UNIVERSITAT POLITÈCNICA DE VALÈNCIA


Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

# Entorno de negociación automática bilateral

**Sistemas Multiagente**

José Ángel González Barba – jogonba2@dsic.upv.es

# Índice

# 1. Introducción

## 1.1 Proyecto

Se trata de un entorno de negociación automática bilateral, desarrollado para la asignatura de Sistemas Multiagente, que facilita el desarrollo de agentes negociadores. Con este entorno, se proporcionan funciones de utilidad, estrategias de concesión, criterios de aceptación, métodos de generación de ofertas y modelos de aprendizaje automático para predecir la respuesta del contrario ante una oferta.

Además, es posible implementar otros tipos de estrategias, de forma sencilla, mediante la inserción de código en los scripts del proyecto. Las estructuras de datos, los parámetros que recibe cada estrategia y la forma de implementar estrategias se discuten en apartados posteriores. Con ello, los scripts mencionados son:

- **GenerateOffers.py**: es el script donde se definen las estrategias de generación de ofertas. Con el entorno se proporcionan varias estrategias, a destacar:
    - **Métodos basados en optimización de la función de utilidad**:
        - **Hyperopt**: hace uso de árboles de estimadores de Parzen (**TPE**) para optimizar hiperparámetros [1]. La implementación dada no permite manejar conocimiento del oponente (no se ha implementado por falta de tiempo).
        - **Scipy**: se emplean los optimizadores de la librería **Scipy** [2] que permiten manejar restricciones (en apartados posteriores se muestra la necesidad de ello) para optimizar la función de utilidad. La implementación proporcionada no está completa pero puede servir como base para continuar.
        - **Algoritmo genético**: adaptación del algoritmo genético, implementado en una asignatura anterior, aplicado a regresión simbólica. En este caso, se emplea para optimizar la función de utilidad y solo permite manejar valores enteros y categóricos. No garantiza que la utilidad de la oferta generada supere el valor de concesión **s** en un step determinado, para esto es necesario ajustar parámetros (dependientes del problema). Esta implementación permite utilizar conocimiento del oponente.
    - **Otros métodos**:
        - **Aleatorio**: genera ofertas con valores aleatorios para cada atributo, dentro de los intervalos permitidos. La implementación proporcionada también permite manejar conocimiento del oponente.

- **ReceiveOffers.py**: en este script se implementan las estrategias de recepción de ofertas. En la implementación dada, se proporciona una estrategia de ejemplo (**receive_max_offers**) para aceptar la oferta que maximiza el valor de utilidad.

- **Utility.py**: en este script se definen las funciones de utilidad que permiten modelar las preferencias de los agentes. Con el entorno solo se proporciona la implementación de una función de utilidad lineal, que es el modelo de preferencia multiatributo más común, pero no es capaz de modelar dependencias complejas entre atributos.

- **Concession.py**: se definen las estrategias de concesión a utilizar por los agentes. Estas estrategias determinan cómo, cuánto y cuándo concede un agente (modifica su valor de aspiración). Las estrategias de concesión implementadas son:

    - **Concesión temporal**
    - **Toma y daca**
        - **Relativo**
        - **Absoluto**
        - **Promediado**
    - **Concesión aleatoria**
    - **No concesión**

- **Utils.py**: se emplea para añadir todo el código auxiliar necesario para definir las estrategias e.g. cargar información del oponente o preprocesarla.

- **Acceptance.py**: en este script se definen los criterios de aceptación, empleados para determinar cuándo un agente acepta una oferta determinada.

- **Classifier.py**: en el script se añaden clasificadores que permiten resolver problemas de clasificación o regresion dependientes de información del oponente. Principalmente, en el entorno proporcionado, se utilizan durante la generación de ofertas para determinar si el agente oponente aceptará o no una oferta, pero se pueden utilizar en cualquier parte desarrollada por el usuario p.e. en la definición de criterios de aceptación. Se proporcionan los clasificadores del *framework* **scikit-learn** [3]:

    - **Multilayer perceptron**
    - **Naïve Bayes (Gaussian)**
    - **Naïve Bayes (Multinomial)**
    - **Naïve Bayes (Bernoulli)**
    - **SVM**
    - **AdaBoost**
    - **Random Forest**
    - **Decision Trees**
    - **Nearest Centroid**
    - **K Nearest Neighbours**

- **Statistics.py**: en este script se definen las estadísticas utilizadas para evaluar a los agentes tras la ejecución de un torneo. Se utilizan los estadísticos proporcionados por **numpy** [4].

    - **Media artimética**

    - **Mínimo**
    - **Máximo**
    - **Desviación típica**
    - **Primer percentil**
    - **Segundo percentil**
    - **Tercer percentil**

- **Agent.py**: es la clase que representa los agentes, para ello, carga la definición de sus parámetros (información, dominio y parámetros de estrategias) desde los ficheros de agentes **json**. En ella se especifican los métodos para generar y recibir ofertas, actualizar el nivel de aspiración, getters y setters de información del agente etc. En principio, no es necesario modificar este fichero.

- **Exchange.py**: es el script que controla el bucle principal de la negociación. Maneja parámetros como el número de ofertas a emitir en cada *step*, el *path* donde se encuentra la información de los agentes, el *path* dónde almacenar conocimiento sobre los agentes o la forma de mostrar el resultado de la negociación (gráfica y por consola). Además, un detalle a destacar es que el tiempo de negociación se discretiza en pasos de negociación. En principio, tampoco es necesario modificar este script.

- **Graphics.py**: contiene los métodos para graficar el proceso de negociación. Es posible representar todas las ofertas simultáneamente o de forma interactiva siguiendo el orden en el que fueron realizadas. En cualquier caso, la representación de ofertas se realiza al final de la negociación para no ralentizar el proceso.

- **Messages.py**: en este script se definen métodos para generar cualquier cadena a mostrar por la aplicación, principalmente, resultados de los procesos e información de la negociación y de las ofertas. Se agrupan todas las cadenas en una clase para facilitar una posible traducción de la herramienta.

- **Main.py**: es el punto de entrada de la aplicación, carga el fichero de configuración de la herramienta e inicia el proceso de negociación en función de los parámetros de configuración.

Por otro lado, con respecto a la definición de agentes, estos se definen mediante ficheros **json** (según la configuración por defecto, en el directorio **Bots**), donde se indica información del agente, el dominio en el que pretende negociar y los parámetros de las estrategias empleadas por el agente. Se pueden consultar los tres agentes de ejemplo en la carpeta **Bots**.

Por último, durante el desarrollo de esta memoria, se pretende introducir al lector al uso de la herramienta desarrollada, es por ello por lo que primero se comentarán detalles de la arquitectura y por último se llevará a cabo una guía de cómo desarrollar un agente y evaluarlo mediante la negociación con otros agentes negociadores.

## 1.2 Dependencias

El entorno desarrollado requiere ciertas dependencias para poder funcionar. Más concretamente, dichas dependencias son:

- **Numpy**: es utilizado para manipular arrays, estadísticos para torneos y otras funciones numéricas.

- **Scipy**: principalmente empleado para los optimizadores usados durante la generación de ofertas y otras funciones numéricas.

- **Matplotlib**: librería gráfica utilizada para graficar el proceso de negociación.

- **Scikit-learn**: librería utilizada para proporcionar clasificadores de ejemplo en el script **Classifier.py**.

- **Deap**: es un *framework* de computación evolutiva utilizado en la implementación del algoritmo genético de generación de ofertas.

- **Hyperopt**: librería para optimización en espacios de búsqueda complejos formados por valores multidimensionales de tipo real, discreto o condicional. Es utilizada en la generación de ofertas **Hyperopt**.

Para instalar las dependencias mencionadas, se proporciona el script **setup.py**, que permite realizar la instalación mediante el gestor de paquetes **pip** (lo instala automáticamente si el usuario no dispone de él) si se le indica el parámetro **install** (**python setup.py install**). Si la ejecución finaliza por algún error, es recomendable instalar la plataforma **Anaconda** [5] y utilizarla para instalar a mano los paquetes anteriores (**conda install <package>**).

## 1.3 Modos de ejecución

En este apartado se discuten los modos de ejecución y los parámetros que se le pueden pasar a la herramienta (mediante el fichero **config.json**) para modificar ciertos aspectos del proceso de negociación, independientes de los agentes negociadores.

En primer lugar, si abrimos el fichero **config.json** en la raíz del proyecto, se puede observar información sobre el proyecto en el campo **project_info**, este campo únicamente sirve para controlar la versión actual de la herramienta y almacenar información sobre el autor. Modificar este campo no afecta al proceso de negociación y únicamente puede variar la salida por pantalla.

En segundo lugar, encontramos el campo **agents**. En este campo se especifican los agentes que participarán en el proceso de negociación, si solo se consideran dos agentes, será un escenario

de negociación bilateral, mientras que si se añaden más agentes, se llevará a cabo un torneo de negociaciones bilaterales entre todo par de agentes especificados.

Por otro lado, el campo **knowledge** permite controlar cómo va a manejar la herramienta, información de los agentes negociadores, para que, posteriormente, los agentes puedan utilizarla durante su ejecución. Concretamente, los atributos del campo **knowledge** son:

- **corpus_path**: este atributo permite especificar la ruta donde los agentes van a almacenar y cargar información sobre los oponentes (en caso de que la usen). La obtención de información durante el proceso de negociación la controla el núcleo (**Exhange.py**) para eliminar dicha tarea de la parte de los agentes y facilitar el trabajo a los desarrolladores. En apartados posteriores se comenta cómo y qué información almacenan los agentes sobre sus oponentes.

- **save_corpus**: mediante este atributo le indicamos a la herramienta si durante la negociación se va a almacenar conocimiento del oponente o no. El directorio donde se almacenará dicha información viene especificado por el parámetro **corpus_path** anterior, y el fichero con información sobre un determinado agente, toma como nombre el nombre del agente.

El último campo que encontramos en el fichero **config.json** es **params**. Este campo permite indicar algunos de los parámetros que utilizará la herramienta para modificar aspectos del proceso de negociación. Entre estos parámetros encontramos:

- **n_offers**: permite modificar el número de ofertas que se envían en cada paso de la negociación. Por defecto, se envía una única oferta. Si el parámetro es $> 1$, al recibir un conjunto de ofertas, se aceptará la mejor de todas las recibidas.

- **show_offers**: permite mostrar por pantalla, al final del proceso de negociación, las ofertas que han sido enviadas por cada agente en cada paso de negociación.

- **first_random**: si su valor es True, el primero en iniciar la negociación será uno de los dos agentes escogidos de forma aleatoria. Si no, el agente que inicia la negociación es el agente que primero se carga desde el fichero **json**.

- **graphic_process**: se le indica a la herramienta si se debe graficar, al final del proceso, el resultado de la negociación (valores de utilidad de las ofertas propuestas por cada agente, representado en una gráfica 2D, donde los ejes x e y son el nivel de aspiración de los agentes uno y dos, respectivamente.

- **graphic_interactive**: indica la forma de graficar el resultado de la negociación. Hay dos formas implementadas, si el parámetro es False, todas las ofertas se representan de forma estática. Sin embargo, si el valor del parámetro es True, se representarán las ofertas de forma secuencial en el orden en que fueron enviadas.

Un ejemplo de fichero **config.json** válido se muestra en la Figura 1.

```
{

  "project_info" : {


    "version" : "0.4",
    "author" : "JAGB"
  },

  "agents" : {


    "Tyrion" : "./Bots/Tyrion.json",
    "JohnSnow" : "./Bots/JohnSnow.json"

  },

  "knowledge" : {


      "corpus_path":"./",
      "save_corpus" : true
  },

  "params" : {


      "n_offers" : 1,
      "show_offers" : true,
      "first_random" : false,
      "graphic_process" : true,
      "graphic_interactive" : false
  }

}
```

Figura 1. Ejemplo de fichero **config.json** válido.

