Temporal Difference Learning for Connect6

I-Chen Wu, Hsin-Ti Tsai, Hung-Hsuan Lin, Yi-Shan Lin,
Chieh-Min Chang, and Ping-Hung Lin

Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan
{icwu,fs751130,stanleylin,yslin,aup,bhlin}@java.csie.nctu.edu.tw

Abstract. In this paper, we apply temporal difference (TD) learning to Connect6, and successfully use TD(0) to improve the strength of a Connect6 program, NCTU6. The program won several computer Connect6 tournaments and also many man-machine Connect6 tournaments from 2006 to 2011. From our experiments, the best improved version of TD learning achieves about a 58% win rate against the original NCTU6 program. This paper discusses three implementation issues that improve the program. The program has a convincing performance in removing winning/losing moves via threat-space search in TD learning.

1 Introduction

Temporal difference (*TD*) *learning* [13][15], a kind of *reinforcement learning*, is a model-free method for adjusting the state values of the subsequent evaluations. This method has been applied to computer games such as Backgammon [18], Checkers [11], Chess [3], Shogi [4], Go [13] and Chinese Chess [20]. TD learning has been demonstrated to improve world class game-playing programs in the two following cases, TD-GAMMON [18] using TD(λ) and CHINOOK [11] using TDLeaf.

In this paper, we apply TD learning to Connect6, and successfully use TD(0) to improve the strength of a Connect6 program, NCTU6. NCTU6 won the gold medal in the Connect6 tournaments [7][17][23][25] several times from 2006 to 2011, and defeated many top-level human Connect6 players [8][16][29] in man-machine Connect6 championships from 2008 to 2011.

Our experiments showed that the best version of TD learning obtained about a 58% win rate against the original NCTU6 program. The results demonstrated that TD(0) learning can also be used to improve a high-performance world-class game-playing program.

In this paper, we discuss three implementation issues for TD learning, (a) selecting features, (b) removing winning/losing moves (found by threat-space search, which will be described in Section 4), and (c) using the moves played by strong human players for training. Our experiments demonstrate that the issue (b) is quite significant to improve the playing strength of NCTU6.

This paper is organized as follows. Section 2 reviews the game Connect6 and the program NCTU6. Section 3 reviews TD learning including TDLeaf and bootstrapping,

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 121–133, 2012.
© Springer-Verlag Berlin Heidelberg 2012

and describes our design of TD learning for Connect6. Section 4 discusses all the implementation issues of using TD learning, and Section 5 shows the experimental results. Section 6 provides concluding remarks.

2 Connect6 and NCTU6

Connect6 [22][27] is a kind of six-in-a-row game that was introduced by Wu *et al*. Two players, named Black and White in this paper, alternately place two black and white stones respectively on empty intersections of a Go board (a 19×19 board) in each turn. Black plays first and places one stone initially. The first player who obtains six consecutive stones of his own horizontally, vertically, or diagonally wins. The game has been played in one of tournaments held in Computer Olympiads [23][25] (as well as in some other tournaments [7][17]) since 2006.

From [22][27], we know that threats are the key to winning Connect6 (like Go-Moku [1][2] and Renju [10]). According to their definitions, a position is t-threat against the opponent, if and only if t is the smallest number of stones that the opponent needs to place to prevent from losing the game on the next move. A move is called a *1-threat* (also called *single-threat*) *move* if the position after the move is 1-threat, a *2-threat* (*double-threat*) *move* if 2-threat, and a *3-threat* (*triple-threat*) *move* if 3-threat. In Connect6, one player clearly wins by a *3-threat-or-more* move.

In Connect6, many *line patterns* (abbreviated as *patterns* in this paper), such as live-l and dead-l, can grow into threats. As defined in [22][27], live-l (dead-l) of a player can turn into 2-threat (1-threat) if the player places $(4 - l)$ additional stones. For example, live-3 (dead-3) can turn into a 2-threat (1-threat) after *one* additional stone is placed.

In [22][27], a type of winning strategy, called *Victory by Continuous Double-Threat-or-more* moves (*VCDT*) is described. The idea is to win by making continuously double-threat moves and ending by a triple-threat-or-more move or connecting up to six in all variations. It is similar to *Victory by Continuous Four* (*VCF*), a term used in the Renju community [10]. Similarly, the type of winning strategy with additional single-threat moves allowed is called *Victory by Continuous Single-Threat-or-more moves* (*VCST*). In the communities of Connect6 (Renju also), professionals are commonly keen to find these strategies, if there exists any.

