Self-learning Game Player

Connect-4 with Q-learning

TONIE JAKOBSSON
and JOHN PERSSON

KTH Computer Science
and Communication

Bachelor of Science Thesis
Stockholm, Sweden 2011

Self-learning Game Player

Connect-4 with Q-learning

T O N I E   J A K O B S S O N
a n d   J O H N   P E R S S O N

Bachelor’s Thesis in Computer Science (15 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2011
Supervisor at CSC was Johan Boye
Examiner was Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2011/
jakobsson_tonie_OCH_persson_john_K11064.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Preface

We choose this subject because of both large interest in computer and console games, and we thought that this was a way to get some insight in that field of expertise. Most of the time during the progress of this project has been spent side by side. In the initial phase both of us gathered information about the subject to gain in depth knowledge of it. Later on the work was divided so that John with more skills with programming handled the implementation of the program, while Tonie with his experience in writing reports handled that matter. During the final phase both of us pitched in to give assistance when the other part was in need of such.

Abstract

The microprocessors emerged in the beginning of the 70s; with them the computers got greater capacity to process huge numbers of data. This was of great help to the software engineers who worked in the artificial intelligence (AI) field, with machine learning as a central part to AI research. Our main task is to develop a simple board game of *connect-4* then implement a self learning game player Agent using reinforcement learning. This report will include the methods that will be implemented on the Agent and the following results after a large amount of games have been executed.

Sammanfattning

Datorerna fick bättre kapacitet att hantera stora mängder data när Intel introducerade mikroprocessorerna på marknaden I början av 70-talet. Det var till stor hjälp för ingenjörerna inom mjukvaru utveckling som arbetade med artificiell intelligens (AI). Ett delområde inom AI är maskininlärning d.v.s. algoritmer som lär sig genom erfarenhet. Vårt mål är att utveckla ett simpelt brädspel *fyra i rad* och implementera en algoritm som ska lära sig spelet utan instruktioner om hur den ska agera. Algoritmen lär sig själv genom att få belöningar för lyckade omgångar och då minnas hur den gick tillväga för att lyckas när den stöter på samma scenario igen. Den här rapporten beskriver de metoder som kommer att implementeras på vår algoritm samt resultaten som uppnås.

Table of Contents

1. Introduction

The aim of this bachelor project is to explore the topic of machine self-learning and how it can be implemented in a simple board game context. We will implement a self-learning game player that will teach itself, without outside interaction, to play the game Connect-4. To do this we will use the algorithm called Q-learning. The finished product of this project will be a player that is able to learn the game of Connect-4 and successfully play it against other players. These other players will be one that uses a completely random strategy and a player that use simple tactics such as to always chose the furthest right available.

1.1 Background

The use of reinforcement learning methods are most suited to problems that are too large or stochastic to be solved by heuristic search or dynamic programming. It has been proven to be very effective, producing the best of all known methods for playing backgammon, dispatching elevators, assigning cellular-radio channels and scheduling space-shuttle payload processing[1]. Our project is about teaching a machine to play a board game and evaluate its performance. Board games has been widely regarded as an ideal testing ground for exploring concepts and approaches of AI and machine learning. This is because board games offers complexity and sophistication to play at expert level, while the rules are simple and the problem inputs and performance measures are clear-cut and well defined. Chess and backgammon are popular areas where learning systems are used, were the Rybka 4 64-bit chess player (Vasik Rajlich) and the TD-Gammon (Gerald Tesauro) is the most distinguished. TD-Gammon has competed at the very top level of international human play in backgammon. Tesauro entered the TD-Gammon in several tournaments, even though TD-Gammon won none of these tournaments. It came sufficiently close that it is now considered one of the best few players in the world[2]. The Rybka chess player managed to win over a grandmaster chess player in an event held 2008, proving to be one of the world's best chess players/machines[3].

1.2 Terminology

1.2.1 Agent

In the context of machine learning the Agent is the one who takes actions upon the Environment. The Agent then learns from its interactions with the Environment using stimulus returned to it from the Environment. In the context of this project the implemented self-learning player will be referred to as the Agent.