# 2. Agentes

## 2.1 Definición de agentes

En el presente apartado, se comenta la estructura de un agente, es decir, las diferentes partes que componen el fichero **json** con la definición de dicho agente, en concreto: información sobre el agente, el dominio en el que negociará y los parámetros de las estrategias seguidas por el mismo. También se expondrá la clase Agente, que actúa como un contenedor de estrategias parametrizadas por los atributos de un agente definido en **json**. Por último, se mostrarán y comentarán los agentes de ejemplo proporcionados con la herramienta.

### 2.1.1 Información

El primer campo de un fichero de agente **json**, **agent_info**, permite almacenar información asociada al agente. La mayoría de esta información no es usada por la herramienta, pero sería interesante emplearla como conocimiento del oponente para algunas estrategias. Actualmente, la información considerada es:

- **Name**: permite identificar a los agentes, es utilizado en los *logs*, para almacenar ficheros con conocimiento sobre un agente etc.
- **Gender**: permite indicar el género del agente. Este campo no es utilizado actualmente por la herramienta.
- **Country**: permite indicar el país del agente. Este campo no es utilizado actualmente por la herramienta.
- **Birthday**: permite indicar la fecha de creación del agente. Este campo no es utilizado actualmente por la herramienta.
- **Reputation**: permite indicar la reputación del agente. Este campo no es utilizado actualmente por la herramienta.
- **Description**: contiene una definición del agente. Este campo no es utilizado actualmente por la herramienta.
- **Domain**: contiene el nombre del dominio en el que un agente puede negociar. El campo no es utilizado actualmente por la herramienta, pero puede ser útil para identificar agentes interesados en dominios similares.

Un ejemplo de información del agente se muestra en la Figura 2.

```
"agent_info" : {
    "name": "Tyrion",
    "gender": "male",
    "country": "spain",
    "birthday": "16/08/91",
    "reputation": 0.3,
    "description" : "tyrion, interested in ...",
    "domain" : "car sale"
}
```

Figura 2. Información del agente en el fichero **json**.

Ninguno de los campos es obligatorio, aunque sí es recomendable definirlos siempre (en especial el nombre) porque se pueden utilizar en ciertas situaciones, como por ejemplo al emplear conocimiento del oponente. Sí que es obligatorio definir el campo **agent_info** aunque no se especifiquen algunos atributos en él. También será obligatorio definir los demás campos asociados con el dominio y con las estrategias básicas de los agentes, esto se discute en los dos siguientes apartados.

### 2.1.2 Dominio

El dominio en el que participa un agente es una parte imprescindible en el proceso de negociación y determina los atributos y las preferencias sobre éstos que utilizarán los agentes en las negociaciones.

Dentro del fichero **json** de un agente, encontramos uno de los campos de obligatoria definición, relativo al dominio. Este campo es **rol** y en él se especifica el fichero que contiene la definición del dominio. Este fichero debe estar formado por tres campos, más detalladamente, estos campos son:

- **attributes**: el atributo es un diccionario que contiene los atributos por los que negociará el agente y el tipo de dichos atributos. Un ejemplo de dicho atributo se muestra en la Figura 3.

```
"attributes" : {
    "price" : "integer",
    "color" : "categorical",
    "abs"   : "categorical"
}
```

Figura 3. Campo **attributes** del fichero **json**.

- **weights_attr**: mediante este atributo se definen las preferencias de los agentes, utilizadas por la función de utilidad (se discutirán dichas funciones en apartados posteriores). En la Figura 4 se muestra un ejemplo de este atributo, donde se observa que la suma de los pesos de los atributos debe ser 1.

```
"weights_attr" : {
   "price" : 0.6,
   "color" : 0.2,
   "abs"    : 0.2
},
```

Figura 4. Campo **weights_attr** del fichero **json**.

- **values_attr**: controla el valor que toman los atributos dentro de ciertos intervalos. A su vez, está compuesto por un diccionario con tres valores: **integer**, **float** y **categorical** donde se definen los atributos de tipo entero, real y categórico respectivamente. Dichas definiciones de atributos están formadas por dos campos más:

    - **properties**: definen información sobre el dominio de los tipos. Si el tipo es **integer** o **float**, se debe especificar el valor mínimo y el máximo. Si por el contrario, el tipo es categórico, se deben enumerar las posibles elecciones (**choices**). En este ultimo caso, es recomendable utilizar valores enteros como posibles elecciones (sobretodo si se va a utilizar conocimiento del oponente), pero también es posible utilizar cadenas en algunas ocasiones (no se han testeado). En la Figura 5 se muestra un ejemplo de cada tipo (**integer** y **categorical**).

```
"values_attr" : {
 "integer"     : {
    "price" : {
      "properties" : {
         "max" : 3000,
         "min" : 0
       },
              ...
    },
      ...
  },
   ...
}
```

```
"values_attr" : {
  "categorical" : {
    "color" : {
      "properties" : {
        "choices" : [0, 1, 2]
       },
            ...
    },
      ...
  },
   ...
}
```

Figura 5. Campo **properties** del campo **values_attr** para los tipos de datos.

    - **conditions**: en este campo se especifican los intervalos o elecciones categóricas que determinan el valor de un atributo si la cantidad de este se encuentra dentro de dichos intervalos o coincide con las elecciones especificadas.

En el caso de atributos numéricos (**integer** y **float**)**,** es necesario especificar intervalos continuos (el valor máximo del intervalo $i$ debe coincidir con el valor mínimo del intervalo $i + 1$). Además, el valor mínimo del primer intervalo debe coincidir con el valor especificado en el campo **min** de **properties** y el valor máximo del último intervalo debe coincidir con el valor del campo **max** de **properties**.

Así, las condiciones sobre atributos numéricos, se especifican de la forma mostrada en la Figura 6. Donde cada intervalo se define como un par **<identificador> : [$\mathbf{min}_i$, $\mathbf{max}_i$, utilidad]**.

```
"conditions" : {
    "cond_1" : [0, 1000, 1],
    "cond_2" : [1000, 2000, 0.5],
    "cond_3" : [2000, 7500, 0.3],
    "cond_4" : [7500, Infinity, 0]
}
```

Figura 6. Definición de intervalos sobre atributos numéricos.

Si el atributo es categórico, las condiciones se definen como un par **<identificador> : [elección, utilidad]**. Tal como se muestra en la figura 7.

```
"conditions" : {
    "cond_1" : [0, 0.4],
    "cond_2" : [1, 0.1],
    "cond_3" : [2, 1]
}
```

Figura 7. Definición de condiciones sobre atributos categóricos.

Con todo ello, un ejemplo completo de especificación de un dominio, en este caso de venta de coches, se muestra en la Figura 8, donde se pueden identificar las diferentes partes ya comentadas.

```
"attributes" : {"price" : "integer", "color": "categorical", "abs":
"categorical"},

"weights_attr" : {
   "price" : 0.8,
   "color" : 0.1,
   "abs"   : 0.1
},

"values_attr" : {
    "integer"     : {
        "price" : {
            "properties" : {
                "max" : 6000,
                "min" : 0
            },
```

```
            "conditions" : {
                "cond_1" : [0, 1000, 0.1],
                "cond_2" : [1000, 5000, 0.7],
                "cond_3" : [5000, Infinity, 1]
            }
        }
    },

    "float"       : {},

    "categorical" : {
        "color" : {
            "properties" : {
                "choices" : [0, 1, 2]
            },
            "conditions" : {
                "cond_1" : [0, 0.1],
                "cond_2" : [1, 0.5],
                "cond_3" : [2, 1]
            }
        },
        "abs" : {
            "properties" : {
                "choices" : [0, 1]
            },
            "conditions" : {
                "cond_1" : [1, 1],
                "cond_2" : [0 , 0]
            }
        }
    }
}
```

Figura 8. Especificación completa del dominio dentro del fichero **json** de definición del agente.

### 2.1.3 Parámetros de estrategias

El siguiente campo del fichero de definición de un agente es **params**. En él se especifican los parámetros para las diferentes estrategias utilizadas por el agente durante el proceso de negociación. Un ejemplo de este campo se muestra en la Figura 9.

```
"params" : {

  "general" : {
    "ur": 0.45,
    "revoke_step": 200,
    "s": 0.75,
    "window_offer": 0.1,
    "use_knowledge" : true,
    "upper_bound_knowledge": 5000
  },
```

```
   "strategies": {
       "utility_type": {
         "module" : "Utility",
         "func" : "linear",
         "params" : {}
       },

       "concession_type": {
         "module" : "Concession",
         "func" : "behavioural_averaged_concession",
         "params" : {
           "delta": 2,
           "beta": 0.8
         }
       },

       "acceptance_type": {
         "module" : "Acceptance",
         "func": "rational",
         "params" : {}
       },

       "generate_offer_type": {
         "module" : "GenerateOffers",
         "func" : "hyperopt_offer",
         "params" : {
           "max_eval" : 15
         }
       },

       "receive_offer_type": {
         "module" : "ReceiveOffers",
         "func" : "receive_max_offers",
         "params" : {}
       },

       "ml_model": {
         "module" : "Classifiers",
         "func": "mlp_nn_classifier",
         "params": {
           "solver": "lbfgs",
           "alpha": 1e-5,
           "hidden_layer_sizes": [128, 4],
           "random_state": 1
         }
       }
    }
}
```

Figura 9. Ejemplo de campo **params** en la definición de un agente.

En el entorno proporcionado, el campo **params** del fichero de definición de agentes, hace referencia a todas las estrategias y valores auxiliares de los que hace uso el agente durante el proceso de negociación. Por tanto, aquí se indicarán valores iniciales de algunos parámetros obligatorios (campo **general**) y las estrategias con los parámetros constantes que se requiera (la forma de pasar parámetros adicionales que varían durante el proceso de negociación se muestra mediante ejemplos en el siguiente capítulo).

De forma detallada, los parámetros de un agente son:

- **general**: el campo general agrupa un conjunto de atributos básicos y generales del agente en el proceso de negociación. Entre estos atributos encontramos:

    - **ur**: requiere un valor real para indicar la utilidad de reserva de un agente. Es la aspiración mínima de un agente en una negociación.

    - **revoke_step**: requiere un entero que indica el límite de rondas, para un agente, en una negociación.

    - **s**: requiere un valor real para indicar el nivel de aspiración inicial de un agente en una negociación.

    - **window_offer**: requiere un valor real para indicar en las estrategias de generación de ofertas que se deben generar ofertas entre [**s**, **s+window_offer**]. Si **s+window_offer** > 1, el valor se debe truncar a a un valor máximo y, además, es conveniente tener en cuenta el caso **window_offer** < 0. Por ultimo, destacar que el parámetro es constante durante el proceso de negociación, por lo que si se quiere hacer dependiente de algún parámetro es necesario especificarlo en la definicón de la estrategia.

    - **Use_knowledge**: requiere un valor booleano para indicar si el agente utilizará conocimiento del oponente en el proceso de negociación. Se requiere para cargar los modelos de conocimiento del oponente al inicio de dicha negociación.

    - **upper_bound_knowledge**: requiere un valor entero que permite controlar el número de veces que se ha usado conocimiento sobre el oponente en una estrategia dada. Un ejemplo de su uso se puede ver en la estrategia de generación de ofertas **random_offers (Offers.py**) donde se emplea para controlar el número de ofertas generadas teniendo en cuenta conocimiento del oponente.

