Searching by learning: Exploring artificial general intelligence on small board games by deep reinforcement learning

Wang, H.

Chapter 2

Classical Q-learning in GGP

2.1 Introduction

Traditional game playing programs are written to play a single specific game, such as Chess, or Go. The aim of *General* Game Playing [39] (GGP) is to create adaptive game playing programs; programs that can play more than one game well. To this end, GGP uses a so-called Game Description Language (GDL) [40]. GDL-authors write game-descriptions that specify the rules of a game. The challenge for GGP-authors is to write a GGP player that will play any game well. GGP players should ensure that a wide range of GDL-games can be played well. Comprehensive tool-suites exist to help researchers write GGP and GDL programs, and an active research community exists [38, 41, 42, 43].

The GGP model follows the state/action/result paradigm of reinforcement learning [6], a paradigm that has yielded many successful problem solving algorithms. For example, the successes of AlphaGo are based on two reinforcement learning algorithms, MCTS [7] and Deep Q-learning (DQN) [16, 44]. MCTS, in particular, has been successful in GGP [45]. However, few works analyze the potential of Q-learning for GGP, not to mention DQN. The aim of this chapter is to be a basis for further research of DQN for GGP and applying promising MCTS enhancements for neural network based reinforcement learning approaches.

Q-learning with deep neural networks requires extensive computational resources. Table-based Q-learning might offer a viable alternative for small games. Therefore, following Banerjee [46], in this chapter the convergence speed of table-based Q-learning is addressed. Three small two-player zero-sum games: Tic-Tac-Toe,

Hex and Connect Four, and table-based Q-learning are used. This chapter introduces two enhancements: dynamic ϵ, and, borrowing an idea from [18], a new version of Q-learning is created, inserting MCS into Q-learning, using online search for offline learning[1].

The main contributions of this chapter can be summarized as follows:

1. **Dynamic ϵ:** The classical Q-learning is evaluated, finding (1) that Q-learning works and converges in GGP, and (2) that Q-learning with a dynamic ϵ can enhance the performance of TD(λ)[2] baseline with a fixed ϵ [46].

2. **QM-learning:** To further improve performance the classical Q-learning is enhanced by adding a modest amount of Monte Carlo lookahead (QM-Player) [47]. This improves the convergence rate of Q-learning, and shows that online search can also improve the offline learning in GGP.

The chapter is organized as follows. Sect. 2.2 presents related work and recalls basic concepts of GGP and reinforcement learning. Sect. 2.3 presents the designs of the QPlayer with fixed and dynamic ϵ and QMPlayer for two-player zero-sum games for GGP to assess the potential of classical Q-learning in detail. Sect. 2.4 presents the experimental results. Sect. 2.5 summarizes the chapter and discusses directions for future work.

2.2 Related Work and Preliminaries

2.2.1 GGP

A General Game Player must be able to accept formal GDL descriptions of a game and play games effectively without human intervention [42], where the GDL has been defined to describe the game rules [48]. An interpreter program [43] generates legal moves (actions) for a specific board (state). Furthermore, a Game Manager (GM) is at the center of the software ecosystem. The GM interacts with game players through TCP/IP protocol to control the match. The GM manages game descriptions and matches records and temporary states of matches while the game is running. The system also contains a viewer interface for users who are interested in running matches and a monitor to analyze the match process.

[1] source code: https://github.com/wh1992v/ggp-rl

[2] one of temporal difference methods, see [6]

2.2.2 Reinforcement Learning

Since Watkins proposed Q-learning in 1989 [49], much progress has been made in reinforcement learning [50, 51, 52, 53]. However, few works report on the use of Q-learning in GGP. In [46], Banerjee and Stone propose a method to create a general game player to study knowledge transfer [54], combining Q-learning and GGP. Their aim is to improve the performance of Q-learning by transferring the knowledge learned in one game to a new, but related, game. They found knowledge transfer with Q-learning to be expensive. In [18], Gelly and Silver combine online and offline knowledge to improve learning performance.

