STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLVI**, Number 2, 2001

A REINFORCEMENT LEARNING INTELLIGENT AGENT

GABRIELA ŞERBAN

Abstract. The field of Reinforcement Learning, a sub-field of machine learning, represents an important direction for research in Artificial Intelligence, the way for improving an agent's behavior, given a certain feed-back about his performance. In this paper we design an Intelligent Agent who learns (by reinforcement) to play the "Game 21", a simplified version of the game "Black -Jack". The algorithm used for training the agent is the **SARSA** algorithm.

Keywords: Artificial Intelligence, Reinforcement Learning, Intelligent Agents.

1. Intelligent agents

An *agent* [1] is anything that can be viewed as **perceiving** its environment through *sensors* and **acting** upon that environment through *actions*. One of the task of an agent is to assist the user, achieving tasks in his place, or teaching the user what he should do. An *agent* is characterized by:

- the *architecture part*, or the agent's behavior - the action performed after any given sequence of percepts;
- the *program part*, or the agent's built-in part - the internal functionality of the agent.

The aim of Artificial Intelligence is to design the agent program: a function that implements the agent mapping from percepts to actions.

So, an artificial intelligent agent should be endowed with an *initial (built-in) knowledge* and with the capability of *learning*. The learning capability ensures the agent's *autonomy* - the capability of deducing his behavior from its own experience.

An intelligent agent has a *utility function*, which measure the performance of the agent's actions. The utility is a function that associates a real number to an agent's state, as a measure of the state's degree of success (a state preferred by the agent in comparison with others has a bigger utility function).

2000 *Mathematics Subject Classification.* 68T05.

1998 *CR Categories and Descriptors.* I.2.6[**Computing Methodologies**]: Artificial Intelligence – *Learning*.

As a conclusion, the *intelligence* of an agent is included in his program part. At a given moment, the agent will choose the best way of action, as he was programmed to do it. In situations in which the program has incomplete information (knowledge) about the environment in which the agent acts and lives, *learning* is the only way for the agent to acquire the knowledge he needs in order to achieve his task.

So, an important task is to design the program part of an intelligent agent, and even more, to implement the capability of learning.

2. Reinforcement Learning

Reinforcement Learning (RL) [3] is an approach to machine intelligence that combines two disciplines to solve successfully problems that neither discipline can address individually: the fields of dynamic programming and supervised learning. In RL, the computer is simply given a goal to achieve. The computer then learns how to achieve that goal by trial-and-error interactions with its environment.

A reinforcement learning problem has three fundamental parts [3]:

- *the environment* - represented by "states". Every RL system learns a mapping from situations to actions by trial-and-error interactions with a dynamic environment. This environment must at least be partially observable by the reinforcement learning system;
- *the reinforcement function* - the "goal" of the RL system is defined using the concept of a reinforcement function, which is the exact function of future reinforcements the agent seeks to maximize. In other words, there exists a mapping from state/action pairs to reinforcements; after performing an action in a given state the RL agent will receive some reinforcement (reward) in the form of a scalar value. The RL agent learns to perform actions that will maximize the sum of the reinforcements received when starting from some initial state and proceeding to a terminal state.
- *the value (utility) function* - explains how the agent learns to choose "good" actions, or even how we might measure the utility of an action. Two terms were defined: a policy determines which action should be performed in each state; a policy is a mapping from states to actions. The value of a state is defined as the sum of the reinforcements received when starting in that state and following some fixed policy to a terminal state. The value (utility) function would therefore be the mapping from states to actions that maximizes the sum of the reinforcements when starting in an arbitrary state and performing actions until a terminal state is reached.

In a reinforcement learning problem, the agent receives a feedback, known as *reward* or *reinforcement*; the reward is received at the end, in a terminal state, or

in any other state, where the agent has correct information about what he did well or wrong.

2.1. **Q-learning.** *Q-learning* [3] is another extension to traditional dynamic programming (value iteration) and solves the problem of the *non-deterministic* Markov decision processes, in which a probability distribution function defines a set of potential successor states for a given action in a given state.

Rather then finding a mapping from states to state values, Q-learning finds a mapping from state/action pairs to values (called *Q-values*). Instead of having an associated value function, Q-learning makes use of the Q-function. In each state, there is a Q-value associated with each action. The definition of a Q-value is the sum of the (possibly discounted) reinforcements received when performing the associated action and then following the given policy thereafter. An *optimal Q-value* is the sum of the reinforcements received when performing the associated action and then following the optimal policy thereafter.

