The Improvement of Convergence Rate in n-Queen Problem Using Reinforcement Learning

Soo-Yeon Lim, Ki-Jun Son

Department of Computer Engineering, Kyungpook National University,
Daegu, 702-701, Korea

nadalsy@hotmail.com

Abstract

The purpose of reinforcement learning is to maximize rewards from environment, and reinforcement learning agents learn by interacting with external environment through trial and error. Q-Learning, a representative reinforcement learning algorithm, is a type of TD-learning that exploits difference in suitability according to the change of time in learning. The method obtains the optimal policy through repeated experience of evaluation of all state-action pairs in the state space. This study chose *n*-Queen problem as an example, to which we apply reinforcement learning, and used Q-Learning as a problem solving algorithm. This study compared the proposed method using reinforcement learning with existing methods for solving *n*-Queen problem and found that the proposed method improves the convergence rate to the optimal solution by reducing the number of state transitions to reach the goal.

Keyword: reinforcement learning, *n*-Queen problem, Q-Learning

I. Introduction

The recent explosive increase of electronic information satisfies people's various needs for diverse types of information but it is greatly burdensome to find desired information due to the expansion of the volume of data to be processed. Moreover, as the form of information is becoming diversified and complicated, we need technologies for quick and accurate information search. In this situation, one of important technologies is to detect past experiences and environmental changes and, based on them, change existing knowledge and adapt to new environment. Intelligent human agents in social environment learn not only through trial and error but also through immediate information, learned knowledge and special experiences.

Reinforcement learning does not require preliminary knowledge and learns not through examples but through experience and observation. Input information about environment can be learned in state environment, which is solely delayed scalar reward, through trial and error for efficient decision policies. Here, the goal is to maximize long-term rewards discounted for each action. Although its early learning rate is slow but it is efficient for performing complex tasks.

The present study chose *n*-Queen problem as an example, to which we apply reinforcement learning, and proposed an algorithm that exploits reinforcement learning to solve the problem. This study also carried out an experiment to compare the proposed algorithm with existing problem-solving methods in order to prove that the proposed algorithm is faster and more efficient in given environment.

II. Relevant Researches

n-Queen problem is to position n queen pieces on a $n \times n$ square chessboard. Here, a restriction is that any two queens cannot be on the same column, the same row or the same diagonal line. The number of candidate solutions is $n!$.

A. Backtracking

Backtracking is a modified depth-first search (DFS) of trees used in solving a problem of selecting the order of objects belonging to a set while satisfying certain restrictions in the set. Backtracking is the most common method of solving *n*-Queen problem. It first tests each node in a state space tree through depth-first search to see if it is promising, and if the node is not promising, it tracks back to its parent node. Backtracking is the same as depth-first search except that it visits the child nodes of a node only when the node is promising and there is no solution in the node.

B. Backtracking with MonteCarlo algorithm

One of more efficient methods of solving n-Queen problem is using MonteCarlo algorithm. MonteCarlo algorithm is a probability-based algorithm, which decides the next command to execute at random and estimates the expected values of random variables defined in the sample space based on the random means of the sample space. That is, it first deploys queen pieces at random for a number of columns and applies backtracking only to the other columns[4].

C. Reinforcement learning

Depth-first search is a common method of exploring a state space tree and finding the solution of problems. It tests each node visited during exploration and tests if the state of the node is promising, namely, is possible to be a solution.

In order to solve n-Queen problem using reinforcement learning, we must select the best among the next available nodes from each node. For this, it is necessary to estimate how good condition each node is in, and the function to produce an estimated value is called an evaluation function. Most evaluation functions evaluate nodes heuristically. The simplest algorithm is to apply an evaluation function to all selectable nodes from the current node and to diverge to the node with the largest value. Because an evaluation function evaluates basically a node, however, we do not know whether the selected node ultimately brings a win even if it is the best at present. The next section explains reinforcement learning in detail.

