Modular Reinforcement Learning for Playing the Game of Tron

MINGI JEON1, JAY LEE2, AND SANG-KI KO.1
1Department of Computer Science and Engineering, Kangwon National University, Gangwon-do, 24341 South Korea (e-mail: {ckawoalt,sangkiko}@kangwon.ac.kr)
2Data & Investment Division, Hana Bank, Seoul, South Korea (e-mail: intelleej@gmail.com)
Corresponding author: Sang-Ki Ko (e-mail: sangkiko@kangwon.ac.kr).

ABSTRACT Tron is a simultaneous move two-player game where a wall is created along the path where two agents move and the agent that crash with the wall first is defeated. Due to the fact that the same action may result in different outcomes (non-stationarity), it is difficult to utilize the basic approach of reinforcement learning. In this paper, we present a modular reinforcement learning approach to tackling the game of Tron by decomposing the game into two phases where the first phase is non-stationary and the second phase is stationary. We evaluate the performance of our algorithm by comparing with previous algorithms including the state-of-the-art algorithm for the game of Tron (called a1k0n) in different grid sizes.

INDEX TERMS modular learning, reinforcement learning, Tron, non-stationary environment

I. INTRODUCTION
Since the advent of deep Q-networks (DQN) [1], the field of deep reinforcement learning (RL) has been developed rapidly through numerous approaches to improving the efficiency and stability of learning and the performance of the trained RL agents.

As the popularity of RL continues to grow, the target applications of RL also rapidly expanded from board games and classic video games with simple rules and actions to more complicated and actionable games [2], multi-agent games [3], [4], and NP-hard graph problems [5], [6].

Since RL researchers started to tackle more complex and difficult problems, there have been many attempts to breaking the complex and difficult tasks down into less complex and easier sub-tasks. This approach is called the modular reinforcement learning (MRL) [7], [8], based on a simple idea that it would be more efficient to decompose a task into several sub-tasks and train an agent for each sub-task.

Sun et al. [9] applied decomposition unit to the critic network for efficient learning of multi-agent games with multiple objects. In other words, sub-tasks were defined based on discrete objects. Simpkins et al. [7] proposed a method to improve learning performance with the task-specific reward scale for a single agent game with multiple goals defined. In this case, sub-tasks are defined based on the objectives that should be considered at the same time. Andreas et al. [10] applied a method to maximize the total reward by applying different policies for multiple sequential tasks. As such, there are various criteria and methods of presenting tasks in take composition and MRL-related studies.

In this paper, we consider the game of Tron, a two-player simultaneous move game played on a discrete square grid. Tron is a competitive game in which a wall is created along the path where two agents move simultaneously, and the agent that crash the wall first is defeated. It is a multi-agent game when sharing the same space with an opponent. However, after the space is separated from the opponent, the goal is to survive alone as long as possible because it cannot interfere with each other’s moves [11], [12].

There have been many strong heuristics proposed to play the game of Tron [11], [13]–[18] including the Monte-Carlo Tree Search (MCTS) algorithm and minimax algorithm. Recently, Knegt et al. [13], [15] utilized a deep neural network as Q function approximator. Perick et al. [16] applied the MCTS to the game of Tron and performed a comparison of different node selection strategies for the MCTS algorithm. In order to exploit the non-stationary property of the game of Tron, Knegt et al. [13], [15] proposed a concept of opponent modeling to predict the opponent’s next move which is subsequently used in MCTS.

