AI Agent for Light Rider
CS221 Final Project Report

Wantong Jiang
`wantongj@stanford.edu`

Yipeng He
`yipenghe@stanford.edu`

Di Bai
`dibai@stanford.edu`

Abstract

This is the final project report for CS221 Artificial Intelligence Fall 2018-2019. In this project, we implemented an AI agent for ‘Light Rider’ game. ‘Light Rider’ is a typical two-player simultaneous adversarial game. We use game theory method like *minimax* with alpha-beta pruning optimization and evaluation functions with various strategies. Then we use reinforcement learning algorithms like *Q-Learning* to self-play the game and learn policy. We show that evaluation function intending to attack and block the opponent gets the best win rate. Q-Learning algorithm learned against a minimax agent effectively decreases lose rate to less than 30%.

1 Task Definition

Our project is to implement an AI agent for ‘Light Rider’ game. The game is originally provided on *Riddles.io* [1] online AI game competition platform (`https://playground.riddles.io/competitions/light-riders`).

1.1 Game Introduction

‘Light Rider’ is a typical two-player simultaneous adversarial game. On the same grid, two players begin from different cells. In each round, two players select one of its adjacent cells which is empty and occupy it. The occupied cells become walls after this round and could not be occupied again. The player who has no directions to move or mistakenly moves into an occupied cell or out of the grid loses the game. The player who lasts longer wins.

1.2 Methods

To solve this problem, we use two categories of methods. The first is based on game theory. We change this simultaneous game into a turn-based game and use *minimax* method with alpha-beta pruning optimization. When reaching specified maximum search depth, we call an evaluation function to score the state and choose the state with the highest score. We try multiple evaluation functions with intuition to leave more space for the agent and less space for the opponent.

In the second stage, we try reinforcement learning method. We use *Q-Learning* with epsilon greedy and function approximation. To learn the value of feature weights, we play the Q-Learning agent against a minimax agent and use stochastic gradient descent to update weights each round.

1.3 Local Simulator

The original game of ‘Light Rider’ is provided online. *riddles.io* platform allows players to upload their agent bot, simulates the game, and shows competition results. However, uploading and compiling process online is complex and very time-consuming. As a result, in order to support local debugging, accelerate code evolution process, as well as collect statistical data, we implemented a game simulator.

More details of the implementation will be introduced in next few sections.

2 Background

'Light Rider' game is very similar to the classic 'Snake' game. In 'Snake', the snake moves to eat apples and increases its body length. In 'Light Rider', two "snakes" do not have eat apples but increase their body length by 1 each round. We then found some previous CS221 projects implementing the 'Snake' game AI agents.

In [2], Benoit Z. et al. implemented an AI agent for traditional 'Snake' game. They used deep Q-Learning approach and trained a 4-layer neural network with 50000 games. They concluded that exploration rate decay rate was very important to the learning result and 'Snake' game was so complex that more game data were needed, especially when the length of the snake got very large.

In [3], Felix C. et al. implemented an AI agent for multi-player 'Snake' game. Instead of only one snake on the board in the traditional version, multiple snakes moved simultaneously on one board. They used minimax as well as Q-learning. They modeled Q as a linear function of features and also played against minimax agents. They drew a conclusion that the linear model could easily beat a minimax agent with search depth 2 but had difficulty beating depth 4 minimax agent.

3 Game Model

In this section, we introduce our game model. As we mentioned in previous sections, we model the game as a two-player adversarial game. We then define the game states and rewards.

3.1 Game State

Each game state is composed of the whole board information. Figure 1 shows a state on a $4 * 4$ board. '.' represents an empty cell which both players can occupy. '*' represents an occupied cell. Note that we don't distinguish whether a cell is occupied by player 1 or player 2. An occupied cell becomes wall that can not be knocked into again. '1' and '2' represent the current positions of the two players. This is all the information the agent need to make an action decision. The action is simply one of the four directions: "up", "down", "left", and "right".

*	*	.	.
0	.	.	.
.	.	1	.
.	*	*	.

Figure 1: $4 * 4$ game grid sample

Then we define $Succ(s, a)$. Because we change it into a turn-based game, we only consider one player's movement. When player p takes an action, trying to move to an adjacent cell, we update its position to the cell and update the original cell state to be occupied.

