TAG: Advancements in AI-Driven Tabletop Games
Tutorial @ IEEE CoG'23

-

Running TAG (Simple)

1 Overview

The objective of this lab is to familiarise yourself with TAG and process the information that is given to you in the different methods and classes. You should make sure you can run TAG in your machine (See the Getting Started section) before continuing.

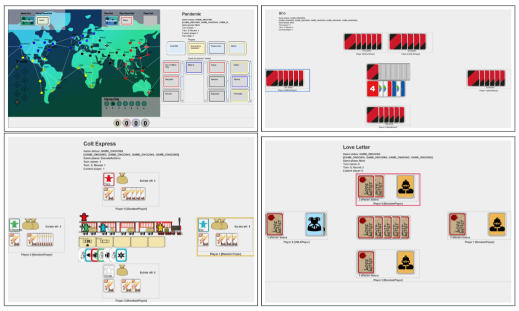
Framework Description

The Tabletop Games Framework (TAG)[2] promotes research into general AI in modern tabletop games, facilitating the implementation of new games and AI players, while providing analytics to capture the complexities of the challenges proposed. The following definitions will help you understand the rest of this document:

We define an **action** as an independent unit of game logic that modifies a given game state towards a specific effect (e.g. player draws a card; player moves their pawn). These actions are executed by the game players and are subject to certain **rules**: units of game logic, part of a hierarchical structure (a game flow graph). Rules dictate how a given game state is modified and control the flow through the game graph (for instance, checking the end of game conditions and the turn order).

A **turn order** defines which player is due to take the next action, possibly handling player reactions forced by actions or rules. At a higher level, games can be structured in **phases**, which are time frames where specific rules apply and/or different actions are available for the players.

All tabletop games use **components** (game objects sharing certain properties), whose state is modified by actions and rules during the game. TAG includes several predefined components to ease the development of new games, such as tokens (a game piece of a particular type), dice (with N sides), cards (with text, images or numbers), counters (with a numerical value), grid and graph boards. Components can also be grouped into collections: an area groups components in a map structure in order to provide access to them using their unique IDs, while a deck is an ordered collection with specific interactions available (e.g. shuffle, draw, etc.). Both areas and decks are considered components themselves.



Dive Deeper 1 The first step is to inform yourself about the game - how it works, what kind of objects and actions are available, rules, etc. For this, you are recommended to first take a quick look at the documentation about the framework (introduction, game definition, framework structure, etc.) in the framework's Wiki:

`http://www.tabletopgames.ai/wiki/`

This Wiki includes definitions of terminology to better understand the framework, and information on games currently within it. The TAG wiki is generally a good place to find information about the framework.

You can also find more information about TAG in this article published at EXAG 2020:

`https://rdgain.github.io/assets/pdf/papers/gaina2020tag.pdf`

2 Running the framework

There are different ways to run the framework and play yourself against the built-in AIs. We'll start with two simple ones: `core.Game.java` and `gui.Frontend.java`.

2.1 Game

In the `Game` class (found in the `core` package), you'll find the `main` method which can be run at the end of the file. Here, you can customise the way you run games by modifying some of the code. 4 steps are identified in the comments in this method:

2.1.1 Action Controller

The `Action Controller` is a Java object which is set up in the framework to capture user input in the Graphical User Interface (GUI). This needs only be created and attached to the game in order to work:

```
ActionController ac = new ActionController();
```

If this object is set to `null`, the game will run *without* visuals. However, there's a more intuitive way of setting the GUI on or off, the *useGUI* parameter.

Exercise 2 Locate this line at the beginning of the main() method in Game.java:

```
boolean useGUI = Utils.getArg(args, "gui", true);
```

Set the last value to *false*, which indicates that the GUI must run without visuals:

```
boolean useGUI = Utils.getArg(args, "gui", false);
```

Run the game now and verify that the no GUI is created. In the console message at the bottom of the IDE should indicate that the execution was run with no errors ("Process finished with exit code 0").

Finally, revert your change so GUI is enabled for the following exercises.