We tackle the game of Tron by decomposing the game into two sequential sub-tasks. The first part of the game is to play until two agents are separated on the grid so that there is no possibility of colliding with each other. Since the next state...
of an agent cannot be determined only by the agent’s action, we consider these state to be non-stationary. The second part of the game is that the agent needs to move along the empty cells as long as possible. Since the agent at this point does not need to care about the opponent’s move, we can consider these states as stationary. In summary, we decompose the game of Tron into two phases where the first phase is inherently non-stationary as the opponent’s move can change the environment and reward regardless of the chosen action and the second phase can be considered stationary as we can simply ignore the opponent’s region from the grid. In this paper, we demonstrate that the modular RL approach to the game of Tron is robust compared to the previous approaches including the alkon algorithm [12] who won the Google Tron AI Challenge in 2010. Especially, we show that our algorithm is considerably faster than the previous heuristics as we do not perform the exhaustive tree search. Considering that Tron is a real-time game, our algorithm can be employed in time-sensitive scenarios. We verify the performance of our algorithm through comparisons with other algorithms on different size of boards.

II. BACKGROUND

A. REINFORCEMENT LEARNING

There are two well-recognized criteria for classifying the research directions of the RL. First, the RL algorithms can be classified into either model-free and model-based algorithms. The model-free RL algorithms (such as DQN [1], DDPG [19], etc.) try to estimate the optimal policy without using or estimating the transitions and reward functions of the environment. On the other hand, the model-based RL algorithms (such as AlphaZero [20], MuZero [21], etc.) employ the transition function and the reward function to estimate the optimal policy.

Second, we can also classify the RL algorithms into two classes, namely policy-based and value-based. In the policy-based RL, we directly learn a policy that maps states to actions (mapping $\pi: s \rightarrow a$) without using a value function. On the contrary, in the value-based RL, we only use a value function that estimates the values of the states and derive the policy from the value function by picking the action with the best value based on the value function.

The policy-based and value-based RL implicitly mixed in the field of RL but explicitly combined in the advantage actor-critic algorithms (such as A3C [22], A2C, AC-TKR [23], etc.). The actor-critic algorithm consists of two networks called the actor and the critic. The actor network (approximating the policy function) chooses an action at each step and the critic network (approximating the value function) evaluates the quality of the given state. As the critic network learns which state is better or worse, the actor relies on the critic to choose the states leading to better future rewards.

Reinforcement learning is a machine learning method that trains neural networks for decision-making processes based on the Markov Decision Process (MDP) [24]. In MDP, state transitions $t(s_{t+1}, a_t)$ occur when a particular action $a_t$ is selected according to policy $\pi(a_t|s_t)$ in a given state $s_t$, and agents receive rewards $R_t$. In order to know the exact sum of future rewards, we make choices at each time and add the total rewards. However, as the search space increases, it is practically impossible to find the correct sum for all possible situations. Therefore, we need an expression that can approximate the total future rewards with only the current information, which is called the $Q$-value. As the reinforcement learning is based on the Bellman equation [25], we aim to train an agent that selects an action that maximizes the $Q$-value defined as follows:

$$Q(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a')]$$

where $\gamma$ is the decay factor. Unlike present rewards that have a clear value, future rewards have unclear value. Therefore, when evaluating future rewards at the present time, a penalty is given by multiplying $\gamma$.

B. ACTOR-CRITIC ALGORITHMS

The Actor-Critic [22], [23], [26] algorithm employs two neural networks: an actor network that determines an action from a state, and a critic network that estimates a value of a state. The objective function for training the actor network is defined using the advantage function $A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$, which is the difference between the actual $Q$ value and the estimated value $V(s_t)$ by the critic network. Then, we optimize the actor network by maximizing the probability of actions with higher advantages. In other words, the objective function for the actor network is

$$\mathbb{E}[A(s_t, a_t) \nabla \pi \log \pi(a_t|s_t)]$$

On the other hand, we train the critic network by minimizing $A(s_t, a_t)^2$, the squared difference between the $Q$ value and the estimated value by the critic network. Finally, we add the following term called the entropy loss to the objective function to encourage exploration and discourage premature convergence to a sub-optimal policy:

$$-\sum \pi(a|s) \log \pi(a|s).$$
In order to increase the diversity of training data, the asynchronous advantage actor-critic (A3C) [22] algorithm executes a set of environments in parallel and the parallel agents update the global network asynchronously. The advantage actor-critic (A2C) is a synchronous version of A3C introduced by OpenAI in their published baselines. In A2C, all of the updates by the parallel agents are collected to update the global network. The actor-critic using Kronecker-factored trust region (ACKTR) [23] is an advanced method from A2C. The ACKTR improves sample efficiency by applying a Fisher information matrix which is mathematically identical to the derivative of Kullback-Leibler divergence in the optimization process of A2C. At this point, ACKTR uses approximations using the basic properties of Kronecker product instead of second-order derivation to reduce computational costs.