If $Q(a, i)$ denotes the value of doing the action a in state i, $R(i)$ denotes the reward received in state i and M_{ij}^{a} denotes the probability of reaching state j when action a is taken in state i, the Bellman equation for Q-learning (which represents the constraint equation that must hold at equilibrium when the Q-values are correct) is the following [1]:

$$Q(a, i) = R(i) + \sum_{j} \max_{a'} Q(a', j) \tag{1}$$

or, equivalently

$$Q(a, i) = R(i) + \gamma \cdot \max_{a'} Q(a', j) \tag{2}$$

The temporal-difference approach requires no model of the environment, so the update equation for TD Q-learning is:

$$Q(a, i) = Q(a, i) + \alpha(R(i) + \max_{a'} Q(a', j) - Q(a, i)) \tag{3}$$

which is calculated after each transition from state i to state j, and $\alpha \in [0, 1]$ is called the *learning rate*.

2.2. **Selection mechanisms.** In this subsection we present two of the simplest action-selection mechanisms (policies).

The Greedy and ϵ-Greedy policy. One of the simplest action-selection mechanism is the *Greedy* mechanism and its extension ϵ *-Greedy*. This mechanism operates this way:

- chooses the action with the maximum value Q(s,a), with the probability *1* - ϵ, if it exists;
- otherwise, chooses a random action.

The ϵ-Greedy methods are better than Greedy methods, because they allow exploration and continue to improve their chances for recognizing the optimal actions. *The SoftMax policy.* In a SoftMax policy, at the t-th game, action a will be chosen with the probability:

$$\frac{e^{\frac{Q_t(a)}{\tau}}}{\sum_{b=1}^{n} e^{\frac{Q_t(b)}{\tau}}} \tag{4}$$

where τ is a positive parameter called *temperature*. The high temperatures make almost all actions to be equal-probable.

Low temperatures make a bigger difference between the selection probabilities of actions. At the limit, when $\tau \to 0$, the SoftMax action selections behave like Greedy action selections.

2.3. **The SARSA Algorithm.** SARSA is a reinforcement Q-learning algorithm, which combines the advantages of Temporal difference learning and Monte Carlo learning methods.

From the TD methods, the algorithm takes the advantage of learning at each step, instead of waiting the end of an episode. From the Monte Carlo methods, SARSA takes the advantage of going back and using the rewards obtained in each state in order to update the values of the previous action-state pairs and the capacity of functioning without the model of the environment in which the learning takes place [4].

The idea of the SARSA algorithm [2]is to apply the Temporal Difference methods to the state-action pairs, in comparison with the classical methods, where this methods are applied only to the states. So, the update equation for the action-state pairs will be :

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{5}$$

This update is done after every transition from a nonterminal state s_t. If s_{t+1} is terminal, then $Q(s_{t+1}, a_{t+1})$ is defined as 0. This rule uses every element of the quintuple of events, $s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}$, that make up a transition from one state-action pair to the next. This quintuple gives rise to the name SARSA for the algorithm.

The convergence properties of the SARSA algorithm depends on the nature of the policy's dependence on Q (one could use ϵ-Greedy or SoftMax policies). SARSA converges with probability 1 to an optimal policy and action-value function as long as all state-action pairs are visited an infinite number of times and the policy converges in the limit to the Greedy policy.

The general form of the SARSA algorithm is given in Figure 1.

Initialize $Q(s, a)$ arbitrarily
Repeat(for each episode)
Initialize s
Choose a from s using policy derived from Q(ϵ-Greedy, SoftMax)
Repeat(for each step of the episode):
Take action a, observe r, s'
Choose a' from s' using policy derived from Q(ϵ-Greedy, SoftMax)
$Q(s, a) = Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
s $\rightarrow$ s';
a $\rightarrow$ a';
until s is terminal

FIGURE 1. The SARSA Algorithm

3. The application "Agent 21"

3.1. **General presentation.** In this section we describe an Intelligent Agent which learns by reiforcement to play the "Game 21". We propose a game with two players: a dealer (which shares the cards) and an agent, in fact the computer (which uses the SARSA algorithm for learning the game's rules and which plays against the dealer). The agent tries to win, obtaining a score (the sum of the cards' values) less or equal than 21, but greater than the dealer's score.