Some of the authors developed a lambda-based [19] threat-space search (TSS) technique in [24], named *relevance-zone-oriented proof (RZOP) search*, to find these winning strategies, VCDTs or VCSTs, efficiently and accurately in most of the cases in which there exists any. The RZOP search was incorporated into a Connect6 program, named *NCTU6*, which won several computer Connect6 tournaments and man-machine Connect6 tournaments [7][8][16][17][23][25][29] from 2006 to 2011. When finding no winning strategies, NCTU6 [26] is back to use *alpha-beta search* to find the best move. In the alpha-beta search tree [6], the leaf values are estimated by an evaluation function, and the values of the internal nodes are calculated in the mini-max manner.

In order to make the search more accurate, NCTU6 used the RZOP search [24] to find the winning/losing moves in most nodes in the alpha-beta search. The underlying principle is to avoid choosing losing moves. For extra RZOP search, the averaged time for node evaluation/expansion in alpha-beta search is long. Hence, the number of nodes

in alpha-beta search is relatively small, about 50-500 per second in NCTU6, and the depth of the tree is small too, only about four in NCTU6. Since the alpha-beta search tree is small, a more sophisticated node evaluation/expansion was used to make the search more accurate. In this paper, such a search with heavy node computation is said to be *coarse-grained*. In contrast, strong Chinese-Chess programs (similar to Chess programs) are *fine-grained*, normally expanding about a million nodes per second and searching deeply.

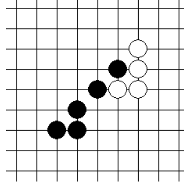

Fig. 1. An example of evaluating *V(s)* for Black

In NCTU6, the evaluation function of positions can be viewed as a function of features, such as threats, live-*l*s and dead-*l*s. Although the function was actually quite complicated, we modified it into a linear combination of features [5] for TD learning. In the example in Figure 1, Black has 1 single-threat, 2 live-2s and 7 live-1s, and White has 1 live-3, 2 dead-2s and 5 live-1s (note that dead-1 is not discussed in this paper for clarity). Given feature weights, NCTU6 evaluates the value of the position for Black in Figure 1 as

$$1 \times w_{T1} + 2 \times w_{L2} + 7 \times w_{L1} + 1 \times w_{-L3} + 2 \times w_{-D2} + 5 \times w_{-L1}$$

where w_f is the weight of feature *f*, which indicates *n*-threat by Tn, live-*n* (dead-*n*) by Ln (Dn), and the opponent's features by $-f$. Note that for 2-threat we use $1 \times w_{T2}$, instead of $2 \times w_{T1}$. Let $\varphi(s)$ denote a vector of feature numbers in a position (or state) s, and θ denote a vector of feature weights. Thus, the value of a position s is

$$V(s) = \varphi(s) \cdot \theta \qquad (1)$$

In the example in Figure 1, $\varphi(s) = [1,2,7,1,2,5]$, if the vector of features is $[T1, L2, L1, -L3, -D2, -L1]$.

In the original NCTU6, the weights θ were hand-tuned from some experiences of games against the top-level human players. However, as the number of features grew, it became hard to produce these weights accurately. The goal of this paper is to use TD learning to help adjust these weights automatically.

3 TD Learning for Connect6

This section first reviews TD learning and TDLeaf/bootstrapping in Subsections 3.1 and 3.2 respectively, and then describes our design for TD learning in Subsection 3.3.

3.1 TD Learning

As described in Section 1, TD learning is a kind of reinforcement learning. In TD(0) (see [13][15]), the value function V of a state is used to approximate the expected return, instead of waiting until the complete return has been observed. The error between states s_t and s_{t+1} is $\delta_t = r_{t+1} + V(s_{t+1}) - V(s_t)$, where r_{t+1} is the reward at time $t+1$. In Connect6 as well as some other computer games, the reward, say 1 for winning and –1 for losing, is obtained at the end game, and the reward is zero during the game playing. For clarity, for the end state (or the end game) s_T, let the value of $V(s_T)$ be r_T. Then, the error is simplified as $\delta_t = V(s_{t+1}) - V(s_t)$. The value of $V(s_t)$ in TD(0) is expected to be adjusted by the following value difference $\Delta V(s_t)$,

$$\Delta V(s_t) = \alpha\delta_t = \alpha\big(V(s_{t+1}) - V(s_t)\big) \tag{2}$$

where α is a step-size parameter to control the learning rate. For general TD(λ) (also see [13][15]), the value difference is

$$\Delta V(s_t) = \alpha\left((1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}V(s_{t+n}) + \lambda^{T-t-1}V(s_T) - V(s_t)\right). \tag{3}$$

Note that the TD(1) learning is similar to the learning with Monte-Carlo tree search.

