Monte Carlo Q-learning for General Game Playing

Hui Wang, Michael Emmerich, Aske Plaat

Leiden Institute of Advanced Computer Science, Leiden University,
Leiden, the Netherlands
h.wang.13@liacs.leidenuniv.nl
http://www.cs.leiden.edu

Abstract. After the recent groundbreaking results of AlphaGo, we have seen a strong interest in reinforcement learning in game playing. General Game Playing (GGP) provides a good testbed for reinforcement learning. In GGP, a specification of games rules is given. GGP problems can be solved by reinforcement learning. Q-learning is one of the canonical reinforcement learning methods, and has been used by (Banerjee & Stone, IJCAI 2007) in GGP. In this paper we implement Q-learning in GGP for three small-board games (Tic-Tac-Toe, Connect Four, Hex), to allow comparison to Banerjee et al. As expected, Q-learning converges, although much slower than MCTS. Borrowing an idea from MCTS, we enhance Q-learning with Monte Carlo Search, to give QM-learning. This enhancement improves the performance of pure Q-learning. We believe that QM-learning can also be used to improve performance of reinforcement learning further for larger games, something which we will test in future work.

Keywords: Reinforcement Learning, Q-learning, General Game Playing, Monte Carlo Search

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 [1] (GGP) is to create adaptive game playing programs; programs that can play more than one game well. To this end, GGP applies a so-called Game Description Language (GDL) [2]. 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 run efficiently. Comprehensive tool-suites exist to help researchers write GGP and GDL programs, and an active research community exists [3,4,5].

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 recent successes of AlphaGo are based on two reinforcement learning algorithms, Monte Carlo Tree Search (MCTS) [7] and Deep Q-learning (DQN) [8,9]. MCTS, in particular, has been successful in GGP [10].

The AlphaGo successes have also shown that for Q-learning much compute power is needed, something already noted in Banerjee [11], who reported slow convergence for Q-learning. Following Banerjee, in this paper we address the convergence speed of Q-learning. We use three 2-player zero-sum games: Tic-Tac-Toe, Hex and Connect Four, and table-based Q-learning. Borrowing an idea from MCTS, we then create a new version of Q-learning,[1] inserting Monte Carlo Search (MCS) into the Q-learning loop.

Our contributions can be summarized as follows:

1. We evaluate the classical Q-learning algorithm, finding (1) that Q-learning works in GGP, and (2) that classical Q-learning converges slowly in comparison to MCTS.
2. To improve performance, and in contrast to [11], we enhance Q-learning by adding a modest amount of Monte Carlo lookahead (QMPlayer) [12]. This improves the rate of convergence of Q-learning.

The paper is organized as follows. Section 2 presents related work and recalls basic concepts of GGP and reinforcement learning. Section 3 provides the design of a player for single player games. We further discuss the player to play two-player zero-sum games, and implement such a player (QPlayer). Section 4 presents the player, inserting MCS, into Q-learning (QMPlayer). Section 5 concludes the paper and discusses directions for future work.

2 Related Work and Preliminaries

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 [4]. A Game Description Language (GDL) has been defined to describe the game rules [13]. An interpreter program [5] generates the 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 the 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.

2.2 Reinforcement Learning

Since Watkins proposed Q-learning in 1989 [14], much progress has been made in reinforcement learning [15,16]. However, only few works report on the use

[1] Despite the success of deep-learning techniques in the field of game playing, we consider it to be valuable to develop more light-weight, table-based, machine learning techniques for smaller board games. Such light-weight techniques would have the advantage of being more accessible to theoretical analysis and of being more efficient with respect to computational resources.

of Q-learning in GGP. In [11], Banerjee and Stone propose a method to create a general game player to study knowledge transfer, 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 our work, instead, we use Monte Carlo lookahead to get knowledge directly, in a single game.

Recently, DeepMind published work on mastering Chess and Shogi by self-play with a deep generalized reinforcement learning algorithm [18]. With a series of landmark publications from AlphaGo to AlphaZero [9,17,18], these works showcase the promise of general reinforcement learning algorithms. However, such learning algorithms are very resource intensive and typically require special GPU hardware. Further more, the neural network-based approach is quite inaccessible to theoretical analysis. Therefore, in this paper we study performance of table-based Q-learning.

In General Game Playing, variants of MCTS [7] are used with great success [10]. Méhat et al. combined UCT and nested MCS for single-player general game playing [19]. Cazenave et al. further proposed a nested MCS for two-player games [20]. Monte Carlo techniques have proved a viable approach for searching intractable game spaces and other optimization problems [21]. Therefore, in this paper we combine MCS to improve performance.

2.3 Q-learning

A basic distinction of 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 of every state s, $s \in S$ [6]. In Q-learning, 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 a reward, usually a numerical score, which is to be maximized (highest reward). In order to achieve this goal, the agent must find the optimal action for each state. The reward of current state s by taking the action a, denoted as $Q(s, a)$, is a weighted sum, calculated by the immediate reward $R(s, a)$ of moving to the next state and the maximum expected reward of all future states' rewards:

$$Q(s, a) = R(s, a) + \gamma(max_{a'} Q(s', a')) \qquad (1)$$

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

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(R(s, a) + \gamma(max_{a'} Q(s', a'))) \qquad (2)$$

where $\alpha \in [0, 1]$ is the learning rate. The Q-values are guaranteed to converge by some schemas, such as exploring every (s, a), which should be ensured by a suitable exploration and exploitation method (such as ϵ-greedy).

