IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 15 March 2022, accepted 20 April 2022, date of publication 16 May 2022, date of current version 20 June 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3175299

Modular Reinforcement Learning for Playing the Game of Tron

MINGI JEON[1], JAY LEE[2], AND SANG-KI KO[1]
[1]Department of Computer Science and Engineering, Kangwon National University, Chuncheon, Gangwon-do 24341, South Korea
[2]Data & Investment Division, Hana Bank, Seoul 04523, South Korea

Corresponding author: Sang-Ki Ko (sangkiko@kangwon.ac.kr)

This study was supported by 2021 Research Grant from Kangwon National University.

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 (MRL) 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 train two separate models where the first model deals with the non-stationary environments such that two models move simultaneously and affect each other while the second model deals with the stationary environment when two agents are separated by walls created and cannot affect each other. We show that the latter model can be effectively pre-trained using randomly generated stationary environments. 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. As a result, we demonstrate that the proposed algorithm based on MRL outperforms all previous algorithms on 6×6 and 8×8 grids. Although our algorithm shows slightly worse performance on 10×10 grid than the strongest baseline a1k0n, we show that our algorithm exhibits better scalability in terms of time complexity as the grid size increases than search-based heuristics including the a1k0n.

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. Andreas *et al.* [9] presented an MRL approach for multitask RL by decomposing a task into a sequence of subtasks (policy sketch) while subtasks are shared across multiple tasks. Mendez *et al.* [10] proposed to tackle MRL problems via neural compositional learning framework by training an agent to construct a different policy for every task by selecting proper modules from a set of available modules. They show that each selected module can be improved during the process of learning different tasks and eventually demonstrates better performance in the tasks.

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].

The associate editor coordinating the review of this manuscript and approving it for publication was Alba Amato.

© 2022 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

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 a1k0n 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. We claim that the proposed algorithm is novel compared to previous works on MRL as we employ the concept of pre-training the model for the stationary environment using randomly generated environments.

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.

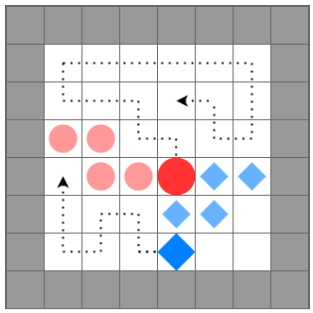

FIGURE 1. Two tron agents that started the game from a point-symmetric location on a 6 × 6 game board. A large circle and a large diamond are the current location of the agents. The longest path of the circle-shaped agent is 16, and the diamond-shaped agent's is 7.

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, ACTKR [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 $t(s_{(t+1)}|s_t, a_t)$ 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-valve defined as follows:

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

where γ 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 γ.

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

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

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:

$$-\sum^{a} \pi(a|s) \log \pi(a|s).$$

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* [27], [28] 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.

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 ε_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 [29] 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.

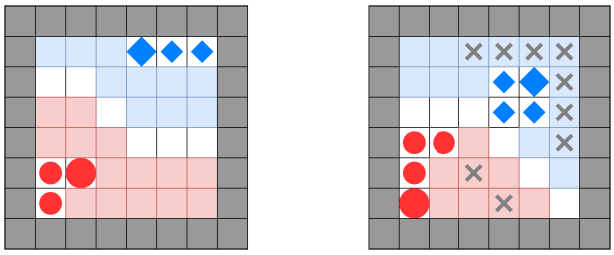

FIGURE 2. The Voronoi regions of two agents in the game of Tron (left). The area of the circle-shaped (red) agent's region is 14 and the area of the diamond-shaped (blue) agent's region is ten. Articulation points obtained from the Voronoi regions of two agents (right).

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 [30]. 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 [31] 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 [32] 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: a1k0n

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 [27], [28], 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 [33], 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 $\mathrm{P} \neq \mathrm{NP}$.

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 from a non-stationary environment.

