AI for Paper.io - CS221 Project Progress Report

SUNet ID: jkc1
Name: Junshen Kevin Chen

Progress Overview

During the previous few weeks, I have progressed with the project by:

1. Modifying the game simulator to be accommodate a multiple agent (both human and AI) environment for the game;
2. Adding mechanism to provide imperfect information (an observable radius for each agent around itself);
3. Clearly defining game states and constructing such API for each agent's decision making process;
4. Implementing a simple Minimax agent with 4 different strategies to compare their performance.

1 Game Mechanics

Paper.io is played by multiple agents on a map. Each agent has a limited field of vision centered around itself, and can only observe other agents' and their area when they are close by. Furthermore, there is no information on the absolute position the agent is on the map.

At each tick, the game receives an action from the agent (input for game / output of agent). All agent must move to a different location at each tick.

On the way of exploring outwards, an agent leaves a trail, any other agent (or itself) touching the trail before the agent completes the loop results in the agent dying.

2 Modeling the Game in Detail

This section will go over how we model the game, and how information is provided to each agent to simulate a human-like gameplay environment, in the level of detail omitted from the proposal document.

2.1 Simulator

My simulator incorporates much of the logic from `BlocklyIO` implemented by `theKidOfArcrania` (github: `https://github.com/theKidOfArcrania/BlocklyIO`), which is a clone of the original Paper.io without the multiplayer aspect in Javascript. Because this implementation is not optimized for training AI agents and definining custom rules, I have opted to write my own simulator in Python.

2.2 Objective

The objective of Paper.io is to occupy as much territory as possible. As an .io game, it does not have a start or an end, as players come and go in real time. However, for the purpose of training AI agents, this project ends the game if:

1. All agents have been eliminated except one (winner);
2. Or, after `MAX_TICK` amount of time, there are multiple surviving agents; the one with the highest territory area wins.

2.3 Game State

2.3.1 Game Manager

The game manager handles the overall mechanics of the game. It contains information that defines the gameplay environment for all Agents, such as:

- `MAX_TICK`, after which the game ends
- `MAP_SIZE`, a 2-tuple defining a rectangle of the arena in which the agents travel and interact
- `VISION_RADIUS`, an integer that defines a box of visible area around each agent, which is a `2 * VISION_RADIUS + 1` square

The game manager also maintains game progress, the percentage of area that each agent occupies at the current time.

2.3.2 Game State

The full game state is only observable to the game manager (and a god-mode agent, which would act as an oracle for agents of each model, to be implemented). At each tick, the game state could be represented as such (we will use this example for the rest of the report):

```
X  X  X  X  X  X  X  X  X  X
X  a  a  a  a  .  .  .  .  X
X  a  a  a  *a *a A  .  .  X
X  .  .  .  .  .  .  .  .  X
X  .  .  B  *b b  b  b  b  X
X  .  .  .  .  b  b  b  b  X
X  .  .  .  .  b  b  b  b  X
X  X  X  X  X  X  X  X  X  X
```

The above denotes a game played in a 6 * 8 map, in which agent A occupies some of the top left territories (denoted `a`) and agent B occupies some of the bottom right territories (denoted `b`). Both agents are extending outwards from their respective territories, their trails are denoted as `*a` and `*b` respectively. `X` denotes edge walls.

All of the above information is stored in the game manager and not visible to any agent.

2.4 Agents of Incomplete information

At each time step (tick), the game manager asks each agent for a direction to move towards by calling a function `get_move`, and feeding each agent with a state within their observable region.

For this example, the observable radius for the agents is 2, then each of the agents has an observable area of a 5*5 square around itself. `observation`, a parameter to `get_move`, it is represented as a dictionary where the keys are relative position to the agent itself, and the values are the content in that location, if any.

Each agent also receives the current game progress `progress` in a dictionary. This information is currently unused, except for each agent to know what other agents are on the map.

For example, for agent A at the current game state:

```
observation = {
    (-2,-2): 'X', (-2,-1): 'X', (-2, 0): 'X',
    (-2, 1): 'X', (-2, 2): 'X',              # walls
    (-1,-2): 'a',                            # own territory, visible
    ( 0,-2): '*a', ( 0,-1): '*a',            # own trail
    ( 2,-2): '*b',                            # adversary trail
    ( 2,-1): 'b', ( 2, 0): 'b',
    ( 2, 1): 'b', ( 2, 2): 'b'                # adversary territory
}
progress = {
    'A': 0.14583,
    'B': 0.25,
}
```

Finally, to better model agent like a human, each agent maintains a "memory" on where, relative to the current position, is the agent's own territory.