In order to correct the value $V(s_t)$ by the difference $\Delta V(s_t)$, we can adjust the feature weights θ by a difference $\Delta\theta$ based on $\nabla_\theta V(s_t)$. For linear TD(0) learning, where $V(s_t)$ is linear like formula (1), the difference $\Delta\theta$ is

$$\Delta\theta = \Delta V(s_t)\varphi(s_t) = \alpha\delta_t\varphi(s_t) \tag{4}$$

In order to control the learning rate better, the above difference is modified with normalization, like the NLMS [14], as follows.

$$\Delta\theta = \Delta V(s_t)\frac{\varphi(s_t)}{\|\varphi(s_t)\|^2} = \alpha\delta_t\frac{\varphi(s_t)}{\|\varphi(s_t)\|^2} \tag{5}$$

3.2 TDLeaf and Bootstrapping

The researchers in [3] proposed the so-called TDLeaf to improve the weights of the features for their Chess program KNIGHTCAP. The method is to run the normal alpha-beta search and choose the leaves of the principal variation (PV) for TD learning, instead of the roots. However, as pointed out by [21], the method has the following three drawbacks. First, only one update is used for each search and other information is wasted. Second, the updates are only based on the positions of best play, which may not represent all the moves. Third, the target search is accurate only when both the player and opponent are strong.

In order to solve these problems, the researchers in [21] proposed a new method for bootstrapping from the minimax game-tree search. In their method, for all subtrees of the search, if their PVs are available, nodes on PVs are used for training. Note that their

method can be extended to alpha-beta search. Thus, they update many nodes with PVs in the search tree, rather than a single node, and the outcome of a deep search is used for training, instead of the outcome of a subsequent search.

3.3 Our TD Learning

Although the bootstrapping method seems promising, our design of TD learning for NCTU6 brings us back to the original TD(0) learning as explained below (see the last paragraph). Now, we want to describe the necessity of using a two-ply update, $\Delta V(s_t) = \alpha\big(V(s_{t+2}) - V(s_t)\big)$, instead of a one-ply update as in formula (2). In a one-ply update, updating the nodes between both players may cause overweighting the updates, since the player to move has always one move less (less advantage) than the other. Thus, the phenomenon of overweighting may cause a large fluctuation of the evaluated values. This problem is even more serious for Connect6 (recall two stones per move). Our algorithm for TD(0) is designed based on Silver's (cf. Algorithm 3 in [13]) as follows.

Algorithm TD(0) Learning Applied to NCTU6

```
Procedure TD_Learning(n)
 1:  i = 0
 2:  while i < n do
 3:     board.Initialize()
 4:     SelfPlay(board)
 5:     i++
 6:  end while
end procedure

Procedure SelfPlay(board)
 1:  t = 0
 2:  V0,φ0 = Eval(board)
 3:  while not board.Terminal() do
 4:     at = Greedy(board,ε)
 5:     board.Play(at)
 6:     t++
 7:     Vt, φt = Eval(board)
 8:     if t ≥ 2 then
 9:        δ= Vt - Vt-2
10:        Norm = ||φt-2[i]||^2
11:        for all i ∈φt-2 do
12:           θ[i] += αδφt-2[i] / Norm
13:        end for
14:     end if
15:  end while
end procedure
```

```
Procedure Greedy(board, ε)
 1:  if Bernoulli(ε) = 1 then return Random(board)
 2:  if board.BlackToPlay() then
 3:     a* = Pass; V* = 0
 4:     for all a ∈ board.Legal() do
 5:        board.Play(a)
 6:        V = Eval(board)
 7:        if V ≥ V* then
 8:           V* = V; a* = a
 9:        end if
10:        board.Undo()
11:     end for
12:  else // omitted for White to play
13:  end if
14:  return a*
end procedure

Procedure Eval(board)
 1:  φ = board.GetFeatures()
 2:  v = 0
 3:  for all i ∈φ do
 4:     v += φ[i]θ[i]
 5:  end for
 6:  V = 1/(1+e^-v)
 7:  return V, φ
end procedure
```

Our TD learning performs *n* training games by calling *TD_Learning(n)*. In each training game, we initialize the state (or board) by *board.Initialize*(), which selects initial boards from our database, mainly selected from Little Golem [9]. Then, we call the procedure *SelfPlay(board)*, to make the subsequent moves of a game.

