Trabajo de asignatura
Sistemas Multi Agentes

Master en Inteligencia Artificial,Reconocimiento de Formas y Imagen Digital

Universidad Politècnica de Valencia

a.a. 2018 /2019

"Programación de un entorno de negociación automática bilateral en Python (plataforma Spade)"

Francesco Garavaglia

Joakim Olsen

Index

SPADE

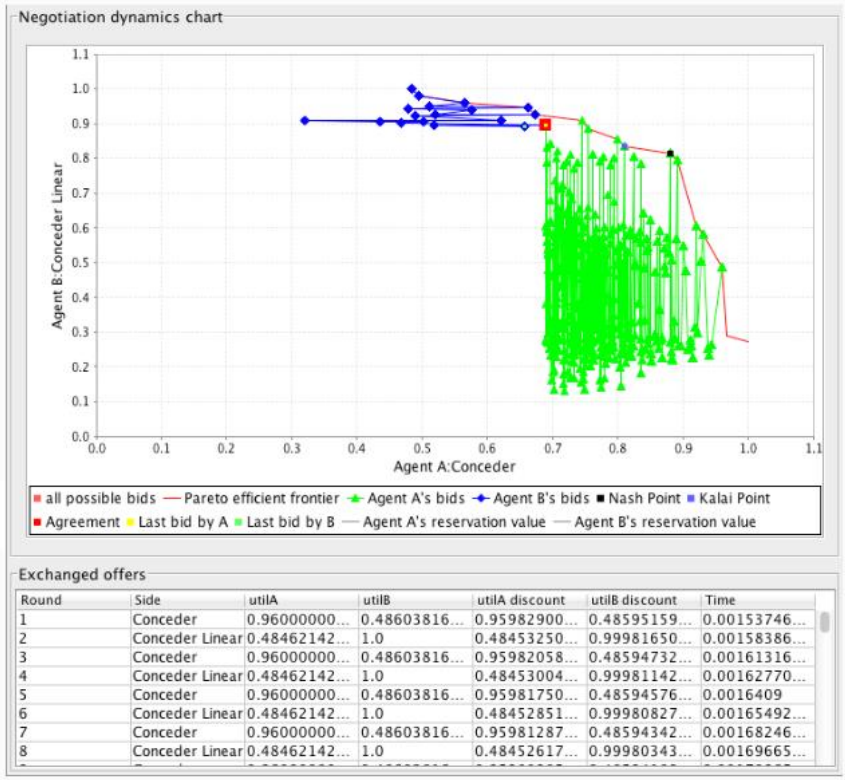
Spade (Smart Python Agent Development Environment) es una plataforma para sistemas multi agentes desarrollada en python que se basa en mensajes instantàneos (XMPP)[1]

Spade permite el Desarrollo de agentes que pueden chatear con otros agentes y seres humanos.

Para el uso de SPADE se necesita el uso de un server XMPP. Dado que hemos ejecutado el entorno en Windows, el server elejido fue Openfire.[2]

Genius

Genius (General Environment for Negotiation with Intelligent multi-purpose Usage Simulation) es un entorno de Negotiation que implementa una architectura abierta para la negotiation entre agentes.[3]



Round	Side	utilA	utilB	utilA discount	utilB discount	Time
1	Conceder	0.96000000...	0.48603816...	0.95982900...	0.48595159...	0.00153746...
2	Conceder Linear	0.48462142...	1.0	0.48453250...	0.99981650...	0.00158386...
3	Conceder	0.96000000...	0.48603816...	0.95982058...	0.48594732...	0.00161316...
4	Conceder Linear	0.48462142...	1.0	0.48453004...	0.99981142...	0.00162770...
5	Conceder	0.96000000...	0.48603816...	0.95981750...	0.48594576...	0.0016409
6	Conceder Linear	0.48462142...	1.0	0.48452851...	0.99980827...	0.00165492...
7	Conceder	0.96000000...	0.48603816...	0.95981287...	0.48594342...	0.00168246...
8	Conceder Linear	0.48462142...	1.0	0.48452617...	0.99980343...	0.00169665...

Pantalla de ejecuciòn de una negotiaciòn bilateral automatica entre dos agentes en Genius

Dominio de desarrollo

En este trabajo hemos desarrolado un entorno para la negociaciòn bilateral automatica entre agentes en Python con el uso de la plataforma SPADE.