- **strategies**: en este campo se define qué estrategias utilizar en cada paso de negociación para: calcular el valor de utilidad de las ofertas, generar ofertas, recibir ofertas, estrategia de concesión, criterio de aceptación, clasificar conocimiento del oponente. Para esto, los campos mostrados dentro de **strategies** en la Figura 9 son obligatorios y todos tienen el mismo número de atributos:

    - **module**: requiere un **string** que permite indicar a la herramienta de qué clase cargar una determinada estrategia. No se recomienda cambiar los módulos, se han fijado así para evitar problemas por parte de los desarrolladores de agentes.

    - **func**: requiere un **string** para determinar qué método dentro de la clase especificada en **module** implementa la estrategia deseada.

    - **params**: requiere un diccionario donde se especifican los parámetros constantes que recibirá el método de la estrategia. Tal como se verá en el próximo capítulo, las estrategias reciben un diccionario de parámetros **parameters** que contiene

los parámetros de este campo (más bien reciben todo el **json** de la definición) y otros que evolucionan de forma dinámica conforme avanza el proceso de negociación. Esto permite la construcción de estrategias complejas que hacen uso de un número no prefijado de parámetros, debido a que el diccionario **parameters** actúa como un contenedor donde se puede definir cualquier variable desde cualquiera de las estrategias implementadas.

Con ello, debe quedar claro que cualquier estrategia implementada (de las seis mostradas en el campo **strategies** de la Figura 9, que son obligatorias) debe contener tres campos para indicar el módulo, el método y los parámetros iniciales que recibe la estrategia. Con respecto a las estrategias, las seis obligatorias son:

- **utility_type**: contiene la información asociada a la función de utilidad (módulo, método y parámetros iniciales). En la Figura 9, se define el uso de una función de utilidad lineal implementada en el método **linear** de la clase **Utility** que no recibe parámetros iniciales (no significa que el método no reciba parámetros, simplemente que no requiere parámetros adicionales a los que se encuentran en el diccionario **parameters**).

- **concession_type**: contiene la información asociada a la estrategia de concesión. En la Figura 9, se define el uso de una estrategia de concesión toma y daca promediado implementada en el método **behavioural_averaged_concession** de la clase **Concession** que recibe como parámetros **delta** y **beta** que representan los parámetros $\delta$ y $\beta$ de las estrategias de concesión toma y daca promediado y temporal respectivamente (las expresiones vistas en clases de teoría).

- **acceptance_type**: contiene la información asociada al criterio de aceptación. En la Figura 9, se define el uso de un criterio de aceptación racional (acepta si $utility \geq s$) implementado en el método **rational** de la clase **Acceptance**. Este criterio no recibe parámetros adicionales.

- **generate_offer_type**: contiene la información asociada a la estrategia de generación de ofertas. En la Figura 9, se define el uso de una estrategia de generación de ofertas basada en la optimización de la función de utilidad mediante la herramienta **Hyperopt**, implementada en el método **hyperopt_offer** de la clase **GenerateOffers**, que recibe como parámetro adicional, **max_eval**, para determinar el número de iteraciones de la búsqueda de hiper-parámetros realizada por **Hyperopt**.

- **receive_offer_type**: contiene la información asociada a la estrategia de recepción de ofertas. En la Figura 9, se define el uso de una estrategia de recepción de ofertas implementada en el método **receive_max_offers** de la clase **ReceiveOffers**, que no recibe parámetros adicionales. Esta estrategia, devuelve, si acepta alguna oferta según la estrategia de aceptación empleada, la oferta cuya utilidad es máxima, en cualquier otro caso, devuelve **False**.

- **ml_model**: contiene la información asociada al método de clasificación utilizado para clasificar ofertas con clasificadores entrenados a partir de conocimiento del oponente.

En la Figura 9, se define el uso del clasificador de **scikit-learn** instanciado en el método **mlp_nn_classifier** (perceptron multicapa) de la clase **Classifiers** que recibe como parámetros: el solver utilizado para optimizar los pesos (**solver**), el factor de aprendizaje (**alpha**), el número de neuronas en cada capa (**hidden_layer_sizes**) y la semilla del generador pseudo-aleatorio (**random_state**).

Por último, mencionar que, en el entorno proporcionado, es posible ver varios ejemplos de agentes ya definidos en el directorio **Bots**.

## 2.2 Clase Agente

De forma adicional al fichero de definición **json** de un agente, encontramos la clase Agente en el script **Agente.py**. En esta clase se cargan todos los parámetros especificados en el fichero de definición y se implementan los métodos que utilizarán los agentes durante el proceso de negociación, así como métodos **setter** y **getter** de dichos parámetros y otros métodos auxiliares.

Con el entorno proporcionado hay implementado un conjunto de métodos a utilizar, sin embargo, es posible modificar esta clase para considerar más parámetros de la definición de agentes y nuevos métodos, aunque, en principio, no es necesaria su modificación. Más concretamente, los métodos que actualmente están implementados son:

- **__is_valid**: método privado para la clase agente, permite controlar, en mayor o menor medida, que un agente está correctamente definido en su fichero **json**. Actualmente, solo se comprueba que los campos **agent_info**, **rol** y **params** estén definidos en el fichero de definición del agente.

- **__load_rol_attrs**: otro método privado para la clase, se encarga de leer el fichero **json** con el dominio en el que negociará un agente.

- **__load_parameters**: método privado de la clase **Agent**, se emplea para construir el diccionario de parámetros que se utiliza en todas las estrategias implementadas por el usuario para el agente.

- **__get_attr_strategies**: método privado utilizado para cargar los métodos (direcciones de memoria) en los que se implementan las estrategias especificadas por el usuario.

- **__load_file**: método privado utilizado para cargar campos específicos del fichero de definición **json** del agente.

- **emit_offer**: método que se encarga de llamar a la estrategia de generación de ofertas utilizada por el agente para generar una nueva oferta cuya utilidad se encuentre entre **[s, s+window_offer**]. Al generar la oferta, en la implementación actual, se guarda en una memoria de ofertas enviadas por el agente, además, también se guarda en la

memoria de ofertas **aceptadas** por el agente (se asume que cualquier oferta que el agente envía, sería aceptada por éste si la recibiese del oponente) que se emplea para que el núcleo (**Exhange.py**) almacene conocimiento sobre el agente (notar que esto es equivalente a que el otro agente guardase dicho conocimiento de este agente).

- **emit_n_offers**: método que llama a **emit_offer n** veces para generar **n** ofertas diferentes. Es utilizado cuando a la aplicación se le indica que se deben enviar **n**>1 ofertas en cada **step** de negociación.

- **receive_offers**: método utilizado para procesar las ofertas recibidas por el agente en un **step** de negociación, llamando a la estrategia de recepción de ofertas especificada en el fichero de definición.

- **update_s**: método utilizado para actualizar el nivel de aspiración del agente en cada **step** de negociación. Este método se encarga de llamar a la estrategia de concesión definida para el agente.

- **get_benefits**: este método simplemente es un **wrapper** de la función de utilidad del agente, utilizado por el núcleo de la aplicación para obtener el valor de la última oferta en caso de que se haya alcanzado un acuerdo durante el proceso de negociación.

- **get_valid**: método getter del atributo **is_valid** donde se guarda si un agente está correctamente definido o no.

- **ready**: método que permite determinar si un agente está dispuesto a continuar con la negociación. En la implementación dada, únicamente se comprueba la condición $t < \text{revoke\_step}$, por lo que queda pendiente, definir esta función como parte del agente en su fichero **json** para considerar condiciones más complejas.

- **get_name**: método getter del nombre del agente almacenado en la variable **agent_name**.

- **get_use_knowledge**: método getter del atributo **use_knowledge** que determina si un agente utiliza conocimiento del oponente.

- **get_weights_attr**: método getter del atributo **weights_attr** donde se almacena el campo **weights_attr** del fichero indicado en el campo **rol** dentro de la definición del agente.

- **get_values_attr**: método getter del atributo **get_values_attr** donde se almacena el campo **get_values_attr** del fichero indicado en el campo **rol** dentro de la definición del agente.

- **get_ur**: método getter del atributo **ur** que indica la utilidad de reserva del agente.

- **get_s**: método getter del atributo **s** donde se almacena el nivel de aspiración en cada step de negociación.

- **get_window_offer**: método getter del atributo **window_offer** que contiene el tamaño de la ventana de utilidad a emplear para la generación de ofertas cuyo valor de utilidad se encuentre en el intervalo [**s, s+window_offer**].

- **get_knowledge**: método que devuelve el conocimiento almacenado por el agente durante todo el proceso de negociación. El método **get_knowledge** devuelve los atributos **accepted_offers** y **revoked_offers**, que contienen las ofertas enviadas por el oponente que han sido rechazas o aceptadas por el agente. Este atributo lo utiliza el script **Exchange.py** para almacenar ficheros con conocimiento sobre los agentes.

- **set_t**: método setter del atributo **t** que indica el step en el que se encuentra la negociación. Como ya se ha mencionado, las rondas de negociación no se representan con unidades temporales (p.e. segundos) sino que se emplean valores enteros con incrementos de 1.

- **get_memory_proposal_offers**: método getter del atributo **memory_proposal_offers** que contiene las ofertas emitidas por el agente.

- **get_memory_received_offers**: método getter del atributo **memory_received_offers** que contiene las ofertas recibidas por el agente.

- **set_oponent**: método setter del atributo **oponent_agent** que contiene el nombre del oponente.

- **load_oponent_knowledge**: método que se encarga de cargar conocimiento del oponente, representado como una lista de ofertas (representadas a su vez como diccionarios) y transformarlo en una lista de vectores $S = \{v_1, v_2, \dots, v_N : v_i \in \mathbb{R}^D \ , \ D = |attributes|, N = |offers|\}$ donde cada vector es una representación vectorial de una oferta dada, formado por los valores de los atributos de dicha oferta.

Por último, es conveniente destacar que faltan algunas modificaciones y métodos por implementar debido a la falta de tiempo para continuar el trabajo p.e. añadir los métodos getters y setters restantes, sin embargo, el usuario es libre de modificar cualquier aspecto de la clase tratada en este apartado.

# 3. Desarrollo de un agente

En este capítulo se expone la forma de implementar agentes negociadores en el entorno proporcionado. Más concretamente, se comentan los pasos que se deben llevar a cabo (en cualquier orden) para implementar todas las estrategias usadas en el proceso de negociación por un nuevo agente de ejemplo, diferente a los proporcionados en el directorio **Bots**.

## 3.1 Introducción al desarrollo

Antes de exponer el ejemplo de implementación de un nuevo agente, es necesario comentar la forma en que la herramienta, construye y utiliza los parámetros en todas las nuevas estrategias implementadas. Así, para permitir que toda la información que se genera en el proceso de negociación se pueda utilizar en nuevas estrategias, se ha optado por emplear un diccionario de parámetros que mantiene información estática, definida en el fichero **json** del agente e.g. valores iniciales para parámetros de estrategias; e información dinámica, que se va generando o modificando conforme avanza el proceso de negociación e.g. **s**, **t**, memoria de ofertas recibidas etc.

De forma más detallada, el diccionario de parámetros (**parameters**) contiene dos campos: **static** y **dynamic** donde se almacena información estática y dinámica respectivamente. En primer lugar, el campo **static** contiene tres campos más:

- **info**: contiene el campo **agent_info** del fichero de definición **json**.
- **params**: contiene el campo **params** del fichero de definición **json**.
- **domain**: contiene tres campos más, **weights_attr**, **values_attr** y **params**, que se corresponden con los campos definidos en el fichero de definición del rol.

