CS221 Project progress - Light cycle racing in Tron

Jean-Baptiste Boin

1 Introduction

In this work, we are building agents for the light cycle game inspired by a scene from the movie *Tron*. In our version of the game, 2 agents (blue and orange) move on a 2D grid. Each of them can move in 4 directions (up, down, left, right). After each move, the previous cell turns into a wall. The game ends when one of the agents moves to a wall cell or if they move to the same cell, in which case the game ends on a draw. The agents are not allowed to stay in the same cell, so at every move empty cells are consumed, which gives an explicit maximum bound on the number of moves before the game is over.

In the classic definition of the game, both agents move simultaneously at every step. We decided to play a slightly different turn-based variant of the game. In this variant, the agents move one after the other. The symmetry is broken because this means that we need to decide who gets to start, which may bring an advantage. Like for chess, we arbitrarily decide that the blue agent gets this advantage. However, we decided to work with this simpler variant of the game so that we could use algorithms like minimax, instead of more complex methods that involve a payoff matrix. This formulation does not fundamentally change the problem but makes it more relevant to concepts learned in this class.

We will first present some previous work done on this game. We will then briefly describe our formulation of the problem and what engine we used for this project. We then describe a few basic strategies that act as a baseline and evaluate them against each other. After giving intermediate results obtained with a relatively simple minimax agent, we show how some strong game dependent heuristics make a big difference in the performance. Finally, we discuss weaknesses of our agent we developed and how they could be addressed.

2 Related work

A few works attempted to build an automatic agent that would perform well at this game. Early work attempted to find winning strategies on a given board [1], but this was proved to be a NP-hard problem.

More recently, Monte-Carlo Tree Search (MCTS) techniques, that proved to work remarkably well for playing go, were attempted in [5] and further improved in [4]. These techniques struggle at getting better results than heuristic-based methods, because they are very sensitive on the methods used for each step (selection, expansion, play-out, backpropagation). An interesting hybrid approach was taken in [2] and improved in [1] where they base these steps on strong heuristics.

In this work, we tried to use MCTS methods but they ended up being quite slow with our infrastructure and did not yield better results than heuristic-based methods in the reasonable time that we ran them, so we decided to choose the heuristic approach instead.

3 Engine building

In order to be able to run many experiments, it was necessary to build a fast and flexible engine to play this game. The first version of the engine that we built suffered from a lack of flexibility. In the long term, this would have probably hindered our capacity to run good experiments. In the meantime, the `pacman` assignment was released. Since that assignment contained a relatively compact framework for 2D grid-based

game that was very relevant to our project, we decided to completely rebuild our engine using that one as a new basis.

Some considerable amount of work was put in that direction. We now have a very easy to use engine with a nice command line interface, allowing us to run many experiments using agents with custom strategies, as well as custom layouts loaded from external text files. For the visualization of games, we did not use the nice graphic interface from the `pacman` assignment but instead stuck to the relatively crude text-based output, which does the job. An example of the output is shown in Fig. 1. The walls are represented by the `%` characters, and the blue and orange agents are respectively represented by a `B` and `O` character.

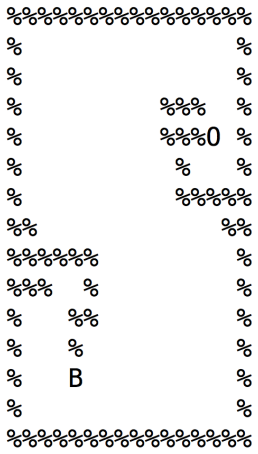

Figure 1: Example output of a game in progress.

4 Baseline

4.1 Definition of basic strategies

In order to get a better understanding of the game and of different strategies that may or may not work, we first implemented basic strategies that could be used as a baseline.