3.2 Rewards

Just as a typical game, we don't define a reward during the game progress. When it's not end, the reward is 0. When the agent wins, which means the agent still has a direction to go while the opponent has no way to go, it gets an absolute large value, like 5000. When the agent loses, it gets the opposite, -5000. When there's a draw, the agent gets a reward of 1000.

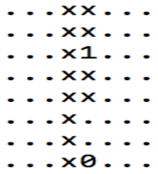
4 Simulator

4.1 Start Code

The online platform does supply us with a simple start code. It receives the board information as well as other game settings from standard input, and returns an action string introduced in last section to standard output. The platform is responsible for state update and competition result check.

4.2 Simulator

Though the platform is developed well enough with clear friendly user interfaces, the uploading process is still time-consuming and we could hardly collect any statistical data. As a result, we wrote our own game simulator. The simulator could take two agents with different strategies (different algorithms, evaluation functions, search depth etc.). It prints each round's board state on the standard output as figure 2 shows.



```
...xx...
...xx...
...x1...
...xx...
...xx...
...x....
...x....
...x0...
```

Figure 2: Simulator prints the state of the board each round

5 Adversarial Approach

5.1 Techniques

As the first step, we tried minimax algorithm. Although the game is a simultaneous one, we solved it as a turn-based game with our agent going first, just like pacman game in our homework. When we uploaded our bot onto the game platform, we found that with search depth 3, the computation might exceed the platform time limit, thus greatly damaged the results. Then we applied alpha-beta pruning technique, and search depth could reach 4, with much better results.

5.2 Evaluation Functions

When search depth reaches the specified max depth, instead of searching deeper, we used an evaluation function to score a state and choose the action that results a state with highest score. We tried multiple evaluation functions. The ultimate intuition is to leave more space for agent and less space for opponent. We implemented multiple versions of evaluation functions and compared them through 1v1 competitions. The strategy details are as follows.

1. **Naive**: Give constant scores and randomly choose a legal action
2. **Straight**: Give high scores to the current direction, which means if going forward is legal, the agent is not likely to change direction.
3. **All_Direct**: Compute how many steps the agent can go in each of the next three directions. Evaluate how flexible the agent is going in each direction, by computing how far the wall is to the left and right side of the agent. Add the combined scores (steps and flexibility scores) for each direction together.
4. **Max_Direct**: Only pick the highest score among the three directions. Intuitively, the agent shall only go one path, so we only need to find the best one.
5. **Attack**: Give high scores if the agent is getting closer to the opponent, which means leaving more walls around the opponent to let it die earlier.
6. **Block**: Give high scores if the agent is getting closer to the escape positions of the opponent. Intuitively if we block the opponent in a confined space by occupying its escape positions, then it's easier to die.

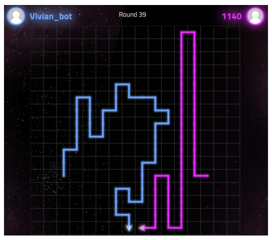
7. **Attack+Block**: Combine All_Direct, Attack, and Block strategies.

The first and second one do not involve learning. The first naive one simply chooses a random action and returns. The second one is an easy greedy algorithm. The third and fourth ones aim to choose the an action that results in a state with more move flexibility and thus more potential for the agent to survive. To represent as formula, we have:

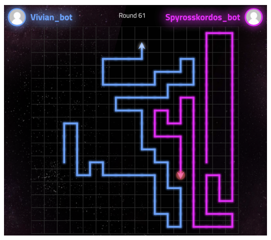
$$Score_{all\_direct}(s) = \sum_{dir \in legalDir(s)} (w_{go} Space(s, dir) + w_{flex} Flexibility(s, dir))$$

$$Score_{max\_direct}(s) = max_{dir \in legalDir(s)}(w_{go} Space(s, dir) + w_{flex} Flexibility(s, dir))$$

The “attack" and “block" evaluation functions move further based on previous efforts: besides seeking more space for the agent, we also try to constraint opponent’s available space. Figure 3a shows the “attack" policy. The blue bot (our agent) moves downwards to attack the opponent bot so that it has no way to go. However, this policy could kill the agent sometimes as well. So to keep the agent safer, we designed the last strategy, “block", which instead of always moves towards the position of the opponent, moves to block opponent’s potential moving directions. In figure 3b, at the first several rounds, our agent detects that if the opponent wants to expand its territory, it should escape from the bottom part of the grid. So our agent moves downwards first to block the space at the right bottom even though the opponent is on top right of the grid at that time. This way, the agent could constraint the opponent as well as prevent itself from being trapped.



