Using the NegoSim to Simulate the Automated Negotiations

Arash Ebrahimnezhad[1] and Katsuhide Fujita[2]

[1] Faculty of electrical and computer engineering, Babol Noshirvani University of Technology, Babol, Iran
Arash.ebrah@gmail.com
[2] Institute of Engineering, Tokyo University of Agriculture and Technology, Tokyo, Japan
katfuji@cc.tuat.ac.jp

Abstract. NegoSim is an automated negotiation simulator. This document helps users to install and run NegoSim, develop their automated negotiation agents, protocols, analysis entities, and customize the GUIs of NegoSim.

1 Architecture of NegoSim

Fig. 1. The most important entities of NegoSim are the negotiation table (NegoTable), parties, analysis entity, and protocol entity. The protocol is the most critical entity which controls all negotiation events.

In Fig. 1 the important entities of NegoSim are demonstrated. In table 1 the entities of NegoSim and their description are detailed. The important classes and their relationships are shown in Fig. 2. As it is shown in Fig. 2 the parties do not have direct access to the protocol but each time the protocol calls the parties to send their bid, the parties are provided access to the protocol. The *send_bid* abstract method in *AbstractNegoParty* class gets the protocol as its argument, see Fig. 3.

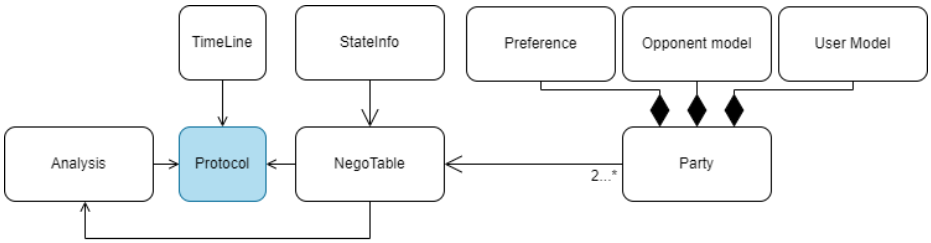

Fig. 2. Partial UML of core of NegoSim. The important classes of NegoSim are depicted.

```python
@abstractmethod
def send_bid(self, protocol: ProtocolInterface) -> Bid:
    raise NotImplementedError()
```

Fig. 3. send_bid is an abstract method that should be implemented by the agent developer. This method has an arguments *protocol*. This method is called by the protocol and it passes this parameter to the *send_bid* method, so the party has access to the protocol in this method.

Table 1. The most important entities of NegoSim are described.

Entity	Description
Negotiation table (NegoTable)	It has access to the negotiation parties and status information (StateInfo) which refers to the negotiation state (0: negotiation is ongoing, 1: negotiation is ended with reaching an agreement, -1: negotiation is ended without reaching an agreement). Exchanged offers would be added to the negotiation table; in other words, the negotiation table contains the exchanged offers.
Parties	The participants in the negotiation are known as parties.
Analysis	It has access to NegoTable and consequently parties' preferences, parties' opponent models, and user models (if those exist) to analyze the negotiation result and agents' performance.
Protocol	The most important entity which controls the negotiation is protocol. A protocol has access to three entities: 1. It accesses the negotiation table and consequently the parties and their exchanged offers. 2. It accesses to the Analysis entity which can be used to analyze the negotiation and parties' performance. 3. It accesses the TimeLine entity which is used by the protocol to control the negotiation time. Parties can access the protocol to ask any question

	about the negotiation and the protocol answers them according to the negotiation laws.

2 Domain, Preference

If it is assumed the negotiation would be on the buying/selling laptop (Laptop domain) and if it is assumed a laptop has three negotiable issues (Brand, Hard disk capacity, and Monitor size), each of these issues could have several items, see table 2.

Table 2. Laptop domain with 3 negotiable issues

Domain name	Issues and weights	Values and evaluations
Laptop	**Brand** weight = (0.6)	**Dell** evaluation = 1
		Lenovo evaluation = 2
		Mac evaluation = 3
	Hard disk capacity weight = (0.25)	**1 T** evaluation = 1
		2 T evaluation = 4
		3 T evaluation = 5
	Monitor size weight = (0.15)	**19 inch** evaluation = 7
		17 inch evaluation = 5
		15 inch evaluation = 1

Each issue has a weight that represents its priority e.g. $w(Brand) = 0.6$ and for each item, a value is assigned e.g. $eval(Dell) = 1$.

3 Utility Space

NegoSim does not restrict the users (researchers/developers) to using predefined utility spaces. They can develop their own utility spaces. In NegoSim linear additive utility space is defined (see *NegoSim/utility_spaces/AdditiveUtilitySpace.py*). To create a special utility space the developer should extend *AbstractUtilitySpace* (NegoSim/core/*AbstractUtilitySpace*). The abstract methods of *AbstractUtilitySpace* class are shown in Table 3.

Table 3: abstract methods of *AbstractUtilitySpace* class that should be implemented.

Method	Description
get_utility	This method receives a bid and returns a utility of it.
get_utility_distinct	This method receives a bid and returns a utility of it according to the negotiation time passed from the beginning of the negotiation: $u'(\omega) = u(\omega) * d^t$ in which u' is the utility of ω with time pressure, u is the utility of bid ω without time

	pressure and d is discounted factor which is defined in preference and t is the time passed from the beginning of the negotiation (see section 4.1).

The *AbstractUtilitySpace* class which you can find in path *NegoSim/utility_spaces/ AbstractUtilitySpace.py* uses additive utility space. This kind of utility space uses a linear additive formula to calculate the utility of a bid, see Eq. 1. In this way of calculating the utility of a bid will be in the range of 0 and 1 (Eq. 2).

$$u(\omega) = u(v^1, \dots, v^n) = \sum_i^n w(i) * \frac{eval(v^i)}{\max(eval(I_i))} \quad (1)$$

$$u(\omega) \in [0,1] \quad (2)$$

In Eq. 1, ω and $u(\omega)$ refer to the possible outcome of a domain and utility of the ω respectively and $(v^1, \dots, v^n)$ refers to the values of the issues, if it is assumed the $\omega = (v^1, v^2, v^3) = (Dell, 1T, 19\ inch)$ then $u(\omega)$ equals to:

$$u(\omega) = u(Dell, 1T, 19\ inch) = \sum_{i=1}^{3} w(i) * \frac{eval(v^i)}{\max(eval(I_i))}$$

$$= w(Brand) * \frac{eval(Dell)}{\max(eval(Dell), eval(Lenovo), eval(Mac)} + w(Hard\ Disk) * \frac{eval(1T)}{\max(eval(1T), eval(2T), eval(3T)} + w(Monitor\ Size) * \frac{eval(19inch)}{\max(eval(19inch), eval(17inch), eval(15inch)}$$

$$= 0.6 * \frac{1}{3} + 0.25 * \frac{1}{5} + 0.15 * \frac{7}{7} = 0.4$$

4 Preference creation

NegoSim provides two ways to create a preference: using XML files and Python dictionary data structure.