C. RL IN STATIONARY AND NON-STATIONARY ENVIRONMENTS

An environment is said to be stationary [9], [27] if the next state $s_{t+1}$ is determined based on the agent’s current action $a_t$ and the current state $s_t$. This is also known as the Markov property [24]. In general, most reinforcement learning algorithms assume a stationary environment where there is no stochastic element involved in determining the next state. See Figure ?? for pictorial description of the stationary and non-stationary environments.

On the contrary, an environment is called non-stationary if the next state $s_{t+1}$ cannot be determined solely based on the agent’s current action $a_t$ and the current state $s_t$. There exist additional factors $\xi_t$ that determine the next state such as other agents’ behavior, randomness, past behaviors and states, etc. It is widely known that basic approaches of reinforcement learning in non-stationary environments are very unstable.

D. MINIMAX ALGORITHM

Minimax [28] is a well-known recursive algorithm that selects the most favorable action through valuation within a game tree. Each node in the game tree has a selection value, which is computed by means of a position evaluation function. When an agent explores its action, the minimax algorithm selects a node with the highest selection value.

Similarly, the opponent is expected to make the most favorable choice from the opponent’s point of view. Therefore, when exploring the opponent’s action, the agent selects the node with the lowest selection value. As the computation cost of the minimax algorithm can be very expensive as the depth of search increases, we can improve the performance of the minimax algorithm by the use of alpha-beta pruning [29]. If an agent finds an action by the minimax algorithm from the game tree of depth $n$, we can ensure that the chosen action is the most favorable action for the agent after $n$ steps.

The Voronoi diagram [30] is used to partition a plane into regions close to each of a given set of points. Figure 2 left shows an example of a Voronoi diagram computed on a grid of Tron. Considering that an agent can only move in four directions in the game of Tron---up, down, left, and right---the Voronoi diagram of an agent represents the set of cells with the closest L1 distance for the agent. Cells with the same L1 distance from both agents [31] belong to neutral region.

We implement a minimax agent playing the game of Tron by relying on the computation of Voronoi diagram at each turn [11]. More specifically, the selection value of a node for agent 1 is defined as $R_1 - R_2$, where $R_i$ is the area of the Voronoi region of agent $i$. In other words, an agent always selects the node in the manner that the area of its Voronoi region is larger than the area of the opponent’s Voronoi region as much as possible. Similarly, the opponent always selects the node with the minimum selection value. If there are multiple nodes with the same maximum value, the agent randomly choose one of the nodes. In our implementation, we set the exploration depth of the minimax algorithm to two due to time constraints.

E. TREE OF CHAMBERS HEURISTIC: A1KON

However, the Voronoi heuristic has a clear limitation as in the situation depicted in Figure 2 right. While the area of the Voronoi region of the diamond-shaped agent is larger than that of the circle-shaped agent, the region will be divided into two regions if the diamond-shaped agent moves toward the wall and chooses one of the directions. If the circle-shaped agent blocks the selected area of diamond-shaped agent, the resulting remaining area of the circle-shaped agent will be larger. Therefore, if an agent can take one of two paths and only reach one of two areas as a result of the choice of the path, the area estimated by the Voronoi heuristic can be very different to the area the agent actually can reach. In these reasons, the a1k0n [12], the Google Tron AI Challenge champion, uses the biconnected component algorithm using the articulation points computed from the Voronoi region of the agent.

