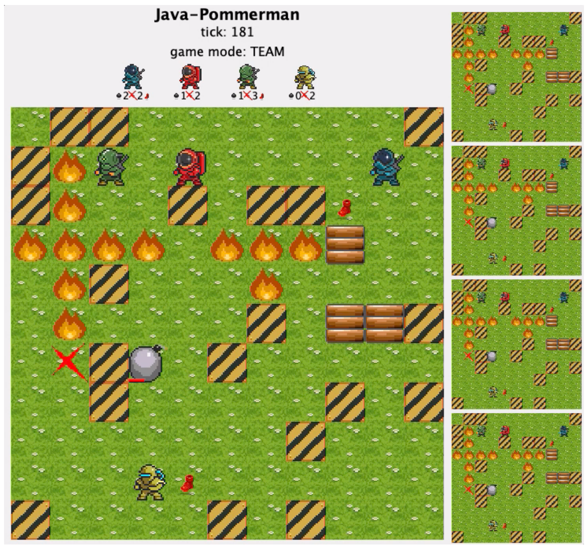
# ECS7002P - Lab Exercises
# Pommerman

8th October 2019

## Brief Game Description

*Pommerman* is a complex multi-player and partially observable game where agents try to be the last standing to win. This game poses very interesting challenges to AI, such as collaboration, learning and planning.

The game takes place in a randomly drawn $11 \times 11$ grid between 4 players. Players are placed in the 4 corners of the grid and the level is scattered with obstacles of two types: wooden and rigid boxes. The board is symmetric along the main diagonal. The game can be played with full or partial observability. The objective of the game is to be the last player alive at the end, or the last standing team. Players can place bombs that explode after 10 game ticks. These bombs, when exploded, generate flames in a cross-shape with size of (initially) 1 cell around the explosion point. These flames eliminate players, items and wooden boxes on collision, as well as explode other bombs. By default, players can't drop more than one bomb at a time. Bombs, as well as obstacles, can't be traversed. Wooden obstacles, when destroyed, may reveal pick-ups, initially hidden to the players. These items are *extra bombs* (adds 1 to the bombs ammo), *blast strength* (adds one to the explosion size) and *kick* (allows kicking a bomb in a straight line). To make sure games are finite, a limit of 800 game ticks is set. The play area is generated automatically using a random `seed`.



## Getting started

In order to get the framework installed and running, follow these instructions:

1. Clone the following repository using git in a Unix or Windows terminal:

   git clone `https://github.com/GAIGResearch/java-pommerman`

   You can also download the repository or check it out with SVN.

2. Set up your IDE to use the project. These instructions allow you to set it up with IntelliJ IDEA, but you can also use any another IDE you feel more comfortable with.

   (a) Open IntelliJ IDEA
   (b) Select 'Create New Project' and select 'Java'. Progress clicking on 'Next'.
   (c) Type in any project name you want and select, as project location, the 'java-pommerman' folder (the one containing a *src/* directory). Click on Finish (say 'yes' to overwrite the project folder if prompted).

3. Verify that *Pommerman* can execute in the IDE:
   (a) Click on File → Project Structure, and go to Libraries.
   (b) Click on the '+' symbol and select Java. A directory browsing window should appear.
   (c) Select the all the jar files in the directory 'lib/' and click on Open.
   (d) Click then on 'Ok' first to add the libraries to the project and then again to close the Project Structure window.
   (e) In the IDE, open the file src/Test.java (you can navigate files on the left panel - click on *1:Project* if the folder hierarchy is not displayed).
   (f) Click on menu Run → Run, and select *Test* in the pop up window. Alternatively, right click the file Test.java and select the option 'Run'.
   (g) It'll take a few seconds for the project to be compiled and then it will run a game with four AIs playing.

## Exercises

The objective of this lab is to familiarise yourself with *Pommerman* and process the information that is given to you in the different methods that you can implement.