4.1 Using xml files

To create a domain, it is enough to create a folder with the domain name in path *NegoSim/Domains*, e.g. *laptop*. In the laptop directory, at least two preference files in XML format should be created, e.g. *laptop_buyer_utility.xml* and *laptop_seller_utility.xml*. Each of these XML files should contain scripts like Fig. 4.

```
1  <utility_space>
2      <objective index="1" etype="objective" type="objective" description="" name="LaptopDomain">
3          <issue index="2" etype="discrete" type="discrete" vtype="discrete" name="Brand">
4              <item index="1" value="Dell"  cost="700"  evaluation="1" description="cheap laptop"></item>
5              <item index="2" value="Lenovo"  cost="800" evaluation="2" description="quality laptop"></item>
6              <item index="3" value="Mac"  cost="900" evaluation="3" description="gamer laptop"></item>
7          </issue>
8          <issue index="3" etype="discrete" type="discrete" vtype="discrete" name="Hard Disk">
9              <item index="1" value="1T"  cost="700"  evaluation="1" description="the default internal HD"></item>
10             <item index="2" value="2T"  cost="800" evaluation="4" description="some extra space"></item>
11             <item index="3" value="3T"  cost="900" evaluation="5" description="enough video editing"></item>
12         </issue>
13         <issue index="4" etype="discrete" type="discrete" vtype="discrete" name="Monitor Size">
14             <item index="1" value="19 inch"  cost="700"  evaluation="7" description="nothing special"></item>
15             <item index="2" value="17 inch"  cost="800" evaluation="5" description="high contrast"></item>
16             <item index="3" value="15 inch"  cost="900" evaluation="1" description="nice to play movies"></item>
17         </issue>
18         <weight index="2" value="0.6"></weight>
19         <weight index="3" value="0.25"></weight>
20         <weight index="4" value="0.15"></weight>
21     </objective>
22     <discount_factor value="1.0"></discount_factor>
23     <reservation value="0.0"></reservation>
24 </utility_space>
```

Fig. 4. A preference for the laptop domain is defined in an XML file.

As it is shown in Fig. 4, all tags must be inside *utility_space* tags (*<utility_space> … </utility_space >*). An issue is defined by the issue tag and inside the issue tags the available items and their evaluations are defined (lines 3 to 7, 8 to 12, and 13 to 17 define three items and their evaluations). The weights of issues are defined in lines 18, 19, and 20.

Some negotiation scenarios could have time pressure, in this situation the utility is calculated by Eq. 3.

$$u'(\omega) = u(\omega) * d^t \tag{3}$$

In Eq. 3, $u(\omega)$ refers to the utility of bid ω, and the utility of that bid with time pressure is shown by $u'(\omega)$. d and t ($t \in [0, 1]$) refer to discounted factor and time respectively. the discounted factor is defined in line 22 in Fig. 4. If discount factor is equal to 1, it means there is not time pressure.

Some negotiation scenarios could have reservation value which means if the negotiation would be ended without an agreement the agent could have the utility equal to the reservation value. In other words, the rational negotiation agent would not accept a bid with a utility that is less than the reservation value. In Fig. 4 the reservation value is defined in line 23.

> **Please Note:** A preference should have reservation value and discounted factor tags, in other words, lines 22 and 23 in Fig. 4 are crucial. If there is no time pressure and reservation value you can evaluate them as 1 and 0 respectively.

4.2 Using Python dictionary data structure

NegoSim provides developers to create a preference by creating a python dictionary like Fig. 5In this way of preference creation the issues are defined as dictionary keys and their values should be a list with 2 items, first item determines the weight of the issue and the second one should be a dictionary in which the key and values of it refer to the item and evaluation of the item, see lines 2 to 4 in Fig.5. In lines 5 and 6 in Fig. 5 discount factor and reservation value are defined.

```
1  {
2      'Barnd': ['0.6', {'Dell': '1', 'Lenovo': '2', 'Mac': '3'}],
3      'Hard Disk': ['0.25', {'1T': '1', '2T': '4', '3T': '5'}],
4      'Monitor Size': ['0.15', {"19 inch": '7', "17 inch": '5', "15 inch": '1'}],
5      'discount_factor': '1.0',
6      'reservation': '0.0'
7  }
```

Fig. 5. Lines 2 to 4 define issues and their weights and items. Lines 5 and 6 define discounted factor and reservation value respectively.

To create a preference, after creating a Python data structure like Fig. 5, you can simply send this data structure as a parameter to the *init* method of the Preference class.

> **Please Note:** A preference should have reservation value and discounted factor, in other words, lines 5 and 6 in Fig. 5 are crucial. If there is no time pressure and reservation value you can evaluate them as 1 and 0 respectively.

5 Protocol

A protocol is the most important entity at the NegoSim. It determines how the agents communicate with each other. NegoSim provides some APIs through which the users can develop their protocol, however, there is a predefined protocol called *SOAP* [1]. In the following, you can find the implementation of *SOAP* and how to create a protocol.

5.1 Create a protocol

To create a protocol, a user at least must implement the *core.ProtocolInterface*. We recommend extending the *core.AbstractProtocol* which is a basic implementation of the *core.ProtocolInterface*. The abstract methods that should be implemented by a protocol developer are demonstrated in table 4:

Table 4. Abstract methods of abstract class *AbstractProtocol* and their descriptions.

Method	Description
negotiate	This method asks for a bid from parties according to sequence and negotiation status, after receiving a bid then convert it to an offer (by adding time to the bid) and add it to the offers on the table, and updates the status of the negotiation. • Negotiation status will be 1 if the last parties' offers are the same which means the negotiation was ended by reaching an agreement. • Negotiation status will be 0 if the last parties' offers are not the same or an instance of the *EndNegotiation* (*core/AbstractNegoParty*) class which means the negotiation is continuing. • Negotiation status will be -1 if at least the one of last received bids is an instance of *EndNegotiation* class or negotiation was ended without an agreement.
get_offers_on_table	This method gets a party name in string type and returns a tuple of offers related to the party name that has got through the object that has called the method. Before returning the offers, the method checks out whether the object that has called the method has the authority to access the offers or not? If there is no permission it returns an error. Predefined SOAP protocol lets all parties know each other's offers.

NegoSim has provided SOAP protocol (see negosim/protocols/SOAP.py). A protocol accesses to the *TimeLine*, *NegoTable* and *AnalysisMan* objects through its parent class and accesses to negotiation *Parties* and *StateInfo* objects through the *NegoTable*. A protocol is obliged 3 tasks:

1. Investigation of negotiation status using this line of code: *self.get_nego_table().get_state_info().get_negotiation_state()*. Lines 1 and 6 in Fig. 6. This line of code can return 3 different values:
    a. 1: the negotiation ended by reaching an agreement
    b. -1: the negotiation ended without reaching an agreement
    c. 0: the negotiation is ongoing.

   A protocol must ask parties for bids if the negotiation is ongoing.

2. If the negotiation is ongoing, before asking parties for bids, a protocol should investigate whether is there negotiation time left? To investigate the negotiation time a protocol must use this line of code: *self.get_time_line().is_time_ended()*, line 4 in Fig. 6. If this line of code return true then the protocol asks parties for bids using this line of code: *party.send_bid(self, self.get_time_line())*, line 7 in Fig. 6, otherwise updates the status of the negotiation to -1 using this line of code:

self.get_nego_table().get_state_info().set_negotiation_state(-1), lines 5 and 17 in Fig. 6.

3. Every time that the protocol gets a bid, the protocol has to convert the bid to an offer using this line of code: *Offer(bid, self.get_time())* then add this offer to the negotiation table using this line of code: *self.get_nego_table().add_offer(party, offer)*, lines 8 and 9 in Fig. 6.
   If you want track each party's accuracy of user modeling and opponent modeling, you should call *cal_estimation_analysis_data()* method, line 11 in Fig. 6. *cal_estimation_analysis_data()* is a method of analysis entity, see section 6.

```
1  while self.get_nego_table().get_state_info().get_negotiation_state() == 0:
2      parties = self.get_nego_table().get_parties()
3      for party in parties:
4          if self.get_time_line().is_time_ended():
5              self.get_nego_table().get_state_info().set_negotiation_state(-1)
6          if self.get_nego_table().get_state_info().get_negotiation_state() == 0:
7              bid = party.send_bid(self, self.get_time_line())
8              offer = Offer(bid, self.get_time())
9              self.get_nego_table().add_offer(party, offer)
10             print(party.get_name(), ' -> ', offer)
11             self.get_analysis_man().cal_estimation_analysis_data()
12         if self.is_agreement() is True:
13             self.get_nego_table().get_state_info().set_negotiation_state(1)
14             print("Negotiation was ended due to reaching Agreement!")
15             break
16         if self.get_time() >= 1.0:
17             self.get_nego_table().get_state_info().set_negotiation_state(-1)
18             print("Negotiation was ended due to reaching deadline!")
19             break
```

Fig. 6. Three crucial tasks that a protocol should implement.

