Journal of Korea Game Society 2016 Apr; 16(2): 27-40
http://dx.doi.org/10.7583/JKGS.2016.16.2.27

게임 기획 및 디자인

Stealthy Behavior Simulations Based on Cognitive Data

Taeyeong Choi*, Hyeon-Suk Na**
School of Computing, Informatics, Decision Systems Engineering, Arizona State University*
School of Computer Science and Engineering, Soongsil University**

인지 데이터 기반의 스텔스 행동 시뮬레이션

최태영*, 나현숙**
애리조나주립대학교 컴퓨터공학과*, 숭실대학교 컴퓨터학부**
taeyeong.choi@asu.edu, hsnaa@ssu.ac.kr

ABSTRACT

Predicting stealthy behaviors plays an important role in designing stealth games. It is, however, difficult to automate this task because human players interact with dynamic environments in real time. In this paper, we present a reinforcement learning (RL) method for simulating stealthy movements in dynamic environments, in which an integrated model of Q-learning with Artificial Neural Networks (ANN) is exploited as an action classifier. Experiment results show that our simulation agent responds sensitively to dynamic situations and thus is useful for game level designer to determine various parameters for game.

요 약

스텔스 게임에서 플레이어의 행동을 예측하는 것은 게임 디자인에 있어서 핵심적인 역할을 한다. 하지만, 플레이어와 게임 환경 간의 상호작용이 실시간으로 일어난다는 점에서 이러한 예측 프로세스를 자동화하는 것은 어려운 문제이다. 본 논문은 동적 환경에서의 스텔스 움직임을 예측하기 위한 강화학습 방법을 소개하며, 이를 위해 Q-learning과 인공신경망이 통합된 형태의 모델이 액션 시뮬레이션을 위한 분류기로 활용된다. 실험 결과들은 이러한 시뮬레이션 에이전트가 동적으로 변하는 주변 상황에 민감하게 반응함을 보여주며, 따라서 게임 레벨 디자이너가 다양한 게임 요소들을 결정하는데 유용함을 보여준다.

Keywords : Reinforcement learning(강화학습); Artificial neural network(인공신경망); Game level design(게임 레벨 디자인); Game simulation(게임 시뮬레이션).

Received: Mar, 3, 2016 Revised : Apr, 8, 2016
Accepted: Apr, 15, 2016
Corresponding Author: Hyeon-Suk Na (Soongsil University)
E-mail: hsnaa@ssu.ac.kr

ISSN: 1598-4540 / eISSN: 2287-8211

1. Introduction

In stealth games such as the Metal Gear Solid series (Konami Digital Entertainment) or the Stealth Hunter series (RAC 7), players must traverse a 2D or 3D map undetected by any Field of View (FoV) of Non-Player Characters (NPC) such as cameras and mobile enemies. It is not a trivial task for game level designers to set an interaction scenario and verify its level of difficulty. This is mainly because the player behaviors would be affected in real time by various factors in game environments such as the geographical feature of a map, the number of enemies, the enemy speed, the enemy path patterns, the locations of starting and target points, and so on. Developing tools that assist game designers with interactive exploration of how players become affected when these factors are modified, has attracted researchers' interests recently.

By empirical studies on simulating the trajectory of players in stealth game, we recognized the need for a virtual agent that can sense objects within a limited range and take an appropriate reaction to the cognitive information. This is because, in many stealth games, the users are required to choose actions referring to what they see from the monitor with fixed number of pixels, thus only a virtual agent with the same restricted sensing ability can simulate properly human player's decisions in dynamic environments.

In this study, we investigate adopting reinforcement learning (RL) with Artificial Neural Networks (ANN) to stealthy behavior simulations. RL has been known to be an advantageous technique for activating autonomous mobile agents since a prior knowledge of the environment is not required for training. Also, RL is feasible for simulating human behaviors because the trained agents are essentially designed to behave optimally to achieve given goals and thus resemble human behaviors following *the principle of rationality*: accomplishing the goals as optimally as possible given a state, taking into account the experiences and beliefs[1,2].

We design and implement an action classifier trained by the relation between players' action selections and sensed information. By experimental results, we show that our agent simulates well real player's sneaky behaviors by making sensible decisions in response to dynamic situations, and is more useful for simulating the player's movement than a Rapidly-exploring Random Tree (RRT). To our best knowledge, this is the first to use RL in stealth game level design for understanding players' action selections given sensed data.

