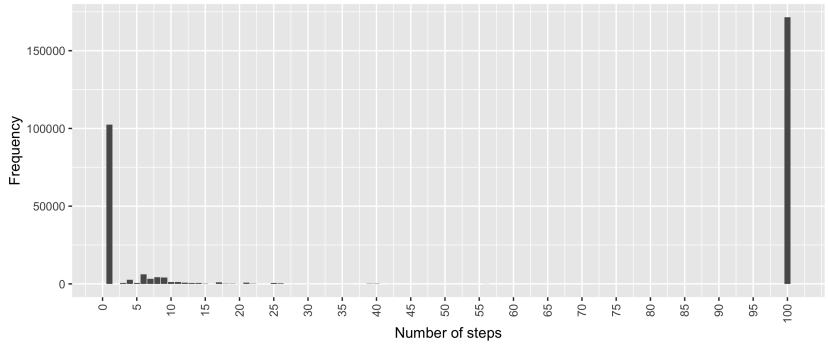


Figure 5.44: Number of total steps in result when training in Env1 and testing in Env5.

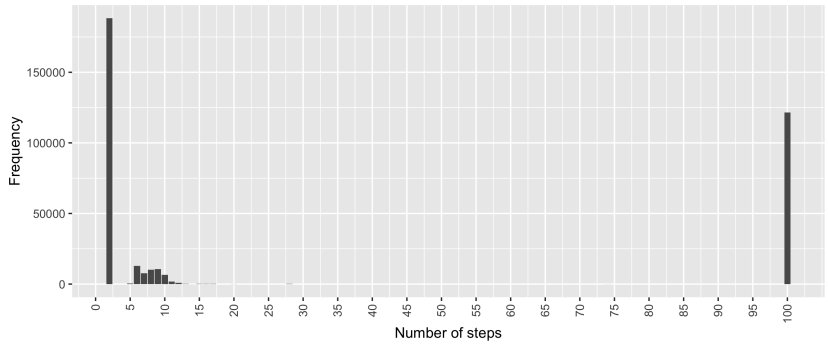


Figure 5.45: Number of total steps in result when training in Env5 and testing in Env1.

In Figures 5.44 and 5.45, the total steps to get coins are plotted. In Env15, the agent takes one or two steps to get a coin most of the time, i.e., the agent gets coin0. Then it gets coin1 with between 6 to 11 main steps. In Env51, the agent takes one step to get a coin most of the time, i.e., the agent gets coin0. Then it takes 6 to 10 steps to get coin1, and sometimes, it takes 11 and 12 steps for coin1. There is a large amount of step 100 in Env15 and Env51 (i.e., 171450 and 121560) when the agent cannot learn any policy to get a coin. Same with Env11 and Env55, some records show

the agent uses 16 to 50 steps to get a coin, but the records could be ignored for two reasons: 1) the number of steps between 16 to 50 to get a coin is really rare. 2) the policy with 16 to 50 to reach a coin is not an optimal policy.

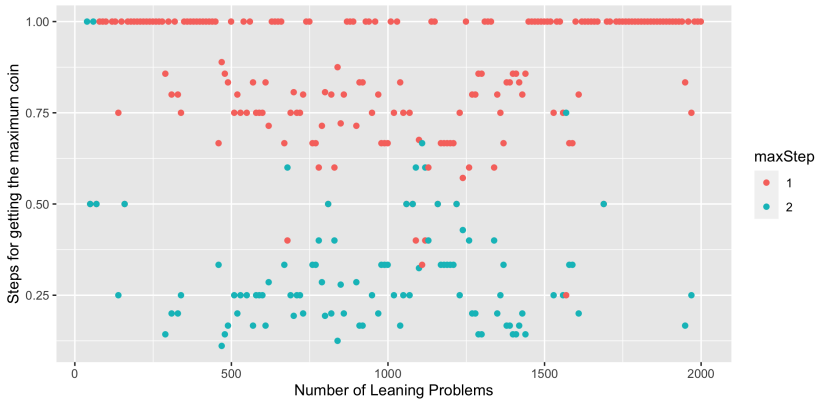


Figure 5.46: Get maximum coin when training in Env1 and testing in Env5.

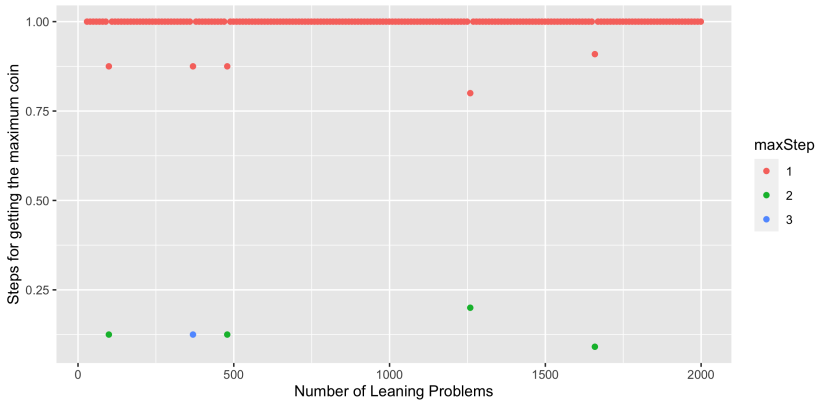


Figure 5.47: Get maximum coin when training in Env5 and testing in Env1.

The results in Figures 5.46 and 5.47 shows the if the agent get the maximum coin with optimal policies or sub-optimal policies in Env15 and Env51, respectively. In Env15, as shown in Group 1, the agent uses the optimal policy or optimum policies to get the large coin more consistently

before 500 and after 1450 learning problems. During 500 to 1500 learning problems, the ratio fluctuates between 50% to 100%. However, the ratio can go as low as 40%, 33.3%, 25% at 680, 1110, 1570 learning problems. For Group 2, the agent uses the non-efficient policy to solve the problem quite a lot at the beginning but is not very stable. For example, the ratio of Group 2 is 100% at 40 and 60 learning problems, but 50% at 50 and 70 learning problems. After that, the policies in Group 2 first drop to 11.1% at 470 learning problems with fluctuation, and then it increases to 66.6% with huge fluctuation, then drops to 14.2% at about 1400 learning problems. After 1500 learning problems, in general, the ratio of Group2 is low, but it reaches 75% to 1570 learning problems. In Env51, the usage of policies in Group 1 is more consistent. The ratio of policy in Group 1 is 100% most of the time, but it drops to 80% at 1260, 87.5% at 100, 370 and 480, and 90.9% at 1660 learning problems. In Group 2, the non-efficient policy has been used with 12.5% at 100 and 480 learning problems and 20% and 9% at 1260 and 1760 learning problems.

5.8 Discussion

5.8.1 Pareto Front

In this subsection, first, the Pareto Front(PF) learned by MOXCS in 'CoinRun Action Bias' and 'CoinRun Step VS Reward' will be analyzed, then a new PF shape will be proposed for a future experiment. Only the PF in the testing environment is analyzed.

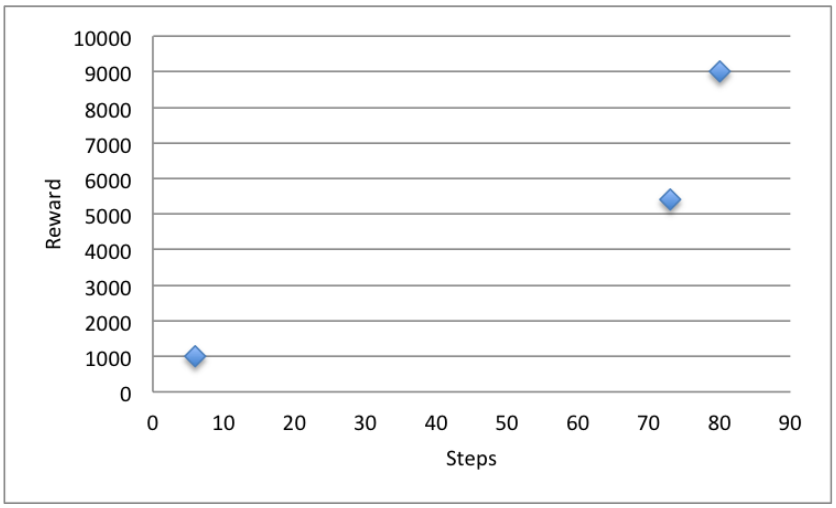


Figure 5.48: Pareto Front of Env1 in 'CoinRun Action Bias'.

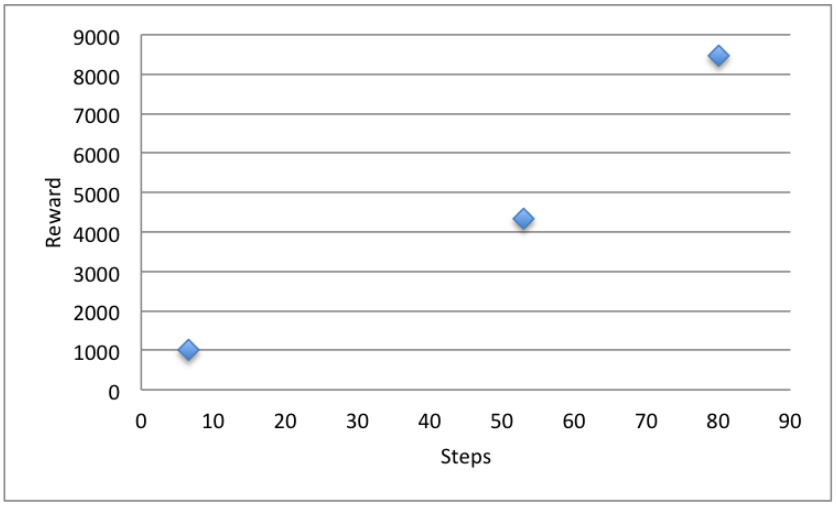


Figure 5.49: Pareto Front of Env5 in 'CoinRun Action Bias'.

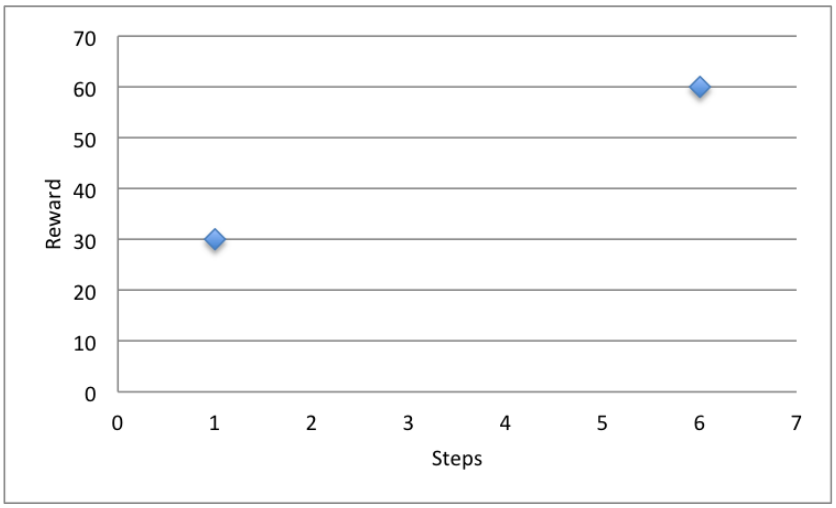


Figure 5.50: Pareto Front of Env1 and Env5 in 'CoinRun Step VS Reward'.

In 'CoinRun Action Bias' (Env1), MOXCS learned a convex PF as it is shown in Figure 5.48. The agent trained by MOXCS has collected the cumulative reward of 1000, 5400, and 9000 with main steps 6, 73, and 80. In 'CoinRun Action Bias' (Env5), MOXCS learned a convex PF as well, and it is shown in Figure 5.49. The agent trained by MOXCS has collected the cumulative reward of 1000, 4350, and 8475 with main steps 6.5, 53, and 80. In 'CoinRun Step VS Reward' (Env1 and Env5), as a result, shown in Figure 5.50, the PF consists of 2 points, namely, 1 step with 30, 6 steps with 60. As in the current 'CoinRun Action Bias' and 'CoinRun Step VS Reward' environment, there are limited solutions on the Pareto Front, and it cannot be determined as a convex or concave PF with limited information. In this case, it is impossible to test if an algorithm can deal with the convex or concave PF or how good an algorithm can find all the solutions on the PF.

A	1	S_{30}	S_{31}	1	S_{33}	S_{34}	1	S_{36}	S_{37}	A
A	S	S	S	S	S	S	S	S_{27}	S_{28}	A
A	S_{13}	S_{14}	S_{15}	S_{16}	S_{17}	$S_{\%}$	S_{18}	S_{19}	1	A
A	1	S_{10}	S_{11}	S_{12}	S	S	S	S	S	A
A	S_5	S_6	S_7	S_8	A	A	A	A	A	A
A	S_1	S_2	S_3	S_4	A	A	A	A	A	A
A	S	S	S	S	A	A	A	A	A	A

Figure 5.51: Multi-Coin CoinRun Environment 1 (MO-Env1).

A	S_{37}	S_{38}	S_{39}	S_{40}	S_{41}	S_{42}	1	S_{44}	S_{45}	A
A	S_{28}	S_{29}	S_{30}	S_{31}	S_{32}	S	S	S_{35}	S_{36}	A
A	S_{20}	S_{21}	S_{22}	1	S_{24}	S_{25}	1	S_{26}	S_{27}	A
A	1	S_{13}	S_{14}	S	S	S	S	$S_{\$}$	S_{19}	A
A	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	$S_{\%1}$	1	A
A	S_1	S_2	S_3	S_4	S	S	S	S	S	A
A	S	S	S	S	A	A	A	A	A	A

Figure 5.52: Multi-Coin CoinRun Environment 5 (MO-Env5).

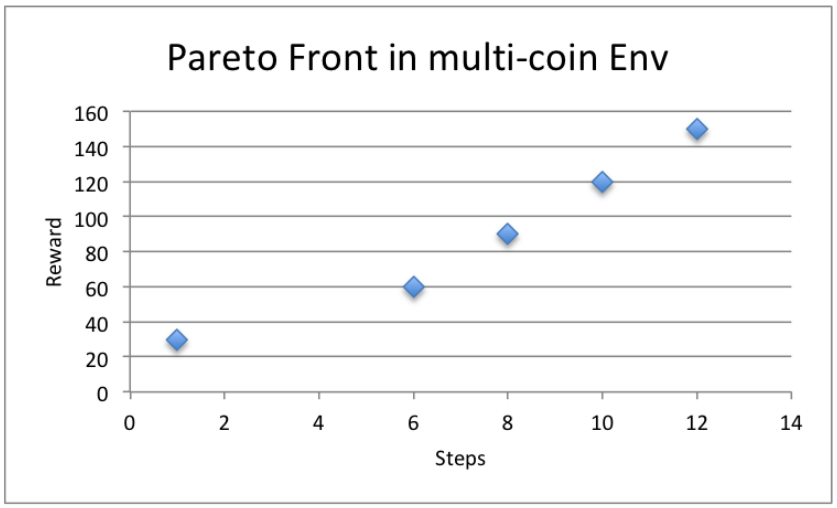


Figure 5.53: Pareto Front of multi-coin CoinRun.

As discussed above, although MOXCS learned multi-objective optimal policies in 'CoinRun Action Bias' and 'CoinRun Step VS Reward', the PF is not very consistent. In the future work, one more coin will be added in the CoinRun Environment Env1 and Env5 as it shows in Figure 5.51 and 5.52, and the game only terminate when the agent collects all coins. In this case, the agent is supposed to learn a more consistent and complex PF as shown in Figure 5.53 in both CoinRun Environment MO-Env1 and MO-Env5.

5.8.2 Parameters of Population and theta GA

In this subsection, to see if MORL needs a large amount of good generic material to begin to function, the different combinations of population size and the threshold for performing the niche GA will be implemented in the experiment.

Several benchmarks were tested in this thesis, for example, Deep Sea Treasure and multi-objective Mountain Car. However, the 'CoinRun Steps vs. Reward' is considered as the best benchmark to test if MORL needs a large amount of good genetic material to begin to function or not for the following reasons:

1. The two objectives in the experiment 'CoinRun Steps vs. Reward' are pretty straightforward to measure as the optimal policies for two goals can be measured by counting the steps in the learned policies for reaching two coins, respectively. MOXCS begins to function when the steps in the learned policies are closed to the optimal policy; otherwise, MOXCS does not begin to function.