> **Please Note:** if you want to use your developed protocol using NegoSim GUI, you have to create a class with the same name as the module name. E.g. if you create a module SOAP.py, you should create an SOAP class.

6 Create a negotiation party

To create an automated negotiation party, the *AbstractNegoParty* (in module *core.AbstractNegoParty*) class should be extended. The abstract methods of *AbstractNegoParty* are shown in table 5.

Table 5. Abstract methods of *AbstractNegoParty* class.

Method	description
send_bid	This method is called by protocol. This method decides which bid should be sent to the opponent. In this method, an agent can send the opponent's last offer if it wants to accept opponent's offer. If an agent wants to end the negotiation without a result (walk away) sends an instance of *EndNegotiation* class otherwise sends its counteroffer
get_name	The name of the agent should be declared here.
get_opponent_model	If you want the accuracy of the opponent model to be investigated by the *AnalysisMan* entity this method should return the opponent model. If you do not want to investigate the accuracy of the opponent model you can write "*return None*".
get_user_model	If your agent was developed to negotiate in uncertain situations and you want the accuracy of the user model to be investigated by the *AnalysisMan* entity, this method must return the user model, otherwise please write "*return None*" in this method.

There are some negotiation parties in the path *NegoSim/Agents*. The RandomParty1 is a simple agent which chooses the random bid to send to the opponent. The important method of RandomParty1 is *send_bid* which is shown in Fig. 7:

```
1  def send_bid(self, protocol) -> Bid:
2
3      utility_space = self.get_utility_space()
4
5      parties = protocol.get_parties()
6      opponent = list(filter(lambda party: party is not self, parties))[0]
7      opponent_offer = protocol.get_offers_on_table(opponent)
8
9      bid = self.generate_random_bid()
10     if len(opponent_offer) > 0:
11         op_bid = opponent_offer[len(opponent_offer) - 1].get_bid()
12         if utility_space.get_utility(op_bid) >= utility_space.get_utility(
13                 bid) and utility_space.get_utility(op_bid) > 0.7:
14             return op_bid
15     return bid
```

Fig. 7. This Fig. shows the *send_bid* method for the RandomParty1.

The agent's utility space is gained in line 3. An agent can access the protocol so if the agent has any questions about the opponent(s), exchanged bids and etc. should ask protocol, so in Fig. 1 the RandomParty1 asks protocol about all participants in the negotiation, line 5. The opponent object is gained using line 6. All negotiation exchanged offers are on the table, so if the agent wants to access these offers should ask the protocol to return the existing offers on the table. Line 7 shows RandomParty1 wants to know the received offers. The bid which may be sent to the opponent is gen-

erated randomly in line 9 (next_bid). If the opponent had sent the offer at least one offer (line 10) then the last opponent's offer is extracted (op_bid, line 11). The utility of the op_bid and next_bid regarding its utility space are calculated in lines 12 and 13 (op_bid_u, next_bid_u) respectively. Line 12, 13 investigate if the op_bid_u is greater than or equal to next_bid_u and the op_bid_u is greater than 0.7 then accept the opponent's offer (line 14); otherwise, send the counteroffer to the opponent (line 15). In NegoSim if an agent accepts the opponent's offer, then just sends the received offer to the opponent. When the agents' last offers are equal the protocol announces the negotiation has been finished by reaching an agreement. For full implementation of RandomParty1, see NegoSim/Agents/RandomParty1.py.

> **Please Note:** An agent can access the protocol so if the agent has any questions e.g. about the opponent(s), exchanged bids and etc. should ask the protocol.

There is *EndNegotiation* class in the module *AbstractNegoParty* (*core.AbstractNegoParty*). This class has inherited the Bid class so if an agent wants to finish the negotiation (walk away), it can create an instance of *EndNegotiation* class and send it to the opponent. The protocol investigates the agents' offers and if it is an instance of *EndNegotiation* then finishes the negotiation.

> **Please Note:** if you want to use your developed agent using NegoSim GUI, you have to create a class with the same name as the module name. E.g. if you create a module RandomParty1.py, you should create a RandomParty1class.

6.1 BOA framework

The other way to develop a negotiation agent is using the BOA [2] framework. In this way, the user must develop three components: Bidding strategy, Opponent model, and Acceptance strategy. NegoSim provides predefined BOA components. *RandomStrategy* as bidding strategy which selects the bid randomly (see NegoSim /bidding_strategies/RandomStrategy.py), *DefaultOpponentModel* as opponent modeling (see NegoSim/opponent_models/DefaultOpponentModel.py) and, *ACNext* [3] as acceptance strategy (see NegoSim/acceptance_strategies/ACNext.py). RandomParty2 uses the BOA framework (see NegoSim/Agents/RandomParty2.py). To use the BOA framework, the developer of the agent must extend the *AbstractNegoParty* and override the *init* method, see Fig. 8.

```
1  def __init__(self, utility_space: AbstractUtilitySpace):
2      super().__init__(utility_space=utility_space)
3      self.__utility_space = self.get_utility_space()
4      preference = self.__utility_space.get_preference()
5      self.opponent_model = DefaultOpponentModel(preference=preference.get_initial_preference())
6      self.bidding_strategy = RandomStrategy(opponent_model=self.opponent_model, preference=preference)
7      self.acceptance_strategy = ACNext(utility_space=self.__utility_space)
```

Fig. 8. This Figure illustrates how to override the *init* method

Line 3 in Fig. 2 gets the agent's utility space and uses it to extract the agent's preference (line 4). The agent's preference is used to define the opponent model (*self.opponent_model* line 5). In line 5 the bidding strategy was defined (*self.bidding_strategy*). Line 6 defines the agent's acceptance strategy (*self.acceptance_strategy*).

RandomParty2 uses this line of code: bid = self.bidding_strategy.send_bid(timeline) to find a bid to send to the opponent, to update the opponent model uses this line of code: self.opponent_model.update_preference(op_offer) and, to decide whether to accept the opponent offer or not uses this line of code: *self.acceptance_strategy.is_acceptable(offer=op_offer, my_next_bid=bid, opponent_model=self.opponent_model)*. If this line returns True value then the agent accepts the opponent's offer otherwise sends the counteroffer.

Implement Bidding strategy. To implement the bidding strategy in the BOA framework, we recommend extending the *AbstractBiddingStrategy* class. You must implement the abstract method *send_bid*. NegoSim has provided *RandomStrategy* (see NegoSim/bidding_strategies/RandomStrategy.py). *AbstractBiddingStrategy* class provides a method to generate random bids, so the *RandomStrategy* is like Fig. 3:

```
1  from core.AbstractBiddingStrategy import AbstractBiddingStrategy
2  from core.TimeLine import TimeLine
3  from core.Bid import Bid
4
5
6  class RandomStrategy(AbstractBiddingStrategy):
7
8      def send_bid(self, timeline: TimeLine) -> Bid:
9          return self.generate_random_bid()
```

Fig. 9. *RandomStrategy* class has extended the *AbstractBiddingStrategy* and Implemented the *send_bid* method.

Implement Opponent model. To implement an opponent model in the BOA framework, we recommend extending the *AbstractOpponentModel*. This class has a method

get_initial_opponent_preference() which returns an initial preference. In initial preference, the weights are calculated by Eq. 1:

$$w_i = \frac{1}{n} \tag{4}$$

In Eq. 1, w_i and n refer to the weight of i'th issue and the number of issues in the domain respectively. The evaluation of a value is set to 1. *AbstractOpponentModel* class has an abstract method *update_preference(self, offer: Offer)* that should be implemented.

Implement Acceptance strategy. To implement an acceptance strategy, we recommend extending the *AbstractAcceptanceStrategy*. This class has an abstract method *is_acceptable* which should be implemented. This method can return two integer values: 0 and 1 referring to reject and accept the opponent's offer, respectively. NegoSim has provided an acceptance strategy which is called ACNext (see NegoSim/acceptance_strategies/ACNext).