2.1.2 Game Seed

Next, the random seed for the run is initialised. This random seed is used for the random number generator needed to produce stochasticity in the games (for instance, to suffle cards randomly). Two games that are run using the same seed (e.g. `46748483L`) will produce the same sequence of random events. This is a `long` data type, and can take a specific value (to replicate the same results in different runs), or `System.currentTimeMillis()`, a Java method which returns a `long` value based on the current time (and therefore different for different runs).

```
long seed = System.currentTimeMillis();
```

2.1.3 Players

A few lines follow to initialise a list of **players**. This variable is a Java `ArrayList` data type, an ordered list. It needs to contain Java objects representing the players that will take part in the game. All objects added to this list need to extend the `AbstractPlayer` class, and the size of this list will indicate the number of players for the game. Multiple copies of the same player can be added. The following example would run a 3-player game with a human player, an AI player that plays at random, and an AI player controlled by the Monte Carlo Tree Search algorithm:

```java
ArrayList<AbstractPlayer> players = new ArrayList<>();
players.add(new HumanGUIPlayer(ac));
players.add(new RandomPlayer());
players.add(new BasicMCTSPlayer());
```

Exercise 3 'Pandemic' is a game that can be played by maximum of 4 players. Modify the code in the main() function of the Game.java class to run a game of Pandemic with 4 players: a human GUI player, an OSLA Player, an MCTS player and a random player. Note that you may have to import some classes at the top of Game.java for some of these objects (the IDE will give you a hint of what to import).

If the number of players indicated doesn't correspond with the number of players accepted by the game being run, an error will occur. The easiest way to check the number of players is in the class `games.GameType.java`. This class contains the descriptors of all the games in the framework which can be run, including:

- The number of players they accept (minimum and maximum).
- A list of categories the game belongs to.
- A list of mechanics the game uses.
- References to the classes which actually implement the game functionality and the specific ways needed to instantiate each game.

Exercise 4 Go to the class games.GameType.java (located at main/java/games/) and see if you can find the minimum and maximum number of players for the games:

- Tic Tac Toe,
- Connect4,
- Dots and Boxes,
- Pandemic,
- Poker.

The AI players which currently exist in the framework can be found in the `players` package, and can be instantiated for the players ArrayList as follows:

AI Player	Path	Java Construction Code	Opt. Constructor Arguments
FirstActionPlayer	players/simple/	`new FirstActionPlayer()`	-
Random	players/simple/	`new RandomPlayer ()`	`java.util.Random`
OSLA	players/simple/	`new OSLAPlayer ()`	`java.util.Random`
RMHC	players/rmhc/	`new RMHCPlayer()`	random seed **or** `RMHCParams`
MCTS	players/basicMCTS/	`new BasicMCTSPlayer()`	random seed **or** `BasicMCTSParams`
HumanGUIPlayer	players/human/	`new HumanGUIPlayer(ac)`	`ActionController`
HumanConsolePlayer	players/human/	`new HumanConsolePlayer()`	-

A short description of each player is as follows (more details can be found at the end of this document):

- **FirstActionPlayer**: always chooses the first action in the list of available actions in a game state.
- **Random**: executes a random action at every step.
- **OSLA**: One Step Look Ahead. Looks one step into the future, by simulating the effect of all possible actions in the current game state and picks the one leading to the best next state (according to game-specific heuristic).
- **MCTS**: Monte Carlo Tree Search[1]. Incrementally builds a game tree by simulating the effect of different actions starting from the current state up to 10 steps into the future, and picks the next immediate action that statistically leads to a better future state. Note that there are two versions of MCTS implemented in TAG: BasicMCTS and MCTS.

- **RMHC**: Random Mutation Hill Climber[3]. Uses the simplest evolutionary algorithm to evolve a sequence of actions that leads to the best state 10 steps into the future, and picks the first action in the sequence.
- **HumanGUIPlayer**: allows a human to play in the Graphical User Interface (GUI).
- **HumanConsolePlayer**: allows a human to play via the console.

Most of these agents use heuristics to evaluate how good a game state (i.e. the current position of the player in the games, considering tokens, cards or other game elements) is. These heuristics are most often game-specific, but can also be custom. These are found in the package `players.heuristics`.

An example of a simple heuristic is the Score Heuristic (see players.heuristics.ScoreHeuristic.java). This heuristic (like others) evaluates a given game state in the function `evaluateState()`, which in this case looks like this:

```java
public double evaluateState(AbstractGameState gs, int playerId) {
    double score = gs.getGameScore(playerId);
    if (gs.getPlayerResults()[playerId] == CoreConstants.GameResult.WIN)
        return score * 1.5;
    if (gs.getPlayerResults()[playerId] == CoreConstants.GameResult.LOSE)
        return score * 0.5;
    return score;
}
```

This method first retrieves the current score of the game from the game state that is meant to be evaluated (`gs`) for the player we are evaluating the position of (`playerId`):

```java
double score = gs.getGameScore(playerId);
```

Then, it checks if the game has been already won or lost by the player, which can be found in the array returned by the game state function `getPlayerResults()`, in the position `[playerId]`. This array contains the game status for each player, hence the expression

```java
gs.getPlayerResults()[playerId] == CoreConstants.GameResult.WIN
```

resolves to `true` if *playerId* already won. If the game is not over, the value of this state is the same as the score obtained. If the game is over, this function amplifies (or reduces) the value of the state by multiplying by 1.5 for a game won by *playerId* (or by 0.5 for a game lost by *playerId*, respectively).