2. Background and Related Work

Stealthy behavior of a real game player is not easy to simulate with a virtual agent. For instance, if a player encounters an enemy on a corridor with an alcove, he will temporarily hide himself in the alcove, come back to the point after the enemy leaves, and continue the way he was on. One way to simulate such behaviors is to discretize the configuration space including the time parameter and to find the corresponding path from the initial state to

the final state. Some researchers have studied such path finding problems in the context of stealthy game design and analysis.

2.1 Simulating stealthy movements

Shi and Crawfis designed a tool for investigating placement of obstacles and static enemies[3]. They proposed the so-called *damage function*, given a player location and distribution of enemies, measuring the probabilistic damage that the player will receive. Discretizing the domain into graphs with edge weights being average damage and using Dijkstra's algorithm, they sought for placements of obstacles and enemies that result in the minimum damage path, the longest path, and the largest standard deviation path.

Tremblay et al.[4] developed an exploration tool for game level design, integrated into Unity 3D. Their tool allows the game designer for interactive exploration of how a player can be affected as game parameters are modified. It visualizes successful movements of a player constructed by the RRT algorithm, a probabilistic incremental sample-based path-finding algorithm. Based on this tool, some researchers performed more thorough studies in designing and analyzing game levels recently: Tremblay et al.[5] studied three different metrics for measuring risks of stealthy paths, Xu et al.[6,7] developed procedural methods for obtaining enemy's patrol routes and camera placement, and the second and third authors suggested interesting models for simulations involving combats and health packs[8].

2.2 Robot trained by RL and ANNs

Bing-Qiang Huang et al.[9] applied the so-called *reinforcement learning neural network* (RLNN) to the problem that a well-known miniature robot "Khepera" avoids obstacles. This robot has a cylinder-like shape and moves with two wheels in five ways: forward, *right forward, left forward, right rotation and left rotation.* It has eight infrared sensors on the body that can measure the distance from obstacles within distance 0.8 meters. The goal of the robot is to wander any environment without collision with obstacles. Even though the robot agent started to move in absence of any prior knowledge about obstacle positions, it could move in the environment without any collision after a long time learning.

The most relevant work to our concerns was studied by Humphrys[10]. The author implemented a house robot that can only detect objects in a small radius around it, carrying out multiple parallel missions without colliding with any obstacle in a house. The missions are to pick up dirt, to put out fire if exists, to go seeing a visitor, to identify if he is a stranger or not, and so on. Such events happen randomly and even simultaneously, so the agent needs to develop a reasonable system of action selection priorities. For this, the author implemented neural networks with 57 input units to indicate the objects around the robot, and trained them by the following reinforcement learning strategy: a reward value is defined for each step from state x to state y, these values are added to or subtracted from each other to define reward values for more complex situations, and by training with

the final reward function the robot can find what to do first among all possible actions. For instance, according to their reward policy, if a stranger exists but is unseen by the robot, then every step 0.1 points are subtracted from the initial reward, so the robot is forced to search for the stranger to stop getting the continuous punishment. However, if the robot is being recharged at plug, then the robot might choose to defer the search because recharging gives 0.1 points every step and thus making a counterbalance to the punishment. The author showed that with reward values carefully designed, a robot can build a set of reasonable action priorities and achieve several goals.

2.3 Game agent trained by RL

The integration of RL and ANNs has been applied to other computer games in the context of developing game controllers since the reward-punishment in learning process resembles the basic rule of games, and the ANN allows to treat multi-dimensional environments as inputs. Togelius et al.[11] investigated RL Multi-Layer Perceptron (MLP) and Simple Recurrent Network (SRN) with the HyperGP algorithm[12] for controlling the main character, Mario, in Super Mario Bros. In the game, the score is computed based on the numbers of collected coins and killed enemies, and the time taken until the level is cleared. Four types of information were given as the input; the two types indicate if Mario is being on the ground or in the air and the other two if any obstacle or enemy exists around Mario. While the first two cost two binary bits, one for each, the last two types need 18, 50, or 98 bits for the benchmark, which led to 21, 53, and 101 inputs in total including one input as a bias. The output of the network was of 5 bits, indicating Mario's action selection among *walk left, walk right*, *run*, *jump*, and *shoot a fireball*. The author reported a trouble that the controller trained in a higher level did not guarantee the success in lower testing levels.