Algorithm 1 Random Stationary Environment Generation

Input: An empty game map
Output: A random stationary environment (map)

```
Randomly select one of the four sides;
Place a cursor to one of the cells from the selected side;
while cursor is not on one of the three unchosen sides do
    Make the list of directions except for the direction where the cursor started from;
    if cursor moved in the same direction (grid width − 3) times in a row then
        Remove the direction from the list;
    end if
    Remove the opposite direction of the previous cursor move from the list;
    Randomly select one direction from the list;
    Move a cursor to selected direction and place a wall behind;
end while
Choose a side from two separated regions;
Fill walls to the unchosen side;
return A randomly generated map
```

Algorithm 2 Training Agent for Non-Stationary Environment

```
Pretrain a stationary agent on randomly generated stationary environments;
Initialize a non-stationary agent;
while training is not finished do
    Start self-play of same neural network agents in the non-stationary environment;
    if game is changed to the stationary environment then
        Compute the approximate length of the longest path using the pretrained agent;
        Determine the result of the game by comparing the approximate lengths of two agents;
    else
        Determine the result of the game in the non-stationary environment by detecting crash of agents;
    end if
    Give reward to the non-stationary agent based on the result;
end while
```

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 [34]. 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. For 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 [35] in between.

When the grid size is 6×6, the linear layer right after an average pooling layer has a size of 256, and 576 when the grid size is 8×8 or 10×10. 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 $\lfloor kernel\ size/2 \rfloor$.

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.

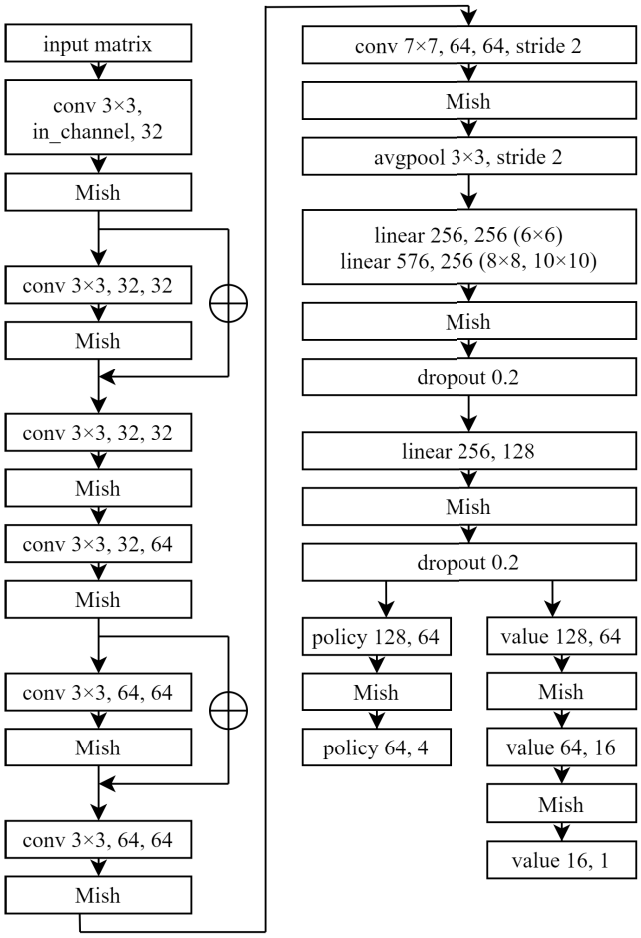

FIGURE 3. The proposed neural network architecture.

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, 8×8 and 10×10 games are almost impossible to use backtracking search due to exponential computational costs. So, this agent is only trained in 6×6 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

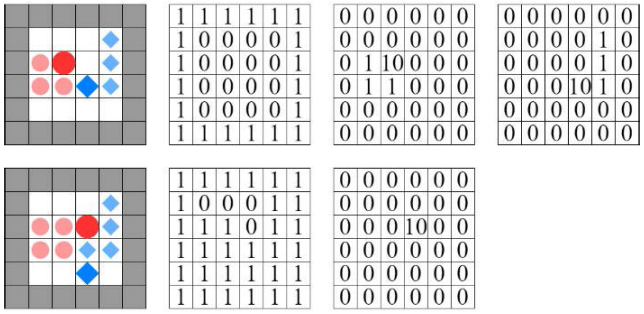

FIGURE 4. Encoding of a non-stationary input state (above) and a stationary input state (below) from circle-shaped (red) agent's point of view.

