AI FOR PAPER.IO - SOLVING ADVERSARIAL GAMES WITH IMPERFECT INFORMATION

Junshen Kevin Chen
Stanford University
jkc1@stanford.edu

ABSTRACT

This project explores ways to solve an adversarial game in which agents do not have perfect information on the game state, and construct adversarial agents (minimax, expectimax) and reinforcement learning agents (TD learning), and evaluate their performance to gain insight on challenges to solve this type of games. By carefully modeling the agents with observation and memory, agents are able to take good actions with limited knowledge on the adversary's state.
Code: https://github.com/CniveK/ai-for-paper-io

1 Motivation and Objective

Currently, many of the popular adversarial games solved by AI agents are games of perfect information, such as Tic-Tac-Toe, Pacman, Go, chess, etc. They are also commonly present in AI courses, as example to teach introductory algorithms. On the other hand, popular adversarial games of imperfect information commonly have a randomness generated by nature, such as Mahjong (receiving a hand from a shuffled pile). Solving these games commonly involves taking the randomness of the game as part the algorithm.

This project aims to develop a simplistic model to solve adversarial games that are perfectly deterministic, meaning that there no random element is present, while simultaneously being imperfect in information, where the agent does not directly observe the entire game state, but only part of it, to get insight on how far we can go with a simple model.

2 Relevant Work

One very relevant work to this project is Kroer and Sandholm's [1] model of "limited lookahead", in which the players' strategy profiles are formalized to establish a heuristic, in order to reason about the game's mechanics and determine the possible action of the opponent given their limited state, by assuming the opponent always plays a Nash equilibrium strategy, for any general game. This paper also mathematically shows the upper bound of performance of this approach.

Because encoding complex heuristics to enable agents to reason according to the game rules is outside of the time limitation of this project, we will not use this exact approach. However, we will take on Kroer and Sandholm's assumption that the adversary performs a rational, Nash equilibrium action, and attempt to have each agent speculate the opponent's state to a some degree of confidence.

Another relevant work is a previous CS 221 project by Jiang, He and Bai [2]. In this project, a similar arena game "Light Rider" is solved by modeling as an adversarial zero-sum game. Unlike Paper.io, Light Rider is a perfect information game. However, in this project I attempt similar methodology and techniques, to attempt to explain why vanilla algorithms do not work under imperfect information, and explore ways to solve this problem.

3 Game Mechanics and Modeling

3.1 Mechanics and win conditions

The game of choice for this project is Paper.io [3] (http://paper-io.com/). We will model this game as an simultaneous, adversarial zero-sum game, and limit the number of agents to 2: a max agent and a min agent.

An agent will start at a certain location, with some territory surrounding it. At each tick, it moves towards an adjacent cell of any of the four direction. While leaving its own territory, it leaves a trail. When the agent finishes exploring outwards and return to its own territory, it completes a loop, and the area enclosed by its trail becomes its territory. To win the game, the agent:

- Eliminate the adversary by stepping on its trail while it is exploring, or
- Obtain and maintain the larger territory area before the game ends.

I built a simulator to run games between agents (including the human agent) to evaluate their performance and train the reinforcement learning agent. Due to the time and computation budget limitation, we define an arena smaller in size than the original game to speed up training. The following encodes a game play in progress.

3.2 Game State and Actions

```
   000 001 002 003 004 005
00  x   x   x       [X]       00
01  x   x   x   .x. .x.       01
02  x  .o. .o. [O]            02
03     .o.       o   o   o     03
04     .o. .o.   o   o   o     04
05               o   o   o     05
   000 001 002 003 004 005
```

Figure 1: Example game state in a 6*6 arena

Where `[X]` denotes the position of the max agent, `[O]` denotes the position of the min agent. Their territories are `x` and `o`, and their trails are `.x.` and `.o.`, respectively.