3 Classical Q-learning

3.1 Exploration/Exploitation: ϵ-greedy

As our baseline we use ϵ-greedy Q-learning [15] to balance exploration and exploitation. In order to find a better baseline player, we create ϵ-greedy Q-learning players($\alpha = 0.1$, $\gamma = 0.9$) with fixed ϵ=0.1, 0.2 and dynamically decreasing $\epsilon \in [0, 0.5]$ to play 30000 matches against Random player, respectively. During these 30000 matches, dynamic ϵ decreases from 0.5 to 0, fixed ϵ are 0.1, 0.2, respectively. After 30000 matches, fixed ϵ is also set to 0 to continue the competition. Results in Fig.1 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.

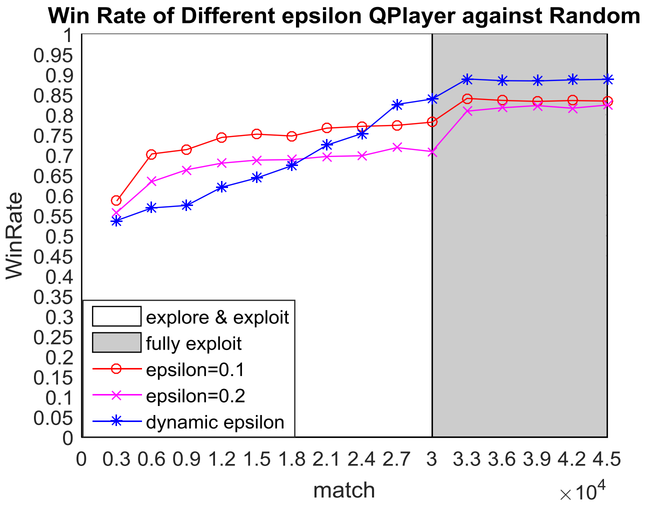


Fig. 1. Win Rate of Fixed and Dynamic ϵ Player vs Random in Tic-Tac-Toe. In the white part, the player uses ϵ-greedy to learn; in the grey part, all players set ϵ=0 (stable performance)

To enable comparison with previous work, we compare TD(λ), the baseline learner of [11]($\alpha = 0.3$, $\gamma = 1.0$, $\lambda = 0.7$, $\epsilon = 0.01$), and our baseline learner($\alpha = 0.1$, $\gamma = 0.9$, $\epsilon \in [0, 0.5]$, Algorithm 1). For Tic-Tac-Toe, from Fig.2, we find that although the TD(λ) player converges more quickly initially (win rate stays at about 75.5% after 9000th match), our baseline performs better when the value of ϵ-greedy decreases dynamically with the learning process.

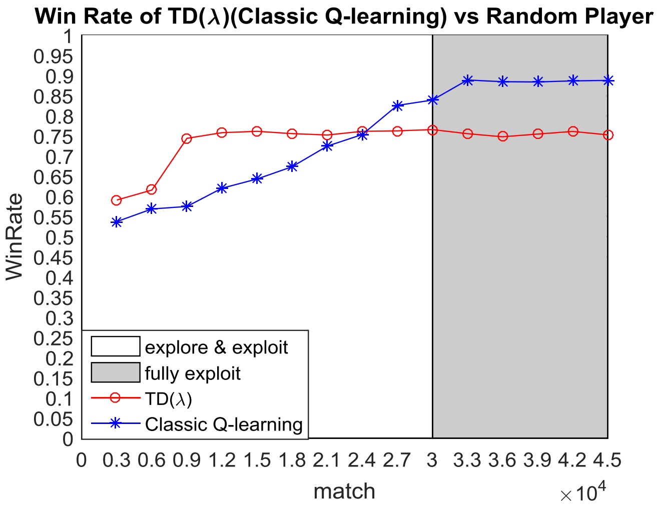


Fig. 2. Win Rate of ϵ-greedy Q-learning and [11] Baseline Player vs Random in Tic-Tac-Toe. In the white part, the player uses ϵ-greedy to learn; in the grey part, all players set ϵ=0 (stable performance)

In our dynamic implementation, we use the function

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

for ϵ, where m is the current match count, l is the total learning match, which we set in advance. a and b can be 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 new action randomly, else the player will choose a best action from the currently learnt $Q(s, a)$ table.

The question is what to do when the player cannot get a record from $Q(s, a)$? In classical Q-learning, it chooses an action randomly (Algorithm 2). While in the enhanced algorithm, we insert MCS (Algorithm 3) inside Q-learning, giving Algorithm 4. QMPlayer combines MCS and Random strategies in the first part of the search, but after enough state-action pairs are learned, it performs just like QPlayer.

