The BANDANA Framework v1.3

Dave de Jonge

December 19, 2016

1 Introduction

BANDANA is a Java framework for the development of automated agents that play the game of Diplomacy. This tutorial explains how to implement your own Diplomacy-playing, negotiating agents, and how to let them play against each other in a Diplomacy tournament. BANDANA is an extension of the DipGame framework.[1] However, it provides a new negotiation server and uses a simplified negotiation language. For this tutorial we will assume you are familiar with the Java programming language and with the rules of Diplomacy. If not, you can find the rules here:
`https://www.wizards.com/avalonhill/rules/diplomacy_rulebook.pdf`.

Changes w.r.t. Version 1.1

- Updated the TournamentObserver class to display the names of the players and the tournament standings.
- Introduced the ScoreCalculator class.

Changes w.r.t. Version 1.2

- Layout of the TournamentObserver slightly modified to align the player names, powers, and scores.
- Previously, the TournamentRunner didn't terminate when the tournament was over. Fixed this bug.
- Fixed a couple of smaller bugs.

Changes w.r.t. Version 1.2.1

- Added method proposeDraw() to ANACNegotiator.
- Added some comments to ExampleANACNegotiator.

[1] `http://www.dipgame.org`

2 Before You Begin

To run a game of Diplomacy, you need to run a game server and 7 agents that will be the players. Optionally, you may also run a negotiation server, if you want the agents to negotiate, and you can run one or more so-called Observers, which are agents that observe the game, but do not play. Observers are useful to collect or display information about ongoing games.

BANDANA does not provide any game server, so you will first need to download and install one. There are two options:

- The DAIDE Server (Windows only)
- Parlance (platform-independent)

In this tutorial we will only consider Parlance.

Step 1: Download and Install Python

Since Parlance is implemented in Python, you first need to make sure that Python is installed (if you are using the DAIDE server instead of Parlance you can skip this step). Don't worry if you are not familiar with Python. No knowledge of Python is required to use BANDANA. Python can be downloaded here:
`https://www.python.org/downloads/`.
We have had some problems running Parlance on Python 3.4, therefore, we advice to install Python 2.7 instead. Once you have installed Python, make sure that 'python' is added to your path variable. For information on how to do that (on Windows), look for example here:
`http://stackoverflow.com/questions/25153802/how-to-set-python-path-in-windows-7`

Step 2: Download and Install Parlance

The next step is to download and install the Parlance game server. Proceed as follows:

1. Go to the following web page:
   `https://pypi.python.org/pypi/Parlance/1.4.1`.
2. Download the file PARLANCE-1.4.1.tar.gz
3. Unzip it.
4. Open the terminal / command line.
5. Navigate to the unzipped folder.
6. Execute the following command:
   `>python setup.py install`

If everything went okay there should be a folder called `Scripts` in your Python installation folder, which contains an executable called `parlance-server.exe` (plus a number of other Parlance related files).

Step 3: download the BANDANA framework

As you had probably already figured out, the BANDANA framework can be downloaded from `http://www.iiia.csic.es/~davedejonge/bandana`.

Download the framework and unzip it. Inside the unzipped folder you will find a folder named `src` containing the Java source files that you will need to set up a tournament and the source code of some example bots. Furthermore, it contains a folder named `lib` that contains all the required libraries. Also, it contains a folder named `agents` with compiled example agents that you can use as opponents to test your own agent.

Step 4: Create a New Project

Next you need to create a new project in your favorite IDE and import the Java classes found in the `src` folder. Make sure that your project references all the libraries in the folder `lib`. We will here explain how to do this in Eclipse, but you may use any other IDE as well (we have used Eclipse Luna, so it could be slightly different if you are using a different version).

1. Create a new Java project in Eclipse, by clicking File –> New –> Java Project. Choose a name for the project, and click Finish.
2. In the package explorer, right-click on the project folder and select 'import'.
3. Select General / File System and click Next.
4. Click on the button 'Browse' next to the field 'From directory'.
5. Navigate to the BANDANA folder that you downloaded.
6. Click on the folder called BANDANA Framework 1.3 (the inner one, which contains the folders 'agents', 'lib', and 'src') and click OK.
7. Select the check box that appears next to BANDANA Framework 1.3 so that everything in the folder is selected.
8. Click Finish.

Next you have to make sure that all the libraries are properly referenced. This goes as follows:

1. In the Package Explorer, right-click on the project, and select properties from the pop-up menu.

2. Select Java Build Path in the menu on the left in the window that appears.
3. Click on the tab Libraries.
4. Click on Add Jars.
5. Open the project folder and select the lib folder.
6. Select all the .jar files that appear.
7. Click OK and then OK again.
8. If everything is okay you will see the project in the package explorer without any errors.

Step 5: Set the Correct Paths

Open the file `TournamentRunner.java` from the BANDANA framework in your IDE. To setup a tournament you need to adapt this class and run it (it is probably easiest if you make a copy of the original file to keep as a backup). However, before you can do this, you have to make sure that the this class knows where to find the agents that will participate in it. Note the following line of code at the top of `TournamentRunner.java`:

```
randomNegotiatorCommand = {"java", "-jar", "agents/RandomNegotiator.jar", "-log", "log", "-name", "RandomNegotiator", "-fy", "1905"};
```

This is the command line needed to run the `RandomNegotiator`, in the form of an array of Strings. Note however, that we are not going to launch this agent directly from the command line, but instead we are going to pass this array to a Java class called `ProcessRunner` which will launch the agent.

This code assumes that the file `RandomNegotiator.jar` is located inside a folder called `agents` which is located directly in your project folder. If you have stored `RandomNegotiator.jar` elsewhere on your file system, then change this line accordingly (the third entry of the array). Do the same with the command lines for the other 3 agents.

You also may want to change the location where the log files will be stored, by changing this line:

```
final static String LOG_FOLDER = "log";
```

If you do not change this line then all log files will be stored in a folder called `log` directly in your project folder and you need to make sure this folder exists. For each tournament that you start, a new folder will be created inside this log folder, with the date and time as its name. All results of that tournament will be logged there.