The first step is to inform yourself about the game - how does it work, what kind of objects and actions are available, rules, etc. For this, the first exercise is for you to read the documentation about the framework (introduction, game definition, framework structure, etc) in the framework's Wiki:

`https://github.com/GAIGResearch/java-pommerman/wiki`

Note: it's very important that you read the game rules so you understand the game. Take a look at the rules described here: `https://github.com/GAIGResearch/java-pommerman/wiki/Pommerman-Game-Rules`.
To get a better understanding of the framework, have a look at the code structure described here:

`https://github.com/GAIGResearch/java-pommerman/wiki/Framework-Structure`

The *Pommerman* wiki is generally a good place to find information about the framework.

Next, this document (which will be updated as we progress) contains more exercises for you to get familiarised with the framework. You're not expected to submit all these exercises (they are not assessed), but the advice is that you go through them before starting your work on the main assignment. You are welcome to attempt all of the exercises listed here, in the order you prefer, although the suggestion is that you do them in order:

- Exercise 1 will encourage you to play the game yourself and understand the rules and possible strategies of the game. It will also allow you to play AI against AI in multiple settings, as a single match, or a bunch of games.
- Reading a paper about *Pommerman* .
- Exercise 2 asks you to analyze the simplest agents of the framework: DoNothing, Random, OSLA and SimpleAgent. You will also implement your first heuristic agent.
- Reading two more papers about *Pommerman* .
- Exercise 3 asks you to analyze and modify the more complex agents in the framework: MCTS and RHEA.
- Reading two papers about MCTS and RHEA.

- (Optional) Exercise 4 investigates parameter optimization options for tuning the advanced AI agents (MCTS and RHEA).
- More exercises coming soon.

Finally, the end of this document includes a reading list with papers relevant to this assignment. Read paper [4] before the second lab.

## 1 Exercise 1 - Running the game

There are two classes, Test.java and Run.java, in the src/ directory that can be used to run your bots and also to play the game yourself against the built-in AIs.

### 1.1 Single Games

For now, open the class **Test.java**. You'll see that, from line 40 onward, lines like the following are repeated:

```
players.add(new ...);
```

*players* is a Java *ArrayList* that contains all players for a single game. Before a game is started, this array must contain four players - no less, no more. The order of the players in the array matters: from first to fourth, the each player is put in a different corner of the map following this order: top-left, bottom-left, bottom-right, top-right.

By default, the game has two "SimplePlayer" and two MCTS playing. The different players you can use at the moment in this framework are:

| **Bot** | **Java construction code** |
|---|---|
| DoNothing | new DoNothingPlayer (playerID++) |
| Random | new RandomPlayer (seed, playerID++) |
| OSLA | new OSLAPlayer (seed, playerID++) |
| SimplePlayer | new SimplePlayer (seed, playerID++) |
| RHEA | new RHEAPlayer(seed, playerID++) |
| MCTS | new MCTSPlayer(seed, playerID++) |

All the player classes are in src/players/. The seed is used to generate a random starting board for the game, which determines the location of the obstacles and power-ups. Each seed number genreates a completely different map to play in.

By changing the players that are added to the *players* array, try now different combinations of agents playing the game and see how well they do, by running the class **Test.java**. Also take a quick look now at `https://github.com/GAIGResearch/java-pommerman/wiki/AI-Players` and read about the different AI players available. We'll be back to analysing these players in Exercise 3 (next week), but it's good for you to try to understand them on your own for now.

#### 1.1.1 Playing as a human

You can also include a human player in the groups of agents that play. A human player is created as follows:

```
new HumanPlayer(ki1, playerID++)
```

where *ki1* is a *KeyController* object (created in line 26). Substitute the first agent in the array of players for a new human player. Run the Test class and play the game using the arrow keys to move and the Space key to drop a bomb.

You can actually play the game with up to two human agents:

| **Human Agent** | **Key Controller** | **Keys** |
|---|---|---|
| Human player 1 | KeyController ki1 = new KeyController(true); | Cursor keys + Space |
| Human player 2 | KeyController ki2 = new KeyController(false); | WASD + Left Shift |

Configure the game now to play with 2 humans and 2 bots. You can play a few games with a colleague sitting next to you!