This procedure *SelfPlay(board)* plays on its own by repeatedly calling *Greedy(board,ε)* to make moves. The procedure *Greedy(board,ε)* selects a move

according to the so-called ε-*Greedy policy* [13][15]. The ε-Greedy policy is to play at random with probability ε, as in Line 1 of this procedure, and to play the best move with probability (1-ε), as in Lines 2 to 14 (the case of White to play is omitted). The best move (or action) is chosen among all moves of which the values are determined by the evaluation procedure *Eval*.

The procedure *Eval* returns the state value and the vector of feature numbers of the board. As formula (1), the value function $V(s) = \varphi(s) \cdot \theta$ is evaluated in Lines 1 to 5 and adjusted into the range [0,1] by a function $1/(1 + e^{-V(s)})$ in Line 6. In case that Black (White) wins, the procedure *Eval* returns the state value one (zero).

Now, let us compare TD(0) with the bootstrapping method [21]. As described in Section 2, the search tree in NCTU6 is coarse-grained and shallow (the averaged depth of the tree is only about four). Consider the PV in a tree search, $\{s_0, s_1, s_2, s_3, s_4\}$, where s_0 is the root and s_4 is a leaf. Due to two-ply updates, we can only update both from s_4 to s_0 and from s_4 to s_2 in the bootstrapping method. For many of other subtrees, if their PVs are available, say $\{s_1, s_2, s_3, s_4\}$, we can only update from s_4 to s_2. According to our analysis on NCTU6, only about 400 updates can be used for training in an alpha-beta search tree with 10,000 nodes expanded. In contrast, 10,000 updates can be used for training in TD(0), when 10,000 nodes expanded. Thus, TD(0) apparently has more updates than bootstrapping.

4 Implementation Issues

This section discusses three issues when implementing the linear TD(0) learning for Connect6. These issues include (a) selecting features, (b) removing winning/losing moves found by threat-space search, and (c) using moves played by strong human players. These issues are discussed in the following three subsections respectively.

4.1 Feature Selection

As described above, this paper modifies the evaluation function into a linear combination of features, including the types of patterns, the distance of the patterns from the board center (or border), the direction of the pattern, and the game stages.

As described in Section 2, the types of patterns mainly include 1-threat, 2-threat, live-3 to live-1, dead-3 to dead-1, etc. In fact, there are more complex patterns, such as the pattern with live-1 and dead-2 at the same time. For example, the diagonal line containing two white stones in Figure 1 includes both dead-2 and live-1 at the same time. This pattern is actually stronger than dead-2 and live-1. However, for clarity of discussion, such patterns are disregarded in this paper.

Some other important features related to patterns are discussed as follows. The patterns on the border of the board tend to threaten the opponent less. The diagonal line patterns tend to be stronger, since diagonal line patterns normally cover a larger territory for attacking.

Next, we want to investigate features in different stages. Like some other games, such as Chinese Chess, the playing strategies in the three stages, opening, middle-game and end-game, are somewhat different. So, in our TD learning, the moves and features are also treated differently in the three stages. For instance, according to top-level

human players, live-2 in the opening is more important than in the end-game, since threats are not many in the opening.

4.2 Threat Space Search

It is a really important issue to remove winning/losing moves found by threat-space search (TSS). The key reason is that these losing moves are removed and not evaluated in alpha-beta search (like what NCTU6 does). Training these moves becomes noise in learning. In Connect6, it is a well-known strategy for players not to abuse playing 2-threats or 1-threats, if no winning strategies are found yet. Just like Go, beginners are taught not to abuse playing atari. Although the authors of NCTU6 [26] knew the idea upon designing, it was hard to tune the weights by hand. This is also one of the motivations of this paper.

Fig. 2. Avoiding winning/losing paths

If we do not remove these moves, the TD learning tends to make the weights of 1-threats or 2-threats excessively high. In the case that one player wins by a VCDT in the TD learning, the player plays many 2-threat moves in the last moves of the training games as shown in Figure 2. Consequently, 2-threat is wrongly regarded as a rather important feature and therefore adjusted to be overweighed.

In order to solve this problem, we propose to use TSS, the RZOP search [24] also used in NCTU6, to remove those winning/losing moves near the end as above. More specifically, TSS is performed to check winning of the position before running *Greedy* in Line 4 of *SelfPlay*, and once a winning move is found, *SelfPlay* terminates the game and restarts another training game. One minor drawback of this approach is the higher computation time for training, since the times spent on TSS are longer than node evaluation/expansion.