III. Reinforcement learning

A decision process means that an agent in a certain state performs an action and is rewarded from environment, and Markov decision process (MDP) means a decision process that a new

state is decided based on the current state and action. In particular, learning of actions to perform based on rewards from environment is called reinforcement learning.

A. Reinforcement learning model

Reinforcement learning is the combination of dynamic programming and teacher learning, in which an agent learns through trial and error while interacting with external environment. That is, the agent attempts actions that it can take against given environment while learning is executed, and is reinforced with scalar reinforcement values received from the external environment as rewards for actions the agent chose. The learning is based on MDP, reinforcing actions little by little in a better direction by receiving rewards for actions that the agent carried out.

In choosing the optimal action in the current state, MDP calculates all available actions to achieve a given goal and select the action of the highest value. However, it is disadvantageous in that it does not work when the number of states is too large or accurate probabilities or reward values are not available. Compared to MDP, reinforcement learning is an online technique close to the traditional optimization technique called dynamic programming. The method learns the optimal action in each state little by little through experiences based on many times of trial and error instead of deciding the optimal action in the current state through calculation.

Here, environment is composed of states, which are changed by actions taken. In particular, actions taken to achieve the goal are called a policy and the mission of an agent is to learn the control policy. When the agent's schedule, namely, its control policy is represented as π, what we have to solve is to find the best π, which means π that brings the highest accumulated reward.

In order to select the next action in the current state, the agent learns policy $\pi : S \to A$. We define the cumulative reward $V^{\pi}(s_t)$ achieved by following an arbitrary policy π from an arbitrary initial state s_t as follows.

$$V^{\pi}(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

Where the sequence of rewards r_{t+i} is generated by beginning at state s_t and by repeatedly using the policy π to select actions as described above (*i.e.*, $a_t = \pi(s_t)$, $a_{t+1} = \pi(s_{t+1})$, *etc.*).

Here $0 \le \gamma \le 1$ is a constant that determines the relative value of delayed versus immediate rewards. In particular, rewards received i time steps into the future are discounted exponentially by a factor of γ^i. Note if we set $\gamma = 0$, only the immediate reward is considered. As we set γ closer 1, future rewards are given greater emphasis relative to the immediate reward. The quantity $V^{\pi}(s)$ defined by above equation is often called the discounted cumulative reward achieved by policy π from initial state s.

In MDP, the goal of an agent is to learn policy π that maximizes $V^{\pi}(s)$ for all states. The policy is called the optimal policy and is indicated by π^*. The maximum discounted cumulative reward is indicated by $V^*(s)$.

$$\pi^* \equiv \arg\max_{\pi} V^{\pi}(s), (\forall s)$$

$$V^{\pi^*}(s) \Rightarrow V^*(s)$$

Then, how do agents obtain the optimal policy? It is hard to obtain optimal policy π^* directly. It is because learning examples are not provided directly in the form of ordered pairs of $< s, a >$ but the learner receive rewards necessary for learning.

Thus agents need a method of learning the optimal policy through interaction with environment without knowing such a function. It is easier to learn a numeric evaluation function than learning a specific policy, and the numeric evaluation function to learn is V^*. The goal of an agent is to obtain rewards as much as possible. If it comes to know V^* for all states, it will change the state according to V^*, and as the change is made in sequence the agent will reach the optimal policy.

For certain $s \in S$, there can be optimal policy π^* as follows. The optimal action in state *s* is the action *a* that maximizes the sum of the immediate reward *r(s,a)* plus the value V^* of the immediate successor state, discounted by γ.

$$\pi^*(s) \equiv \arg\max_a [r(s,a) + \gamma \ V^*(\delta(s,a))]$$

Here *δ(s,a)* denotes the state resulting from applying action *a* to state *s*. Thus the agent can require the optimal the optimal policy by learning V^*, provided it has perfect knowledge of the immediate reward function *r* and the state transition function *δ*. Unfortunately, learning V^* is a useful way to learn the optimal policy only when the agent has perfect knowledge of *r* and *δ*.