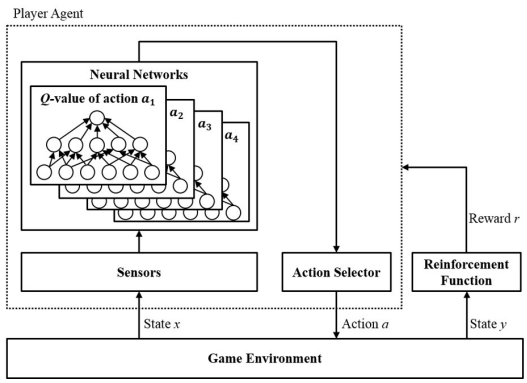
Mnih et al.[13] introduced Deep Q-Networks (DQN) for training agent controllers with the cognitive data obtained from the game screen. DQN was tested in seven of Atari 2600 games implemented in the Arcade Learning Environment[14]. Input pixel data are preprocessed into a 84×84 image in gray-scale, and four contiguous such images are given as an input into a neural network to express the situation-change of the environment. After a three-layered processing through the network, predicted Q-values are computed for all valid actions, and referring to them, the action with the highest output is selected.

In this paper, we encode the sensory data in a similar way, but handle a more complex game level even with a simpler learning model. Our autonomous agent can learn not only to accomplish the goal of stealthy games, arriving at the target point and not colliding against obstacles or any detection by the dynamic FoV of enemies, but also to simulate such human player's stealthy behaviors based on cognitive information as hiding in an alcove when an enemy is coming.

3. Learning Model

In this section, we describe our player-agent

model for simulating stealthy behaviors in dynamic game environments. The player in stealth games does not completely know the environment, but has to decide the next movement only based on what he sees. Reinforcement learning admits our player-agent to learn online without a complete knowledge of the environment. Using three layered neural network makes it possible to classify a large number of states (input sensory data), which seems inevitable for obstacle avoidance problem in dynamic environment, into only four selected actions (output data). In the following, we explain further technical details and advantages of this model.



[Fig. 1] Structure of the proposed learning System

3.1 Basics of RL and ANNs

RL is a learning technique to determine the best action to be taken in a state[15]. As shown in [Fig.1], the agent gets sensing data by its sensors, chooses an action by a reasoning process, and receives a feedback as a reward value. The reward value would be positive if the chosen action was considered good, but otherwise negative. Repeating this process, the agent increasingly learns which action would maximize the accumulated reward value and thus is the best choice given a state.

Q-learning is a method of RL, introduced by Watkins[16], for learning the so-called *Q-value* for each state-action pair (x, a) as follows. For each time t, an agent observes state x, takes action a, observes new state y, receives immediate reward r_{t+1}, and updates the current estimate Q-value. This process is often formulated as:

$$Q_{t+1}(x,a) \leftarrow Q_t(x,a) + \alpha[r_{t+1} + \gamma \max_{b \in A} Q_{t+1}(y,b) - Q_t(x,a)] \quad (1)$$

where α is the learning rate and $0 \leq \gamma \leq 1$ is the discount factor representing the importance of future rewards, in the sense that as it approaches 1, the learning has more emphasis on long-term rewards than myopic rewards. The optimal Q-value function is unknown and the system only estimates it. It was shown in [16] that, the more interactions with the world occur, the faster the noise (the term multiplied with the learning rate α in Eq.(1)) is eliminated, and thus the faster the estimate converges to the optimal value.

Look-up tables are usually used to store Q-values of state-actions pairs, but we use three-layer artificial neural networks with the back-propagation method[17]. They provide us with a more compact representation by interpolating many unvisited state-action pairs, and require much less spaces for a huge number of pairs of sensing inputs and actions. By applying the QCON model introduced by Lin[18], we build 4 independent networks, one per action. Breaking interference between

different output flows makes it easier to calculate the $\max_{b \in A} Q_{t+1}(y,b)$ term – just to enumerate the actions and obtain the value from each (output) network[10]. For each network, the output unit and the hidden units are designed to produce normalized Q-values between 0 and 1 by using the sigmoid function.