3.2 Q-learning for Single Player Games

We start by introducing the simplest algorithm, playing single player games. Since games are played by only one player, we just need to build one $Q(s, a)$ table for the player to select the best action under the specific state, see Algorithm 1 [6]:

Algorithm 1 Basic Q-learning Player For Single Player Games

```
Input:
game state: $S$;
legal actions:$A$
learning rate: $\alpha$
discount factor: $\gamma$;
reward: $R(S, A)$;
updating table: $Q(S, A)$;
Output:
selected action according to updating table: $Q(S, A)$;
function BasicQlearningSingle($S, A$)
    for each learning match do
        for each game state during match do
            Update $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(R(s, a) + \gamma max_{a'} Q(s', a'))$;
        end for
    end for
    selected = false;
    expected_score = 0;
    for each $q(s, a)$ in $Q(S, A)$ do
        if(current game state equals s and expected_score < q(s,a));
        expected_score = q(s,a);
        selected_action = a;
        selected = true;
    end for
    if selected == false then
        selected_action = Random();
    end if
    return selected_action;
end function
```

3.3 Q-learning for Two-Player Games

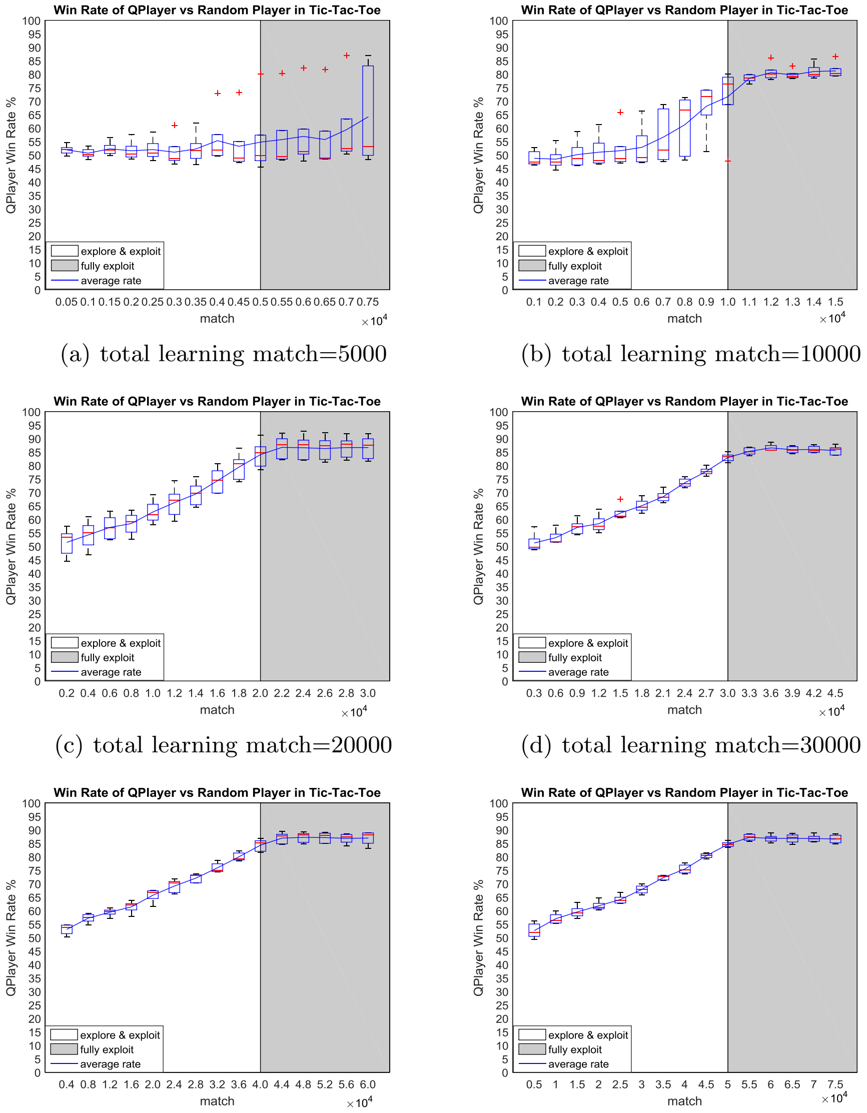
Next, we consider more complex games played by two players. In GGP, the *switch-turn* command allows every player to play the game roles by turn. The Q-learning player should build corresponding $Q(s, a)$ tables for each role.

Since our GGP games are two-player zero-sum games, we can use the same rule, see Algorithm 2 line 13, to create $R(s, a)$ rather than to use a reward table. In our experiments, we set $R(s, a) = 0$ for non-terminal states, and call the *getGoal*() function for terminal states. In order to improve the learning effectiveness, we update the corresponding $Q(s, a)$ table only at the end of the match.

