On Verifying Game Designs and Playing Strategies using Reinforcement Learning

Dimitrios Kalles
Computer Technology Institute
Kolokotroni 3
Patras, Greece
+30 - 61 – 221834
kalles@cti.gr

Panagiotis Kanellopoulos
Computer Technology Institute
Kolokotroni 3
Patras, Greece
+30 - 61 - 221834
kanellop@ceid.upatras.gr

Keywords
Reinforcement learning, machine learning, games, playability, design verification.

ABSTRACT
In this paper we elaborate on the application of reinforcement learning to the design of a new strategy game. We deal with playability and learning issues, attempting to use intelligently generated self-playing sequences to determine playability of various initial board configurations. The machine's *a priori* knowledge about the game is restricted to the rules only, so, the initially encouraging and intuitive results suggest that this design verification strategy may be useful to a board range of design problems.

1. INTRODUCTION
Scientists have been focusing on game theory since many decades. By using artificial intelligence methods, there is an ongoing effort to create "smart" programmes, that can compete with humans in several games. In 1955, Samuel [6] created a checkers programme, however chess programmes have been developed since the sixties. During the last decade, IBM has made strenuous efforts to develop (first with Deep Thought, later with Deep Blue) a chess programme equivalent to the best human player. Checkers experienced also a fascinating development, yet less AI-based [7].

A classic method used is based upon the creation of a tree, where the numerous game states are modeled as vertices and the possible moves as edges. By searching the tree in depth, using the *minmax* algorithm [9], the "best" move is computed, according to the value estimation of each vertex-position. In order to accelerate computation, we can use the (α, β) pruning method [3], to search the tree in more depth in the same amount of time. However, it should be clear that in the above case, the programmer's ability in the particular game is a vital component to the success or failure of the experiment, due to the need of correctly modeling several attributes at the same time. Moreover, the "Achilles' heel" of this approach is the *horizon effect*, which means that the value estimated depends on the depth that the tree is searched, resulting sometimes in bad estimations.

SAC 2001, Las Vegas, NV

On the above grounds, as well as for the sake of experimentation, we chose to make use of *machine learning* methods, and more specifically *reinforcement learning (RL)*. There have been already developed several games, that exploit RL, the most prominent of which is TD-Gammon by Tesauro [15], which led even to the reconsideration of the policy human players use. Afterwards, games like Tetris, Blackjack, Othello [5], chess [16] etc, were analyzed under the same approach (see also [1][8][10] for alternative approaches and related experiences).

For the sake of completeness, we note that RL can be briefly described as producing an iterative optimization of the values of search states in a search space, using a guided battery of trial-and-error experiments.

The aim of this research is the discovery of a "smart" strategy for the strategy game defined in the next section, whereas we should emphasize the fact that the machine's initial knowledge for the game is restricted to the rules, and there is no other *a priori* knowledge. By aiming to utilize RL, not only can we make an attempt at determining the playability of the game, but we also pave the way for conjecturing that machine learning methods provide a powerful toolkit in the improved *design* of rule-based games.

2. DESCRIPTION OF THE GAME
The game, designed by one of the authors, is played on a rectangular board, with each side having size n, where at the lower left and the upper right part of the board the players' bases are located, also rectangular of size α for each side. Each player possesses β pawns, that initially are inside the corresponding base, which is considered as one square and not as a set of squares (see Figure 1). Each player's goal is to get one of his pawns into the enemy base. If a player runs out of pawns, the opponent is declared as winner. Each pawn can move to an empty square that is vertically or horizontally adjacent, provided that the pawn's maximum distance from its base is not decreased.

For example, a pawn that stands on square (x,y) can move to square (x,z), if

$$\max(x-a, y-a) \leq \max(x-a, z-a),$$

if the first player moves, or else if

$$\max(n-a-x, n-a-y) \le \max(n-a-x, n-a-z).$$

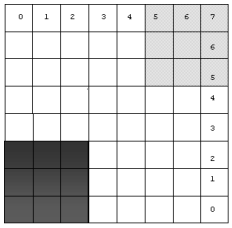

Figure 1. The game board.

This way, pawns can move "backwards" only towards one dimension, so that their placing requires attention and discretion. Note that such a movement is considered "backward" from the point of view of the enemy; regarding its own base a pawn either advances or maintains the same distance. Figure 2 shows examples of legal and illegal backward moves, where pawn X can move ''backwards'' only to the ''OK'' square.