3.2 Action Selection

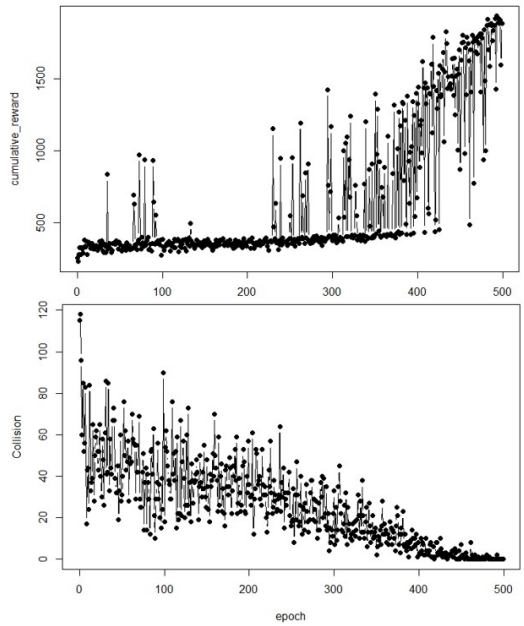
In the beginning phase of the learning process, the simulated virtual player agent must try to take random actions for exploration, albeit it causes many collisions and low rewards. This is because the agent needs to obtain knowledge of the environments through the interactions. As the learning proceeds, the action selection must gradually become greedier in selecting actions, which means that an action with higher Q-value is chosen with a higher probability. A common and easy way for controlling the system like this is to let the system choose an action according to the Boltzmann probability distribution, defined as Eq.(2)[19].

$$P(a \mid x) = \frac{e^{Q(x,a)/T}}{\Sigma_{b \in A} e^{Q(x,b)/T}} \quad (2)$$

To be specific, the higher the parameter T is, the more randomly an action is selected. As T converges to 0, the action selection follows the greedy policy with higher probability. After the learning finishes, we use the greedy action selection to maximize the accumulated rewards. In other words, an action at state x is determined by the following criteria.

$$a(x) = \arg\max_{b \in A} Q(x,b) \quad (3)$$

An example of such a learning approach is shown in [Fig. 2].



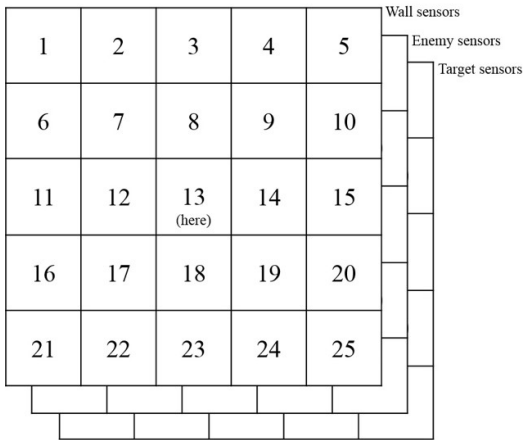
[Fig. 2] In the beginning of the training, the agent randomly selects actions for exploration, which causes low rewards and many collisions. The more the agent learns, the higher probability is set to optimal action selection. After learning, the agent maximizes the cumulative rewards and minimizes the number of collisions by taking the best actions.

3.3 Sensory Information

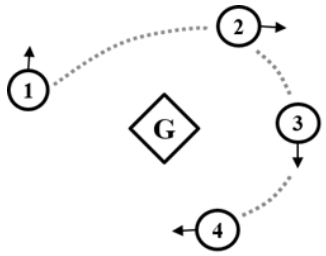
For successful simulation of the player who moves towards the target point and effectively responds to sensed data, we use five types of sensors: *wall sensors, enemy sensors, target sensors, target direction sensors, and player direction sensors*. The first tree types are designed for expressing what the player agent perceives, and the other types for representing the player's relative location to the target point. A 5-tuple of their values represents for a state as an input of our ANNs.

What the virtual player sees is expressed by

78 sensors in total, consisting of 26 *wall sensors* for detecting obstacles, 26 *enemy sensors* for the FoVs of enemies, and 26 target sensors for the target point. In the case of the *wall sensors*, each of the first 25 units indicates 1 if a wall exists in the corresponding block shown in [Fig. 3], and 0 otherwise, and the last is set to be 1 if the first 25 values are all 0, and 0 otherwise, which makes it easier for the neural networks to identify and classify the inputs[20]. The *enemy sensors* and the *target sensors* work in the same way as the wall sensors, except that they are respectively related to the FoV of any enemy and the target point. Such a modeling enables the agent to move freely without any collision and being detected, as well as to travel to the target point when the target lies in the field of vision.