2. Deep Sea Treasure is a good multi-objective benchmark. However, it is a good benchmark to test if the MOXCS works effectively, for example, how many treasures the agent collects, rather than if MOXCS begins to function or not.

3. Multi-objective Mountain Car has been tested, but the performance is unstable, so it would not be a good benchmark for testing the GA-related experiment.

To test the influence of the population size and the threshold for performing the niche GA, except for these two parameters, all the other parameters settings for training and testing in the same environment are the same as the settings in subsection 5.6.2. For training and testing in different environments, the parameters settings are the same as the settings in 5.7 including the rules for collecting results.

To show the difference in the results from different parameters settings, only a few measurements are selected for analysis. Two measurements, 'Ratio of reaches coin' and 'Ratio of reaches coin1' are selected for the result analysis as they can measure if the MORL algorithm enables the agent to get the coin, especially if the agent reaches coin1. Another essential measurement, 'Optimal policy reaches coin1', is added to measure if the agent can collect coin1 within 15 steps by optimum and sub-optimum policies.

The results shown in Table 5.4 are collected with 500 learning problems, where the results can converge when training and testing in the same environment, or the trends can be observed when training and testing in

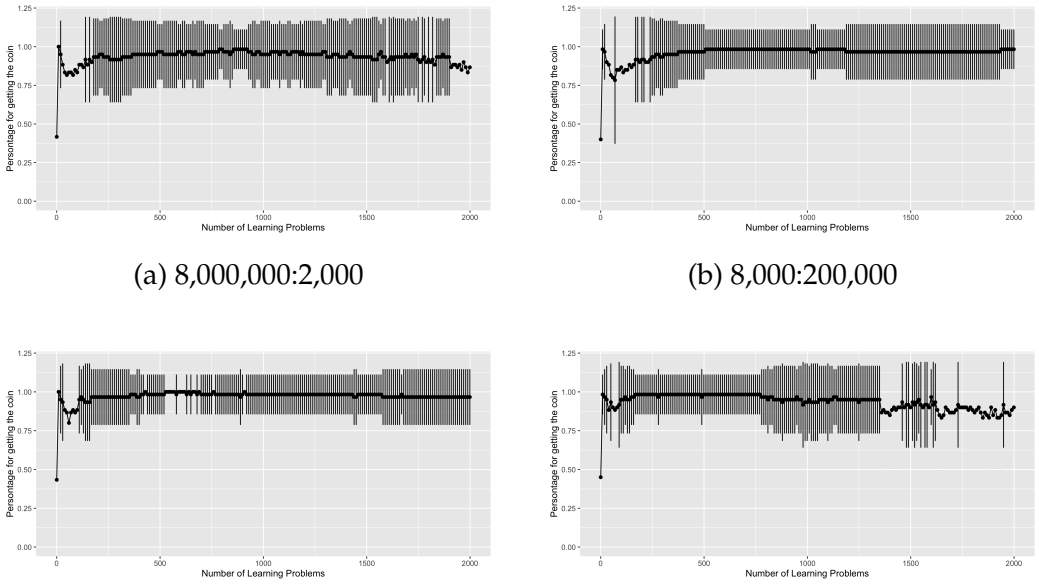
Table 5.4: The results of different combination of population size and the threshold for performing the niche GA.

Test Env	Parameters (Pop vs GA)	Ratio of Reaches Coin	Ratio of Reaches Coin1	Optimal Policy Reaches Coin1(steps<=15)
11	8000:200,000	0.96	0.5	1
	80,000:20,000	0.98	0.49	1
	800,000:2,000	0.98	0.508	1
	8,000,000:2,000	0.96	0.5	1
55	8000:200,000	0.96	0.46	1
	80,000:20,000	0.98	0.49	1
	800,000:2,000	0.98	0.49	1
	8,000,000:2,000	1	0.5	1
15	8000:200,000	0.48	0.17	1
	80,000:20,000	0.30	0.05	1
	800,000:2,000	0.53	0.15	0.8
	8,000,000:2,000	0.44	0.23	1
51	8000:200,000	0.66	0.2	1
	80,000:20,000	0.55	0.09	1
	800,000:2,000	0.66	0.22	1
	8,000,000:2,000	0.61	0.13	1

different environments. It shows the evidence that MOXCS with different combinations of population size and the threshold for performing the niche GA can solve the 'CoinRun Steps vs. Reward' problem when training and testing in the same environment, and it has the potential to solve the problem when training and testing in slightly different environments. However, when training and testing in different environments, the results haven't converged and have a high standard deviation. In this case, the results couldn't be used to measure the influence of population size and GA as many other factors are causing the results.

As when training and testing in the same environment, as the performance is quite similar in both Env1 and Env5 at 500 learning problems (as shown in Table 5.4) but with fluctuation and standard deviation, thus it is hard to use the results at 500 learning problems for comparing the performance of different combinations of population size and the threshold for performing the niche GA. In this case, the converging stability of percentage for the agent to get the final reward and ratio to get coin1 are measured. Note, in this analysis, the percentage for the agent to get the final reward is more important than the ratio to get coin1 as the performance of the ratio to get different coin types are quite similar. In this case, if the percentage for the agent to get the final reward is low, even the ratio to get coin0 and coin1 achieves the best performance which should be 50% of coin0 and 50% of coin1, the agent is still cannot collect much coin1.

First, as shown in Figure 5.54, in Env1, for the measurement of the percentage of the agent get the final reward, the performance of Population vs. GA of 8,000:200,000 and 80,000:20,000 are quite similar, and they converge better than 8,000,000:2,000 and 800,000:2,000, where the performance of 8,000,000:2,000 is better than 800,000:2,000. In this case, the measurement of collecting different coin types by the agent for 8,000:200,000 and 80,000:20,000 are compared. As shown in Figure 5.55, in Env1, the performance of Population vs. GA of 8,000:200,000 is better than 80,000:20,000 is the best because 8,000:200,000 achieves the best performance of reaching



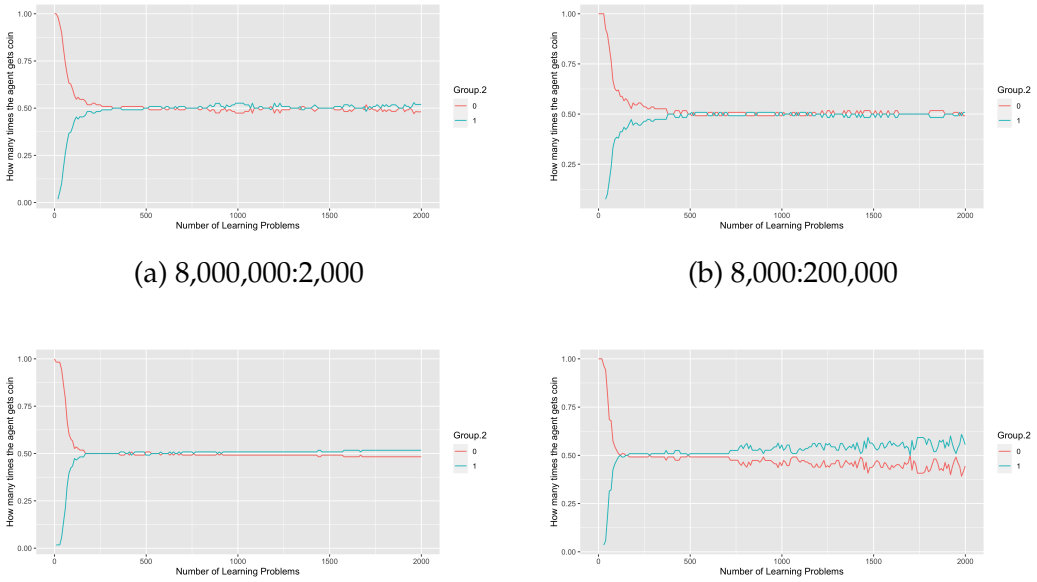
(a) 8,000,000:2,000

(b) 8,000:200,000

(c) 80,000:20,000

(d) 800,000:2,000

Figure 5.54: Percentage of the agent get to the final reward with different combination of Population vs. GA in Env1.



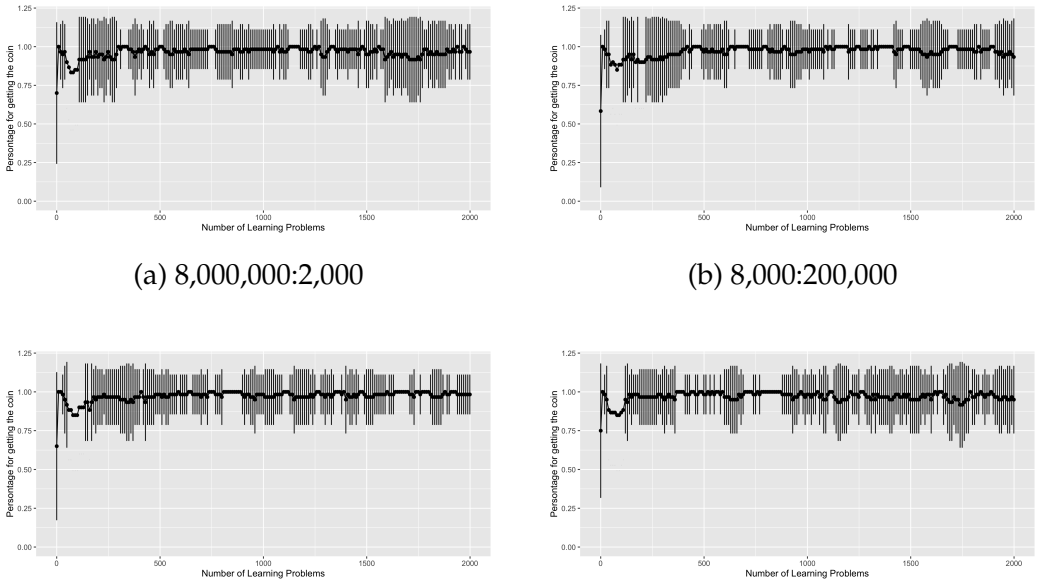
(a) 8,000,000:2,000

(b) 8,000:200,000

(c) 80,000:20,000

(d) 800,000:2,000

Figure 5.55: Ratio of different coin types obtained by the agent with different combination of Population vs. GA in Env1.



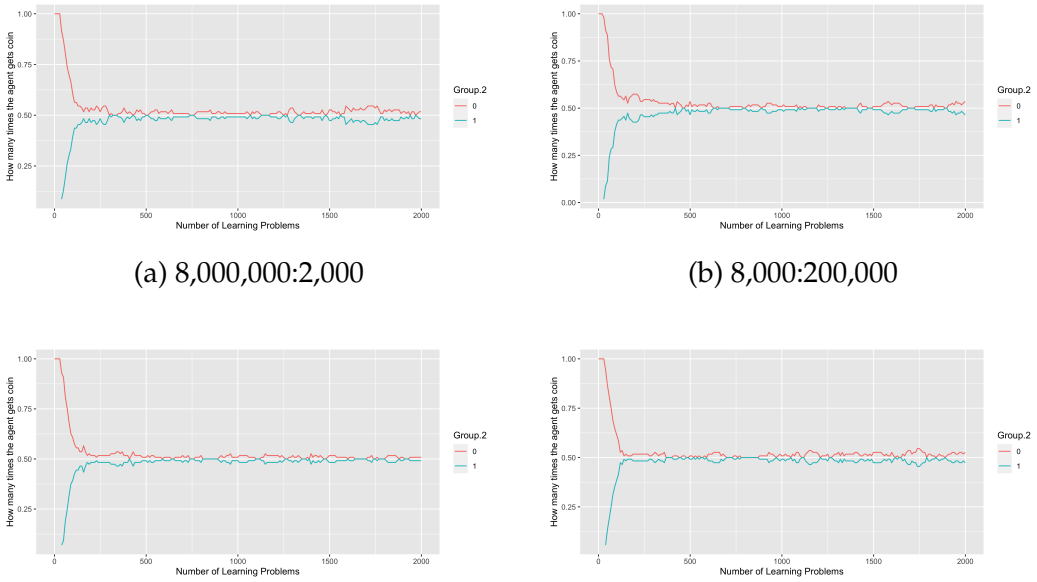
(a) 8,000,000:2,000

(b) 8,000:200,000

(c) 80,000:20,000

(d) 800,000:2,000

Figure 5.56: Percentage of the agent get to the final reward with different combination of Population vs. GA in Env5.



(a) 8,000,000:2,000

(b) 8,000:200,000

(c) 80,000:20,000

(d) 800,000:2,000

Figure 5.57: Ratio of different coin types obtained by the agent with different combination of Population vs. GA in Env5.

different coin types which should be 50% of coin0 and 50% of coin1 more frequently than 80,000:20,000. In this case, in Env1, for the stability measurement, the performance order of Population vs. GA is 8,000:200,000, 80,000:20,000, 8,000,000:2,000, 800,000:2,000.

Second, in Env5, as shown in Figure 5.56, for the measurement of the agent gets the final reward, the performance of Population vs. GA of 80,000:20,000 is better than 8,000:200,000, and they are better than the other two combinations in terms of the converging stability. The performance of Population vs. GA 800,000:2,000 is better than 8,000,000:2,000 because the measurement of reaching the final reward and reaching different coin types, the worst performance is the same, however, 800,000:2,000 is less frequently reach the worst performance than 8,000,000:2,000, especially after 1500 learning problems. In this case, in Env5, for the stability measurement, the performance order of Population vs. GA is 80,000:20,000, 8,000:200,000, 800,000:2,000, 8,000,000:2,000.

From the discussion above, the conclusions can be made. First, the requirement of MORL on population is not the larger the better. For example, a population size of 8,000 and 80,000 has a better performance than a population size of 800,000 and 8,000,000 in both Env1 and Env5. Second, a proper population with high GA works better than a high population with low GA. For example, in both Env1 and Env5, the population size of 8,000 and 80,000 are better than 800,000 and 8,000,000, while the GA of 200,000 and 20,000 are better than 2,000.

5.8.3 Generalization Ability

As the results are shown in subsection 5.6.1, MOXCS can generalize the action *right jump* to get the coin at the right-hand side in both 'CoinRun Steps VS Reward' Env1 and Env5. In this section, the experiments are designed to test if MOXCS can jump when necessary and if it can generalize other actions.

Jump Block

As the results in subsection 5.6.1, the agent trained by MOXCS is able to take action *right jump* to solve the 'CoinRun Action Bias' problem. However, we don't know if the agent only takes *right jump* when necessary. In this case, the environment is updated to see if the agent can jump over a block at different positions in the environment to achieve the goal and if the agent can jump only when there is a need for jump action rather than take *right jump* action all the time. The first goal can be measured by the ratio to reach coin, the ratio of getting coin1, and how many steps to reach coin1. To measure the second goal, the average number of actions going right, jump, and right jump in the optimal policies are counted.

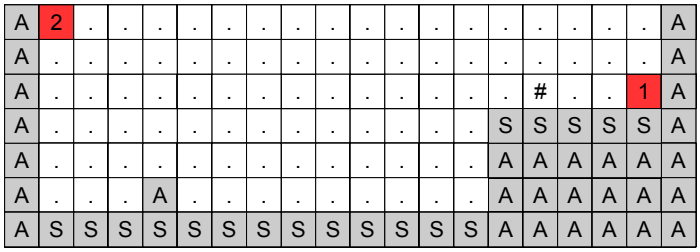


Figure 5.58: States of CoinRun Jump Block4 Environment 1(Env1).

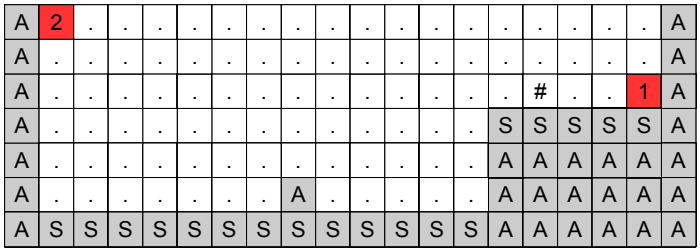


Figure 5.59: States of CoinRun Jump Block7 Environment 1(Env1).

The experiment environment is extended from 'CoinRun Steps vs. Re-

Table 5.5: The results of different settings of training testing blocks of Jump Block experiments.