En segundo lugar, el campo **dynamic** contiene inicialmente un conjunto inicial de parámetros que, variarán conforme avance el proceso de negociación o requieran construirse en tiempo de ejecución, estos parámetros son:

- **oponent_model**: contiene el clasificador entrenado con conocimiento del oponente.
- **memory_proposal_offers**: contiene las ofertas emitidas por el agente.
- **memory_received_offers**: contiene las ofertas recibidas por el agente.

- **t**: paso actual de la negociación.
- **s**: nivel de aspiración actual del agente.
- **accepted_offers**: contiene las ofertas recibidas por el agente que han sido aceptadas (únicamente contendrá una oferta en caso de que el agente acepte una oferta del oponente).
- **revoked_offers**: contiene las ofertas recibidas por el agente que han sido rechazadas (tendrá **revoked_step**-1 ofertas en cada negociación).
- **oponent_agent**: contiene el nombre del oponente.

Este campo puede mantener más parámetros y actúa como un contenedor de variables definidas en las nuevas estrategias implementadas. Un ejemplo de esto se puede observar en la Figura 10, donde se define una nueva variable **lambda** en el campo **dynamic** que será modificada en cada ejecución de la función en el proceso de negociación.

```
@staticmethod
def exponential_decay_example(parameters):
    s = parameters["dynamic"]["s"]
    t = parameters["dynamic"]["t"]
    l = parameters["dynamic"]["lambda"] if "lambda" in
parameters["dynamic"] else 1
    parameters["dynamic"]["lambda"] = uniform(0, 1) * l
    return s*(e**(-l*t))
```

Figura 10. Uso del campo **dynamic** para almacenar nuevas variables.

Además, es conveniente notar que, al enviarse el diccionario **parameters,** de forma automática, a todos los nuevos métodos implementados, es posible pasar variables de un nuevo método a otro almacenándolas en el diccionario **dynamic** de **parameters**. Con respecto al uso de los parámetros estáticos, se muestra un ejemplo en la Figura 11.

```
@staticmethod
def receive_random_offers(offers, parameters):
    acceptance_type =
parameters["static"]["params"]["strategies"]["acceptance_type"]["func
"]
    utility_func =
parameters["static"]["params"]["strategies"]["utility_type"]["func"]
    offer = offers[randint(0, len(offers)-1)]
    parameters["dynamic"]["utility"] = utility_func(offer,
parameters)
    accept = acceptance_type(parameters)
    if accept:
        return offer
```

```
    else:
        return False
```

Figura 11. Uso del campo **static** para acceder a los campos del.

En este caso, se accede al campo **func** de las estrategias de aceptación y de utilidad, que contiene la función (método) Python de dichas estrategias. Sin embargo, esto es solo un ejemplo de acceso a unos parámetros determinados y es posible obtener cualquier parámetro de los listados en la enumeración anterior. Con todo ello, en los próximos apartados se hará uso, de las formas comentadas, del diccionario **parameters** para construir un nuevo agente.

Por otro lado, con respecto a los nuevos métodos implementados, tal como se ha podido notar en las dos figuras anteriores, es necesario especificarlos como métodos estáticos con @staticmethod debido a que las clases de las estrategias actúan como contenedores de nuevas estrategias.

## 3.2 Aceptación

Se ha optado por implementar primero el criterio de aceptación del nuevo agente. Para esto, abrimos el script **Acceptance.py**, donde encontramos la clase **Acceptance** que vamos a modificar. En ella debemos añadir un nuevo método estático con el nombre que se desee (el nombre se usará en el fichero **json** de definición del nuevo agente). La cabecera del método se muestra en la Figura 12.

```
@staticmethod
def name_acceptance_strategy(parameters):
    return [True|False]
```

Figura 12. Cabecera de los nuevos criterios de aceptación

Con ello, queremos que el agente de ejemplo siga un criterio de aceptación racional hasta alcanzar el último step de la negociación. Así, si el agente recibe una oferta en su último step, siempre la aceptará, sea cual sea. Esto podemos hacerlo comprobando si $\boldsymbol{t = revoke\_step - 1}$ para aceptar la última oferta.

Como se mostró en el apartado **Introducción al desarrollo**, podemos acceder a **t** en el campo **dynamic** de **parameters** y a **revoke_step** a partir del campo **static** debido a que se carga del fichero **json** de definición del agente. El código de la estrategia mencionada se muestra en la Figura 13.

```
@staticmethod
def rational_until_end(parameters):
    if
parameters["dynamic"]["t"]==parameters["static"]["params"]["general"]
["revoke_step"]-1:
        return True
```

```
    else:
        return parameters["dynamic"]["utility"] >=
parameters["dynamic"]["s"]
```

Figura 13. Implementación de un nuevo criterio de aceptación, **rational_until_end**.

## 3.3 Función de utilidad

La segunda estrategia a implementar es la función de utilidad, para calcular el valor de utilidad de cada oferta. En este caso, abrimos el script **Utility.py**, donde encontramos la clase **Utility** que vamos a modificar. En ella debemos añadir un nuevo método estático con el nombre que se desee (el nombre se usará en el fichero **json** de definición del nuevo agente). La cabecera del método se muestra en la Figura 14.

```
@staticmethod
def name_utility_function(offer, parameters):
    return offer_utility
```

Figura 14. Cabecera de las nuevas funciones de utilidad.

Se debe notar que, además del diccionario **parameters** se recibe otro parámetro adicional, **offer**. Este atributo contiene una determinada oferta, generalmente construida por la estrategia de generación de ofertas o recibida del oponente. Dicha oferta se representa mediante un diccionario de pares clave-valor, donde las claves son los atributos por los que se negocia y el valor es la cantidad o elección del atributo. Se puede ver un ejemplo de oferta en la Figura 15.

```
{
'color_naranjas': 2,
'precio_melones': 5300.0,
'precio_navajas': 2395.857783170088,
'color_melones': 0,
'metal_navaja': 4,
'precio_cobre': 602.88477558349,
'precio_naranjas': 1617.0,
'calidad_cobre': 1
}
```

Figura 15. Ejemplo de oferta.

En este caso, queremos implementar una función de utilidad lineal como la vista en clases de teoría. Para ello, únicamente necesitamos los campos **weights_attr**, **values_attr** y **attributes** almacenados a partir del campo **domain** en **dynamic[“static”]** que contienen la información de las preferencias del agente y de los atributos por los que se va a negociar. Con esto, simplemente tenemos que recorrer los atributos de la oferta para obtener su valor y ver qué peso le asigna el agente a dicho atributo y qué valor $V \in [0,1]$ le asigna al valor actual del atributo.

Así, la principal dificultad de la implementación radica en manejar correctamente el diccionario **values_attr** del campo **domain** en **dynamic[“static”]** para obtener el valor V en función de las condiciones para cada tipo de atributo (**integer**, **float** y **categorical**). Debido a la falta de tiempo para desarrollar el presente manual, se obvia la explicación detallada de este proceso, por este motivo, se recomienda al lector repasar la implementación proporcionada en el método **linear**, mostrado en la Figura 16, para observar las bases que permiten construir nuevas funciones de utilidad.

```
@staticmethod
def linear(offer, parameters):
    attributes = parameters["static"]["domain"]["attributes"]
    values_attr = parameters["static"]["domain"]["values_attr"]
    weights_attr = parameters["static"]["domain"]["weights_attr"]

    utility = 0
    for item in offer:
        value_item = offer[item]
        type_item = attributes[item]
        for condition in values_attr[type_item][item]["conditions"]:
            if type_item=="integer" or type_item=="float":
                left_bound =
values_attr[type_item][item]["conditions"][condition][0]
                right_bound =
values_attr[type_item][item]["conditions"][condition][1]
                if left_bound<=value_item<=right_bound:
                    value_item =
values_attr[type_item][item]["conditions"][condition][2]
                    break

            elif type_item=="categorical":
                if
value_item==values_attr[type_item][item]["conditions"][condition][0]:
                    value_item =
values_attr[type_item][item]["conditions"][condition][1]
                    break

            else: return 0
        utility += weights_attr[item] * value_item
    return utility
```

Figura 16. Implementación de la nueva función de utilidad, **linear**.

## 3.4 Concesión

La tercera estrategia a implementar es la estrategia de concesión, para determinar la forma en qué se reduce el nivel de aspiración conforme avanza el proceso de negociación. En este caso, abrimos el script **Concession.py**, donde encontramos la clase **Concession** que vamos a modificar. En ella debemos añadir un nuevo método estático con el nombre que se desee (el nombre se usará en el fichero **json** de definición del nuevo agente). La cabecera del método se muestra en la Figura 14.

```
@staticmethod
def name_concession_strategy(parameters):
    return new_s
```

Figura 14. Cabecera de las nuevas estrategias de concesión.

En este caso, se quiere seguir una estrategia *exponential decay* definida por la expresión $N(t) = N_0\, e^{-\lambda t}$ donde $N(t)$ es el nivel de aspiración en el instante t, $N_0$ es el nivel de aspiración en el instante $t-1$ y $\lambda$ es un parámetro que controla la caída de la función. Además, se quiere que $\lambda$ se modifique en cada iteración a partir de un valor aleatorio entre 0 y 1 y el valor de $\lambda$ en la iteración anterior.

Así, podemos acceder al valor de **t** en el campo **dynamic** de **parameters** y al valor inicial de $\lambda$ (**lambda**) a partir del campo **static**, sin embargo, ¿cómo podemos hacer que **lambda** se modifique en cada ejecución de la estrategia? La respuesta es, como ya se ha mencionado, utilizar el campo **dynamic** como un contenedor de variables y definir en él una nueva variable **lambda.**

El código de la estrategia mencionada se muestra en la Figura 15.

```
@staticmethod
def exponential_decay(parameters):
    s = parameters["dynamic"]["s"]
    t = parameters["dynamic"]["t"]
    l = None
    if "lambda" in parameters["dynamic"]:
        l = parameters["dynamic"]["lambda"]
    else:
        l = parameters["static"]["params"] \
                  ["strategies"]["concession_type"]\
                  ["params"]["lambda"]
    l = l * uniform(0, 1)
    parameters["dynamic"]["lambda"] = l
    return s*(e**(-l*t))
```

Figura 15. Implementación de una nueva estrategia de concesión, *exponential decay*.

En él, se puede observar cómo si el parámetro **lambda** no está almacenado en **dynamic** se considera el valor inicial especificado en el campo **params** de la estrategia de concesión para calcular la expresión $N(t) = N_0\, e^{-\lambda t}$. Finalmente, se almacena el valor del parámetro en **dynamic** para poder utilizarlo en siguientes ejecuciones.

## 3.5 Generación de ofertas

La siguiente estrategia a implementar es la de generación de ofertas, que se encarga de construir las ofertas a emitir en cada paso del proceso de negociación. En este caso, abrimos el script **GenerateOffers.py**, donde encontramos la clase **GenerateOffers** que vamos a modificar. En ella debemos añadir un nuevo método estático con el nombre que se desee (el nombre se usará

en el fichero **json** de definición del nuevo agente). La cabecera del método se muestra en la Figura 16.

```
@staticmethod
def name_generate_offer_strategy(space, parameters):
    return offer, utility_offer
```

Figura 16. Cabecera de las nuevas estrategias de generación de ofertas.

Llama la atención que el método reciba un parámetro adicional, **space**, este parámetro es enviado por el entorno a cualquier nueva estrategia de generación de ofertas implementada y contiene un diccionario que tiene como claves los atributos por los que se va a negociar y como valor una tupla.