[Fig. 3] Three types of sensors to monitor 25 points around the player agent



Loc.	Target Direction		Player Direction
	X	Y	
1	010	100	1000
2	100	100	0100
3	100	001	0010
4	100	010	0001

[Fig. 4] Illustration of 10 direction sensors corresponding to a trajectory of the virtual player

We use two more types of sensors, called *target direction sensors* and *player direction sensors*. The *target direction sensors* are made up of 6 units in total including three *target x-direction sensors* and three *target y-direction sensors*. As illustrated in [Fig. 4], the first unit of the *target x-direction sensors* is set to be 1 if the player's x-coordinate is higher than the target's x-coordinate, the second if the player's x-coordinate is lower than the target's, and the last if the player's x-coordinate is equal to that of the target. The three *target y-direction sensors* work in a similar way as the *target x-direction sensors*. The *target direction sensors* are, however, not enough to lead the player to the target, because they just divide a map into nine areas and inform the player which area the target is located in, according to the relative position of the player. To determine whether to turn right or left, or to advance forward or stay, the player needs knowledge of its current view direction. Thus, we use 4

binary units for indicating the current view direction: *EAST*, *WEST*, *SOUTH*, and *NORTH*. The first sensor is set to be 1 if the player is facing east, and 0 otherwise. The other sensors work in the same way.

3.4 Reinforcement learning

In stealth game, the virtual player needs to accomplish multiple goals, avoiding obstacles and arriving at the target spot. So we compute reinforcement values by two steps, as shown in [Fig. 5].

Reinforcement function(state x, state y)

v := 0.
If (*arrived at the target*) add 1 to v.
Else if (*collision occurred*) subtract 1 from v.
Else
  If (*chose FORWARD*) add 0.02 to v.
  Else subtract 0.015 from v.
  If (*being closer to the goal*) add 0.6 to v.
  Else add 0.3 to v.
Return v.

[Fig. 5] Reinforcement function, where x stands for the current state and y for the next state. Depending on y and the action taken, the return value v is decided.

First, the player gets +1 for arriving at the target and -1 for any kind of collisions. Otherwise, according to the selected action, the player gets a positive reward of 0.02 for *FORWARD* and a negative reward of -0.015 for *STAY*, *RIGHT-TURN*, and *LEFT-TURN*. Next, if the player moves closer to the target point with respect to the Euclidean distance, the value 0.6 is added to the current value, and otherwise 0.3 is added. Hence, except for the cases where the player arrives at the target point or collides with a wall or FoV of any guard, the player is rewarded either 0.62 or 0.32 by taking *FORWARD,* in the respective cases when the player moves in the direction to the target point or in the other directions. By the other three movements, it gets 0.285 reward value. As will be shown in the next section, the reinforcement values computed in this way motivate the agent to achieve the expected goals.

4. Experiments & Results

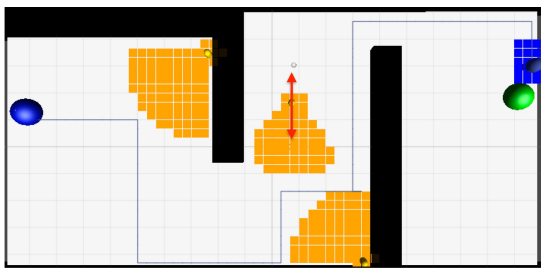
In this section, we demonstrate the effectiveness of our model by some experiments with the tool of [4]. The first experiment is to check if our agent is able to safely traverse a map with multiple cameras and one moving guard and to reach the target point. The next one is to check if our agent can react sensitively to dynamic environments, and the final is to compare the paths traversed by our agent and by the RRT algorithm proposed in [4].

In all experiments, we let the virtual player start to move from the starting point, monitor its visual field (shown as a blue area), and take one among the four actions for each step: *STAY*, *FORWARD*, *RIGHT-TURN*, and *LEFT-TURN*. During the journey in the learning phase, the agent is rewarded a reinforcement value for the selected action and the effect, which trains the four ANNs. If the player collides with any wall or enemies' view, the worst punishment is given. In this case, we set the player to continue the learning process from this failure spot unless the

agent fails 50 times consecutively. This is because such a way showed better learning performance than restarting the agent from the starting point as soon as a failure occurs. Also, in order to shorten the computation time for a solution path, we set the player agent to check each time if it is currently located within a predetermined area around the target point. If so, it means a success of the level.