Recently, DeepMind published work on mastering Chess and Shogi by self-play with a deep, generalized reinforcement learning algorithm [10]. With a series of landmark publications from AlphaGo to AlphaZero [10, 16, 17], these works showcase the promise of general reinforcement learning algorithms. However, such learning algorithms are very resource-intensive and typically require special GPU/TPU hardware. Furthermore, the neural network-based approach is quite inaccessible to theoretical analysis. Therefore, this chapter starts from studying the performance of table-based Q-learning.

In GGP, variants of MCTS [7] are used with great success [45]. Tom Vodopivec et al. studied MCTS with planning methods inspired by reinforcement learning [55]. Méhat et al. combined UCT (Upper Confidence bound applied to Trees) and nested MCS for single-player GGP [56]. Cazenave et al. further proposed a nested MCS for two-player games [57]. Monte Carlo techniques have proved a viable approach for searching intractable game spaces and other optimization problems [58]. Therefore, in this chapter MCS is combined as an enhancement to improve performance.

2.2.3 Q-learning

A basic distinction between reinforcement learning methods is that of "on-policy" and "off-policy" methods. On-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy *different* from that used to make decisions [6]. Q-learning is an off-policy method. The reinforcement learning model consists of an *agent*, a set of states S, and a set of actions A available in state S [6]. The agent can move to the next state s', $s' \in S$ from state s after following action a, $a \in A$, denoted as $s \xrightarrow{a} s'$. After finishing the action a, the agent gets an immediate reward $R(s, a)$, usually a numerical score. The cumulative return of current state s by taking

the action a, denoted as $Q(s, a)$, is a weighted sum, calculated by $R(s, a)$ and the maximum $Q(s', a')$ value of all next states:

$$Q(s, a) = R(s, a) + \gamma \; max_{a'} Q(s', a') \tag{2.1}$$

where $a' \in A'$ and A' is the set of actions available in state s'. γ is the discount factor of $max_{a'} Q(s', a')$ for next state s'. $Q(s, a)$ can be updated by online interactions with the environment using the following rule:

$$Q(s, a) \leftarrow (1 - \alpha) \; Q(s, a) + \alpha \; (\; R(s, a) + \gamma \; max_{a'} Q(s', a')) \tag{2.2}$$

where $\alpha \in [0, 1]$ is the learning rate. The Q-values are guaranteed to converge after iteratively updating.

2.3 Design

2.3.1 Classical Q-learning for Two-Player Games

GGP games in experiments for this chapter are two-player zero-sum games that alternate moves. Therefore, the same rule can be used, see Algorithm 1 line 5, to create $R(s, a)$, rather than to use a reward table. In experiments, $R(s, a) = 0$ is set for non-terminal states, and call the *getGoal*() function for terminal states. In order to improve the learning effectiveness, the $Q(s, a)$ table is updated only at the end of the match. During offline learning, QPlayer uses an ϵ-greedy strategy to balance exploration and exploitation towards convergence. While the ϵ-greedy strategy is enabled, QPlayer will perform a random action. Otherwise, QPlayer will perform the best action according to Q(S,A) table. If no record matches current state, QPlayer will perform a random action. The pseudo code for this algorithm is given in Algorithm 1.

Algorithm 1 Classical Q-learning Player with Static ϵ

```
1: function QPlayer(current state s, learning rate α, discount factor γ, Q table: Q(S, A))
2:     for each match do
3:         if s terminates then
4:             for each (s, a) from end to the start in current match record do
5:                 R(s,a) =s′ is terminal state? getGoal(s′, myrole) : 0
6:                 Update Q(s, a) ← (1−α) Q(s, a)+α ( R(s, a)+γ max_a′ Q(s′, a′))
7:         else
8:             if ϵ-greedy is enabled then
9:                 selected_action = Random()
10:            else
11:                selected_action = SelectFromQTable()
12:                if no s record in Q(S, A) then
13:                    selected_action = Random()
14:                                          ▷ To be changed for different versions
15:            performAction(s, selected_action)
16:    return Q(S, A)
```