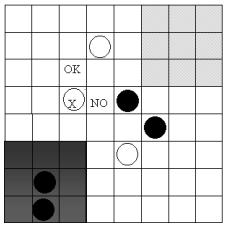

Figure 2. Backward moves.

As soon as a move has taken place, all pawns that have no legal move are pronounced "inactive" and are removed. Figure 3 shows such an example, where pawn *X* cannot move and is removed, if it is that player's turn to move.

This rule prevents the frontal engagement of the two "armies" and the total blocking of the position, where upon no moves would be possible. The rule can be considered as enforcing a type of inertia on an army.

Since the squares, that are adjacent to the base, are actually the "winning" squares, each player's goal is to place one of his pawns in such a square. Figure 4 shows a situation, where the player to move can win the game, while in Figure 5 the white player must move one of his "base" pawns to square (5, 4) or else black wins by playing there.

Figure 6 describes a short game, from start to end (moves are left to right, top to bottom).

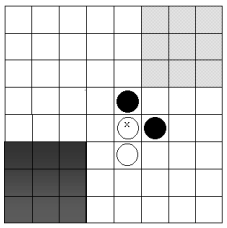

Figure 3. An example of a pawn turned inactive.

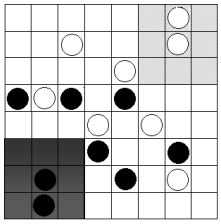

Figure 4. An example of a winning configuration.

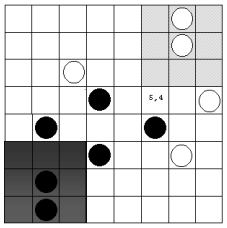

Figure 5. An example of a configuration leading to a forced move.

2.1 Playability "defined"

An important feature of this game is that (to our knowledge) **it has, practically, never been played by humans**. This implies our ignorance of how interesting and challenging it is, by means of the different policies that could be tried.

We have set out with the goal that the two players should have roughly the same chances to win. A counterexample is "tic-tac-toe", where several playing policies are well-known and the first player has significantly better chances. Furthermore, the existence of a "defending" policy -that could provably prevent defeat- would be a setback, as this would make the game trivial.

3. ANALYSIS OF THE GAME USING REINFORCEMENT LEARNING

Our approach is based on reinforcement learning. The *a priori* knowledge, the machine possesses, consists of the rules of the game, because this way there is more flexibility during the learning procedure, since it is not subject to any bias. Our aim is, through a number of "self-playing" games, to improve the machine performance. RL is based on a fact, familiar from psychology, that the consequences of an action affect positively or negatively its recurrence.

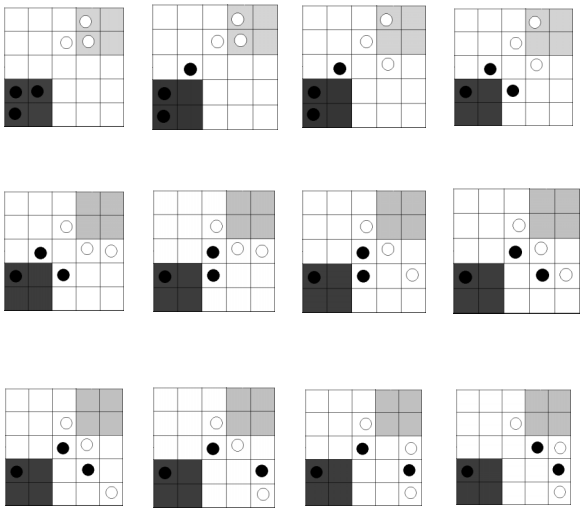

Figure 6. A short game.

RL combines the fields of dynamic programming and supervised learning to yield powerful machine-learning systems. It is an approach to machine intelligence that pieces two disciplines together to successfully solve problems that neither discipline can individually address. RL is almost an orthogonal approach to other machine learning paradigms, though as will be shown again, it can team up with at least the neural networks paradigms. In RL, the computer is simply given a goal to achieve; and then learns how to achieve that goal by trial-and-error interactions with its environment.

This environment must at least be partially observable by the reinforcement learning system, and the observations may come in the form of sensor readings, symbolic descriptions, or possibly "mental" situations (e.g., the situation of being lost). The actions may be low level (e.g., voltage to motors), high level (e.g., accept job offer), or even "mental" (e.g., shift in focus of attention). If the *reinforcement learning* system can observe perfectly all the information in the environment that might influence the choice of action to perform, then the system chooses actions based on true "states" of the environment. This ideal case is the best possible basis for reinforcement learning and, in fact, is a necessary condition for much of the associated theory.