4.3 Learning from the Games Played by Strong Human Players

In Subsection 3.3, our TD learning program uses the procedure *Greedy* to make a move by the ε-Greedy policy. It is also an interesting issue to use the games played by strong human players, instead of using *Greedy*, as in [12]. Namely, in Line 4 of SelfPlay, *Greedy* is replaced by a routine which retrieves moves from the game record. This paper collected the games, about 30,197 games, where at least one of the players was ranked with points higher than 1800 from Little Golem [9]. The collection of records of these games is called *the expert collection* in this paper.

One advantage of using these games for learning is to spend less time on making a move. In order to remove winning/losing moves by TSS, we used a preprocessor to run TSS backwards from the leaf. Normally, TSS runs faster on winning/losing positions than the positions without winning/losing. Since running TSS backwards is performed on at most one position without winning/losing, the computation time is lower than that for running TSS forwards as in the algorithm in Subsection 3.3. A second advantage is to let the program learn to play like these players, if possible. For example, these players usually do not abuse playing 1-threats or 2-threats.

5 Experiments

In this section, we first describe our experimental environment in Subsection 5.1. Then, we analyze our experiments in different aspects, including stages, threat-space search, and training games, which are discussed in Subsections 5.2 to 5.4, respectively. Finally, we summarize and discuss the experimental results in Subsection 5.5.

5.1 Experimental Environment

In our experiments, we used TD(0) learning as shown in the algorithm in Subsection 3.3. We set $\alpha = 0.1$ and $\varepsilon = 0.1$. The number of features in total was about 500.

In our experiments, we measured the strength of the program learned from TD learning like [11] as follows. After finishing TD learning, we replaced the original feature weights of NCTU6 by the trained feature weights. Let *NCTU6-TD* denote the NCTU6 with the newly trained feature weights. In order to compare the strength of NCTU6-TD with that of the original NCTU6, we selected 176 popular openings from the expert collection. Namely, the openings that were played in at least 30 games in the collection. For each selected opening, let NCTU6-TD play twice against NCTU6, one for Black and the other for White, respectively. Thus, NCTU6-TD played 352 games against NCTU6 in total. NCTU6-TD obtained 2 points for a win, 1 for a draw, and nothing for a loss. The win rate was the total obtained points divided by 704, after finishing all the 352 games.

For each experiment of TD learning, all feature weights were initialized to 0 (zero knowledge), and the numbers of training games we ran were 0, 100, 300, 1000, 3000, 10000 and 30000. Since it took long times to do the experiments, we used the volunteer computing system in a job-level manner as described in [28].

5.2 Stages

As mentioned in Subsection 4.1, the playing strategies in the three stages, opening, middle-game, and end-game, are somewhat different. In our experiments, the first 10 moves in a game were considered to be played in the opening, the next 20 moves were in the middle-game, and the rest were in the end-game.

In this subsection, we tried four versions of stages for comparisons. The first version, called *1-stage*, was to have one stage only. The second version, called *3-stage*, was to have three stages as above. Thus, each feature had the different weights in

different stages in 3-stage, but the same in 1-stage. The third version, called *hybrid-3-stage*, was to use the feature weights of the original NCTU6 in opening, but use the feature weights trained by the second version in both middle-game and end-game. The fourth version, called *hybrid-2-stage*, was the same as hybrid-3-stage except that middle-game and end-game are combined into one stage.

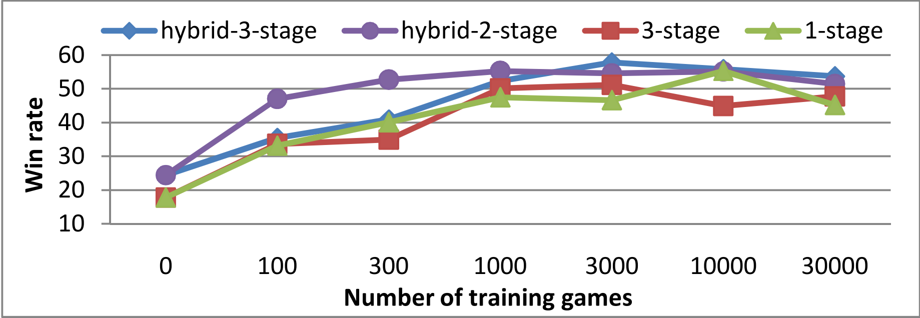

Fig. 3. The win rates for TD learning with different versions of stages

Figure 3 demonstrates the experimental results for the four different versions. In all of these experiments, we removed moves via TSS (namely the RZOP search) as described in Subsection 4.2. The results showed that both hybrid-2-stage and hybrid-3-stage were consistently better than 1-stage and 3-stage. The best was the one after 3000 training games in hybrid-3-stage. In our analysis, we observed two reasons for the phenomenon (of being the best) as follows. First, in opening, the features for those threats or patterns far away from the center were rarely used and therefore trained only few times. Thus, it became hard to learn these feature weights well in the first two versions. Second, for TD(0) learning, it was slow to learn the feature weights in opening since the learning propagation from end-game to opening was slow. Thus, the features in opening were relatively hard to learn.