Esta tupla, a su vez, contiene como primer elemento el tipo del atributo (**integer**, **float** o **categorical**) y, dependiendo de dicho tipo, la tupla tendrá dos o tres elementos. Si el tipo es **integer** o **float** el segundo y tercer elemento indicaran los valores mínimo y máximo que puede tomar el atributo, mientras que, si el tipo es **categorical**, contendrá una lista con las posibles elecciones que puede tomar el atributo. Un ejemplo del parámetro **space** se muestra en la Figura 17.

```
{

'precio_melones': ('integer', 200, 7500),
'metal_navaja': ('categorical', [0, 1, 2, 3, 4, 5, 6, 7]),
'color_melones': ('categorical', [0, 1, 2, 3]),
'precio_cobre': ('float', 100, 1500),
'precio_naranjas': ('integer', 500, 2500),
'color_naranjas': ('categorical', [0, 1, 2, 3, 4]),
'calidad_cobre': ('categorical', [0, 1, 2, 3]),
'precio_navajas': ('float', 50, 3200)
}
```

Figura 17. Ejemplo de parámetro **space** en un dominio de ejemplo.

Así, con el parámetro **space** el usuario tiene disponible los intervalos de valores válidos para cada atributo y con el parámetro **parameters** la información estática especificada en la definición del agente y dinámica que se genera y/o modifica en tiempo de ejecución. Con esto, ya es posible implementar nuestra estrategia de generación de ofertas. En este caso, se ha optado por implementar una estrategia que genera las ofertas de manera aleatoria, escogiendo, para cada atributo, un valor válido delimitado por los intervalos de dicho atributo.

Este proceso se debe repetir hasta que se garantice que la oferta generada cumple algún criterio para poder emitirla, por ejemplo, hasta que la utilidad de la oferta, **utility**, supere el nivel de aspiración, **s**, del agente en el paso de negociación en el que se emite la oferta. Es importante notar que, el usuario es el encargado de realizar dicha comprobación (esto proporciona una mayor flexibilidad para realizar comprobaciones más complejas), aunque, si con el tiempo se comprueba que resulta muy tedioso, no es difícil de gestionarla desde la clase **Agent** para evitar que el usuario sea el responsable de la comprobación.

Con todo ello, el código para implementar la estrategia **random_offer** de generación de ofertas se muestra en la Figura 18.

```
@staticmethod
def random_offer(space, parameters):
    max_range = parameters["dynamic"]["max_range"]
    s = parameters["dynamic"]["s"]
    utility_function =
parameters["static"]["params"]["strategies"]["utility_type"]["func"]
    t = parameters["dynamic"]["t"]
    act_s = 0
    while act_s<s or max_range<act_s:
        res = {}
        for key in space:
            if space[key][0]=="integer": res[key] =
randint(int(space[key][1]), int(space[key][2]))
            elif space[key][0]=="float": res[key] =
uniform(space[key][1], space[key][2])
            elif space[key][0]=="categorical": res[key] =
choice(space[key][1])
        act_s = utility_function(res, parameters)
    return res, act_s
```

Figura 18. Implementación de la nueva estrategia de generación de ofertas, **random_offer**.

En el código se observa cómo, primero se cargan las variables que se utilizarán en la estrategia (se recomienda hacerlo así para no tener que acceder a los diccionarios cada vez que se use alguna variable). Estas variables son:

- **max_range**: es una variable almacenada en **dynamic** que contiene el valor de utilidad máximo que se quiere que alcancen las ofertas generadas. Este parámetro es **s+window_offer**, ya comentado en apartados anteriores.

- **utility_function**: es una variable que contiene un puntero a la función de utilidad especificada por el usuario para el agente. Esta variable se emplea como función para calcular la utilidad de las ofertas generadas.

- **s**: variable que contiene el nivel de aspiración actual del agente en la negociación.

- **t**: variable que contiene el step actual de la negociación.

Una vez se obtienen las variables requeridas desde los diccionarios de parámetros, se generan ofertas hasta que se construya una que cumpla algún criterio, en este caso, superar el nivel de aspiración actual (bucle **while**). Para la generación de ofertas, dentro de dicho bucle **while**, simplemente se recorren los atributos del dominio, almacenados en el diccionario **space**, comprobando el tipo de dicho parámetro para asignarle aleatoriamente un valor escogido dentro del intervalo válido de valores para el parámetro e.g. si el atributo es **float** se escoge de forma uniforme un valor dentro del intervalo o si el atributo es **categorical** se escoge aleatoriamente una elección de la lista de elecciones.

Por ultimo, tal como se mostró en la cabecera del método (Figura 16), es necesario que la nueva estrategia devuelva la oferta generada (en el formato ya visto de la Figura 15, en el apartado

dedicado a la función de utilidad de este capítulo), que cumple un criterio determinado; y su valor de utilidad.

## 3.6 Recepción de ofertas

La siguiente estrategia a implementar es la de recepción de ofertas, que se encarga de procesar las ofertas recibidas en un paso determinado de la negociación. En este caso, abrimos el script **ReceiveOffers.py**, donde encontramos la clase **ReceiveOffers** que vamos a modificar. En ella debemos añadir un nuevo método estático con el nombre que se desee (el nombre se usará en el fichero **json** de definición del nuevo agente). La cabecera del método se muestra en la Figura 19.

```
@staticmethod
def name_receive_offers_strategy(offers, parameters):
    return [offer|False]
```

Figura 19. Cabecera de las nuevas estrategias de recepción de ofertas.

Este método recibirá, además del diccionario de parámetros **parameters**, el conjunto de ofertas recibidas por el agente (en caso de que se utilice la herramienta con **n_offers>1**, la talla **T** del conjunto será **>1**). En este caso, se quiere escoger aleatoriamente una de las ofertas recibidas y comprobar si se acepta. En caso de que dicha oferta sea aceptada, esta será devuelta por el método implementado, en otro caso, se retornará False. Con ello, el código de la estrategia se muestra en la Figura 20.

```
@staticmethod
def receive_random_offers(offers, parameters):
    acceptance_type =
parameters["static"]["params"]["strategies"]["acceptance_type"]["func
"]
    offer = offers[randint(0, len(offers)-1)]
    accept = acceptance_type(parameters)
    if accept:
        return offer
    else:
        return False
```

Figura 20. Implementación de la nueva estrategia de recepción de ofertas, **receive_random_offers**.

En la nueva estrategia, se obtiene la función de aceptación definida por el usuario, se escoge una oferta aleatoria del conjunto de ofertas y, finalmente, se comprueba si se acepta dicha oferta mediante la llamada a la función de aceptación.

## 3.7 Aprendizaje

En este apartado se propone un ejemplo de utilización de conocimiento del oponente en la estrategia de generación de ofertas, de forma que, de las ofertas generadas que cumplan algún criterio (en nuestro caso **utility≥s**), se devolverá aquella que, según el clasificador entrenado con conocimiento del oponente, pueda ser aceptada por el oponente. También, es necesario destacar que es posible utilizar dicho conocimiento en cualquier tipo de nueva estrategia implementada.

Para poder llevar a cabo dicha comprobación adicional y determinar si el oponente puede aceptar una oferta que al agente que la emite le interesa, debemos emplear el clasificador entrenado con conocimiento del oponente, almacenado en el campo **ml_model** de **dynamic** en **parameters**; y el campo **upper_bound_knowledge** de los parámetros del fichero de definición del agente para determinar a partir de qué iteración, del algoritmo de generación de ofertas, dejar de usar conocimiento del oponente.

En este ejemplo, se pretende que en la estrategia de generación de ofertas **random_offer** se utilice conocimiento del oponente de la forma mencionada. Con esto, tenemos que realizar una serie de modificaciones sobre el método ya implementado.

En primer lugar, se requiere cargar el modelo **ml_model** y los campos **use_knowledge** y **upper_bound_knowledge**, además, inicializamos una nueva variable **iters** para llevar la cuenta de cuántas ofertas se han clasificado con el modelo del oponente, para, finalmente, dejar de utilizer dicho modelo cuando se supere **upper_bound_knowledge**.

Posteriormente, dentro del bucle **while** para comprobar que **utility≥s**, se debe realizar una comprobación adicional para determinar si se cumple el criterio **utility≥s**. En caso de que se cumpla, si se está usando conocimiento y todavía no se ha superado **upper_bound_knowledge**, convertimos la oferta en un vector mediante la utilidad **knowledge_to_sample_test** y la clasificamos con el modelo del oponente. Si según el modelo del oponente la oferta sería aceptada por este, la oferta se acepta, en cualquier otro caso, se el método continuará generando nuevas ofertas.

Con todo ello, el código de la nueva estrategia de recepción de ofertas, que usa conocimiento del oponente, se muestra en la Figura 21.

```
@staticmethod
def random_offer(space, parameters):
    max_range = parameters["dynamic"]["max_range"]
    s = parameters["dynamic"]["s"]
    utility_function = parameters["static"]["params"] \
                                    ["strategies"]["utility_type"]["func"]
    use_knowledge = parameters["static"]["params"]\
                                ["general"]["use_knowledge"]
    upper_bound_knowledge = parameters["static"]["params"] \
                                        ["general"]["upper_bound_knowledge"]
    ml_model = parameters["dynamic"]["oponent_model"]
    t = parameters["dynamic"]["t"]
    act_s = 0
    iters = 0
    while act_s<s or max_range<act_s:
        res = {}
```

```
        for key in space:
            if space[key][0]=="integer":
                res[key] = randint(int(space[key][1]), int(space[key][2]))
            elif space[key][0]=="float":
                res[key] = uniform(space[key][1], space[key][2])
            elif space[key][0]=="categorical":
                res[key] = choice(space[key][1])
        act_s = utility_function(res, parameters)
        if s<=act_s and act_s<=max_range:
            if use_knowledge and iters<upper_bound_knowledge:
                x = Utils.convert_knowledge_to_sample_test(res, t)
                oponent_accepts = ml_model.predict([x])[0]
                if oponent_accepts==0: act_s = float("-inf")
                iters += 1
    return res, act_s
```

Figura 21. Implementación de la nueva estrategia de generación de ofertas, **generate_random_offers** con conocimiento del oponente.

## 3.8 Definición del agente

La última etapa de la construcción de un agente consiste en definir el nuevo agente mediante su fichero de definición **json**. Para esto, únicamente tenemos que especificar la información del nuevo agente, su rol y las estrategias utilizadas con sus parámetros.

En nuestro caso, podemos definir la información del nuevo agente, por ejemplo, tal como se muestra en la Figura 22.

```
"agent_info" : {
  "name": "Agent007",
  "gender": "male",
  "country": "spain",
  "birthday": "16/08/94",
  "reputation": 0.8,
  "description" : "negotiating in product sale domain.
Other domains of interest: car sale",
  "domain" : "Product sale"
}
```

Figura 22. Información del nuevo agente.

Tras especificar la información del agente, es necesario asignarle un **rol**. Así, el nuevo agente **Agent007** tomará el rol de vendedor, cuyas preferencias se definen en el fichero **Rols/Seller.json**. La forma de definir el **rol** del agente ya se vió en el apartado 2.1.2 y se muestra en la Figura 23.

```
"rol" : "Rols/Seller.json"
```

Figura 23. Rol del nuevo agente.

Por último, tenemos que especificar las estrategias que empleará el agente en el proceso de negociación, los parámetros iniciales de dichas estrategias y los parámetros generales del agente.