2.3.2 Dynamic ϵ Enhancement

In contrast to the baseline of [46], which uses a fixed ϵ value, a dynamically decreasing ϵ-greedy Q-learning [50] is used. In the implementation, the function

$$\epsilon(m) = \begin{cases} a(\cos(\frac{m}{2l}\pi)) + b & m \leq l \\ 0 & m > l \end{cases} \tag{2.3}$$

is used for ϵ, where m is the current match count, and l is a number of matches set in advance to control the decaying speed of ϵ. During offline learning, if $m = l$, ϵ decreases to 0. a and b is set to limit the range of ϵ, where $\epsilon \in [b, a + b]$, $a, b \geq 0$ and $a + b \leq 1$. The player generates a random number num where $num \in [0, 1]$. If $num < \epsilon$, the player will explore a random action, else the player will exploit best action from the currently learnt $Q(s, a)$ table. Note that in this function, in order to assess the potential of Q-learning in detail, l is introduced for controlling the decay of ϵ. This parameter determines the value and changing speed of ϵ in current match count m. Instances in experiments are shown in Fig 2.1:

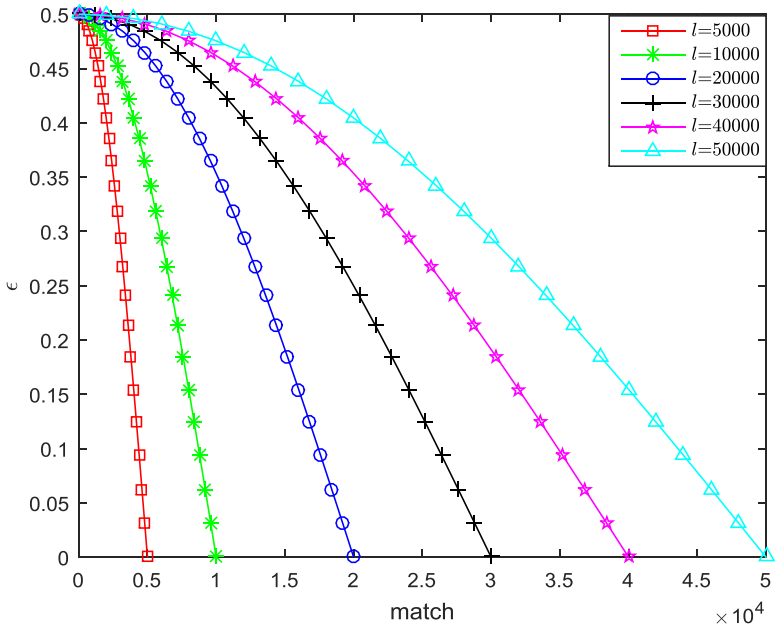


Figure 2.1: Decaying Curves of ϵ with Different l. Every curve decays from 0.5 (learning start, explore & exploit) to 0 ($m \geq l$, fully exploit).

2.3.3 QM-learning Enhancement

The main idea of MCS [47] is to make some lookahead probes from a non-terminal state to the end of the game by selecting random moves for the players to estimate the value of that state. To apply Monte Carlo in game playing, a time-limited version is used, since in competitive game playing time for each move is an important factor for the player to consider. The time limited MCS used in GGP is written as ***MonteCarloSearch(time_limit)***.

In Algorithm 1 (line 13), we see that a *random action* is chosen when QPlayer can not find an existing value in the $Q(s, a)$ table. In this case, QPlayer acts like a random player, which will lead to a low win rate and slow learning speed. In order to address this problem, a variant of Q-learning combined with MCS is introduced. MCS performs a time limited lookahead to find better moves. The more time it has, the better the action it finds will be. To achieve this, ***selected_action = MonteCarloSearch(time_limit)*** is used to replace the line 13, giving QM-learning. By adding MCS, a local version of the last two stages of MCTS is effectively added to Q-learning: the playout and backup stage [7].

2.4 Experiments and Results

2.4.1 Dynamic ϵ Enhancement

In experiments, the ϵ-greedy Q-learning players ($\alpha = 0.1$, $\gamma = 0.9$) with fixed ϵ=0.1, 0.2 and with dynamically decreasing $\epsilon \in [0, 0.5]$ is created to play 30000 matches first (l=30000) against a Random player, respectively. During these 30000 matches, the dynamic ϵ decreases from 0.5 to 0 based on the decay function, see equation 2.3. The fixed values for ϵ are 0.1 and 0.2, respectively. After 30000 matches, fixed ϵ is also set to 0 to continue the competition. For Tic-Tac-Toe, results in Fig. 2.2 show that dynamically decreasing ϵ performs better. We see that the final win rate of dynamically decreasing ϵ is 4% higher than fixed ϵ=0.1 and 7% higher than fixed ϵ=0.2. Therefore, in the rest of the experiments, dynamic ϵ is used for further improvements.