Exercise 5 Inspect the classes in the package players.heuristics for the following Heuristics. Can you understand how do they work?

- Random Heuristic,
- Pure Score Heuristic,
- Win Only Heuristic,
- Leader Heuristic.

2.1.4 Run One

The last step is to choose the run method for your game. The main one is `RunOne`. You can see that this is the method that is being called in Game.java by default:

```java
runOne(TicTacToe, players, seed, ac, randomizeParameters, listeners, turnPause);
```

This function takes the following arguments:

- Game to play: the data time of this argument is `GameType`, so you can check the class previously mentioned for the possible values (i.e. games available in the framework).
- Path to file with configuration for the run: it's possible to supply a file that contains the parameters for the run (e.g. players, seed, etc.). If this is `null`, parameters as specified in the code will be used.
- We know already the next 2 parameters: players and seed.
- `randomizeParameters`: if true, the parameters of the game will be randomised; if false, the game will run with default parameters. You can leave this value as `false` for now.
- `listeners`: game listeners can be attached to react to specific events or record data. More on this later.

- `ActionController`, as defined above.
- `turnPause`: indicate here the number of milliseconds that the GUI pauses between game turns.

There are several ways of indicating *which* game to run. The simplest one is to change the default value on the first line of the main() method:

```
//The String ''Pandemic'' is the default argument for the variable ''game''
String gameType = Utils.getArg(args, "game", "Pandemic");
```

If the framework is run without providing arguments (as you can run it now, by default), the default value of this (and subsequent) variables are being used.

Exercise 6 Modify the players list so two random players play a game, and make sure your GUI is enabled. Then, run the following games by modifying the default game to use. For this you have to indicate the following Strings (use the indicated exact capitalization):

- "TicTacToe"
- "Connect4"
- "DotsAndBoxes"
- "ExplodingKittens"
- "Uno"
- "Poker"
- "Dominion"

Why do you think these games are already finished when you run then? You can change the value of the variable *turnPause* (for instance, to 100) to introduce a pause between turns and be able to see the moves players make.

2.1.5 Run Many

An alternative is to use the `runMany` method, which exists to run multiple games at once with the same set of players. This method will take a list of games as the first argument, instead of just one. The method looks like this:

```
runMany(games, players, seed, nRepetitions, randomizeParameters, detailedStatistics,
    listeners, turnPause);
```

In this case, `games` substitutes a single String, as multiple games can be played in a batch. This can be defined as follows:

```
new ArrayList<GameType>() {{add(Uno);}} //list for 1 game
new ArrayList<GameType>() {{add(Uno); add(Pandemic);}} //list for 2 games
```

Additionally, two new arguments are added to this method, with respect to *runOne()*:

- nRepetitions: indicates how many times each game is played.
- detailedStatistics: if true, shows some extra statistics about the games played. You can set this to *false* for now.

Exercise 7 Set up the main() method in the following way:

- 2 players to play games: RandomPlayer and BasicMCTSPlayer.
- Comment out the line of code that calls *runOne()* in the main function.
- Uncomment the last line of the main() method, which calls the *runMany()* function.
- Pass 3 games as the first parameter of runMany(), as indicated above:

    - TicTacToe
    - Connect4
    - DotsAndBoxes

- Set the number of repetitions to 5.

Run the framework and observe the results. The games tic tac toe and connect 4 will finish quite quickly, while dots and boxes will take a few seconds. At the end you'll observe an output like the following (numbers may defer):

```
Overall RandomPlayer-0: -0.93
Overall MCTSPlayer-1: 0.93
```

This numbers indicate the average of victories (counted as 1) and loses (as -1) of both players. You can also see this win rates for each game separately.

Dive Deeper 8 Choose a game and play twice 2-player games, first you against the random player, then against MCTS (do not use the cooperative game "Pandemic" for this).

Dive Deeper 9 Read the EXAG 2020 TAG paper: `https://rdgain.github.io/assets/pdf/papers/gaina2020tag.pdf`

References

[1] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Pérez Liébana, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. In *IEEE Trans. on Computational Intelligence and AI in Games*, volume 4, pages 1–43, 2014.

[2] Raluca D. Gaina, Martin Balla, Alexander Dockhorn, Raul Montoliu, and Diego Perez-Liebana. TAG: a Tabletop Games Framework. In *Proceedings of the AIIDE workshop on Experimental AI in Games*, 2020.

[3] Raluca D. Gaina, Sam Devlin, Simon M Lucas, and Diego Perez-Liebana. Rolling Horizon Evolutionary Algorithms for General Video Game Playing. *IEEE Transactions on Games*, 2021.