Con ello, con respecto a los parámetros generales, queremos que su nivel de aspiración, **s**, inicial sea de 0.75, su nivel de aspiración mínimo, **ur**, sea de 0.45, que finalice el proceso de negociación, **revoke_step**, en el paso 200, que use conocimiento del oponente (**use_knowledge** a True), con una ventana de generación de ofertas **s+0.1** (**window_offer** a 0.1) y que deje de generar ofertas teniendo en cuenta conocimiento del oponente desde que se hayan generado 250 ofertas rechazadas.

Por otro lado, en lo relativo a las estrategias utilizadas haremos uso de las implementadas en los apartados anteriores de este mismo capítulo. Además, queremos que el valor inicial de **lambda** para la estrategia de concesión *exponential decay* sea de 0.5 y que el clasificador utilizado sea un perceptron multicapa de dos capas con 128 y 4 neuronas, entrenado con el solver **lbfgs**, con un factor de aprendizaje de 1e-5 y con una semilla prefijada a 1 para el generador de números aleatorios.

De esta forma, el campo **params** del nuevo agente se define en la Figura 24.

```
"params" : {

  "general" : {
    "ur": 0.45,
    "revoke_step": 200,
    "s": 0.75,
    "window_offer": 0.1,
    "use_knowledge" : true,
    "upper_bound_knowledge": 5000
  },

  "strategies": {
      "utility_type": {
        "module" : "Utility",
        "func" : "linear",
        "params" : {}
      },

      "concession_type": {
        "module" : "Concession",
        "func" : "exponential_decay",
        "params" : {
          "lambda": 0.5,
        }
      },

      "acceptance_type": {
        "module" : "Acceptance",
        "func": "rational_until_end",
        "params" : {}
      },

      "generate_offer_type": {
        "module" : "GenerateOffers",
        "func" : "random_offer",
        "params" : {
          "max_eval" : 15
        }
      },
```

```
        "receive_offer_type": {
          "module" : "ReceiveOffers",
          "func" : "receive_random_offers",
          "params" : {}
        },

        "ml_model": {
          "module" : "Classifiers",
          "func": "mlp_nn_classifier",
          "params": {
            "solver": "lbfgs",
            "alpha": 1e-5,
            "hidden_layer_sizes": [128, 4],
            "random_state": 1
          }
        }
    }
}
```

Figura 24. Definición del nuevo agente en **json**.

Juntando los tres campos mencionados en el fichero de definición, ya podemos lanzar el agente a negociar en el entorno contra otro agente que tenga otro (o el mismo) rol de nuestro agente. Así, para probar el agente desarrollado **Agent007** se ha llevado a cabo una negociación entre este y otro agente (**Comprador1**) proporcionado con el entorno que toma el rol de comprador. En las figuras 25 y 26 se muestran algunos resultados devueltos por la herramienta.

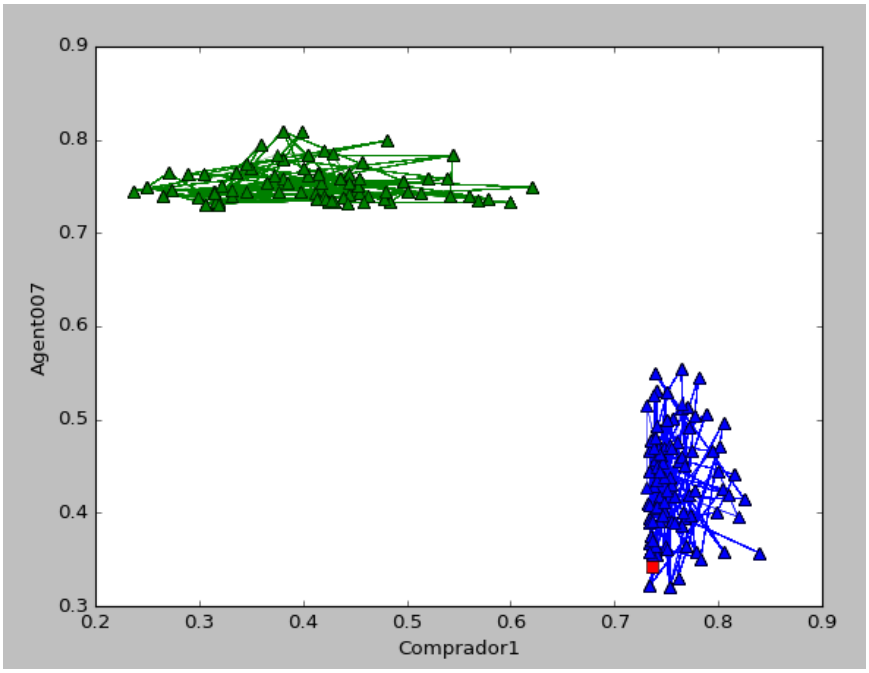


Figura 25. Resultado de la negociación entre **Agent007** y **Comprador1**.

```
*************
* Bilateral *
*************


Step  200 ...

* Deal accepted by dealer Agent007 at step 200 *

  · Benefits dealer Agent007: 0.343000
  · Benefits dealer Comprador1: 0.736000
  · Concession dealer Agent007: 0.730529
  · Concession dealer Comprador1: 0.730172
  · Offer: {
     'precio_cobre': 863.383237800694,
     'color_naranjas': 4,
     'precio_melones': 2486.0,
     'calidad_cobre': 1,
     'precio_navajas': 1707.6759280002434,
     'color_melones': 2,
     'precio_naranjas': 561.0,
     'metal_navaja': 4
   }
```

Figura 26. Salida de la negociación entre **Agent007** y **Comprador1**.

Se puede observar como la negociación avanza hasta alcanzar el ultimo step donde nuestro agente **Agent007** acepta la oferta debido a la estrategia de aceptación **rational_until_end**.

Por ultimo, en el siguiente capítulo se muestra una guía de utilización de la herramienta con otros agentes y otro dominio, sin embargo, se recomienda al lector repetir los pasos de dicho capítulo, utilizando el agente desarrollado en este.

# 4. Utilización de la herramienta

## 4.1. Instalación

En primer lugar, es necesario instalar las dependencias de la aplicación para poder ejecutarla. Como ya se comentó en el apartado del capítulo uno dedicado a la instalación, podemos hacer uso del script **setup.py** (**python setup.py install**) proporcionado por la herramienta, la salida de la ejecución debe ser como se muestra en la figura 27.

```
~$ python setup.py install

Installing pip ...
Installing numpy ...
Installing scipy ...
Installing matplotlib ...
Installing scikit-learn ...
Installing deap ...
Installing hyperopt ...
All installed.
```

Figura 27. Instalación de las dependencias.

Si la ejecución finaliza con algún error, como ya se comentó, es recomendable instalar la plataforma **Anaconda** e instalar los paquetes necesarios desde ella. Sin embargo, el objetivo de la guía rápida es marcar los pasos necesarios para llegar a ejecutar la herramienta con los ejemplos proporcionados, por lo que no se ha detallado una instalación de esta manera y se asume que, en este punto, el lector ya dispone de todas las dependencias requeridas por el entorno.

## 4.2. Primeras negociaciones

Una vez se ha completado la instalación de las dependencias, ya es posible ejecutar en el entorno una negociación de prueba entre los agentes de ejemplo proporcionados. Así, como primera prueba pondremos a los agentes **Tyrion** (**Bots/TyrionPrimeraPrueba.json**) y **John Snow** (**Bots/JohnSnowPrimeraPrueba.json**) a negociar en el dominio de venta de coches. Para más información sobre la definición de los agentes y cómo diseñar nuevas estrategias de negociación para estos, consultar los capítulos dos y tres.

Con respecto a los ficheros de definición de agentes, como es la primera negociación entre ellos, los agentes no podrán utilizar conocimiento del oponente, por lo que el valor del campo **use_knowledge** del fichero **json** de los agentes debe ser **false**.

Por otro lado, en la configuración (**config.json**) empleada en esta prueba se han especificado los agentes que participarán en la negociación, se ha indicado que se quiere que los agentes guarden información del oponente para utilizarla en pruebas siguientes (**save_corpus=True**), además, se emitirán tres ofertas en cada paso de negociación (**n_offers=3**) y se mostrará, al final de la negociación, una representación gráfica estática del proceso (**graphic_process=True** y **graphic_interactive=False**) y el *log* con las ofertas emitidas en cada paso. Así, el fichero **config.json** queda de la siguiente manera (Figura 28):

```
{

  "project_info" : {
    "version" : "0.5",
    "author" : "JAGB"
  },

  "agents" : {
    "Tyrion" : "./Bots/TyrionPrimeraPrueba.json",
    "JohnSnow" : "./Bots/JohnSnowPrimeraPrueba.json"

  },

  "knowledge" : {
      "corpus_path":"./",
      "save_corpus" : true
  },

  "params" : {
    "n_offers" : 1,
    "show_offers" : true,
    "first_random" : false,
    "graphic_process" : true,
    "graphic_interactive" : false
  }

}
```

Figura 28. Fichero **config.json** empleado en la primera prueba

Con esto, basta con ejecutar **Main.py** (**python Main.py config.json**) indicando el fichero de configuración a utilizar (en nuestro caso **config.json**) para iniciar un proceso de negociación bilateral entre los agentes mencionados. Tras ejecutar la aplicación obtenemos la salida por consola mostrada en la Figura 29 y la representación gráfica de la Figura 30.

```
~$ python Main.py


*************
* Bilateral *
*************

Step  52 ...

* Deal accepted by dealer JohnSnowPrimeraPrueba at step
52 *

 · Benefits dealer JohnSnowPrimeraPrueba: 0.760000
 · Benefits dealer TyrionPrimeraPrueba: 0.700000
 · Concession dealer JohnSnowPrimeraPrueba: 0.500000
 · Concession dealer TyrionPrimeraPrueba: 0.581675
 · Offer: {'color': 2, 'price': 1528.0, 'abs': 1}

* Offers *

- Step 0 from TyrionPrimeraPrueba :

         {'color': 2, 'price': 164.0, 'abs': 1}
         {'color': 2, 'price': 135.0, 'abs': 1}
         {'color': 2, 'price': 699.0, 'abs': 1}

- Step 1 from JohnSnowPrimeraPrueba :

         {'color': 2, 'price': 2550.0, 'abs': 1}
         {'color': 2, 'price': 2694.0, 'abs': 1}
         {'color': 1, 'price': 3356.0, 'abs': 1}

                        ...

- Step 52 from TyrionPrimeraPrueba :

         {'color': 2, 'price': 1116.0, 'abs': 1}
         {'color': 2, 'price': 1201.0, 'abs': 1}
         {'color': 2, 'price': 1528.0, 'abs': 1}
```

Figura 29. Salida de la primera prueba.