Reinforcement learning is one of reliable methods of solving these types of problems, and one of non-model reinforcement learning methods, which do not need an environment model, is Q-Learning.

B. Q-Learning

Q-Learning[1] proposed by Watkins is a widely used reinforcement learning method, which is based on stochastic dynamic programming. Q-Learning is a type of TD-Learning that utilizes difference in suitability according to time change in learning. It produces satisfactory results because it learns Q-value indicating the suitability of actions without information on environment models. Q-value means the value of the optimal cumulative reward given when the agent chooses an action at a certain state. Thus if an agent has Q function, at any state it can find the action it should perform based on Q-value for each action.

A Q-Learning agent needs to select a series of states-actions for the convergence rate of Q-value that decides the optimal policy in the process of learning. The learning decision policy of the algorithm is decided by state-action value function Q that measures the long-term discounted reward for each state-action pair. The agent repeatedly experiences all state-action pairs in the state space and learns the value of reward given by the environment based on evaluation value Q. Here, the agent uses a lookup table to store the values of all state-action pairs. In reinforcement learning, the value function determines the value of each node in winning. Of course, the value function is formulated through trial and error. The basic concept of Q-Learning is expressed as follows.

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$

Here, $Q(s,a)$ means the discounted cumulative reward obtainable when starting from state s, applying a as the first action and following the optimal policy. If $Q(s,a)$ is obtained, the optimal policy can be found.

The gem of Q-Learning is that it can describe with a number the situation in which the agent is and the action that the agent can take. Values of Q function are stored in the memory in the form of a table, which is unreasonable for a problem with a large number of states and actions.

$$\pi^*(s) = \arg\max_a Q(s,a)$$

Thus, sometimes methods adopting neural networks [6] or clustering are used.

IV. *n*-Queen algorithm using Q-Learning

For its problem solving algorithm, the present study used Q-Learning algorithm, which is one of non-model reinforcement learning. A tree is created for all possible nodes in n-Queen problem, and the value function has a corresponding value for each node like a lookup table. The function sets the initial value of table, which is the result of the value function, at 0 and begins to simulate through play. The queen put on the first column selects a node at random irrespective of the value function and then selects one of promising nodes for the queen to be put on the next column. The process is repeated to find the solution. If a promising node is not found any more in the middle of process, exploration is stopped and the value of Q-table is updated. In this way, an event row composed of perception, action and reward experienced by an agent is called an episode. This means that interaction between an agent performing learning and the external environment finishes in a natural way. It has been already proved that the update of Q-value through a number of episodes brings convergence to the optimal solution [7].

Q-Learning algorithm updates the value function by reflecting the result of play in the process of play when it has reached a node from which it cannot proceed any more. It propagates the value of the value function for the last node visited to previous nodes to a certain degree (propagation constant). In this study, the immediate reward for promising nodes was 1 and the propagation constant (discount factor) was 0.9. In this way, reinforcement learning runs by itself, giving reward (1) for a necessary action and penalty (0) for an unnecessary action. Although this requires a learning process of simulating a large number of trials and errors, it is advantageous in that it can gradually improve performance.

In order to learn for each Q-value, Q-Learning agents maintain n tables ($n \times n$) for their Q-values, and agents that have finished Q-Learning become able to estimate the results of long-term actions. Although it does not rise in other types of learning, one of problems in reinforcement learning is the balance between exploration and exploitation. Exploration is selecting unknown states and actions to collect new information and exploitation is taking learned states and actions. Exploitation guarantees that all available states and actions are sufficiently explored to satisfy the rule of Q-Learning convergence, and exploration adopts the greedy policy [5].