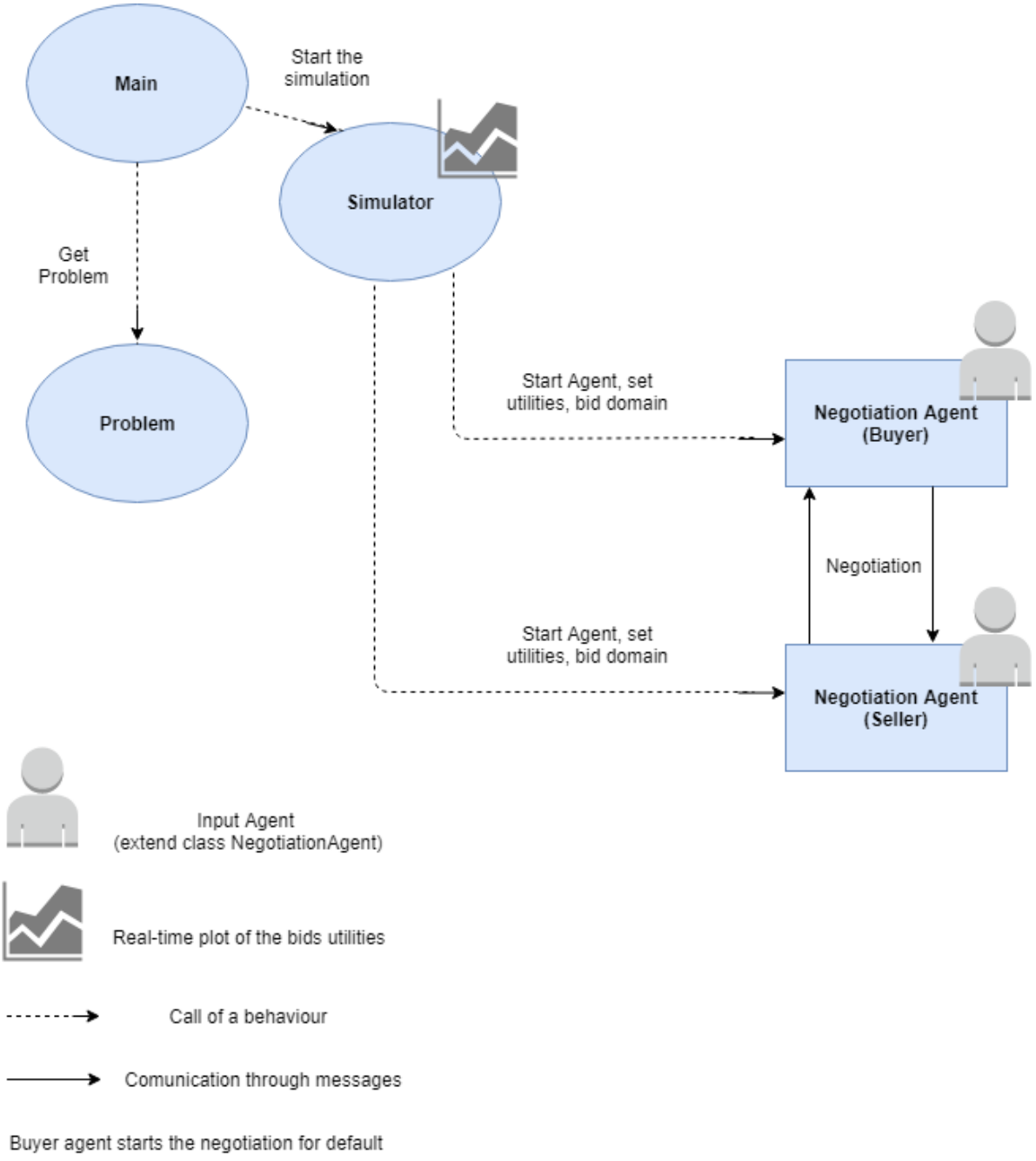
[1] http://spade-mas.readthedocs.io/

[2] https://www.igniterealtime.org/projects/openfire/

[3] http://ii.tudelft.nl/genius/

El entorno desarrolado permite el mismo funcionamento de negociaciòn bilateral entre dos agentes usado en Genius. No creo que podemos decir este. Genius tiene un montón de cosas, y nosotros solo tenemos “the bare necessities” para hacer negociacion. Quizas podemos decir que nuestro objetivo era obtener algo parecido que Genius, aunque mas simple?