[1] Richard S. Sutton, Reinforcement learning.
[2] Gerald Tesauro, Temporal Difference Learning and TD-Gammon
[3] Vasik Rajlich, Rybka

1.2.2 Environment

The Environment is the stage in which the Agent acts. It is also the Environment that rewards or punishes the Agent for its actions. In the context of this project the game of connect-4 and its playing field will be referred to as the environment.

1.2.3 Markov decision process (MDP)

Markov Decision Process is a discrete time stochastic control process. At each time step, the process is in some state *s*, and the Agent may choose any action *a* that is available in state *s*. The process responds at the next time step by randomly moving into a new state *s*', and giving the Agent a corresponding reward *Ra*(*s*,*s*').

1.3 Theory

1.3.1 Connect-four

Connect-four is a game for two players, both players have 21 identical 'coins' usually red and yellow. The game is played on a vertical rectangular board with six rows and seven columns. The players take turns to put their coin in a row and the coin falls down to the lowest unoccupied slot i.e. the player makes his move. Connect-4 is a turn based type of game so the player has exactly one move per turn. The object of the game is to connect four of one's own coins of the same color next to each other vertically, horizontally or diagonally before one's opponent can do so(Fig 1 Red player wins with four red coins in a horizontally). If all 42 coins are played and no player has achieved this goal, the game is a draw[4].

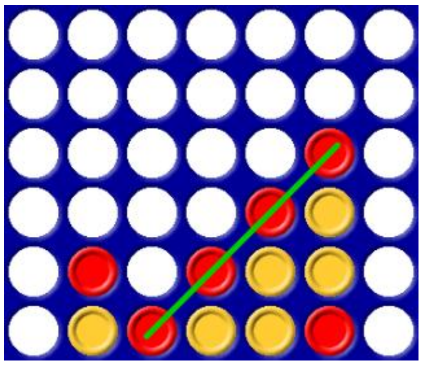

Fig 1.1 Connect-4 board

[4] Victor Allis, A Knowledge-based Approach of Connect-Four.

1.3.1.1 Complexity of Connect-Four

In order for the Agent to learn and use the knowledge of previously played games, it has to store all visited states in a table. The problem is that the number of different states which can be achieved, if the game is played according to the rules. Each slot in a Connect-Four board can be of three states i.e. empty, red or yellow so the number of possible states is at most 3^42 which is a very large sum. But the upper bound is not as large if played by the rules of connect-four. Victor Allis solved this mathematical problem by using a program written in the C programming language. His conclusion for the standard 7x6 board, was an upper bound of 7.1*10^13 (<3^42) states[4]. The database containing all possible states of the game will be too large, so we will not be able to store all the different states in the Agent.

1.3.2 Reinforcement learning

Reinforcement learning is learning what to do and how to map situations to actions, so as to maximize a numerical reward signal. The learner or the Agent in this case is not told which actions to take, but must discover which actions yield the most reward by experiencing them. This is the thing that differs between reinforcement learning and supervised learning, whereas the latter is learning from examples provided by an external supervisor. If a supervised Agent fails it will be told exactly what it should have done, whereas reinforcement learning only says that the behavior was inappropriate and how inappropriate it was. Reinforcement learning is more common in the nature and can be translated to the learning cycle of a newborn animal that has to learn how to walk for the first time. First it tries out various movements, some work and it moves forward and gets rewarded, if it stumbles and falls down it is punished with pain. In our case we would like to simulate the learning of real biological systems for our Agent. Modern reinforcement learning research uses the formal framework of MDPs (Fig 1).

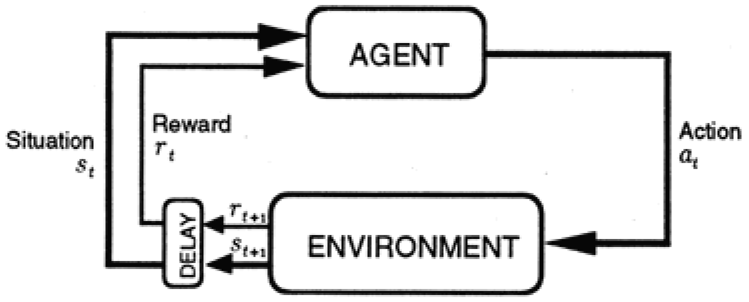

Fig 1.2 the framework of Reinforcement Learning

In this framework, the agent and environment interact in a sequence of discrete time steps, *t*= 0,1,2,3,… On each step, the agent perceives the environment to be in a state, *st*, and selects an action, *at*. In response, the environment makes a stochastic transition to a new state, *st*+1, and stochastically emits a numerical reward, *rt*+1∈ℜ. [5] The goal of reinforcement learning is to figure out how to choose actions in response to states so that reinforcement is maximized.