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 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 choose 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.

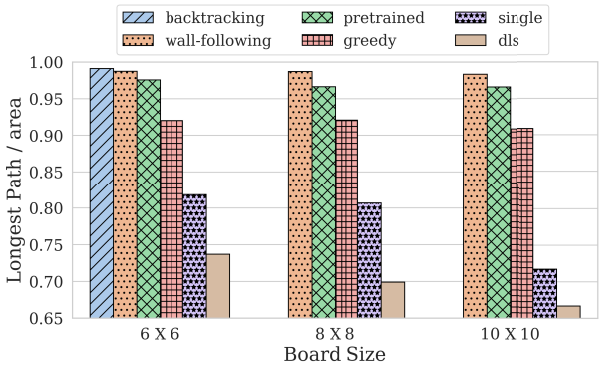

FIGURE 5. The ratio of longest path to width of each agent in randomly generated stationary environments.

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.

D. RESULTS

1) 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

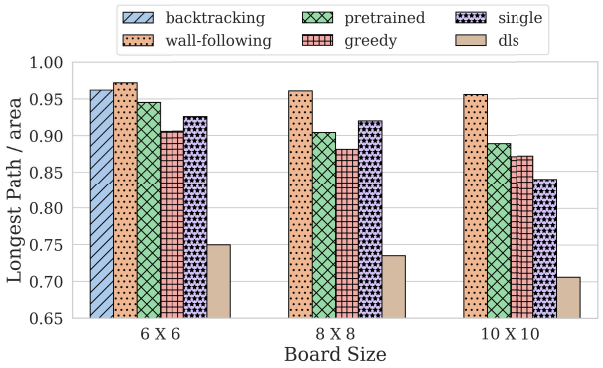

FIGURE 6. The ratio of longest path to area of each agent in stationary environments generated during the simulation of real games.

TABLE 1. Results of the round-robin tournament in 6 × 6 grid.

Agent	vs P	vs G	vs S	vs M	vs A	vs RP	vs RG
P	-	**254**	**244**	**472**	**285**	**515**	**483**
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	-

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.

2) 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

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
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	-

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

Agent	vs P	vs G	vs S	vs M	vs A	vs RP	vs RG
P	-	**299**	**349**	**504**	259	**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
P	**99 / 742 / 159**	**38.372%**
G	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%

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.

3) 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

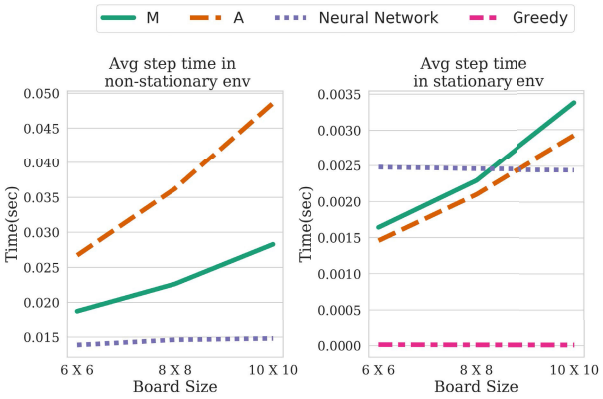

FIGURE 7. Comparison of different algorithms in terms of the elapsed time for each action.

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.