The probabilistic nature of this game make it an interesting problem for testing the learning algorithms, even if the learning of a good playing strategy is not obvious.

Learning from a teacher (the supervised learning) is not useful in this game, as the output data for a given state are unknown. The agent has to explore different actions and to develop a strategy by selecting and retaining the actions that maximize his performance.

The purpose of this application is to study the behavior of an agent, which initially doesn't know neither the rules nor the game's goal, and to follow the way for assimilating this goals.

After learning, the agent (the computer) wins about 54% of the games. This is due to the fact that the "Game 21" is a random game. The agent tries to manage as well as he can with the cards received from the dealer.

As an observation, even if the game has a strong probabilistic characteristic, after the training rounds the agent plays approximately identically or maybe even better than a human player (which has a random playing strategy).

3.2. **The application.** The application is written in Visual Basic 6.0 and has two parts:

- *Learning*, which trains the agent using the SARSA algorithm. Before starting the learning process, the user can select:
  - the number of episodes of the agent's training and the number of games per episode;
  - the rewards obtained by the agent when he wins or looses, the values for α and γ (the step size and the reducing factor from the basis equation of the SARSA algorithm);
  - the selection mechanisms (ϵ-Greedy or SoftMax). After the learning process, the user can see a graphic that contains the percentage of the games that the agent won and the values of the action-state pairs.
- *Game*, which simulates the game between the agent and the dealer. Before starting the game, the user can select the score at which the dealer stops, the strategy used by the agent. If the learning took place, the *current strategy* can be selected and the agent will use the experience from the training games, otherwise a *random strategy* can be selected and the agent will play randomly.

3.3. **The Agent's design.** The basis classes used for implementing the agent's behavior are the following:

- **CGame** - contains functions for starting the game, for calculating the points, calls the functions for design the cards on the game table and at the end of the game displays if a player wins or looses;
- **CPlayer** - contains the settings for the players: for the computer adds the current strategy. In this class are called the functions for designing the cards on the game table, for calculating the score. The main method of the class is *ComputerPlay*, which implements the game function for the computer;
  - the computer applies the selected strategy. If the agent has gone through all the training rounds, he will use his experience and will apply the strategy used in the training rounds (ϵ-Greedy or Softmax).
- **CPlayers** - is the class for creating the players and setting the game strategy for the dealer and the computer;
- **CReinforcementLearning** - the main class of the agent which implements the SARSA algorithm. Here the learning function for the agent is

implemented and the estimation function is updated. The main method is *Learn*.

The learning strategy is very conservative: it stops if the score is greater then 11. Anyway, it is very interesting that the algorithm determines such a threshold without knowing the game's rules or it's goal (only from experience and from the obtained rewards). In fact, the SARSA agent learns to continue (to ask for one more card) only if he is sure that this will not make him to loose. The agent discovers from experience the double value of the ace, which determines the following strategy: continues if the score is lower than 11 or he has an ace; stops in all the other situations.