Train Block	Test Block	% coin	% coin1	OP coin1	OP steps	R	J	RJ	LJ
4	4	0.816	0.387	1	12.51	4.9	0.637	4.9	0.619
7	7	0.816	0.387	0.894	12.3	5.669	0.635	5.26	0.675
4	7	0.66	0.275	1	12.1	6.27	0.616	4.45	0.612
7	4	0.816	0.387	1	12.8	5.817	0.634	5.73	0.604

ward' in Env1. In this environment, the length of the ground floor is extended from 4 to 13 units, and a block A is added on the 4th and 7th position of the ground level as shown in Figures 5.58 and 5.59. In addition, all the non Markov settings like T and N in subsection 5.3.2 are removed from this environment.

The parameter settings are derived from that of 'CoinRun Steps vs. Reward' in Env1, and population size N and niche GA θ_{ga} is 8000 and 200000, as it is the best parameter settings for Environment 1 from the result with 'Population vs. GA' experiment.

The data in results are collected at 500 learning problems and are shown as follows. In the table, some titles are shorted as limited space; for example, '%coin' is the percentage of the agent reaches one of the coins, '%coin1' denotes the percentage of the agent reaches coin1, 'OP coin1' denotes the optimal policy to coin1, which means the agent can get coin1 within 15 steps, 'OP steps' denotes how many steps in 'OP coin1' on average, 'R', 'J', 'RJ' and 'LJ' denotes how many 'go right', 'jump', 'right jump' and 'left jump' in the optimal policy to get coin1 'OP for coin1' on average.

From Table 5.5, several conclusions can be drawn. First, the agent trained by MOXCS can jump over a block at different positions and get the coin. With 500 learning problems, when the block is in the same position for training and testing, the agent can get one of the coins in 81.6%

of the time, and the percentage to get coin1 is 38.7%. The percentage of an optimal policy when setting the block on the 4th and 7th unit is 100% and 89.4%, and it may be because the agent takes more steps to the 7th unit. Thus the search space is larger than that of setting the block on the 4th unit. When the block is in a different position for training and testing, for training on the 4th block and testing on the 7th block, the agent can get one of the coins in 66% of the time, the percentage to get coin1 is 27.5%. For training on the 7th block and testing on the 4th block, the agent can get one of the coins in 81.6% of the time the percentage to get coin1 is 38.7%, as when the block is set at the 7th unit, the problem is trickier. Second, the agent trained by MOXCS can jump when there is a need for jump action rather than take *right jump* action all the time. When training and testing in the same environment, the average steps in the optimal policy is 12.51 and 12.3 for setting block on 4th and 7th unit, and the average number of action 'go right', 'jump', 'right jump' and 'left jump' are 4.9, 0.637, 4.9, 0.619 and 5.669, 0.635, 5.26, 0.675 for setting block on 4th and 7th unit respectively. The average steps in the optimal policy are 12.1 and 12.8 for setting block on 4th and 7th unit for training, 7th and 4th unit for testing when different block settings for training and testing environment. The average number of action 'go right', 'jump', 'right jump' and 'left jump' are 6.27, 0.616, 4.45, 0.612 and 5.817, 0.634, 5.73, 0.604 for setting block on 4th and 7th unit for training, 7th and 4th unit for testing respectively. From all the testings, the agent takes most action 'go right', then is 'right jump', which means the agent does not take *right jump* all the time. At the same time, the agent takes a jump and left jump as well, which means the agent can choose different jump types when there is a need.

Jump Left

In this section, to test if the agent trained by MOXCS can or has the potential to generalize other actions, like 'jump', 'go left', and 'jump left'. The data in results are collected with 100 learning problems at this stage.

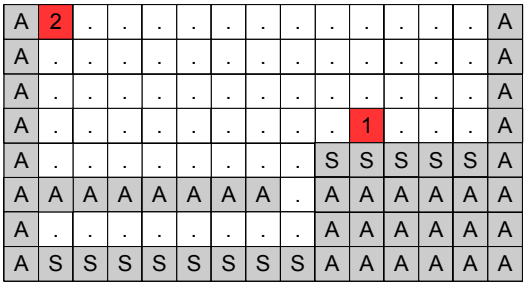


Figure 5.60: States of CoinRun Jump Left Environment 1(Env1).

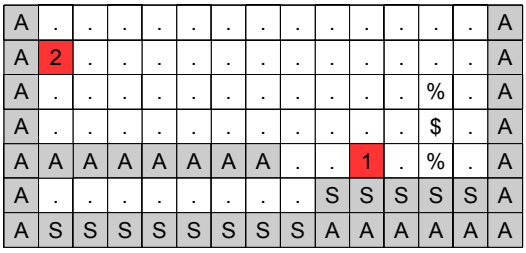
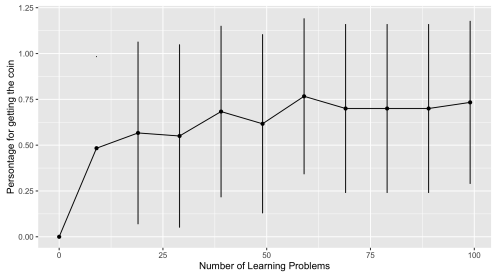
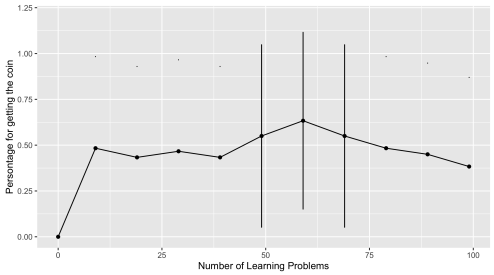


Figure 5.61: States of CoinRun Jump Left Environment 5(Env5).

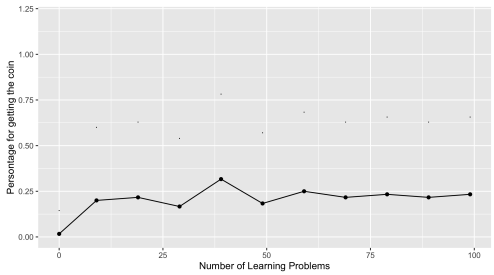
The experiment environment is extended from 'CoinRun Steps vs. Reward' in Env1 and Env5, with several changes as shown in Figures 5.60 and 5.61. First, the length of the ground floor is extended from 4 to 7 units. Second, a second layer consisting of 7 blocks A is added, where starts from the left and 2 units higher than the ground floor. Last, for the reward settings, the position of the first coin moves 3 units closer to the start state, and the reward value of those two coins are exchanged, where the coin on the right-hand side has a small value and the coin on the top of the start state has a large value.



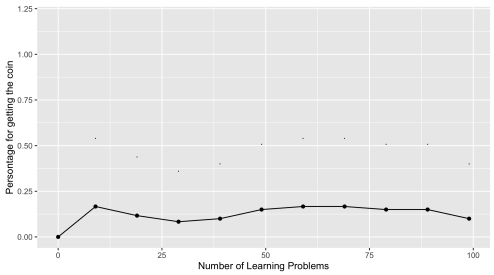
(a) Env11



(b) Env55



(c) Env15

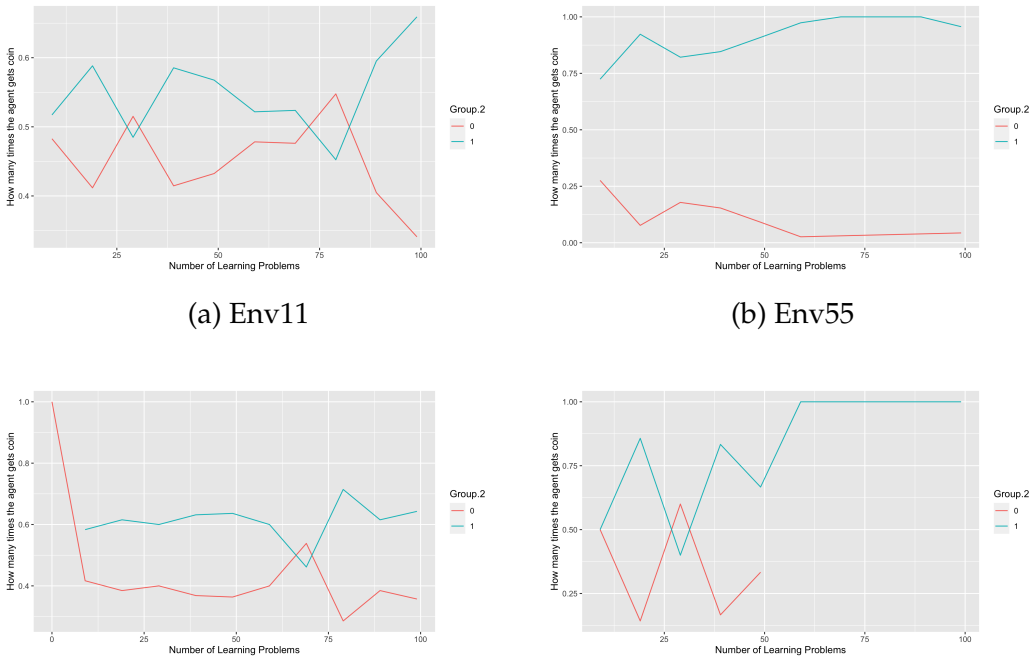


(d) Env51

Figure 5.62: Percentage of the agent get to the final reward in different training and testing environments of experiment Jump Left.

The parameter settings are similar to that of 'CoinRun Steps vs. Reward' in Env55, but with some differences. More details, when training and testing in the same environment, the population size N and niche GA θ_{ga} is 80000 and 20000, the payoff decay rate γ is 0.93, the mutation rate μ is 0.05, the fitness f for the first objective and second objective is 0.00000001 and 0.0005 respectively. Comparing the parameters in Env55 of 'CoinRun Steps vs. Reward' experiment, when training and testing in different environments, the difference of the parameters as follows, the population size N and niche GA θ_{ga} is 8000 and 200000, the probability of generating a hash in a condition $P_{\#}$ is 0.6, the error e and the fitness f for the first objective 0.2 and 0.0001.

The percentage for the agent to get the final reward is shown in Figure 5.62. From the result, we can see that the performance of training



(a) Env11

(b) Env55

(c) Env15

(d) Env51

Figure 5.63: Ratio of different coin types obtained by the agent in different training and testing environments of experiment Jump Left.

and testing in the same environment is much better than that of training and testing in different environments. In all the testing environments, the performance increases dramatically to 0.48, 0.48, 0.20, and 0.16 within 10 learning problems in Env11, Env55, Env15, and Env51 respectively. After that, the trend of the performance in different environments is getting more stable, the best performance in those four environments are 0.76, 0.63, 0.25, and 0.16. Though the performance in Env15 and Env51 is not as good as that in Env11 and Env55, it shows the potential of MOXCS to solve the 'Jump Left' problem with its generalization ability.

The ratio of getting coin0 is much higher than that of coin1 in all the environments as shown in Figure 5.63, where the coin0 is the coin with a large reward on the left-hand side and coin1 is the coin with a small reward on the right-hand side. It shows that the performance of learning

Test Env	OP for coin0	OP avg steps	R	J	RJ	L	LJ	J%	L%	LJ%
11	1	10.42	3	0.72	0.76	2.66	2.38	6.91	25.53	22.84
55	1	10.73	3	0.57	0.64	1.96	3.61	5.31	18.27	33.64
15	1	10.69	3	0.58	0.71	1.51	4.31	5.43	14.13	40.32
51	1	10.58	3	0.41	1	1.89	3.95	3.88	17.86	37.33

Table 5.6: The optimum policies for reaching coin0 of Jump Block experiments.

Test Env	OP for coin1	OP avg steps	R	J	RJ	L	LJ	J%	L%	LJ%
11	1	6.84	3.93	0.38	0.99	0.02	0.36	5.56	0.29	5.26
55	1	7.58	4.41	0.25	1.16	0.33	0.45	3.30	4.35	5.94
15	1	7.01	3.79	0.35	2.01	0.03	0.35	4.99	0.43	4.99
51	1	6.92	4	0.53	0.84	0.07	0.61	7.66	1.01	8.82

Table 5.7: The optimum policies for reaching coin1 of Jump Block experiments.

the optimum policies by maximizing the long-term payoff of MOXCS is better than minimizing the steps. However, as the goal of this experiment is to test if the agent can generalize actions 'jump', 'go left', and 'jump left', we will focus on the learned policies by maximizing the long-term payoff for coin0 at this stage.

The percentage of optimum policies to reach coin0 and coin1, average steps in those optimum policies, and how many 'go right' (R), 'jump' (J), 'right jump' (RJ), 'left' (L), 'left jump' (LJ), percentage of 'left' (L%), and percentage of 'left jump' (LJ%) in those optimal policies on average are shown in Tables 5.6 and 5.7.

From the results in Tables 5.6 and 5.7, we can see that when the agent has collected coin0 or coin1, it collects the coin with the optimum poli-

cies (within 15 steps) at 100% of the time in all the environments, and the number of steps in the optimum policies for coin0 and coin1 is over 10 steps and about 7 steps. The ratio of action 'left' and 'left jump' in the optimum policies for coin0 is much higher than that for coin1 as the agent needs to go left to reach coin0. For example, the sum of the ratio of action 'left' and 'left jump' in the optimum policies for coin0 is over 50% (except Env11, where is 48.37%), but that for coin1 is maximum 10.29%. Obviously, with the training of MOXCS, the agent can generalize action 'left' and 'left jump' to collect coin0 as it needs to be collected with go left or jump left. For those testing environments except Env11, the agent takes most action 'left jump', then is 'left', after that is 'jump'. This is because if the agent moves to the middle of the environment, it can reach coin1 only by 'jump left'. In Env11, the agent takes most action 'left', then is 'left jump' and 'jump'. However, the ratio of 'left jump' is slightly lower than other environments, whereas the ratio of 'left' and 'jump' is slightly higher than other environments. This is because when the agent takes more 'left' to get coin1, it needs to take more 'jump'. In addition, the difference in the performance of getting the final reward of Env11 and Env15 is higher than that of Env55 and Env51 is because the pattern on the action distribution (L, LJ) in the optimum policies is more similar of Env55 and Env51 than that of Env11 and Env15.

From the discussion above, several conclusions can be made from the 'Jump Left' experiment. First, MOXCS can solve the 'Jump Left' problem when training and testing in the same environment, and it can solve the problem when training and testing in different environments with its generalization ability. Second, though MOXCS can learn the optimum policies by maximizing the long-term payoff and minimizing the steps, the performance of learning the optimum policies by maximizing the long-term payoff is much better than minimizing the steps. More importantly, with the training of MOXCS, the agent can generalize action 'left' and 'left jump' when the coin needs to be collected with go left.

5.8.4 Stochastic Decision Making

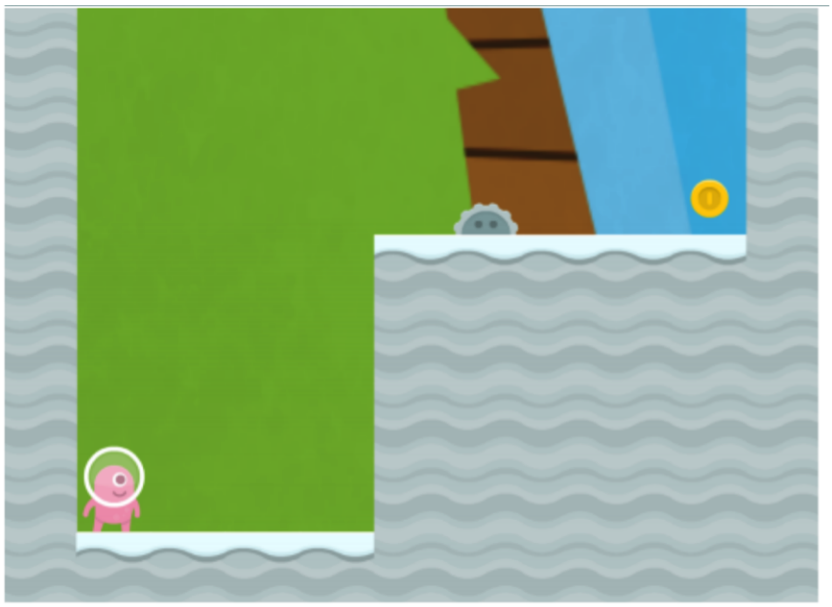


Figure 5.64: Monster in CoinRun Environment (Stationary).