6.2 Developing a party with preference uncertainty

In preference uncertainty the agent does not know the exact preference of the user; instead of it the user gives the agent some ranked bids:

$$\omega_1 \preccurlyeq \omega_2 \preccurlyeq \cdots \omega_{k-1} \preccurlyeq \omega_k \tag{5}$$

In Eq. 1 ω refers to a possible outcome of domain space, symbol $\preccurlyeq$ determines the desirability of a possible outcome relative to the others, e.g. $\omega_1 \preccurlyeq \omega_2$ demonstrates ω_2 is more appropriate than ω_1 for the user. The agent can ask the user the rank of special ω but it might have some penalty which is called bothering cost.
To develop a party with preference uncertainty, NegoSim has provided the EUBOA framework, see Fig. 10.

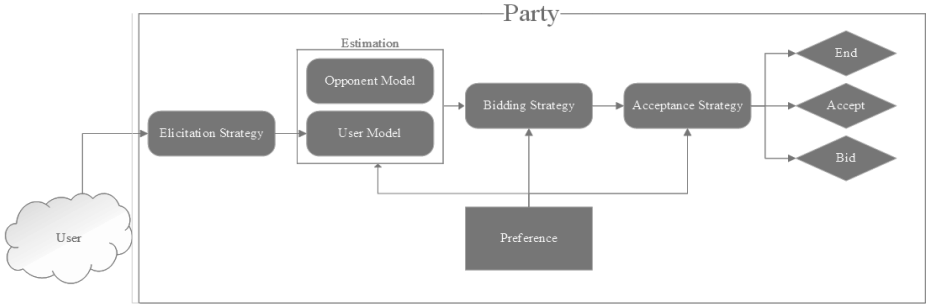

Fig. 10. EUBOA framework components are the **E**licitation strategy, **U**ser model, **B**idding Strategy, **O**pponent model, and **A**cceptance strategy. The opponent model and User model components make the estimation entity.

EUBOA framework is comprised an Elicitation strategy that decides when and which bid would be asked from the user to get its rank, and User model which tries to model the user preference, and other three components (BOA) which are like the BOA section (section 4.1). NegoSim has provided *DefaultElicitationStrategy* class as elicitation strategy (see *NegoSim/elicitation_strategies/DefaultElicitationStrategy.py*) which chooses a bid randomly to elicit and *DefaultUserModel* class as user model (see *NegoSim/user_models/default_user_model.py*). To develop an agent with preference uncertainty, we recommend extending *AbstractNegoPartyUncertainCondition* class. First of all, you have to override the *init* method, see Fig. 11:

```
1  def __init__(self, utility_space: AbstractUtilitySpace, user: UserInterface):
2      super(RandomAgentEUBOA, self).__init__(utility_space=utility_space, user=user)
3      self.__user = self.get_user()
4      self.initial_preference_user_model = self.get_initial_preference()
5      self.initial_preference_opponent_model = self.initial_preference_user_model.__copy__()
6
7      self.__user_model = DefaultUserModel(self.initial_preference_user_model)
8      self.__elicitation_strategy = DefaultElicitationStrategy(user=self.__user, user_model=self.__user_model)
9      self.__opponent_model = DefaultOpponentModel(self.initial_preference_opponent_model)
10     self.__bidding_strategy = RandomStrategy(opponent_model=self.__opponent_model,
11                                              preference=self.initial_preference_user_model)
12     self.__acceptance_strategy = ACNext(utility_space=AdditiveUtilitySpace(self.initial_preference_user_model))
```

Fig. 11. init method of the agent which works with uncertain preference.

In Fig. 11, in line 3 the user which should be represented by the agent is extracted (*self.__user*), in lines 4 and 5 the initial preference is assigned to variables *self.initial_preference_user_model* and *self.initial_preference_opponent_model* respectively. From lines 7 to 12 the EUBOA components are defined.

To develop the agent with preference uncertainty, the *AbstractNegoPartyUncertainCondition*'s abstract methods need to be implemented. The abstract methods are the same as in table 5.

```
1  def send_bid(self, protocol: ProtocolInterface, timeline: TimeLine) -> Bid:
2      """
3      send new bid, send same bid refer to accept, send {} refer to end negotiation
4      :return: Bid
5      """
6      state_info = protocol.get_state_info()
7      self.__elicitation_strategy.is_asking_time_from_user(state_info=state_info)
8
9      parties = protocol.get_parties()
10     opponent = list(filter(lambda party: party is not self, parties))[0]
11     opponent_offers = protocol.get_offers_on_table(opponent)
12     bid = self.__bidding_strategy.send_bid(timeline)
13     if len(opponent_offers) > 0:
14         op_offer = opponent_offers[-1]
15         self.__opponent_model.update_preference(op_offer)
16         if self.__acceptance_strategy.is_acceptable(offer=op_offer, my_next_bid=bid,
17                                                     opponent_model=self.__opponent_model):
18             return op_offer.get_bid()
19     return bid
```

Fig. 12. *send_bid method is implemented to decide about accepting the opponent's offer or ending the negotiation or sending a counteroffer.*

In Fig. 12, in line 7 the elicitation strategy is called the *is_asking_time_from_user* method to decide should the user be asked for a certain bid or not. In line 9 all negotiation participants are extracted and consequently, in line 10, the opponent is extracted. The opponent offers are asked from the protocol in line 11. The candidate bid to send to the opponent is generated by the bidding strategy in line 12. In lines 13 to 19, the agent decides whether to accept the opponent's offer or send a counteroffer. You can find a simple agent with preference uncertainty in *NegoSim/Agents_uncertainty/RandomAgentEUBOA.py*.

Implement Elicitation strategy.
To develop an elicitation strategy we recommend extending *AbstractElicitationStrategy* class. This class has an abstract method that should be extended (*is_asking_time_from_user*). An elicitation strategy is obliged to do three tasks:

1. Getting initial ranked bids from the user using this line of code *initial_ranked_bids = self.ask_initial_ranked_bids_from_user()* and submit them to the user model entity to generate initial preference using this line of code *user_model.generate_initial_preference(initial_ranked_bids)*, see lines 2 to 5 in Fig. 13.
2. The user model entity might have a question about the order of some bids, so the elicitation strategy has to ask those bids ranking from the user and then updates the user model. See lines 7 to 12 in Fig. 13.
3. The elicitation strategy can have an algorithm to ask for the order of bids in order to improve the user model. See lines 14 to 19 in Fig. 13.

```
1  def is_asking_time_from_user(self, state_info: StateInfo):
2      user_model: UserModelInterface = self.get_user_model()
3      if self.get_initial_ranked_bids() is None:
4          initial_ranked_bids = self.ask_initial_ranked_bids_from_user()
5          user_model.generate_initial_preference(initial_ranked_bids=initial_ranked_bids)
6      else:
7          offers_must_be_asked = user_model.get_must_be_asked_offers()
8          if len(offers_must_be_asked) > 0:
9              for offer in offers_must_be_asked:
10                 self.ask_offer_rank_from_user(offer=offer)
11             ranked_bids = self.get_ranked_bids()
12             user_model.update_preference(ranked_bids=ranked_bids)
13
14         offers_from_elicitation_strategy = self.simple_elicitation_strategy(
15             user=self.get_user(), user_model=self.get_user_model(), state_info=state_info)
16         for offer in offers_from_elicitation_strategy:
17             self.ask_offer_rank_from_user(offer=offer)
18         ranked_bids = self.get_ranked_bids()
19         user_model.update_preference(ranked_bids=ranked_bids)
```

Fig. 13. This figure shows the implementation of the *is_asking_time_from_user* method.