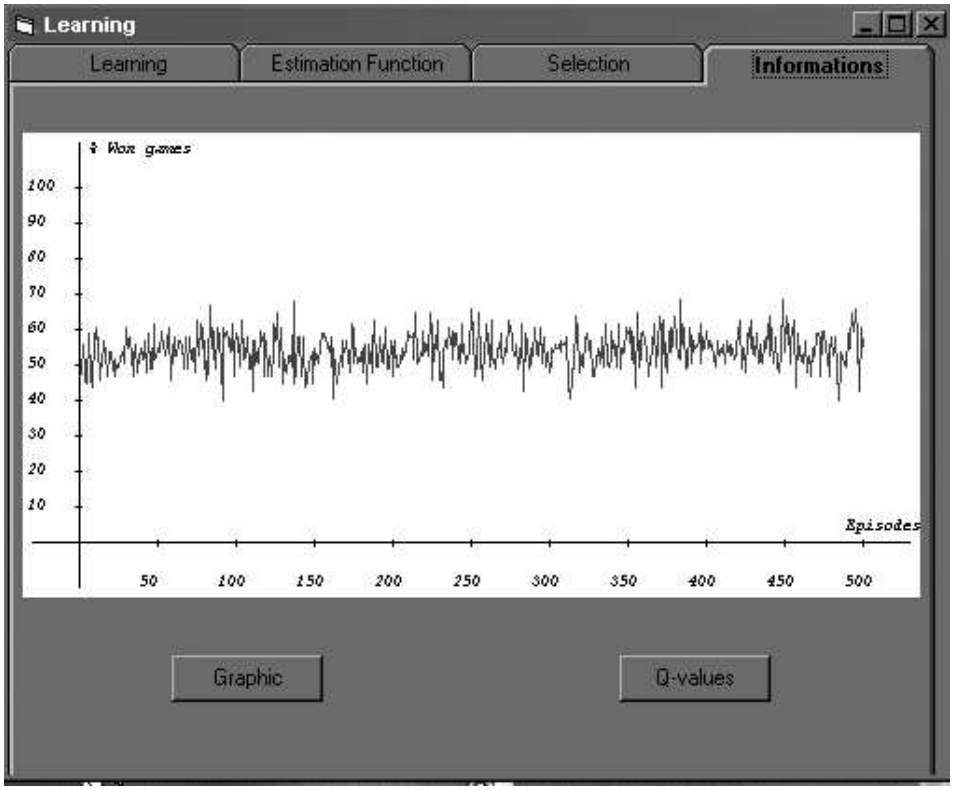

FIGURE 2

3.4. **Experiment.** Using the application "Agent 21" we accomplish the training of the agent for the following settings:

- $\alpha = 0.01$;
- $\gamma = 0.9$;
- the reward for winning the game = 1;
- the reward for loosing the game = -1;
- number of episodes = 500;
- number of games/episode = 100.

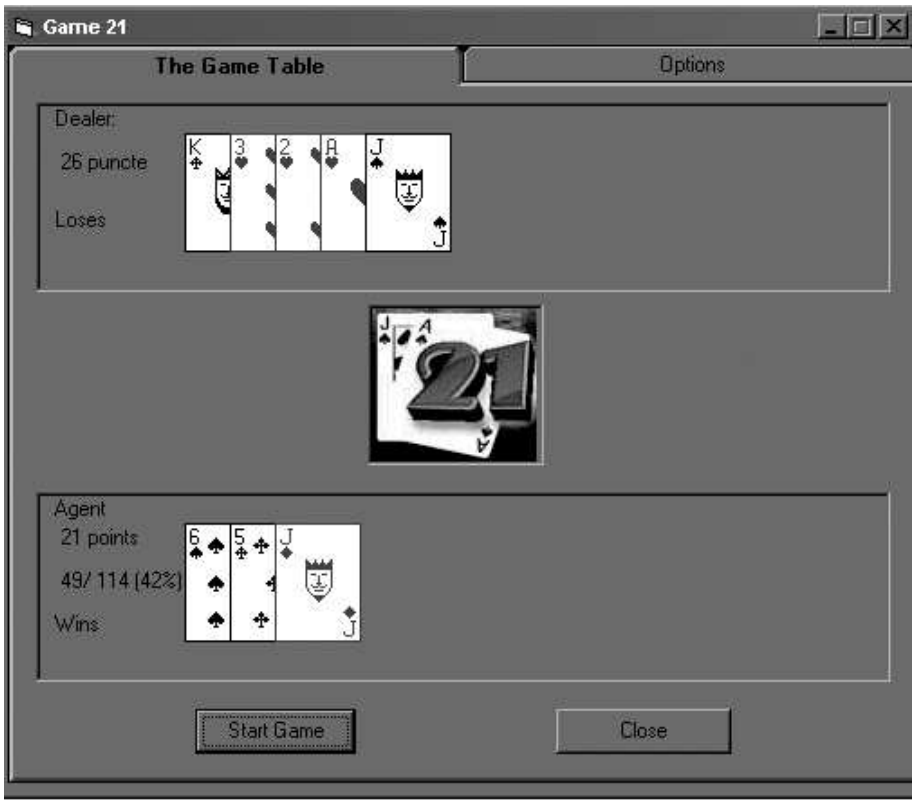

FIGURE 3. The game table with the agent's training

During the learning process, the agent wins 27188 games and loses 22812 games, that is a percentage of 54.376% of the won games. The graphic that contains the percentage of the games that the agent won during the learning is shown in Figure 2.