This closely simulates a human player mode of thinking where: "If I explore to the right, then I should probably eventually return to the left to complete my loop". This is a state owned by the agent, and updated as the agent makes a move decision and observes its own territory.

3 Preliminary Implementation and Evaluation

3.1 Evaluation Function

Because it is not possible to search the game tree until the game terminates (computation budget constraint, and the incomplete information barrier), I define an evaluation function to evaluate the observable state.

$$\begin{aligned}
&Eval(observation, progress, agent) \\
=&a * DistanceToCloseLoop(observation, agent) \\
+&b * DistanceToAdversaryTrail(observation, agent) \\
-&c * AdversaryDistanceToOwnTrail(observation, agent) \\
+&d * OccupiedTerritory(progress, agent)
\end{aligned} \tag{1}$$

And a utility function if the game ends:

$$Util(progress) = \begin{cases} inf & winner = A \\ -inf & winner = B \end{cases}$$

3.2 Minimax Agent

By tweaking the corresponding weights in the utility function, we can construct interesting agents with different strategies:

1. "Naive" (Baseline), $a = b = c = d = 0$,
   Moves randomly, just avoid your own trail.
2. "Builder", $a = 1, d = 1, c = d = 0$,
   Ignores adversary, focus on exploring and building territory.
3. "Attacker", $a = c = d = 0, b = 1$,
   If it sees an adversary trail, run directly to it, otherwise fall back to naive strategy.
4. "Jack of Some Trades", $a = b = c = d = 1$,
   A combination of all of above.

The minimax agent searches the game tree, assuming the adversary is a min agent and the game is zero-sum, then pick a move that maximizes / minimizes the state value / utility.

$$V(o, p, a, d) = \begin{cases} Utility(p) & IsEnd \\ Eval(o, p, a) & d \leq 0 \\ max_{dir} V(*Succ(o, p, a, dir), d-1) & a = A \\ min_{dir} V(*Succ(o, p, a, dir), d-1) & a = B \end{cases}$$

Where $o = observation, p = progress, a = agent, d = 2 * search\_depth$.

I evaluate these agents by running them against each other, each for 50 trials. The following are some preliminary win-loss statistics. Where each cell denotes (win of the column agent / win of the row agent).

-	naive	builder	attacker	JoST
naive	25/25	48/2	40/10	45/5
builder	-	23/27	10/40	20/30
attacker	-	-	21/29	46/4
JoST	-	-	-	28/22

Immediately, we can notice a problem, where Jack of Some Trade performs even worse than Builder. Let's discuss this in the next section.

4 Future Work

4.1 Minimax Agent

I plan to develop more hand-coded features into the state evaluation function (or, if possible, learn them), such as "territory increase if loop closed" and "number of remaining ticks", etc.

A prevailing problem with the existing agent is that when the adversary's trail is visible but the adversary itself is not, the agent does not attempt to estimate where it is, or how dangerous the current situation is, but naively fall back to a random agent. I plan to solve this issue by adding another state in the agent where it tracks the "last seen position" of the adversary.

4.2 Expectimax Agent

Using the same set-up as the minimax agent, an expectimax agent could easily be implemented by simply averaging all possible moves from the opponent.

$$V(o, p, a, d) = \begin{cases} Utility(p) & IsEnd \\ Eval(o, p, a) & d \leq 0 \\ max_{dir} V(*Succ(o, p, a, dir), d-1) & a = A \\ \frac{1}{4} \sum_{dir} V(*Succ(o, p, a, dir), d-1) & a \neq A \end{cases}$$

4.3 Reinforcement Learning

Finally, simulate the game with many episodes and train a model to yield an optimal policy, using Q-learning, where we define some state feature vector and update it with experience as such:

$$\hat{Q}_{opt}(s, a) := (1 - \alpha)\hat{Q}_{opt}(s, a) + \alpha(reward(s, a) + \gamma \hat{V}_{opt}(succ(s, a)))$$

References

1. theKidOfArcrania, BlocklyIO, `https://github.com/theKidOfArcrania/BlocklyIO`
2. Paper.io `http://paper-io.com/`