Each test was performed after the agent had been trained in 6 different environments with the learning rate α of 0.5 and the discount factor γ of 0.6. The maximum value of time samples was set to be between 500 and 2,000, and the value of learning epochs between 500 and 1,000, depending on the size of the map.

4.1 Basic stealthy movements

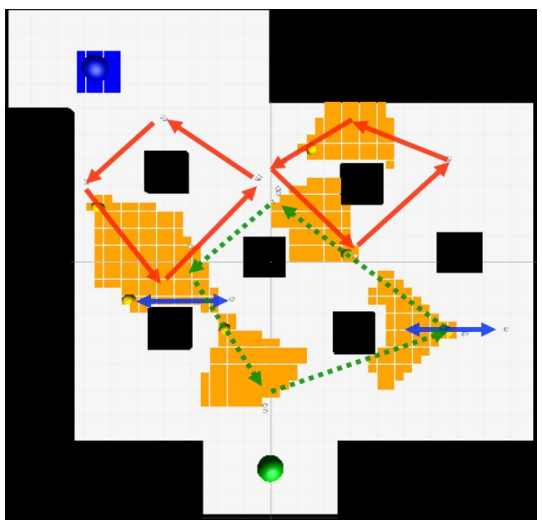


[Fig. 6] Test of capability of basic stealthy movements. With a limited perception area (blue area), the player agent moved along the gray path to avoid the views of two static cameras (orange area) and one patrolling guard, and finally succeeded in approaching the target point (green ball).

We let the virtual player move interacting with a dynamic environment shown in [Fig. 6], where obstacles hinder the player from straightly moving from the starting point (blue ball) to the target point (green ball). Two cameras (yellow dots) are on the watch for any infiltration and one mobile guard is patrolling along the red path. Here the only possibility for the player to win this level is to move along the vertical corridor using some maneuvers to avoid the detection (orange areas) of the multiple cameras and the enemy.

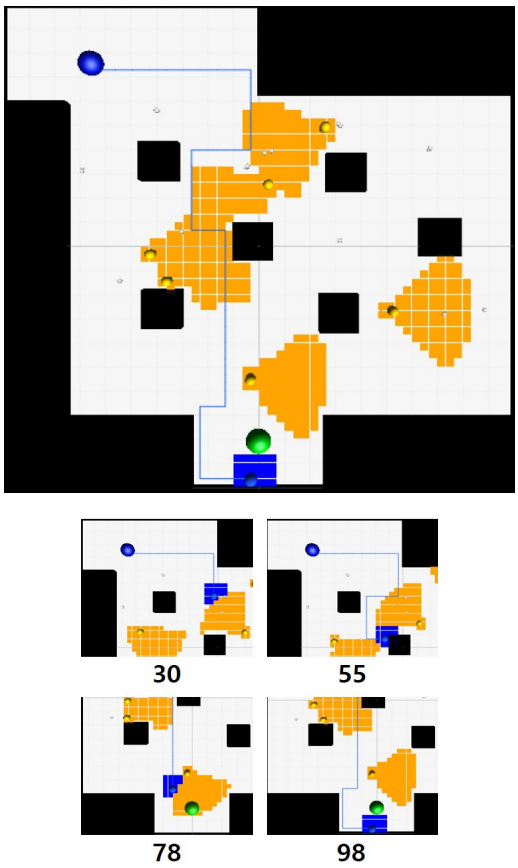
In the simulation, the vitual player secured a safe path (colored with blue). Specifically, the virtual player easily passed the first two cameras referring to the sensed data, but soon encountered an mobile enemy. By perceiving the enemy's access, the agent decided to turn right for avoidance and met the situation that walls were in its way. By turing left and moving beside the wall, the agent reached the target point successfully.

4.2 Sensitive movements



[Fig. 7] Test environment for sensitivity of our agent: there are six square-shaped walls at the center, around which six mobile guards are patrolling. Each of the four guards has its own path (red or blue arrows), and the other two patrol along a path (green dotted arrows) at a distance.

The main reason of using ANNs is that they enable us to deal with a large space of inputs and consequently allow the agent to sense changing situations efficiently. So it is important to check if the player reacts on change of environments in real time. On this purpose, we set a more complex and dynamic environment [Fig. 7], in which six guards patrolling around six square-shaped obstacles. In addition, the player needs to move from the upper-left corner to the target point in the bottom-middle sinkage.

