Design and Analysis of AI Bots for Tron Game

Vinay Nadagoud and Praveen H V
College of Computer and Information Science
Northeastern University,
Boston, USA

Abstract

In this paper, we evaluate the performance of AI agents for the game of Tron that are modeled using minimax, alpha-beta pruning, and Neural Network with Genetic algorithm. Initially, we discuss the performance of minimax agent which we believe is a commanding algorithm given the deterministic nature of this game. Further, we discuss the performance gain achieved by using a better variant of minimax like alpha-beta pruning. Both these agents are conventional algorithmic-based approaches. In contrast to previous approaches, we use Neural Networks, a learning-based approach which closely mimics the human way of solving any problem and discuss the impact of various parameters that govern the results. We compare the performance of each of these agents against different reflex agents and discuss the advantages of one approach over the other.

Introduction

The game of Tron was introduced in 1982 which quickly gained popularity in the US market. It is believed that by the end of 1983 the game generated around $45 Million in revenue [1]. The game consists of four variants which are Grid bugs, Light Cycles, MCP cone and Tanks[1]. The most popular one among these is the Light Cycles which is similar to the well-known Snake game.

The game consists of two players placed in a grid surrounded by walls, competing for survival. Each player moves within the grid leaving behind a trail known as light cycle. The objective is to force the other player into walls or light cycles left behind by any of the player. The last player to be alive wins the game. Diving back into the history of video games Tron was once awarded the best coin operated game of the year. To this day playing this game and winning each game has been a fascinating experience.

[1.] https://en.wikipedia.org/wiki/Tron_(video_game)

Switching the roles, we now step into the shoes of a game developer and investigate the complexity involved in modelling an AI agent that can be a tough competitor for a human player. In doing so we explore the latest techniques and advancements in the field of AI and propose solutions using algorithms like minimax, alpha beta pruning and neural networks.

We begin our exploration by considering the algorithmic based approaches starting with minimax [1]. Minimax is best suitable for two player zero sum deterministic game. At each step the algorithm determines the best possible action for a player to maximize its score. In doing so, the algorithm always assumes that the opponent plays its best move thereby lowering the score for the player. Due to the large state space and available moves for each player the minimax tree grows exponentially.

To reduce the computation time and to eliminate portions of the tree which doesn't influence the score, we use alpha beta pruning. This is an extension of minimax algorithm which maintains two values alpha and beta which tracks the current minimum and maximum values at each node and prunes the sub-tree that doesn't change the current minimum and maximum values [2].

One of the main challenges of the above mentioned algorithmic based approach is the high computation time required to determine next optimal move for each turn. An alternate approach is to use neural nets which have dominated in the recent times for most applications. Neural network as such is not programmed with any task specific rules, it learns to predict the optimal move by training on a huge set of examples by adjusting the weights of the neurons. The way it naturally corrects is through back propagation to adjust weights of neurons but as we can't generate all the possible state space for the problem at hand we move to genetic algorithm.

Outline: In the next section we explain the game specifications and the various parameters which govern the game flow. We show the usage of algorithmic based approach

and its limitations followed by the results obtained using neural networks. The influence of scoring scheme on the prediction is analyzed and shown in the later part of this paper.

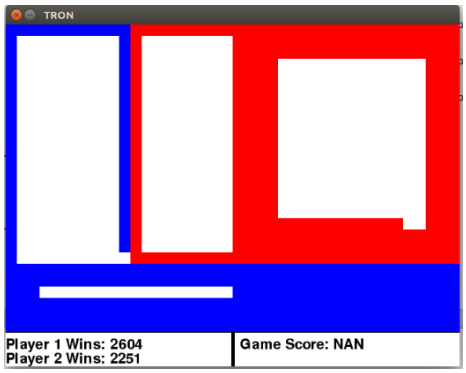

Figure 1. (Tron Game)

Methodology.

The Light Cycle Tron is a multi-player sequential game where each player takes turns to make a move. In this paper we limit the number of players to two. The two players start at any desired position within a grid with dimensions 640 x 480 as shown in Fig 1.0. The grid is surrounded by wall. On each turn a player can move in one of the four directions Left, Right, Up and Down. A player dies if it hits a wall or any of the light cycles. The last player to be alive wins the game.