At this state, the min agent `[O]` could either:

- Move down, completing the loop, and gain 8 cells of territory, or
- Move up, step on the adversary's trail, win instantly

Denote a game state as a tuple $(Position, Territory, Trail)$, where:

- $Position(agent) \rightarrow \mathbf{R}^2$ maps agent to a position on the map
- $Territory(agent) \rightarrow set(\mathbf{R}^2)$ maps agent to a set of positions on the map
- $Trail(agent) \rightarrow set(\mathbf{R}^2)$ maps agent to a set of positions on the map

3.3 Stateful Agents of Imperfect Information

An agent only observes the cells within a set radius around it (**"observation"**). Continuing with the example, if we set the vision radius to 1, then each agent only observes the cells whose row and column index are 1 away from its position.

```
   000 001 002 003 004 005
00                            00
01          x   .x. .x.       01
02         .o. [O]            02
03              o   o         03
04                            04
05                            05
   000 001 002 003 004 005
```

Figure 2: Example observation by min agent `[O]`

The agent then must make a decision on which action to take based on this information. This presents challenge in algorithms such as minimax, whose premise is assuming the adversary takes the action to minimize / maximize utility, since the adversary's state is now not directly observed.

We attempt to solve this problem by maintaining state (**"memory"**) at the agents. We will use the same notation for the agent's known game state, but only populate it with information known to the agent. Specifically, update the agent's state at each move, where it obtains an observation:

Algorithm 1: Updating agent memory

Data: Observation: a partial arena observed around the agent
Update agent position;
for *cells within vision radius* **do**
  Overwrite corresponding cell in Trail, Territory sets with observed cell;
end
for *cells outside vision radius but in memory* **do**
  if *cell contains information on adversary* **then**
    Decrease confidence of the cell;
  end
end

Now, combined with observation at time t, and memory propagated and modified since time 0, the agent keeps a working version of the arena:

```
    000 001 002 003 004 005
00                          00
01          x  .x. .x.      01
02      .o. .o. [O]         02
03      .o.     o   o   o   03
04      .o. .o. o   o   o   04
05              o   o   o   05
    000 001 002 003 004 005
```

Figure 3: Example combined observation and memory of agent [O] in a 6*6 arena

Finally, the agent speculates where the adversary is, if it is not directly observed, by assuming the adversary is the same agent type as itself **("belief")**. For example, if the agent is last seen in row 1, column 4, and the agent is a minimax agent, then it is likely that the adversary is attempting to move up and close the loop, putting it at position row 0, column 4, as denoted by `?X?`

```
    000 001 002 003 004 005
00                  ?X?     00
01          x  .x. .x.      01
02      .o. .o. [O]         02
03      .o.     o   o   o   03
04      .o. .o. o   o   o   04
05              o   o   o   05
    000 001 002 003 004 005
```

Figure 4: Example combined observation and memory of agent [O] in a 6*6 arena

See following sections on how to efficiently make this speculation. Now we have some information on the adversary, we may proceed to implementing algorithms to solve the game.

3.4 Utility

As we would model a zero-sum game, we would define a tie game as having utility 0. However, since there are two ways to win the game, we will define utility as such:

$$Util(s) = \begin{cases} Territory(MaxAgent) - Territory(MinAgent) & IsTimeout(s) \\ ArenaSize & otherwise \end{cases}$$

Since $ArenaSize$ is the most territory any agent can possibly acquire in the game, in order to avoid the uninteresting strategy where agents build up territories and avoid each other at all time, having highest possible territory utility as utility for winning by elimination encourages agents to actively seek out ways to eliminate the opponent.

On the other hand, we do not want to model utility as the classic zero sum game as $\{-\infty, \infty\}$ as this would lead to agents stop attempting to build up territory, and go for each other directly. I find that awarding max territory for elimination is a good balance for a mixed strategy between builder and attacker.