(a) Agent tries to attack the opponent directly



(b) Agent blocks opponent’s way to escape

Figure 3: Attack and Block samples: our agent is the blue one

To represent as formulas, we have:

$$Score_{attack}(s) = Score_{all\_direct}(s) + w_1 ManhDist(agent, opp)^{-k_1}$$

$$Score_{block}(s) = Score_{all\_direct}(s) + w_2 ManhDist(agent, blockPos)^{-k_2}$$

k_1 and k_2 are arbitrary positive integers, so that less distance results in higher scores.

5.3 Test Result

We used the simulator to run 1v1 competitions between different evaluation functions. We ran 100 competitions for each configuration and collected the results. Table 1 shows the result.

Table 1: Minimax with different evaluation functions

	Attack	Block	All_Direct	Max_Direct	Random
Attack+Block	56:9:35	52:8:40	59:11:30	57:10:33	85:6:9
Attack	-	54:9:37	66:6:28	76:1:23	85:8:7
Block	-	-	56:9:35	54:6:40	89:6:5
All_Direct	-	-	-	49:8:43	86:2:12
Max_Direct	-	-	-	-	81:7:12

In the table, each element represents the results of 100 competitions between each pair of evaluation functions. The first value means #wins of the left evaluation function. The last value means #wins of the above evaluation function, while the mid value means #draws. For instance, the top left element, “56:9:35" means in the 100 competitions between minimax agents with “Attack+Block" and “Attack" functions, agent with “Attack+Block" wins 56 times, and agent with “Attack" wins 35 times.

The table shows that, all the five strategies greatly improve win rate compared with simple random strategy. Without awareness of the enemy’s position, “All_Direct" performs slightly better than “Max_Direct", so we should take all available directions into consideration. With intention to constraint the opponent, both “Attack" and “Block" strategies get higher win rate against “All_Direct". After taking all the factors into account, “Attack+Block" wins more than half of the competitions against all the other strategies.

5.4 Future Work

There are still many improvements we could do to this category of methods.

1. **Adaptive Search Depth** Currently we only set search depth to a fixed value. We could dynamically change the depth during search to improve efficiency.
2. **TD Learning** We could improve evaluation function by using TD learning.
3. **Expectimax** We could try more algorithms like expectimax and compare its results with minimax.

6 Reinforcement Learning

6.1 Techniques

In the second stage, we implemented an AI agent with Q-Learning algorithm. We learned the Q function: $\hat{Q}_{opt}(s, a; \mathbf{w}) = \mathbf{w} \cdot \phi(s, a)$ by playing the RL agent against a minimax agent.

Feature Each (state, action) pair has one feature: (board layout, action). ‘board layout’ is the string representation of the board state, it contains the state of each cell.

Rewards Unlike minimax algorithm, where we only need to define rewards at the end of game, we need to define a reward for each (state, action) pair. When the action does not result in an game end, we simply set reward to 0. If the action causes the agent to win, the reward is 100; otherwise, the reward is -100. One thing to mention is that here we set the reward of a draw also -100. We found that if we set it to be a positive value, Q-Learning is more likely to learn a policy to get “draw" rather than “win", getting a local optima. We wanted to improve the win rate, so we only praise win here.

Training We run the RL agent against a minimax agent for 20000 games. The discount γ is set to 0.99 because we don’t want the final result to be much less important. We use epsilon-greedy with $\epsilon = 0.2$ to both explore and exploit actions. Each round, we use stochastic gradient descent to update feature weights: $\mathbf{w} \leftarrow \mathbf{w} - \eta[\hat{Q}_{opt}(s, a; \mathbf{w}) - (r + \gamma \hat{V}_{opt}(s'))]\phi(s, a)$.