Algorithm 2 ϵ-greedy Q-learning Player For Two-Player Zero-Sum Games

```
Input:
game state: $S$;
legal actions:$A$;
learning rate: $\alpha$;
discount factor: $\gamma$;
corresponding updating tables: $Q_{myrole}(S, A)$ for every role in the game;
Output:
selected action according to updating table: $Q_{myrole}(S, A)$;
function epsilonGreedyQlearning($S, A$)
    if $\epsilon$-greedy is enabled then
        for each learning match do
            record = getMatchRecord();
            for each state from termination to the beginning in record do
                myrole = getCurrentRole();
                R(s,a) = getReward(s,a);//$s'$ is terminal state? getGoal($s'$,myrole):0
                Update $Q_{myrole}(s, a) \leftarrow (1 - \alpha)Q_{myrole}(s, a) + \alpha(R(s, a) + \gamma max_{a'} Q_{myrole}(s', a'))$;
            end for
        end for
        selected = false;
        expected_ score = 0;
        for each $q_{myrole}(s, a)$ in $Q_{myrole}(S, A)$ do
            if(current game state equals s and expected_ score < $q_{myrole}(s, a)$);
            expected_ score = $q_{myrole}(s, a)$;
            selected_ action = a;
            selected = true;
        end for
        if selected == false then
            selectedaction = Random();
        end if
    else
        selected_ action = Random()
    end if
    return selected_ action;
end function
```

Experiment 1 In our first experiment we create QPlayer (see Algorithm 2) to play Tic-Tac-Toe. We set parameters $\alpha = 0.1$, $\gamma = 0.9$, $\epsilon \in [0, 0.5]$, respectively. As it learns to play Tic-Tac-Toe, we vary the *total learning match* in order to find how many matches it needs to learn to get convergence, and then make it play the game with Random player for 1.5 × *total learning match* matches. We report results averaged over 5 experiments. The results are shown in Fig.3.



(a) total learning match=5000

(b) total learning match=10000

(c) total learning match=20000

(d) total learning match=30000

(e) total learning match=40000

(f) total learning match=50000

Fig. 3. Win Rate of QPlayer vs Random Player in Tic-Tac-Toe averaged over 5 experiments. The winrate converges, and the variance is reduced as *total learning match* increases

Fig.3(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 (i.e., 2500 matches trained for each role). Fig.3(b) illustrates that after training 10000 matches QPlayer wins about 80% matches. However, during the exploration period (the light part of the figure) the performance is still very unstable. Fig.3(c) shows that QPlayer wins about 86% of the matches while learning 20000 matches still with high variance. Fig.3(d), Fig.3(e), Fig.3(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.

Overall, as the *total learning match* increases, the win rate of QPlayer becomes higher until leveling off around 86.5%. The variance becomes smaller and smaller. More intuitively, the QPlayer performance during the full exploitation period (the convergence results in the dark part of Fig. 3) against different *total learning match* is shown in Fig.4.

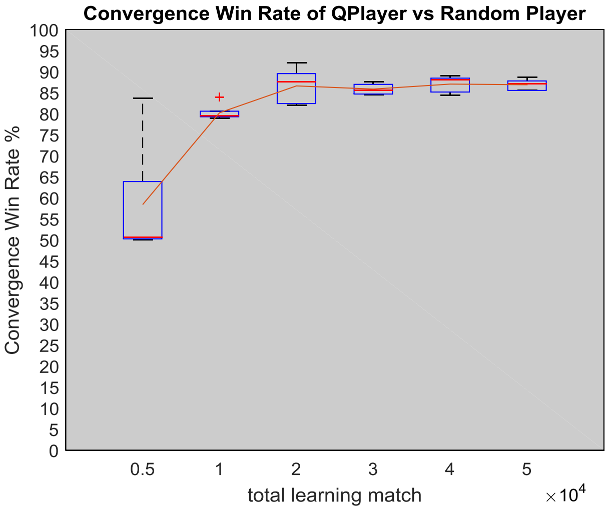


Fig. 4. Convergence Win Rate of QPlayer vs Random in Tic-Tac-Toe. Win rate converges as *total learning match increases*

Fig.4 shows that QPlayer achieves, after convergence, a win rate of around 86.5% with small variance. These experiments suggest indeed that Q-learning is applicable to a GGP system. However, beyond the basic applicability in a single game, we need to show that it can do so (1) *efficiently*, and (2) in more than one game. Thus, we further experiment with QPlayer to play Hex (learn 50000 matches) and Connect Four (learn 80000 matches) against the Random player. The results of these experiments are given in Fig.5 and Fig.6.

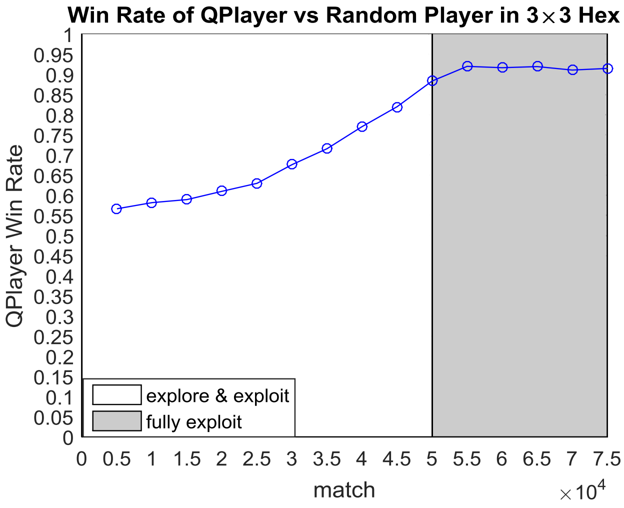


Fig. 5. Win Rate of QPlayer vs Random Player in 3×3 Hex, the win rate of Q-learning also converges