If two agents are separated, then the a1k0n algorithm relies on the simple wall-following heuristic. Note that the original a1k0n algorithm utilizes the iterative deepening search with runtime constraint but we re-implemented the tree search algorithm as the simple depth-limited search with depth one.
III. PROPOSED METHOD
Before explaining our methods more specifically, let us briefly recall how the game of Tron proceeds. The initial position of agent 1 is randomly determined and the initial point of agent 2 is automatically set to be a cell so that the initial grid possesses central symmetry. At each time, agents move one cell of grid up, down, left, and right direction simultaneously. The trajectories where agents have passed will be filled with walls, and the first agent that crash with any kind of wall will be defeated. If both agents crash at the same time (or crash with each other), the result of the game is a draw.

A. RL AGENT FOR STATIONARY ENVIRONMENTS
In Tron, a state that two agents are in an independent space and cannot influence each other is considered to be stationary environments [9], [27], as shown in Figure 1. After beginning the game from a non-stationary environment, the game may transition to a stationary environment if two agents are isolated on the grid. In the stationary environment, the agents only need to survive as long as possible. Note that the problem of finding the longest path in the grid is equivalent to the longest path problem in the grid graph [32], which is known as NP-complete. Therefore, we can assume that there is no polynomial time algorithm that finds the optimal path for agents in the stationary state if P ≠ NP.

Therefore, we aim at solving the NP-complete problem using a neural network as it is proven in many cases that the neural network can solve intractable problems effectively compared to previous successful heuristics [5], [6]. In order to train an agent playing in stationary states to find the longest survival path, we randomly generate stationary environments that the agent may encounter during real games. The algorithm for generating the random stationary environment is explained in Algorithm 1.

B. RL AGENT FOR NON-STATIONARY ENVIRONMENTS
We distinguish between the non-stationary and stationary environments and finish the game as soon as it is converted to a stationary environment by detecting crash of agents; then the win or lose is determined by the remaining longest path of each agent. It is a draw if two agents crash in a non-stationary environment at the same time, or if the remaining longest path is same in a stationary environment. The whole training procedure of the agent playing in non-stationary states is explained in Algorithm 2.

Therefore, the agent that plays non-stationary environments is trained with the goal of making the remaining longest path is longer than opponent agent’s. This agent named P agent.

C. REWARD FUNCTION
The reward function should be designed to guide agents’ behaviours in reinforcement learning. The most intuitive way to design a reward function in the grid world is to give a positive reward for each step so that the agent tries to survive as much as possible. However, we find it inappropriate for the game of Tron as the positive reward for each step results in an unexpected side effect [33]. Since we train agents through self-play training, the agent tries to avoid to get close to the opponent during the self-play games. As a result, the agent trained with positive reward for each step tends to survive more steps but exhibits poorer performance against the other agents compared to the agent trained with negative reward for each step. We notice that negative step reward results in a more aggressive agent playing better against the other agents on average while positive step reward results in a more defensive agent playing worse against the other agents.
this reason, we opt to use the negative reward for each step to training our neural-network agents in our experiments.

**D. NEURAL NETWORK ARCHITECTURE**

Our neural network consists of seven convolutional layers followed by two linear (affine transformation) layers with residual connections as shown in Figure 3. We use the same neural network for both non-stationary and stationary agents. Finally, a policy network and a value network consist of two and three linear layers, respectively, with Mish activation function [34] in between.

**FIGURE 3.** The proposed neural network architecture.

When the grid size is $6 \times 6$, the linear layer right after an average pooling layer has a size of 256, and 576 when the grid size is $8 \times 8$ or $10 \times 10$. The numbers in convolutional layers from Figure 3 are kernel size, number of input channels, and the number of output channels. Note that a stationary environment consists of two channels (due to the absence of the opponent) instead of three channels for a non-stationary environment. The padding size of all convolutional layers and pooling layers is $\lceil \text{kernel size}/2 \rceil$.