In Fig. 13, in line 14 the *simple_elicitation_strategy* refers to the algorithm which is implemented to decide which bid should be asked? See Fig. 14:

```python
def simple_elicitation_strategy(self, user: UserInterface, user_model: UserModelInterface,
                                state_info: StateInfo) -> list:

    preference: Preference = user_model.get_preference()
    total_bothering = user.get_total_bothering()
    if total_bothering <= 0.5:
        random_bid = preference.generate_random_bid()
        if random_bid not in self.get_ranked_bids():
            time = state_info.get_time_line().get_time()
            random_offer = Offer(random_bid, time)
            return [random_offer, ]
```

Fig. 14. This figure shows the simple algorithm which is implemented in elicitation strategy. This algorithm simply selects a random bid to ask user.

NegoSim provides a default elicitation strategy which can be found in *NegoSim/elicitation_strategies/default_elicitation_strategy.py*.

Implement User model.
To develop the user model we recommend extending the *AbstractUserModel* class. This class has three abstract methods that should be implemented. These three methods are described in table 6.

Table 6. A user model has three abstract methods that should be implemented.

Method	Description
generate_initial_preference	This method generates the initial preferences of the user using initial ranked bids.
get_utility	This method uses the estimated preferences to calculate the utility of a bid.
update_preference	This method updates the user estimated preferences using a sort of ranked bids.

NegoSim provides *DefaultUserModel* class as a user model (see NegoSim/user_models/default_user_model.py).

7 Create an Analysis entity

There are two kinds of analysis entities, one for analyzing negotiation sessions and the other one for analyzing tournament.

7.1 Analyzing negotiation session

To create an analysis entity, the *AbstractAnalysisMan* should be extended. This class has two abstract methods which should be implemented. These two methods are described in Table 7.

Table 7. the *AbstractAnalysisMan* class has two abstract methods that should be implemented.

Method	Description
cal_estimation_analysis_data	This method calculates the performance of the user model and opponent model (if they are existing). This method should return a python dictionary.
get_analysis_data	This method calculates the result of the negotiation session, e.g. utility of each negotiation agent, social welfare, etc. This method should return a python dictionary.

Both of those abstract methods return a dictionary. The dictionary's keys indicate a measurement and the values refer to their values, see Fig. 15.

```
{
    'parties_utility': {'party1': 0.74, 'party2': 0.85},
    'social_welfare': 1.59,
}
```

Fig. 15. A python dictionary that the abstract methods of *AbstractAnalysisMan* class should return.

An *AbstractAnalysisMan* class has the method *save_analysis_data* which saves the return data of *cal_estimation_analysis_data* and *get_analysis_data* methods as a python pickle (serialized) file. The saved files for a negotiation session can be found in path *NegoSim/Gui/logs*. If you do not want to save files; you can simply override the save method, see Fig. 16.

```
def save_analysis_data(self):
    pass
```

Fig. 16. If you do not want the analysis data would be saved, you can override this method.

NegoSim provides three analysis entities in path *NegoSim/analysis*. These analysis entities are:

1. *Analysis_man0:* this module investigates the utility of agreement for each party and the social welfare value but does not save them as a pickle file.
2. *AnalysisMan1*: this module investigates the utility of agreement for each party and the social welfare value and saves them as a pickle file.
3. *AnalysisMan2*: this module calculates WRMSE [4] for the opponent model and utility of agreement for each party and the social welfare value and saves them as a pickle file.

Unlike other negotiation simulators, NegoSim lets researchers know how their user and opponent modeling works during the negotiation. For example, if you use an opponent modeling algorithm like frequency modeling, for each bid that your algo-

rithm gets, the estimation would be updated. NegoSim provides you to track what happens during the negotiation (on user and opponent modeling). To do that you need to call *cal_estimation_analysis_data()* method after receiving a bid in the protocol, see section 5.1.

> **Please Note:** NegoSim provides you to track what happens during the negotiation (on user and opponent modeling).

> **Please Note:** if you want to use your developed analysis entity using NegoSim GUI, you have to create a class with the same name as the module name. E.g. if you create a module analysis_man0.py, you should create an analysis_man0 class.

7.2 Analyzing tournament

A tournament is comprised of several negotiation sessions. The result of each negotiation session (NS) is added to a list and this list can be accessed by calling the *get_session_analysis_dataset* method. This method (*get_session_analysis_dataset*) returns a list of dictionaries, e.g. If it is assumed there are two agents (A_1, A_2) and one domain with two preferences $D(p_1, p_2)$ then the tournament has two NSs:

1. NS_1: $A_1 p_1\ vs.\ A_2 p_2$ which means agent A_1 with preference p_1 negotiates against agent A_2 with preference p_2
2. NS_2: $A_2 p_1\ vs.\ A_1 p_2$ which means agent A_2 with preference p_1 negotiates against agent A_1 with preference p_2.

The data would be returned by calling the method *get_session_analysis_dataset* would be like in Fig. 17.

```
[
    {
        'parties_utility': {'party1': 0.6, 'party2': 0.75},
        'social_welfare': 1.25,
    },
    {
        'parties_utility': {'party1': 0.74, 'party2': 0.85},
        'social_welfare': 1.5,
    },
]
```

NS 1

NS 2

Fig. 17. There are two agents (A_1, A_2) and one domain with two preferences $D(p_1, p_2)$ so the tournament has two NSs: NS_1, NS_1.

To create a tournament analyzing entity, the *AbstractTournamentAnalysisMan* class should be extended. This abstract class has an abstract method (*get_tournament_analysis_data*) that should be implemented. In this method in order to access the all analysis data of NSs, the method *get_session_analysis_dataset* should be called and iterate on this list to calculate the average. This method should return a dictionary like Fig. 18.

```
{
    'AVG utility1': 0.67,
    'AVG utility2': 0.8,
    'AVG socialwelfare': 1.375
}
```

Fig. 18. The python dictionary sample that the *get_tournament_analysis_data* should return.

As it is mentioned the task of this method is to make an average on data of the session analysis entity, so the tournament and session analysis entity should be compatible with each other, e.g. *AnalysisTournament1* (in *NegoSim/AnalysisTournament*) is compatible with *Analysis_man0* and *AnalysisMan1*. A simple implementation of the *get_tournament_analysis_data* method is depicted in Fig. 19.

```
1  def get_tournament_analysis_data(self) -> dict:
2
3      session_analysis_dataset = self.get_session_analysis_dataset()
4      i = 1
5      for session_analysis_data in session_analysis_dataset:
6          for key, value in session_analysis_data.items():
7              if key.split('_')[0] == 'party1' or key.split('_')[1] == 'SocialWelfare':
8                  if not isinstance(value, list):
9                      if key not in self.tournament_analysis_data:
10                         self.tournament_analysis_data[key] = value
11                     else:
12                         self.tournament_analysis_data[key] = (
13                             value + (self.tournament_analysis_data[key] * i)) / (i + 1)
14                         i = i + 1
15
16     return self.tournament_analysis_data
```

Fig. 19. A simple implementation of the *get_tournament_analysis_data* method which makes an average on Data which are provided by *Analysis_man0* or *AnalysisMan1*.

In Fig. 18, in line 3 *session_analysis_dataset* variable is filled with a python list of all negotiation sessions' data of a tournament. In lines 4 to 14, there is an iteration on the session_analysis_dataset variable to make an average. Line 16 returns the analysis data of the tournament. The superclass *AbstractTournamentAnalysisMan* has the variable *self.tournament_analysis_data*, which you can use like Fig. 19. Of course, you can simply make a python dictionary variable and return it.

> **Please Note:** *get_tournament_analysis_data* method should have the ability to make an average python list of Data which are provided by a special session analysis entity, e.g. *AnalysisTournament1* is implemented to make on Data that are provided by *Analysis_man0* or *AnalysisMan1*.

> **Please Note:** if you want to use your developed analysis entity using NegoSim GUI, you have to create a class with the same name as the module name. E.g. if you create a module *AnalysisTournament1*.py, you should create an *AnalysisTournament1* class.

8 Developing GUI

NegoSim provides some APIs that you can develop your GUI completely modular (NegoSim does not force you to use these APIs to create GUI. For example, the sessionGUI.py module in path *NegoSim/GUI* was developed without these APIs).
NegoSim APIs for creating GUI use the python *Tkinter* library. Before explanation of how to create a GUI using these APIs, it would be good to define some elements first that are needed.