Finally, let us define winner as:

$$winner(s) = \begin{cases} MaxAgent & Util(s) > 0 \\ MinAgent & Util(s) < 0 \\ None & otherwise \end{cases}$$

4 Adversarial Approach

4.1 Modeling the minimax agent

I implemented a classic minimax agent, with limited depth search and alpha-beta pruning using the known state to each agent. In a minimax setting, the agent chooses an action assuming the adversary has perfect information on itself,, and assumes that the adversary always try to take the overall utility to the opposite extrema.

$$V_{minimax}(s, d) = \begin{cases} Utility(s) & IsEnd(s) \\ Eval(s) & d = 0 \\ \max_{a \in Actions(s)} V_{minimax}(Succ(s, a), d) & Player(s) = \text{a}gent \\ \min_{a \in Actions(s)} V_{minimax}(Succ(s, a), d - 1) & Player(s) = opp \end{cases}$$

$$action = \arg \max_{a \in Actions(s)} V_{minimax}(Succ(s, a), d)$$

We may categorize this minimax agent as "conservative", as it assumes that the adversary as perfect information on itself, and estimates a worst-case utility for the agent. However, this is not the case in our setting, in which the adversary does not have perfect information on the agent itself, and therefore the agent conservatively makes a decision.

4.2 Updating adversary position

Suppose we start searching down the minimax game tree as a max agent, the first layer is a single max node, and in the second layer are 4 min nodes, and so on. As stated, we assume the adversary is also a minimax agent and seeks to minimize utility or state value, then we could take the minimax value of the selected branch in the game tree, and use the action of the second layer in that branch to be the adversary's action, eliminating the need to re-search the game tree to update the agent's belief.

4.3 Evaluation functions

Since we do not have the computation budget to search the game tree to an end state, and therefore we define evaluation functions to approximate the utility of the state at a certain depth of the game tree search.

A simple builder agent:

$$Eval_{builder}(state, agent) = |Territory(agent)|$$

A builder agent that favors states where it is considered "safe" if it can close the loop and return to safety (i.e. having no trail) before the adversary can step on its trail:

$$\begin{aligned} &Eval_{safe\_builder}(state, agent) \\ =&\lambda * |Territory(agent)| + (1 - \lambda)(OpponentDistToTrail - DistToCompletingLoop) \end{aligned}$$

Where $\lambda \in (0, 1)$ is a hyper-parameter to be tuned to favor either exploring or defending.

Even thought neither evaluation function explicitly encourages the agent to eliminate the adversary by stepping on its trail, keeping the magnitude of utility of an end game not less than the magnitude of the evaluation function implicitly

allows the agent to take the opportunity to pursue attacking when it sees a chance, more specifically, per the definition of utility:

$$\forall s : IsEnd(s), |Utility(s)| \geq |Eval(s)|$$

As expected, we see a large proportion of games end with agents eliminating each other using this setting. See "Test Results" section for evaluating agents against each other.

4.4 Alpha-beta pruning

With alpha-beta pruning, we only explore down the game tree if the node is within the lower bound α on value on max nodes and within the upper bound β on value on min nodes.

Table 1: Mean Search Time per Action (ms)

Search depth	1	2	3	4	5	6
Without alpha-beta	0.373	4.01	37.97	544.71	6245.32	timeout
With alpha-beta	0.376	3.46	17.19	141.20	982.22	4451.87

Alpha-beta pruning is able to reduce computation time dramatically, with higher depth search benefitting more from alpha-beta pruning. This enables the minimax agent to achieve a search depth of 6 while maintaining playability of the game.

4.5 Modeling the expectimax agent

An expectimax agent is built similarly, with the only difference being the agent assumes the adversary moves randomly, and without alpha-beta pruning.

$$V_{expectimax}(s,d) = \begin{cases} Utility(s) & IsEnd(s) \\ Eval(s) & d = 0 \\ \max_{a \in Actions(s)} V_{expectimax}(Succ(s,a),d) & Player(s) = \text{agent} \\ \frac{1}{|Actions(s)|} \sum_{a \in Actions(s)} V_{expectimax}(Succ(s,a),d-1) & Player(s) = \textit{opp} \end{cases}$$

$$action = \arg \max_{a \in Actions(s)} V_{expectimax}(Succ(s,a),d)$$

5 Reinforcement Learning

5.1 Modeling the TD learning agent

For reinforcement learning, I chose temporal difference (TD) learning, with extracted state features $\phi(s)$ and a linear weight vector w, and constructed values of the state as a linear function of weight and feature values.