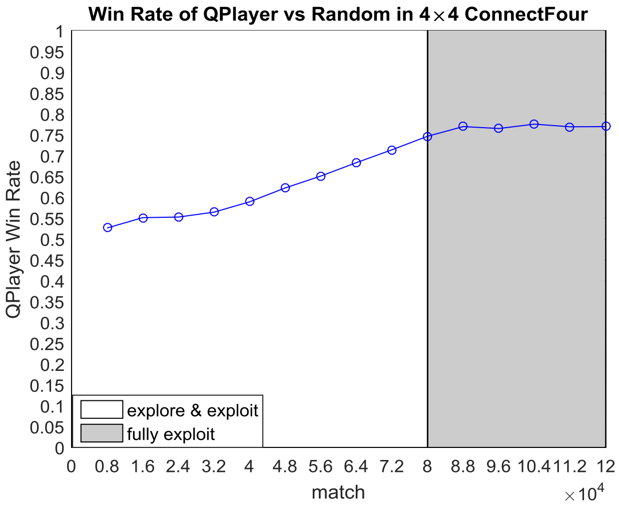


Fig. 6. Win Rate of QPlayer vs Random Player in 4×4 ConnectFour, the win rate of Q-learning also converges

In order to limit excessive learning times, following [11], we play Hex on a very small 3×3 board, and play ConnectFour on a 4×4 board. Fig.5 and Fig.6 show that QPlayer can also play these other games effectively.

However, there remains the problem that QPlayer should be able to learn to play larger games. The complexity influences how many matches the QPlayer should learn. We show results to demonstrate how QPlayer performs while playing more complex games. We make QPlayer learn Tic-Tac-Toe 50000 matches (75000 for whole competition) in 3×3, 4×4, 5×5 boards respectively and show the results in Fig.7:

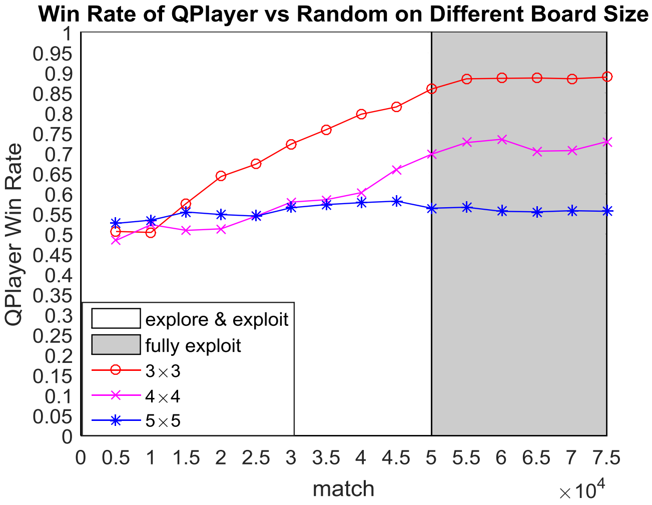


Fig. 7. Win Rate of QPlayer vs Random in Tic-Tac-Toe on Different Board Size. For larger board sizes convergence slows down

The results show that with the increase of game board size, QPlayer performs worse and for larger boards does not achieve convergence.

4 Monte Carlo Q-learning

4.1 Monte Carlo Search

The main idea of MCS [12] 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. The pseudo code of time limited MCS in GGP is shown in Algorithm 3.

Algorithm 3 Time Limited Monte Carlo Search Algorithm

```
Input:
game state: $S$;
legal actions:$A$;
time limit of each searching $t$;
Output:
The selected action $sa$, $sa \in A$;
function MonteCarloSearch(time_limit)
    sa = A.get(0);//default value of sa is set as the first action in $A$
    if A.size() > 1 then
        for int i = 0; i < A.size(); i = (i+1)%A.size() do
            if time_cost > time_limit then
                break;
            end if
            a = A.get(i);
            score = getGoalByPerformingRandomActionsFromNextState(s,a);
            score[i] += score;
            visit[i] += 1;
        end for
        highest_score = 0;
        best_action_index = 0;
        for int i = 0; i < A.size(); i++ do
            expected_score[i] = score[i]/visit[i];
            if expected_score[i] > highest_score then
                highest_score = expected_score[i];
                best_action_index = i;
            end if
        end for
        sa = A.get(best_action_index)
    end if
    return sa;
end function
```