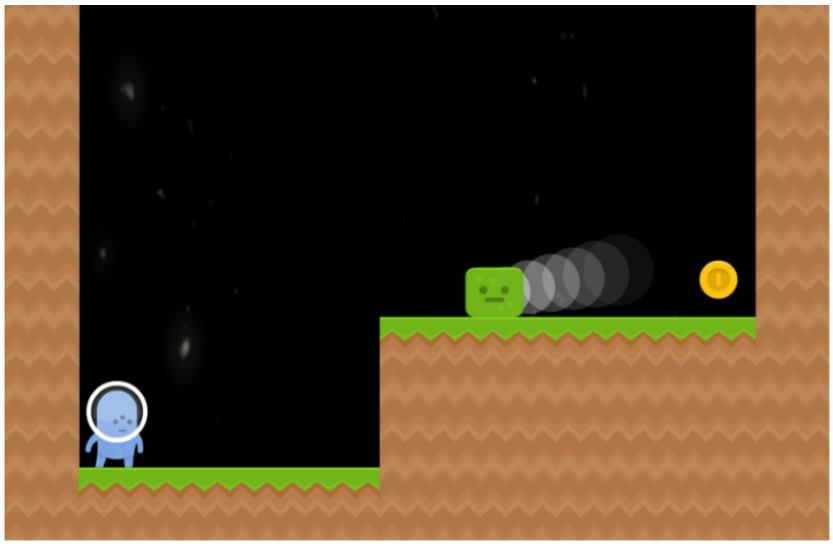


Figure 5.65: Monster in CoinRun Environment (Non-Stationary).

As mentioned in Section 5.2, in some of the CoinRun environments, there are several obstacles (monsters), both stationary and non-stationary, lie between the agent and the coin. A collision with an obstacle results in the agent's immediate death. In this case, the game with obstacles becomes more challenging than that without obstacles.

For the stationary obstacles in Figure 5.64, the monster can be coded as a special character in the environment, then encoded in the condition for LCSs. When the agent moves to a state close to the monster, for example, the monster is on the right-hand side of the agent, then the agent can learn to take an action jump or right-jump to avoid the monster but need to take action at good timing.

The solution for the non-stationary obstacles in Figure 5.65 is challenging to solve. Firstly, most of the LCSs are rule-based machine learning technologies designed to learn deterministic policies. However, the solution policy may often be stochastic. For example, the agent needs to learn to take different actions at the same position, which is not possible using an accurate-based system. In more detail, at any state S, when the monster is not close to the agent, the agent should take action to go right. However, if the monster is close to the agent, it needs to take an action jump or right jump to avoid the monster. Secondly, most of the LCSs can only solve the discrete RL problem. Especially for the methodology in Section 5.3, when using XCS to solve CoinRun with sub-actions, if the monster moves continuously, the agent can easily die when taking the sub-actions as it only senses the environment before taking the main action and ignores the environment before taking sub-actions. It would need allocentric sensors to be added to detect the dynamic monster.

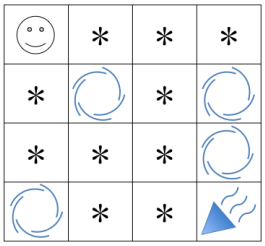


Figure 5.66: Frozen Lake Environment from OpenAI.

Following the research, the Frozen Lake problem is considered as a benchmark to evaluate if XCS can solve the Markov RL problem. The goal of this game is to go from the starting state to the goal state by walking only on frozen tiles and avoiding holes. However, the ice is slippery, so the agent will not always move in the direction as intended. After that, the multi-objective non-stationary RL can be designed as adding another goal state, which is closer to the start state but with less reward.

5.8.5 Measurement of Broad Generalization Ability

Though CoinRun provides an environment to measure the generalization of RL algorithms, it is still challenging to define and evaluate intelligence, especially comparing the intelligence between systems and humans. There are two historical definitions of intelligence, one defines intelligence as a collection of task-specific skills, and another one defines intelligence as a general learning ability [16]. In the AI industry, the current trend is to evaluate the task-specific skills, rather than trying to measure the broad generalization ability, such as developer-aware generalization [16]. The developer-aware generalization is the expectation a system can handle situations that neither the system nor the developer of the system has encountered before. For example, in the CoinRun environment, the

generalization ability of RL algorithms can be tested in unseen generated game levels, but it is still task-specific local generalization as it only evaluates a system in a known distribution rather than a new task.

ARC (The Abstraction and Reasoning Corpus) is developed as a general artificial intelligence benchmark [16] to measure the developer-aware generalization ability. There are 400 tasks in the training set and 600 tasks in the testing set, and each task consists of a small number of demonstration examples and a test example. Each example consists of an 'input grid' and an 'output grid'. The test-taker should learn the pattern from the 'input grid' and an 'output grid' of the demonstration examples and draw the output to the 'output grid' in the test example. Even though there are training and testing sets in the ARC problem, the test-taker is not necessarily trained on the complete sample distribution of the training set, as all the tasks are assumed core knowledge priors such as reasoning and abstraction abilities, which cannot be learned from the training set. In this case, the ARC system can focus on testing the broad generalization ability of AI systems and humans.

Inspiring by the idea of the ARC system, instead of testing the generalization ability of MOXCS in CoinRun game, to test the broad generalization ability of MOXCS algorithm, the MOXCS can be trained in CoinRun game to learn the 'core knowledge priors' such as *go right* when the path is *safe*, and *jump* when encounters obstacles, and tested in another game such as bi-objective Maze or Deep Sea Treasure Corridor in the future work.

5.9 Chapter Summary

CoinRun Env1 and Env5 were successfully addressed by XCS with the technique of discretizing continuing input, adding extra characters in the aliased environment, and utilizing sub-actions. In this case, the PO-MDP is transformed as an MDP and is possible to be addressed by MOXCS.

Then Env1 and Env5 are extended as bi-objective RL environments. In

the experiment 'CoinRun Action Bias', where the first objective is to take as many actions right-jump to get the coin as possible, and the second objective is to minimize the total steps to get the coin. In the experiment 'CoinRun Step VS Reward', the first objective is to minimize the total steps to get any coin, and the second objective is to reach the coin with a larger reward. MOXCS managed to resolve these two bi-objectives successfully within 50 learning problems and 500 learning problems, respectively. In the experiment 'CoinRun Action Bias', the first objective is to take as many as actions *right jump* to get the coin, and the second objective is to reach the coin with a larger reward. MOXCS managed to resolve these two bi-objectives successfully within 50 learning problems.

The generalization ability of MOXCS is also tested on these two experiments, 'CoinRun Action Bias' and 'CoinRun Step VS Reward', by crossover the training and testing environments within each experiment. MOXCS is not able to fully solve the generalization problem. However, the evidence shows MOXCS has the potential of generalization ability for solving the MORL problem in an unseen environment with proper training in a similar environment.

In the discussion, first, how the PF looks like in experiment 'CoinRun Action Bias' and 'CoinRun Step VS Reward' is discussed. As the PF in these two experiments is naturally sparse as shown in Figures 5.48, 5.49 and 5.50, the future work for MOXCS is designed to learn a more complicated and consistent PF as shown in Figure 5.53. Second, the different combinations of population size and the threshold for performing the niche GA are tested on 'CoinRun Steps vs. Reward' to see if MORL needs a large amount of good generic material to begin to function. The result shows the requirement of MORL on population is not the larger the better, and a proper population with high GA works better than a high population with low GA. Then, the generalization ability of MOXCS is tested in experiments 'Jump Block' and 'Jump Left'. The 'Jump Block' experiment shows that MOXCS can generalize the action right jump to get the coin at

the right-hand side when necessary. The 'Jump Block' experiment shows that MOXCS has the potential to generalize the action 'go left' and 'jump left' when the coin needs to be collected with go left.

In addition, in future work, to enable MOXCS to solve environments with non-stationary obstacles in the CoinRun environment, stochastic decision making will be considered to add into MOXCS. To evaluate the ability of XCS and MOXCS to solve large-scale RL and MORL problems, XCS and MOXCS will be tested in mountain car and multi-objective mountain car in the next chapter.

Chapter 6

Using MOXCS to Solve Large-scale MORL Problems

6.1 Introduction

In Chapter 4, the Deep Sea Treasure is solved successfully by MOXCS. However, the Deep Sea Treasure is a small discrete MORL environment, where the optimal policy can calculate manually. However, in the real world MORL problems, one of the common challenges is that the state space can be very large and continuous. In this case, it is challenging as learning can become intractably slow as the state space of the environment grows. Note, in this thesis, a small-scale problem means the solution of the problem can be manually calculated easily, and a large-scale problem means the solution of the problem is hard to calculate manually (i.e. over a day's labour).

6.1.1 Chapter Goals

This chapter is intended to use MOXCS to address the continuous MORL problem in a large sparse domain. Currently, following the MOXCS technique, it is supposed with the generalization capability of MOXCS, the

agent can effectively learn in a sparse domain environment. To deal with the continuous MORL problem, it is required to address the continuous inputs for MOXCS by effectively discretizing the continuous input. In this chapter, to evaluate the learning effectiveness of XCS and MOXCS for solving RL and MORL problems, the experiments will be implemented on single-objective mountain car and multi-objective mountain car. The results will be presented and discussed in this chapter as well.

6.1.2 Organization

This chapter is structured as follows. Section 6.2 presents the mountain car problem and multi-objective mountain car problem as a benchmark for testing the single-objective and multi-objective reinforcement learning algorithm. Next, Section 6.3 presents a methodology that uses XCS and MOXCS to solve single-objective and multi-objective mountain car problems. The results and discussion are given in Section 6.5 and Section 6.6 respectively. The chapter is finally summarized in Section 6.7.

6.2 Problem Description

6.2.1 Mountain Car Problem

The mountain car problem is commonly a benchmark for RL algorithms. In this section, first, the Mountain Car problem is introduced, then it will be extended as a MORL problem and described in the next subsection.

Mountain Car

As shown in Figure 6.1, the Mountain Car is a problem in which a car must learn to climb a steep hill to reach the goal marked by a flag. Since the car's engine is not powerful enough, even at full speed, the car does not have enough momentum to reach the goal. The car is situated in a valley and

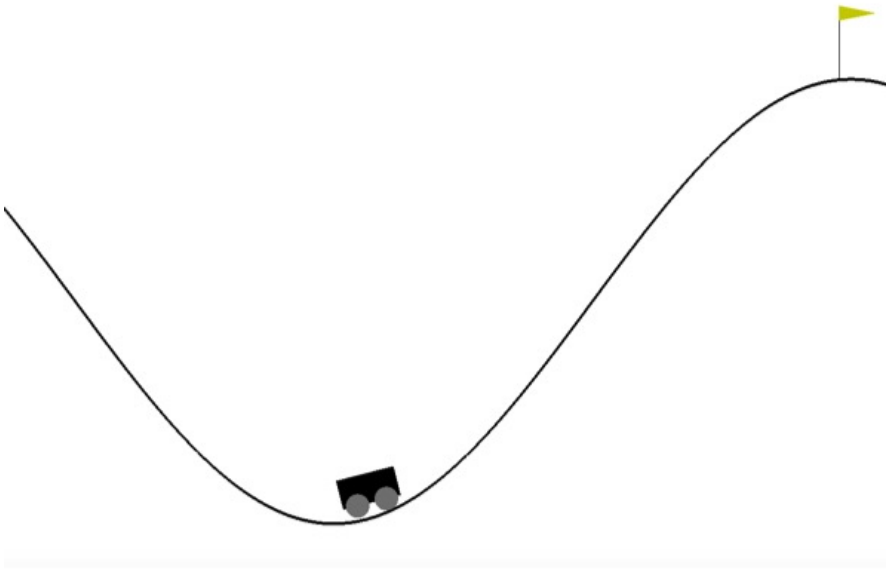


Figure 6.1: Mountain Car problem.

must learn to leverage potential energy by driving up the opposite hill before the car can make it to the goal at the top of the rightmost hill.

The mountain car problem has undergone many iterations. This section will focus on the standard well-defined version from Sutton in 2008 [77]. The problem begins when the car is dropped into the valley and given an initial position from -1.2 to 0.6 and velocity from -0.07 to 0.07 as a vector. This is the car's state. Our agent must then tell the car to take one of three actions: reversing(-1), acceleration(1), or zero-throttle(0). This action is sent to the Mountain Car environment algorithm which returns a new state (position and velocity) as well as a reward. For each step that the car does not reach the goal, the environment returns a reward of -1. The termination condition is the car's position $>=$ 0.6.

Here are the technical details of the mountain car problem.

State variables

Two-dimensional continuous state space.

$Velocity = (-0.07, 0.07)$

$Position = (-1.2, 0.6)$

Actions

One-dimensional discrete action space.

$motor = (left, neutral, right)$

Reward

For every time step:

$reward = -1$

Update function

For every time step:

$Action = [0, 1, 2]$

$Velocity = Velocity + (Action - 1) * 0.001 + \cos(3 * Position) * (-0.0025)$

$Position = Position + Velocity$

Starting condition

Optionally, many implementations include randomness in both parameters to show better generalized learning.

Goal: $Position \geq 0.6$

Multi-Objective Mountain Car

Mountain car is frequently used to test the performance of the single-objective reinforcement learning problem [76]. Here, the Mountain car problem is extended as a multi-objective mountain car. Besides the original goal, which is reaching the final state on the right side as soon as possible, another goal is added to make the mountain car problem the $bi - objective$ mountain car problem which is minimizing the number of reversing and acceleration actions. -1 is received in the corresponding element of the reward vector when the reversing and acceleration action is executed. In the case when the car takes $action_1$, it returns a reward of 0. Here are the technical details of the reward setting in the multi-objective mountain car problem.

Reward

For every time step:

$reward = -1\ if\ action = left\ or\ right$

6.3 Techniques Used to Solve Mountain Car

In this section, to evaluate the efficiency of different RL algorithms for solving RL and MORL problems, Q-Learning, XCS, and MOXCS are implemented to solve mountain car and multi-objective mountain car problems. Q-Learning is a standard benchmark for cooperation. However, those algorithms are normally used for solving discrete RL problems. In this case, to solve continuous RL problems, some changes need to be made in the environment and those algorithms.

6.3.1 Discretization

As Q-learning, XCS and MOXCS can only consume discrete input, the mountain car environment needs to be discretized. In this approach, two continuous state variables representing the position in space and velocity space are pushed into discrete states by bucketing each continuous variable into multiple discrete states. For example, the position space from -1.2 to 0.6 is discredited to 12 bins, namely (-1.2, -1.05], (-1.05, -0.9] until (0.45, 0.6]. Similarly, the velocity space, which is from -0.07 to 0.07, is discredited into 20 bins, namely (-0.07, -0.06] to (0.06, 0.07]. More details about how to discretize position and velocity are in Table 6.1 and 6.2, where the steps are discredited evenly. Q-Learning, XCS, and MOXCS, the velocity space matches integers 0 to 19, and the position space matches integers 0 to 11.

6.3.2 Transforming Integers to Conditions

In XCS and MOXCS, the position and velocity are converted from decimal to a 5-bit binary code respectively. Then, the binary code of position and velocity are merged as a 10-bit binary code, which consists of the condition of the classifiers of XCS and MOXCS.

Table 6.1: Position of Mountain Car

Continuous Input	Integer Input
(-1.2,-1.05]	0
(-1.05,-0.9]	1
(-0.9,-0.75]	2
(-0.75,-0.6]	3
(-0.6,-0.45]	4
(-0.45,-0.3]	5
(-0.3,-0.15]	6
(-0.15,0]	7
(0,0.15]	8
(0.15,0.3]	9
(0.3,0.45]	10
(0.45,0.6]	11

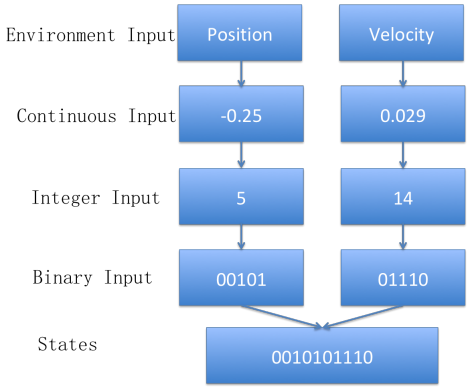


Figure 6.2: Representation of position and velocity in the condition of XCS

As shown in Figure 6.2, the position and velocity of the mountain car are two float numbers -0.25 and 0.029, then it will be converted to integer 5 and 14 respectively. After that, those two integers will be transferred to two 5-bits binary codes 00101 and 01110. Finally, those two 5-bits binary codes will be merged as 0010101110, which is the condition of the current

Table 6.2: Velocity of Mountain Car

Continuous Input	**Integer Input**
(-0.07,-0.063]	0
(-0.063,-0.056]	1
(-0.056,-0.049]	2
(-0.049,-0.042]	3
(-0.042,-0.035]	4
(-0.035,-0.028]	5
(-0.028,-0.021]	6
(-0.021,-0.014]	7
(-0.014,-0.007]	8
(-0.007,0]	9
(0,0.007]	10
(0.007,0.014]	11
(0.014,0.021]	12
(0.021,0.028]	13
(0.028,0.035]	14
(0.035,0.042]	15
(0.042,0.049]	16
(0.049,0.056]	17
(0.056,0.063]	18
(0.063,0.07]	19

state.