6.2 Test Result

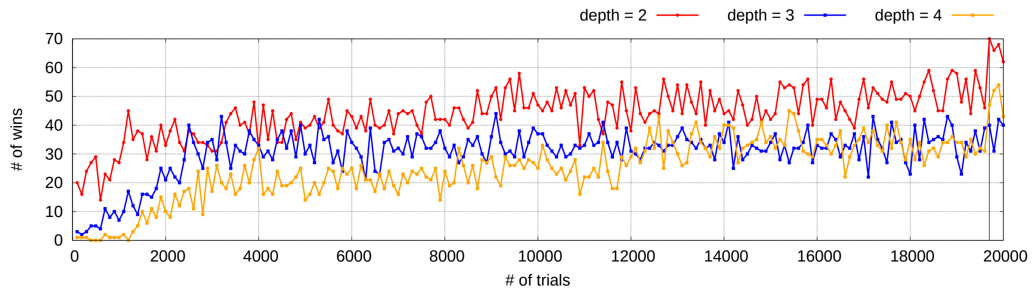
In this section, we show the result of Q-Learning against minimax agent with various settings.

In figure 4 and figure 5, we learned a Q-Learning agent against minimax agents on a $4 * 4$ grid, and in figure 6, we increase the board size of $6 * 6$. We ran 20000 competitions each time for Q-Learning agent to learn, and in the last 400 competitions we turned the epsilon to 0 so that the Q-Learning agent does not do any exploration but only takes the optimal action. We used a black vertical line in the figures to mark this change.

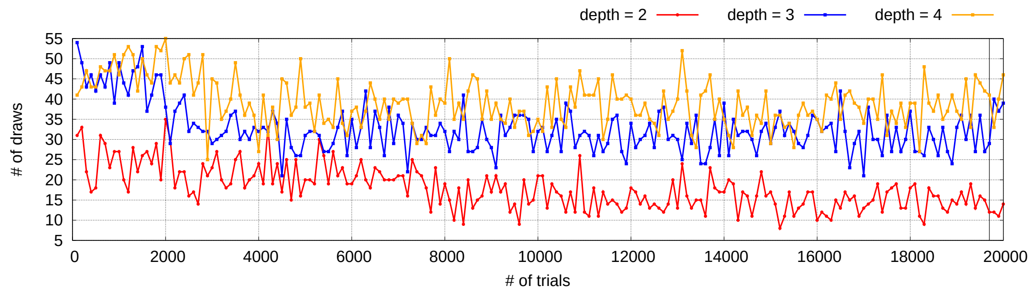
In figure 4, the Q-Learning agent learned against a minimax agent with All_Direct evaluation function and search depth varying from 2 to 4. Figure 4a and 4b show the number of wins and draws every 100 competitions respectively. From the figures, we draw conclusions as follows:

1. During the training, win rate increases while draw rate decreases.

2. After training, the agent is able to achieve a win rate of at least 40%. Win rate plus draw rate could reach approximately 80%, which means the minimax agent could only win about 20% of the game against the Q-Learning agent, whichever the search depth is.
3. Minimax agent with search depth 2 is much more easier to beat (Q-Learning agent could win nearly 70% after training). Training against minimax agent with search depth 3 and 4 get similar results.



(a) Q-Learning agent win rate



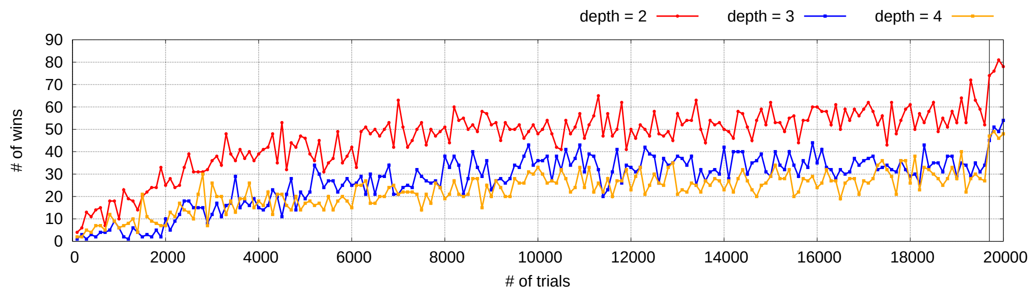
(b) Game draw rate

Figure 4: Win rate and draw rate of Q-Learning against minimax agent with All_Direct Evaluation Function on $4 * 4$ grid during learning process. ($\epsilon = 0$ in last 400 competitions.)