4.2 Inserting MCS inside Q-learning

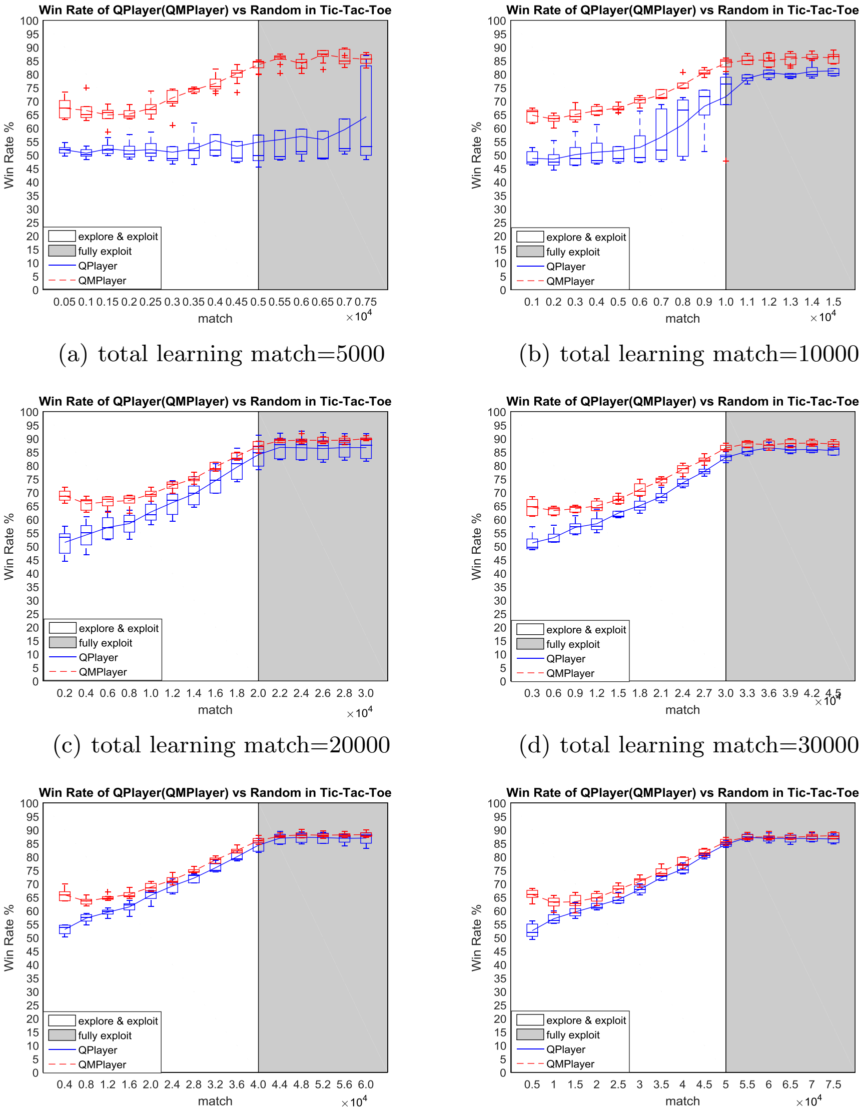
We will now add Monte Carlo Search to Q-learning (Algorithm 4). Starting from plain Q-learning, in Algorithm 2 (line 26), we see that a *random action* is chosen when QPlayer can not find an existing value in the $Q(s, a)$ table. In this case, QPlayer acts like a random game player, which will lead to a low win rate and slow learning speed. In order to address this problem, we introduce a variant of Q-learning combined with MCS. MCS performs a time limited lookahead for good moves. The more time it has, the better the action it finds will be. See Algorithm 4 (line 26).

By adding MCS, we effectively add a local version of the last two stages of MCTS to Q-learning: the playout and backup stage [7].

Algorithm 4 Monte Carlo Q-learning Player For Two-Player Zero-Sum Games

```
Input:
game state: $S$;
legal actions:$A$;
learning rate: $\alpha$;
discount factor: $\gamma$;
corresponding updating tables: $Q_{myrole}(S, A)$ for every role in the game;
Output:
selected action according to updating table: $Q_{myrole}(S, A)$;
function epsilonGreedyMonteCarloQlearning($S, A$)
    if $\epsilon$-greedy is enabled then
        for each learning match do
            record = getMatchRecord();
            for each state from termination to the beginning in record do
                myrole = getCurrentRole();
                R(s,a) = getReward(s,a);//$s'$ is terminal state? getGoal($s'$,myrole):0
                Update $Q_{myrole}(s, a) \leftarrow (1 - \alpha)Q_{myrole}(s, a) + \alpha(R(s, a) + \gamma max_{a'} Q_{myrole}(s', a'))$;
            end for
        end for
        selected = false;
        expected_score = 0;
        for each $q_{myrole}(s, a)$ in $Q_{myrole}(S, A)$ do
            if(current game state equals s and expected_score < $q_{myrole}(s, a)$);
            expected_score = $q_{myrole}(s, a)$;
            selected_action = a;
            selected = true;
        end for
        if selected == false then
            selected_action = MonteCarloSearch(time_limit); // Algorithm 3
        end if
    else
        selected_action = Random()
    end if
    return selected_action;
end function
```

Experiment 2 We will now describe our second experiment. In this experiment, with Monte Carlo-enhanced Q-learning, we use the QMPlayer. See Algorithm 4. We set parameters $\alpha = 0.1$, $\gamma = 0.9$, $\epsilon \in [0, 0.5]$, $time\_limit = 50ms$ respectively. For QMPlayer to learn to play Tic-Tac-Toe, we also set the *total learning match*=5000, 10000, 20000, 30000, 40000, 50000, respectively, and then make it play the game with Random player for 1.5 × *total learning match* matches for 5 rounds. The comparison with QPlayer is shown in Fig.8.