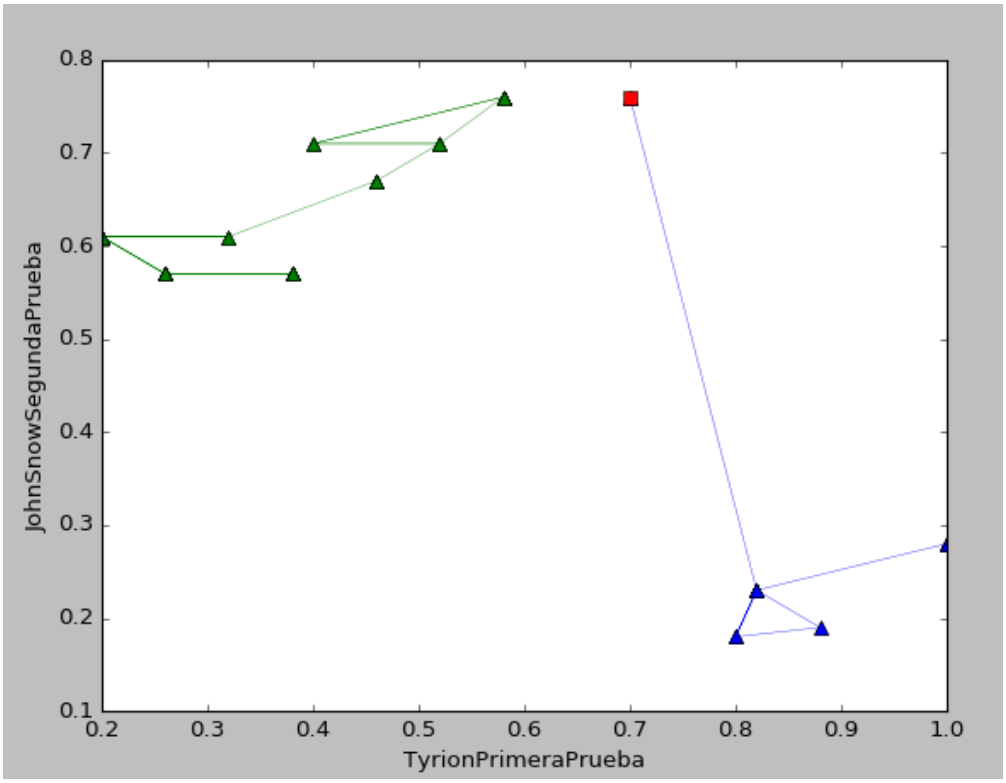


Figura 30. Representación gráfica del proceso de negociación en la primera prueba.

Como se puede observar, se imprime por pantalla la etapa en la que se ha alcanzado el fin de la negociación indicando, si se alcanza un acuerdo, para cada agente, su nivel de aspiración y el beneficio que le proporciona la oferta aceptada. También se imprime la oferta aceptada, que debe coincidir con alguna de las ofertas emitidas en el último paso de negociación.

En este ejemplo no se ha utilizado conocimiento del oponente porque los agentes no disponían de información sobre los oponentes, sin embargo, al haber especificado el parámetro **save_corpus** en la ejecución anterior, se han generado los ficheros **TyrionPrimeraPrueba.dump** y **JohnSnowPrimeraPrueba.dump** que contienen conocimiento sobre el agente que da nombre al fichero (para más información sobre estos ficheros consultar el apartado dedicado al **Aprendizaje** en el capítulo tres). Por este motivo, se muestra a continuación un proceso de negociación donde los dos agentes usan conocimiento del adversario para generar ofertas.

Para ello, podemos mantener el parámetro de **config.json**, **save_corpus** a **True** para continuar almacenando conocimiento del oponente (se añade en binario al fichero) y basta con cambiar el parámetro **use_knowledge** a **True**, en los ficheros de definición **json** de los agentes, para que empiecen a utilizar conocimiento del oponente. Es conveniente notar, que por defecto, solo algunas de las estrategias de generación de ofertas ya implementadas utilizan conocimiento del oponente (**random_offer** y **genetic_offer**), aunque sería simple implementarlo también con la estrategia **hyperopt** o en nuevas estrategias definidas por el usuario.

Por este motivo, para esta prueba, cambiamos, en los ficheros **json** de definición de agentes, el método de generación de ofertas (**generate_offer_type**) de **JohnSnowPrimeraPrueba** y **TyrionPrimeraPrueba** a **random_offer**. Además, sin contar con los parámetros modificados, **use_knowledge** y **offer_type**, se mantiene la configuración de la aplicación y las definiciones de

agentes empleadas en la prueba anterior, por lo que se continúan enviando tres ofertas en cada paso de negociación, se imprime el *log* de las ofertas emitidas y se representa gráficamente el proceso de negociación. Así, la salida por consola se muestra en la Figura 31 y la representación gráfica en la Figura 32.

```
*************
* Bilateral *
*************
Step 19 ...

* Deal accepted by dealer JohnSnowPrimeraPrueba at step
19 *

 · Benefits dealer JohnSnowPrimeraPrueba: 0.760000
 · Benefits dealer TyrionPrimeraPrueba: 0.700000
 · Concession dealer JohnSnowPrimeraPrueba: 0.532329
 · Concession dealer TyrionPrimeraPrueba: 0.658842
 · Offer: {'abs': 1, 'color': 2, 'price': 1363}

* Offers *

- Step 0 from JohnSnowPrimeraPrueba :

        {'abs': 1, 'color': 1, 'price': 4373}
        {'abs': 1, 'color': 1, 'price': 3351}
        {'abs': 1, 'color': 1, 'price': 2482}

- Step 1 from TyrionPrimeraPrueba :

        {'abs': 1, 'color': 2, 'price': 867}
        {'abs': 1, 'color': 2, 'price': 775}
        {'abs': 1, 'color': 2, 'price': 380}

                            ...

- Step 19 from TyrionPrimeraPrueba :

        {'abs': 1, 'color': 2, 'price': 1155}
        {'abs': 0, 'color': 0, 'price': 862}
        {'abs': 1, 'color': 2, 'price': 1363}
```

Figura 31. Salida de la prueba con aprendizaje.

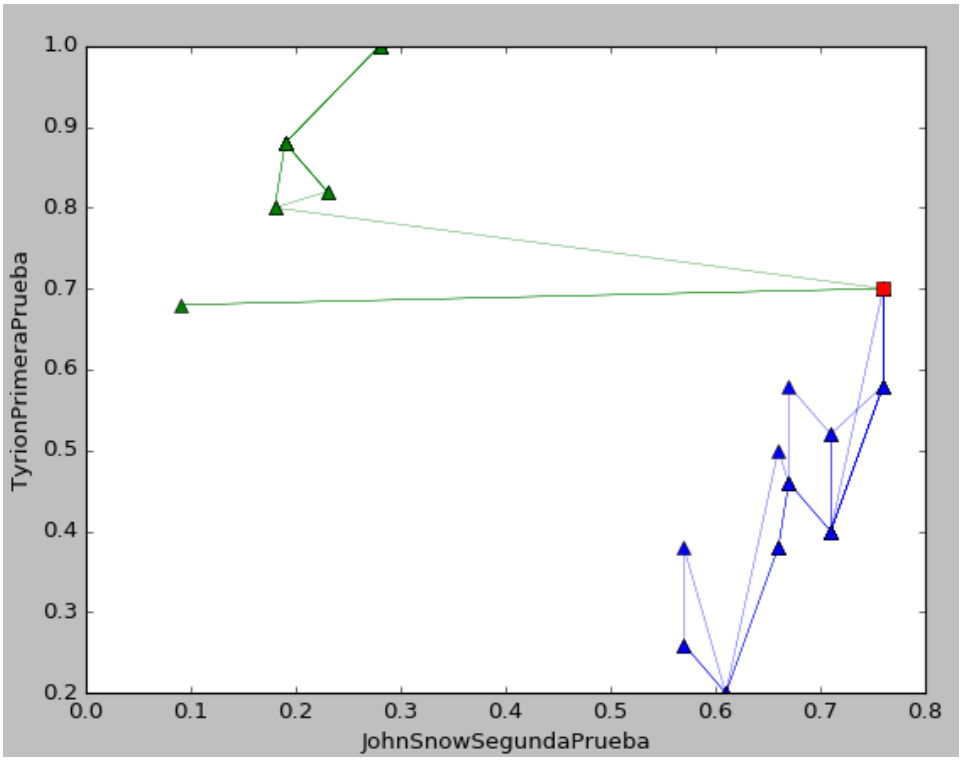


Figura 32. Representación gráfica del proceso de negociación en la prueba con aprendizaje.

En este caso, se observa cómo, si ambos agentes emplean conocimiento del oponente se alcanza la misma solución que si no lo emplean (prueba anterior), pero en un menor número de pasos de negociación. Esto ocurre así porque el dominio es muy simple y, posiblemente, la solución obtenida sea la mejor, por lo que no se puede mejorar, pero sí reducir el número de pasos para alcanzarla. Sin embargo, si el dominio es más complejo, es posible que utilizando aprendizaje se alcancen mejores soluciones.

## 4.3. Torneo

En el apartado anterior se ha llevado a cabo una negociación bilateral entre dos agentes, **TyrionPrimeraPrueba** y **JohnSnowPrimeraPrueba**, sin embargo, la herramienta también permite realizar torneos de negociaciones bilaterales entre todo par de agentes. Para ejecutar el entorno de esta manera, basta con indicar la ruta de más de dos agentes en el campo **agents** del fichero **config.json** que regular el comportamiento de la herramienta.

En este caso, indicamos que queremos realizar un torneo con tres agentes, **TyrionPrimeraPrueba**, **JohnSnowPrimeraPrueba** y **DanerisPrimeraPrueba** (con las mismas preferencias que **TyrionPrimeraPrueba**). Para esto, modificamos el campo **agents** para que quede de la forma mostrada en la Figura 33.

```
                                ...
"agents" : {
    "Tyrion" : "./Bots/TyrionPrimeraPrueba.json",
    "JohnSnow" : "./Bots/JohnSnowPrimeraPrueba.json",
    "Daneris" : "./Bots/DanerisPrimeraPrueba.json"

},

                                ...
```

Figura 33. Campo **agents** del fichero de configuración **config.json**

Además, en esta prueba solo se emitirá una oferta en cada paso de negociación (**n_offers=1** en **config.json**). Tras esto, se ejecuta la herramienta con **python Main.py config.json** y se observa cómo se realiza una negociación bilateral entre cada par de agentes. Por tanto, si hay **n** pares de agentes, se irán mostrando uno a uno los resultados de las **n** negociaciones, de la misma forma que en las primeras pruebas del apartado anterior. Además, si se le ha especificado a la herramienta que genere una representación gráfica, esta se mostrará al finalizar cada una de las **n** negociaciones. Con ello, las 3 negociaciones que se dan en el torneo de ejemplo se muestran en las siguientes figuras.

```
**************
* Tournament *
**************


Round 0 : DanerisPrimeraPrueba - JohnSnowPrimeraPrueba

Step  1 ...

* Deal accepted by dealer Daneris at step 1 *

  · Benefits dealer DanerisPrimeraPrueba: 0.700000
  · Benefits dealer JohnSnowPrimeraPrueba: 0.760000
  · Concession dealer DanerisPrimeraPrueba: 0.600000
  · Concession dealer JohnSnowPrimeraPrueba: 0.700000
  · Offer: {'abs': 1, 'price': 1104.0, 'color': 2}


* Offers *

- Step 0 from DanerisPrimeraPrueba :

          {'abs': 0, 'price': 591, 'color': 1}
```

```
- Step 1 from JohnSnowPrimeraPrueba :

          {'abs': 1, 'price': 1104.0, 'color': 2}


Round 1 : DanerisPrimeraPrueba - TyrionPrimeraPrueba

Step  1 ...


* Deal accepted by dealer DanerisPrimeraPrueba at step 1
*

  · Benefits dealer DanerisPrimeraPrueba: 1.000000
  · Benefits dealer TyrionPrimeraPrueba: 1.000000
  · Concession dealer DanerisPrimeraPrueba: 0.600000
  · Concession dealer TyrionPrimeraPrueba: 0.900000
  · Offer: {'abs': 1, 'price': 272.0, 'color': 2}


* Offers *

- Step 0 from DanerisPrimeraPrueba :

          {'abs': 0, 'price': 461, 'color': 1}

- Step 1 from TyrionPrimeraPrueba :

          {'abs': 1, 'price': 272.0, 'color': 2}


Round 2 : JohnSnowPrimeraPrueba - TyrionPrimeraPrueba

Step  13 ...

* Deal accepted by dealer JohnSnowPrimeraPrueba at step
13 *

  · Benefits dealer JohnSnowPrimeraPrueba: 0.760000
  · Benefits dealer TyrionPrimeraPrueba: 0.700000
  · Concession dealer JohnSnowPrimeraPrueba: 0.594962
  · Concession dealer TyrionPrimeraPrueba: 0.581839
  · Offer: {'abs': 1, 'price': 1637.0, 'color': 2}


* Offers *
```

```
- Step 0 from JohnSnowPrimeraPrueba :

        {'abs': 1, 'price': 1326.0, 'color': 2}

- Step 1 from TyrionPrimeraPrueba :

        {'abs': 1, 'price': 657.0, 'color': 2}

                        ...

- Step 13 from TyrionPrimeraPrueba :

        {'abs': 1, 'price': 1637.0, 'color': 2}

                        ...
```

Figura 34. Salida de la prueba con aprendizaje.