Matrices provided in Figure 4 show how we encode states of the game as matrices to feed the neural network. An input matrix for a non-stationary environment consists of three channels where the first channel encodes the locations of side walls, the second encodes current and previous (already filled with walls) locations of the current agent, and the last encodes the current and previous locations of the opponent. On the other hand, an input matrix for a stationary environment consists of only two channels as there is no need to encode any information about the opponent. In input matrices, a wall is represented by one, an empty cell by zero, and location occupied by an agent is ten.

**IV. EXPERIMENTS**

In this section, we first describe the experimental setup and present the experimental results and discussion for various experiments conducted to verify the proposed idea.

**A. IMPLEMENTATION DETAILS**

The experiments are performed on a computer with a 6-core Intel Core i7-8700K 3.70GHz CPU, 48GB RAM, and an NVIDIA GeForce GTX 1080 Ti GPU. We use Python 3.7.6, PyTorch 1.7.1, CUDA 11.1 running on Ubuntu 18.04.5 LTS.

**B. BASELINES**

We measure the performance of the proposed algorithm (named $P$ agent) against several baseline approaches including the state-of-the-art Tron algorithm.

1) A neural network agent based on backtracking algorithm (named $B$ agent): In stationary environment, the truth longest path is obtained by backtracking search. The non-stationary agent uses a neural network and unlike the $P$ agent, it is trained via backtracking search results. However, $8 \times 8$ and $10 \times 10$ games are almost impossible to use backtracking search due to exponential computational costs. So, this agent is only trained in $6 \times 6$ games with small state-space game.

2) A neural network agent based on greedy length approximation (named $G$ agent): The greedy algorithm is as follows: the agent prioritizes four directions and explore the possible directions as deep as possible according to the priority. A cell cannot be visited again once visited and when we reach a dead end during the search, we search from the last cell where there are still remaining directions to be searched. The length that the agent explored most deeply is considered to be the longest survival path for the agent. An agent playing in non-stationary states uses a neural network trained via greedy length approximation results.

3) A single neural network agent (named $S$ agent): This agent relies on a single neural network trained to play the entire game. Note that the neural network receives three-channel input instead of two-channel even in stationary states as there is no distinction between a stationary and a non-stationary state.

4) A minimax agent and DLS (depth-limited search) approximation (named $M$ agent): How the minimax agent plays in non-stationary environments is described in matrix for a non-stationary environment.
Section II-D. The depth of minimax search is fixed to two. In stationary environments, a length of the longest path is approximated by depth-limited search, which is maximizing remaining area. The depth of search is limited to one.

5) a1k0n agent and wall-following heuristic (named A agent): How the a1k0n agent plays in non-stationary environments is described in Section II-E. The depth of minimax search is fixed to two. In stationary environments, the longest path is approximated by DLS, that wall-following bonus added. If the maximum remaining area is the same for several possible directions, an agent chooses the direction with the most walls around it. The depth of search is limited to one.

6) Neural network agents based on the area of agent’s region (named RP agent and RG agent): The non-stationary agent uses a neural network trained via remaining region measurement without any path searching process. Therefore, this agent is trained with values greater than or equal to the actual possible longest path. This has no problem with training, but it is unfair with other agents during evaluation. Therefore, we use pretrained approximation (RP agent) and greedy approximation (RG agent) in evaluation.

C. EXPERIMENTAL METHODS

We measure how well six methods except for region approximation approximate the ratio of longest path to area in the game board of stationary environments generated by the method mentioned in 3.2. Backtracking method is measured only in 6 × 6 games due to exponential time of computation. Also, the same experiment is carried out in stationary environments generated during real plays. In this case, all games are self-plays for agents with the same approximation scheme (e.g., the pretrained approximation’s ratio of longest path to area is measured in P agents’ self-plays, and the greedy approximation’s is measured in G agents’ self-plays.). Each match is played 10,000 games, and the average is obtained by dividing the total area from the total longest path. The performance of each agent in non-stationary environments is measured in round-robin tournament [11], [16], except for the same agent-to-agent match. Each agent will play 1000 games per match, including match against B agent in 6 × 6 games.