Finally, you need to make sure that the path to Parlance is set correctly. Open the file `ParlanceRunner.java` so you can adapt the two following two lines:

```
private static String PARLANCE_PATH = "C:\\Python27\\Scripts\\parlance-server.exe";
private static String HOME_FOLDER = "C:\\Users\\username";
```

Make sure that PARLANCE_PATH is correctly set to the location of the file parlance-server executable (if you installed Parlance on Windows then it is likely that the given path is already correct). Furthermore, make sure that HOME_FOLDER is correctly set to the home folder of your user account on the computer you are working on. Note that you can't set this path to any folder you like. It must really be your home folder. If this path is not set correctly Parlance will still work, but you will not be able to change the deadlines of the game.

3 Running a Tournament

Congratulations! You are now ready to run a tournament! You can do this by simply running the `TournamentRunner` class. In Eclipse you can do this by right-clicking on TournamentRunner.java in the Package Explorer and then selecting 'Run As' and then clicking 'Java Application'.

By default it will run 3 games, with 1 or 2 instances of each of the 4 example bots included with the framework. You can change the number of games to play, the deadlines for the games, and the year after which the players declare a draw inside the main method.

As you have seen in the previous section, the TournamentRunner contains four lines with the command line parameters for each of the four agents included in the BANDANA framework. They all have the same structure:

```
java -jar [location] -log [logPath] -name [name] -fy [finalYear]
```

Here, [location] is the location where the .jar file of the agent is stored. [logPath] is the path to the folder where you want its log files to be stored. [name] is the name of each agent (when running a tournament make sure that each agent is started with a different name!) and [finalYear] is the year after which the agent will propose a draw to every other player. If all surviving players propose a draw then the server stops the game and declares the outcome to be a draw. This option is very useful because otherwise a game may continue forever if it reaches a state in which no player is able to make any more progress. For some reason however, the Parlance server does not always manage to handle the draw proposals immediately, so the game may sometimes end a couple of rounds later than the intended final year.

Please keep in mind that these command line options apply to the 4 example agents that we have included with the BANDANA framework. Any agent developed by anyone else may require completely different command line arguments.

Once you start the tournament you should see a window appearing that displays information from the TournamentObserver. In the top of the window it displays which game of the tournament is currently running. For example, if we are running a tournament consisting of 3 games then during the first game it will display: `Game: 1/3`. The second line shows the current phase and year of the current game.

Below this, it displays the progress of the current game. It displays the names of the 7 players, and the name of each player is followed by the power it plays and the number of Supply Centers it currently owns. Note that for some agents it does not display a name, but only a question mark. The reason for this is that the Tournament Observer learns their names from the Negotiation Server. The Parlance game server does not disclose the names of the connected players until the end of the game, so during the game the Tournament Observer can only know the names of the agents that are connected to the Negotiation Server. The agents indicated with a question mark are agents that do not negotiate (see Section 6 for more information about this).

In the bottom of the window you see a table that displays the tournament standings so far. During the first game of the tournament it will be empty, but from the second game onward it will show the overall scores of each player over the previous games. By default it displays 4 different score values for each player, namely the following:

- Solo Victories: how many times it won the game by a solo victory (owned 18 Supply Centers or more).
- Supply Centers: how many Supply Centers it conquered in total (the number of Supply Centers it owned at the end of a game, summed over all games).
- Points: a player receives 0 points if it gets eliminated, 12 points for each solo victory, 6 points for each 2-player draw, 4 points for each 3-player draw, 3 points for a 4-player draw, 2 points for a 5-player or 6-player draw and 1 point for a 7-player draw.
- Average Rank: the average over all ranks it obtained in each game. A player obtains rank 1 in a game if it scored the highest number of Supply Centers, rank 2 if it scored the second highest number of Supply Centers, etcetera. If two players both end with 0 Supply Centers the players are ranked according to who was eliminated last. If two players rank equally they both receive the average of the two ranks.

By default, the players are ordered according to the number of solo victories. The player with the highest number of solo victories is displayed at the top

of the table. If two or more players have an equal number of solo victories they are ranked according to who conquered the most Supply Centers. If they still rank equal the tie is broken by the number of points, and finally they are ordered according to their average rank (the player with the lowest average game rank, is the better one). In Section 5 we explain how you can change the order of importance of these scoring systems and how you can even implement your own scoring system. Note that for the first 3 of these scoring systems the table displays the total value, followed the average value between parentheses, while for the last one it only shows the average value.

A new folder to store the log files of this tournament is created inside your main log folder. When the tournament is finished, it contains a number of files and a folder for each agent that played in the game. There is a file called `gameResults.log` which contains the results of all the games played in the tournament. For each game it displays for each player which power it played and with how many Supply Centers it finished, or in which year it got eliminated.

The file `tournamentResults.log` contains the overall results of the tournament. It displays for each player how many games it played, and its final score according to the same scoring systems as you have seen in the window of the Tournament Observer.

If you want to write some code to analyze the results of the tournament you can use the following line after the tournament has finished:

```
ArrayList<GameResult> results = tournamentObserver.getGameResults();
```

It returns a list that contains one GameResult object for each game of the tournament, which represents the results of that game. It provides methods to get the names of the players in that game, the number of Supply Centers owned by each player at the end of the game, the rank of each player, the year of elimination of a player (if it got eliminated) and, in case of a Solo Victory, the name of the winner.

For each player you will find a log folder inside the main log folder which in turn contains one folder for each game played. Each of these folders contains one or two log files. The first log file has a file name starting with `dip_`. This file is created by the DipGame framework and stores the communication between the player and the server. The other log file is created by the player itself and has the name of the Power played as its file name. Note however that not all players create such a log file.

Sometimes, after a game has finished, the console may display an error message coming from the Parlance server saying something like 'An existing connection was forcibly closed...'. You can simply ignore this.