(a) total learning match=5000

(b) total learning match=10000

(c) total learning match=20000

(d) total learning match=30000

(e) total learning match=40000

(f) total learning match=50000

Fig. 8. Win Rate of QMPlayer and 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 the learning. QMPlayer always outperforms Qplayer

In Fig.8(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 end of ϵ learning(match=5000). Finally, the win rate stays at around 85%. Also in the other sub figures, for QMPlayer, the trend of all curves decreases 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 and achieves convergence.

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, we summarize the convergence win rates of different *total learning match* according to Fig. 3 and Fig. 8, in Fig.9.

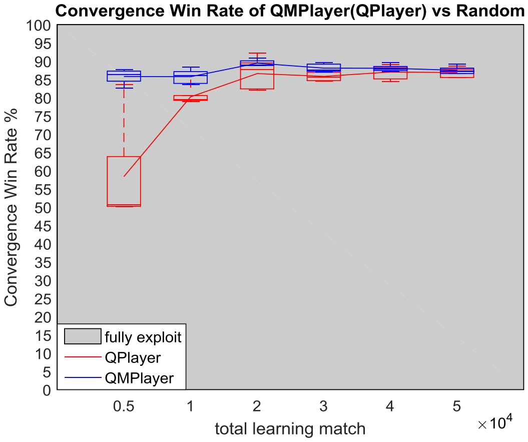


Fig. 9. Convergence Win Rate of QMPlayer and QPlayer vs Random in Tic-Tac-Toe

These results show that combining MCS with Q-learning for GGP can improve the win rate both at the beginning and at the end of the learning period. The main reason is that Monte Carlo-enhanced Q-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*× *number of (state action) pairs*) in training than QPlayer.

4.3 Comparison with MCTS

In order to evaluate the performance of both Q-learning players, we implemented a basic MCTS player [10]. Referring to the basic MCTS algorithm in [7], we present the pseudo code of basic time limited MCTS in GGP in Algorithm 5.

Algorithm 5 Basic Time Limited Monte Carlo Tree Search Player Algorithm For Two-Player Zero-Sum Games

```
Input:
game state: $S$;
legal actions:$A$;
empty game tree:$tree$;
visited nodes: $visited$;
current node:$node$;
Output:
selected action according to updated game tree;
function MCTS($S, A, time\_limit$)
    if legal_moves.size() == 1 then
        selected_ action = legal_moves.get(0);
    else
        while time_cost $\leq$ time_limit do
            while !node.isLeaf() do
                node = selectNodeByUCTMinMax();
                visited.add(node);
            end while
            expandGameTree(); //expand tree based on the number of all next states
            node=selectNodeByUCTMinMax();
            visited.add(node);
            bonus = playout(); //simulate from node to terminal state, get a score.
            backUpdate(); //for every visited node, count+=1; value+=bonus;
            visited.removeAll(visited);//erase visited list
        end while
        selected_child=getChildWithMaxAverageValue(tree.get(0).children)
        selected_action=getMoveFromParentToChild(selected_child);
    end if
    return selected_action;
end function
function selectNodeByUCTMinMax
    for each child in node.children do
        float uct = $\frac{child.totalvalue}{child.visitcount} + \sqrt{\frac{ln(node.visitcount+1)}{child.visitcount}}$;
        if is my turn according to node.game_state then
            if max_value < uct then
                max_value = uct;
                selected_node = child;
            end if
        else
            if min_value > uct then
                min_value = uct;
                selected_node = child;
            end if
        end if
    end for
    return selected_node;
end function
```

First, we make QPlayer learn 50000 matches. Then we set *time_limit*=10s for the MCTS player to build and update the search tree. For MCS, we also allow 10 seconds. With this long time limit, they reach perfect play on this small game. QPlayer and QM-player, in contrast, only get 50ms MCS time, and cannot reach perfect play in this short time period. QPlayer plays against the MCTS player in GGP by playing Tic-Tac-Toe for 100 matches. Then we pit other players against each other. The most relevant competition results of different players mentioned in this paper are shown in Table 1. The cells contain the win rate of the column player against the row player.

	MCTS	Random	QPlayer	QMPlayer	MCS
MCTS	-	0.5%	0	0	35%
Random	99.5%	-	86.5%	87.5%	100%
QPlayer	100%	13.5%	-	-	-
QMPlayer	100%	12.5%	-	-	-
MCS	65%	0	-	-	-

Table 1. Summary of Win Rate of Different Players Against to Each Other. The state space of Tic-Tac-Toe is too small for MCTS, it reaches perfect play. QMPlayer out-performs QPlayer

In Table 1, we find that (1) the state space of Tic-Tac-Toe is too small for MCTS, which reaches perfect play (win rate of 100%). Tic-Tac-Toe is suitable for showing the difference between QPlayer and QMPlayer. (2) MCTS wins 65% matches against QMPlayer since MCTS can win in the first hand matches and always get a draw in the second hand matches while playing with MCS. (3) The convergence win rate of QMPlayer(87.5%) against to Random is slightly higher than QPlayer(86.5%).