- Root frame: the root frame that all other frames and elements would be added to it. The black rectangle in Fig. 20.
- Horizontal frames: the frames that would be added to the root frame. The blue rectangle in Fig. 20.
- Vertical frames: the frames that would be added on a horizontal frame. The green rectangle in Fig. 20.
- Segments: all *Tkinter* widgets that would be added on a vertical frame are called a segment. The dark red rectangle in Fig. 20.

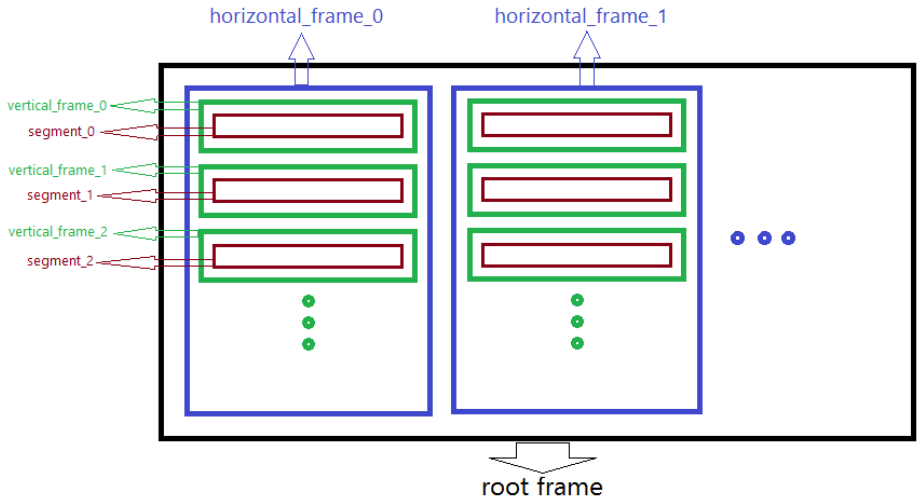

Fig. 20. Black, blue, green, and dark red rectangles refer to the root frame, horizontal frame, vertical frame, and segment respectively.

It would be good to explain a GUI development on an example, so please assume we want to create a GUI like Fig. 21.

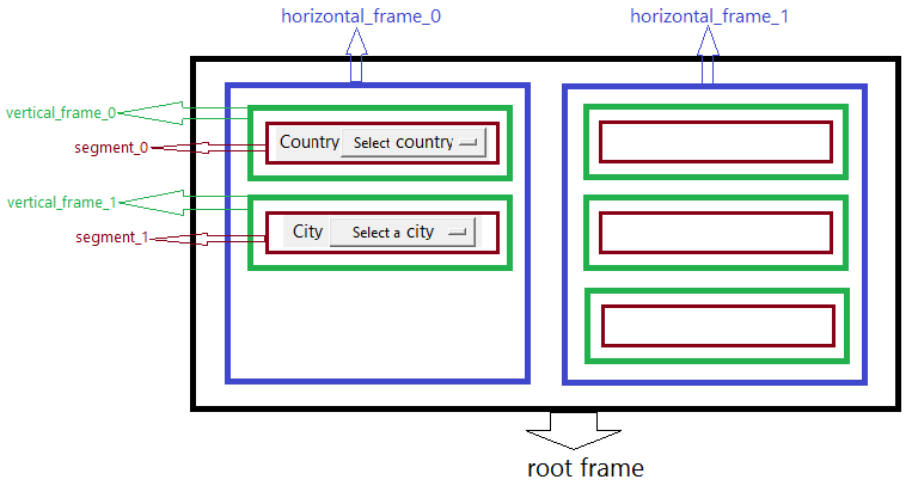

Fig. 21. A sample GUI with 2 horizontal frames. Horizontal frame 0 has two vertical frames. Vertical frame 0 has a segment with 2 *Tkinter* widgets (a label and an options menu) and so on.

A GUI developer has to create a folder, e.g. *my_gui*, and then is obliged to do two tasks:

1. Develop segments:
   1.1. If it is assumed that we want to create a GUI like Fig. 21, then due to that there are two horizontal frames, so we have to create two folders inside the *my_gui* folder: *horizontal_frame_0* and *horizontal_frame_1*. (Each folder represents a horizontal frame). The name of these two folders should be in this format: *y_x* which *y* refers to a favorite name (we choose *horizontal_frame*) and x indicates the index of the horizontal frame which is started from 0.
   1.2. For each segment in the horizontal frame, we have to make a segment module, e.g. since the horizontal_frame_0 has two segments, we have to create two modules: *segment_0.py*, and *segment_1.py*. The name of these two folders should be in this format: *y_x* which *y* refers to a favorite name (we choose "*segment*") and x indicates the index of the segment which is started from 0.
   1.3. We have to develop each segment, e.g. if we want to develop *segment_0.py* we have to create a segment_0 class (please note, the name of the module and the class should be the same). *segment_0* class should inherit *AbstractGUISegment* class, see Fig. 22. The abstract class *AbstractGUISegment* has an abstract method *get_widget* that should be implemented. This abstract method should return a tuple of *Tkinter* widgets. As it is shown in Fig. 21, *segment_0* has two widgets: a tkinter Label and a *Tkinter OptionMenu* (a drop-down menu).

```python
from GUI.AbstractGUISegment import AbstractGUISegment
import tkinter as tk
from controller import Controller


class Segment_0(AbstractGUISegment):

    def get_widget(self):
        country_list = ["Iran", "Japan"]
        var_tuple = self.get_special_segment_var_tuple(0, 0)
        var_tuple[0].set('Select a Country')
        optionMenu_country = tk.OptionMenu(self.get_frame(), var_tuple[0], *country_list)
        optionMenu_country.configure(width=25)
        label = tk.Label(master=self.get_frame(), text='Country')

        return label, optionMenu_country
```

Fig. 22. *Segment_0* has two widgets: *Label* and *OptionMenu* which are returned by the *get_widget* method as a tuple.

In line 9 a list of countries is defined which will be shown in *OptionMenu* (drop-down menu). In the *Tkinter* library, some widgets like *OptionMenu* needs a *StringVar* variable. To reach a *StringVar* variable you can call *get_special_segment_var_tuple(self, horizontal_frame_index: int, segment_index: int)* method. This method has two parameters: the *horizontal_frame_index* parameter which determines the index of a horizontal frame which is 0 in the Fig. 21 example, and the *segment_index* parameter which determines the index of a segment, which is 0 in the Fig. 21 example. This method returns a tuple of *StringVars*. The number of this tuple is determined in the *configurations.py* module which is set to 5 (NUMBER_OF_STRINGVAR = 5), if you need more *StringVar* for each segment, you can increase this number.

2. Create a module in NegoSim/GUI path, e.g. test_gui.py, see Fig. 22:

```python
from GUI.AbstractGUISegment import AbstractGUISegment
import tkinter as tk
from configurations import *
from GUI.gui import GUIsegments


class test_gui:
    def __init__(self, window):
        gui_segments = GUIsegments(window=window, segments_path='../my_gui')
        gui_segments.create_sessionGUI()


if __name__ == '__main__':
    root = tk.Tk()
    root.title('New Session')
    test_gui(root)
    root.mainloop()
```

Fig. 23. This module creates a GUI using GUI segments that exist in the '../my_gui' path.

If you run this module a GUI like Fig. 21 would be created. In NegoSim, *sessionGUI2.py* and *tournamentGUI.py* GUI modules were developed in this way.

Please Note: every method that you need to develop a GUI, exists in *AbstractGUISegment* class in path *NegoSim/GUI*.

9 Run NegoSim

There are two ways to run NegoSim (run a negotiation session or tournament):

1. Using GUI
2. Using NegoSim as a library

9.1 Run NegoSim using GUI

In order to run NegoSim using GUI, you can simply run module *view.py* (in path *NegoSim/GUI*), then you can see a window that shows some information about NegoSim (see Fig. 24).

