ECS7002P - Lab Exercises - Week 2

Pommerman

(with answers)

2 Exercise 2 - AI Agents

All Pommerman AI agents must:

- Extend from the class Player.java
- Implement three methods from the base class:
  - **public Types.ACTIONS act(GameState gs)**: this is the most important method. It receives a copy of the current game state and must return an action to execute in the real game. The valid actions are defined in the enumerator ACTIONS, located at Types.java. Take a look at that data structure.
  - **public Player copy()**: this method creates a copy of this player.
  - **public int[] getMessage()**: this function is called for requesting a message from the player. Not in use for this assignment.

As an example, take a look at the code two very simple agents:

- DoNothingPlayer (src/players/DoNothingPlayer.java), which does, not surprisingly, nothing.
- RandomPlayer (src/players/RandomPlayer.java), which executes a random action at every step. The code of the *act()* method is as follows:

```java
public Types.ACTIONS act(GameState gs) {
    int actionIdx = random.nextInt(gs.nActions());
    return Types.ACTIONS.all().get(actionIdx);
}
```

  Take a look at the different methods involved in this call and make sure you understand what is each one of them doing.

The rest of the agents, more sophisticated than the previous two, are the following:

- OSLAPlayer: One Step Look Ahead. This agent tries each one of the possible actions from the current state, analysing the state reached after applying them. The agent takes the action that leads to the state that is considered to be best by an heuristic.
- SimplePlayer: This agent is a rule based system that analyses the current game state and decides an action without performing any search.
- RHEAPlayer: This agent implements a Rolling Horizon Evolutionary Algorithm.
- MCTSPlayer: This agent implements a Monte Carlo Tree Search algorithm.

All these agents use heuristics to evaluate a good a game state is. These heuristic classes are in src/players/heuristics/.

2.1 Using Forward Models

The OSLA, RHEA and MCTS agents use a Forward Model (FM). This FM is implemented in the GameState (object 'gs' in the *act()* method) and there are two key functions:

- **GameState copy()**: It creates an exact copy of the game state, which is returned by the method.
- **boolean next(Types.ACTIONS[])**: It advances the game state applying all actions received by parameter. The array must have the actions for all the players to be executed in a given step. Each player has an allocated index to insert actions for the next() call, as follows:

```
//Array of actions. One per player (size: 4)
Types.ACTIONS[] actionsAll = new Types.ACTIONS[4];
//Calculate the index where my action goes.
int myIndex = this.getPlayerID() - Types.TILETYPE.AGENT0.getKey();
actionsAll[myIndex] = ... //Here I put the action I want to simulate.
```

As an example, take a look at how the OSLA Agent fills an array in the method (*rollRnd(...)*). In that method, when *rndOpponentModel* is true, the actions assumed for all the other agents are random. If *rndOpponentModel* is false, the other agents will execute nothing (ACTION_STOP) when the state is rolled forward.

Take a look at that method and make sure you understand what's happening (ask if you don't!).

2.2 One Step Look Ahead (OSLA) agent

Now, take a look at the *act()* method of the OSLA agent. Follow the logic and make sure you understand what is going on. Can you answer the following questions? (ask if you can't):

1. What does the main loop in the method do?
   - Answer: Iterates through all possible actions, advances the state once with each one, evaluates the resulting states and picks the action which leads to the next best state.
2. Why, on the first line of the *for* loop, the game state is copied?
   - Answer: Each action needs to be applied from the same current state, therefore we need to make sure we don't alter the state by applying actions - all actions will be applied in a copy of the game state.
3. How is each future state being evaluated?
   - Answer: With the CustomHeuristic function.
4. What does the *noise* function do and why is this used?
   - Answer: Adds a small random value to each of the state values. Used to differentiate between states with the same value (the noise would mean they're no longer exactly the same and we can pick one at random between two states with the same value).
5. What does *rndOpponentModel* do? (you should know this from the previous exercise!).
   - Answer: It advances the game state by applying our given action, and random actions for all other players (if rndOpponentModel set to true, otherwise the other agents will do nothing in the simulations). The next() function requires one action for all players in the game and, while we know what our agent would do, we have to make assumptions about what the opponent would do (opponent modelling) - in this case, acting randomly or not doing anything.

An important part of this agent is the heuristic it uses to evaluate states (CustomHeuristic.java). Take a look at this class and try to understand how does it work.

The next step is to experiment with this agent. Pitch this agent versus other 3 agents (for instance, a random, do-nothing and simple agents) in Run.java, and try running OSLA with different configurations. Can you draw some conclusions on what configurations work better? Some things to try:

- Try with *rndOpponentModel* set to False and compare the results against when it's set to True.
- Change the weights and calculations of CustomHeuristic, so the states are valued differently.

2.3 Simple Agent

The simple agent is a purely heuristic controller, without using the Forward Model. The state of the game is analysed in the act() method of SimplePlayer.java, checking positions of objects and calculating shortest paths between tiles.

Continue with the following two exercises:

- Take a look at the different operations performed in this method, in order to understand how the different functions of the game state can be used.
- For the rest of the lab, create your own agent. You can use components and ideas from the SimpleAgent, but try to create a stronger one. Maybe you can try to think of ways of combining the heuristics of the Simple Agent with the use of the Forward Model.

3 Conclusion Exercise - Sharing your results

Run a tournament between 4 agents in FFA mode. One of the agents must be the player you built in your the previous exercise. The other 3 should be: SimpleAgent, OSLA and Random. The tournament should be run in at least 10 different levels with no less than 5 repetitions for each level. Add a comment to the relevant post in the forum indicating:

- The settings you have chosen for the tournament and the name of your agent.
- The results of the contest.