6.4 Experiment Settings

The result of the Single Objective Mountain Car will be calculated by averaging the results of 30 independent runs of Q-Learning and XCS, separately. In the experiment of XCS, the typical parameter settings recommended in [14] have been followed.

Particularly, the state length of the classifier's condition is 10, the probability of using a # in one attribute in condition $P_{\#}$ is 0.00001, the minimal number of actions that must be present in a match set θ_{mna} is 3, the GA threshold θ_{ga} is 8000000, the discount factor γ is 0.99, the population size N is 4000000, the initial error ε is 0.01, the initial fitness f is 0.01, the initial prediction value p is 0.01, the mutation probability μ is 0.04, the classifier deleting threshold θ_{del} is 25, and the subsumption threshold θ_{sub} is 20. In addition, all the child initial error equals ε. The system will delete the classifier when it is necessary.

θ_{ga} usually is about 25, but here is 8000000 as the results with high GA sometimes is better than small GA from the evidence in subsection 5.8.2, page 127. The discount factor γ is as 0.99, which is usually 0.71, as the policy in mountain car takes hundreds of steps, if discount factor is low, the final reward is hard to learn by the states further from the final state. The population size N is 4000000, which would normally be about 80,000. Such a high population size is because the search space in the mountain car is high.

The result of the Multi-Objective Mountain Car is collected from a successful run of the experiments as the experimental result is not very consistent. The parameters of the Multi-Objective Mountain Car are similar to the Single-Objective Mountain Car with some changes. First, the population size N is 80000000 as it needs more classifiers to maintain different objectives. Second, as there are two objectives in Multi-Objective Moun-

tain Car, the initial error for both objectives ε_0 and ε_1 is 0.01, the initial fitness for both objectives f_0 and f_1 is 0.01, the initial prediction value for both objectives p_0 and p_1 is 0.01.

6.5 Single-Objective Mountain Car

6.5.1 Result

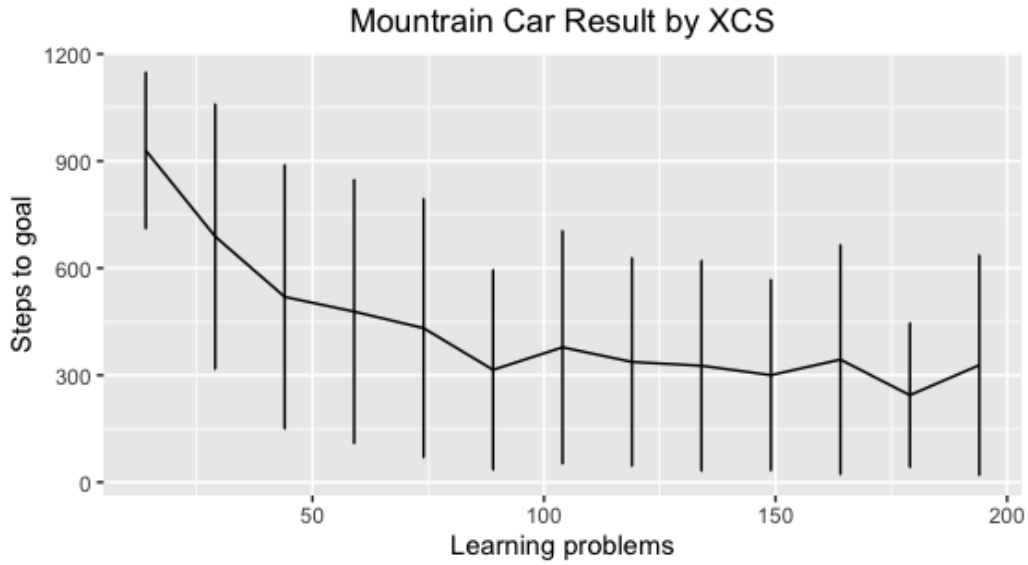


Figure 6.3: Single Objective Mountain Car Result with XCS.

Figure 6.3 shows the number of steps needed to reach the goal in each episode. As can be seen from the graph, the average number of steps for solving the mountain car problem is reduced quickly within 90 learning problems, where it drops from 929.7 to 315.0. It then increases to 378.1 at 105 learning problems, then drops to 300.2 slowly at 150 learning problems. After increases to 343.8 at 165 learning problems, it achieves the best performance of 244.0 at 180 learning problems, then increases to 328.1 at the 195 learning problems.

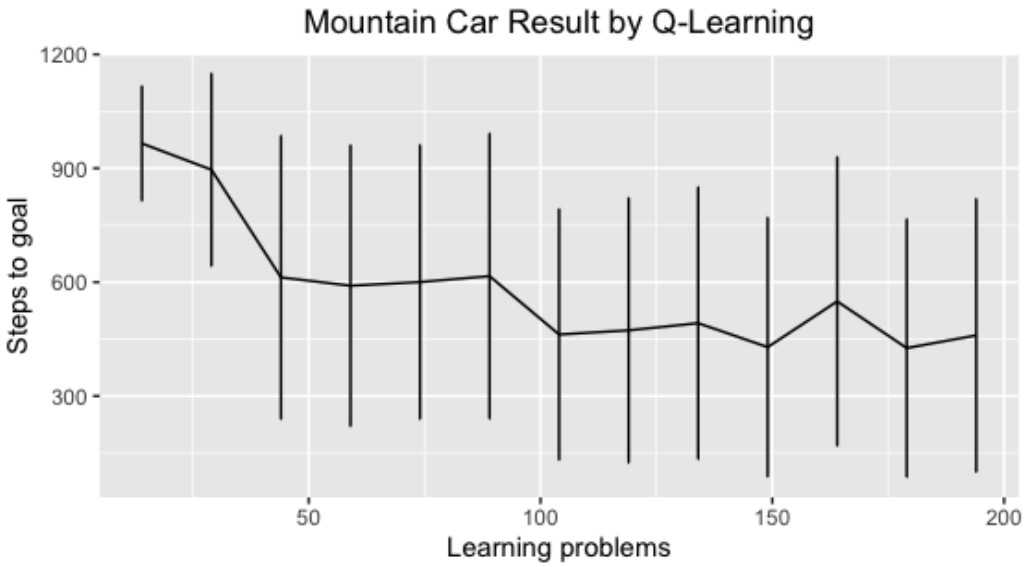


Figure 6.4: Single Objective Mountain Car Result with Q-Learning.

Q-Learning is used to solve the Mountain Car problem and the result is shown in Figure 6.4. As can be seen from the graph, the average number of steps for solving the mountain car problem is reduced from 965.4 to 462.1 within 105 learning problems, but the result stays around 600 from 45 to 90 learning problems. After increases to 491.9 at 135 learning problems, it drops to 428.8 at 150 learning problems, which is close to the best performance. However, the result jumps to 549.3 at 165 learning problems, then achieves the best performance of 426.3 at 180 learning problems. Finally, the result increases slightly to 459.8 at 195 learning problems.

From Figures 6.3 and 6.4, we can see that XCS provides a better solution than Q-Learning to Mountain Car problem. First, XCS converges faster than Q-Learning, where XCS and Q-Learning converge at 90 and 105 learning problems respectively. Especially, the result of XCS converges to 315.0 at 90 learning problems. However, the result of Q-Learning achieves 615.8 at 90 learning problems, then converges 462.1 at 105 learning problems. Second, XCS provides a more stable solution than Q-Learning after

they converge. For example, after 90 learning problems, the worst case of the average number of steps for solving the mountain car problem of XCS is 378.1 at 105 learning problems, but that of Q-Learning is 549.3 at 165 learning problems. Third, XCS provides a better solution than Q-Learning, where the best performance of the average number of steps for solving the mountain car problem of XCS is 244.0 at 180 learning problems and that of Q-Learning is 426.3 at 180 learning problems. In addition, in the single run, the best performance of the number of steps for solving the mountain car problem of XCS is 94 and that of Q-Learning is 114.

6.5.2 Discussion

As we can see the result in Section 6.5, XCS can solve the mountain car problem better than Q-Learning, especially when the result converges. For finding out the reason behind it, the results of steps to goal at 195 learning problems for XCS and Q-Learning are collected and counted in Table 6.3. In this case, it is possible to find out why XCS has a better performance than Q-Learning after they converge.

Optimum solutions

Note, the results for both XCS and Q-Learning are calculated by averaging the results of 30 independent runs, and there are 30 tests in each run, thus there are 900 testing for both of these two algorithms at 195 learning problems. As shown in Table 6.3, the steps to goal are divided into different ranges. The number of steps to goal from 100 to 199 can be considered as the number of steps with the optimum solution. The number of steps to goal from 200 to 299 can be considered as the number of steps with the sub-optimum solution, which shows the potential of the algorithm to solve the mountain car problem. In this case, the optimum solution and sub-optimum solution can be considered the valid solution. If the number of steps is between 300 to 999, the solution will be considered an invalid

Steps to goal	XCS	Q-Learning
100-199	490	229
200-299	227	347
300-399	15	37
400-499	9	8
500-599	2	2
600-699	3	3
700-799	3	2
800-899	0	2
900-999	0	0
1000	151	270

Table 6.3: Counted steps to goal of XCS and Q-Learning at learning problems 195.

solution. Finally, if the number of steps is 1000, it means the algorithm is not able to learn any policy. For comparing these two algorithms, it is quite clear that in 900 tests, XCS and Q-Learning can solve the problem with the optimum solution in 490 tests and 229 tests, respectively, thus XCS has a higher chance to learn the optimum solution. For learning the sub-optimum solution, there are 227 and 347 tests for XCS and Q-Learning achieves this goal. In this case, the valid solution learned by XCS and Q-Learning in 717 and 576 tests respectively. Obviously, XCS can learn more optimum solutions and valid solutions than Q-Learning. In addition, there are 151 and 270 tests that XCS and Q-Learning cannot learn any useful solution, where XCS is less likely to get involved in this worst case. From the analysis, we can see that XCS provides a better solution than Q-Learning as it converges better than Q-Learning for learning the optimal solution and is less likely to does not learn any useful solution.

In Table 6.4, the steps to goal of the optimum solutions less than 200 steps are divided into different ranges by the number of steps. The number

Steps	XCS (N)	XCS (a0)	XCS (a1)	XCS (a2)	Q-L (N)	Q-L (a0)	Q-L (a1)	Q-L (a2)
90-100	2	0.40	0.03	0.56	0	0	0	0
100-110	9	0.26	0.11	0.63	4	0.35	0.21	0.44
110-120	26	0.23	0.18	0.58	6	0.34	0.24	0.42
120-130	9	0.25	0.18	0.56	9	0.35	0.21	0.43
130-140	53	0.27	0.20	0.53	5	0.36	0.20	0.44
140-150	170	0.33	0.23	0.44	10	0.33	0.24	0.43
150-160	108	0.34	0.22	0.44	79	0.36	0.24	0.40
160-170	41	0.29	0.24	0.47	48	0.33	0.23	0.45
170-180	21	0.28	0.28	0.44	42	0.31	0.26	0.43
180-190	53	0.27	0.29	0.44	26	0.31	0.25	0.44
190-200	48	0.27	0.27	0.46	46	0.29	0.28	0.43

Table 6.4: The ratio of taking different actions in the optimum solutions learned by XCS and Q-Learning by each steps to goal range.

of solutions in each range is counted, and the ratio of taking a_0, a_1, and a_2 in each range by XCS and Q-Learning is calculated. For XCS, first, it learns more optimum solutions within 140 steps than Q-Learning. More details, for the optimum solutions with less than 140 steps, there are most action go right a_2 in it (over 0.53), then is action go left a_0 and zero-throttle a_1. Note, the most optimum solution learned by XCS takes 0.56 of a_2, and only 0.03 of a_1, which means when there are more a_2 and less a_1, the mountain car can get the flag on the right-hand side of the hill faster. The reason why XCS can learn the optimum solutions with more a_2 is due to it is generalize a_2 at different positions. Second, there are 170 (between 140 and 150 steps) and 108 (between 150 and 160 steps) optimum solutions learned by XCS, which is clear evidence for the converge. It may be because XCS can maintain the optimum solutions with its generalization ability, then slight changes in the inputs would not affect the results very much. For Q-Learning, there are not many optimum solutions learned by it with less than 140 steps. The ratio of taking action a_0, a_1, and a_2 in the learned optimum solutions by Q-Learning is about over 0.3, 0.2, and 0.4, thus it is less likely to learn the optimum solutions with more a_2. In addition, there is no clear evidence to show the result of Q-Learning converges.

From the discussion, we can see that XCS has a better solution than Q-Learning for solving the mountain car problem due to its generalization ability to generalize action go right a_2 in the optimum solutions, and maintain the optimum solutions.

6.6 Multi-Objective Mountain Car

As shown in Table 6.4, the ratio of taking action a_1 zero-throttle in the most optimum solution with 94 steps has only 0.03 of a_1. In order to see the trade off of using more zero-throttle and increasing total steps in the optimum solutions, the experiment with 3 weights [1, 0], [0.5, 0.5] and [0, 1] are implemented.

Weight	Result	zero-throttle
[1, 0]	164	26
[0.5, 0.5]	169	43
[0, 1]	305	119

Table 6.5: Multi-Objective Mountain Car Result.

6.6.1 Result

In this subsection, only one of the results of the successful run shows in Table 6.5 as the experimental result is not very consistent. From the result in that Table, we can conclude that with 200 learning problems, the bi-objective mountain car problem can be solved by MOXCS with three weights. With weight [1, 0] and [0.5, 0.5], the mountain car achieves the goal with 164 and 169 with 26 and 43 zero-throttle, respectively. With the weights [0, 1], the number of action zero-throttle is higher than the weight [1, 0] and [0.5, 0.5], which makes sense for the second objective.

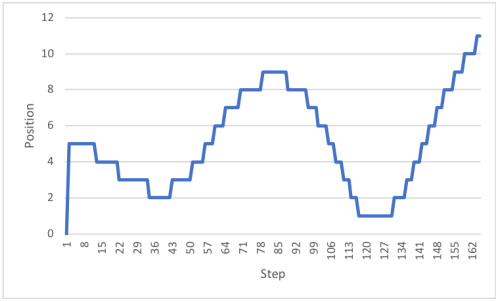
6.6.2 Discussion

Step VS Position

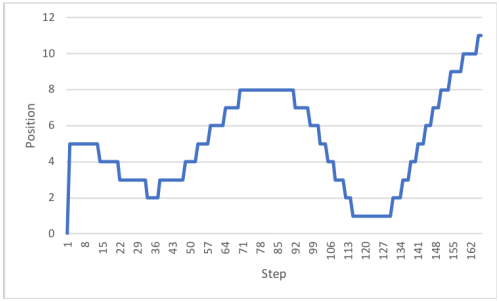
As shown in Figure 6.5, MOXCS spends over 160 steps with weights [1, 0] and [0.5, 0.5], and over 300 steps with weight [0, 1] for solving the multi-objective mountain car problem. There is not much difference in the position of weights [1, 0] and [0.5, 0.5], however, the mountain car with weight [0, 1] moves back and forward more to gain enough momentum than other weights to solve the problem.