Finally, we would like to remark that if you stop the TournamentRunner before the tournament is finished then Parlance may continue running. In that case you may need to kill that process manually via the TaskManager (in Windows) before you can start a new tournament. You may also need to manually kill the players in that case.

Exercises

1. Adapt the code of the TournamentRunner to start a tournament with 4 instances of the DumbBot and 3 instances of D-Brane. Let them play 10 games, and make sure they declare a draw after the year 1910. Set the deadlines to 10 seconds for each type of phase.
2. After the tournament has finished, open the tournamentResults.log file. Which player scored the most Supply Centers?

4 Building Your Own Agent

Now that you have managed to set up the framework and have a tournament running, it is time to start writing your own agent. In order to do this you need to create a Java class that extends the Player class which is defined in the DipGame framework.

As an example, we will take a look at the source code of the RandomBot, which is provided with the framework. This player provides the bare minimum of a Diplomacy-playing agent. It makes only random moves and does not negotiate. We will look at negotiating agents in the following sections.

The agent contains a main method which creates a new object of the `RandomBot` class and then creates a communicator object that establishes the connection between the agent and the game server. The agent is then started by calling `randomBot.start(communicator)`.

The agent will automatically connect to the game server. Once it is connected, the DipGame framework will call the method `init()` of the `Player` class, which is overridden by the `RandomBot`. Also, the DipGame framwork sets the `me` field of the Player class. This field represents the power that the server has assigned to the agent.

Once 7 players have connected to the server, the method `start()` is called, which you can implement to do any stuff you want to happen at the start of the game. From now own you can access the 'game' field which represents the current state of the game (see section below: 'The Game Class').

The most important method inherited from the Player class is the `play()` method. This method is called at the beginning of each new phase and must return a list of orders, containing exactly one order for each of the player's units.

The Game Class

The `Player` class has a field called `game` of class `Game`. This object is updated every round and can be used to obtain all information about the current state of the game. Through this object you can for example obtain the the current year, the current phase, and the positions of all the units on the map. Furthermore, through this object you can get a list of all provinces, and for each province it stores which power currently controls that province and which power is the owner of that province (A power 'controls' a province if it currently has a unit in that province. A power becomes the 'owner' a province if it controls the province after a fall phase and remains the owner until another power becomes the owner of that province).

The Province Class

As you know, the map of Diplomacy is divided into provinces. These provinces are represented in DipGame by `Province` objects. You can get a `Province` object by calling `game.getProvince(name);` where `name` is a string which is the three-letter acronym of that province. For example, to get the `Province` object representing Holland, you call: `game.getProvince("HOL");`. To determine whether a certain province is a Supply Center or not the `Province` class provides the method `isSC()`. To get a list of all provinces, you can call: `game.getProvinces()`.

The following code demonstrates how you can obtain the current owner and the current controller of a province:

```
Province holland = game.getProvince("HOL");
Power ownerOfHolland = game.getOwner(holland);
Power controllerOfHolland = game.getController(holland);
```

The Region Class

Each province in DipGame contains one or more Regions. There are two types of regions: *army-regions* and *fleet-regions* and we can distinguish three kinds of provinces:

- A *Sea Province* is a Province that contains exactly one fleet-region and no army-regions.
- An *Inland Province* is a Province that contains exactly one army-region and no fleet regions.
- A *Coastal Province* is a Province that contains exactly one army-region and one or two fleet-regions.

When a unit is located in a Province, then it is in fact located in one of its Regions. Clearly, if the unit is an army then it must be located in the army-region of that province, and if it is a fleet then it must be located in any of its fleet-regions.

For a given province you can get its regions by calling its `getRegions()` method:

```
Province holland = game.getProvince("HOL");
List<Region> regionsOfHolland = holland.getRegions();
```

Also, you can get a specific region from the game object by calling `game.getRegion(name)`, where `name` is the six-letter acronym of the region. For example, to get the army-region of Holland: `game.getRegion("HOLAMY")`.

We should remark that in DipGame there is no special class to define the players' units. Instead, a unit is simply represented by a Region object, corresponding to the region where that unit is located. To know where the units of any power are located, the agent can call the method `getControlledRegions()` on a `Power` object. For example:

```
Power austria = game.getPower("AUS");
List<Region> unitsOfAustria = austria.getControlledRegions();
```

The Order Class

In every SPR and FAL phase of the game you must give an order to each of your units. This is done by creating a list of `Order` objects which must be returned by the `play()` method.

For example, suppose you want to move an army in Holland to Belgium. Then you must create an object of type MTOOrder (move-to order):

```
Region location = game.getRegions("HOLAMY");
Region destination = game.getRegions("BELAMY");
Order order = new MTOOrder(me, location, destination);
```

If however you want the army in Holland to stay where it is, you must create a HLDOrder (hold order):

```
Region location = game.getRegions("HOLAMY");
Order order = new HLDOrder(me, location);
```

If you want your fleet in the North Sea to support your army in Holland to move to Belgium, you can do this as follows:

```
//Order the army in Holland to move to Belgium
Region location1 = game.getRegions("HOLAMY");
Region destination1 = game.getRegions("BELAMY");
Order order1 = new MTOOrder(me, location1, destination1);

//Order the fleet in the North Sea to support the previous order.
Region location2 = game.getRegions("NTHFLT");
Order order2 = new SUPMTOOrder(me, location2, order1);
```

If you want to support a unit to hold you must create an object of type SUPOrder rather than SUPMTOOrder.

For more information about the types of Orders you can create for AUT, SUM and WIN phases, please take a look at the source code of RandomBot.

Unfortunately, the current version of DipGame does not provide any class for Convoy orders. Therefore, it is not possible for an agent implemented on the DipGame framework (or the BANDANA framework) to use convoys.