It is necessary to exploit known states and actions in order to get reward but it is also important to explore unknown states and actions in order to select better actions in the future. The balance between exploration and exploitation one of very important matters in reinforcement learning and is determined by various factors. To solve the problem related to the balance between exploration and exploitation in reinforcement learning, this study maintained a separate table to store the number of times that a queen piece was put on. In *n*-Queen problem, the solution is sought for simultaneously with the occurrence of the episode

that n queen pieces are put on. If the episode ends before n queen pieces are put on, it is a path of failure. Thus the path is avoided in the next episode and, as the path is set at random, the revisiting of a node that has been visited can be avoided. Therefore, the problem can be solved by inducing exploration toward positions with a small value in the table that stores the number of times. Figure 1 below shows the algorithm explained above expressed in Java.

```
// Find the optimal position of a queen on the corresponding row
// If it is found, recall with the value of the next row with 'queens (the number of the next row)'
 public static void queens (int i)
{
   int j;    int jp = 0;    // Position of the row to move into
   for (j = 1; j <= n; j ++)    // Find the position on the next row to put a queen on
   {
      if (promising (i, j)) {     // Check if a queen can be put on the corresponding column
          if (jp == 0)    // In the first comparison, if the value in the memory
                              for the existing position is 0, change the base to the newly found row
              jp = j;
          if (PT [Ts][i][jp] > PT [Ts][i][j])    // Choose one with a smaller number of times of positioning
              jp = j;
      }
   }
  if (jp != 0)  {        // If a position to move into has been found or the last position has been reached,
     coln ++;          // increase the number of columns that have been processed
     col [coln] = jp;              // Position a queen
     PT [Ts][coln][jp] ++;      // Increase the number of times of positioning a queen
                                   for the corresponding row and column
     runcount1 ++;               // Increase the number of times of positioning a queen
     if (coln != n)              // If the column is last or the positioning of queens has not been finished
         queens (coln + 1);      // Position a queen on the next column
     else  {           // When the last position has been reached
        runcount5 ++;
        QTcount ();
     }
  }
  else       // When the last position has not been reached, calculate Q
  {
     QTcount ();
  }
}
```

Figure 1 *n*-Queen algorithm using Q-Learning

V. Experiments and evaluation

The criterion for evaluating performance in *n*-Queen problem is the number of state transitions to put n queen pieces. We performed Q-Learning using lookup tables for all available states. Each execution is composed of a sequence of trials. In the first trial of each execution, the agent is given a random position. After the trial, the value rewarded is propagated and updates values in Q-tables and the trial is continued. Each trial ends on reaching the n^{th} column. For example, if n=10, 22 times of episode and 143 times of state transition occur on the average.

The present study compared the number of state transitions in finding the optimal solution in three methods, which are general backtracking, backtracking using MonteCarlo technique and Q-Learning. According to the result, the third method found the solution with the least number of state transitions. The method is expected to improve performance significantly for n that requires a large amount of exploration. Table 1 shows the number of state transitions for each n when the three methods are used.

Table 1 Comparison of the number of state transitions

N	Backtracking-1	Backtracking-2 (MonteCarlo)	-learning
4	26	16	8
5	15	16	13
6	171	66	53
7	42	43	39
8	876	171	33
9	333	383	65
10	975	942	143
11	517	1,039	300
12	3,066	1,539	428
13	1,365	5,434	852
14	26,495	16,846	764
15	20,280	14,869	621
20	3,992,510	1,384,727	7,007

Table 2 shows values in Q-table produced when n=10. In the table, the value of Q(1,1) indicates the largest one among values in the Q-table of all states in the 2nd column that can be reached from (1,1). According to the result of analyzing the table, the position of the largest Q-values (colored cells) converges into the solution of the problem.

Table 2 Q-table produced when n=10

	1	2	3	4	5	6	7	8	9	10
1	4.095	4.686	4.095	4.095	4.686	5.695	6.126	5.695	5.695	4.686
2	5.217	4.686	5.217	5.217	5.695	5.217	5.217	3.439	4.095	5.217
3	4.686	4.686	4.686	4.095	4.686	4.095	3.439	5.217	4.686	0.000
4	3.439	4.686	4.095	4.095	4.095	3.439	4.095	4.095	4.095	4.095
5	3.439	3.439	3.439	1.900	2.710	0.000	3.439	2.710	4.095	0.000
6	1.900	1.000	3.439	2.710	1.900	1.000	1.000	2.710	2.710	2.710
7	0.000	1.000	1.000	0.000	1.900	2.710	1.900	1.900	0.000	1.900
8	1.000	1.000	1.000	1.900	1.000	0.000	1.000	0.000	0.000	0.000
9	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
10	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	Goal

VI. Conclusions

The present study proposed an algorithm based on reinforcement learning in order to solve n-Queen problem. According to the result of comparing it with existing algorithms, it reduced the number of state transitions in reaching the goal and, as a result, its convergence rate into the optimal solution was quite fast.