[5] Richard S. Sutton, Reinforcement Learning

1.3.2.1 History of reinforcement learning

There are many factors leading up to the field of reinforcement learning, one of the ideas was of trial-and-error learning that started in the field of psychology. One of the first to succinctly express the essence of trial-and-error learning in 1911 was Edward Thorndike. He had the idea that actions followed by good or bad outcomes have their tendency to be reselected altered accordingly and called it the "Law of Effect"[6].One easy way to describe it is that the Law of Effect is an elementary way of combining search and memory.

Search in the case of trying and selecting between many different actions in each situation whereas memory is the case of remembering what actions yielded the best results and associating them with the situations in which they were the best. By combining them a large and essential part of reinforcement learning is covered. One of the earliest computational investigations of trial-and-error learning was perhaps by Minsky in 1954 when he discussed computational models of reinforcement learning in his Ph.D. dissertation[5]. In it he described his construction of an analog machine composed by components called SNARCs (Stochastic Neural-Analog Reinforcement Calculators).

The idea of "optimal control" is another factor behind reinforcement learning and came in to use in the late 1950s by Richard Bellman, it describes the problem of designing a controller to minimize a measure of dynamical system's behavior over time. The methods for solving optimal control problems came to be known as dynamic programming that uses the concepts of a dynamical system's state and of a "optimal return function" to define a functional equation[5]. Bellman also introduced the discrete stochastic version of the optimal control problem known as Markovian decision processes (MDP's). These theories were essential underlying the algorithms of modern reinforcement learning.

The youngest of the theories behind reinforcement learning is that of Temporal-difference (TD). The basic idea of TD is that the learning is based on the difference between temporally successive predictions. In other words, the goal of learning is to make the learner's current prediction for the current input pattern more closely match the next prediction at the next time step[7]. Arthur Samuel was the first to propose and implement a learning method that included temporal-difference ideas, as part of his celebrated checkers-playing program in 1959[5]. In 1981 Sutton and Baro managed to develop a method for using temporal-difference learning in trial-and-error learning, known as the "actor-critic architecture". Later on Sutton introduced the TD (λ) algorithm and proved some of its convergence properties in 1988[5].

With the introduction of Q-learning from Chris Watkins in 1989, the temporal-difference and optimal control were fully brought together. It is Q-learning our algorithm will be based on when we will create a self-learning game player.

[6] Richard S. Sutton, Andrew G. Barto, Reinforcement Learning: An Introduction
[7] Gerald Tesauro, Temporal Difference Learning and TD-Gammon

1.3.3 Q-learning

Q-learning is a form of reinforcement learning in which the agent learns to assign values to state-action pairs $Q(s_t, a_t)$. If an agent enter a particular state and takes a particular action, then we are interested in any immediate reinforcement received, but also in future reinforcements resulting from new states where further actions can be taken.

The goal is to figure out how to choose actions in response to states so that the reinforcement is maximized. That is, the agent is learning a policy i.e. a mapping from states to actions. Given a particular action in a particular state followed by behavior that follows a particular policy, the agent will receive a particular set of reinforcements. The sum of all of these reinforcements for a state-action pair is called Q-value. Initially the agent has no idea of what the Q-values of any state-action pairs are, if it had it could use the knowledge to select the best action for each state. The agent's goal, is to settle on an optimal Q-value function, one which assigns the appropriate values for all state-action pairs it has and will encounter. This is achieved by using a reinforcement learning algorithm that apply directly to the agent's experience interacting with the environment, changing the policy in real time. The *tabular 1-step Q-learning algorithm*[8] uses the experience of each state transition to update one element ($Q(s_t, a_t)$) of a table.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \qquad (1.1)$$

Where $\alpha \in [0,1)$ is the learning rate parameter, the lower the parameter is the less the agent will learn. r_{t+1} is the immediate reward received for an action taken and γ is the discount factor in the range of [0,1) where a value of 0 indicates that the agent only will consider current rewards (exploitation) and a value closer to 1 will consider long-term rewards (exploration). An important aspect is that the algorithm updates the Q-value for actions that are actually attempted in states that occurred. Nothing is learned about and action that is not tried. The estimated Q-values should be stored somewhere during the estimation process and thereafter. The simplest storage form is a lookup table with every state-action pair. The problem of this method is its space complexity. Problems with a large state-action space lead to slow learning and too large tables with Q-values.