| Agent vs | P vs | G vs | S vs | M vs | A vs | RP vs | RG vs |
|----------|------|------|------|------|------|-------|-------|
| P        | -254 | 244  | 472  | 285  | 515  | 483   | 545   |
| G        | 142  | -    | 275  | 501  | 293  | 483   | 545   |
| S        | 104  | 165  | -    | 417  | 208  | 393   | 370   |
| M        | 95   | 92   | 183  | -    | 141  | 420   | 464   |
| A        | 192  | 217  | 290  | 498  | -    | 525   | 535   |
| RP       | 52   | 105  | 110  | 218  | 108  | -     | 104   |
| RG       | 62   | 47   | 114  | 183  | 96   | 80    | -     |

TABLE 2. Results of the round-robin tournament in 8 × 8 grid.

| Agent vs | P vs | G vs | S vs | M vs | A vs | RP vs | RG vs |
|----------|------|------|------|------|------|-------|-------|
| P        | -    | 203  | 378  | 492  | 308  | 594   | 625   |
| G        | 196  | -    | 315  | 496  | 262  | 612   | 595   |
| S        | 222  | 203  | -    | 418  | 205  | 508   | 509   |
| M        | 134  | 196  | 265  | -    | 172  | 546   | 559   |
| A        | 287  | 314  | 379  | 576  | -    | 663   | 661   |
| RP       | 65   | 51   | 70   | 185  | 74   | -     | 197   |
| RG       | 49   | 63   | 81   | 195  | 62   | 174   | -     |
D. RESULTS

a: Performance of longest path approximation in stationary environments

In randomly generated stationary environments, the ratio of approximated longest path to area by agent is shown in Figure 5. First of all, we can see that the algorithm closest to the backtracking search results is wall-following heuristic. The pretrained approximation is also close to backtracking and we can see that neural networks approximates the longest path problem well enough, which is NP-hard problem. Interestingly, the greedy approximation, which has lower time complexity compared to other methods, showed an approximate ratio of 0.9 or higher on all board sizes. Other approximation methods shows relatively lower performances.

However, the stationary environments generated during the actual game play resulted in different outcomes. Wall-following approximation had a higher rate of filling regions than backtracking agents. This refers to the distribution difference between the stationary environments generated from A agents and the stationary environments generated from B agents. In other words, even though the stationary environments generated by the B agent is lower in longest path to area ratio compared to the A agent, the actual remaining area is wider and the remaining longest path is longer. Additionally, the single approximation results were much higher than those of Fig 6. In 8×8 games, the approximate ratio was higher than even the pretrained approximation, as the S agent was well trained on stationary environments generated by real games, but never learned about randomly generated stationary environments. From this experiment, we can see that the pretrained neural network trained through the random stationary environments generation algorithm shows relatively good performance. However, some degradation from distribution differences from stationary environments generated from real games remains room for improvement.

b: Results of the round-robin tournament

The results of the round-robin tournament with the proposed algorithm and the baseline algorithms are provided in Tables 1, 2, 3 and 4. Each agent plays against the other agents 1,000 times. The number of wins of each agent during the tournament is presented. Note that the number of draws can be obtained by subtracting the numbers of two agents’ wins from 1,000. The P agent recorded a higher number of wins than the number of losses for all agents in 6×6 and 8×8 games, and the A agent recorded a higher number of wins than the number of losses in 10×10 games. G agent achieved slightly lower performance than P and A agent. The RP and RG agent lost more times than winning in all matches. P agent was defeated by A agent in 10×10 games because they used almost the same model for the game board in all sizes. If the P agent have a deeper model structure and can be trained from the approximate results of the improved pretrained model, it is expected to be able to win against the A agent, which is a strong baseline.

c: Results against backtracking agent

We also compare the performances of considered agents against the backtracking agent which is trained with the ground truth longest survival path only on 6×6 grid due to its intrinsic complexity. The experimental results are shown in Table 5. The winning percentage is calculated without