### 1.1.2 Different game modes

The game can be played in two different game modes, with the following rules:

- Free for all (FFA): All four players are competing against each other.
  - Last player standing **wins**.
  - All players that die **lose** the game.
  - Players that are alive at the end **tie**.
  - If the last players alive die on the same tick, they also **tie**.
- Team: The four players are grouped in teams of 2 vs 2.
  - They're always paired $1^{st}$-$3^{rd}$ vs $2^{nd}$-$4^{th}$, sited in opposite corners.
  - Team with no agent **loses**, other team **wins**.
  - If there are players of each team alive at the end, teams **tie**.
  - If the last alive players of each time die on the same tick, teams **tie**.

The team mode is decided when the *Game* object is created. See Test.java, line 23, which by default sets FFA as the game mode:

```
Game game = new Game(seed, boardSize, Types.GAME_MODE.FFA, "");
```

In order to change the mode to TEAM, substitute *Types.GAME_MODE.FFA* for *Types.GAME_MODE.TEAM* and play the game.

### 1.1.3 Different observabilities

All players receive information (presence and location of the different tiles) about their surroundings. By default, the agents receive information about the **full** board (full observability setting). This is indicated by the variable *DEFAULT_VISION_RANGE* in the class *Types.java* (src/utils/Types.java).

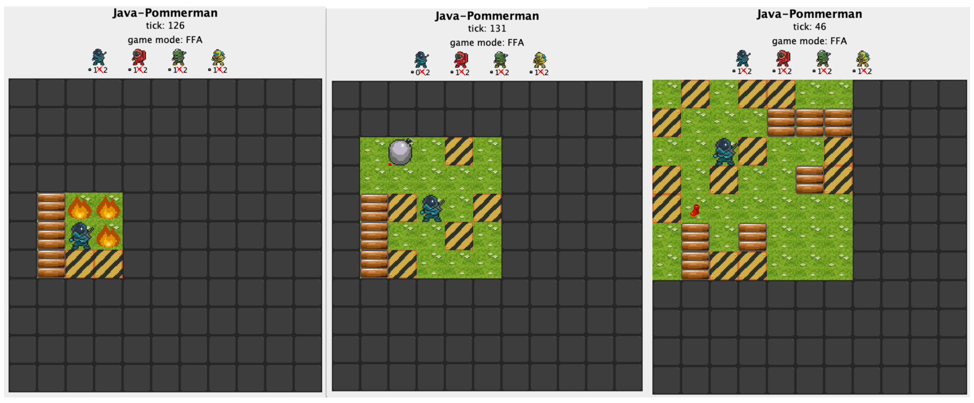
| **utils.Types.DEFAULT_VISION_RANGE value** | **Observability** |
|---|---|
| $-1$ | Full observability |
| $N\ (>0)$ | Partial observability ($N$ tiles from location) |

When a value $N > 0$ is set for the Vision Range, the agent will only receive information about the tiles at a distance $\leq N$ from the location of the agent:

Execute now the game with 4 different **bot** agents and a PO setting of 2. Observe how the game takes place. Note that, with Partial Observability and no human players, you see the vision of an independent observer, while you can see the visibility of every player on the right part of the screen. You can press 1, 2, 3 and 4 keys to toggle which agent's view you display on the main left panel. Press 0 to come back to the independent observer's view.

### 1.1.4 Game settings

*Pommerman* is an easily parameterizable game. This means that there are many ways you can tweak the rules and the game parameters. Take a look at `https://github.com/GAIGResearch/java-pommerman/wiki/Game-Parameters` and modify some of these parameters to see the effect they take on the game and the agents that play it.



PO $N = 1$ PO $N = 2$ PO $N = 4$

Figure 1: *Pommerman* executed with different Partial Observability (PO) settings.

## 1.2 Multiple Games

When trying to determine if an AI is better at *Pommerman* (or any other environment, really), it is never enough to play a single game. Because of this, the framework includes another class (Run.java) that allows you to run multiple games and get comprehensive statistics at the end of the execution.

