COMP3211 Project

Artificial Intelligence for Tron game

Group 7

Chiu Ka Wa (20369737)
Chun Wai Wong (20265022)
Ku Chun Kit (20123470)
Abstract

Tron is an old and popular game based on a movie of the same name. There are totally four sub games, and “Light Cycles” is one of the more popular ones. Various attempts have been made to create an artificial intelligence that can play the game well. There is also an online platform of Light Cycles for developers to write their own A.I. and compete against each other. In this report, we will review on the methodology we have taken to approach the game.

Keywords
Minimax, Flood filling, Pathfinding

Introduction to Tron Battle

Problem Definition
Light Cycle consists of a 30 (width) by 20 (height) rectangular grid with two players starting at different random locations. The two AI players move on the grid, leaving a trail on the way. A player loses if he makes a move that puts him outside the grid, or colliding with trails by himself or by others. The last surviving player is the victor.

Input of the game
The program reads from STDIN, with the following inputs: [1]
- Line 1: Two integers N and P. Where N is the total number of players and P is your player number for this game.
- The N following lines: One line per player. First line is for player 0, next line for player 1, etc. Each line contains four values X0, Y0, X1 and Y1. (X0, Y0) are the coordinates of the initial position of the light ribbon (tail) and (X1, Y1) are the coordinates of the current position of the light ribbon (head) of the player. Once a player loses, his/her X0 Y0 X1 Y1 coordinates are all equal to -1 (no more light ribbon on the grid for this player).

Game mechanics
In each turn, every player is required choose an action in {UP, DOWN, LEFT, RIGHT} within 100ms. Every move is legal, but making a move that collides with the trails left behind by any player, or moving out of the game board would causes the player to lose the game immediately, and turning trail he has left behind turns back to empty cells.

After an AI has decided the next move, it may collide with a trail left behind by players or moving out of game grid, which means instant defeat of the player colliding with a trail. His own trail will be removed and the remaining players compete until there is only one player left who will in turn becomes the victor.

Our project made use of an online platform Codingame [1] which let us compete with thousands other player. The board size in the platform is 30x20, and we kept
player number to 2 for simplicity.

![flooding.gif]

Fig.1 A typical Tron battle game ([flooding.gif](#))

**Challenges**

Although the game only involve deciding one action from a set of 4 possible actions, it is difficult for computers to brute force the optimal solution like the game of Go. There are many different strategies to win such as wall-hugging, blocking off opponent, or simply trying to survive as long as possible and wait to the opponents to make mistake, it can be difficult to interpret the best move based on the game state, or to compare the utility of different game states.

Moreover, the game a repeated game of many steps and future possibilities while the computation time given to an AI each step is capped at 100ms. The complexity of the problem also increases greatly with the number of players. For example, if we want to do a simple depth-first search, the worst case can get us $O(4^n)$ for a depth of $n$. Some developers also attempted using monte-carlos search, but minimax is more discussed in the game forum among contestants.

A number of heuristics analyzing the free space on the board and the relative positions of the players have yielded promising results. Due to the limited time in calculation of steps, minimax with limited depth and a heuristic to evaluate the game states on the tree leaves are widely adopted.

**Related Work**

Tron has been an old game and a lot of attempts have been made to create an artificial
intelligence that can perform well in this game. Some have tried to find a winning strategy for the game [2], but was found to be an N-P hard problem. Different techniques have been employed to find optimal solutions, including the heuristic techniques and the monte-carlo search. Tron is chosen as a game for programmers to design artificial intelligence to compete against, and a popular algorithm used is minmax [3]. In our work, we also implemented minmax algorithm in our design. In previous works [4], attempts have been made to improve heuristics to increase the performance of the agents, such as trapping the opponent and avoid getting trapped.

Baseline and Oracles
Our baseline emulates a very dumb AI, which does not do anything except choosing randomly between possible action set S: { Up, Down, Left, Right }. This AI is rather suicidal, so we’d also constrain it to not suicide unless necessary in the next step. So it searches forward to check for constraints, but it doesn’t backtrack to fix it’s own mistake because it will only decide randomly for next step among the possible action set Sp: { z | z in S, z won’t lead to suicide }.

The oracle AI of this game plays very well, like a real human or even better. We expect an oracle AI to be able to perform tricks like not trapping itself, prefer larger space than smaller place that is more likely to get trapped later. More importantly, it should not suicide unless it is inevitable. Oracle AI could be implemented using minimax search, with alpha-beta pruning to speed up the search process, combined with an appropriate heuristics to evaluate the utility at the leaves. Additionally, this AI should consider the opponent’s playing pattern, whether it is moving randomly or according to minimax, since the prediction of the opponent’s action plays a significant role in minimax search.

Our Approach
In this project, we try to build an optimal solution that performs similarly to the oracle of the game. We decided to use minimax, and explore different heuristics based on reasonable strategies, for example, always leaving a row as an emergency escape path, to build the AI bot we have at the end.

Minimax
Due to the time limitation imposed by the game engine (we had only 100ms to make a decision), we had to use minimax with fewer depths. After experimenting with different heuristics, we discovered that a depth of 6 levels is the furthest we can go. The minimax search expand the nodes of the players’ heads and maximizes each player’s option accordingly. After expanding the nodes around the AI’s head, the AI would choose the cell with the highest heuristics value based on the minimax search.
**Heuristics**

**Admissible move**
The most basic idea to survive in the game is not to run into walls or trails until there is no options left. These actions are omitted in the searches to save run time.