Exercises

1. Adapt RandomBot such that at the beginning of the game it prints out a list of names of all the provinces, and for each province a list of names of all its regions.
2. Let's make the RandomBot a bit more intelligent and a bit less random. Therefore, adapt it such that:
   - If it has a unit inside a Supply Center, which it currently does not own, then make sure that it holds.
   - Otherwise, if a unit is in a province adjacent to a Supply Center which it does not own, make sure it moves there.
3. Adapt the agent of the previous exercise such that, whenever two of its units want to move to the same province, instead, one unit will give support to the other.
4. Run a tournament to see if your new agent is better than the original RandomBot. To do this, make sure that both the RandomBot and your new agent are compiled into a jar-file, and adapt the code of the TournamentRunner so that it can run this agent.

5 Implementing your own Scoring System

Above we have explained that the Tournament Observer calculates 4 types of scores for the players and we have explained how it uses these scores to rank the players. In this section we explain how you can change this behavior and implement your own scoring system.

5.1 Changing the Order of Importance of the Scoring Systems

Note that the source code of the TournamentRunner contains the following lines:

```
ArrayList<ScoreCalculator> scoreCalculators = new ArrayList<ScoreCalculator>();
scoreCalculators.add(new SoloVictoryCalculator());
scoreCalculators.add(new SupplyCenterCalculator());
scoreCalculators.add(new PointsCalculator());
scoreCalculators.add(new RankCalculator());
```

The list of ScoreCalculators created here is passed on to the constructor of the TournamentObserver. The order in which the ScoreCalculators are added to this list determines their order of importance.

Suppose for example that you are not interested in the number of solo victories or the number of points. Instead, you want the players to be scored according to their average rank, and you want to use the number of supply centers as a tie breaker. You can then simply replace the above 5 lines by the following 3 lines:

```
ArrayList<ScoreCalculator> scoreCalculators = new ArrayList<ScoreCalculator>();
scoreCalculators.add(new RankCalculator());
scoreCalculators.add(new SupplyCenterCalculator());
```

Now the TournamentObserver and the tournamentResults log file will no longer show the number of solo victories or the number of points. They will order the players firstly based on their average ranks (because the RankCalculator was added first to the list) and will use the number of Supply Centers as the tie breaker (because it was added second to the list). You can add as many ScoreCalculators to this list as you like (but minimally one) and you can add them in any order you like.

5.2 Implementing Your Own ScoreCalculator

If you are not satisfied with any of the four scoring systems supplied with the Bandana framework you may implement your own Score Calculator.

For example, suppose that you want to run a tournament in which a player receives 100 points for every game in which it conquers 10 or more Supply Centers, and 0 points for all other games. In order to do so, you can create a new class that extends the abstract class`ScoreCalculator`. In this example we will call this class `MyScoreExample`. You then need to implement the following four methods:

- `calculateGameScore(GameResult newResult, String playerName)`
- `getTournamentScore(String playerName)`
- `getScoreSystemName()`
- `getScoreString(String playerName)`

The `calculateGameScore()` method is called once for every player at the end of each game. The `ScoreCalculator` base class uses this to calculate the sum and the average of the scores obtained over all games played. Following our example, we can implement it as follows:

```
public double calculateGameScore(GameResult newResult, String playerName) {
    if(newResult.getNumSupplyCenters(playerName) >= 10){
        return 100.0;
    }else{
        return 0.0;
    }
}
```

The `getTournamentScore()` method is the most important method. It is also called once for every player at the end of each game. The Tournament Observer uses the value returned by this method to sort the players in its window and in the tournamentResults log file. In most cases you would simply want this method to return the average of the scores returned by `calculateGameScore()` for each game previously played. Since this average is already calculated by the `ScoreCalculator` base class and can be obtained by calling `getAverageScore()`, we can simply implement it as follows:

```
public double getTournamentScore(String name) {
    return this.getAverageScore(name);
}
```

In some cases however, you may want this method to return something more complicated than the average score over all games. For example, you may want it to return the highest value obtained in any of the games, or the average over the 10 highest values obtained. In such cases you will need to write your own code to calculate this.

The method `getScoreSystemName()` is used to display the name of the scoring system in the top of the table in the window of the TournamentObserver. We can simply let it return the String "MyScoreExample". The method `getScoreString()` determines how the player's score will be displayed in the table and in the log file. The standard implementation is as follows:

```
public String getScoreString(String playerName) {
    long total = Math.round(this.getTotalScore(playerName));
    double average = Utils.round(this.getAverageScore(playerName), 3);
    return "" + total + " (av. = "+ average + ")";
}
```

This returns a String consisting of the total score obtained by the player, followed, between parentheses, by the average value rounded off to 3 decimals. The string returned by this method can be anything and does not need to be related to the score of the player at all, but then the TournamentObserver and the tournamentResults file would not show any interesting information.

Finally, we need to add a constructor to our example:

```
public MyScoreExample() {
    super(true);
}
```

The boolean value `true` passed on to the super constructor indicates that a higher score is to be interpreted as a better result. If you look at the source code of the four provided `ScoreCalculator` implementations you see that only the `RankCalculator` passes `false` to its super constructor. Indeed, a player is considered better if it obtains a *lower* average rank.

Exercises

1. Implement a score system based on the number of supply centers, but in which the score of Russia is multiplied by $\frac{3}{4}$ (because Russia starts with 4 units, while all other powers start with 3 units).
2. Implement a score system based on the number of supply centers conquered, but in which the supply centers conquered in the last 3 games count double.

3. Implement a score system that orders players purely based on their names. The lower a player's name comes in alphabetical order, the higher it is ranked. Note that since this is completely independent of the results a player obtained in the games, the implementation of `calculateGameScore()` is in this case totally irrelevant.
4. Run a tournament in which the players are ranked according to their score returned by the Score Calculator you implemented for Exercise 1. Use the Score Calculator of Exercise 2 as a first tie breaker, and use the one of Exercise 3 as a second tie breaker.

6 Negotiations

6.1 The Negotiation Protocol

BANDANA provides a default multilateral negotiation protocol. The protocol can be changed, but we will not go into that in this tutorial. Unlike the common Alternating Offers Protocol, in our protocol agents do not take turns. This means that any agent is allowed to make any proposal or accept any proposal whenever it wants.