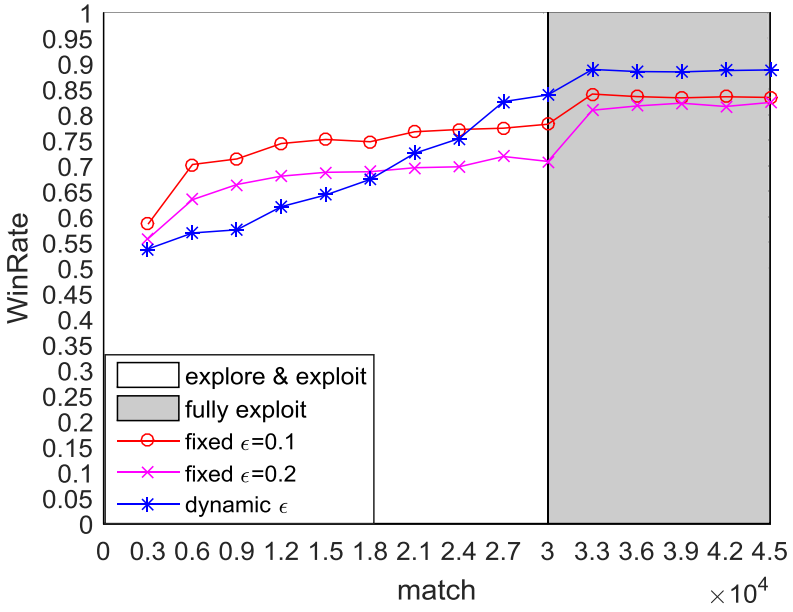


Figure 2.2: Win Rate of the Fixed and Dynamic ϵ Q-learning Player vs a Random Player Baseline. In the white part, the player uses ϵ-greedy to learn; in the grey part, all players set ϵ=0 (stable performance). The color code of the rest figures are the same

To enable comparison with previous work, TD(λ) is also implemented, the baseline learner of [46]($\alpha = 0.3$, $\gamma = 1.0$, $\lambda = 0.7$, $\epsilon = 0.01$), and dynamic ϵ

learner($\alpha = 0.1$, $\gamma = 0.9$, $\epsilon \in [0, 0.5]$, l=30000, Algorithm 1). For Tic-Tac-Toe, from Fig. 2.3, It is found that although the TD(λ) player converges more quickly initially (win rate stays at about 75.5% after 9000th match) the dynamic ϵ player performs better when the value of ϵ decreases dynamically with the learning process.