Esquema de funcionamento de el entorno desarrollado:



Call of a behaviour – Should maybe be call of a function/class instance? The behaviours are started inside the agent on setup, so the simulator only does the instanciation of the agent, not calls of behaviours.

Carpeta a descargar:

- problem.py: define los parameters para las ofertas y las funciones de utilitad
- main.py : inicià el entorno (SPADE) para la negociaciòn con los parametròs definidos en el problema como input
- simulator.py: crear los agentes para inicià la negotiation (entre los agentes) y hace un plot real-time de las ofertas enviadas durante la negociaciòn
- negotiationAgent.py: define la clase NegotiationAgent con los behaviours para el agente de negociaciòn, las funciones necesario para hacer el negociacion y define el protocol de intercambio entre los agentes.
- concederAgent.py extiende negotiationAgent con la definiciòn de las estrategias para el agente
- boulwardAgent.py : extiende negotiationAgent con la definiciòn de las estrategias para el agente

Protocolo de intercambio

Los agentes de una negociaciòn son definidos como extension de la clase NegotiationAgent asì definida:

StartBidBehaviour:

- este behaviour es añadido solo al agente “buyer” para empezar la negociaciòn.
- ejecuta ProposeOffer() para seleccionar una oferta para enviar al agente de rol “seller”
- envia un mensaje con la oferta propuesta al agente “seller”
- es definido como OneShotBehaviour

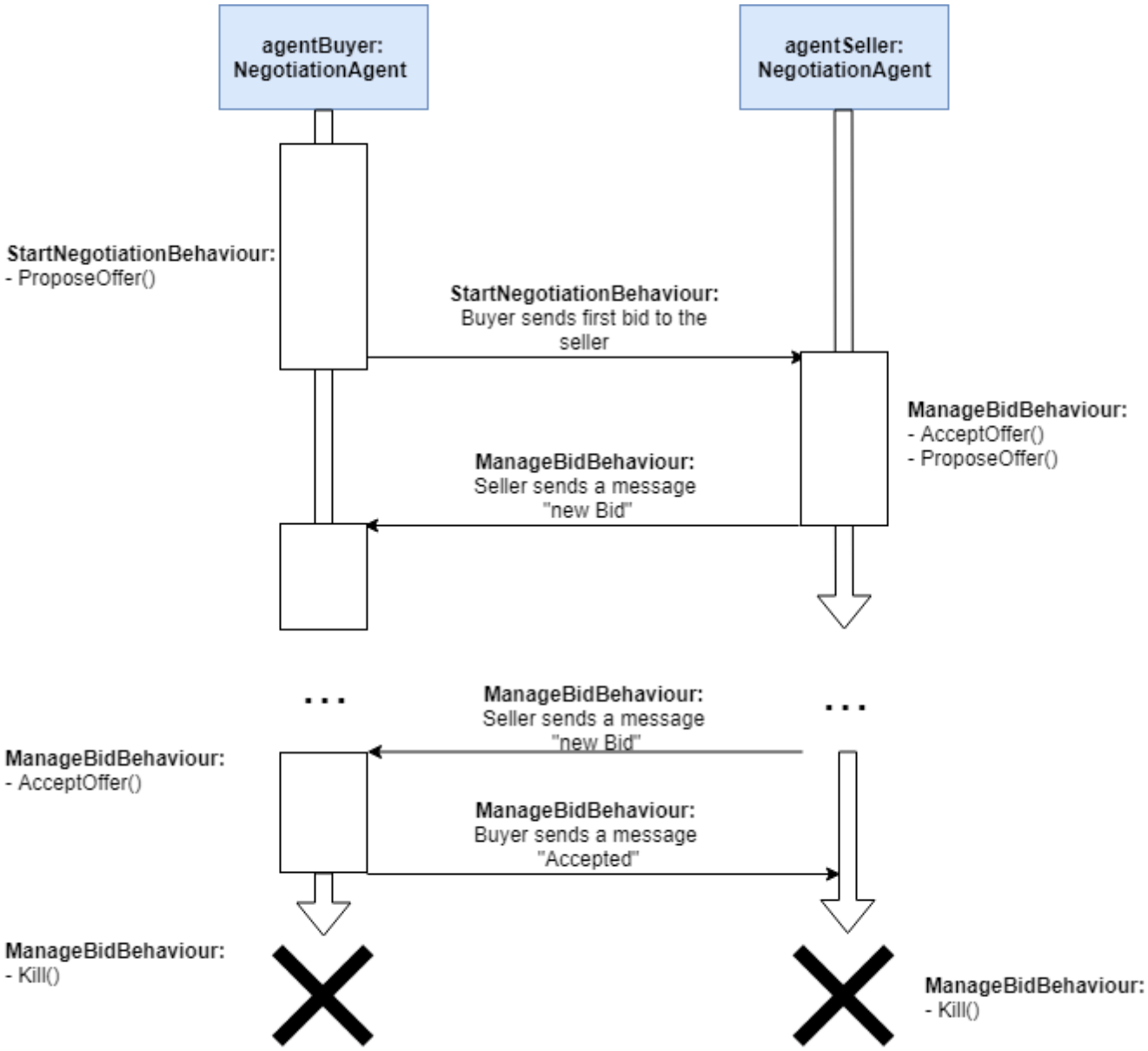
ManageBidBehaviour:

- este behaviour es añadido a los agentes “buyer” y “seller” para (recibir gestionar,?) recibir y enviar ofertas
- ejecuta ProposeOffer() y AcceptOffer()
- funcionamento:
  - Agente a espera un mensaje desde agente B
  - A lee el mensaje de B
  - If(B ha aceptado oferta enviada)
    - Acuerdo encontrado
    - Kill A (termina la negociaciòn)
  - Else if (AcceptOffer())
    - Envia un mensaje “True” para avisar B que hay un acuerdo
    - Kill A (termina la negociaciòn)
  - Else
    - ProposeOffer()
    - Envia un mensaje a B con la nueva oferta

Uso de Template() con *metadata("performative", "bid")* para enviar un mensaje desde A a B:

- Un mensaje puede contener una oferta ([value, value, ..., value])
- Un mensaje puede ser "True" si el agente quiere comunicar que acepta una oferta

Esquema de funcionamento de la comunicaciòn entre dos agentes en una negociacion bilateral en el entorno que hemos implementado.



Funciones de utilidad

Para hacer negociación simple, los agentes necesitan saber el tiempo, y su utilidad de una oferta. Ya que el agente tiene el tiempo de empezar y el duración del negociación de inicialización, calcular el tiempo es trivial. Para explicar como calculan su utilidad, utilizamos nuestro ejemplo. Tenemos definido una problema de negociación lineal con dos variables, *price* y *quality* con

$$price = x_1 \in [0, 10000] \quad quality = x_2 \in [0, 100]. \tag{1}$$

El "seller" tiene pesos [1.2, -0.2], en otros palabras, un precio alto es mas importante, pero un calidad mas bajo tamién le gusta. El buyer tiene pesos [-0.3, 1.3]. Entonces, alta calidad es mas importante, pero también le gusta un precio bajo. También tenemos funciones de valor:

$$V_1(x) = \begin{cases} 0.133 & 0 \leq x < 99.5 \\ 0.141 & 99.5 \leq x < 199 \\ 0.148 & 199 \leq x < 298.5 \\ \dots & \\ 0.867 & 9950 \leq x \end{cases} \qquad V_2(x) = \begin{cases} 0.2 & 0 \leq x < 0.995 \\ 0.206 & 0.995 \leq x < 1.99 \\ 0.212 & 1.99 \leq x < 2.985 \\ \dots & \\ 0.8 & 99.5 \leq x \end{cases} \tag{2}$$