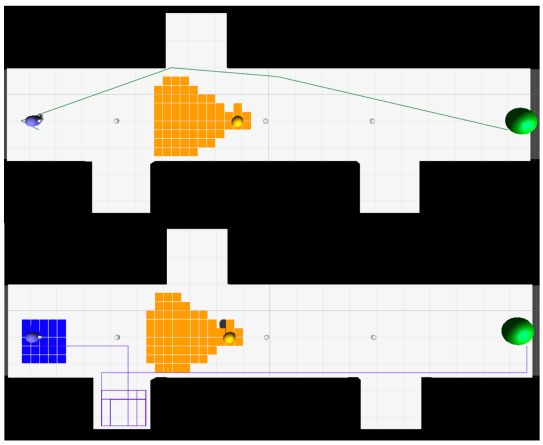


[Fig. 8] Successful trajectory (upper) from the starting point (blue ball) to the target point (green ball). Interesting reactions of our agent to dynamic situations marked with time (lower).

As shown in [Fig. 8], the agent safely arrived at the target point by the following interesting sequence of actions: The agent started to move heading east, soon encountered a wall, so it turned right. At time 30, it had to turn right again not to collide with the FoV of an enemy going upward. It also determined to turn left two times to avoid another wall and another FoV. At time 55, the agent seemed to be almost caught, surrounded by a wall and enemy's view, but it was able to find a path downward to escape as well as access to the target spot. Even though another enemy nearby the target point blocked the agent's way, the agent used a maneuver and reached the target point.

4.3 Comparison with a RRT-based agent

In the map shown in [Fig. 9], an enemy is patrolling in the middle of the hall and the player has to move from the starting point in the left side to the target point in the right side. This kind of map may offer game users great fun, because there are some alcoves to be used for the users to temporarily hide themselves just before encountering the enemy. Also, such an environment configuration would provide the game users with an opportunity to challenge diverse strategies. Yet, when we simulated player behaviors by two different methods, RRT and RL, the obtained routes were significantly dissimilar.



[Fig. 9] Dissimilar traces gained by a RRT-based agent (upper) and our RL-based agent (lower), in the map where a guard is patrolling in the middle of a corridor, and there are some alcoves the player can hide in for avoidance.

The RRT-based agent implemented in [4] calculated the output path faster than our RL-based agent because the former does not require any learning phase of environments. However, the path that the RRT player derived was too simple to represent for a real player's path, as seen in [Fig. 9] (upper); it chose only three actions until reaching the goal. Such a path may be useful in simulating a game user who can see the entire game environment in all time spaces. However, it might not provide utility in simulating the game user who has limited sensing range and cannot foresee the future.

On the other hand, our agent's path looks more realistic, since a real player who has limited sensing range and cannot foresee the future would likely hide in the first alcove when encountering the enemy, and then move towards the target when the situation becomes secure.

5. Conclusions

In this study, we implemented an integrated learning model of Q-learning and artificial neural networks that can be useful in stealthy behavior simulation. This model is useful since in many stealth games, the game user has to decide actions based on restricted visual information from the monitor, and thus virtual agent having such sensing limitation simulates better realistic player behaviors. Our experimental results showed that the agent reacts sensitively to the dynamic changes around it, and thus in some cases its behaviors are more realistic than those of a RRT-based agent.

We encoded the relative direction of the agent to the target point into ten binary units, consisting of six *target direction sensors* and four *player direction sensors*. For the same purpose, nine units may be enough, one unit indicating one relative location among nine relative locations of target point to the current agent position. We represented the game screen by 25 binary values and the action of the player by five actions. If one increases the size of the screen to be handled and diversifies possible actions including the combat with guards and the disguise as an environmental object for hiding, then more interesting realistic behaviors can be obtained from the simulated vitual agent. Since real users in most genres can usually choose an optimal action given a state, our learning model can be extended to simulate another type of games such as role-playing, first-person shooter, or racing genres. It could be done just by setting available actions in the game as the outputs

from the ANNs and adjusting the reward system according to the specified goals.

REFERENCES

[1] D.C. Dennett, The intentional stance. Cambridge, MA: MIT Press, 1987.