Before moving on, we reiterate same basic nomenclature.

By *state s* we mean the condition of a physical system as specified by a set of appropriate variables. A *policy* determines which action should be performed in each state; a policy is a mapping from states to actions. *Reward r* is a scalar variable that communicates the change in the environment to the reinforcement learning system. For example, if an RL system is a controller for a missile, the reinforcement signal might be the distance between the missile and the target (in which case, the RL system would learn to minimize reinforcement). The *value V(s)* of a state is defined as the sum of the rewards received when starting in that state and following some fixed policy to a terminal state. The optimal policy would therefore be the mapping from states to actions that maximizes the sum of the rewards when starting in an arbitrary state and performing actions until a terminal state is reached. Under this definition the value of a state is dependent upon the policy. The *value function* is a mapping from states to state values and can be approximated using any type of function approximator (e.g., multi-layered perceptron, memory based system, radial basis functions, look-up table, etc.)

Temporal difference (*TD*) learning [13] is a very exciting approach to reinforcement learning. It is a combination of Monte Carlo and dynamic programming ideas. TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (this is called *bootstrapping*). Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to $V(s_t)$ (only then is the reward known), TD methods need wait only until the next time step. *Eligibility traces* are one of the basic mechanisms of reinforcement learning. For example, in the *TD(λ)* algorithm, λ refers to the use of an eligibility trace and reflects the learning experience (past moves considered). It can be seen as a temporary record of the occurrence of an event, such as the visiting of a state. When a TD error occurs, only the eligible states or actions are assigned credit or blame for the error.

The game is a discrete Markov procedure of discrete time, since there are finite states and moves, while each episode does terminate. The algorithm in use is *TD(λ)*, as described in [14]. Like the methods presented in the introduction, each position is modeled with a vertex and each move with an edge. The difference is that it suffices to check only the adjacent vertices, instead of the entire tree. Each value is backed up according to the following equation.

$$V(s)_{new} = V(s)_{old} + ae(s)[r + V(s') - V(s)$$

where s is the state-position, $V(s)$ its value, $e(s)$ the eligibility trace, r the reward from the transition, α the learning rate and s' the resulting state-position.

The complexity of this game depends mainly on the value of parameters n, α, and β. The number of the several positions that might occur, is bounded from above by

$$\sum_{i=0}^{\beta}\sum_{j=0}^{\beta}\binom{n^2-2a^2}{i+j}\binom{i+j}{i}(\beta+1-i)(\beta+1-j)$$

A better estimation based on n, α, β is not possible, as the validity of a position depends does not on the number of the pawns, but on their exact placing on the board. For large values of n, α, and β this number gets too large, so that the use of a map with each different position becomes unrealistic. Hence, there is a need for generalization, e.g. through a neural net like in [15], see also section 3.2 of this paper.

3.1 Implementation Issues For the Tabular Case

We used *temporal difference learning, TD(λ)*, and more specifically the *on-line, tabular* algorithm. Thus, the update takes place after each move, and not in batch mode. The discount parameter is set to one (γ=1), because the game is of discrete and finite time, so that the reward is not diminished. Moreover, we chose a medium value for λ, which is λ=0.5, meaning that the *credit assignment* is practically restricted to the last 6-7 moves. Regarding parameter α, that determines the convergence rate, the choice was to use the value 0.001. As to the eligibility trace, we

chose to use *replacing* [11] instead of *accumulating* traces, because the latter approach has been known to suffer from some weaknesses, the main one being that a repeated wrong action hinders learning, due to a large bad trace. These weaknesses are well analyzed in [12].

Thus, for each state s ,

$$e_t(s) = \gamma\lambda e_{t-1}(s), \quad s \neq s_t$$

otherwise

$$e_t(s) = 1, \quad s = s_t$$

where s_t is the state selected at time t. Accumulating traces are alike, except for the chosen state; then

$$e_t(s) = \gamma\lambda e_{t-1}(s) + 1, \quad s = s_t .$$

Finally, each action-move is given reward *-1*, unless the resulting state is a final one; then it is *+50* for the winner and *-50* for the loser's last move, since it led to defeat.