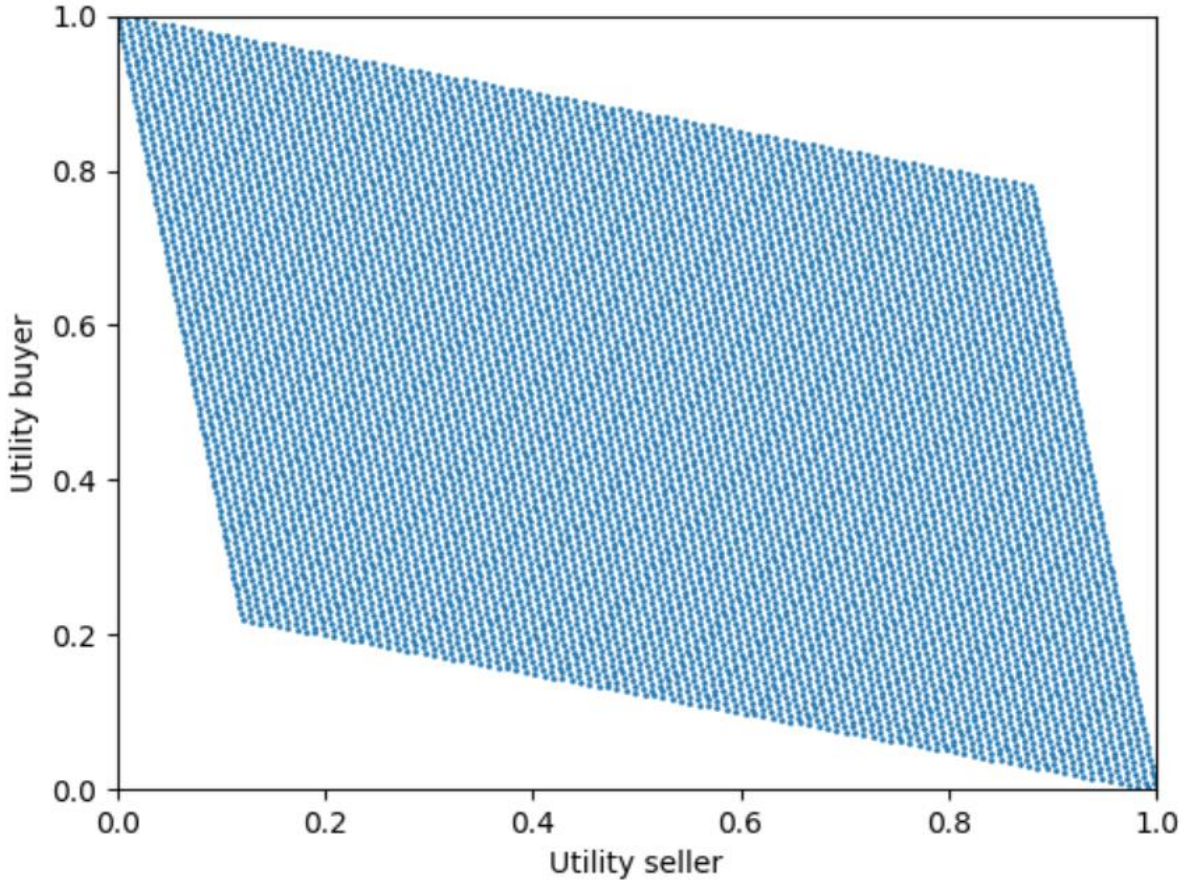
Los funciones parecen un poco raros, pero hay un razón. Tiene 100 distintos valores cada uno para tener un buen cantidad de ofertas posibles. También tiene puntos de empezar/terminar un poco extraños. Los valores de empezar/terminar son elejidos para que el "seller" y el "buyer", con los pesos arriba, ambos tiene utilidad entre 0 y 1. Si asumimos que todos los problemas tienen funciones de valor con *c* distintos valores entre *a* y *a + b,* con la misma differencia, podemos calcular los funciones según la ecuación

$$V_i(x) = a_i + \frac{\lfloor x/b_i \rfloor}{c_i} \cdot d_i. \tag{3}$$

Entonces, para nuestra problema, tenemos $a_1 = 0.133$, $b_1 = 99.5$, $c_1 = 100$, $d_1 = 0.733$, $a_2 = 0.2$, $b_2 = 0.995$, $c_2 = 100$ y $d_2 = 0.6$. Ya que los agentes tienen también estos valores de su inicialización, pueden calcular V_1 y V_2 según ecuación (3). Al final, ya que tenemos una problema lineal, los utilidades son calculados como

$$U(x_1, x_2) = V_1(x_1) \cdot w_1 + V_2(x_2) \cdot w_2. \tag{4}$$

Los agentes pueden calcular la utilidad ya que tienen también sus pesos. Figure 1 muesta todas las ofertas posibles con este problema.