4) RUNTIME COMPLEXITY OF ALGORITHMS

Since Tron is a real-time game, the runtime complexity of algorithms playing the game of Tron is a very important factor. In order to analyze and compare the runtime complexity of algorithms, we present the elapsed time of each agent determines an action in Figure 7. It is readily verifiable that the neural network agents and greedy approximation are more time-efficient compared to the other algorithms. In non-stationary states, agents using neural networks spent almost the same amount of time to determine actions regardless of grid size. In addition, the amount of elapsed time slightly increases even when grid size increases to 10×10 thanks to the parallel computation of GPU. On the other hand, the elapsed time to determine an action of **A** agent and **M** agent increases noticeably particularly in non-stationary environments. Even in stationary environments, the elapsed time of the wall-following heuristic increases much more than the pretrained approximation. Obviously, the fastest longest path approximation heuristic is greedy approximation whose time complexity is linearly proportional to the number of cells. Although the wall-following heuristic shows the best approximation performance especially on larger grid, the pretrained or greedy approximation can be competent methods considering the time complexity of algorithms. Considering the fact that Tron is normally implemented on much larger grid sizes, the scalability of our algorithm can be a huge advantage when deployed in real-time game environment.

V. CONCLUSION

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 for the longest path problem on the grid graph, which is known to NP-complete, compared to the other agents using various longest path approximation heuristics. Especially, our approach demonstrates a better performance against the a1k0n algorithm, the Google Tron AI Challenge winner, on 6×6 and 8×8 grids. On 10×10 grid, our approach does not reach the level of the a1k0n algorithm but shows potential in terms of computational cost. We expect that the modular RL approach can be effectively applied to many complicated real-life problems or games, which can be seen as combinations of multiple tasks to achieve the ultimate goals. We also believe that our algorithm can be improved by training the agent playing in stationary states by generating more realistic random stationary states as we suspect that the main reason for performance degradation is due to (observed) differences in randomly generated stationary states and stationary states encountered from real game plays. We leave the problem of generating more realistic stationary states using generative networks such as generative adversarial networks (GANs) or variational autoencoders (VAEs) to train the stationary agent better as a follow-up research idea.

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 *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," in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2017, pp. 6379–6390.

[4] K. Zhang, Z. Yang, and T. Başar, "Multi-agent reinforcement learning: A selective overview of theories and algorithms," 2019, *arXiv:1911.10635*.

[5] K. Abe, Z. Xu, I. Sato, and M. Sugiyama, "Solving NP-hard problems on graphs with extended AlphaGo zero," 2019, *arXiv:1905.11623*.

[6] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," in *Proc. Workshop Track 5th Int. Conf. Learn. Represent.* OpenReview.net, 2017.

[7] C. Simpkins and C. Isbell, "Composable modular reinforcement learning," in *Proc. AAAI Conf. Artif. Intell.*, 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 *Proc. 21st Nat. Conf. Artif. Intell. 18th Innov. Appl. Artif. Intell. Conf.*, 2006, pp. 318–323.

[9] J. Andreas, D. Klein, and S. Levine, "Modular multitask reinforcement learning with policy sketches," in *Proc. 34th Int. Conf. Mach. Learn.*, vol. 70, 2017, pp. 166–175.

[10] J. A. Mendez, H. van Seijen, and E. Eaton, "Modular lifelong reinforcement learning via neural composition," in *Proc. 10th Int. Conf. Learn. Represent.*, 2022.

[11] L. Kang, "Endgame detection in Tron," Maastricht Univ., Tech. Rep., 2012.

[12] A. Sloane. (Aug. 2, 2011). *Google AI Challenge Post-Mortem*. Accessed: Apr. 7, 2021. [Online]. Available: https://www.a1k0n.net/2010/03/04/google-ai-postmortem.html