Open Run.java (in src/). This class is executed passing 7 parameters as arguments (although passing none executes a default mode). The usage instructions are as follows:

```
Usage: java Run [args]
   [arg index = 0] Game Mode. 0: FFA; 1: TEAM
   [arg index = 1] Number of level generation seeds (i.e. different levels).
   [arg index = 2] Repetitions per seed [N]. "1" for one game only with visuals.
   [arg index = 3] Vision Range [VR]. (0, 1, 2 for PO; -1 for Full Observability)
   [arg index = 4-7] Agents. When in TEAM, agents are mates as indices 4-6, 5-7:
     0 DoNothing
     1 Random
     2 OSLA
     3 SimplePlayer
     4 RHEA 200 itereations, shift buffer, pop size 1, random init, length: 12
     5 MCTS 200 iterations, length: 12
     6 Human Player (controls: cursor keys + space bar).
```

The total number of games played will be the number of seeds (arg index 1) multiplied by the number of repetitions by seed (arg index 2). For instance, running:

```
java Run 0 1 1 -1 2 3 4 5
```

Executes one FFA game with 1 as random seed, full observability and the players (in this order): OSLA, SimplePlayer, RHEA and MCTS. This is also the default setting when no parameters are passed to the class.

Run now 'Run.java' to see what happens (Menu 'Run' → 'Run' and select 'Run').

In order to set your own parameters in IntellijIDEA, click on 'Run' → 'Edit Configurations...' and introduce them (separated by spaces) in the field 'Program arguments:'. Try different arguments, setting different number of seeds (i.e. different levels) and repetitions per seed. You should obtain an output with this format:

```
0-2-2--1-2-3-4-5 [OSLA,RuleBased,RHEA,MCTS]
40665, 0/4, [LOSS, LOSS, LOSS, WIN]
40665, 1/4, [LOSS, LOSS, LOSS, WIN]
83757, 2/4, [LOSS, LOSS, LOSS, WIN]
83757, 3/4, [LOSS, LOSS, WIN, LOSS]
```

```
N    Win    Tie    Loss   Player
4  0.0%   0.0%   100.0%   players.OSLAPlayer
4  0.0%   0.0%   100.0%   players.SimplePlayer
4  25.0%  0.0%   75.0%  players.rhea.RHEAPlayer
4  75.0%  0.0%   25.0%  players.mcts.MCTSPlayer
```

The lines of the output are described as follows:

- Line 1: execution string (with the parameters specified) and agents that played the games.
- Lines 2 onward: seed for the level, game number, result per player (same order as in the first line).
- Last 5 lines: table of results, where each column indicates, from left to right, number of games played, percentage of wins, ties and losses, and the player.

Run this class a few times and try to determine which is the strongest agent of the ones available by pitching them against each other.

## 2 Paper Read

For next week's lab, read the following *Pommerman* paper[1] [4].

## 3 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.

[1]Available in the *Pommerman* Github repository and also here: `http://www.diego-perez.net/papers/PommermanAIIDE19.pdf`

- 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/.

## 3.1 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:

```java
//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!).

## 3.2 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.

### 3.3 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. Can you think of ways of combining the heuristics of the Simple Agent with the use of the Forward Model?

## 4 Paper Read

Read the following two papers about Pommerman:

- The original Pommerman paper (about the original Python framework) [5][2] (6 pages).
- A Hybrid Search Agent in Pommerman [6][3] (Short Paper - 4 pages).

## 5 Exercise 3 - Advanced AI Agents

### 5.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a highly selective best-first tree search method. This iterative algorithm balances between *exploitation* of the best moves found so far and *exploration* of those that require further investigation, respectively. On each iteration of the algorithm, the standard MCTS performs 4 steps: selection, expansion, simulation and back propagation.

The MCTS agent implemented for Pommerman can be found at `src/players/mcts/`. The main classes here are:

- **MCTSPlayer**: it's the class that extends from Player.java and implements the agent API.
- **SingleTreeNode**: implements an MCTS node and all the logic required for the MCTS algorithm.
- **MCTSParams**: a class for defining the parameters of MCTS.

MCTSParams can be passed to the MCTSPlayer constructor in order to specify the configuration of MCTS. Three parameters can be defined in this class:

[2] `https://arxiv.org/pdf/1809.07124.pdf`
[3] `http://www.akhalifa.com/documents/hybrid-search-pommerman.pdf`

```java
public double K = Math.sqrt(2); //Constant for UCB1
public int rollout_depth = 12;  //Number of steps the tree can grow from
    the root
public int heuristic_method = CUSTOM_HEURISTIC; //Heuristic to evaluate a
    state.
```

You can also change the budget settings to determine when the algorithm stops execution to choose an action:

```java
public int stop_type = STOP_TIME; //See Types.java for other STOP
    conditions
public int num_iterations = 200;
public int num_fmcalls = 2000;
public int num_time = 40;
```

In order to change these parameters, create an MCTSParams object and assign the desired values before calling the MCTSPlayer constructor (i.e. from Test, or Run).

**Part 1:** In this agent, take a look at the following methods and try to understand the flow of the algorithm and how it is implemented. Can you answer the following questions (ask if you can't)?

1. In SingleTreeNode.mctsSearch(...), can you identify the four steps of the algorithm?
2. Can you explain and understand the different lines of code included in the UCB1 calculation (method SingleTreeNode.uct())?
3. How does the selection step stop so that the algorithm moves to the expansion step?
4. In the expansion, how do you select which action must be chosen to add a new node?
5. In the simulation step, how are the actions chosen for the rollout? What opponent model is MCTS using? When does a rollout terminate?
6. How does the stopping condition operate?
7. How and where (in the code) are the statistics of each node being updated after the state at the end of the rollout has been evaluated?
8. How is the action to be returned chosen?

**Part 2:** Executing agents with the Run class, modify different parameters of MCTS and see the effect they have. For instance, does having a shorter/longer rollout depth make the algorithm weaker/stronger? You can also try different values for K, modify the budget given for decision making, etc.

## 5.2 Rolling Horizon Evolutionary Algorithms

Rolling Horizon Evolutionary Algorithms (RHEA) are a family of algorithms that use evolution in real-time to recommend an action on each turn for the player to make. In its standard form, RHEA evolves sequences of L actions, evaluating the state found at the end of the sequence. RHEA uses a population of N individuals and the regular evolutionary operators apply. At the end of the computational budget, RHEA returns the first action of the best sequence found during evolution to be played in the game.

The RHEA agent implemented for Pommerman can be found at `src/players/rhea/`. The main classes here are:

- **RHEAPlayer**: it's the class that extends from Player.java and implements the agent API.
- **RollingHorizonPlayer**: implements RHEA (with the aid of evo/Evolution.java) and executes an Evolutionary Algorithm to evolve sequences of actions.
- **GameInterface**: links RHEA with the game API for sequence evaluation and any other methods requiring game-specific logic.
- **utils/RHEAParams**: a class for defining the parameters of RHEA.

- **evo/*.java**: Classes that implement the evolutionary algorithm used to evolve action sequences in RHEA.

As with MCTS, RHEAParams can be passed to the RHEAPlayer constructor to define the parameters of RHEA. RHEA has, however, more parameters than MCTS. Take a look at all the parameters available at the wiki page (`https://github.com/GAIGResearch/java-pommerman/wiki/AI-Players`).

**Part 1:** Take a look at RollingHorizonPlayer.getAction() and Evolution.iteration(), and try to answer the following questions:

1. How are those two methods integrated? What are the different steps of the former (getAction) and when is the latter (iteration) being called?
2. (In Evolution.java), how is the initial population generated? Why does it also evaluate each new individual?
3. What different types of selection operators are available? How do you change them and how do they work?
4. How is offspring generated in Evolution? What parameters can you change in RHEAParams (and to what values) to alter this operator?
5. The class GameInterface.java implements a method, *evaluate(...)*, that (with the help of *evaluateRollout(...)*) takes a sequence of actions, executes it in the game and gives a value to the state reached at the end. Are you able to follow the logic of this code and understand how it works?

**Part 2:** Executing agents with the Run class, modify different parameters of RHEA and see the effect they have. For instance, you can alter population size, individual length, shift buffer (on/off) and iteration budgets.

You may notice that a second implementation for a much simpler RHEA exists as well, at `src/players/SimpleEvoAgent.java`. This agent uses a Random (or Stochastic) Mutation Hill Climber to evolve sequences instead, with less customization. Can you identify the similarities (and differences) between the two agents?