Todas las ofertas posibles con dado problema.

Generaciòn de ofertas

Todas las ofertas entre las cuales un agente tiene que decidir cual proponer a el otro agente son calculadas con el funcion GetOffers(S_low, S_high) en el script Simulator. Este funcion al principio produce 20 ofertas aleatorias. El objetivo de la funcion es modificar los ofertas para tener todos con utilidad entre S_low y S_high. Para hacer este modificación, un tipo de "gradient descent" esta implementado. Para cada oferta, al principio, el funcion comproba si la oferta tiene una utilidad demasiada alta o baja (o si está bien). Si es demasiada baja, es necesario aumentar la utilidad. Ya que asumimos que tenemos funciones lineales, con funciones de valor como en ecuación (3), vimos que para incrementar el funcion de valor un paso, necesitamos añadir el valor de b_i al valor de x_i. Ya que los pesos pueden ser negativos en nuestro problema, multiplicamos el paso con el peso tambien, para saber que estamos aumentando la utilidad. Si la utilidad de la oferta es demasiada alta, hay que hacer el paso en el otro dirección, y por eso multiplicamos el paso con *-1* en este caso. Al final, un paso de nuestro versión de "gradient descent" es definido como $x_{new} = x_{old} + 0.5bwk$ donde k = 1 si la utilidad es demasiada baja, y k = -1 si la utilidad es demasiada alta. El constante 0.5 solo es un "steplength" para tener pasos mas pequeños. Este paso esta repitido hasta que la oferta tiene la utilidad que queremos. Al final, el funcion GetOffers(S_low, S_high) vuelve 20 ofertas aleatorias que esta modificado para tener utilidad entre S_low y S_high. El codigo pseudo está abajo:

```
def getOffers(S_low, S_high):

    generate 20 offers randomly

    for offer in offers:
        if offer too low:
            k = 1
        else if offer too high:
            k = -1
        else:
            continue

        while offer too high/low:
            offer = offer + 0.5 * b * w * k

    return offers
```

Terminar la negociaciòn

En el entorno desarrollado la negociaciòn termina en dos casos:

- los dos agentes encuentran un acuerdo, y paran sus ejecuciònes
- hay un tiempo de negociaciòn definido en input al entorno. Si expira, termina la ejecuciòn de los agentes que no encuentran un acuerdo

Observaciòn: extender al caso en que dos agentes encuentran siempre un acuerdo, es suficiente que los agentes gestionen el problema en sus clases y no debe ser definido en el entorno.

Ejecutar el entorno

El entorno desarrollado se puede ejecutar en la consola con el orden

python main.py <sellerAgentName> <buyerAgentName> <negotiationTime(seconds)> <host (xmpp server name)>

después haber activado el server XMPP

- <sellerAgentName> es el script python que defines el agente de rol = “seller”
- <buyerAgentName> es el script python que defines el agente de rol = “buyer”
- <negotiationTime(seconds)> duraciòn de la negociaciòn
- <host (xmpp server name)> domain xmpp de el server usado para el entorno SPADE

Los dos agentes extendon a la clase NegotiationAgent definida en el script negotiationAgent.py e por cada uno de ellos es posible editar algunos parametros para la negociaciòn y hacer el override de los funciones definidos en la clase padre.

Parametròs definidos para la estrategia de conceciòn:

- RU: defines la utilitad de reserva de el agente
- beta: nivel de concesiòn de el agente (beta = 1 lineal)
- S: nivel de aspiraciòn (S es calculado cada vez el agente debe proponer una nueva oferta)

$$s_a(t) = 1 - (1 - RU_{a_i}) \left(\frac{t}{T_{a_i}} \right)^{\frac{1}{\beta_{a_i}}}$$

Hemos definido dos diferentes agentes para mostrar un ejemplo de la ejecuciòn de el entorno desarrollado

Ejemplo de ejecution

En los siguientes paragrafos estan definidos los parametros y las estrategias empleadas en el ejemplo de ejecuciòn de el entorno.

Agentes empleados:

- boulwardAgent: <sellerAgentName>
- concederAgent: <buyerAgentName>

Activar el server XMPP y luego executar en la consola:

python main.py boulwardAgent concederAgent 60 localhost

Salvo por la estrategia de conceciòn no hay ninguna diferencia entre los dos agentes empleados por el ejemplo.