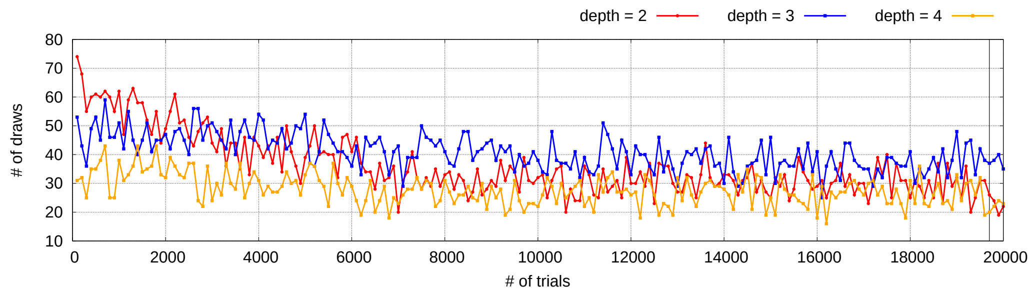
In figure 5, all the settings are same but the minimax agent changes its evaluation function from All_Direct to Attack+Block strategy. From the figures, the trends of win rate and draw rate during training are similar to those against minimax agent with All_Direct evaluation function. During training, Q-Learning gets lower win rate comparing to training against minimax with All_Direct evaluation function in figure 4a. However, what's surprising is that Q-Learning seems to get higher win rate after turning off exploration: Q-Learning wins about 80% of games against minimax with search depth 2, and about 50% of games against minimax with search depth 3 and 4. Remember minimax with Attack+Block evaluation function gets best performance when playing against other minimax agents. Here is an interesting phenomenon that this type of minimax only gets a win rate of 5% against Q-Learning agent in the last 400 competitions.

In figure 6a and figure 6b, we show training process on $6 * 6$ grid. The trends are similar to those in figure 4 and 5: as training processes, win rate of Q-Learning increases, while draw rate decreases. However, there are some interesting findings:

1. We see a sharp increase when turning off exploration probability in the last 400 competitions: the win rate of Q-Learning grows from approximately 40% to at most 80%. On the other hand, draw rate does not drop significantly: only from about 15% to 10%.
2. Comparing training on $4 * 4$ grid, it is harder for the agents to get draw. In figure 4b and 5b, we witness the draw rate descending from about 50% but in figure 6b the max draw rate does not exist 25%. This is reasonable since on larger grid, the agents have more flexibility to move so it is not as easy to get draws.
3. On $4 * 4$ grid, there is an obvious difference between minimax agent with search depth 2 and search depth 3 or 4: you can easily find the "gap" between red line and blue line in figure 4a and figure 5a. However, in figure 6a, the three lines mix together, which means there is



(a) Q-Learning agent win rate



(b) Game draw rate

Figure 5: Win rate and draw rate of Q-Learning against minimax agent with Attack+Block Evaluation Function on $4 * 4$ grid during learning process.($\epsilon = 0$ in last 400 competitions.)

no evident difference among them. This significant difference may be caused by the grid size. Minimax agent with search depth 3 or 4 is intuitively harder to beat on a smaller grid, since 3 or 4 steps on a $4 * 4$ grid is a lot (every agent can only take 7 moves). But on a $6 * 6$ grid, every agent could take 17 moves so minimax agents with search depth 2, 3, 4 perform similarly.

In conclusion, after training, Q-Learning agent could achieve at least 40% win rate and around 30% draw rate against minimax agent it is trained against on $4 * 4$ grids. Q-Learning agent could reach a win rate of about 60% with 10% draw rate on $6 * 6$ grids. This reveals that the Q-Learning agent could perform slightly better than the minimax agent it's trained against after nearly 20000 competitions.