For both hybrid-2-stage and hybrid-3-stage, hybrid-3-stage performed better for 3000 training games or more, but worse for less than 3000, for the following reason. Since hybrid-3-stage had more features, the learning rate was much slower. However, hybrid-3-stage performed better if there were sufficient training games.

5.3 Threat Space Search

As explained in Subsection 4.2, threat-space search (TSS) is a quite important issue. Figure 4 shows the results for TD learning with and without removing winning/losing moves found by TSS (namely the RZOP search [24]). In all of these experiments, we used hybrid-3-stage as above.

The results demonstrated significant and consistent improvements in all cases. The win rates with TSS (used to remove winning/losing moves) were about at least 7.1% higher than those without TSS. From the results, we may conclude that TSS plays a quite significant role in the TD learning.

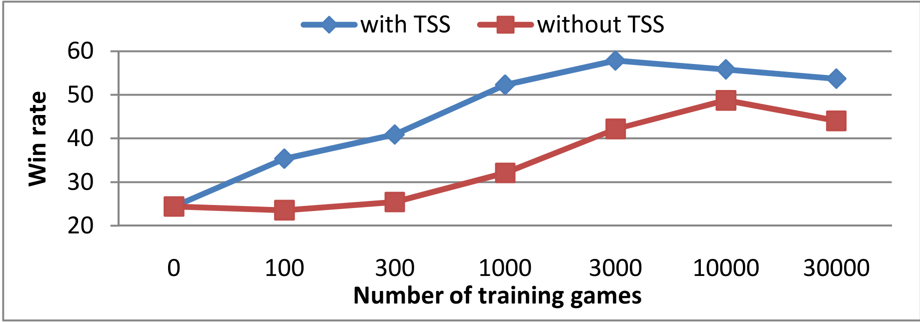

Fig. 4. The win rates for TD learning with and without using TSS

Table 1. The weights of features learned from TD learning with/without TSS

Feature Weights	With TSS	Without TSS
W_{T2}	0.52982	1.73220
W_{T1}	0.51070	0.83796
W_{L3}	0.49358	0.73046
W_{D3}	0.27506	0.25531
W_{L2}	0.20028	0.07715

Table 1 shows the weights of the features, 2-threats, 1-threat, live-3, dead-3, and live-2, learned from the TD learning with and without using TSS. The result clearly shows that the weight of 2-threat was high relatively to others (W_{T2} is nearly double of W_{T1}) in TD learning without TSS.

5.4 Training Games

As described in Subsection 4.3, selecting training games is a delicate issue for TD learning. The experiments in this subsection were done to investigate this issue by considering TD learning that (1) used the game records in the expert collection and that (2) used ε-*Greedy* to generate moves. In addition, for each case, we also considered TD learning with and without TSS (removing winning/losing moves).

Figure 5 showed the results for four kinds of TD learning. In all of these experiments we also used hybrid-3-stage as above. Still, it also showed that the TD learning with TSS was consistently and clearly better than the learning without TSS.

Below we consider using TSS. The results showed that the TD learning with ε-*Greedy* was slightly better than that with the expert collection for 1000 to 10,000 training games. In the case of using 30,000 training games, the TD learning with the expert collection was slightly better. Since the collection included about 30,000 games only, it was unsure about whether the win rate would be higher for more games.

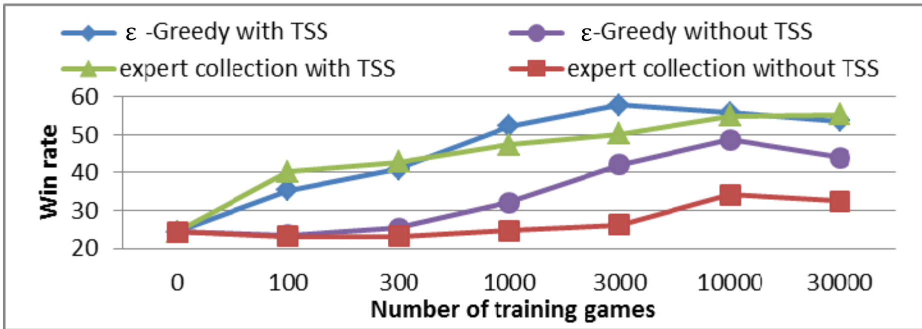

Fig. 5. The win rates for TD learning using ε-Greedy and the expert collection

5.5 Discussion

From Subsections 5.2 to 5.4, we may conclude that TD learning with TSS (used to remove winning/losing moves) is the most important factor to improve the strength. The win rates with TSS were at least 7.1% higher than those without it. Using the version of hybrid-3-stage also helps improve TD learning. The merit of using the expert collection is unclear yet.

