Game Theory Assignment 2 Report

Daniil Dvoryanov, B17-SE01

I. Task Description and My Interpretation

In this assignment we were asked to implement a behavioural model for the moose in a simple environment consisting of three fields and simple rules of competition.

A. What is the Problem Behind Choosing the Best Move?

We begin a round (between 2 moose) with field values of 1. This means that after the first move only one field will add to the scores of a player. This introduce competition into the game. Moreover, if two moose fight, not only do they both get the score of 0, but the field also decreases in the value. So fighting is worse for both moose than some form of cooperation.

But since we don't know which move the other player is going to choose, even if booth moose players want to be cooperative to their advantage, it is impossible without an external agreement between the two players.

But then, if we know for certain that the other player is going to cooperate, we will be better off just cheating and taking more resources. And the other player knows that. And we know that they know that we know. And so on. Therefore it is impossible to create cooperative (optimal for both players) strategy without some externalities (complete honesty and trust, survival (for now we are just talking about scores, not lives), etc.).

II. Considered Strategies

My approach to this task was to simply try various strategies and see which one wins most scores.

The strategies (players) that I tested are the following:

- **Random**: at each move choose the field at random
- **Greedy**: at each move choose the field with the highest score
- **Copycat**: at each move repeat the previous move of the other player.
- **Lingering**: the player stays on the same field as long as it has some payout (score). Once the field is depleted, it chooses field with the maximum payout.
- **Sequential**: the player cycles through fields starting with field A (A, B, C, A, B, C, ...).
- **Previous Field**: whatever the opponent's last move was, choose the previous field in the list (if opponent's last move was A, then choose C). Also, it does not stay in the same field for two moves: if the other player stays in the same field, it moves to the previous field. This is to resolve deadlocks between players with similar strategies.
- **Random Direction Move**: based on a "coin flip", either choose next field or previous field from the opponent's last move.

```
Tournament summary:
| Random direction | Random direction | A | B | C |
| C/0.00 pts       | C/0.00 pts       | 2 | 2 | 0 |
| A/0.00 pts       | A/0.00 pts       | 1 | 3 | 1 |
| B/0.00 pts       | B/0.00 pts       | 2 | 2 | 2 |
| A/3.81 pts       | C/3.81 pts       | 1 | 3 | 1 |
| A/2.31 pts       | B/4.53 pts       | 0 | 2 | 2 |
| A/0.00 pts       | C/3.81 pts       | 0 | 3 | 1 |
| B/4.53 pts       | A/0.00 pts       | 0 | 2 | 2 |
```

Fig. 1. Example of table output

III. Testing implementation

In order to test my players, I used several classes:

- `GamePlayer`: wrapper around Player that stores player's name (for visual purposes), score, number or tournaments won, and its previous move.
- `Environment`: stores the score for each field, and provides convenient methods for updating and retrieving field values.
- `Logger`: contains static methods useful for outputting and representing various types of information as `Strings`.
- `Parameters`: contains global parameters that can easily be changed in one place.
- `Round`: stores two players and Environment, and has methods to run a number of moves (specified in parameters) between the players.
- `Tournament`: initialized with a list of players, for every player creates a Round with all other players. Has method to run all the rounds and display the results.

These are the classes I use for testing my players (excluding inner (private) classes). Now let me describe some of them in depth for better understanding of testing implementation.

A. Logger

It turned out to be very useful piece of software, as better visualisation and description allows for quicker and easier results extrapolation.

The most important tool (format) of logging in this testing system are tables. I implemented general algorithm which takes an array of rows (list of String arrays) (all rows must be of equal length) and converts them to well-aligned table (see Figure 1).

As you can see, this really helps in understanding the interactions between various players.

B. Round

The main logic is contained in `nextMove` method. It works like this:

1) Get current field values from environment;
2) Get both players' last moves;
3) Call `move` on both players and store the output;
4) Based on the players' move, calculate their scores and update each player instance;

```
Final scores:
| Careful | Random | Greedy | Copycat | Sequential | Previous field | Random direction |
| 172.83  | 173.56 | 210.86 | 65.76   | 213.77     | 259.13         | 259.13           |

Number of won rounds:
| Careful | Random | Greedy | Copycat | Sequential | Previous field | Random direction |
| 4       | 1      | 4      | 0       | 4          | 4              | 4                |
```

Fig. 2. Results of the tournament between all players

5) Return the results in `MoveResults` instance (simple data class for storing resulting state after a single move).

Another important method of `Tournament` is `PlayRound`, which consecutively launches specified number of tournaments (default: 20), and outputs the final result in the form of a table (as in Figure 1). It also outputs the round scores for each player, and updates win counter for a winner (if any).

C. Tournament

This class on initialization creates `Round` between each pair of players (including the player itself). After that, we can launch all rounds sequentially by calling `playAll`. After all rounds are done, it outputs the score score table and table with number of wins for each player.

IV. Results and Scores

Now that we have seen the testing implementation and players description, let me tell you about some interesting findings and the results of the competition.

As I said before, I included in the tournament not only the rounds between a player and all other players, but also rounds between two players with the same strategy (Random vs. Random, Greedy vs. Greedy, etc.). This way, I can see how well the player performs against similar strategy player.

See Figure 2 for the results of the tournament between all of the implemented players. As you can see, although Copycat was a very good strategy in 2x2 games, here it is simply useless. Greedy performs very well, simply because with the given amount of information, it is always better to go for the highest score field. If the other player also goes there, than the next move greedy will choose another (or the same) field. In worst case (e.g. Greedy vs. Greedy) both players will have 0. It is an equivalent of Always Cheat from 2x2 games.

But, although Greedy has a very concise and logical strategy, it only comes 3rd in the tournament. First two places hold the ”directional” players. Why is that? This strategy is similar to greedy, as it finds a field that was free last move. But it does not always chase after the field with maximum score, it just picks a field with a non-zero score. So in a way, this is a more cooperative version of greedy (by the way, it ties with greedy but not at 0).

Random direction is similar to Previous Field, but it does not always find a not-affected field, it is more random.

V. Conclusion

This game is very different from 2x2 games of cheating/cooperating. There does not exist a cooperative strategy without prior agreement (and even than, you need to have mutual trust between each other for the strategy to work). There does, however, exist Always Cheat strategy, which never looses, but never tries to find common ground and often ties at 0 points.

The best strategy that I discovered simply uses the field unaffected by the previous move. It is indeed a more cooperative version of greedy, as it allows both players to benefit from the environment.