1.3.4 Exploration vs. Exploitation

For the agent to obtain a lot of reward, it has to prefer actions that it has tried in the past and found to be effective in producing reward. An agent is then *exploiting* its current knowledge of the Q-function when it selects the action with the highest estimated Q-value.

But to discover which actions these are it has to select actions that it has not tried before, so the agent also has to explore in order to make better action selections in the future. The dilemma is that neither exploitation nor exploration can be pursued exclusively without failing at the task[8]. The simplest solution to the action decision process is to choose randomly between the action with the highest Q-value(exploit) or pick a random action(explore). A better solution could be to assign probability to each action, basing it on the Q-value for the action in the state. The higher

[8] Richard S. Sutton, Reinforcement Learning

an action's Q-value, the greater the probability of choosing it. It also makes sense to have the probability of exploration depend on the length of the time that the agent has been learning.

2. Implementation

When implementing the agent for playing the game we are confronted with some practical problems. One of which is the memory required to save the data structure for the Q-learning table. We wrote the program in Java, using hashmap to store all the states of the board to a file. The program is using three different classes.
Environment is the class that builds the board and decides which players turn it is. The class also gives rewards/punishments to the winner/loser.
Qagent is the class that holds the hash functions and the learning algorithms.
SimpleAgent/RandomAgent is the class that builds the opponents against the Q-agent.

2.1 Methodology

Our Q-agent will face two different kinds of opponents, one is a simplistic agent and the other one is a random agent. We will have our Q-agent facing its opponents 10000 times each ten times and then calculate the mean value for the results, with different setting to its learning rate. Hopefully the graph will show that, over some time, our Q-agent has improved its winning proficiency. And that we could see which parameter setting is the most optimal for our Q-agent.

2.1.1 Simplistic agent

The simplistic agent will use a tactic that is very predictable for human players but not for the Q-learning agent. The simplistic agent will always put its 'coin' in the first available left slot it can find. This tactic will not generate much different states because the simplistic player will win the most of the games in the beginning until our agent find a solution to the simplistic static.

2.1.2 Random agent

The random agent will always choose where to put its 'coins' on the board randomly. In the beginning all the states have equal value for our Q-agent, which will make it choose the next move randomly. So the outcome should be that our agent will win approximately half the games played in the beginning. When two random agents is facing each other, the one that starts the game will have a larger chance to win due to that it will most of the times have one more move than its opponent and the player that starts the game gets the opportunity to put a coin in the middle slot, which put the player in favor of winning the game.[9] This should lead to that our Agent that start each game will win more than 50% of the games in the beginning.

[9] Victor Allis, A Knowledge-based Approach of Connect-Four

3 Experiment

Throughout the tests our agent had to play 100 games with the explorative threshold set to 1 (max) and after that play 100 games with the setting set to 0 (min). This was done 100 times so in the end our agent has played 20 000 games on each different learning setting. To make it easier to visualize the progress of our agent, we only plotted the data for the exploitive agent on the graphs. The different learning settings were 0.25, 0.5 and 0.75. We did all the tests three times and divided the sum from them with three to get a mean value to be sure that the results will be as correct as possible. Throughout all the tests we used a discount factor (γ) of 0.9.