[13] S. J. Knegt, M. M. Drugan, and M. A. Wiering, "Learning from Monte Carlo rollouts with opponent models for playing Tron," in *Proc. 10th Int. Conf. Agents Artif. Intell., Revised Sel. Papers*, 2018, pp. 105–129.
[14] A. Shmakov, J. Lanier, S. McAleer, R. Achar, C. Lopes, and P. Baldi, "ColosseumRL: A framework for multiagent reinforcement learning in n-player games," 2019, *arXiv:1912.04451*.
[15] S. J. Knegt, M. M. Drugan, and M. A. Wiering, "Opponent modelling in the game of Tron using reinforcement learning," in *Proc. 10th Int. Conf. Agents Artif. Intell.*, 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 *Proc. IEEE Conf. Comput. Intell. Games*, 2012, pp. 242–249.
[17] M. J. W. Tak, M. Lanctot, and M. H. M. Winands, "Monte Carlo tree search variants for simultaneous move games," in *Proc. IEEE Conf. Comput. Intell. Games*, 2014, pp. 1–8.
[18] S. Samothrakis, D. Robles, and S. M. Lucas, "A UCT agent for Tron: Initial investigations," in *Proc. IEEE Conf. Comput. Intell. Games*, 2010, pp. 365–371.
[19] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *Proc. 4th Int. Conf. Learn. Represent.*, 2016.
[20] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.
[21] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. P. Lillicrap, and D. Silver, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.
[22] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proc. 33nd Int. Conf. Mach. Learn.*, vol. 48, 2016, pp. 1928–1937.
[23] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba, "Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation," in *Proc. Annu. Conf. Neural Inf. Process. Syst.*, 2017, pp. 5279–5288.
[24] R. Bellman, "A Markovian decision process," *J. Math. Mech.*, vol. 6, no. 5, pp. 679–684, 1957.
[25] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, Dept. Comput. Sci., King's College, Cambridge, U.K., May 1989.
[26] V. R. Konda and J. N. Tsitsiklis, "On actor-critic algorithms," *SIAM J. Control Optim.*, vol. 42, no. 4, pp. 1143–1166, 2003.
[27] S. Padakandla, K. J. Prabuchandra, and S. Bhatnagar, "Reinforcement learning algorithm for non-stationary environments," *Appl. Intell.*, vol. 50, no. 11, pp. 3590–3606, 2020.
[28] C. Sun, W. Liu, and L. Dong, "Reinforcement learning with task decomposition for cooperative multiagent systems," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 5, pp. 2054–2065, 2021.
[29] M. Willem, *Minimax Theorems*, vol. 24. Springer, 1997.
[30] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artif. Intell.*, vol. 6, no. 4, pp. 293–326, 1975.
[31] M. Erwig, "The graph Voronoi diagram with applications," *Netw., Int. J.*, vol. 36, no. 3, pp. 156–163, 2000.
[32] A. Rosenfeld and J. L. Pfaltz, "Sequential operations in digital picture processing," *J. ACM*, vol. 13, no. 4, pp. 471–494, 1966.
[33] A. Itai, C. H. Papadimitriou, and J. L. Szwarcfiter, "Hamilton paths in grid graphs," *SIAM J. Comput.*, vol. 11, no. 4, pp. 676–686, 1982.
[34] G. Sartoretti, J. Kerr, Y. Shi, G. Wagner, T. K. S. Kumar, S. Koenig, and H. Choset, "PRIMAL: Pathfinding via reinforcement and imitation multi-agent learning," *IEEE Robot. Autom. Lett.*, vol. 4, no. 3, pp. 2378–2385, 2019.
[35] D. Misra, "Mish: A self regularized non-monotonic neural activation function," in *Proc. 31st Brit. Mach. Vis. Conf.*, 2020.

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, in 2022, where he is currently pursuing the M.S. degree in computer science and engineering.

From 2017 to 2019, he served in the military and was discharged from the military service as a Sergeant. His research interests include deep neural networks and reinforcement learning for game AI.

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, in 2021.

From 2017 to 2019, he served in the military and was discharged from the military service as a Sergeant. Since 2021, he has been working as a Data Analyst at the AI Laboratory Unit, KEB Hana Bank, Seoul, South Korea. His research interests include deep reinforcement learning for game AI and computational complexity of algorithmic problems.

SANG-KI KO received the B.S. and Ph.D. degrees in computer science from Yonsei University, Seoul, South Korea, in 2010 and 2016, respectively.

From 2016 to 2017, he was a Research Associate at the University of Liverpool, U.K. From 2017 to 2019, he worked as a Senior Researcher at the Artificial Intelligence Research Center, Korea Electronics Technology Institute, South Korea. He is currently working as an Assistant Professor at the Department of Computer Science and Engineering, Kangwon National University, 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.