The encoding used to describe the base, is to name each square after its axis coordinates, i.e. every square is named (*x,y*), with *x, y* taking values from 0,…,*n*-1. Each position is described uniquely with a string, that contains as substrings the placing of the two opponents' pawns. Each substring is computed through the equation

$$S = \sum_{j=1}^{\beta'} n^{2j}(x_j n + y_j + 1)$$

with β' being the number of "active" pawns, that are sorted according to their coordinates.

The policy, that the machine uses to select between moves is the ε -greedy policy, with ε *=0.9* for both players, i.e. with probability *0.9* the machine chooses the best, according to present knowledge, move (*exploitation*), otherwise a random move is played (*exploration*).

The array that contains the state values -the reason why the approach is called tabular- is stored not as one piece, but as *clusters* of values, and a data structure (hashtable) is used, in order to store the states during execution time, to reduce the burden of hard disk access time. A possible mapping of each state to a file, would create many files, and as most systems have a minimum disk space that a file needs, there would be a significant waste of capacity.

3.2 Implementation Issues For the Neural Network Case

As stated above, for most values of *n*, α, and β the use of the tabular algorithm is impossible due to memory requirements. Thus, there is a need for generalization, so that we can approximate the whole state space. To do that we use an artificial neural network. In fact, we use two networks, one responsible for each player's behavior, because the state spaces for the two players are separated, having no state in common. Unlike games like chess or backgammon, each position is associated to the player, whose turn is to move, i.e., a specific board position with player A to move cannot ever appear on player's B turn. Thus, each player has a unique state space to learn. This fact enables further experimentation, because we can try different configurations for the two opponents.

The algorithm used for the training was "vanilla" backpropagation, with γ=0.95 and λ=0.5. The reward given for victory is now +1, for defeat -1 and 0 otherwise. Using $\gamma \neq 1$ we favor quick victories, as the reward lessens decreases over time.

Network weights constitute a vector $\vec{\theta}$, where the update occurs according to

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\delta_t \vec{e}_t$$

where δ_t is the TD error,

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)$$

and $\vec{e}_t$ is a vector of eligibility traces, one for each component of $\vec{\theta}_t$, updated by

$$\vec{e}_t = \gamma\lambda \vec{e}_{t-1} + \nabla_{\vec{\theta}_t} V_t(s_t) ,$$

with $\vec{e}_0 = \vec{0}$.

On representation issues, each of the $n^2 - 2a^2$ "free" squares is assigned two input nodes, one for each player, that describe whether there is a pawn on that square. Two more nodes show if a pawn has captured the enemy base, and eight more nodes serve to show the number of pawns that still reside in the bases. More specifically, we check if this number exceeds $\frac{\beta}{4}$, $\frac{\beta}{2}$ or $\frac{3\beta}{4}$ and "turn on" the appropriate node. Thus, we have a total of $n^2 - 2a^2 + 10$ input nodes, and half as many hidden nodes. All nodes make use of the nonlinear sigmoid function, $h(j) = \frac{1}{1 + e^{-\sum_i w_{ij}\phi(i)}}$, whose values range from zero to one. Thus, the only output node can be regarded as an approximator of the probability of winning, beginning from a specific position.

Fast convergence to the real state values depends on the parameters. The most important factor seems to be the *rewards* being used, because they have a deep impact on the machine's bias towards different moves. In order to give more credit to a short victory over a longer one and to avoid repetition of moves, each move is given a small negative reward (in fact a penalty). If the reward given for victory is small compared to each move's reward, then if the "episode" lasts long enough, the total reward could be negative. This would mean that the machine gets discouraged to repeat such actions, although they led to success. Moreover, if the reward is big enough, then a good move that was followed by poorer choices and finally led to defeat, would face a significant decrease in its value.

Another important factor is λ, because it determines the point until which *credit assignment* takes place. When λ=0 only the last state has its value affected, while as λ approaches 1 almost the entire sequence gets updated. An appropriate choice for λ allows faster

convergence. As far as the learning rate α is concerned, we can be assured of convergence (with probability 1) only if

$$\sum_{\kappa} a_{\kappa} = \infty , \quad \sum_{\kappa} a_{\kappa}^2 < \infty$$

In our application, a small enough value (*0.001*) was decided on implementation grounds.