3.1 Q-Agent Vs Random agent results

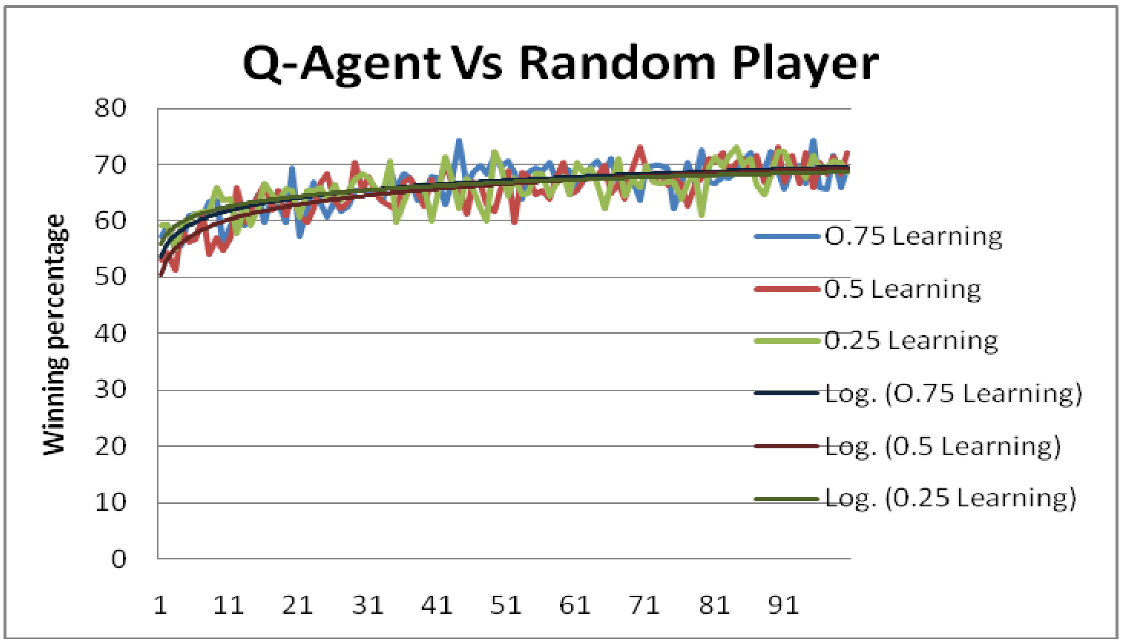

*Fig 3.1 This is the graph showing the winning percentage for our Q-agent against a random player. The x-axis shows the amount of played games (1*10^4). Each line on the graph is a Q-agent with different settings on learning rate. The logarithmic line is to see how the results converge after an amount of played games.*

The graph shows us that we got the result we were looking after. Where all the curves start with 55-59% wins and after a large amount of played games they all converge around 70%. It is a 10-15% rise off winnings against the random player. The different parameter settings did not have a dramatic difference. It where only for the first 1000 games before the curves converge that we could see that the best learning rate against a random player is 0.25. After that amount of games, the difference between learning parameters do not matter.