4.1.1 Random

The first strategy is the *random* strategy. At each turn, an agent chooses randomly (with a uniform distribution) among the possible four actions that would not cause the agent to crash into a wall. The purely random strategy does not last longer than a few turns so we added this very simple check so that this baseline is slightly more interesting. It is to be noted that this very short-term strategy only tries to avoid a loss in the next turn, but does not try to avoid a draw. An action that would cause a draw (if the cell we move to is already occupied by the other agent) can be selected even if there are other free cells directly accessible.

4.1.2 Hunter

The next strategy we implemented is the *hunter* strategy. This strategy not only looks at the freely accessible cells around the agent but also at the position of the other agent. The cell that is chosen is the one that minimizes the Manhattan distance to the opponent. If this minimum distance can be achieved by more than one action, we choose one of these randomly. In practice, this agent will try to confront the other agent as fast as possible.

We implemented two variants of this strategy. The difference in those is the way they handle collisions with the opponent.

- Colliding hunter: This agent works exactly as described above, and thus will try to collide with the other agent (Manhattan distance of 0), causing a draw to happen relatively often.
- Non-colliding hunter: This agent introduces a special case for a zero Manhattan distance: actions that lead to collisions will not be chosen if there are other empty cells available, which means that this agent will try to get very close to the other agent but will try to trap him instead of colliding with it. As we will see, in this case draws happen relatively less often.

4.1.3 Runner

The *runner* is the opposite of the hunter. This agent will try to be as far as possible to the other agent, in order to take fewer risks. The implementation is very similar to the hunter, except that this agent will try to maximize the Manhattan distance to the opponent, again with randomization if this maximum is reached for several actions.

4.2 Comparison of strategies and analysis

Now that we defined different basic strategies, we want to see how they measure with each other. Our experiment works as follows. For the blue and orange agent, we choose one of the four strategies presented above (random, colliding hunter, non-colliding hunter, runner) and run 1000 games for each of them. The layout used is an empty 13×14 rectangle surrounded by walls where the two agents start from opposite positions, as shown in Fig. 2. We report in Table 1 the ratio of blue wins, draws, and orange wins respectively.

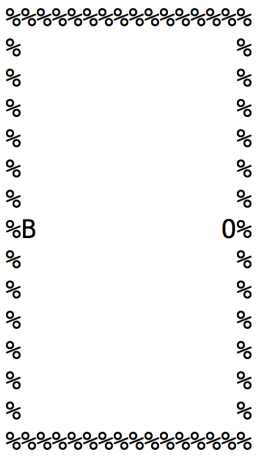

Figure 2: Layout used for our comparison experiment.

Table 1: Comparison of blue win/draw/orange win ratio for different agent types for 1000 games each

agent		ORANGE			
		random	collid. hunter	non-coll. hunter	runner
BLUE	random	0.47/0.07/0.46	0.11/0.52/0.38	0.10/0.53/0.37	0.12/0.03/0.86
	collid.hunter	0.28/0.66/0.06	0.00/1.00/0.00	0.00/1.00/0.00	0.15/0.81/0.03
	non-coll. hunter	0.62/0.12/0.26	0.51/0.11/0.39	0.49/0.11/0.40	0.64/0.11/0.25
	runner	0.85/0.01/0.14	0.18/0.56/0.26	0.19/0.53/0.28	0.52/0.01/0.47

The interesting remark on these values is regarding the asymmetry due to our turn-based formulation of the game, the most blatant case being the case colliding hunter vs. non-colliding hunter. When blue (who

is the starting player) is a colliding hunter and orange a non-colliding hunter, there is a draw everytime, but the situation is very different when the roles are inverted. This is due to the fact that blue is the starting player, but also to the distance between the initial position of each players. In both cases, all games will start the same way, with both players going towards the other until they are both in adjacent cells. At this point it is blue's turn. Thus, if it is a colliding agent, it will deterministically collide with its opponent in the next step. If the initial distance between the agents was odd instead of even, the results would be reversed. This means that being the starting player does not give an advantage in itself, but it also depends on the layout used.

For the future, if we stick to this variant of the game, **it will be more representative to give the starting advantage randomly** instead of assigning it consistently to the blue player.