$$V(s;w) = \phi(s) \cdot w$$

A weight vector is randomly initialized, then trained by playing against a minimax agent and generating episode data for learning. The generated data is episodes of states leading up to an end game. Then, perform stochastic gradient descent to update the weight vector:

$$w =:= w - \eta(V(s;w) - (Reward(s) + \gamma V(s';w)))\phi(s)$$

Where $\eta = 0.02$ is the learning rate, $\gamma = 0.7$ is the discount factor, and reward is 0 for all states except the end state, which is set to the utility of the game:

$$Reward(s) = \begin{cases} Util(s) & IsEnd(s) \\ 0 & otherwise \end{cases}$$

Then, in the decision making, we use an epsilon-greedy strategy to have the agent explore via a random action of probability $\epsilon = 0.02$, while exploiting the highest value state otherwise.

$$action = \begin{cases} Random(Actions(s)) & Random(0,1) < \epsilon \\ \arg\max_{a \in Actions(s)} w \cdot \phi(Succ(s,a)) & otherwise \end{cases}$$

5.2 TD learning over Q learning: symmetric actions

For this particular game I favored TD learning over Q learning because of the symmetric nature of actions. Consider a state feature "distance between two agents", and the following state:

```
    000 001 002 003 004 005
00                          00
01                          01
02      [X]         [O]     02
03                          03
04                          04
05                          05
    000 001 002 003 004 005
```

If the optimal action is to move away from each other, then for agent `[X]`, it would be moving left, and for agent `[O]` it would be moving right. However, the feature "distance between agents" is equal to 3 for both agents, and therefore a linear weight for this feature could not enable these two agents to move towards an opposite direction under Q-learning, where the value of the state is dependent on a set value of action:

$$V(s) = \max_{a \in Actions(s)} w \cdot \phi(s,a)$$

Since action "left" for agent `[X]` and action "right" for agent `[O]` have the same value, we cannot encode this information. However, for TD learning, the value of the successor state does not depend on the concrete value of the action, therefore enabling an agent to always go towards an optimal state:

$$a = \arg \max_{a \in Actions(s)} w \cdot \phi(Succ(s,a))$$

5.3 State features

Features $\phi(s)$ extracted from the states are:

1. Distance from agent to its own territory
2. Distance from agent to adversary 's trail
3. Distance from adversary to agent's trail
4. Distance from adversary to adversary's territory
5. Agent's territory size
6. Agent's trail size
7. Agent's distance from the closes arena wall ("mobility")
8. If agent is the winner
9. If adversary is the winner
10. Number of remaining ticks

5.4 Updating adversary position

For the TD-agent, since state values are a linear combination of weights and features, it is computationally inexpensive. A TD-agent would use the same weight vector and observed state feature to determine an action for the adversary, then update the adversary's position in the agent's belief.

5.5 Training Performance

After weight initialization, the TD agent is trained by playing against minimax agents in different configurations. After each game play, the episode is saved to perform gradient descent as described above, then a new game is started. Training is stopped after the TD agent is able to consistently beat the minimax agent by elimination, or after 25000 episodes.

The following plots the expected utility and win/loss of training different agents.