Due to the *self-play* approach, we can experiment with the policy the two opponents follow. For example, they could have different ε in *ε-greedy* policy for deciding their actions, as to enable more experimentation, while if ε=1 for one of the opponents, then we have learning versus a specific deterministic opponent.

3.3 A Summary of the Experimental Results

The alternatives described above were fully implemented to experiment with.

During experiments with the tabular approach and small parameter values (n=5, α=2, β=1) we observed that the games are too short, the majority lasted the minimum number of 4 moves, and the first player wins most of the time. This could mean that the machine does not understand "blocking" as a defensive mean. If two humans played under such configuration, then the first player should always win. Approximately 5,000 games were analyzed.

With different values (n=6, α=2, β=3), we observed that each player wins about the same number of games, very few games last the minimum possible of 6 moves, with some games lasting as long as 40 moves. Approximately 30,000 games were analyzed.

On the other hand, during experiments with the neural net approach, with larger parameter values (n=8, α=2, β=5), we observed that almost all games last more than 100 moves, each player winning approximately half of the games. After 74,000 games, the average number of moves per game is 184.5, while the standard deviation is 108.4. White won 49.26% of the games, while the rest 50,73% were won by black. Table 1 shows for every 10,000 games the number of moves played, and the games won by white and by black.

In short, the preliminary results are that for n=5,6, games are quite short, while for n=8 they seem worth playing (note that we do not detect playing cycles; see discussion next).

Table 1: Summary of game statistics

Games played *10,000	Average number of moves	Standard deviation	White wins	Black wins
0 – 1	179.94	108.02	47.83%	52.17%
1 – 2	182.88	108.44	47.72%	52.28%
2 – 3	187.87	109.17	50.30%	49.70%
3 – 4	186.88	109.58	49.98%	50.02%
4 – 5	184.90	108.62	50.03%	49.97%
5 – 6	183.53	107.56	49.00%	51.00%
6 – 7	184.76	106.87	49.79%	50.21%
7 –7.4	186.37	109.05	49.85%	50.15%

An obvious observation is that the larger the board, the larger average number of moves that a play lasts. This is a mildly positive result in the sense that it does not uncover any inherent faults in the game's logic or in the algorithms. However, the fundamental issue raised is that if one would assign time unit duration to each move, the results directly relate to the extent that a game may be interested or boring. An average of 180 moves (*per player*), as is the case for the large board may be a bit too much for human players, but then again we have not analyzed cyclic movements yet. On the other hand, it does seem that neither player enjoys an advantage (or disadvantage) of pole position, so an extra level of fairness is accommodated.

We do not have a comment yet on the number of games used in the analysis. After all, truly experimenting only as much as required seems an elusive goal; to attain it we must first positively deal with "similar" games, otherwise there is a real risk of training against automatic opponents with superficial differences. A self-learning experiment must deal effectively and efficiently with cycles in the state-space search.

4. EXTENSIONS

There are a number of key research and development extensions that need to be made to validate the game's playability and improve its analysis. These range from improving accessibility to the game software, to analyzing alternative starting configurations (including selected openings) and endgames, to experimenting with different rule sets, and to exploring multi-player (more that two) games for agent-oriented research. We elaborate on those directions below.

Our next step would be to make the program public, through the Internet. That would make the learning procedure more interactive; and would also result in a critical test regarding playability; not to mention the possibility of attracting people to actually test it out in an educational context [4].

That this is a necessity prompted us to write all code in Java. However, this has resulted in inefficiencies. From an architectural-implementation point of view, we plan to separate the basic computation-intensive layer from the interface.

Furthermore, since there is no "benchmark" program for the game, we would like to train a neural network and when it reaches an acceptable level of play, we can stop the learning procedure and use it as a benchmark. It would be interesting to test how efficiently machines with different choices of parameters, for the same configuration, improve when playing versus the "benchmark" machine.

To improve our machine's ability, we can combine RL with conventional searching e.g. we can use two-ply search and only then apply learning. Experimenting with board and base size is also interesting. Intuitively, a small base size would result in more space to maneuver, but would make tenacious defense easier to accomplish. Moreover, when playing with few pawns, the first

player seems to have an important advantage (as we have already stated there is no way for the second player to win when β=1).

Adding "special" features either to the board or to the pawns is worth experimenting. Each pawn could be able to "restart" from its base, no matter its position, once in its "lifetime", or it could make a "backward" move for a restricted number of times. Placing obstacles in the middle of the board, (e.g. the central squares could be impassable) as if a forest or a lake resides there, would probably force the two opponents to make use of the whole board. Alternatively, the central squares could be seen as a temporary position for a pawn, e.g. a pawn could stay in the centre for a limited number of moves.