The protocol involves a special agent, which we call the *Notary* agent (or sometimes the *Protocol Manager*), which monitors the negotiations. The Notary is started automatically when the negotiation server is started. If an agent has made a proposal and all other agents that are involved in the proposal have accepted that proposal then the Notary agent will send a 'confirm' message to all agents involved in the proposal. This message should be considered the official confirmation that the agreement has become binding.

If your agent has accepted a proposal, but later changes its mind, it can send a reject message to withdraw from the proposal and hence prevent it from becoming confirmed. However, if this is done after the Notary has already sent a confirm message for that proposal, then your agent is too late. The reject message will be ignored, and your agent is still committed to the agreement.

Even though we say that proposals are 'officially' confirmed by the Notary, it is important to remark that there is no mechanism to control that players indeed obey the agreements they make. Therefore, one should always take into account that some players break their promises and when implementing an agent it is recommended to keep track of which players have broken their promises, so you can make sure your agent will no longer negotiate with them.

6.2 Deals You Can Propose

The BANDANA framework allows you to propose deals that specify the following two components:

- A (possibly empty) set of *Order Commitments*
- A (possibly empty) set of *Demilitarized Zones*.

Definition 1. *An **Order Commitment** oc is a tuple: $oc = (y, \phi, o)$, where y is a 'year' (an integer number higher than 1900), and ϕ is a phase i.e. $\phi \in \{Spring, Summer, Fall, Autumn, Winter\}$ and o is any legal order.*

An order commitment is a promise that a power will submit a certain order during a certain phase and year. For example: *"The army in Holland will move to Belgium in the Spring of 1902"*. However, if the order is a HLDOrder, then the corresponding power is still allowed to submit a SUPOrder or SUPMTOOrder for that unit instead of the HLDOrder.

Definition 2. *A **Demilitarized Zone** dmz is a tuple: $dmz = (y, \phi, A, B)$ with y and ϕ as above, A a nonempty set of Powers:*

$$A \subset \{Austria, England, France, Germany, Italy, Russia, Turkey\}$$

and B a nonempty set of Provinces.

A Demilitarized Zone consists of a phase, a year, a set of Powers and a set of Provinces, with the interpretation that none of these Powers is allowed to enter (or stay inside) any of these Provinces during that phase and year. For example the Demilitarized Zone $(1903, Fall, \{FRA, GER, ENG\}, \{NTH, ECH\})$ has the interpretation *"In the Fall of 1903 FRA, GER, and ENG will not enter the North Sea and will not enter the English Channel"*.

Definition 3. *A **Deal** d is a set:*

$$d = \{oc_1, \ldots oc_n, dmz_1, \ldots dmz_m\}$$

where each oc_i is an Order Commitment, each dmz_i is a Demilitarized Zone, and with $n \geq 0$, $m \geq 0$, and $d \neq \emptyset$.

When a deal is confirmed by the Notary, the interpretation is that all powers involved in the deal agree that all Order Commitments in the deal and all Demilitarized Zones in the deal will be respected.

A proposed deal can only be accepted or rejected in its entirety. It is not possible to only accept part of the deal. In the case you wish to accept

only a part of the deal, you simply need to propose a new deal which only consists of those Order Commitments and Demilitarized Zones you wish to include in it. For example, if Austria proposes a deal $d = \{oc_1, oc_2\}$ to Germany, then Germany can choose to either accept d or to reject d, but cannot choose to only accept $\{oc_1\}$. Instead however, Germany can decide to make a new proposal $d' = \{oc_1\}$ to Austria, so then it is up to Austria to determine whether to accept or reject d'.

Deals in the BANDANA framework are represented by objects of the BasicDeal class, which implements the abstract Deal class. Currently, BasicDeal is the only implementation of Deal but in the future we may allow different kinds of deals than the ones described above. Also, we may in the future allow users to define their own Deal classes so that you can use the BANDANA framework without being bound to the specific type of deal described above.

6.3 The RandomNegotiator

As you have seen, the BANDANA framework comes with an agent called RandomNegotiator. The source code is included in the package. This agent is just the RandomBot extended with negotiation capabilities. We will now take a look at how it works.

Note that, apart from adding negotiating capabilities, we have also added some other functionality, such as a logger for debugging. Furthermore, we have added two important fields:

```
DiplomacyNegoClient negoClient;
ArrayList<BasicDeal> confirmedDeals = new ArrayList<BasicDeal>();
```

The `negoClient` field is the client that will connect to the negotiation server. This field is essential if you want your agent to negotiate via the BANDANA negotiation server. The second field is a list that we will use to store the deals that have been confirmed by the Notary.

6.3.1 Connecting to the Negotiation Server

In order to negotiate we need to connect to the negotiation server, which is done in the `init()` method of the `RandomNegotiator`:

```
this.negoClient.connect();
this.negoClient.waitTillReady();
if(this.negoClient.getStatus() == STATUS.READY){
//successfully connected to the server.
```

```
...
}else{
//connection failed.
...
}
```

The first method establishes a connection with the negotiation server in a separate thread. The second method lets the current thread hang until either the connection is established, or the connection has failed. With the if-else statement we can check whether the client has connected to the server successfully. It is not possible to connect to the negotiation server before the `init()` method is called. This is because the negotiation client needs to know which Power you are playing and this information is only available once the `init()` method is called.

6.3.2 Negotiating

The `play()` method of the RandomNegotiator is identical to the `play()` method of the RandomBot, except that we have added a call to a `negotiate()` method for SPR and FAL seasons. The RandomNegotiator does not negotiate during the other seasons, but you can implement your own agent to do so.

The `negotiate()` method contains a loop that runs for a couple of seconds, and which consists of two parts. The first part handles incoming messages, while the second part searches for proposals to make.