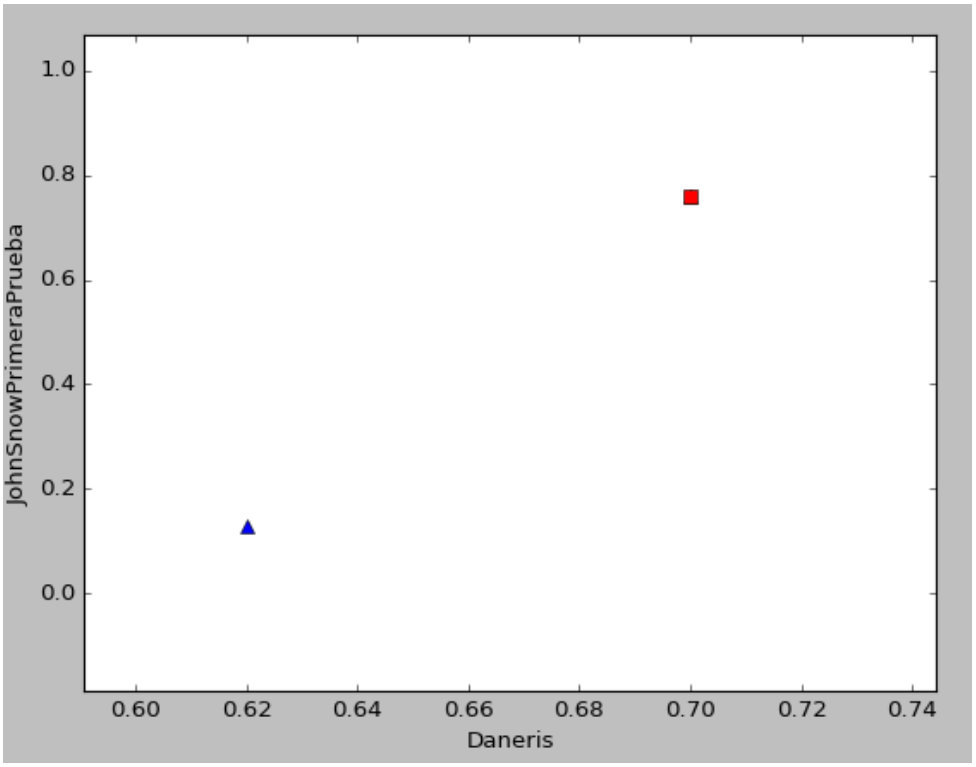


Figura 35. Representación gráfica de la negociación entre **JohnSnow** y **Daneris**.

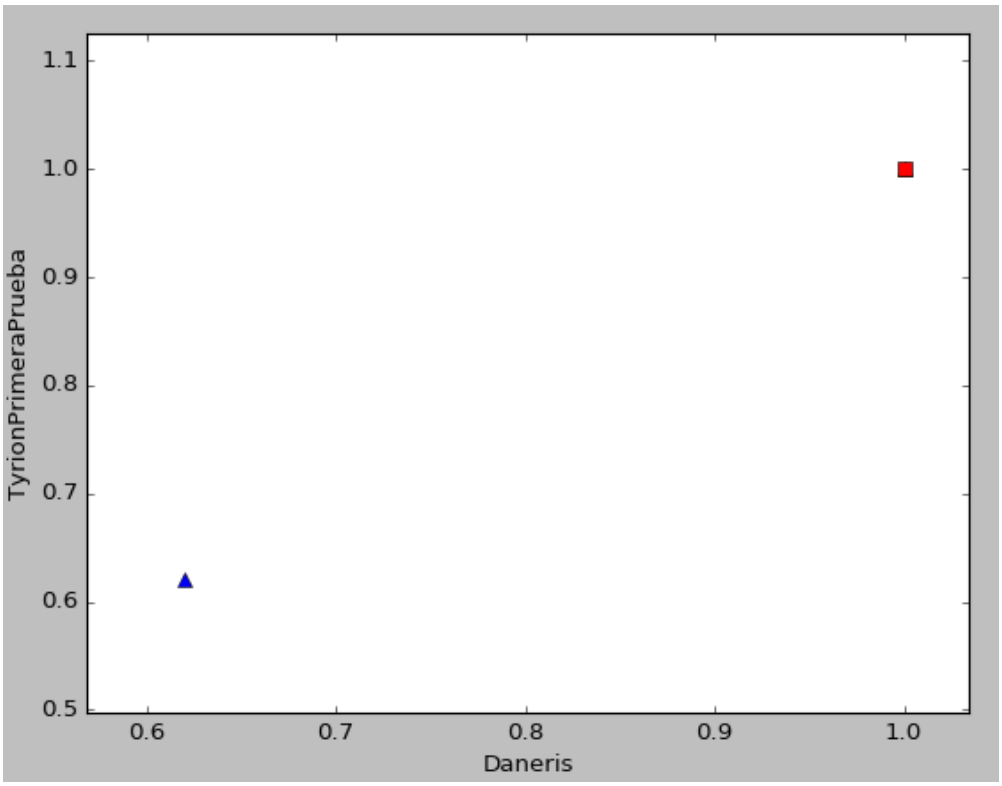


Figura 36. Representación gráfica de la negociación entre **Tyrion** y **Daneris**.

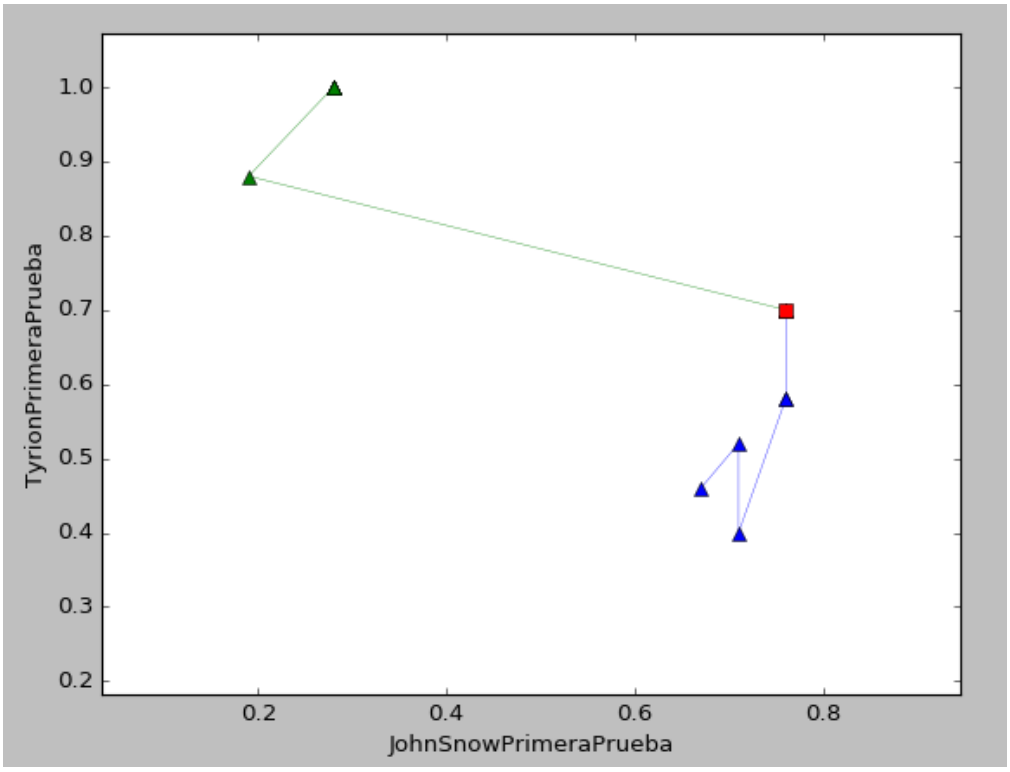


Figura 37. Representación gráfica de la negociación entre **Tyrion** y **JohnSnow**.

Además de estas salidas, asociadas a negociaciones bilaterales entre dos agentes, se imprimen por consola varias medidas para evaluar la bondad de los agentes en global, durante todas las negociaciones del torneo en las que ha participado. En el entorno proporcionado, las únicas métricas son los estadísticos que proporciona **numpy** ya comentados en el primer apartado del primer capítulo, pero es posible añadir todos los que se requiera en el script **Statistics.py**. Siguiendo con nuestro ejemplo del torneo entre tres agentes, los resultados de las distintas métricas mostradas por pantalla son:

```
*** Tournament statistics ***

JohnSnowPrimeraPrueba :

        · mean : 0.76
        · min : 0.76
        · max : 0.76
        · std : 0.0
        · var : 0.0
        · first percentile : 0.76
        · second percentile : 0.76
        · third percentile : 0.76

DanerisPrimeraPrueba :

        · mean : 0.85
        · min : 0.7
        · max : 1.0
        · std : 0.15
        · var : 0.0225
        · first percentile : 0.703
        · second percentile : 0.706
        · third percentile : 0.709

TyrionPrimeraPrueba :

        · mean : 0.85
        · min : 0.7
        · max : 1.0
        · std : 0.15
        · var : 0.0225
        · first percentile : 0.703
        · second percentile : 0.706
        · third percentile : 0.709
```

Figura 38. Salida de la prueba con aprendizaje.

Donde se observa que los agentes **DanerisPrimeraPrueba** y **TyrionPrimeraPrueba** se comportan mejor en el torneo que el agente **JohnSnowPrimeraPrueba**, principalmente debido a que los dos primeros agentes tienen las mismas preferencias y es fácil que alcancen acuerdos muy beneficiosos para ambos (valor de utilidad 1 para los dos, en este caso).

Esta es la última experimentación comentada, a pesar de que se han realizado muchas más, debido a la falta de tiempo para continuar el trabajo. Sin embargo, es conveniente destacar que todo lo que es posible usar en una negociación bilateral simple, también se puede emplear en torneos p.e. conocimiento sobre el oponente.

# 5. Bibliografía

[1] J. Bergstra, Rémi Bardenete, Yoshua Bengio, Balázs Kégl. Algorithms for hyper-parameter optimization. NIPS proceedings, 2011.

[2] Jones E, Oliphant E, Peterson P, *et al.* **SciPy: Open Source Scientific Tools for Python**, 2001-, http://www.scipy.org/ [Online; accessed 2016-12-30]

[3] Scikit-learn: Machine Learning in Python, Pedregosa *et al.*, JMLR 12, pp. 2825-2830, 2011.

[4] Travis E. Oliphant (2006) Guide to NumPy, Trelgol Publishing, USA

[5] *"Anaconda End User License Agreement". continuum.io. Continuum Analytics. Retrieved May 30, 2016*