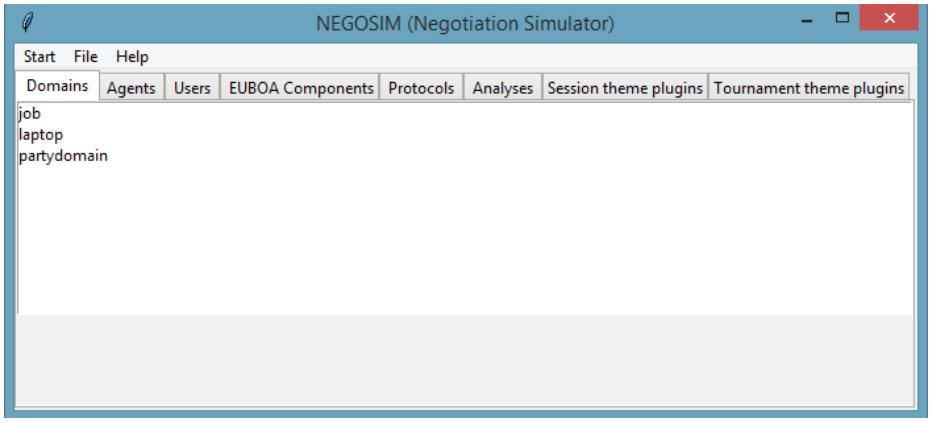

Fig. 24. This is the main window of NegoSim that show some information about NegoSim and its entity.

If you click on *start* you have two choices: (1) Negotiation, (2) Tournament (see Fig. 25).

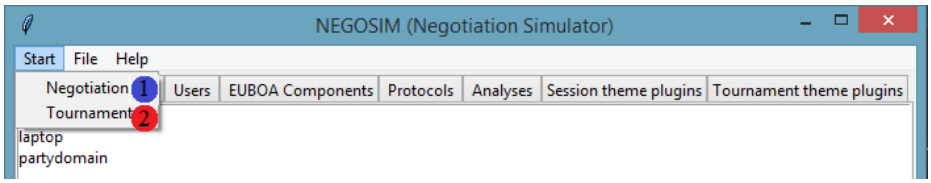

Fig. 25. *Start* menu of main GUI of NegoSim.

Run a negotiation session:

If you click on Negotiation (Number 1 in Fig. 25), you can run a negotiation session (see Fig. 26).

(a) (b)

Fig. 26. The Negotiation session window is shown in (a) and (b). There are 11 widgets. The negotiation settings are determined by them. When a domain is selected (#1 in (b)), the related preference drop-down widget is activated (#5 in (b)).

As it is shown in Fig. 26, the negotiation session window has 11 widgets in which the user can determine the settings of the negotiation session. The widgets of the negotiation session window are explained in table 8:

Table 8. Description of widgets of the negotiation session window.

Widget number	description
#1	You can choose a protocol for the negotiation session
#2	You can choose a domain for the negotiation session (e.g. Job domain)
#3	You have to choose an analysis entity for the negotiation session. This entity will monitor the negotiation session and analyze the negotiation session result and the agent's performance.
#4	You can select a party for the negotiation session.
#5	You can choose a preference for the agent using this widget.
#6	You can select a utility space for the agent. E.g. you can choose '*additive_utility_space*' for the agent. NegoSim lets the agents negotiate against each other with different utility spaces.
#7	You can add first and second negotiation agents to the #8 list box widget using this button. After adding an agent you can change the values of #4, #5, and #6 and add another agent to the #8 list box. These two agents will negotiate against each other.
#8	This list box widget shows the negotiation agents and their preferences and their utility spaces.
#9	You can use this button if you want to delete the current agent and select another agent.
#10	You have to determine a deadline for the negotiation session.
#11	You can start the negotiation session using this button.

After clicking on the Start Negotiation button (#11), the negotiation will be started. After finishing the negotiation session the result window will be shown as Fig. 27 (black rectangle).

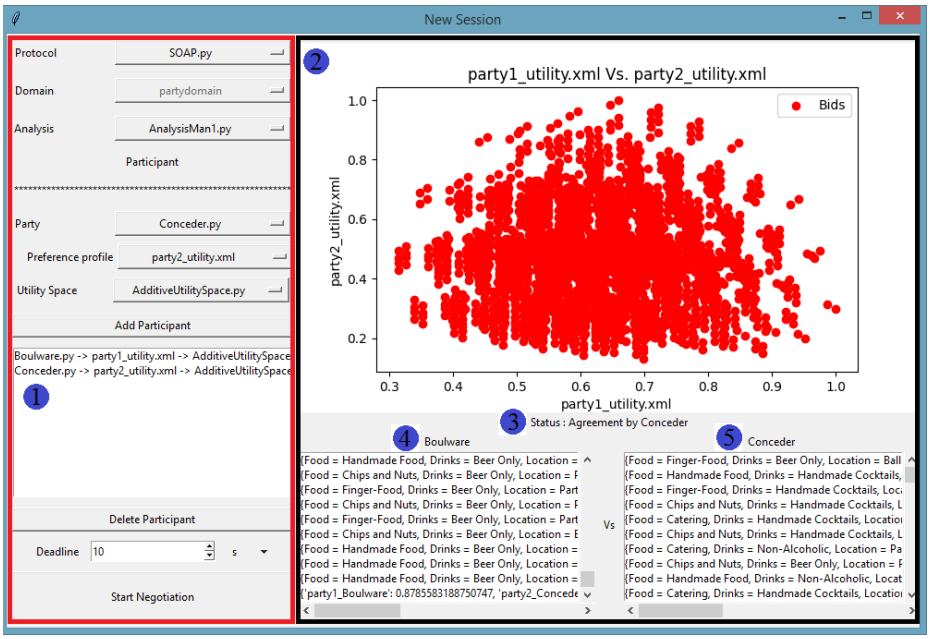

Fig. 27. There are red and black rectangles. The red rectangle shows the settings of the negotiation session and the black rectangle shows the results of the negotiation session.

All exchanged offers, and results of negotiation are shown in the console, but if you want to save all details, you need to make your particular session and tournament analysis entity, see section 7.

In Fig. 27 red rectangle shows the settings of the negotiation session. In this example *SOAP*, *partydomain*, and *AnalysisMan1* were selected as protocol, domain, and analysis entity, respectively, and the deadline was valued at 10 seconds.

The list box widget (#1) shows two agents: 1) *Boulware* party with *party1_utility.xml* as preference and *AdditiveUtilitySpace* as its utility space. 2) *Conceder* party with *party2_utility.xml* as preference and *AdditiveUtilitySpace* as its utility space.

In Fig. 27 black rectangle shows all possible outcomes as a diagram (#2), the result of the negotiation session (#3) (in this example the result of the negotiation is agreement), Boulware's offers (#4), and Conceder's offers (#5). The last row of the list box widget in #4 and #5 is shown the utility and social welfare.

In this way of running a negotiation session, the analysis data is saved in *NegoSim/GUI/logs*.

Run a tournament:

If you click on Tournament (Number 2 in Fig. 25), you can run a tournament (see Fig. 28).