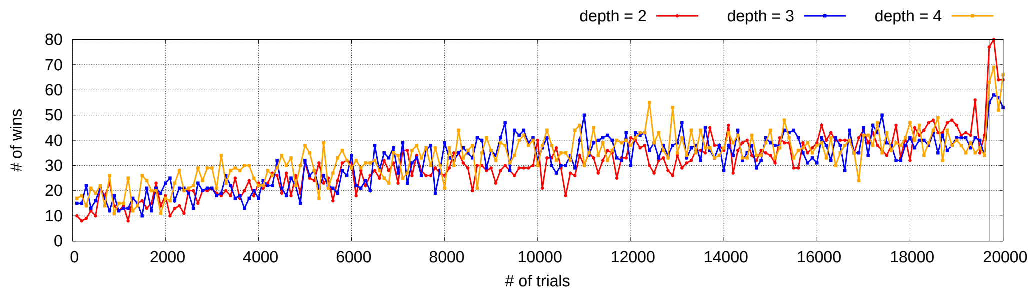
6.3 Future Work

In this section, we provide several future improvement directions.

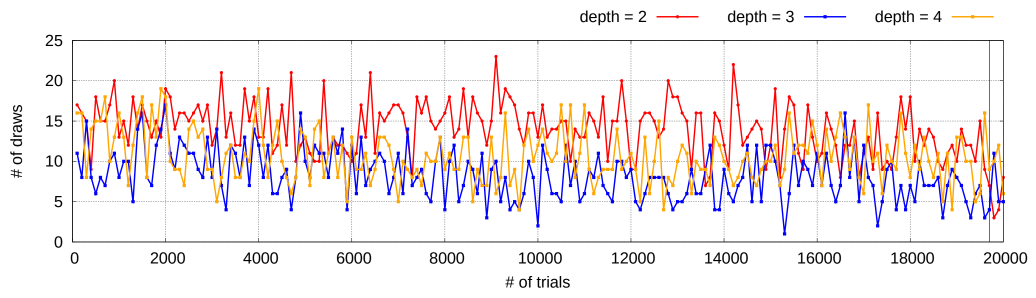
1. **Feature Extractor** We currently use an identity feature extractor. For each (state, action), there's only one feature: (board layout, action). Because different states have different board layout, this feature provides no generalization. With grid size increasing, the number of states will increase exponentially. Without generalization, Q-Learning won't get satisfying results because exploiting each state is very difficult. So better feature extractor is necessary for larger grids. We might extract features like Manhattan distance between two players and maximum distance the agent could move in four directions.

2. **Deep Q-Learning** Currently, we define Q as a linear function of features. However, with opponent's strategy becoming more complex, linear function of features is too simple to use. Therefore, in the next step, we could train a neural network to learn Q. Besides training while playing against another agent, we could also record competitions played by two arbitrary agents and feed into the neural network to train.

3. **Larger Board** As presented in the last section, we only trained Q-Learning agent on $4 * 4$ and $6 * 6$ grids. Because of computation resource limitation, training on larger grids would take much more time. For future work, we could try larger boards and compare the result.



(a) Q-Learning win rate



(b) Game draw rate

Figure 6: Win rate and draw rate of Q-Learning against minimax agent with Attack+Block Evaluation Function on $6 * 6$ grid during learning process.($\epsilon = 0$ in last 400 competitions.)

7 Conclusion

In the scope of this project, we implemented an AI agent for game 'Light Rider' provided by *Riddles.io*. We used both adversarial game theory methods as well as reinforcement learning algorithms. We show that evaluation function intending to constraint the opponent significantly increases win rate. Linear Q-Learning algorithm performs no worse than the minimax agent it's trained against. In the future, we could try more algorithms like Monte Carlo Tree Search, and non-linear function approximation for TD-Learning and Q-Learning to refine the project and generate more complicated tactics.

References

[1] Riddles.io. URL: `https://playground.riddles.io/competitions/light-riders`.

[2] Benoit, Z., Michael, S. & Mo, T. (2017) Stanford CS221 Course Project *An Intelligence Snake in Python* `http://web.stanford.edu/class/cs221/2018/restricted/posters/rschoenh/poster.pdf`

[3] Felix, C., Sebastien, D. & Sebastien, L. (2017) Stanford CS221 Course Project *Mutliplayer Snake AI* URL: `https://sds-dubois.github.io/2017/01/03/Multiplayer-Snake-AI.html`.