Estrategia de conceciòn

- boulwardAgent
    - RU = 0.5
    - beta = 0.5
    - S = 0.99

- concederAgent
    - RU = 0.5
    - beta = 2
    - S = 0.99

Criterio de aceptaciòn

El oferta es aceptada si su funciòn de utilidad es major de el nivel de aspiraciòn S.

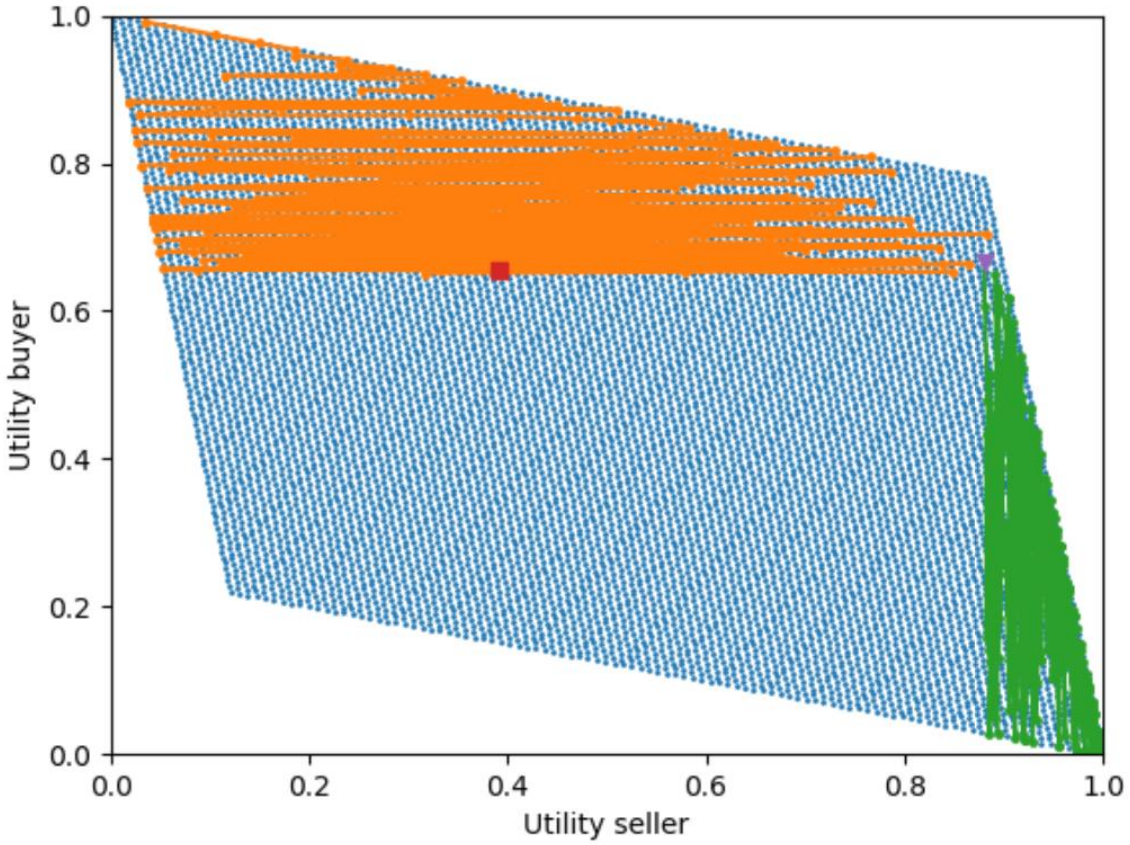
Generaciòn de Ofertas

Entre las 20 ofertas generadas con OGD se selecciona de manera aleatoria una oferta que tiene un utilidad entre el interval [S , S+0.01] donde S es el nivel de aspiraciòn actual de la agente que genera una oferta.

Plot de los resultados

En el script simulator se muestran en tiempo real las ofertas enviadas entre los agentes durante la negociaciòn.

Plot de los resultados obtenidos ejecutando la negociaciòn entre los dos agentes propuestos en nuestro ejemplo (RU = 0.5):



Acuerdo encontrado. El punto púrpura indica la utilidad con la cual los agentes han encontrado el acuerdo.