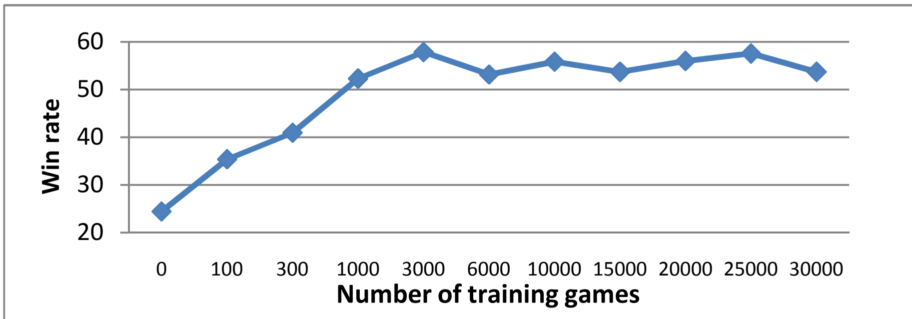

Fig. 6. The win rates for TD learning with more different training games

In the rest of this subsection, we discuss the convergence and computation times for training. In order to see whether TD learning in our experiments converges, we also ran 6000, 15000, 20000 and 25000 training games for the experiment with hybrid-3-stage,ε-Greedy, and TSS. Figure 6 shows that these values converged around 53-58% after running more than 3000 training games.

As for the training times, Table 2 showed the total times spent on training 10,000 games by using TSS or not, and by using the expert collection or not. Apparently, the versions with TSS ran much more slowly than the ones without TSS, due to the extra TSS overhead. Here, we consider the two versions with TSS. The one with the expert collection ran much faster than that with *ε-Greedy*, since we removed winning/losing moves backwards from the leaves in the former, as explained in Subsection 4.2.

Table 2. The comparison of the time spent with/without TSS

TD Learning	Times for 10,000 training games
ε-Greedy with TSS	677 min. (or 11 hr. 17 min.)
ε-Greedy without TSS	31 min.
Expert collection with TSS	32 min.
Expert collection without TSS	2 min.

6 Conclusion

In this paper, we demonstrate a solid application of TD(0) learning for Connect6. We successfully use TD(0) learning to improve the strength of NCTU6, a Connect6 program. Our experiments showed that the best version, improved via our TD learning method, obtained about a 58% win rate against the original NCTU6 program.

This paper also discusses several issues of implementing TD learning. From them we may conclude that TD learning plays a quite important role to remove winning/losing moves found by TSS (namely the RZOP search used in NCTU6). Our experiments demonstrated significant and consistent improvements in all cases. Using the version of hybrid-3-stage also helps improve TD learning. The merit of using the professional collection is unclear.

Although the bootstrapping method was not tried, this paper demonstrated that TD(0) learning worked sufficiently well for NCTU6. From this paper, it is conjectured that TD(0) should also work for other programs with coarse-grained and shallow search trees, though the comparison between TD(0) and bootstrapping is still to be performed in the future.

Acknowledgement. The authors would like to thank the anonymous referees for their valuable comments, and thank the National Science Council of the Republic of China (Taiwan) for financial support of this research under contract numbers NSC 97-2221-E-009-126-MY3, NSC 99-2221-E-009-102-MY3 and NSC 99-2221-E-009-104 -MY3.