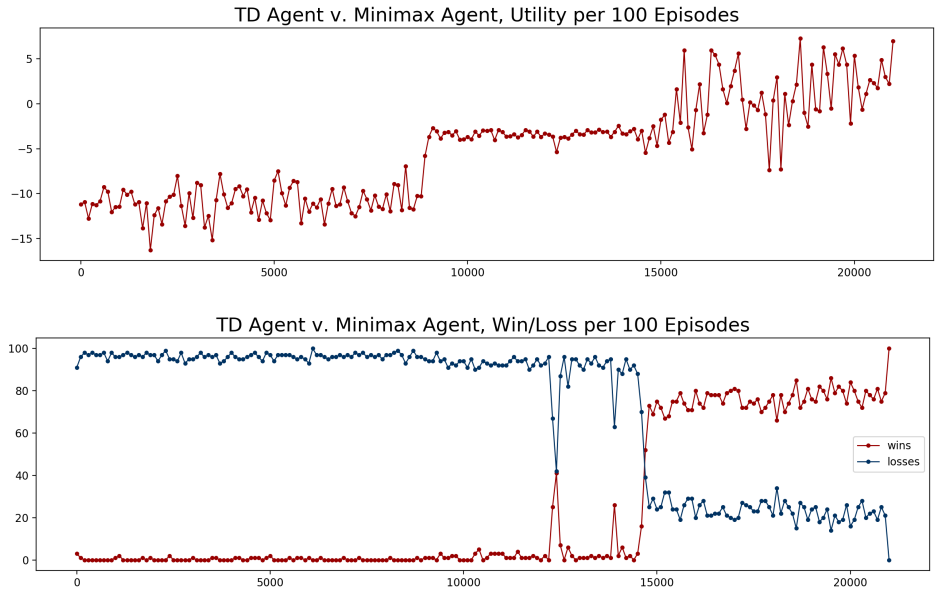

Figure 5: Training TD agent against minimax agent - depth 1

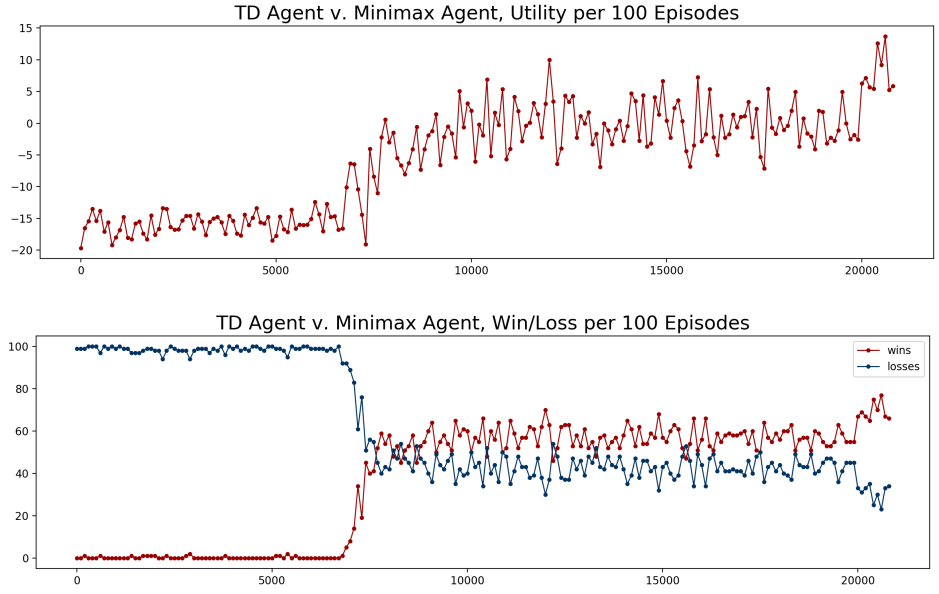

Figure 6: Training TD agent against minimax agent - depth 2

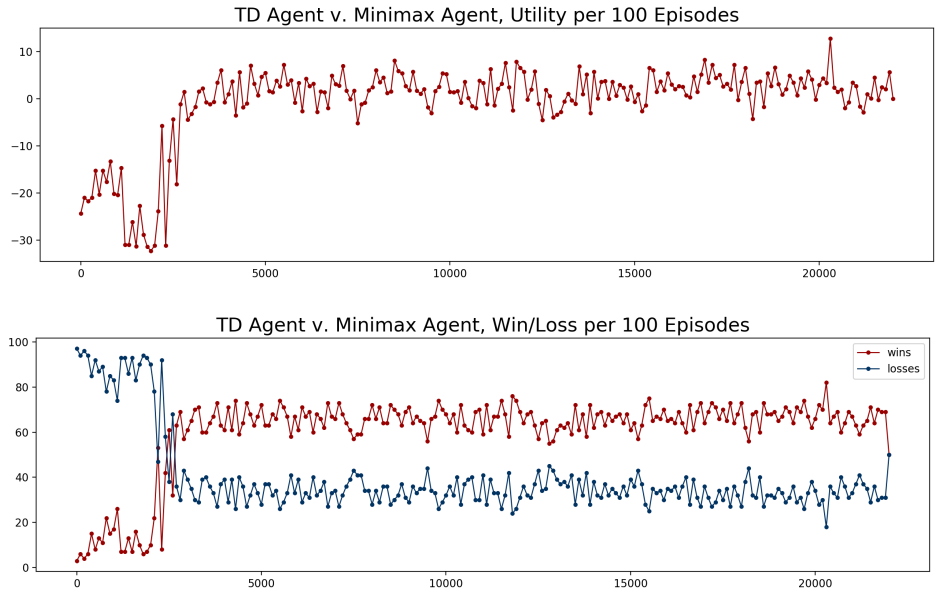

Figure 7: Training TD agent against minimax agent - depth 3

We may obtain some interesting observations from theses training behaviors:

1. All TD agents are initially moves randomly and loses all games, then at some point it learns to stop losing by safely building up its territory while avoiding the adversary, resulting in roughly equal win/loss. Then, the agent exploits the safe strategy until epsilon-greedy allows it to explore to a state where it would eliminate the adversary for a much higher utility, then it would start converging to the attacker behavior.
2. Observing the expected utility behavior towards the end of training, we notice that there are two critical points: 1) the agent learns to avoid the adversary, and build up territory safely, but still loses most games, 2) the agent learns to attack adversary to win most games. This leads to the conclusion that the reinforcement agent is able to learn that in order to achieve the most utility, it must expose itself to some more danger, as shown in the plot where expected utility fluctuates more towards the end, as the agent either kills or gets killed.

5.6 Generalizability

The TD-agent generalizes surprisingly well to a change in game setting such as arena size and vision radius, while adversarial agent remains the same (but also receives change in vision radius).

For example, we train a TD agent within a 7 by 7 arena with vision radius 1, against a depth 2 minimax agent, then change the parameter to the arena and observe that the TD agent behaves well when vision radius increases or arena size decreases, and is able to win almost half as many games when the game becomes perfect information.

Table 2: Mean utility; max agent: TD, min agent: minimax-2

Arena Parameter	7 by 7, vision 1 (original)	7 by 7, vision 2	7 by 7, vision 10	6 by 6, vision 1
Expected utility	11.52	10.77	-3.73	4.2

6 Test Results and Evaluation

Finally, we may evaluate the performance of different types of agents by simulating them playing against each other, as well as a baseline and an oracle.

Random: a **baseline**, an agent that moves randomly, while avoiding stepping on its own trail.

Minimax-God: a **reasonable oracle**. there is no trivial possible implementation of an oracle. We will use a depth-3 minimax agent with a "god view" of perfect information, to give it an unfair advantage over the others. This should be the upper bound of agent performance.

Minimax-2-Pure: minimax agent of depth 2 search, using the pure builder evaluation function.

Minimax-3-Pure: minimax agent of depth 3 search, using the pure builder evaluation function.

Minimax-3-Safe: minimax agent of depth 3 search, using the safe builder evaluation function.

Expectimax-2-Pure: expectimax agent of depth 2 search, using the pure builder evaluation function.

Expectimax-2-Safe: expectimax agent of depth 2 search, using the safe builder evaluation function.

TD-2: TD agent trained against a depth 2 minimax agent.

I ran 300 games for each possible match-up, in a 7 by 7 arena with vision radius 2, timing out each game at 70 ticks. The following tables show the mean utility and win-loss for each match-up, with the row agent being the max agent, and the column agent being the min agent.

Table 3: Mean Utility between Agents

	Random	God	M2Pure	M3Pure	M3Safe	E2Pure	E2Safe	TD
Random	-0.492	-	-	-	-	-	-	-
Minimax-God	+44.75	-1.59	-	-	-	-	-	-
Minimax-2-Pure	+38.90	+0.80	+5.33	-	-	-	-	-
Minimax-3-Pure	+38.79	-0.84	-9.97	+0.24	-	-	-	-
Minimax-3-Safe	+42.56	-7.16	-6.69	-21.73	-2.43	-	-	-
Expectimax-2-Pure	+38.35	-14.54	-11.45	-6.14	-7.58	-2.44	-	-
Expectimax-2-Safe	+38.37	-19.16	-10.96	-12.12	-12.24	+1.74	0.60	-
TD-2	+39.20	+4.77	+5.96	+18.00	+16.42	+16.13	+12.31	+1.24