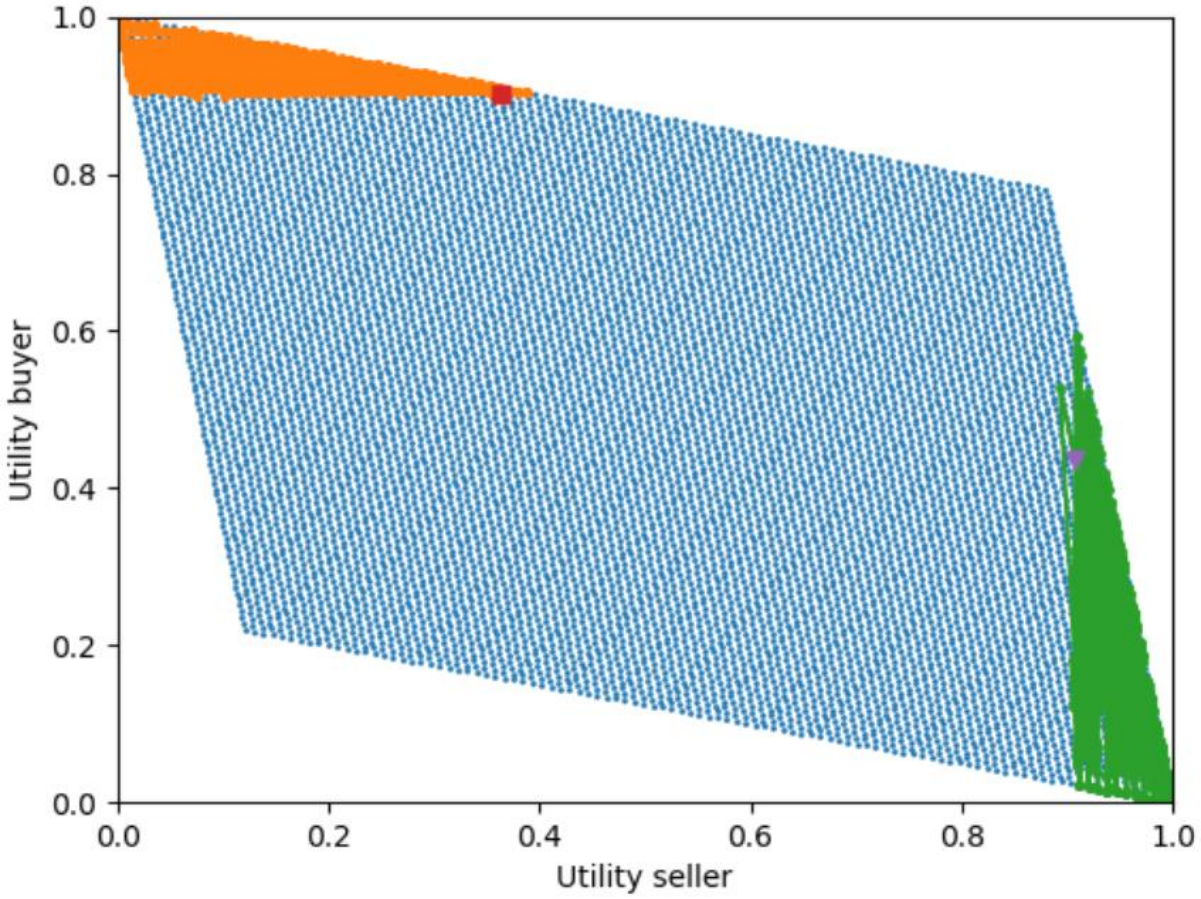
Oferta aceptada del Buyer: [9827 (price), 85 (quality)]

```
seller@localhost received message
last offer from seller@localhost not accepted by buyer@localhost
seller@localhost sending offer 9826.8714224841,84.92238076532944 to buyer@localhost

buyer@localhost received message
last offer from buyer@localhost not accepted by seller@localhost
buyer@localhost accepts offer received by seller@localhost
terminating agent

seller@localhost received message
last offer from seller@localhost accepted by buyer@localhost
terminating agent
```

Plot obtenido ejecutando el mismo problema ponendo RU = 0.9 (utilidad de reserva) por ambos los agentes:



Los agentes no han encuentrado un acuerdo y el tiempo de negociacion (60 segundos) ha expirado.