Step VS Velocity

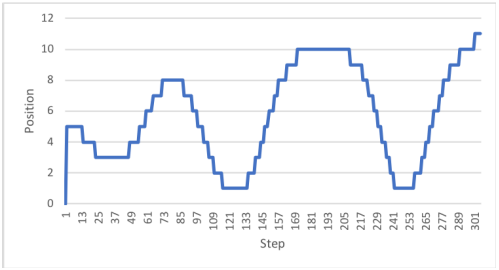
Similar to the position, there is not much difference in the velocity of weights [1, 0] and [0.5, 0.5] as shown in Figure 6.6, they achieve the veloc-



(a) Weight [1, 0]

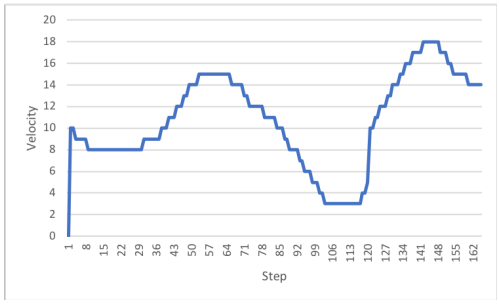


(b) Weight [0.5, 0.5]

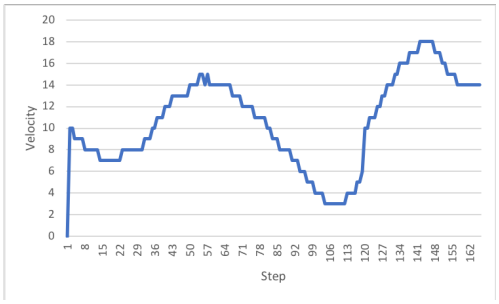


(c) Weight [0, 1]

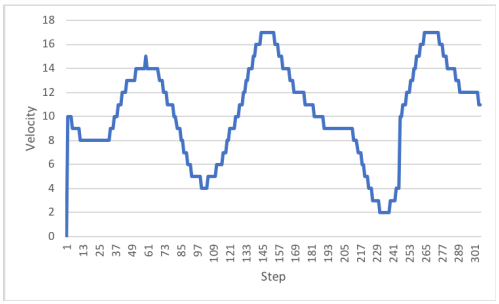
Figure 6.5: MOXCS Learning performance, i.e. The position of each step with different weights.



(a) Weight [1, 0]

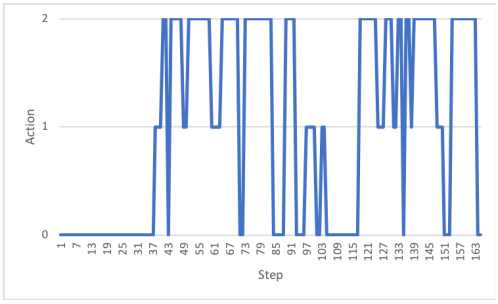


(b) Weight [0.5, 0.5]

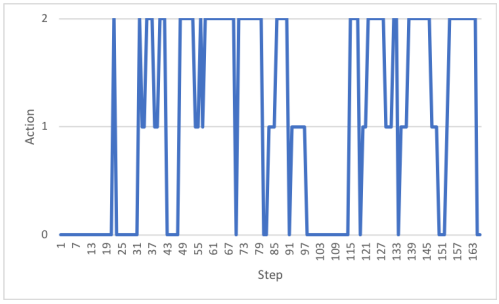


(c) Weight [0, 1]

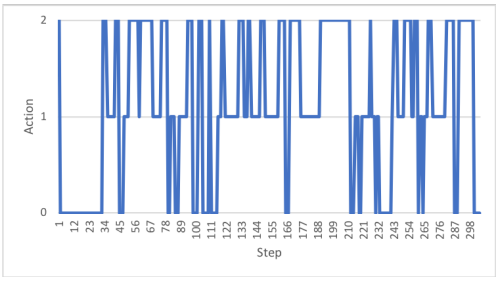
Figure 6.6: MOXCS Learning performance, i.e. The velocity of each step with different weights.



(a) Weight [1, 0]



(b) Weight [0.5, 0.5]



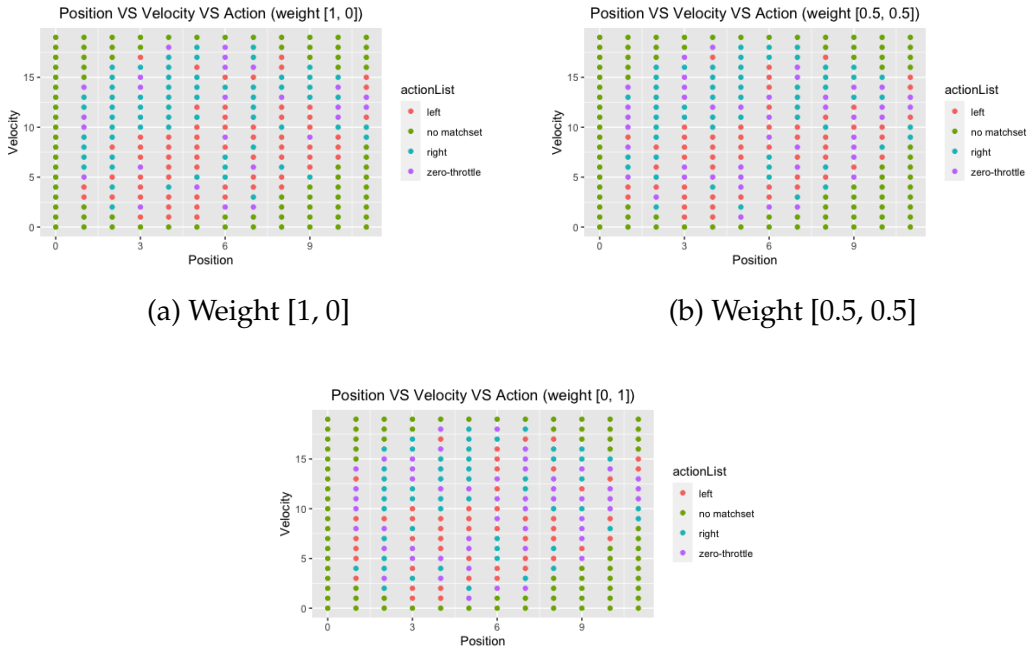
(c) Weight [0, 1]

Figure 6.7: MOXCS Learning performance, i.e. The action of each step with different weights.

ity of 3 at over 100 steps, which enable them to gain enough momentum to reach the goal on the right-hand side with over 160 steps. However, for weight [0, 1], after it achieves the velocity of 4 at over 100 steps, it still does not have enough momentum to reach the goal on the right-hand side. In this case, it has to back to the left to gain enough momentum to reach the goal on the right-hand side.

Step VS Action

As shown in Figure 6.7, the actions with weights [1, 0] and [0.5, 0.5] is more similar and consistent than weight [0, 1]. More details, for weights [1, 0] and [0.5, 0.5], the mountain car takes action left a_0 first, whereas the mountain car takes action right a_2 first. For the consistency of the actions,



(a) Weight [1, 0]

(b) Weight [0.5, 0.5]

(c) Weight [0, 1]

Figure 6.8: MOXCS Learning performance, i.e. The action distribution with different weights.

for weights [1, 0] and [0.5, 0.5], the mountain car takes action right a_2 more consistently. In addition, the action zero-throttle a_1 with weight [0, 1] are much more than other weights.

Heat map

As it is shown in Figure 6.8, the actions at different states with those 3 weights are plotted. There are three actions go left, go right, and stay, which are colored by red, blue and purple. The state plots with green mean there is no match set for that state. In the plot, the ideal actions should be taking actions right on the upper part of the figure, and actions left on the second half. With weights [1, 0] and [0.5, 0.5], the action of going right on the upper part of the figure is 46 and 42, the action of going left on the

second half of the figure is 45 and 38. With weight [0, 1], the action of going right on the upper part of the figure is 35, the action of going left on the second half of the figure is 35. In this case, with weights [1, 0] and [0.5, 0.5], there are more ideal actions taken by the mountain car than that of weight [0, 1], thus the mountain car problem is solved faster with weights [1, 0] and [0.5, 0.5]. For the distribution of action zero-throttle, there are 26, 43, 53 with weights [1, 0], [0.5, 0.5], [0, 1] respectively. From the distribution of action zero-throttle, we can see that when there are less zero-throttle and fewer steps in the solution when the objective is to minimize steps to reach the goal with weight [1, 0], and there are more zero-throttle when the objective is to reach the goal with more zero-throttle and more steps in the solution with weight [0, 1]. When considering both of the objectives with weight [0.5, 0.5], the steps in the solution is just 5 steps higher than weight [1, 0], but there are 17 actions zero-throttle than that of weight [1, 0], which shows MOXCS can maintain both of the objectives with a proper weight.

6.7 Chapter Summary

In this chapter, MOXCS is used to solve the continuous RL and MORL problems in a large sparse domain. First, XCS and Q-Learning are implemented to resolve the mountain car problem, and XCS converges better than Q-Learning. Especially, for the best performance of the average steps to goal, XCS and Q-Learning spend about 244 steps and 426 steps to resolve the mountain car problem. The generalization ability of XCS is the key reason for XCS provides a better solution than Q-Learning, as it generalizes action goes right to enable the mountain car to gain enough momentum to reach the goal at the right-hand side, in this case, it helps the mountain car to learn the optimum solutions with more action go right. In addition, the generalization ability also helps maintain the optimum solutions.

Then, the mountain car is extended as a bi-objective MORL problem by adding another objective to push the mountain car with more action zero-throttle. In this case, there are two objectives, the first one is minimizing the steps to achieve the final state and the second one is to reach the final state with as many zero-throttle actions as possible. Only when the mountain car takes zero-throttle, it gets a reward of 0, otherwise -1. MOXCS is implemented to resolve the multi-objective mountain car problem, but the result is not consistent. In this case, a positive case in the results is analyzed to show MOXCS has the potential to resolve the multi-objective mountain car problem. In this positive case, it spends 164, 169 and 305 steps to resolve the problem with weight [1, 0], [0.5, 0.5] and [0, 1]. Though the steps in the valid solution with weights [0.5, 0.5] and [0, 1] are higher than the weight [1, 0], there is more action zero-throttle in these two weights, which meets the goal of the second objective.

XCS and MOXCS perform better than Q-Learning as XCS and MOXCS have a better generalization ability than Q-Learning.

Chapter 7

Conclusions and Future Work

The main academic issue addressed by this thesis is to solve the discrete MORL problems utilising the LCS technique. There are four objectives in the thesis. The first two of them are discrete problems, in Chapters 3 and 4. The other two of them are transferring continuous problems to discrete problems by changing the environments then using the technology of solving discrete problems to solve them, in Chapters 5 and 6. Similarly, the partially observable problem in this work (mainly in Chapters 5 and 6) is solved by changing the environment rather solving it by algorithms. Only one large-scale problem MORL has been investigated but not been solved, in Chapter 6. The generality to similar tasks has been investigated and solved in Chapter5. In this case, the main contribution of this thesis is solving the discrete MORL problems by LCSs-based algorithms.

The overall goal of this thesis is to determine compact and effective solutions to the MORL problems by developing LCS-based algorithms. This was accomplished through the use of LCSs, especially the accuracy-based classifier systems, by integrating them with multi-objective algorithms. This updated the single-objective LCS algorithms to enable them to handle multiple objectives. The developed systems were evaluated using various complex problems used in the literature and the results were compared with the existing related systems.

The rest of this chapter presents the summary of work from each chapter, contributions, future work, and chapter summary that stem from this research work.

7.1 Summary of Work

The following research objectives have been fulfilled by this work to achieve the overall research goal.

7.1.1 MO-XCS: Adding Pareto Dominance to XCS to Solve MORL Problems

In this work, a new reinforcement learning algorithm based on XCS is developed, which is called MO-XCS. The algorithm is designed to learn a group of Pareto optimal solutions through a single learning process. For this purpose, four technical issues in XCS have been identified and addressed in this work.

- The reward signal has to be updated as a multi-dimensional reward.
- As cl.P and cl.OP are two sets of vectors, the error estimation in MO-XCS is the distance between those two sets, see section 3.2.1, page 71.
- The explore and exploit strategy is based on the largest number of non-dominated Q-vectors or the largest *hypervolume*.
- Rather than tracking the maximum Q-value in XCS during the testing process, MO-XCS constructs the Pareto Front and tracks the Pareto Q-value when exploiting the learned policies.

Experimental studies on three bi-objective maze problems further demonstrate the effectiveness of the algorithm. In the experiments, we have

specifically examined the learning effectiveness of four alternative distance measures and two separate approaches for action selection. The results of the experiments show that the action selection method has more influence on learning performance than the distance measure. Specifically, hypervolume-based action selection allows us to explore promising regions of the learning space more effectively than alternatives and therefore helps to achieve more effective learning.

Although MO-XCS can learn a group of Pareto optimal solutions through a single learning process, the LCS must store the whole history of predictions for the states it visited, and select one such backup prediction as the prediction of the classifier. However, this work exhibits several limitations. First, the backup strategy may consume a large amount of storage. Second, the tracing mechanism employed for constructing Pareto optimal policies will only choose one of the classifiers in the next state, thus removing the generalization ability of XCS.

7.1.2 MOXCS: Decomposition based Multi-Objective Evolutionary Algorithm in XCS for Multi-Objective Reinforcement Learning

In this work, a new XCS-based multi-objective reinforcement learning algorithm, MOXCS, is developed. With the help of a decomposition multi-objective algorithm (MOEA/D), MOXCS can address problems with complicated Pareto fronts. To achieve this goal, several technical issues in XCS have been identified and addressed in this work by adopting the following.

- Following the idea of MOEA/D, MOXCS is to decompose a MORL problem to N single objective RL sub-problems by initialization of weights and neighbor size.
- For the classifier's structure, to maintain the predictions of m objec-

tives, each classifier has m predictions, errors, and finesses.

- For updating classifier parameters, $\max(PA)$ is only formed by the classifiers that match the current state and the current weight $\overrightarrow{\lambda^i}$.
- Inspired by MOEA/D, in the Genetic Algorithm (GA) process, classifiers with similar weights are used to search for the optimal solutions.

A new ZCS-based multi-objective reinforcement learning algorithm, MOZCS, is developed as well. Similar to MOXCS, the standard ZCS is updated in four aspects as mentioned before. The only difference between MOXCS and MOZCS is when updating the classifier structure for maintaining predictions, only the predictions of classifiers need to be updated, as there are no errors and fitness in classifiers in ZCS.

Both MOXCS and MOZCS have been tested on three benchmark maze problems with two separate objectives. The performance was measured by evaluating the hypervolume and policy match rate. From the results, the evidence shows that MOXCS and MOZCS can solve MORL problems successfully. Analyzing the influence of reward settings, the results showed that when placing more weight on 'gold', compared with 'steps' to 'gold', the action selection policy will be similar to the policy only considering the 'gold' setting on the second objective. On the other hand, when the weights vary, then the learned optimal policy will change as well. It showing the 'gold' setting will influence the learned optimal policy.

MOXCS and MOZCS are also used to solve two MORL benchmarks, Deep Sea Treasure (DST) and Deep Sea Treasure Corridor (DSTC). For the DST problem, MOXCS and MOZCS handle the PO-MDP environment by updating the condition to consume the integer input of each state. The DSTC problem, the PO-MDP environment is handled by adding extra characters. The results of MOXCS and MOZCS show that they can successfully solve DSTC and DST by $\%OP100\%$ over 500, and 1000 learning problems, respectively. In this work, the learned PF by MOXCS and XCS

is compared, where the experiment results indicate that MOXCS can effectively learn more non-dominated Pareto policies. In addition, the finding shows MOXCS and MOZCS can find Pareto Optimal Policies with both convex and concave PF.

The Multi-Maze and Multi-Maze Connection are introduced to test the generalization ability of MOXCS. As the results of the previous testing of Bi-Objective mazes, DSTC and DST show that MOXCS has similar performance and MOXCS even has a better performance than MOZCS, only the generalization ability of MOXCS is experimentally evaluated on Multi-Maze. The agent trained by MOXCS can solve three same mazes in one environment. However, when updating the testing environment slightly, the problem cannot be solved in a deterministic analysis as the reward settings changes the action distribution.

7.1.3 Quantifying Generalization Ability of MOXCS in Multi-Objective Reinforcement Learning Problems

Two single-objective CoinRun environments (Env1 and Env5) were successfully solved by XCS with the technique of discretizing a continuous input, adding extra characters in PO-MDP Environment and sub-actions. This creates a new test rather than solve the old one, unless the agent learns to place the 'blocks' itself. Following the experiment, we test the generalization ability of XCS by training the agent in one environment and testing in another environment.

Then, these two environments are extended as MORL environments called the bi-objective RL CoinRun. The experimental results show MOXCS managed to resolve these two bi-objective CoinRun problems. The evidence shows MOXCS has the generalization ability for solving the MORL problem in an unseen environment or in a similar environment.

Finally, we discussed the Pareto Front learned by MOXCS, thus find the potential solution to solve the non-stationary obstacles in CoinRun en-

vironment.

7.1.4 Using MOXCS to Solve Large-Scale Non-Markov MORL Problems

In this chapter, XCS was first used to solve the mountain car problem by discretizing the environment and encoding integers to classifier conditions. Then the multi-objective mountain car is introduced in this work to test MOXCS. The position, action, and velocity related to different step are plotted to demonstrate not only what is the result, but help to understand why is the result. The value function is plotted to demonstrate what actions are taken at different states.