5 Conclusion

This paper examines the applicability of Q-learning, the canonical reinforcement learning method, to create general algorithms for GGP programs. Firstly, we show how good canonical implementations of Q-learning perform on GGP games. The GGP system allows us to easily use three real games for our experiments: Tic-Tac-Toe, Connect Four, and Hex. We find that (1) Q-learning is indeed general enough to achieve convergence in GGP games. With Banerjee [11], however, we also find that (2) convergence is slow. Compared against the MCTS algorithm that is often used in GGP [10], performance of Q-learning is lacking: MCTS achieves perfect play in Tic-Tac-Toe, whereas Q-learning does not.

We then enhance Q-learning with an MCS based lookahead. We find that, especially at the start of the learning, this speeds up convergence considerably. Our 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 (QM-learning) 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.

A final result is, that, where Banerjee et al. used a static value for ϵ, we find that 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 [11]. 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 [8].

Our use of Monte Carlo in QM-learning is different from the AlphaGo architecture, where MCTS is wrapped around Q-learning (DQN) [8]. In our approach, we inserted Monte Carlo *within* the Q-learning loop. Future work should show if our 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 [17]. Additionally, we plan to study nested MCS in Q-learning [20]. Implementing Neural Network based players also allows the study of more complex GGP games.

Acknowledgments. Hui Wang acknowledges financial support from the China Scholarship Council (CSC), CSC No.201706990015.

References

1. Genesereth M, Love N, Pell B: General game playing: Overview of the AAAI competition. AI magazine **26**(2), 62–72 (2005)
2. Love, Nathaniel and Hinrichs, Timothy and Haley, David and Schkufza, Eric and Genesereth, Michael: General game playing: Game description language specification. Stanford Tech Report LG-2006-1 (2008)
3. Kaiser D M: The Design and Implementation of a Successful General Game Playing Agent. In: David Wilson, Geoff Sutcliffe. International Florida Artificial Intelligence Research Society Conference 2007, pp. 110–115. AAAI Press, California (2007)
4. Genesereth M, Thielscher M: General game playing. Synthesis Lectures on Artificial Intelligence and Machine Learning **8**(2), 1–229 (2014)
5. Świechowski M, Mańdziuk J: Fast interpreter for logical reasoning in general game playing. Journal of Logic and Computation **26**(5), 1697–1727 (2014)
6. Sutton R S, Barto A G: Reinforcement learning: An introduction. 2nd edn. MIT press, Cambridge (1998)
7. Browne C B, Powley E, Whitehouse D, et al: A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in games **4**(1), 1–43 (2012)
8. Mnih V, Kavukcuoglu K, Silver D, et al: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)

9. Silver D, Huang A, Maddison C J, et al: Mastering the game of Go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
10. Mehat J, Cazenave T: Monte-carlo tree search for general game playing. Univ. Paris **8**, (2008)
11. Banerjee B, Stone P: General Game Learning Using Knowledge Transfer. In: Manuela M. Veloso. International Joint Conference on Artificial Intelligence 2007, pp. 672–677. (2007)
12. Robert C P: Monte carlo methods. John Wiley & Sons, New Jersey (2004)
13. Thielscher M: The general game playing description language is universal. In: Toby Walsh. International Joint Conference on Artificial Intelligence 2011, vol. 22(1), pp. 1107–1112. AAAI Press, California (2011)
14. Watkins C J C H: Learning from delayed rewards. King's College, Cambridge, (1989)
15. Even-Dar E, Mansour Y: Convergence of optimistic and incremental Q-learning. In: Thomas G.Dietterich, Suzanna Becker, Zoubin Ghahramani. Advances in neural information processing systems 2001, pp. 1499–1506. MIT press, Cambridge (2001)
16. Hu J, Wellman M P: Nash Q-learning for general-sum stochastic games. Journal of machine learning research **4**, 1039–1069 (2003)
17. Silver D, Schrittwieser J, Simonyan K, et al: Mastering the game of go without human knowledge. Nature **550**(7676), 354–359 (2017)
18. Silver D, Hubert T, Schrittwieser J, et al: Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. arXiv preprint arXiv:1712.01815, (2017).
19. Méhat J, Cazenave T: Combining UCT and nested Monte Carlo search for single-player general game playing. IEEE Transactions on Computational Intelligence and AI in Games **2**(4), 271–277 (2010)
20. Cazenave, T., Saffidine, A., Schofield, M. J., & Thielscher, M: Nested Monte Carlo Search for Two-Player Games. In: Dale Schuurmans, Michael P.Wellman. AAAI Conference on Artificial Intelligence 2016, vol. 16, pp. 687–693. AAAI Press, California (2016)
21. B Ruijl, J Vermaseren, A Plaat, J Herik: Combining Simulated Annealing and Monte Carlo Tree Search for Expression Simplification. In: Béatrice Duval, H. Jaap van den Herik, Stéphane Loiseau, Joaquim Filipe. Proceedings of the 6th International Conference on Agents and Artificial Intelligence 2014, vol. 1, pp. 724–731. SciTePress, Setúbal, Portugal (2014)