### TABLE 3. Results of the round-robin tournament in 10 × 10 grid.

| Agent | P vs | G vs | S vs | M vs | A vs | RP vs | RG vs |
|-------|------|------|------|------|------|-------|-------|
| P     | -    | 299  | 349  | 504  | 598  | 584   |       |
| G     | 197  | -    | 281  | 482  | 228  | 544   | 553   |
| S     | 218  | 197  | -    | 386  | 148  | 479   | 507   |
| M     | 194  | 192  | 290  | -    | 131  | 675   | 666   |
| A     | 399  | 401  | 473  | 593  | -    | 752   | 761   |
| RP    | 52   | 34   | 37   | 133  | 39   | -     | 192   |
| RG    | 46   | 34   | 48   | 138  | 42   | 229   | -     |

### TABLE 4. Winning percentages of each agent from the round-robin tournament (without counting draws).

| Agent | On 6 × 6 | On 8 × 8 | On 10 × 10 |
|-------|----------|----------|------------|
| P     | 77.690%  | 73.178%  | 70.100%    |
| G     | 71.786%  | 70.622%  | 66.386%    |
| S     | 57.675%  | 58.120%  | 56.695%    |
| M     | 37.866%  | 44.214%  | 48.996%    |
| A     | 66.617%  | 72.672%  | 79.957%    |
| RP    | 22.390%  | 17.170%  | 12.938%    |
| RG    | 18.878%  | 16.552%  | 14.132%    |

### TABLE 5. Results of games against the backtracking agent in 6 × 6 grid.

| Agent | Wins / Draws / Losses | Winning Percentage |
|-------|------------------------|--------------------|
| M     | 99 / 742 / 159         | 38.372%            |
| A     | 120 / 641 / 239        | 33.426%            |
| S     | 120 / 586 / 294        | 33.426%            |
| M     | 110 / 374 / 516        | 28.956%            |
| A     | 167 / 503 / 330        | 33.602%            |
| RP    | 87 / 402 / 511         | 14.548%            |
| RG    | 73 / 394 / 533         | 12.046%            |
taking draws into account. The proposed P agent showed the highest winning percentage, and A agent showed higher winning percentages compared to other agents except A agent. G agent achieved slightly lower winning percentage than A agent. On the other hand, the RP and the RG agents showed the lowest winning percentage. This means that the non-stationary agent certainly plays better when trained with more accurate longest path approximation heuristic.

V. CONCLUSIONS

In this work, we propose a modular RL approach to solve the game of Tron especially by decomposing a game into the first non-stationary phase and the second stationary phase. We train two separate agents playing in the two phases to reduce the training complexity of the game. We find that the agent trained to play in a stationary state exhibits a competent approximation performance especially on larger grid sizes, the scalability of our algorithm can be a huge advantage when deployed in real-time game environment.