References

[1] Allis, L.V., van den Herik, H.J., Huntjens, M.P.H.: Go-Moku Solved by New Search Techniques. Computational Intelligence 12, 7–23 (1996)
[2] Allis, L.V.: Searching for Solutions in Games and Artificial Intelligence, Ph.D. Thesis, University of Limburg, Maastricht (1994)
[3] Baxter, J., Tridgell, A., Weaver, L.: Learning to Play Chess Using Temporal Differences. Machine Learning 40(3), 243–263 (2000)
[4] Beal, D.F., Smith, M.C.: First Results from Using Temporal Difference Learning in Shogi. In: van den Herik, H.J., Iida, H. (eds.) CG 1998. LNCS, vol. 1558, pp. 113–125. Springer, Heidelberg (1999)

[5] Buro, M.: From Simple Features to Sophisticated Evaluation Functions. In: van den Herik, H.J., Iida, H. (eds.) CG 1998. LNCS, vol. 1558, pp. 126–145. Springer, Heidelberg (1999)
[6] Knuth, D.E., Moore, R.W.: An Analysis of Alpha-Beta Pruning. Artificial Intelligence 6, 293–326 (1975)
[7] Lin, H.-H., Sun, D.-J., Wu, I.-C., Yen, S.-J.: The 2010 TAAI Computer-Game Tournaments. ICGA Journal 34(1), 51–55 (2011)
[8] Lin, P.-H., Wu, I.-C.: NCTU6 Wins in the Man-Machine Connect6 Championship 2009. ICGA Journal 32(4), 230–233 (2009)
[9] Golem, L.: Online Connect6 games (2006), `http://www.littlegolem.net/`
[10] Renju International Federation, The International Rules of Renju (1998), `http://www.renju.nu/rifrules.htm`
[11] Schaeffer, J., Hlynka, M., Jussila, V.: Temporal Difference Learning Applied to a High-Performance Game-Playing Program. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence, pp. 529–534 (August 2001)
[12] Schraudolph, N.N., Dayan, P., Sejnowski, T.J.: Learning to Evaluate Go Positions via Temporal Difference Methods. In: Baba, N., Jain, L. (eds.) Computational Intelligence in Games, vol. 62. Springer, Berlin (2001)
[13] Silver, D.: Reinforcement Learning and Simulation-Based Search in Computer Go, Ph.D. Dissertation, Dept. Comput. Sci., Univ. Alberta, Edmonton, AB, Canada (2009)
[14] Simon, H.: Adaptive Filter Theory. Prentice Hall (2002)
[15] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998)
[16] Taiwan Connect6 Association, Connect6 homepage (2007), `http://www.connect6.org/`
[17] TCGA Association, TCGA Computer Game Tournaments, `http://tcga.ndhu.edu.tw/TCGA2011/`
[18] Tesauro, G.: TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play. Neural Computation 6, 215–219 (1994)
[19] Thomsen, T.: Lambda-Search in Game Trees - with Application to Go. ICGA Journal 23, 203–217 (2000)
[20] Trinh, T., Bashi, A., Deshpande, N.: Temporal Difference Learning in Chinese Chess. In: Mira, J., Moonis, A., de Pobil, A.P. (eds.) IEA/AIE 1998. LNCS, vol. 1416, pp. 612–618. Springer, Heidelberg (1998)
[21] Veness, J., Silver, D., Uther, W., Blair, A.: Bootstrapping from Game Tree Search. In: Bengio, Y., Schuurmans, D., Lafferty, J., Williams, C. K. I., Culotta, A. (eds.), Advances in Neural Information Processing Systems 22. pp. 1937–1945 (2009)
[22] Wu, I.-C., Huang, D.-Y.: A New Family of *k*-in-a-Row Games. In: van den Herik, H.J., Hsu, S.-C., Hsu, T.-s., Donkers, H.H.L.M(J.) (eds.) CG 2005. LNCS, vol. 4250, pp. 180–194. Springer, Heidelberg (2006)
[23] Wu, I.-C., Lin, P.-H.: NCTU6-Lite Wins Connect6 Tournament. ICGA Journal (SCI) 31(4), 240–243 (2008)
[24] Wu, I.-C., Lin, P.-H.: Relevance-Zone-Oriented Proof Search for Connect6. IEEE Transaction Computer Intelligence AI Games 2(3) (September 2010)
[25] Wu, I.-C., Yen, S.-J.: NCTU6 Wins Connect6 Tournament. ICGA Journal (SCI) 29(3), 157–159 (2006)
[26] Wu, I.-C., et al.: The Search Techniques in NCTU6 (in preparation)
[27] Wu, I.-C., Huang, D.-Y., Chang, H.-C.: Connect6. ICGA Journal 28(4), 234–242 (2006)
[28] Wu, I.-C., Lin, H.-H., Lin, P.-H., Sun, D.-J., Chan, Y.-C., Chen, B.-T.: Job-Level Proof-Number Search for Connect6. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 11–22. Springer, Heidelberg (2011)
[29] Wu, I.-C., Lin, Y.-S., Tsai, H.-T., Lin, P.-H.: The Man-Machine Connect6 Championship 2011. ICGA Journal 34(2), 103–106 (2011)