7.2 Contributions

This work will have contributions to the fields of LCSs and MORL as below:

1. MO-XCS is a novel multi-objective reinforcement learning algorithm based on XCS. Most of the research on reinforcement learning algorithms addresses only a single learning objective. Recently, several multi-objective reinforcement learning algorithms have been proposed. Different from many recently proposed learning algorithms that typically rely on tabular representations of the value function, this new algorithm will facilitate a more scalable representation in the form of a population of classifiers. In addition, it is designed to learn multiple Pareto optimal policies through a single learning process. According to our literature review, it is the first time to propose an XCS-based algorithm for this type of learning task. The experimental results show that the proposed MO-XCS can learn Pareto optimal policies on the introduced bi-objective mazes.

2. The decomposition-based Multi-Objective evolutionary reinforcement learning algorithms MOXCS and MOZCS were proposed for solving MORL problems. Different from most existing multi-objective LCS algorithms, they are not only able to solve the MORL problems, but also keep a good generalization ability. We have measured the performance of MOXCS and MOZCS by evaluating the hypervolume and policy match rate. From the results, we can see that MOXCS and MOZCS can solve MORL problems, i.e. bi-objective mazes and Deep Sea Treasure Corridor. In addition, the evidence in the experimental results show MOXCS can learn both concave and convex Pareto Front in complex scenarios.

3. In this work, new environments are generated for evaluating the generalization ability of MORL algorithms, i.e. constructing distinctive training and testing sets. More notably, we add the second objective into two selected environments from CoinRun as the benchmark for evaluating generalization in MORL, named CoinRun Action Bias and CoinRun Step VS Reward. Using CoinRun Action Bias and CoinRun Step VS Reward, we find that MOXCS can solve the MORL problems by training and testing in a slightly different environment.

4. In this work, a new technique is introduced into MOXCS to solve the non-Markov MORL problem by updating the condition of classifiers in MOXCS for addressing the integer inputs. The experimental results show that the updated MOXCS can learn Pareto optimal policies on the Deep Sea Treasure problem. Further, the updated MOXCS has shown the potential to solve the large-scale MORL problem multi-objective mountain car. In addition, we demonstrated and analyzed the advantages of MOXCS compared to other algorithms by comparing the converge speed and the Value Function on each state.

5. The advantage, disadvantage and when to use them are listed as below. For MO-XCS, the advantage is the prediction value for different objectives is updated and stored at once. The disadvantage is MO-XCS will consume a large amount of storage to store the exploring history, and lack of generalization ability. MO-XCS suits solving small-scale MDP problems, as it needs storage space and lacks generalization ability. On the other hand, for MOXCS, the advantage is that it doesn't need to consume a large amount of storage, and has a better generalization ability than MO-XCS. In addition, MOXCS converges faster than MO-XCS. The disadvantage is that prediction value needs to be updated with all the weights, and some weights may generate the same solution. MOXCS suits solving the large-scale MDP problem due to MOXCS's good generalization ability, and it converges fast. It can also solve the PO-MDP problem, especially when the correct action is the same in the aliasing states, but the expected value is different.

7.3 Future Work

Four hypotheses can be tested in future work:

The developed LCS-based MORL algorithms can solve several MORL benchmarks, but they cannot learn all the optimal solutions on a more complicated and consistent Pareto Front than the existing work in this thesis due to limited time for the agent to explore and understand the environment. In the future, more Multi-Objective algorithms will be considered to be integrated with LCS to help resolve the MORL problems. NSGA II will be considered to resolve the multi-objective problem. This is because in low dimension space or when the dataset has fewer features, NSGA II has a better performance than MOEA/D.

The second potential future work can explore the continuous solution for the multi-objective CoinRun problem. In this work, as the solution

learned by MOXCS is discrete, especially when the agent takes an action, then it will take several of the same sub-actions. Often, the obstacle is in the position that the agent could not avoid in a sequence of continuous sub-actions. To make the actions more flexible, XCSF will be used to tackle multistep reinforcement learning problems involving continuous inputs. The results in [13] shows that in domains involving continuous inputs and delayed rewards XCSF can evolve compact populations of accurate maximally general classifiers, which represent the optimal solution to the target problem.

The third potential future work could focus on improving the generalization ability on solving large-scale partially observable MORL problems. In this work, the integer-MOXCS solved the partially observable MORL problems by hard coding the environment, so the generalization ability is limited as the condition is specified no matter what action should take in the current state. In future work, ACS will be considered for solving the partially observable MORL problems. ACS evolves a model that specifies not only what to do in a given situation but also provides information of what will happen after a specific action is executed. In this case, the classifiers can be generalized to those states that should take the same action.

The fourth potential future work can focus on using MOXCS on a simulated robot. There are three reasons for this research. Firstly, though many researchers apply LCS on the simulated robot, they are mainly focusing on solving single-objective problems. Secondly, the solution of LCS is interpretable and can be made to be human interpretable IF:THEN statements. Last, the computing resource requirement is relatively low, so it is more feasible to implement on the simulated robot with reasonable cost on the resource.

7.4 Chapter Summary

This research work in this thesis has shown that LCS-based algorithms can solve MORL problems. The proposed MORL algorithm MO-XCS introduced a method to add Pareto Dominance to XCS for solving MORL problems. The experimental results show the capability to solve three introduced bi-objective maze problems. Inspiring by the decomposition-based multi-objective algorithm MOEA/D and LCS, the MORL algorithm MOXCS and MOZCS is developed. Both MOXCS and MOZCS can learn Pareto optimal solutions on introduced benchmarks such as bi-objective maze, Deep Sea Treasure, and MOXCS has similar performance to Pareto Front in terms of the number of learned Pareto optimal solutions. The generalization ability of MOXCS is tested on a Multi-Maze connection but the environment was found not ideal for testing the MORL algorithm. Then tested XCS and MOXCS on CoinRun environment and Multi-Objective CoinRun environment, the experimental results demonstrated the generalization ability of XCS and MOXCS. Finally, use XCS and MOXCS to solve the mountain car and multi-objective mountain car problem. It shows the advantage of XCS for solving large-scale non-Markov RL problems and the potential of MOXCS for solving large-scale multi-objective MORL problems.

Bibliography

[1] 0001, A. Z., WU, Y., AND PINEAU, J. Natural environment benchmarks for reinforcement learning. *CoRR abs/1811.06032* (2018).

[2] 0001, B. X., AND ZHANG, M. Evolutionary computation for feature manipulation: Key challenges and future directions. In *CEC* (2016), IEEE, pp. 3061–3067.

[3] 0002, G. C., ZHANG, M., PANG, S., AND DOUCH, C. Stochastic decision making in learning classifier systems through a natural policy gradient method. In *Neural Information Processing - 21st International Conference, ICONIP 2014, Kuching, Malaysia, November 3-6, 2014. Proceedings, Part III* (2014), C. K. Loo, K. S. Yap, K. W. Wong, A. T. B. Jin, and K. Huang, Eds., vol. 8836 of *Lecture Notes in Computer Science*, Springer, pp. 300–307.

[4] ABBASS, H. A. A Memetic Pareto Evolutionary Approach to Artificial Neural Networks. In *The Australian Joint Conference on Artificial Intelligence* (Adelaide, Australia, Dec. 2001), Springer. Lecture Notes in Artificial Intelligence Vol. 2256, pp. 1–12.

[5] ANDERSON, C. W. Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Mag.* (Apr. 1989), 31–37.

[6] BARRETT, L., AND NARAYANAN, S. Learning all optimal policies with multiple criteria. *Proceedings of the 25th International Conference on Machine Learning* (2008), 41–47.

[7] BELLEMARE, M. G., NADDAF, Y., VENESS, J., AND BOWLING, M. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res 47* (2013), 253–279.

[8] BERNADÓ, E., LLORA, X., AND GARRELL, J. M. A comparative study of two learning classifier systems on data mining. *Advances in Learning Classifier Systems* (2002), 115–132.

[9] BLAIR, C. E. Axioms and examples related to ordinal dynamic programming. *Math. Oper. Res 9*, 3 (1984), 345–347.

[10] BUTZ, M. V. An algorithmic description of ACS2. In *Advances in Learning Classifier Systems*, P. L. Lanzi, W. Stolzmann, and S. W. Wilson, Eds., vol. 2321 of *LNAI*. Springer-Verlag, Berlin, 2002, pp. 211–229.

[11] BUTZ, M. V. *Anticipatory learning classifier systems*. Kluwer Academic Publishers, 2002.

[12] BUTZ, M. V., GOLDBERG, D. E., AND LANZI, P. L. Gradient descent methods in learning classifier systems: Improving xcs performance in multistep problems. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION 9* (2005), 452–472.

[13] BUTZ, M. V., LANZI, P. L., AND WILSON, S. W. Function approximation with XCS: Hyperellipsoidal conditions, recursive least squares, and compaction. *IEEE Trans. Evolutionary Computation 12*, 3 (2008), 355–376.

[14] BUTZ, M. V., AND WILSON, S. W. An algorithmic description of XCS. Tech. Rep. 2000017, Illinois Genetic Algorithms Laboratory, 2000.

[15] BUTZ, M. V., AND WILSON, S. W. An algorithmic description of xcs. *Third International Workshop on Advances in Learning Classifier Systems* (2001), 253–272.

[16] CHOLLET, F. On the measure of intelligence. *CoRR abs/1911.01547* (2019).

[17] COBBE, K., KLIMOV, O., HESSE, C., KIM, T., AND SCHULMAN, J. Quantifying generalization in reinforcement learning. *CoRR abs/1812.02341* (2018).

[18] CRITES, R. H., AND BARTO, A. G. Improving elevator performance using reinforcement learning. *Advances in Neural Information Processing Systems 8* (1996), 1017–1023.

[19] CRUZ-RAMÍREZ, M., SÁNCHEZ-MONEDERO, J., FERNÁNDEZ-NAVARRO, F., FERNÁNDEZ, J. C., AND HERVÁS-MARTÍNEZ, C. Memetic pareto differential evolutionary artificial neural networks to determine growth multi-classes in predictive microbiology. *Evolutionary Intelligence 3*, 3-4 (2010), 187–199.

[20] DAHL, F. A. The lagging anchor algorithm: Reinforcement learning in two-player zero-sum games with imperfect information. *Machine Learning 49*, 1 (Oct. 2002), 5–37.

[21] DAM, H. H., ABBASS, H. A., LOKAN, C., AND YAO, X. Neural-based learning classifier systems. *IEEE Trans. Knowl. Data Eng. 20*, 1 (2008), 26–39.

[22] DARWIN, C. On the tendency of species to form varieties; and on the perpetuation of varieties and species by natural means of selection. *J. Proc. Linn. Soc. (Zoll.) 3* (1858), 45–62.

[23] DEISENROTH, M. P., NEUMANN, G., AND 0001, J. P. A survey on policy search for robotics. *Foundations and Trends in Robotics 2*, 1-2 (2013), 1–142.

[24] DIEDERIK, M., PETER, V., SIMON, W., AND RICHARD, D. Using the xcs classifier system for multi-objective reinforcement learning problems. *Artificial Life 13* (2007), 1064–5462.

[25] ECOFFET, A., HUIZINGA, J., LEHMAN, J., STANLEY, K. O., AND CLUNE, J. First return then explore. *CoRR abs/2004.12919* (2020).

[26] EL-SHORBAGY, M. A. TRUST REGION-PARTICLE SWARM FOR MULTI-OBJECTIVE ENGINEERING COMPONENT DESIGN PROBLEMS.

[27] FGER, H., STEIN, G., AND STILLA, U. Multi-population evolution strategies for structural image analysis. In *The First IEEE Conference on Evolutionary Computation* (Orlando, Florida, June 1994), IEEE, pp. 229–234.

[28] FIELDSEND, J. E., AND SINGH, S. Pareto evolutionary neural networks. *IEEE Transactions on Neural Networks 16*, 2 (Mar. 2005), 338–354.

[29] GABEL, C., AND RIEDMILLER, M. On a successful application of multi-agent reinforcement learning to operations research benchmarks. *Symposium on Approximate Dynamic Programming and Reinforcement Learning* (2007), 68–75.

[30] GÁBOR, Z., KALMÁR, Z., AND SZEPESVÁRI, C. Multi-criteria reinforcement learning. In *Proc. 15th International Conf. on Machine Learning* (1998), Morgan Kaufmann, San Francisco, CA, pp. 197–205.

[31] GADALETA, S., AND DANGELMAYR, G. Optimal chaos control through reinforcement learning. *Chaos 9*, 3 (1999), 775–788.

[32] GARVIN, D. A. What does product quality really mean? *Sloane Management Review 26*, 1 (1984), 25–43.

[33] GOMEZ, F. J., AND MIIKKULAINEN, R. Solving non-markovian control tasks with neuroevolution, Apr. 06 1999.

[34] GRUAU, F., WHITLEY, D., AND PYEATT, L. A comparison between cellular encoding and direct encoding for genetic neural networks.

In *Genetic Programming 1996: Proceedings of the First Annual Conference* (Stanford University, CA, USA, 28–31 July 1996), J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds., MIT Press, pp. 81–89.

[35] HARRIES, L., LEE, S., RZEPECKI, J., HOFMANN, K., AND DEVLIN, S. Mazeexplorer: A customisable 3d benchmark for assessing generalisation in reinforcement learning. In *2019 IEEE Conference on Games (CoG)* (2019), pp. 1–4.

[36] HAYKIN, S. *Neural Networks: a Comprehensive Foundation*. Macmillan, New York, NY, 1994.

[37] HOLLAND, J. H. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT Press, 1992.

[38] HOLLAND, J. H., AND REITMAN, J. S. Cognitive systems based on adaptive algorithms. *SIGART Newsletter 63* (1977), 49.

[39] HOLZINGER, A., PLASS, M., HOLZINGER, K., CRISAN, G. C., PINTEA, C.-M., AND PALADE, V. A glass-box interactive machine learning approach for solving np-hard problems with the human-in-the-loop. *CoRR abs/1708.01104* (2017).

[40] HU, H., XU, L., WEI, R., AND ZHU, B. Multi-objective tuning of nonlinear pid controllers for greenhouse environment using evolutionary algorithms. In *IEEE Congress on Evolutionary Computation* (2010), IEEE, pp. 1–6.

[41] HUMPHRYS, M. Action selection methods using reinforcement learning. MIT Press/Bradford Books.

[42] ILG, W., AND BERNS, K. A learning architecture based on reinforcement learning for adaptive control of the walking machine LAURON. *Robotics and Autonomous Systems 15*, 4 (1995), 321–334.

[43] IQBAL, M. *Improving the Scalability of XCS-Based Learning Classifier Systems*. PhD thesis, Victoria University, New Zealand, 2014.

[44] JIANG, S., AND YANG, S. A strength pareto evolutionary algorithm based on reference direction for multiobjective and many-objective optimization. *IEEE Transactions on Evolutionary Computation 21*, 3 (2017), 329–346.

[45] JOHANSSON, E. M., DOWLA, F. U., AND GOODMAN, D. M. Backpropagation learning for multi-layer feed-forward neural networks using the conjugate gradient method. *International Journal of Neural Systems 2* (1991), 291–302.

[46] JONG, K. D. Learning with genetic algorithms: An overview. *Machine Learning 3* (1988), 121.

[47] KARUNAKARAN, D., CHEN, G., AND ZHANG, M. J. Parallel multiobjective job shop scheduling using genetic programming. *Artificial Life and Computational Intelligence 9592* (2016), 234–245.

[48] KIM, M., HIROYASU, T., MIKI, M., AND WATANABE, S. SPEA2+: improving the performance of the strength pareto evolutionary algorithm 2. In *Parallel Problem Solving from Nature - PPSN VIII* (Birmingham, UK, 18-22 Sept. 2004), X. Yao, E. Burke, J. A. Lozano, J. Smith, J. J. Merelo-Guervós, J. A. Bullinaria, J. Rowe, P. T. A. Kabán, and H.-P. Schwefel, Eds., vol. 3242 of *LNCS*, Springer-Verlag, pp. 742–751.

[49] KOBER, J., AND PETERS, J. Reinforcement learning in robotics: A survey. *Adaptation, Learning, and Optimization 12* (2012), 579–610.

[50] KOZA, J. R., BANZHAF, W., CHELLAPILLA, K., K. DEB, M. D., FOGEL, D. B., GARZON, M. H., GOLDBERG, D. E., IBA, H., AND R. RIOLO, E. Generalization in the xcs classifier system. *Conf. Genetic Program.* (1998), 665–674.