In any case, this bias being accounted for, there are still some methods that work particularly well against others. The highest win rate is obtained when a runner agent confronts a random agent. This completely makes sense: the point of the runner is to not risk getting trapped by the opponent and try to save time while letting the opponent trap itself. Thus, it works very well against a random agent who is very likely to trap itself after some time. However, a runner agent consistently performs worse than hunter agents. This is because a runner will typically try to stay close to the outer wall so as to escape from the hunter, but as the hunter catches up, the runner is likely to get trapped between an outer wall and the hunter path, and thus lose. This gives us the insight that staying close to the walls may not be a great strategy since it restrains the possible actions. Finally, both kinds of hunter agents seem to perform better than the random agents, even though they perform worse than the runner agents.

5 Minimax

5.1 Vanilla minimax

The turn-based formulation of this problem sounds at first well suited to applying an algorithm such as minimax, but there are some issues that first need to be overcome.

The main problem is related to the fact that it is difficult to define intermediate goals for this problem: the reward only comes at the very last step, either if there is a win or if there is a loss. Thus, a relatively large value for the depth should be used if we want this approach to be successful in trapping another player. Still, with a typical branching factor of 3 per agent (usually an agent has the choice between 3 directions, given that the one it is coming from is excluded because it became a wall), having a very large depth may not be possible to achieve.

This issue is exacerbated by the fact that a bad choice can doom the player many turns later. Indeed, it is easy to break the original game space into multiple connected components, and once this is done a player is doomed to stay in its own connected component. It is easy to see that at the moment that a player divides the space and can choose which component to enter (and stay forever into), making a good decision is critical, and only being able to predict the game for a few turns may not help in making this decision, so it is important to be able to learn to detect these cases that are looking bleak.

5.2 Minimax implementation and optimizations

Our first implementation for the minimax does not include any optimization. In that case, it is not practical to go beyond depth 4 because of the time it takes for each move. The rewards are only defined for the end states: a positive reward of 500 points is awarded to the winning agent, and a penalty of -500 points is given to the losing agent. In case of a draw, no agent gets any reward or penalty. As expected, a clear winning state is usually quite hard to get unless the opponent traps itself (especially with a depth of 4), but the minimax allows the agents not to make too short-sighted choices that would cause an imminent crash (like entering a "hole" with no exit). So, the minimax algorithm is actually quite useful for staying alive longer, which is in fact quite important for a problem where survival is the key.

Seeing this, we decided to improve the quality of this survival mechanism. By looking at a few games, we noticed that indeed, our algorithm could still lead in bad choices that caused issues beyond its prediction horizon and could be easily avoided, such as taking a narrow dead-end path. Thus, we added a new reward for the non-end state leaves that corresponds to the number of possible moves at that position. The idea would be that we want to end up in configurations that are as open as possible. This improved our success rate.

To speed up the minimax search in most cases, we also used the fact that an agent can only travel so far in one turn: the Manhattan distance between two agents can only decrease by 2 at each turn. This means that if the two agents have a Manhattan distance strictly larger than twice the depth, then we are sure that no action taken by the opponent can modify the walls in the agent's neighborhood. Then, the "minimize" stages of the algorithm can be removed. We use this simpler version when the agents are far enough, but switch back to the regular minimax when the distance is below that threshold. It is to be noted that this brings imperfections to the normal minimax, because even an agent that is far away can have, for example by cutting off a big part of the map. Still, this speedup was appreciated when running many games and was worth this slightly degraded performance.

The results obtained by our current version of the minimax (with depth 4) against the other agents for 100 games are given in Table 2. This version achieves more wins than losses against every single agent, and is especially successful against the random agent. The reported results correspond to the version where the player who gets the first move is chosen randomly.