Algorithmic based approach.

To apply minimax and alpha beta pruning algorithms we first represent the game of Tron as an adversarial search problem as shown below.

Start State: Random position for heads of two Tron's within the grid of 640 x 480

Players: Blue Tron which is Minimizer and the Red Tron which is Maximizer

Actions: LEFT, RIGHT, UP, DOWN

Transition (s,a,s'): Probability of ending in state's'' after taking an action a from state s. In our case we consider it uniform.

Terminal Test: True if Minimizer or Maximizer dies else False.

Terminal utilities: 2 if Maximizer wins –2 if it loses else 0. The game is now represented as a tree with each node as a state and an edge as an action as shown in the figure Fig.2

[2] CS-5100 Adversarial Search Christopher Amato

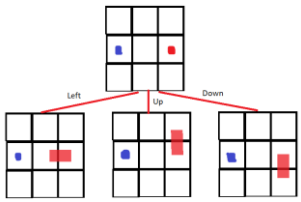

Figure 2 (Tree Representation)

The minimizer and maximizer play alternatively at each level. Minimizer tries to minimize the score in contrast to the maximizer that tries to do the opposite. When the terminal state is reached the state is evaluated and a reward is

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Figure 3. (Minimax) [3]

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 4. (Alpha-Beta) [3]

propagated back to the root of the tree. The maximizer will now choose the action that led to the maximum score as the next move. In ideal scenario the tree is explored until the leaf nodes are terminal nodes. But this will be compu-

tationally very expensive. The pseudocode for the minimax algorithm is given in figure 3. With a branching factor of 3 and depth d the time complexity for this algorithm is $O(b^d)$ and space complexity is O(bd).

Evaluation of terminal state.
The terminal state is when one of the players die. If Maximizer is dead, then a score of –2 is returned and if Minimizer is dead then 2 is returned.

Optimizations.
Method 1. As an improvement to above method, we consider alpha beta pruning algorithm that prunes the subtree that does not influence the decision of choosing the optimal move for the Maximizer. In order to prune the subtree, this algorithm maintains two values alpha (Maximizer's best option on path to root[2]) and beta (Minimizer's best option on path to root[2]). The pseudocode is given in figure 4.

Method 2. Exploring the game tree till the terminal nodes are reached could be highly expensive. In order to address this, we introduce a maximum depth d. If depth d is reached while exploring the tree, then we consider these nodes as terminal nodes. We evaluate these nodes by checking if the play area is divided into 2 regions by walls and light trails such that the Trons are unreachable from each other. If so, we calculate the number of moves available for each Tron in their respective regions before suicide becomes inevitable and return a score 2 or -2 depending on whether the Maximizer or the Minimizer has the most number of moves left to play. If no regions are formed, then a neutral score of 1 is returned. In all our experiments we have set the depth to 4. To determine the reachability from one Tron to other, we use DFS algorithm. We start the DFS algorithm from the head of one of the Tron and see if the other Tron's head can be reached. We could also use BFS to check the reachability but we did not experiment this.

Method 3. We propose to store the information about the unreachable regions within the game state. With the help of this knowledge we will be able to decrease the overall runtime of game by stopping the exploration of the game tree as soon as the Trons are in unreachable regions. The Flood Fill algorithm should be run soon after the regions are formed to store the longest path i.e. list of moves, that will keep the Tron alive for maximum time possible. At each remaining turns the agent will simply pop the next move from the list and returns it as the optimal move.

Learning Based Approach.
In the analysis of Neural Network using genetic algorithm we try different scoring schemes and consider a scoring scheme as good in which number of wins for our agent is more than the number of wins for the opponent. We also consider the results where the start positions are varied for each game. In the next section we explain the algorithms that we used, and its working followed by their results.

Implementation of Learning Based Approach:
For our implementation of neural networks, we consider 7 inputs which are players' current co-ordinates in the grid followed by the available actions for the agent. We make use of two hidden layers as represented in figure 6 which is again connected to three output neurons whose value ranges from -1 to +1 where -1 is considered a left turn and a +1 is considered a right turn and 0 to continue moving in the same direction. Hidden layer nodes are activated using the activation function tanh and output nodes are activated using SoftMax function as shown below.