Futhermore, we can add more players (probably two more, for a total of four, with one base each), so we can experiment with multiagent learning. Allowing alliances, e.g. diagonally opposed allies, would lead to team play learning.

On a more research-oriented direction, the speed-up of the RL process would benefit the game analysis. To this end, the incorporation of Explanation Based Learning strategies might prove useful [2]; the concept is clearly interesting but its applicability has been limited to date. Yet, it seems that it may be a good idea to couple RL with deductive approaches, since these could lead to discovering (automatically) circular or semi-circular paths in the learning trajectory. Note that our current approach does not deal with cycles and we have no firm position as to whether we should hand-test the generated games for cycles, and then implement a work-around, or we should allow RL to discover them -and appropriately circumvent them. Furthermore, the EBL-RL framework could help resolve whether the present game can be dealt with in pieces (opening, endgame) or a more ''continuous'' scheme is called for. In the latter case it may turn out that a player may have to first achieve a good status in the game, then destroy it to trade it with an advance. Classical RL will probably find this very difficult.

We have decided to first invest our efforts towards improving the game's playability and then towards further research on actual algorithms.

The above extensions and variations to the basic theme can be rapidly designed and tested, as regards playability and interestingness. It is crucial to observe that RL need not be aware of those various options; it can evolve playing policies without such knowledge. So, in essence, **we posit that** with the methodological steps described in this paper, **one can substantially speed-up the design of games** where strategy is core, by efficiently and massively experimenting with various rule sets.

To this end, we believe that this paper not only validates the approach put forward by other games-analyzers [2,9,10] but, most importantly, it puts-forward that using such analysis techniques one can verify the design of a game, thus saving enormous amounts of man-effort. It is also extremely promising in opening up the experimental validation of designs where different sequences of actions may lead to very different conclusions, in *any* application field.

We would also hope that the games industry is listening.

5. REFERENCES

[1] M Buro. How machines have learned to play Othello. IEEE Intelligent Systems 14, No. 6, Nov-Dec 1999, pp. 12-14.

[2] Dietterich, T. G., Flann, N. S. (1997). Explanation-based Learning and Reinforcement Learning: A Unified View. Machine Learning, 28, 1997, pp. 169-210.

[3] D.E Knuth, R.E Moore. An analysis of alpha beta pruning. Artificial Intelligence 6(4), 293-326, 1975.

[4] D. Kumar. Pedagogical Dimensions of Game Playing. ACM intelligence: New visions of AI in Practice 10, No. 1, Spring 1999, pp. 9-10.

[5] A. Leouski. Learning of Position Evaluation in the Game of Othello. Master's project: CMPSCI 701. University of Massachussetts, Amherst, 1995.

[6] A. Samuel. Some Studies in Machine Learning Using the Game of Checkers. IBM Journal of Research and Development 3, 210-229, 1959.

[7] J. Schaeffer. One Jump Ahead: Challenging Human Supremacy in Checkers. Springer Verlag, 1997.

[8] J. Schaeffer. One Jump Ahead: Challenging Human Supremacy in Checkers. Springer Verlag, 1997.

[9] C.E. Shannon. Programming a computer for playing chess. Philosophical Magazine 41, 256-275, 1950.

[10] B. Sheppard, Mastering Scrabble. IEEE Intelligent Systems 14, No. 6, Nov-Dec 1999, pp. 15-16.

[11] S.P. Singh, R.S. Sutton. Reinforcement Learning with replacing eligibility traces. Machine Learning 22, 123-158, 1996.

[12] R.S. Sutton. Temporal Credit Assignment in Reinforcement Learning. Ph.D. Thesis, University of Massachusetts, Amherst, 1984.

[13] R.S. Sutton. Learning to Predict by the Methods of Temporal Differences. Machine Learning 3, 9-44, 1988.

[14] R.S. Sutton, A.G. Barto. Reinforcement Learning. An Introduction. MIT Press, Cambridge, Massachussetts, 1997.

[15] G.J. Tesauro. Temporal Difference Learning and TD-Gammon. Communications of the ACM 38, 58-68, March 1995.

[16] S. Thrun. Learning to Play the Game of Chess. Advances in Neural Information Processing Systems 7, 1995.