Table 2: Comparison of win/draw/loss ratio for the minimax agent agains other agents (100 games each)

agent	random	collid. hunter	non-coll. hunter	runner
minimax	0.89/0.01/0.10	0.45/0.33/0.22	0.46/0.19/0.35	0.70/0.00/0.30

6 Improved heuristics

Looking at a few games allowed us to understand the weaknesses of this simple minimax agent. In this part, we present some additional heuristics that improved our evaluation function and allow us to better understand what are good configurations for a player.

6.1 Flood-fill algorithm

The main problem of the minimax algorithm is its exponential time with respect to the depth, so it is not practical to run it at very high depths, even when heavily optimized. Instead, it is a better investment to craft a more suitable evaluation function that is capable of doing a better job at detecting configurations that may become problematic. As mentioned in the previous part, the minimax algorithm prevents choices that cause imminent death but when splitting a space in two connected components, the agent can still randomly choose the disadvantageous component as long as it is large enough.

This motivated us to use a flood-fill algorithm in our evaluation function, so as to evaluate how big the accessible space can be at each leaf of our minimax tree. This replaces the weak heuristic previously developed which only tried to maximize the number of legal moves for a leaf. The flood-fill algorithm is a simple graph search problem (formulated recursively in our case as a DFS) and it can be done in $O(MN)$ (worst case) on a $M \times N$ board. This is obviously slower than the previous weak heuristic but could still be done reasonably fast when adjusting the depth. More consideration about this speed constraint is given in 6.4.

It is important to note that the result of this algorithm just gives a higher bound on the number of steps an agent can take until the game is over, but this higher bound may not be reached, even if the agent is isolated in its own connected component. An example is given in Fig. 3. In this example, the agent has

access to all empty cells, but because it cannot go back on its path, it is impossible to find a single path that goes through all these cells.

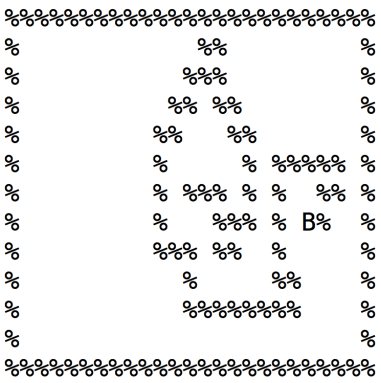

Figure 3: Example of a bad region filling for a single agent (different map from the one we used for our experiment ; this one is just used for illustrative purposes).

6.2 Early game

6.2.1 Trap the opponent

In the first phase of the game, both players will usually be in the same connected component and a good strategy for an agent is to manage to trap the opponent in a smaller connected component than the one the agent has access to. If this is done, a victory is almost guaranteed in the end game (which starts as soon as the players are located in different connected components) if the player plays wisely.

In order to incentivize this behavior, we add an additional positive or negative reward to our evaluation function which gets unlocked for leaf states in which the players are separated. If the space that the current agent can access is larger than the space that the opponent can access, a positive reward of 300 is given, while a penalty of 300 is given if the opponent has access to a larger area of the map.

6.2.2 Avoid getting trapped

After adding this reward or penalty to the evaluation function, we realized that we were still playing quite badly against the hunter agents. In fact, the strategy for these agents is usually quite a good one: while the other agent is exploring its corner, a hunter goes directly at it, so usually gets closer to the center of the map faster. As a result, the other agent can easily get trapped between the hunter and a wall, and this can happen very fast, making a win impossible.

The last heuristic we added for the early game is related to the notion of personal space. When two agents are roaming the same connected component, the agent that is closer to the center usually has an advantage because he can access most of the cells of the map before the other agent. Thus, one good heuristic, also discussed by [2], is to count the number of grid cells that can be accessed first by each agent, which gives a Voronoi diagram of the map [6].

Once this is obtained, we use the difference between the number of agent cells and the number of opponent cells in our evaluation function. The evaluation function is thus higher when a player has access to a larger connected component, but also when the player gets to a more advantageous position on the board.