## 6 Paper Read

Read the following two papers about MCTS and RHEA:

- Real-Time Monte Carlo Tree Search in Ms Pac-Man [2][4].
- Rolling Horizon Evolution versus Tree Search for Navigation in Single-Player Real-Time Games [3][5].

## 7 Exercise 4 (Optional) - Optimizing AI Agents

Since the AI agents in the framework each have several parameters controlling their decision making process, it can often be the case that some parameter configurations are better than others. We can use an optimization algorithm to figure out what good parameter values could be.

The N-Tuple Bandit Evolutionary Algorithm (NTBEA) [1] is an optimization algorithm using several concepts such as N-Tuple systems to build a fast internal model of the search space, N-armed bandits to help guide the search for solutions and evolutionary algorithms to evolve solutions to a problem. An implementation of this algorithm can be found at `players/optimisers/` and it is set up to optimize agent parameters for Pommerman. The key classes include:

- **players/optimisers/ntbea/RunNTBEA.java**: main class responsible for running NTBEA. The configuration for NTBEA can be controlled here, such as kExplore, epsilon, nEvals (iterations of the algorithm) or tuples to use (lines 51-54 in RunNTBEA.java).

[4]`https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6731713`
[5]`http://www.diego-perez.net/papers/GECCO_RollingHorizonEvolution.pdf`

- **players/optimisers/ParameterizedPlayer.java**: in order for an agent to be compatible with NTBEA, they must extend from this super class (instead of `Player.java`) and implement the additional methods for setting and getting parameter sets, as well as translating integer genomes into their corresponding parameter set.
- **players/optimisers/evodef/EvaluatePommerman.java**: this class sets the fitness function for NTBEA, i.e. assigning a fitness to a given parameter configuration.

**Part 1:** Take a look at the optimization set up, and try to answer the following questions:

1. Can you figure out how to tune the parameters for RHEA? What about MCTS?
2. What is the fitness function doing?
3. How is each player informed of a new set of parameters, after initialization?
4. How can you use an optimized parameter configuration (solution obtained at the end of an NTBEA run) to run the optimized player in experiments (in Test.java or Run.java)? (Hint: have a look at the `ParameterizedPlayer.translateParameters()` method)

Read the information on the wiki page for further details (`https://github.research.its.qmul.ac.uk/eecsgameai/pommerman/wiki/Optimizing-Agent-Parameters`).

**Part 2:** Try to tune MCTS to beat SimplePlayers in the TEAM game mode with partial observability (vision range set to 4).

1. What do you have to modify for this (Hint: you should only have to modify 2 classes, to set up the optimized player and the fitness evaluation).
2. **Test** the optimized player in several runs (with the same game configuration) against SimplePlayers.
3. In the test runs (with the same optimized parameter configuration), what happens if you change the opponent team to OneStepLookAhead agents? What if you change the partial observability settings or game modes? Does the optimized player still work as well?

## References

[1] Simon M Lucas, Jialin Liu, and Diego Perez-Liebana. The N-Tuple Bandit Evolutionary Algorithm for Game Agent Optimisation. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–9. IEEE, 2018.

[2] Tom Pepels, Mark HM Winands, and Marc Lanctot. Real-Time Monte Carlo Tree Search in Ms Pac-Man. *IEEE Transactions on Computational Intelligence and AI in games*, 6(3):245–257, 2014.

[3] Diego Perez, Spyridon Samothrakis, Simon Lucas, and Philipp Rohlfshagen. Rolling Horizon Evolution versus Tree Search for Navigation in Single-Player Real-Time Games. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 351–358. ACM, 2013.

[4] Diego Perez-Liebana, Raluca D Gaina, Olve Drageset, Ercüment Ilhan, Martin Balla, and Simon M Lucas. Analysis of Statistical Forward Planning Methods in Pommerman. In *Proceedings of Artificial Intelligence in Interactive Digital Entertainment*, 2019.

[5] Cinjon Resnick, Wes Eldridge, David Ha, Denny Britz, Jakob Foerster, Julian Togelius, Kyunghyun Cho, and Joan Bruna. Pommerman: A Multi-agent Playground. In *MARLO Workshop, AIIDE-WS Proceedings*, pages 1–6, 2018.

[6] Hongwei Zhou, Yichen Gong, Luvneesh Mugrai, Ahmed Khalifa, Andy Nealen, and Julian Togelius. A Hybrid Search Agent in Pommerman. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, page 46. ACM, 2018.