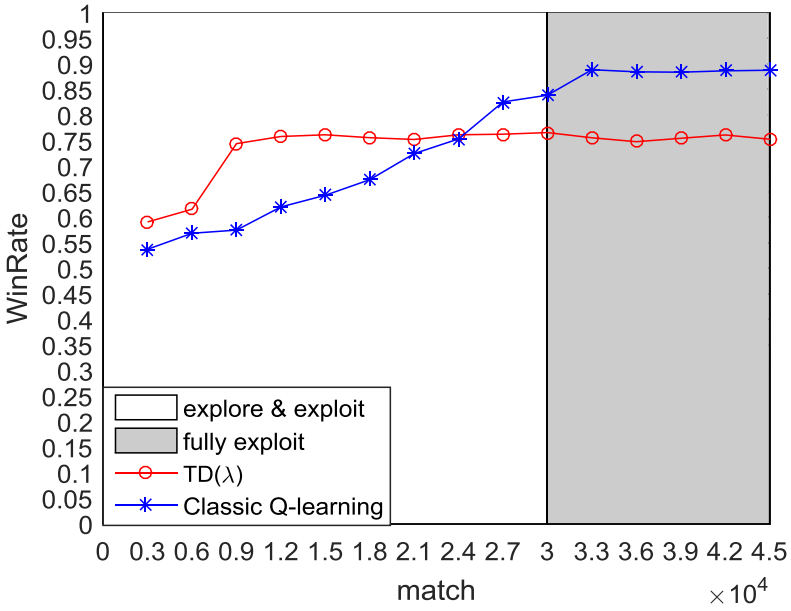
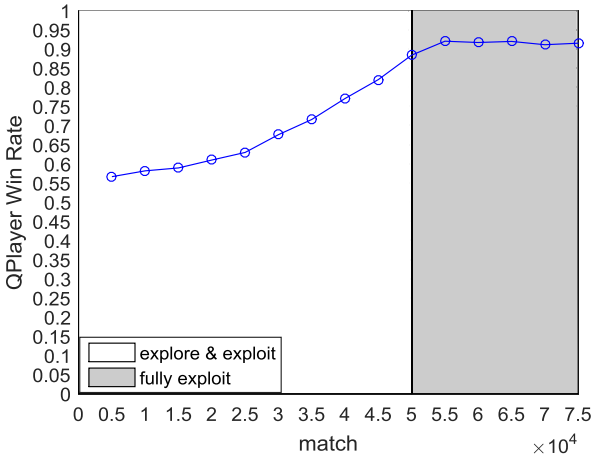
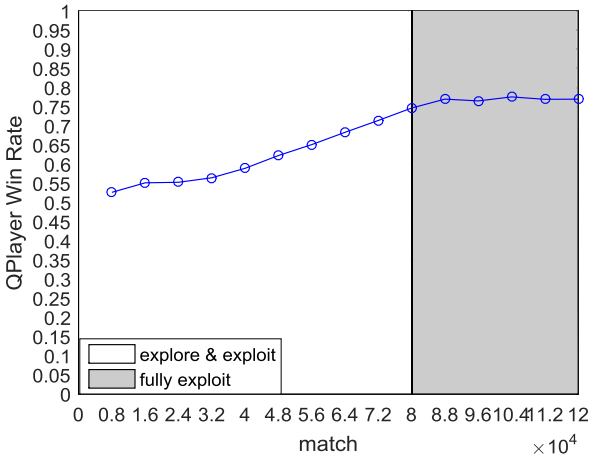


Figure 2.3: Win Rate of Classical Q-learning and [46] Baseline Player vs Random.

Experiments above suggest the following conclusions: that (1) classical Q-learning is applicable to a GGP system, and that (2) a dynamic ϵ can enhance the performance of fixed ϵ. However, beyond the basic applicability in a single game, showing that it can do so (1) *efficiently*, and (2) in more than one game is needed. Thus, further experiments with QPlayer to play Hex (l=50000) and Connect Four (l=80000) against the Random player are investigated. In order to limit excessive learning times, following [46], a very small 3×3 board Hex and ConnectFour on a 4×4 board are played. The results of these experiments are given in Fig. 2.4. We see that QPlayer can also play these other games effectively. Note that the reason why the player achieves different win rates could be that the game space of 3×3 Hex is much smaller than 4×4 ConnectFour.



(a) 3×3 Hex



(b) 4×4 Connect Four

Figure 2.4: Win Rate of QPlayer vs Random Player in Different Games. For Hex and Connect-Four the win rate of Q-learning also converges

However, so far, all studied games are small. QPlayer should be able to learn to play larger games. The complexity influences how many matches the QPlayer should learn. Now results will be shown to demonstrate how QPlayer performs while playing more complex games. QPlayer plays Tic-Tac-Toe (a line of 3 stones is a win, l=50000) in 3×3, 4×4 and 5×5 boards, respectively. and the results are shown in Fig. 2.5.