In terms of implementation, this comes at almost no extra cost compared to the flood-fill algorithm. Instead of using a DFS starting from the current agent location, what we did was to run two BFS starting from each agent, and alternating between those for each level of the tree. If a free tile was not explored, it is assigned to the agent that reaches it first. At the end, when both queues are exhausted, we just count how many cells are assigned to each agent. This algorithm has the same complexity as the flood-fill algorithm

and gives additional information. It also tells for free whether we are in the end game or not (this is the case if one queue visits a cell already visited by the other queue), which can be very valuable.

6.3 End game: Survive

For the end game, the only goal is to survive as long as possible. At this point it is not needed to consider the opponent anymore, so in that phase, for efficiency reasons we just do a single agent minimax search (similar to what is done in the early game when the agents are far from each other). The evaluation function is also simpler compared to the early game, and it only consists of the available space for that agent.

6.4 Alpha-beta pruning

This added complexity in our evaluation function prevents us from using the same depth as with our original simple minimax agent (Table 2). We want each agent to not exceed 1 or 2 seconds of computation per move, and using a depth of 4 pushed us beyond that limit, so we were limited to a depth of 3 at this point. This is why we implemented alpha-beta pruning to speed up that tree search. Indeed, after implementation, we could go back to depth 4, which improves our predicting power.

7 Results

7.1 Simulation results

These added heuristics and optimizations made a considerable difference in the performance of our agent. In order to test that, we run this final agent against our four baseline strategies, and we compare with our previous simple minimax agent. A total of 10000 games are played against each agent, and the results are given in Table 3.

Table 3: Comparison of win/draw/loss ratio for the final agent against other agents (10000 games each)

agent	random	collid. hunter	non-coll. hunter	runner
final	0.9889/0.0108/0.0003	0.4215/0.5652/0.0133	0.8330/0.1437/0.0233	0.9991/0.0001/0.0008

7.2 Comparison with simple minimax agent

It is very interesting to compare these values with the ones obtained for our simple minimax agent in Table 2.

The good performance of our agent against the random and the runner agents is striking: almost no games were lost against these agents (less than 0.1% of the games) and the number of draws is also very low. This makes sense: our heuristic allow for a good performance in the end game, so our agent is quite good at surviving and making good use of the available space. These techniques are particularly good against the random and runner agents.

For the colliding and non-colliding hunter agents, we can notice that the loss ratio is also particularly low, although higher than against the other agents. The win ratio for the non-colliding hunter is also much higher, which is satisfying. However, the ratio of draws is still quite high, especially against the colliding hunter. In fact, when playing against the colliding hunter, the ratio of wins is slightly lower than for the simple minimax agent (45% vs. 42.15%). The way we explain it after looking at some runs is the following: when it is clear that a hunter is succeeding at trapping a player, the final agent prefers to **force a collision** into the opponent while it is possible rather than getting trapped and risking getting a loss. In the case of the simple minimax agent, such a collision is not a desired output because it yields no reward, while staying alive actually gives a positive reward, so the agent may prefer to get trapped in a smaller part of the board.

In that case, it is still possible that the agent gets lucky and that the opponent kills itself, which is why there are more wins for that agent. However, in many cases the agent will die first, so this higher win ratio comes at a cost of a much higher loss ratio.

Table 4: Comparison of win/loss ratio for the minimax and the final agent against other agents when **excluding draws**, using the data from our simulations

agent	random	collid. hunter	non-coll. hunter	runner
minimax	0.8990/0.1010	0.6716/0.3284	0.5679/0.4321	0.7000/0.3000
final	0.9997/0.0003	0.9694/0.0306	0.9728/0.0272	0.9992/0.0008

In fact, we can consider that getting a draw can be a viable strategy, so a better metric would be to consider that when there is a draw that game should be ignored and a new game should be played instead. In table 4, we use such a metric: we ignore the draws and just look at the win and loss ratio (the sum should be 1). In that case, it is much more clear that our final agent performs way better than the simple minimax agent, and this is the case against all basic strategies.