When we have received a message, we can check what type of message it is by calling `receivedMessage.getPerformative()`. If the message is a proposal, then we can extract the proposed deal from that message using the following line:

```
DiplomacyProposal receivedProposal = (DiplomacyProposal)receivedMessage.getContent();
```

We can then analyze the proposal and determine whether we want to accept it or not. You can then get the orders and Demilitarized Zones of the deal as follows:

```
BasicDeal deal = (BasicDeal)receivedProposal.getProposedDeal();
List<DMZ> dmzs = deal.getDemilitarizedZones();
for(DMZ dmz : dmzs){
List<Power> powers = dmz.getPowers();
List<Province> provinces = dmz.getProvinces();
//TODO: decide if we like this DMZ or not.
}
```

```
for(OrderCommitment orderCommitment : deal.getOrderCommitments()){
Order order = orderCommitment.getOrder();
//TODO: decide if we like these orders.
}
```

Note however, that in the RandomNegotiator we have added some extra code in these loops to check that the Demilitarized Zones and Order Commitments are not outdated. With this we mean for example an Order Commitment that proposes to make a certain move during Spring 1902, while the game is already in the Fall 1902 phase, so it cannot be obeyed any more. This may for example happen because the message has arrived too late, or simply because there is some other agent that makes senseless proposals. It does not make sense to accept such a proposal, so we can ignore it. For this purpose we have implemented the `isHistory()` method that returns `true` if and only if the specified year and phase lie in the past with respect to the current year and phase of the game. Note that since the proposal must be accepted in its entirety the whole proposal becomes senseless even if only one of its Order Commitments or Demilitarized Zones is outdated.

Each proposal automatically gets an ID assigned to it when it is proposed. You need this ID to accept the proposal, which is done by the following line:

```
this.negoClient.acceptProposal(receivedProposal.getId());
```

The RandomNegotiator simply accepts the proposal with a probability of 50% regardless of the contents of the proposal.

You can call `receivedProposal.getParticipants()` to get a list of names of all powers that are involved in the deal. All these powers need to accept the deal before it becomes confirmed.

When the RandomNegotiator receives a confirm message it means that it is committed to fulfil its part of the confirmed proposal. Therefore it stores the confirmed deal in a list:

```
confirmedDeals.add(confirmedProposal.getProposedDeal());
```

This way, once it must decide its moves to make, it can access the confirmed deals and make sure that the orders it chooses are consistent with the confirmed deals.

6.3.3 Checking Confirmed Deals are Still Valid

Note that at the beginning of the `play()` method we have included some code to check that previously confirmed deals are still valid. A deal is invalid

if it includes an order for a unit that is no longer possible to execute, because the game evolved in a way different than expected.

We can illustrate this with the following example:

1. AUS plans to move his unit in Holland into Belgium during the SPR 1902 phase.
2. AUS agrees with GER that it will use that same unit in the following FAL 1902 phase to give support from Belgium to some unit of GER.
3. However, if during the SPR 1902 phase the unit in Holland fails to move to Belgium (e.g. because it is blocked by FRA), then in the next phase AUS cannot give the promised support.

In that case we say the agreement has become invalid and no longer needs to be obeyed. We check whether a confirmed deal is still valid by calling `Utilities.testValidity(game, confirmedDeal)`.
If this method returns `null` it means the deal is still valid and must be obeyed. If it returns a String however, it means that it is no longer valid, so we can ignore it. The returned String is a description of why the deal is invalid. This may be useful for debugging purposses.

6.4 Rejecting Proposals

You can reject an incoming proposal by calling

```
this.negoClient.rejectProposal(proposal.getID());
```

There are three reasons why you may wish to reject a proposal.

The first is simply to communicate to the other agents that you are not interested in a proposal that you received. Note that if this is the case it is not required to send a REJECT message. Instead, you may also choose to simply ignore the proposal. However, it may be helpful for your allies if you explicitly reject it.

Another case where you may wish to send a REJECT message is when you first make a proposal, or accept an incoming proposal, but then later change your mind. If the Notary receives your REJECT message before all other agents involved in the deal have accepted it you prevent the deal from becoming confirmed. If you send this message to late however, and the deal has already been confirmed by the Notary, then there is nothing you can do anymore. The deal has definitively become a binding agreement.

The third case in which you may want to send a REJECT message is when you have made two or more inconsistent proposals and one of them becomes confirmed. In that case you should reject the other proposals,

because otherwise they may also become confirmed and you will not be able to obey them all.

Note however, that normally this third case is not relevant, because the Notary already checks consistency and only confirms deals that are consistent with earlier confirmed deals (see Section "Inconsistent Deals"). Therefore, this third case is only relevant if you have disabled the Notary's consistency checking mechanism.

6.5 Searching for Profitable Deals

Finding good deals to propose is probably the hardest task when implementing a negotiating Diplomacy player. The RandomNegotiator however simply generates a random deal (in the method `generateRandomDeal()`) and proposes it. A deal is proposed by calling `negoClient.proposeDeal(newDealToPropose);`

6.6 Obeying the Confirmed Deals

As explained, when all players involved in a proposed deal accept the proposal, and none of them has rejected it after accepting it, then the Notary sends a 'CONFIRM' message to all players involved in the deal. From this moment the deal is considered official, so all involved players are expected to obey the agreement (although we can never be sure that they really will).

In general, whenever another player accepts a proposal in which you are also involved, you receive an ACCEPT message. However, when all but one of the involved players have already accepted it, and the last player finally also accepts it, then the Notary will send a CONFIRM message. You will not receive the last ACCEPT message, but instead you will just receive the CONFIRM message.

We have adapted the method `generateRandomMoveOrders()` to make sure that it only generates orders that are consistent with its confirmed agreements. That is, we have added some code that collects all provinces we are not allowed to enter during the current phase, and all orders that we are committed to. Then, when generating orders, we make sure that we don't generate any orders for units that are already committed to an order, and that the orders we do generate do not move into any demilitarized province.

Note however that we have not adapted the methods `generateRandomBuildOrders()`, `generateRandomRetreatOrders()` and `generateRandomRemoveOrders()` to obey confirmed deals. We leave this as an exercise.