[51] KRISHNA LAKSHMANAN, A., ELARA MOHAN, R., RAMALINGAM, B., VU LE, A., VEERAJAGADESHWAR, P., TIWARI, K., AND ILYAS, M. Complete coverage path planning using reinforcement learning for tetromino based cleaning and maintenance robot. *Automation in Construction 112* (2020), 103078.

[52] LAI, W., AND DENG, Z. Nsga2. (improved nsga2 algorithm based on dominant strength). 187–192.

[53] LANZI, P. L. Learning classifier systems from a reinforcement learning perspective. *Soft Computing 6* (2002), 162–170.

[54] LANZI, P. L., AND COLOMBETTI, M. An extension to the XCS classifier system for stochastic environments. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-99)* (1999), W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., Morgan Kaufmann, pp. 353–360.

[55] LEHMAN, J., AND STANLEY, K. *Novelty Search and the Problem with Objectives*. 11 2011, pp. 37–56.

[56] LEHMAN, J., AND STANLEY, K. O. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation 19*, 2 (2011), 189–223.

[57] LIU, C., XU, X., AND HU, D. Multiobjective reinforcement learning: A comprehensive overview. *IEEE Transactions on Systems, Man, and Cybernetics: Systems, 45*, 3 (2015), 385–398.

[58] LOVEJOY, W. S. A survey of algorithmic methods for partially observable Markov decision processes. *Annals of Operations Research 28*, 1 (1991), 47–65.

[59] MANNING, C. D., RAGHAVAN, P., AND SCHÜTZE", H. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[60] NATARAJAN, S., AND TADEPALLI, P. Dynamic preferences in multi-criteria reinforcement learning. *International Conference on Machine Learning* (2005), 601–608.

[61] NGUYEN, S., ZHANG, M., JOHNSTON, M., AND TAN, K. C. Automatic design of scheduling policies for dynamic multi-objective job shop scheduling via cooperative coevolution genetic programming. *IEEE Trans. Evolutionary Computation 18*, 2 (2014), 193–208.

[62] NITTA, T. A back-propagation algorithm for complex numbered neural networks. In *Proceedings of 1993 IEEE International Conference on Neural Networks (ICNN'93)* (Nagoya, Japan, Oct. 1993), vol. 2, IEEE/INNS, pp. 1649–1652. ETL.

[63] PIROTTA, M., PARISI, S., AND RESTELLI, M. Multi-objective reinforcement learning with continuous pareto frontier approximation. *CoRR abs/1406.3497* (2014).

[64] RICHARD S. SUTTON, DAVID MCALLESTER, S. S. Policy gradient methods for reinforcement learning with function approximation. *Advances in Neural Information Processing Systems* (2000), 1057–1063.

[65] ROIJERS, D. M., VAMPLEW., P., WHITESON, S. A., AND DAZELEY, R. A survey of multi-objective sequential decision-making. *Journal of Artificial Intelligence Research 48* (2013), 67–113.

[66] RUMMERY, G. A., AND NIRANJAN, M. On-line Q-learning using connectionist systems. Tech. rep., Oct. 04 1994.

[67] SINGH, S., JAAKKOLA, T., AND JORDAN, M. Learning without state estimation in partially observable environments. *Proceedings of the Eleventh Machine Learning Conference*.

[68] SIPPER, M., OLSON, R. S., AND MOORE, J. H. Evolutionary computation: the next major transition of artificial intelligence? *BioData Mining 10*, 1 (2017).

[69] SPRAGUE, N., AND BALLARD, D. H. Multiple-goal reinforcement learning with modular sarsa(0). In *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003* (2003), G. Gottlob and T. Walsh, Eds., Morgan Kaufmann, pp. 1445–1447.

[70] STANLEY, K. O., AND MIIKKULAINEN, R. Evolving neural networks through augmenting topologies. Tech. Rep. AI01-290, The University of Texas at Austin, Department of Computer Sciences, June 1 2001. Mon, 28 Apr 103 21:15:41 GMT.

[71] STANLEY, K. O., AND MIIKKULAINEN, R. Efficient evolution of neural network topologies, Nov. 21 2009.

[72] STONE, C., AND BULL, L. For real! XCS with continuous-valued inputs. *Evolutionary Computation 11*, 3 (2003), 298–336.

[73] STOOKE, A., AND ABBEEL, P. Accelerated methods for deep reinforcement learning. *CoRR abs/1803.02811* (2018).

[74] STREHL, A. L., 0001, L. L., AND LITTMAN, M. L. Reinforcement learning in finite mdps: Pac analysis. *J. Mach. Learn. Res 10* (2009), 2413–2444.

[75] STUDLEY, M., AND BULL, L. Using the xcs classifier system for multi-objective reinforcement learning problems. *Artificial Life* (2007), 69–86.

[76] SUTTON, R. S. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems 8* (1996), 1038–1044.

[77] SUTTON, R. S., AND BARTO, A. G. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, Massachusetts London, England, 1998.

[78] SUTTON, R. S., AND BARTO, A. G. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, 1998.

[79] SZITA, I. Reinforcement learning in games. *Adaptation, Learning, and Optimization 12* (2012), 539–577.

[80] TESAURO, G. Temporal difference learning and TD-gammon. *Commun. ACM 38*, 3 (1995), 58–68.

[81] URBANOWICZ, R. J., AND MOORE, J. H. Learning classifier systems: A complete introduction, review, and roadmap. *Journal of Artificial Evolution and Applications 2009* (2009). Article ID 736398.

[82] URBANOWICZ, R. J., AND MOORE, J. H. Learning classifier systems: A complete introduction, review, and roadmap. *Journal of Artificial Evolution and Applications 2009* (2009). Article ID 736398.

[83] VAMPLEW, P., YEARWOOD, J., DAZELEY, R., AND BERRY, A. On the limitations of scalarisation for multi-objective reinforcement learning of pareto fronts. *LNAI* (2008), 372–378.

[84] VAN ECK, N. J., AND VAN WEZEL, M. Reinforcement learning and its application to othello.

[85] VAN ECK CONRADIE, A. *A Neurocontrol Paradigm for Intelligent Process Control using Evolutionary Reinforcement Learning*. PhD thesis, Department of Chemical Engineering, University of Stellenbosch, 2004.

[86] VAN MOFFAERT, K., AND NOWE, A. Multi-objective reinforcement learning using sets of pareto dominating policies. *Journal of Machine Learning Research 15* (2014), 3483–3512.

[87] WAKAHARA, T., AND MIKAMI, S. Adaptive nutrient water supply control of plant factory system by reinforcement learning. *JACIII 15*, 7 (2011), 831–837.

[88] WANG, W., AND SEBAG, M. Hypervolume indicator and dominance reward based multi-objective monte-carlo tree search. *Machine Learning 92* (2013), 403–429.

[89] WATKINS, J. C. H., AND DAYAN, P. Q-learning. *Machine Learning 8* (1992), 279–292.

[90] WHITEHEAD, S. D., AND LIN, L. J. Reinforcement learning of non-markov decision processes. *Artif. Intell 73*, 1-2 (1995), 271–306.

[91] WHITESON, S. *Evolutionary computation for reinforcement learning*. Springer, 2012.

[92] WHITESON, S., AND STONE, P. Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research 7* (2006), 877–917.

[93] WILLIAMS, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning 8* (1992), 229.

[94] WILSON, S. W. ZCS: A zeroth level classifier system. *Evolutionary Computation 2*, 1 (1994), 1–18.

[95] YAO. Evolving artificial neural networks. *PIEEE: Proceedings of the IEEE 87* (1999).

[96] ZHANG, Q., AND LI, H. Moea/d: A multiobjective evolutionary algorithm based on decomposition. *IEEE Transactions on Evolutionary Computation 11*, 6 (2007), 712–731.

[97] ZHANG, Q., AND LI, H. MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. *IEEE Transactions on Evolutionary Computation 11*, 6 (Dec. 2007), 712–731.

[98] ZHOU, A., QU, B.-Y., LI, H., ZHAO, S.-Z., SUGANTHAN, P. N., AND ZHANG, Q. Multiobjective evolutionary algorithms: A survey

of the state of the art. *Swarm and Evolutionary Computation 1*, 1 (2011), 32–49.

[99] ZIELINSKI, K., AND LAUR, R. Differential evolution with adaptive parameter setting for multi-objective optimization. In *2007 IEEE Congress on Evolutionary Computation* (Singapore, 25-28 Sept. 2007), D. Srinivasan and L. Wang, Eds., IEEE Computational Intelligence Society, IEEE Press, pp. 3585–3592.

[100] ZITZLER, E., LAUMANNS, M., AND THIELE, L. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. In *EUROGEN 2001. Evolutionary Methods for Design, Optimization and Control with Applications to Industrial Problems* (Athens, Greece, 2002), K. Giannakoglou, D. Tsahalis, J. Periaux, P. Papailou, and T. Fogarty, Eds., pp. 95–100.

[101] ZITZLER, E., AND THIELE, L. Multiobjective evolutionary algorithms: A comparative case study and the strength Pareto approach. *IEEE Transactions on Evolutionary Computation 3*, 4 (1999), 257–271.

Chapter 8

Appendix

8.1 Novelty Search

Though the objective-based search method is widely used in Evolutionary Computation algorithms, there researches[55] [56] that hypothesises a fundamental problem in objective-based search: Most ambitious objectives do not illuminate a path to themselves as the search space is huge and often the searches terminates in the local optimal solution.

There are several indirect and direct evidence to support this hypothesis. First, the result of the objective-driven search does not often lead to great discoveries. For example, natural evolution innovates through an open-ended process rather than a final objective. Similarly, the large-scale cultural evolutionary processes, such as mathematics and art do not have a fixed goal. Lastly, the non-objective based search Novelty Search algorithm proposed by Lehman and Stanley[55] is direct evidence for this hypothesis.

Instead of searching for a final objective, the Novelty Search learning method employs a novelty metric to reward instances with functionality significantly rather than rewards based on how close is the final objective. For example, in a biped locomotion domain, the biped needs to learn to walk. An objective function may reward falling the farthest, but it may

lead to a local optimum as falling the farthest is not related to the objective of walking. On the other hand, novelty search rewards different types of falling, after different ways to fall are discovered, the only way to be rewarded is to find a behavior that does not fall right away. Eventually, the biped learns to walk.

The novelty search uses a novelty metric, which measures how different an individual is from other individuals so creating a constant pressure to take new behaviors. A simple measure of sparseness in the behaviors space at a point is the average distance to the k-nearest neighbors of that point, where k is a fixed parameter that is determined experimentally. The sparseness ρ at point x is given by 8.1 as following:

$$\rho(x) = \frac{1}{k} \sum_{k}^{i=0} dist(x, \mu_i) \tag{8.1}$$

where μ_i is the i th-nearest neighbor of x with respect to the distance metric distance in the behavior space, which is a domain-dependent measure of the behavioral difference between two individuals in the search space. If the sparseness ρ at point x is large then it is in a sparse area and the behavior is more novel or it is in a dense region if ρ is small and the behavior is less novel.

To test the performance of novelty search, three search methods (Novelty, Fitness, and Random) were implemented on two randomly generated maze problems created by the recursive division algorithm. The experimental result shows 1) Novelty search is the most effective in terms of the success rate among those three algorithms although its performance also degrades with increasing problem complexity. 2) The solution of novelty search converges faster to the maze complexity than fitness-based search or random search [55].

8.2 The Evolution of Connection Weights

In ANN, the aim of training Weights is usually to minimize an error function so as to get the expected output. Most training algorithms, such as Back Propagation (BP) [62] and conjugate gradient algorithms [45], are based on gradient descent. However, gradient algorithms are often trapped in a locally optimal solution. Therefore, using evolutionary computation methods to mutate and select the best neural network is a good solution to overcome the local optima issue. Though, premature convergence may happen because the progress of the population across iterations can be directed to one or more local optima, making individual solutions progressively more similar to each other. This process is also known as "diversity loss". Therefore, diversity maintenance is key for avoiding premature convergence.

In the simplest form of Neuroevolution, it only involves neural networks with fixed representations. In this case, all the networks have the same topology during the training process. For example, the hidden nodes and the edges between the nodes are fixed. The only differences between the networks are weights of these edges, which are optimized by evolution. However, the fixed network representations have a significant limitation. The primary reason is that a good representation should be specified by the user in advance. Clearly, picking a network with too simple representation will result in poor performance since it is impossible to describe high-quality solutions. However, choosing a representation with too complex can be also harmful. Even this complex representation can present good solutions, finding them may become infeasible and too expensive. Therefore, in most situations, the user is not able to specify the right representation of the network.

A previous study showed that fixed-topology neuroevolution can solve an challenging benchmark on RL problem: the double pole balancing task [5] [33] [34]. However, Stanley and Miikkulainen have shown that evolv-

ing topology can indeed increase performance in terms of speed on the double pole balancing problem [71]. The reason why evolving topology can improve the performance is because predefined topologies incur issues. If the topology is too simple, it cannot solve the problem, whereas evolving an overly complex topology will result in a high computational time.

8.3 MO-XCS vs MOXCS

In the thesis, %OP (Percentage of true Pareto Optimal Policies) and THV (Total Hypervolume) have been calculated evaluating the performance of MO-XCS and MOXCS, but %OP is more direct for evaluating the learned Optimal Policy as without achieving the best performance of THV the optimal policy still can be learned. In this case, %OP is used for comparing the differences in terms of performance of MO-XCS and MOXCS in solving the MORL problem by evaluating the learned optimal policies.

However, there are some differences in calculating %OP of MO-XCS and MOXCS. For MO-XCS, the %OP has been calculated in Formula 8.2

$$\%OP = \frac{S_{aopsn} \subseteq S_{eopsn}}{S_{eopsn}} X100\% \tag{8.2}$$

, which has been calculated based on the optimal solutions on the PF. More details, for each open state, the number of steps of all the Pareto Optimal Policy on the PF over each state is first manually calculated as a policy set, and denoted as Expected Optimal Policy Step Number Policy Set (S_{eopsn}). Then the record of the number of steps of the learned Pareto optimal policies over each state in the experiment result is collected for each open state as a policy set, and denoted as Actual Optimal Policy Step Number (S_{aopsn}). Note, the redundant record in S_{aopsn} will be deleted. Finally, calculate the percentage of how many optimal policies in S_{aopsn} can match that in S_{eopsn} over all the open states to get the %OP of MO-XCS. On the other hand, for MOXCS, the %OP has been calculated in Formula 8.3,

which is based on the learned optimal policies.

$$\%OP = \frac{S_{aopsn} \subseteq S_{eopsn}}{S_{aopsn}} X100\% \tag{8.3}$$

In this case, while %OP of MO-XCS is measuring how much optimal policies on PF have been learned, %OP of MOXCS is measuring if MOXCS learned the OP on the PF. Thus, in the result of MOXCS in this work, though even %OP of MOXCS is 100% for all the weights, it could not prove that all the solutions on PF have been learned for the following reasons, eg. the learned optimal policies would be affected by changing the discount factor, and the weights used for measuring the learned policy.

But, the result of %OP of MOXCS still valid because:

(1) As the discount factor changes the reward on the final state, the long-term payoff at some state may change, then the learned optimal policy may change, thus some OP on the PF wouldn't be learned. However, in the experiment, the discount factor is 0.99, thus the long-term payoff for the start state wouldn't be changed.

(2) As different weights may learn different solutions, if the testing weights were not chosen properly, some of the solutions on PF may not be learned. However, in the results of two-objective maze experiments in chapter4, as the testing weights are set from preferring first objective $0.78, 0.22$ to second objective $0.22, 0.78$, all the solutions on the PF should be learned if %OP of MOXCS is 100%.

For comparing MO-XCS and MOXCS on the efficiency of learning Optimal Policies on PF, the %OP achieve/close to 100% of MO-XCS and MOXCS has been compared in Table 8.1 as below:

Table 8.1: Number of learning problems for %OP close to 100%

Test Env	MO-XCS	MOXCS
Bi-obj Maze4	500	400
Bi-obj Maze5	1200	550
Bi-obj Maze6	700	550

From the table above, we can see that %OP of MOXCS achieve 100% (or close to 100%) are faster than that of MO-XCS in all the bi-objective mazes. More details, MOXCS spend 400, 550 and 550 to enable %OP close or achieve 100% in three test environments: bi-objective 4, 5 and 6, while MO-XCS spend 500, 1200 and 700 for that.