8 Weaknesses and points of improvement

Some additional work could be put in the end game strategy. Fig. 3 shown earlier is a possible path that our final agent can choose. Indeed, at every move, the size of the connected component was just decreased by 1, so the evaluation function does not judge this path worse than a path that would be more compact. This is a considerable weakness in the end game and may be responsible for a big part of the losses we suffered from against our baseline agents.

Still, our heuristic is still usually good enough and only waste a minor part of the space. In fact, when having two final agents play against each other, the map gets often almost completely filled. An example is shown in Fig. 4: the game started in the middle of the map and quickly the agents managed to get separated from the other, BLUE taking the upper half of the map and ORANGE taking the lower half. The game was quite balanced until ORANGE did some poor choices that led to part of its region to become inaccessible in the long run, so it ended up getting trapped, while BLUE could still have almost filled its own region without problem.

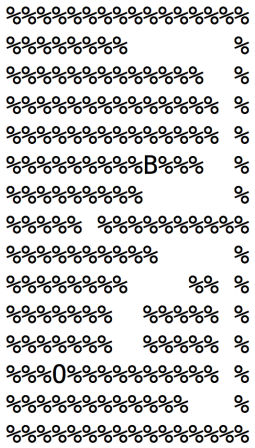

Figure 4: End state of a match between two final agents. ORANGE loses.

Some solutions exist to tackle this issue. In [2], they use articulation points as part of their heuristic.

These points corresponds to vertices in a connected graph that, once removed, cause the graph to become disconnected. These points are critical in understanding the topology of the map in terms of the regions that can be accessed and the ones that cannot. With a smart heuristic on that, it could be possible to make wiser choices in the end game. In any case, finding the longest possible path is a NP hard problem so it is not be possible to get the optimal solution in a reasonable time in every situation.

9 Conclusion

In this work, we built an agent capable of playing a turn-based variant of the light cycle race game in Tron. This work gave us a lot of insight on the importance of good heuristics for the evaluation function when exploring a minimax game tree. Looking at lost games was very informative about how to improve our agent's strategies, and in the end we could perform extremely well against all of the baseline strategies we compared our agent to.

For future work it would be interesting to study the high-level topology of the current map (articulation points, etc.) in order to inform the choice of actions in a better way than we currently do. This may help generalize to very different maps, which we did not attempt in this work. Coming up with better strategies to compare our agent to (e.g. mixed strategies of our current baseline strategies, or strategies learned using Q-learning) could also be an interesting avenue and may inform us on more weaknesses of our current heuristics, so that it could at some point perform well enough to beat a human player.

References

[1] Hans Leo Bodlaender and Ton Kloks. *Fast algorithms for the tron game on trees*. Citeseer, 1990.

[2] Niek GP Den Teuling and Mark HM Winands. Monte-carlo tree search for the simultaneous move game tron. *Univ. Maastricht, Netherlands, Tech. Rep*, 2011.

[3] Marc Lanctot, Christopher Wittlinger, NGP Den Teuling, and MHM Winands. Monte carlo tree search for simultaneous move games: A case study in the game of tron. In *BNAIC 2013: Proceedings of the 25th Benelux Conference on Artificial Intelligence, Delft, The Netherlands, November 7-8, 2013*. Delft University of Technology (TU Delft); under the auspices of the Benelux Association for Artificial Intelligence (BNVKI) and the Dutch Research School for Information and Knowledge Systems (SIKS), 2013.

[4] Pierre Perick, David L St-Pierre, Francis Maes, and Damien Ernst. Comparison of different selection strategies in monte-carlo tree search for the game of tron. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 242–249. IEEE, 2012.

[5] Spyridon Samothrakis, David Robles, and Simon M Lucas. A uct agent for tron: Initial investigations. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 365–371. IEEE, 2010.

[6] Georges Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. premier mémoire. sur quelques propriétés des formes quadratiques positives parfaites. *Journal für die reine und angewandte Mathematik*, 133:97–178, 1908.