After the training took place, the agent uses his experience and wins about 50% of the the games (Figure 3).

Without training (with a random strategy), the agent won about 30% of the games (Figure 4).

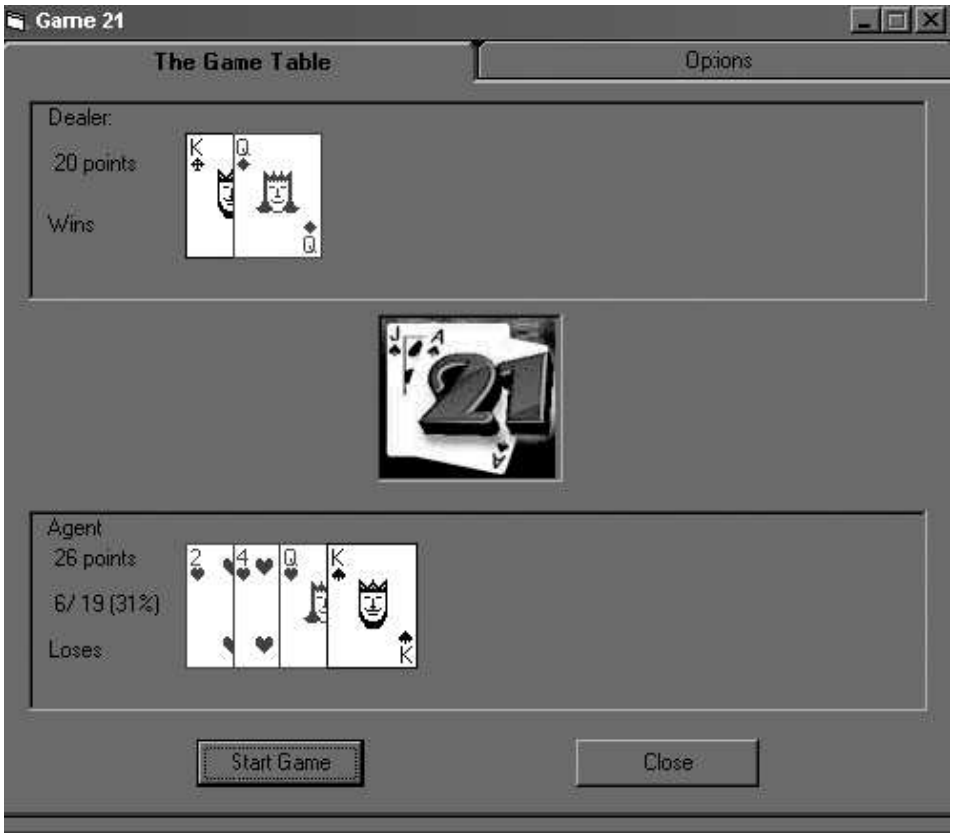

FIGURE 4. The game table without the agent's training

4. CONCLUSIONS

The agent implemented using the SARSA algorithm in the "Game 21" uses an optimal strategy. These allows him to win about 50% of the games, because of the strong random characteristic of the game. Knowing that, using a random strategy, the computer wins only about 30%, it's easy to notice the intelligence degree obtained by the agent after learning. However, if a human player and an agent, who assimilated the optimal strategy, play against the dealer, it can be observed that the number of the games won by the human player is approximately equal to the number of the games won by the agent.

Anyway, Reinforcement Learning represents an important way for improving the performance and the behavior of an Intelligent Agent.

References

[1] S.J.Russell, P.Norvig: "Artificial intelligence. A modern approach", Prentice-Hall International, 1995
[2] R. Sutton, A. Barto: "Reinforcement Learning", The MIT Press, Cambridge, London, 1998
[3] M. Harmon, S. Harmon: "Reinforcement Learning — A Tutorial", Wright State University, www-anw.cs.umass.edu./ mharmon/rltutorial/frames.html, 2000
[4] A. Perez-Uribe, E. Sanchez: "Blackjack as a TestBed for Learning strategies in Neural Networks", Proceedings of the IEEE International Joint Conference on Neural Networks IJCNN'98

Babeş-Bolyai University, Department of Computer Science, Cluj-Napoca, Romania
E-mail address: gabis@cs.ubbcluj.ro