REFERENCES

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, “Human-level control through deep reinforcement learning,” Nature, vol. 518, no. 7540, pp. 529–533, 2015.
[2] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev et al., “Grandmaster level in starcraft ii using multi-agent reinforcement learning,” Nature, vol. 575, no. 7782, pp. 350–354, 2019.
[3] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, “Multi-agent actor-critic for mixed cooperative-competitive environments,” arXiv preprint arXiv:1706.02275, 2017.
[4] K. Zhang, Z. Yang, and T. Başar. “Multi-agent reinforcement learning: A selective overview of theories and algorithms,” arXiv preprint arXiv:1911.10635, 2019.
[5] K. Abe, Z. Xu, I. Sato, and M. Sugiyama, “Solving np-hard problems on graphs with extended alphago zero,” arXiv preprint arXiv:1905.11623, 2019.
[6] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, “Neural combinatorial optimization with reinforcement learning,” arXiv preprint arXiv:1611.09940, 2016.
[7] C. Simpkins and C. Isbell, “Composible modular reinforcement learning,” in Proceedings of the AAAI Conference on Artificial Intelligence, 2019, pp. 4975–4982.
[8] S. Bhat, C. L. Isbell, and M. Mateas, “On the difficulty of modular reinforcement learning for real-world partial programming,” in AAAI, 2006, pp. 318–323.
[9] C. Sun, W. Liu, and L. Dong, “Reinforcement learning with task decom- position for cooperative multiagent systems,” IEEE transactions on neural networks and learning systems, 2020.
[10] J. Andreas, D. Klein, and S. Levine, “Modular multitask reinforcement learning with policy sketches,” in International Conference on Machine Learning. PMLR, 2017, pp. 166–175.
[11] L. Kang, “Endgame detection in tron,” Maastricht University, Tech. Rep., 2012.
[12] A. Sloane, “Google A1 challenge post-mortem,” https://www.a1kon.net/201003/04/google-ai-postmortem.html, last modified August 2. 2011., accessed April 07. 2021.
[13] S. J. Knegt, M. M. Drugan, and M. A. Wiering, “Learning from monte carlo rollouts with opponent models for playing tron,” in International Conference on Agents and Artificial Intelligence. Springer, 2018, pp. 105–129.
[14] A. Shmakov, J. Lanier, S. McAleer, R. Achar, C. Lopes, and P. Baldi, “Colosseumrl: A framework for multigait reinforcement learning in n-player games,” arXiv preprint arXiv:1912.04451, 2019.
[15] S. J. Knegt, M. M. Drugan, and M. A. Wiering, “Opponent modelling in the game of tron using reinforcement learning,” in ICAR (2), 2018, pp. 29–40.
[16] P. Perick, D. L. St-Pierre, F. Maes, and D. Ernst, “Comparison of different selection strategies in monte-carlo tree search for the game of tron,” in 2012 IEEE Conference on Computational Intelligence and Games (CIG). IEEE, 2012, pp. 242–249.
[17] N. G. Den Teuling and M. H. Winands, “Monte-carlo tree search for the simultaneous move game tron,” Univ. Maastricht, Netherlands, Tech. Rep., 2011.
[18] S. Samothrakis, D. Robles, and S. M. Lucas, “A uct agent for tron: Initial investigations,” in Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games. IEEE, 2010, pp. 365–371.
JAY LEE was born in Chuncheon, Gangwon-do, South Korea in 1996. He received the B.S. degree in computer science and engineering from Kangwon National University, Chuncheon, South Korea, in 2021. From 2017 to 2019, he served in the military and was discharged from the military service as a sergeant. From 2021, he is currently working as a Data Analyst at AI Lab Unit of KEB Hana Bank, Seoul, South Korea. His research interest includes deep reinforcement learning for game AI and computational complexity of algorithmic problems.

SANG-KI KO received the B.S. and Ph.D. degree in computer science from Yonsei University, Seoul, South Korea, in 2010 and 2016, respectively. He is currently working as an Assistant Professor at Department of Computer Science and Engineering of Kangwon National University in South Korea. From 2016 to 2017, he was a Research Associate at University of Liverpool in the UK. From 2017 to 2019, he worked as a Senior Researcher at artificial intelligence research center of Korea Electronics Technology Institute in South Korea. His research interest includes both the theoretical and practical side of computer science. From the theoretical side, he worked on the problems about the descriptional complexity of regular languages and edit-distance computation between formal language instances. Recently, he is actively working on various applications of machine learning algorithms and deep neural networks such as program synthesis, time-series data analysis, game AI, and recommender systems.

MINGI JEON was born in Chuncheon, Gangwon-do, South Korea in 1996. He received the B.S. degree in computer science and engineering from Kangwon National University, Chuncheon, South Korea, in 2022. He is currently pursuing the M.S. degree in computer science and engineering at Kangwon National University, Chuncheon, South Korea. From 2017 to 2019, he served in the military and was discharged from the military service as a sergeant. His research interest lies in deep neural network and reinforcement learning for game AI.