Table 4: Win-loss between Agents

	Random	God	M2Pure	M3Pure	M3Safe	E2Pure	E2Safe	TD
Random	142-157	-	-	-	-	-	-	-
Minimax-God	295-3	146-153	-	-	-	-	-	-
Minimax-2-Pure	293-7	149-150	164-131	-	-	-	-	-
Minimax-3-Pure	279-20	150-150	121-178	150-150	-	-	-	-
Minimax-3-Safe	290-10	130-169	133-164	83-217	154-146	-	-	-
Expectimax-2-Pure	283-17	104-196	113-187	130-170	125-175	142-157	-	-
Expectimax-2-Safe	279-20	87-213	115-185	112-188	111-188	152-147	151-147	-
TD-2	279-21	170-127	170-128	209-91	204-96	208-89	191-107	150-148

6.1 Expected performance

All AI agents are able to consistently defeat the baseline random agent by eliminating it, resulting in a high win-loss ratio and a very high utility.

God-mode minimax agent **does not significantly outperform** its limited-vision counterpart (depth-3 pure minimax) despite having much better vision. This is due to good modeling agents' memory, observation, and belief that drastically nullifies the unfair advantage of perfect information. This shows that, under the limitation of minimax algorithm, our approach of modeling a stateful agent does a good job in approximating the unknown state.

Minimax agents slightly outperforms expectimax agents.

TD agent outperforms the minimax agent that it is trained against, meaning that it succeeds in learning a dominant strategy in this specific setting.

6.2 Unexpected performance

The more complex evaluation function, "safe builder" does not significantly increase agent performance. In fact, for depth-3 minimax and depth-2 expectimax, it consistently result in the agent losing by being eliminated. This is

possibly due to that the function's provides misleading information, or that it is incompatible with higher depth search, resulting in a less-than-optimal strategy.

TD agent's ability to generalize, and defeat other agents **exceeded expectation**. As shown in the tables, TD agent is able to consistently defeat not only the agent it is trained against, but also a higher depth minimax agent and expectimax agents with different evaluation functions. This further shows that the TD agent succeeded in learning a dominant strategy in playing within this particular arena setting.

7 Conclusion

In this project, I implemented numerous AI agents for the game Paper.io, with an adversarial approach with minimax and expectimax agents, as well as a reinforcement learning approach with a TD learning agent.

By carefully modeling each agents to possess a merged state of observation, memory, and belief, I find that it is possible to create agents of imperfect information that perform reasonably well with only classic algorithms, without overly complex human modeling to enable agents to reason about the game itself.

8 Future Work

8.1 Multi-agent adversarial game

Rework the arena to simulation among multiple agents, with each agent trying to maximize utility while assuming all other agents attempts to minimize utility.

8.2 RL with multi-layer network

Currently, the TD agent learns linear weight and evaluates the state as a function of linear combination of features. This may not be expressive enough to represent the game state as the number of feature grows, or as arena size or number of agents grow.

As a future work, the value of the state can be calculated as a multi-layer perceptrons (neural net), with activation functions at each layer, and learned via gradient descent and back-propagation.

8.3 MCTS agent

Monte Carlo Tree Search (MCTS) is a simple yet powerful algorithm to enable agents learn the game by taking random actions and search through the game tree while self playing. This could be helpful in better generalizing the game without humanly crafted features.

8.4 Decaying memory confidence

Currently, the agents state contains a "confidence" value for each cell that decreases at since every tick that cell is not observed. However, this information is currently unused, and all cells are treated as equally confident. As a future work, develop a way to utilize this decaying memory confidence into the search algorithms to better model agents' memory.

References

[1] Christian Kroer and Tuomas Sandholm: Limited Lookahead in Imperfect-Information Games, in Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015), https://www.ijcai.org/Proceedings/15/Papers/087.pdf

[2] Wantong Jiang, Yipeng He, Di Bai: AI Agent for Light Rider, CS221 Final Project, https://github.com/jwtty/CS-221-Project/blob/master/reports/final.pdf

[3] Paper.io, http://paper-io.com/