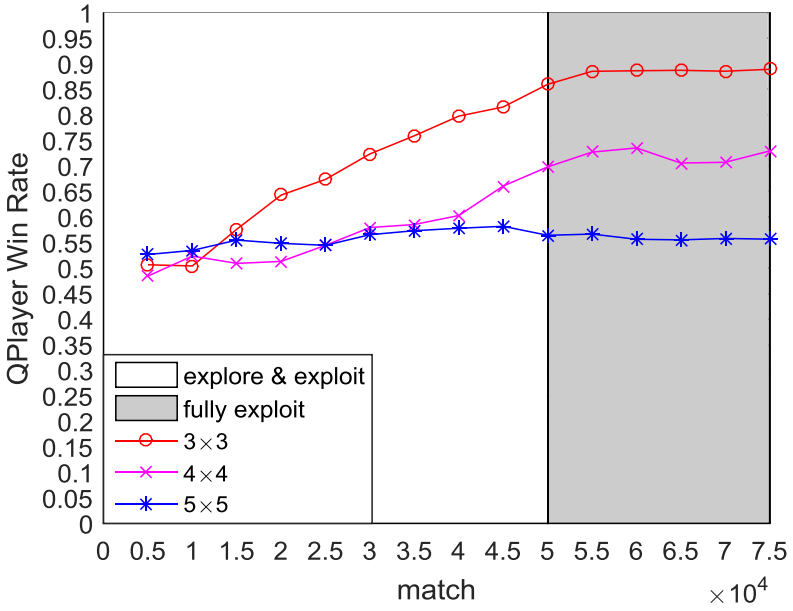
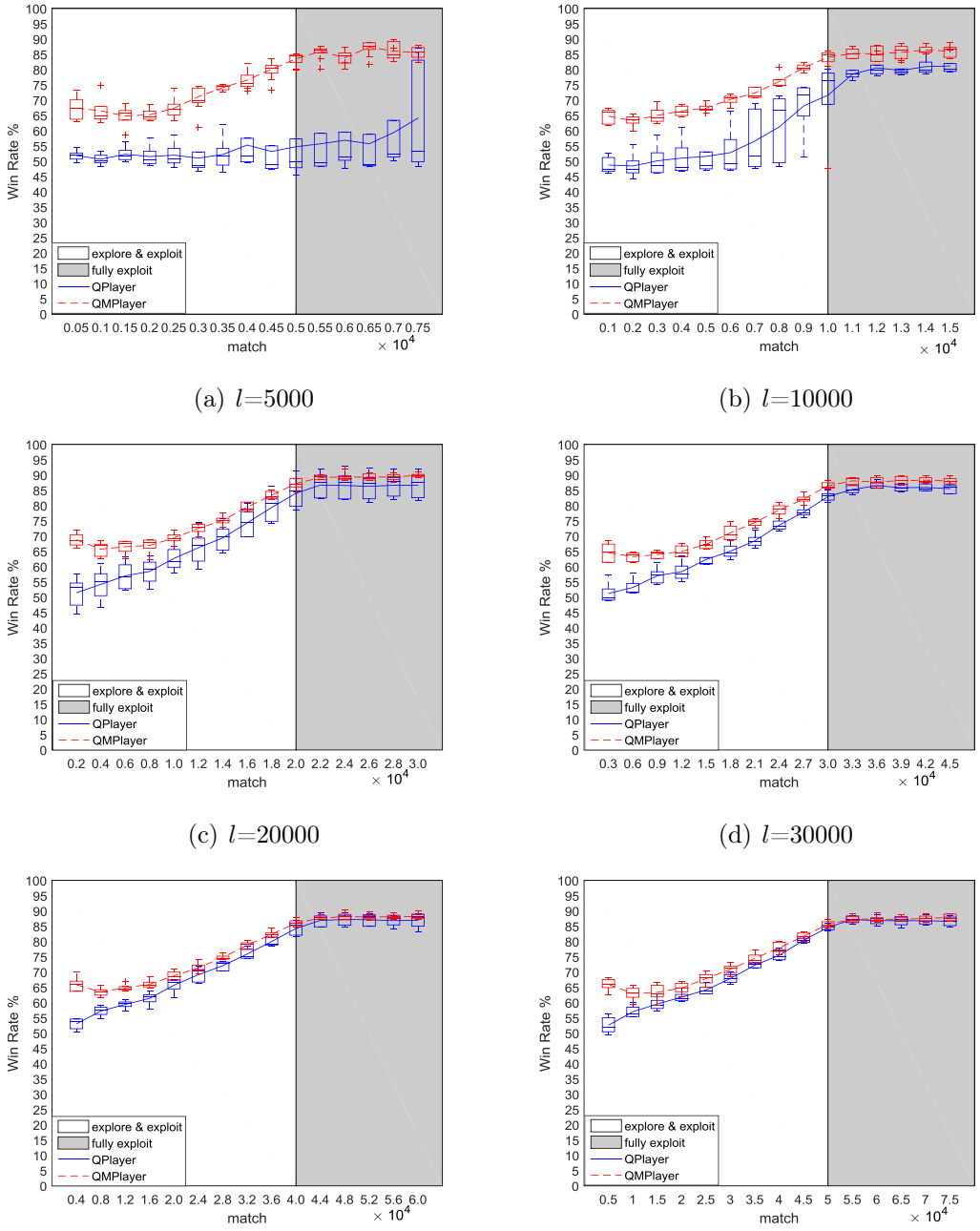