[2] G. Gergely, Z. Nadasdy, G. Csibra., and S. Biro., Taking the intentional stance at 12 months of age., Cognition, Vol. 56, pp. 165 - 193, 1995.

[3] Y. Shi and R. Crawfis, Optimal Cover Placement against Static Enemy Positions, in Proc. of the 8th International Conference on Foundations of Digital Games (FDG), pp. 109-116, 2013.

[4] J. Tremblay, P.A. Torres, N. Rikovitch, and C. Verbrugge, An Exploration Tool For Predicting Stealthy Behaviour, in Proc. of the 2013 AIIDE workshop on Artificial Intelligence in the Game Design Process, 2013.

[5] J. Tremblay, P.A. Torres, and C. Verbrugge, Measuring Risk in Stealth Games, in Proc. of the 9th International conference on foundations of Digital Games (FDG), 2014.

[6] Q. Xu, J. Tremblay, and C. Verbrugge, Generative Methods for Guard and Camera Placement in Stealth Games, in Proc. of the Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 2014.

[7] Q. Xu, J. Tremblay, and C. Verbrugge, Procedural Guard Placement for Stealth Games, in Proc. of the 5th workshop on Procedural Content Generation (PCG), 2014.

[8] J. Tremblay, P.A. Torres, and C. Verbrugge, An Algorithmic Approach to Analyzing Combat and Stealth Games, in Proc. of the International Conference on Computational Intelligence and Games (CIG), 2014.

[9] B.Q. Huang, G. Y. Cao, and M. Guo, Reinforcement Learning Neural Network to the Problem of Autonomous Mobile Robot Obstacle Avoidance, in Proc. of 2005 International Conference on Machine Learning and Cybernetics (ICMLC), 2005

[10] M. Humphrys, Action Selection Methods using Reinforcement Learning, in PhD Thesis, University of Cambridge, 1997.

[11] J. Togelius, S. Karakovskiy, J. Koutnk, and J. Schmidhuber, Super mario evolution, in Proc. of the IEEE Symposium on Computational Intelligence and Games (CIG), pp. 156-161, 2009.

[12] Z. Buk, J. Koutnik, and M. snorek, NEAT in HyperNEAT substituted with genetic programming, in Proc. of the International Conference on Adaptive and Natural Computing Algorithms (IICANNGA), 2009.

[13] V. Minih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, Playing Atari with Deep Reinforcement Learning, in Neural Information Processing Systems (NIPS) Deep Learning Workshop, 2013.

[14] M.G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, The arcade learning environment: An evaluation platform for general agents, in Jounal of Artificial Intelligence Research (JAIR), Vol. 47, pp. 253-279, 2013.

[15] M.J.L. Boada, R. Barber, and M.A. Salichs, Visual Approach Skill for a Mobile Robot using Learning and Fusion of Simple Skills,

in Robotics and Autonomous Systems, Vol. 38, pp. 157-170, 2002.

[16] C.J.C.H. Watkins and P. Dayan, Q-Learning, in Machine Learning, Vol. 8, pp. 279-292, 1992.

[17] A. Onat, Q-learning with recurrent neural networks as a controller for the inverted pendulum problem, in Proc. of the Fifth International Conference on Neural Inforamtion, pp. 837-840, 1998.

[18] L.J. Lin, Reinforcement Learning for Robots using Neural Networks, in PhD thesis, Carnegie Mellon University, School of Computer Science, 1993.

[19] E. Cervera and A.P.D. Pobil, Sensor-based Learning for Practical Planning of fine Motions in Robotics, in Information Sciences, Vol. 145, pp. 147-168, 2002.

[20] R. Gavin and N. Mahesan, On-line Q-learning using Connectionist systems, in Technical Report, No. 166, University of Cambridge Engineering Department, 1994.

최 태 영(Taeyeong Choi)

2015 숭실대학교 컴퓨터학부 공학사
2015- 애리조나주립대학교 컴퓨터과학 박사과정
관심분야: 기계학습, 스웜로보틱스, 로봇 플래닝

나 현 숙(Hyeon-Suk Na)

1993 서울대학교 수학과 이학사
1995-2002 포항공과대학교 수학과 이학 석·박사
2001-2003 프랑스 INRIA, HK UST Post Doc.
2003- 숭실대학교 IT대학 컴퓨터학부 교수
관심분야: 알고리즘, 계산기하학, 컴퓨터그래픽스