Furthermore, if a deal has OrderCommitments and/or Demilitarized Zones for more than one round of the game, and one player did not obey his part of the agreement in the first round of the agreement, then you may also want to ignore your part of that agreement for the future rounds. For example:

1. AUS and GER agree that AUS will move his unit from Bohemia to Galicia in the Spring of 1904, and that GER will move his unit from Tyrolia to Venice in the Fall of 1904.
2. In Spring 1904, AUS does not obey this agreement. He moves his unit from Bohemia to Munich.
3. In Fal 1904, GER may now also decide not to obey his part of the agreement and instead move his unit from Tyrolia to Trieste.

In order to check whether the other players have obeyed their agreements the Player class provides the `receivedOrder()` method. This method is automatically called after the players have submitted their moves, and is called once for each order submitted by any other player.

6.7 Inconsistent Deals

Agent may make several proposals that are inconsistent with each other. For example, in one deal it proposes that its army in Belgium will move to Holland, while in in another deal it proposes that that its army in Belgium will move to Picardy.

Proposing inconsistent deals in not a problem. However, once all agents involved in a proposed deal have accepted it, before sending a CONFIRM message, the Notary will check that this deal is consistent will proposals that have been confirmed earlier and are still valid. If this is not the case the Notary will simply not send a CONFIRM message for this new deal and the deal is not considered a binding agreement. Therefore, you can always be sure that the proposals that have been confirmed by the Notary are all consistent with each other.

If you want to set up a tournament in which the Notary does not not check whether deals are consistent or not then you can do that by adding the following line

```
NegoServerRunner.ENABLE_CONSISTENCY_CHECKING = false;
```

before the line

```
NegoServerRunner.run();
```

in the TournamentRunner.

6.8 Sending Informal Messages

Apart from proposing, accepting and rejecting deals, the BANDANA also allows you to send *informal* messages.

An informal message can simply be anything. The content is completely ignored by the Notary, and therefore has no formal meaning. You can use this if you want to allow any kind of communication between the players other than the formal negotiation protocol.

We have put an example of such a message in the `start()` method of the RandomNegotiator:

```
List<Power> receivers = game.getPowers();
this.negoClient.sendInformalMessage(receivers, "Hello World! I am " + me.getName());
```

If you want to set up a tournament in which you do not want the players to use informal messages you can disable this possibility by including the line

```
NegoServerRunner.ALLOW_INFORMAL_MESSAGES = false;
```

before the line

```
NegoServerRunner.run();
```

in the TournamentRunner.

Exercises

1. The RandomNegotiator randomly chooses to accept a proposal or not. Instead, change the code such that it will always accept any proposal that contains an order for one of its own units to move to a Supply Center it currently does not own.
2. Instead of randomly proposing deals, try to find a deal in which one its own units moves into a Supply Center, and one of its opponents' units supports that move.

7 Building a Negotiating Agent on top of the D-Brane's Tactical Module

Arguably the most important contribution of the BANDANA framework is the ability to build your own negotiation algorithm on top of the existing tactical module of D-Brane. This allows you to do research on Negotiations without having to worry about the underlying tactical aspects of the game.

D-Brane is a high-quality Diplomacy player that has won the Computer Diplomacy Challenge at the 2015 ICGA Computer Olympiad.

In order to demonstrate how this works we have added the DBraneExampleBot to the framework, of which you can find the source code in the folder src/ddejonge/bandana/exampleAgents.

Note that this class looks very similar to the RandomNegotiatior. However, we have added a field called dBraneTactics. Whenever you pass a list of deals to this object, it will try to find the best list of orders for units. Of course there is no guarantee that the returned list is the theoretically best set of orders, it will most often be very good. It is used as follows:

```
Plan plan = dbraneTactics.determineBestPlan(game, power, deals, allies);
List<Order> myOrdersToSubmit = plan.getMyOrders()
```

Here, game is simply the game object that represents the current state of the game. The given power is the Power for which you want to obtain the set of moves. Normally you would pass the power that you play, but you can also pass any other power, for example if you want to predict what your coalition partners might do given these agreements. The argument 'deals' can be any list of BasicDeal objects, and 'allies' is a list of powers that you consider your allies.

The Plan object that is returned contains a list of orders for the given power, such that they all obey the given deals and such that non of your units invades any Supply Center currently owned by any of the given allies. The idea here is that this list of orders is the set of deals that maximizes the number of Supply Centers you conquer in that round. In other words: it is a greedy player that does not think ahead more than one step at a time. Although this may seem a very naive strategy, it turns out to play better than most other existing Diplomacy bots. You can get the number of Supply Centers the DBraneTactics object expects to conquer by calling:

```
int numberOfConqueredSCs = plan.getValue();
```

We should note however that this is only an *expected* number. There is no guarantee that this number is correct (although usually it should at least be a correct lower bound, assuming all agents obey the agreements).

In the `play()` method of the DBraneExample bot we demonstrate how the DBraneTactics object can be used after the negotiations to get the best plan that obeys the made agreements. However, it can also be used by the negotiation algorithm itself. This is demonstrated in the method `searchForNewDealToPropose()`. In this method we first determine what

happens if we do not make any new agreements, by asking DBraneTactics for the best plan under the already confirmed agreements. Next, we generate a number of random deals, and for each such deal we ask DBraneTactics what would be the best plan if that random deal were also confirmed. We finally return the random deal for which the returned plan gives us the highest expected number of conquered Supply Centers (unless this number is not higher than if we make no new deal at all, in which case we return `null`).

Of course, the number of conquered Supply Centers in the current round is only a very rough indication of the quality of the deal, so instead you may want to write a more sophisticated algorithm to determine its quality, using the list of orders returned by `plan.getMyOrders()`

8 Useful Tools

In this section we describe some tools that may come in handy when implementing your own Diplomacy bot.

8.1 The Internal Adjudicator

When implementing an agent you may encounter the problem that you would like to know for a given game configuration, a given set of orders for your units, and a given set of orders for the units for your opponents, what the outcome of those orders will be. The process of determining the outcome of a set of orders is known as *adjudication*. Unfortunately, this task is far from trivial, as the rules of Diplomacy are fairly complex.