However, basic reinforcement learning like Q-Learning shows low performance for big problems because reward should be back-propagated to the original state at which the action was taken. In addition, the state space may expand excessively depending on the nature of problems. A large state space requires a large amount of exploration and only the Q-values of experienced state-action pairs are updated. This means that the speed of learning is low. Moreover, the biggest problem in ordinary reinforcement learning methods is that they cannot be applied as they are to complex problems that have a large state space. Expression using lookup tables is considered to be an inappropriate method. Therefore, it is necessary to make research on the use of a small action space for agents to learn as well as on the improvement of speed and efficient function approximation.

What is more, it is necessary to study methods of exploiting knowledge about given environment in the process of learning in order to learn models similar to the real world as well as various forms of reinforcement learning algorithms.

References

[1] C. J. Watkins, and P Dayan, “Technical note : Q-Learning,” Machine Learning, 8, pp. 279-292, 1992.

[2] J. A. Boyan, and A. W. Moore, “Generalization in reinforcement learning : Safely approximating the value function,” Advances in neural Information Processing Systems. Vol. 7, pp. 369-376, 1995.

[3] N. Ono, and K. Fukumoto, “Multi-agent Reinforcement learning : A Modular Approach,” Proceedings of the Second International Conference on Multi-Agent Systems, AAAI Press, pp. 252-258, 1996.

[4] R. E. Neapolitan and K. Naimipour, Foundations of Algorithms, 2nd Ed, Jones and Bartlett Publisher, 1998.

[5] R. S. Sutton and A. G. Barto, Reinforcement Learning : An Introduction. The MIT Press, 1998.

[6] S. Haykin, Neural Network, 2nd Ed, Prentice-Hall, 1999.

[7] T. M. Mitchell, Machine Learning, McGraw-Hill, 1997.

[8] A. Mccallum, K. Nigam, J. Rennie and K. Seymore, “Building domain-specific search engines with machine learning techniques,” In AAAI-99 Spring Symposium on Intelligent Agents in Cyberspace, pp. 135-141, 1999.

[9] B. T. Zang and Y. W. Seo, “Personalized Web-Document Filtering Using Reinforcement Learning,” Applied Artificial Intelligence, vol. 15, pp. 665-685, 2001.

[10] J. Rennie and A. McCallum, "Using Reinforcement Learning to Spider the Web Efficiently," In proceedings of the 16th International Conference on Machine Learning(ICML-99), pp. 335-343, 1999.

Soo-Yeon Lim

- Dept. of Electrical Engineering and Computer Science, Kyungpook National Univ., Bachelor(1988)
- Dept. of Computer Engineering, Kyungpook National Univ., M.S.(1991)
- Dept. of Computer Engineering, Kyungpook National Univ., Ph.D(2004)
- 2004~Present Researcher of Department of Computer Engineering, Kyungpook National Univ.
- Research Interests : Informational Retrieval, Ontology, Machine Learning, Natural Language Processing
- E-mail : nadalsy@sejong.knu.ac.kr

Ki-Jun Lim
- Dept. of Computer Engineering, Sangju National Univ., Bachelor(2000)
- Dept. of Computer Engineering, Kyungpook National Univ., M.S.(2003)
- 2003~Present Ph.D candidate of Department of Computer Engineering, Kyungpook National Univ.
- Research Interests : Natural Language Processing, Machine Learning, Informational Retrieval
- E-mail : kjson@sejong.knu.ac.kr