**Surrounding empty spaces**
One simple calculation is to sum the surrounding empty cells as the heuristic, as empty cells are key to surviving in this game. However, this method does not take opponent’s movement and strategies into consideration. It also has little meaning if the minimax has too few depths.

**Distance from opponents**
The algorithm has very limited depths, getting close to the opponent showed better results given limited number of depths because getting closing to the opponents effectively reduce the area that needs to be considered, and less depths is required.

**Flood filling**
When there are multiple admissible actions, flood filling helps deciding between which action should be taken. Generally speaking, the AI should move towards the direction with more empty spaces to minimize the chance of getting blocked or trapping itself inside.

**Implementations**

**Minimax with Heuristic**
In this version, a few heuristics were chosen to evaluate the score of each cell. The idea of this algorithm is to get close to the opponent as quickly as possible, up to a minimum distance. After that, the player would choose the cell with the least trails and walls around. The player would also specifically avoid travelling on the edge to increase escaping options.

The score is calculated by the sum of the values of the surrounding cells, a positive score for an empty cell, negative score for trails, and even larger penalty for cells out of the game board, and a reciprocal of the distance between the cell and the opponent with a huge weight. The large weight for the distance component means this will dominate the player movement. The weight than become 0 after the distance drops to 5, where the cell depends on the sum of surrounding cells. Undesirable moves are given low score, and losing moves are given very low scores that indicate negative infinity.

Based on the evaluation method, a minimax with a maximum of 5 depths are used to
calculate the best move.

Source code of this implementation is available at: https://github.com/tommyku/comp3211-tron/blob/master/minimax/tron.cpp

You may run the code by copying the source code into the official game site https://www.codingame.com/ide/puzzle/tron-battle, and choose to run it with programming language C++.

**Flood Filling**

This implementation went through several phases, from simple local space counting to flood filling that yielded great results.

Phase one was simple local space counting. First the bot evaluates all admissible actions that wouldn’t lead to suiciding, and for each admissible option it checks for the surrounding 8 cells, and pick the one with highest empty cell count. This gave us some degree of success, but it is still rather suicidal, you may refer to the following game recordings.

![Fig.2 Local space counting avoiding trapping itself (localspace.gif) (our AI is in orange)](localspace.gif)
In phase two, we tried to avoid most death by trapping ourselves and tried to increase survival rate by choosing the direction with the widest accessible space. Simply increasing the kernel size of local space counting wouldn’t work because the road may be blocked off, so we need flood filling instead. We use a flood filling algorithm with limited size of 7 because we would run out of computation time otherwise. Interestingly, if we increase the size to, say, 21, the algorithm actually did worse than with 7.

The result was good and it brought us from the Wood league to Bronze league (where the AIs are stronger).
Our flood filling algorithm successfully avoid the apparent trap at the bottom set by the opponent, and it survived long enough for the opponent to make mistake. This is powerful because we didn’t even do a minimax, just simple flood filling.

This game showcases the behavior of our flood filling algorithm in which in the middle of the game it sort of stayed in the middle but later turned to the left as
the opponents claim the space on the right, this is exactly as what we have expected, it should seek space and stay alive.

An alternative to this is of course to be aggressive and seek the opponent and try to limit their available space. However we didn’t do this in our project.

Source code of this implementation is available at: https://github.com/tommyku/comp3211-tron/blob/master/floodfilling/ver5-flood-fill.rb

You may run the code by copying the source code into the official game site https://www.codingame.com/ide/puzzle/tron-battle, and choose to run it with programming language Ruby.

**Evaluation**

The minimax AI was ranked 1246/1262 on the Bronze league and the flood filling AI was ranked 245/1262. We believe this result is satisfactory as our AIs avoided many suicidal behaviors such as getting trapped and they survived by setting up traps to limit the opponents and stayed away from traps laid by ourselves and by the opponent. We identified some winning conditions of the game, and undesirable approaches.

**Minimax bot**

The approach to get close to the opponents yielded interesting results. Sometimes the player can skillfully block opponents’ paths, but sometimes it worked the other way around. Upon closer examination of the cases, the outcome of this approach depends heavily on their initial positions. In the future, we might look into possibilities of algorithms that makes use of, or be aware of the starting advantages.
After implementing the minimax algorithm, the performance of the player greatly reduced. It was likely due to the heuristics not correctly predicting the opponent’s’ moves. There are many possible heuristics, running minimax using only simple heuristics would greatly reduce options originally available. The minimax algorithm also only support very few layers as the computation time is limited to 100ms.

**Flood filling bot**

This approach worked well for most of the cases, it avoid suicidal behaviors and never hit the walls unless it is inevitable. However, it is too passive that when confronting with an aggressive player, the living space of the bot sometimes are hugely confined. See the following recording for example.
We believe further tweaking of the algorithm by integrating with minimax, or using opponent-seeking strategy to block off opponents before we get ourselves blocked, then switch into “survival mode” to avoid mistakes in the end game should give us better performance.

**Conclusion**

After testing the different approaches and heuristics, we identified some strategies that are more effective such as flood filling. We have seen that minimax isn’t automatically the best approach because it faces limited resource and we needed to tune a better heuristic to evaluate the game states at the leaves. Rather, an algorithm as simple as flood filling, could avoid many mistakes despite it didn’t try explicitly to predict the future.

We realized several ways that we can improve. In the future we may try to analyze the opponent’s’ behaviors and incorporate the results in the minimax algorithm, and try to analyze the game state from a different perspective.

**References**


[3] "Coding Games and Programming Challenges to Code Better", CodinGame,