Luckily however, we have added an adjudicator for you to the BANDANA framework[2]. It works in a very simple way. First you create an object of the InternalAdjudicator class. Then, every time you wish to use it, you call its `clear()` method to reset it, then you call `clear(game, listOfOrders)` to execute the adjudication process. After that, you can call `getResult(order)` for each order. This will return `true` if the order succeeded, and `false` if the order failed.

More precisely, if a MTOOrder succeeds, it means that the unit will indeed move to the intended province. If it fails it will either stay in the same place, or it is dislodged (it is kicked out of its current location by an opponent, and will have to retreat in the next SUM or AUT phase). For

[2]The implementation of the adjudicator is based on this article: `http://diplom.org/Zine/S2009M/Kruijswijk/DipMath_Chp1.htm` by Lucas Kruijswijk

a HLDOrder, if it succeeds it will stay in its current location, and will be dislodged if it fails.

For a SUPOrder or SUPMTOOrder if it succeeds it will stay in its current location. However, if it fails it may or may not be dislodged. The fact that it failed means that it didn't manage to give support to the supported unit, because the support was cut. However, it does not tell you whether it successfully managed to stay in its current location or if it was dislodged.

For failed SUPOrders, SUPMTOOrders and MTOOrders to know whether it was dislodged or not you have to check whether there was some other unit that successfully moved into its location.

For an example of how to use the InternalAdjudicator, please take a look at the source code of the AdjudicatorExampleBot that we have included in the BANDANA package. This agent is almost identical to the RandomBot, except that in SPR and FAL phases it now generates a set of orders for *all* agents instead of only for itself. It then uses the InternalAdjudicator to check how many supply centers it gains if those orders are indeed submitted by the players. It repeats this process several times and finally picks the set of orders that yields the highest numbers of Supply Centers and will play his moves from that list (of course, there is no guarantee that this strategy works because the opponents will likely submit completely different orders, but it just serves as an example of how the adjudicator works).

We should note that the InternalAdjudicator does not check whether an order is legal or not. Therefore, it is not suitable to be used inside a game server. It is only intended to be used internally by a single agent (hence the name *internal* adjudicator). The agent itself is responsible for checking that it does not supply this adjudicator with illegal moves (for example an army trying to move into a sea-province, or a player submitting an order for a unit of an opponent).

8.2 The DiplomacyGameBuilder

Another useful tool is the DiplomacyGameBuilder. This tool allows you to generate a custom game, with units positioned on the map any way you like. This can be very handy during the development of your agent when you want to test your algorithms on specific test cases.

In the following example, we create game which is in the 1903 FAL phase, and with three units. We place a Russian fleet in Sevastopol, a French fleet in the Spanish North Coast, and an Italian army in Rome. Furthermore, we set England as the current owner of London. All other regions of the map will remain empty, and all other Supply Centers will not have any owner.

After placing the units and setting the owners of the Supply Centers we create the Game object by calling `createMyGame()`.

```
DiplomacyGameBuilder gameBuilder = new DiplomacyGameBuilder();

gameBuilder.setPhase(Phase.FAL, 1903);

gameBuilder.placeUnit("RUS", "SEVFLT");
gameBuilder.placeUnit("FRA", "SPANCS");
gameBuilder.placeUnit("ITA", "ROMAMY");

gameBuilder.setOwner("ENG", "LON");

Game myGame = gameBuilder.createMyGame();
```

If you want to create the standard board configuration, as it is at the beginning of any standard game, you can simply do the following:

```
DiplomacyGameBuilder gameBuilder = new DiplomacyGameBuilder();
Game myGame = gameBuilder.createDefaultGame();.
```

Note: the game builder may fail if you try to add more than 34 units on the map. Also, it will fail if you do not set the phase and year of the game.

8.3 The Diplomacy Mapper

The Diplomacy Mapper is a tool that displays a visual representation of an ongoing game, so that you can follow the game with your own eyes. Furthermore it even provides an interface for humans to play, so you can play against your own agents.

This tool however is not part of BANDANA or DipGame, as it was developed by others. Nevertheless, you can still use it perfectly in combination with BANDANA, and you can download it from here:
`http://www.ellought.demon.co.uk/dipai/`
Unfortunately, it only works on Windows.

9 Tips, Tricks and Warnings

9.1 Using HashMaps

The Power, Region and Province classes of the DipGame framework unfortunately do not override the `hashcode()` method while they do override the

`equals()` method (two Powers, Regions, or Provinces are considered equal if they have the same name). This can lead to problems if you want to you use any of those classes as the Key class in a HashMap. In order to avoid such problems you can use String as the Key class instead, and use the name of the Power, Region or Province object as the key.

9.2 Proposing Draws

When your agent proposes a draw by calling the `proposeDraw()` method the proposal is sent via the game server and not via the negotiation server. Therefore, your agent can propose draws even if it does not have the `negoClient` field or if it is not connected to the negotiation server.

Furthermore, note that for this reason the protocol for draw proposals is a bit different from the protocol described in Section 6.1. Specifically, once you have proposed a draw you cannot reject it anymore. Also, when another player proposes a draw you are not notified of this, and you cannot decide to accept or reject it. Instead, you simply decide to propose a draw whenever you want, and the server declares a draw if and only if all players have proposed a draw in the same round.

9.3 Play Diplomacy Online

No matter how well you understand the game theoretically, we think it is absolutely essential that you also play the game a couple of times yourself before you will be able to implement a good player. An excellent place to start playing Diplomacy is `http://www.playdiplomacy.com/`. Here you can play online with people from all over the world.

10 Contact

If you still have any questions about Diplomacy, or BANDANA, please do not hesitate to contact us. Also if you have any suggestions on how to improve the BANDANA framework or this manual, then we are happy to hear from you. You can contact us by sending an e-mail to davedejonge@iiia.csic.es