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

Where Z is a vector of real values of K dimensions. This vector is crushed, and the resultant value is between the interval (0,1).

Once a move has been predicted we move in the direction and for every move we provide a living reward. In some case the next move might result to a terminal state i.e. leads to termination of the game. We then check which player is alive, based on that we assign scores to the player. Once the scores are decided we can evaluate if the action was favorable.
For the problem at hand a high score is awarded if the opponent is eliminated. If the agent doesn't get such a score, there's something to learn from it and improve. This is a perfect example for optimization problem. One of the proven methods for optimization problems using Neural Networks is genetic algorithm.

The algorithm starts by considering a set of solutions know was initial population using this it produces offspring's by taking best solutions from the population. This process is repeated for N iterations where each iteration is known as generation. Each game in a generation produces fitness value which is the scoring scheme selected for the game. After each generation some fittest solutions are picked and are mated by selecting the best weight values. This process is known as crossover as show in Figure 5.

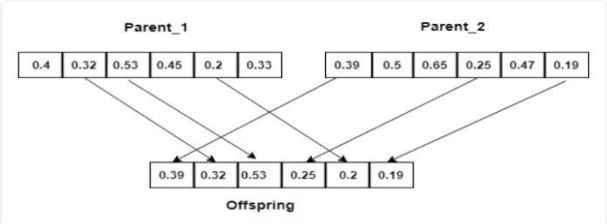

Figure 5(Offspring)

In order to maintain diversity and inhibit premature convergence, with some probability a portion of the offspring is subjected to mutation. In our case we consider the initial population represented by a 50 * 645 matrix where 50 denotes the number of games in a generation and 645 represents the weights for the selected neural network configuration as show in figure 6.

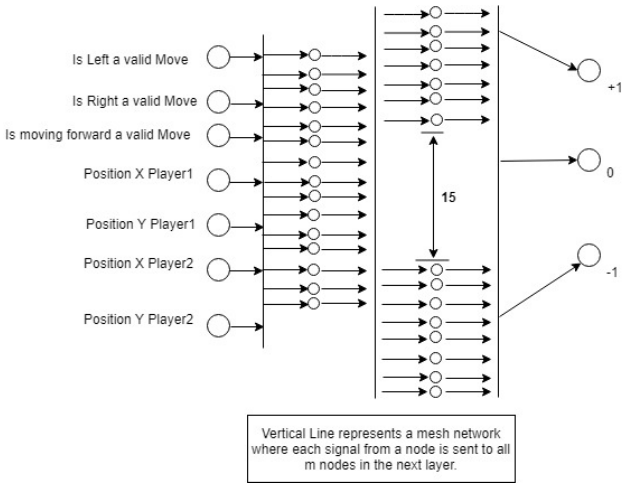

Figure 6. (Neural Network used for Tron).

Fitness Score is calculated using the below variables. These variables act as knobs for the way prediction works. We show the influence of these variables in the next section.

$$\text{Fitness Score} = \text{Length of Tron} + \text{Opponent is dead} - \text{Agent is dead} + \text{clock cycle}$$

Experiments:

We play our agent against different versions of reflex agents that we created and with different start states and find the results as depicted below.

Opponents Considered:

1. Reflex Turn Right – Turns right when next move hits the wall.
2. Reflex Turn Left – Turns Left when next move hits the wall.
3. Reflex Random – Chooses Randomly from available legal moves.

Experiment 1.

Agent that uses Alpha-Beta algorithm with depth = 4 until 2 unreachable regions are formed, then it uses Reflex Random turn left.

Opponent Agent	Start Position	Win Ratio[Opponent agent : Alpha-Beta Agent]
Reflex Turn Right	Random	5:10
Reflex Turn Left	Random	7:10
Reflex Random	Random	10:8

We can see that the Alpha Beta agent is outperforming the Reflex agents in most of the cases. This is because the agent tires to divide the play area into 2 regions such that the two Tron's are unreachable from each other. While doing so it tries to maximize its region. The reason for failures is due to 2 flaws in this approach. Firstly, our agent becomes Reflex Turn Left agent when 2 unreachable regions are formed. In this case even though the regions formed are in favor of our agent it stumbles when it tries to fill the region using reflex turn left. This can be improved using Flood Fill (Method 3). In other cases, the random agent has the advantage by virtue of its start position and also due to the fact that we only explore till depth 4. In most of these cases even a human player would have lost.