Figure 2.5: Win Rate of QPlayer vs Random in Tic-Tac-Toe on Different Board Size. For larger board sizes convergence slows down

The results show that with the increase of game board size, QPlayer performs worse. For larger boards can not achieve convergence. The reason for the lack of convergence is that QPlayer has not learned enough knowledge. The experiments also show that for table-based Q-learning in GGP, large game complexity leads to slow convergence, which confirms the well-known drawback of classical Q-learning.

2.4.2 QM-learning Enhancement

The second contribution of this chapter is QM-learning enhancement, the QPlayer and QMPlayer are both implemented based on Algorithm 1 and section 2.3.3. For both players, parameters are set to $\alpha = 0.1$, $\gamma = 0.9$, $\epsilon \in [0, 0.5]$ respectively and the l=5000, 10000, 20000, 30000, 40000, 50000, respectively. For QMPlayer, $time\_limit = 50ms$. Next both players play the game with the Random baseline player for $1.5 \times l$ matches for 5 rounds respectively. The comparison between QPlayer and QMPlayer is shown in Fig. 2.6.



(a) l=5000

(b) l=10000

(c) l=20000

(d) l=30000

(e) l=40000

(f) l=50000

Figure 2.6: Win Rate of QMPlayer (QPlayer) vs Random in Tic-Tac-Toe for 5 experiments. Small Monte Carlo lookaheads improve the convergence of Q-learning, especially in the early part of learning. QMPlayer always outperforms Qplayer

Fig. 2.6(a) shows that QPlayer has the most unstable performance (the largest variance in 5 experiments) and only wins around 55% matches after training 5000 matches. Fig. 2.6(b) illustrates that after training 10000 matches QPlayer wins about 80% matches. However, during the exploration period (the white part of the figure) the performance is still very unstable. Fig. 2.6(c) shows that QPlayer wins about 86% of the matches while learning 20000 matches still with high variance. Fig. 2.6(d), Fig. 2.6(e), Fig. 2.6(f), show us that after training 30000, 40000, 50000 matches, QPlayer gets a similar win rate, which is nearly 86.5% with smaller and smaller variance.

In Fig. 2.6(a), QMPlayer gets a high win rate (about 67%) at the very beginning. Then the win rate decreases to 66% and 65%, and then increases from 65% to around 84% at the 5000th macth. Finally, the win rate stays at around 85%. Also in the other sub figures, for QMPlayer, the curves all decrease first and then increase until reaching a stable state. This is because at the very beginning, QMPlayer chooses more actions from MCS. Then as the learning period moves forward, it chooses more actions from Q table.

Overall, as the l increases, the win rate of QPlayer becomes higher until leveling off around 86.5%. The variance becomes smaller and smaller, which proves that Q-learning can achieve convergence in GGP games and that a proper ϵ decaying speed makes sense for classical Q-learning. Note that in every sub figure, QMPlayer can always achieve a higher win rate than QPlayer, not only at the beginning but also at the end of the learning period. Overall, QMPlayer achieves a better performance than QPlayer with the higher convergence win rate (at least 87.5% after training 50000 matches). To compare the convergence speeds of QPlayer and QMPlayer, the convergence win rates of different l is summarized according to Fig. 2.6 in Fig. 2.7.