3.2 Q-Agent Vs Simplistic agent results

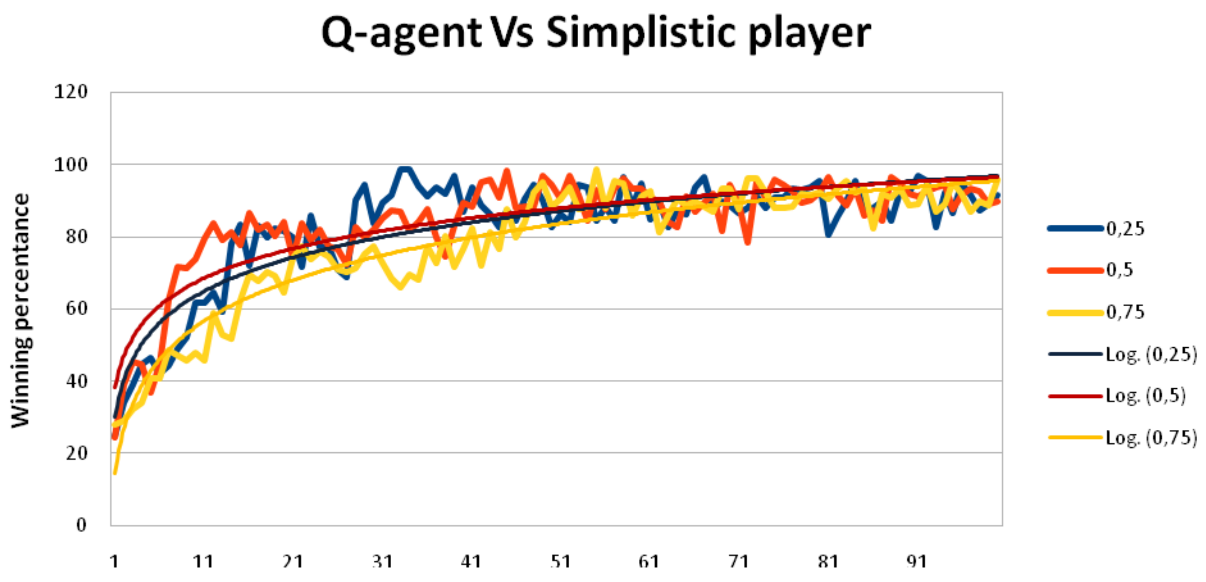

*Fig 3.1 This is the graph showing the winning percentage for our Q-agent against a simplistic player. The x-axis shows the amount of played games (1*10^4). Each line on the graph is a Q-agent with different settings on learning rate. The logarithmic line is to see how the results converge after an amount of played games.*

Our agent did have problem with the simplistic player the first 700 games, but when the Agent discovers the tactic to beat the player, the result gets better quick. The logarithmic curve start to converge after half the games played and almost ends up with a 100% winning rate. We can see the same as for the Q-agent against the random player that the different parameter settings do not matter in the end result. A learning rate of 0.5 do learn fast in the beginning but with a large spread, to compare with 0.75 learning rate that has a slow rate of learning but also has a more even spread.

4. Conclusion

The Q-agent learns quicker against the simplistic player than against the random player. Because the amount of states that the Q-agent will encounter against the simplistic player will be few, the Q-agent can exploit the simple tactics of its competitor and reach a high winning percentage earlier than against the random player. This is as we expected. The random player keeps surprising the Q-agent with new moves which, apart from making it harder to win against than the simple, also makes the Q-states table grow much faster.

Raising the Learning Rate makes the Q-agent more prone to use only the most recent information. In our tests we saw little difference in the end results from changing it (between 0.25, 0.5 and 0.75). A Learning Rate of 0.5 had the fastest response in the tests against the simple player. But its winning percentage was more spread than the slower but steadier Q-agent with Learning Rate 0.75. Setting the Learning Rate to 0.25 gave a faster result than 0.75 but its winning percentage was even more erratic than the one with a rate of 0.5. The rate of 0.25 also gave a slightly higher convergence in the end than the rate of 0.5. In the tests against the random player there was hard to discern any sure trend amongst the different Learning Rates. They all started to converge around 400 played games.

We do believe there is a reason that the Learning Rate is a parameter and not a constant. Leaving it as a constant 0.5 would not hurt in our implementation but there might be areas where a Q-agent could be better off with either a higher or lower rate. One way to is to implement a Q-agent that, with time, lowers its Learning Rate as it fills its Q-table which would lead to a more even convergence in the long run.

The aim for this bachelor essay was to implement a learning algorithm to the board game connect-four, and see if we could make it learn without supervision. The results we got from the agent facing a random player and especially the simple player shows that we succeeded with our goal.

5. References

CCRL, Computer Chess Rating List
http://www.computerchess.org.uk/ccrl/4040/rating_list_all.html
Date of Access 2011-02-08

Gerald Tesauro, Temporal Difference Learning and TD-Gammon
http://www.research.ibm.com/massive/tdl.html
Association for Computing Machinery, 1995.
Date of access 2011-03-01

Leslie Pack Kaelbling, Michael L. Littman, Reinforcement Learning: A Survey
http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a-html/rl-survey.html
Computer Science Department, Brown University, Providence, USA, 1996
Date of access 2011-03-01

Richard S. Sutton, Reinforcement learning.
http://webdocs.cs.ualberta.ca/~sutton/papers/Sutton-99-MITECS.pdf
Department of computer science, University of Alberta, CA, 1998
Date of access 2011-02-08

Richard S. Sutton, Andrew G. Barto, Reinforcement Learning: An Introduction
http://webdocs.cs.ualberta.ca/~sutton/book/ebook/the-book.html
The MIT Press, Cambridge, Massachusetts, London, England
Date of Access 2011-04-06

Vasik Rajlich, Rybka
http://www.rybkachess.com/index.php?auswahl=Events
Date of Access 2011-02-08

Victor Allis, A Knowledge-based Approach of Connect-Four.
http://www.connectfour.net/Files/connect4.pdf
Department of Mathematics and Computer Science, Amsterdam, The Netherlands, 1988.
Date of access 2011-02-27