Experiment 2.

Agent that uses Alpha-Beta algorithm with depth = 3 and uses evaluation function from Method 2.

Opponent Agent	Start Position	Win Ratio[Opponent agent : Alpha-Beta Agent]
Reflex Turn Right	Random	3:10
Reflex Turn Left	Random	2:10
Reflex Random	Random	2:10

It performs slightly better than the previous approach. As in this case the agent keeps using alpha beta agent till the end. It can detect if the next move is not optimal and can change its course accordingly. This approach still has the disadvantage of exploring the tree only till depth 3. Also the disadvantage of start position for our agent hinders to achieve the goal of Maximum Win ratio.

Analysis of Neural Network with genetic Algorithm.

We measure the performance of neural network against two reflex agents.

In the first scenario the start position of reflex agent is fixed, and we see that our agent starts to perform better and improves every generation depending on the scoring parameters. A graph of agent vs opponent win ratio is shown in the figure 7 for the various fitness variable configurations as shown in table 1.

	Score Awarded			
Config	If Opponent is dead	If Agent Dies	If Opponent is Alive	For Length of Agent
1	1000	-5000	-10	0
2	10000	-5000	-10	0
3	10000	-5000	-500	0
4	10000	-10000	-50	0
5	3000	-5000	-5	length of light cycle

Table 1.

Based on the configuration we can see that having a higher negative value for the condition “opponent is alive” results

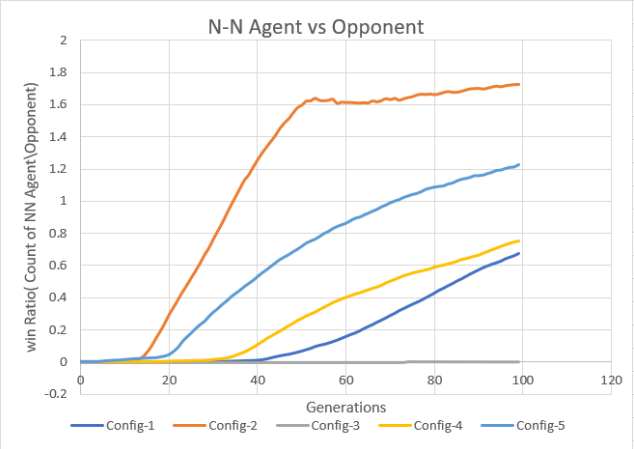

Figure 7

into the agent hitting the wall in order to maintain a higher fitness score. Based on the graph we can assume that having higher reward for “trapping the opponent” allows the agent to perform better than the opponent thereby increasing the win ratio.

The slow progress for config 5 can be explained using the fitness score. As the state space is huge and length of light cycle is used for fitness calculation, the algorithm tries to increase the fitness value for each generation but in doing so it might kill itself. However, as the agent performed better than the previous generation it considers it as a good score.

For config-2 the slope increases drastically as there is no reward for increasing the length of the light cycle, so it rather gains more fitness score by trapping the opponent and maintains the same performance, after some generations the exploration moves towards saturation and the high value solution deteriorates and normalizes.

Once the ratio is greater than 1, we can save the weights and keep using the same weights which might guarantee a win but will fail for a different environment which is considered in the next section.

In the next scenario we fix the start position of our agent and select random position for the opponent. We see that our agent takes many generations to start winning most of the games in a generation, this is due to less knowledge of the game state which in our case is the knowledge of light cycles.