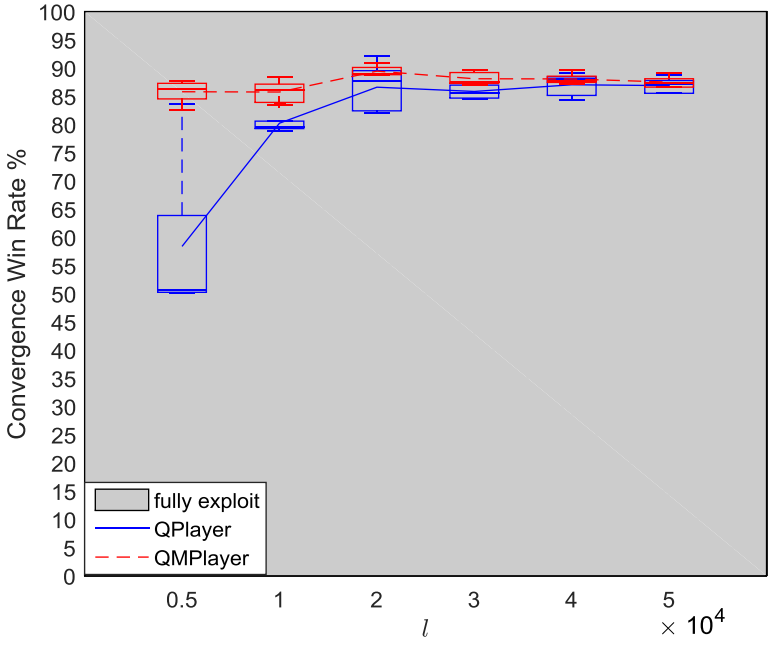


Figure 2.7: Convergence Win Rate of QMPlayer (QPlayer) vs Random in Tic-Tac-Toe

These results show that combining online MCS with classical Q-learning for GGP can improve the win rate both at the beginning and at the end of the offline learning period. The main reason is that QM-learning allows the $Q(s, a)$ table to be filled quickly with good actions from MCS, achieving a quick and direct learning rate. It is worth to note that, QMPlayer will spend slightly more time (at most is *search time limit*$\times$ *number of (state-action) pairs*) in training than QPlayer. It will be time consuming for MCS to compute a large game, and this is also the essential drawback of table-based Q-learning, so currently QM-learning is also only applicable for small games.

2.5 Summary

This chapter examines the applicability of Q-learning, a canonical reinforcement learning algorithm, to create general players for GGP programs. Firstly, it has shown how good canonical implementations of Q-learning perform on GGP games. The GGP system allows to easily use three real games for the experiments: Tic-Tac-Toe, Connect Four, and Hex. It is found that (1) Q-learning

is indeed general enough to achieve convergence in GGP games. However, convergence is slow. In accordance with Banerjee [46], who used a static value for ϵ, it is also found that (2) a value for ϵ that changes with the learning phases gives better performance (start with more exploration, become more greedy later on). The table-based implementation of Q-learning facilitates theoretical analysis, and comparison against some baselines [46]. However, it is only suitable for small games. A neural network implementation facilitates the study of larger games, and allows meaningful comparison to DQN variants [44].

Still using the table-based implementation, Q-learning is then enhanced with an MCS based lookahead. It is found that, especially at the start of the learning, this speeds up convergence considerably. The Q-learning is table-based, limiting it to small games. Even with the MCS enhancement, convergence of QM-learning does not yet allow its direct use in larger games. The QPlayer needs to learn a large number of matches to get good performance in playing larger games. The results with the improved Monte Carlo algorithm show a real improvement of the player's win rate, and learn the most probable strategies to get high rewards faster than learning completely from scratch. This enhancement shows how online search can be used to improve the performance of offline learning in GGP. On this basis, different offline learning algorithms can be assessed (or follow Gelly [18] to combine it with neural networks for larger games in GGP).

The use of Monte Carlo in QM-learning is different from the AlphaGo architecture, where MCTS is wrapped around Q-learning (DQN) [44]. In this work, Monte Carlo is inserted *within* the Q-learning loop. Future work should show if the QM-learning results transfer to AlphaGo-like uses of DQN inside MCTS, if QM-learning can achieve faster convergence, reducing the high computational demands of AlphaGo [16]. Additionally, nested MCS in Q-learning [57] could also be an option. Implementing Neural Network based players also allows the study of more complex GGP games [59].

Importantly, this work inspires a deep reinforcement learning framework to play GDL-games on GGP [60] and inspires the further studies of applying MCTS enhancements to improve neural network based reinforcement learning (Alphazero-like self-play) in Chapter 5 and Chapter 6.