From figure 8 we can see that the number of wins for our agent keeps increasing up to a certain point and starts to improve slowly. For most of the random positions our agent is able to force the opponent to hit the wall. But for some start positions our agent gets trapped as it tries to use the same scheme learnt from previous solutions

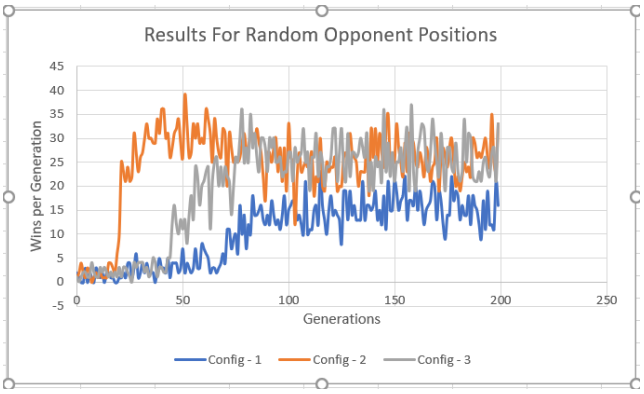

Figure 8.

Even though we see improvement in the number of wins there is a lot of fluctuations. As the number of generations increases our agent learns different ways to trap the opponent. As the game of tron has huge game state, training the agent for all the possible combinations of start states will be beneficial or we can provide more features to help the Neural Network optimize the game. Some of the features we could think of is providing the random start position or providing a vector with light cycles.

Conclusion.

Given the stochastic nature of the game, it is not possible to achieve a 100%-win rate in this game. Many factors like start position of the two Trons and the player with the first move etc. makes it impossible to achieve this. The alpha beta pruning agent however is guaranteed to solve the given problem to achieve maximum win rate as possible but has certain drawbacks. In order for the alpha beta agent to fully solve the problem and achieve maximum win rate the tree has to be explored completely at each turn. This is computationally highly expensive and very unrealistic method to play as the time taken to make next move will grow exponentially with the size of the grid. A solution to this is to limit the depth of exploration of the tree which results in lower win rate. We can use clever techniques like Unreachable Region Detection/Creation and Flood fill to improve the overall runtime for the game as discussed in this paper.

Though Neural Network with genetic algorithm can solve a large domain of problems exploring a large set of examples, we could see from the experiments conducted above that neural nets starts to perform good once it learns the best weights for all the features. In our experiments we could see that when the tron start positions are fixed our

agent quickly learns the model and outperforms the opponent by maximizing the score on each generation. Whereas when the start position is random our agent takes a lot of generations to learn the best weights which leads to higher winning ratio. With random start position our model requires much more training compared to the previous model. Our hypothesis is that if we train the model for about more than a million games, our agent will start performing better. But training for such a huge number of generations takes a lot of time.

Comparing both algorithmic and learning based approaches we conclude that the algorithmic based approach has zero training time and are capable of computing the optimal next move on the go. However, the computation time to get the optimal next move is unreasonably high for a reasonably small game. This makes the agent unfriendly and not suitable for play. In case of learning-based approach like neural networks, there is a high training cost. The agent has to be trained for more than millions of games. Only then would the underlying model learn the best possible weights for each feature. Once the model is trained enough and the weights are known, it can play at very high speeds out performing even humans and almost always winning the game.

Future Work.

In the algorithmic based approach, we wanted to try out the "alpha beta and flood fill" agent and compare the run time with the traditional pure "alpha-beta" agent. Our hypothesis is that once the 2 regions where Trons are unreachable from each other are formed, it makes complete sense to just use flood fill to kill the time until the opponent crashes into the wall. In case of neural network there is lot of scope for performing feature engineering and identifying the dominant features. The better features we feed to the neural network, the better would be the performance. One such idea was to feed all the wall positions to the neural network. Given the position of walls we believe the neural network model will be in better shape to predict the outcome of the game.

References

[1] Ronald L. Rivest, 'Game tree searching by min/max approximation', Artificial Intelligence, 34(1), 77–96, (1987).

[2] Knuth, D.E., & Moore, R.W. (1975). An Analysis of Alpha-Beta Pruning. *Artif. Intell., 6*, 293-326.

[3] Stuart J. Russell and Peter Norvig Artificial Intelligence A Modern Approach Third Edition.

[4] https://theailearner.com/2018/11/09/snake-game-with-genetic-algorithm/

[5] https://code.google.com/archive/p/tron2/

[6] Knegt, Stefan & Drugan, Madalina & Wiering, Marco. (2018). Opponent Modelling in the Game of Tron using Reinforcement Learning. 10.5220/0006536300290040.

[7] https://www.sifflez.org/misc/tronbot/