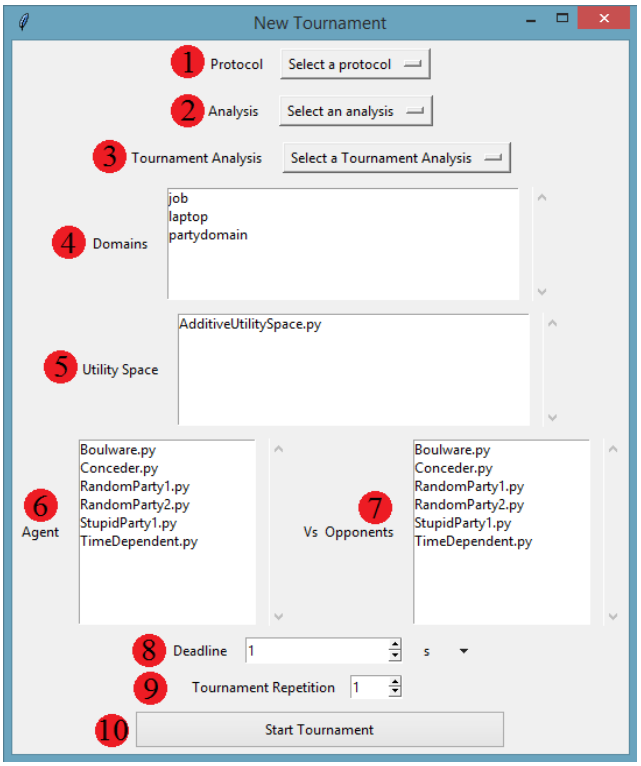

Fig. 28. A tournament GUI window.

As it is shown in Fig. 28, the tournament window has 10 widgets where the user can define a setting for the tournament. Table 9 is detailed all widgets.

Table 9. Description of widgets of Tournament GUI.

Widget number	description
#1	You can choose a protocol for negotiation sessions of the tournament using this widget, e.g. *SOAP* protocol.
#2	You have to select an analysis entity for each negotiation session of the tournament, e.g. Analysis_man0.
#3	You should choose a tournament analysis entity to analyze the tournament, e.g. *AnalysisTournament1*.
#4	You can choose one or more domains for the tournament. NegoSim iterates over all preferences of all domains.
#5	You can choose one or more utility spaces for agents in a tournament. For example, if you choose two different utility spaces, NegoSim iterates over all selected utility spaces and assigns these utility spaces to agents.
#6	You have to choose one or more agents using this widget. The agents that the user selected will negotiate against opponents.

	NegoSim analyzes the results and performance of these agents.
#7	You have to choose one or more opponents using this widget. The agents that the user selected in #6 will negotiate against opponents.
#8	You can choose a deadline for the negotiation sessions of the tournament.
#9	If you want, your tournament to be repeated, you can set this widget to more than one.
#10	You can start the negotiation using the Start Tournament button.

After clicking on the Start Tournament button, the tournament is started. After finishing the tournament the tournament GUI window will be like Fig. 29.

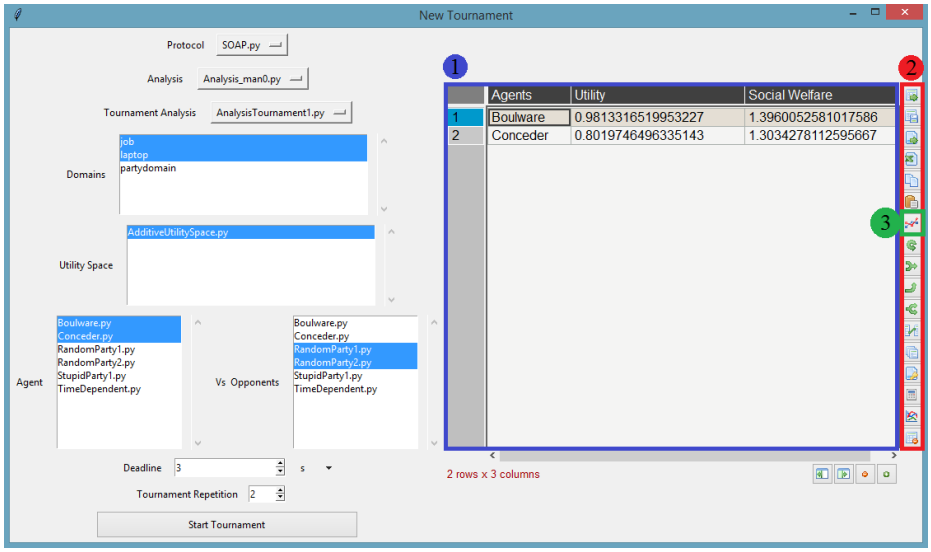

Fig. 29. The results of the tournament are shown as a table.

Fig. 29 shows a tournament in which *SOAP*, *Analysis_man0*, and *AnalysisTournament1* were selected as protocol, analysis entity for negotiation sessions, and analysis entity for the Tournament, respectively. Job and laptop were selected as domains of the tournament which NegoSim iterated over all preferences of these two domains. The only utility space that is selected for the negotiation parties is *AdditiveUtilitySpace*, so NegoSim assigns this utility space to all negotiation parties. Boulware and Conceder were selected as the agents who negotiated against RandomParty1 and RandomPary2 as opponents. NegoSim investigates the results and performance of the agents (in this example Boulware and Conceder). The results of Boulware and Conceder are shown in the blue rectangle (#1). NegoSim uses *pandastable* library [5] to show the results that have been prepared by the tournament analysis entity. On the right side of the result window of the tournament GUI, there is

a toolbox (red rectangle #2) that provides some facilities. E.g. the user can make a chart using the tool that is signed with the green square (#3), see Fig. 30.

A tournament is consisting of several negotiation sessions, and the analysis data for each negotiation session is stored in path NegoSim/GUI/logs. The analysis data of a tournament are stored in path NegoSim/GUI/TournamentLogs.

All exchanged offers, results of negotiation, and tournament are shown in the console, but if you want to save all details, you need to make your particular session and tournament analysis entity, see section 7.

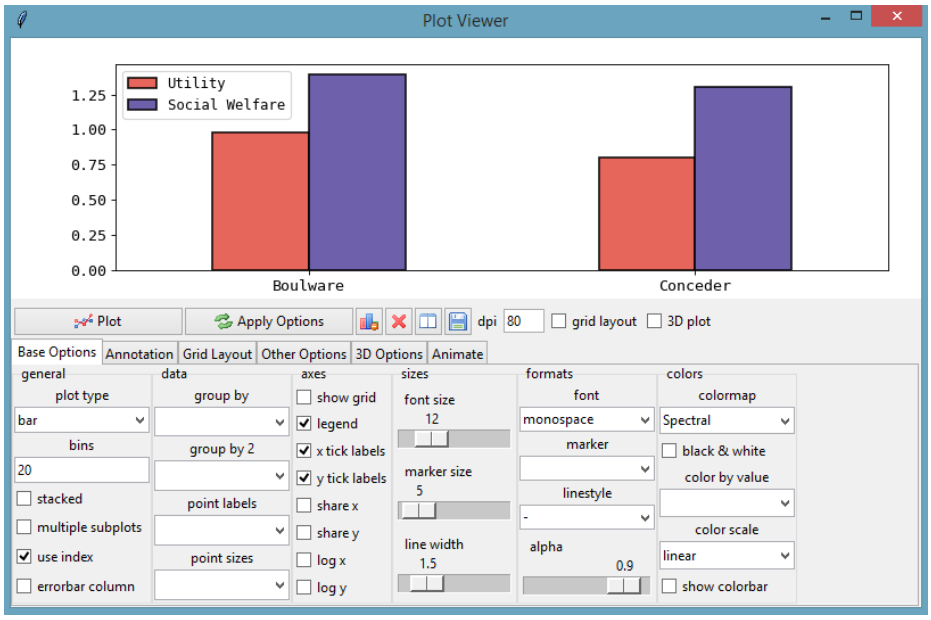

Fig. 30. The results of the tournament are shown as a bar chart.

9.2 Run NegoSim as a library

Run a negotiation session:

To use NegoSim as a library you can make a folder inside of NegoSim folder and then create a python module (e.g. see *NegoSim/Run/run_session.py*) and write a piece of code like Fig. 31:

```
1  from core.BilateralSession import BilateralSession
2
3
4  bilateral_session = BilateralSession(protocol_name='SOAP',
5                                       analysis_man_name='Analysis_man0',
6                                       deadline='5',
7                                       deadline_type='s',
8                                       first_preference='Jobs_util1.xml',
9                                       second_preference='Jobs_util2.xml',
10                                      party1_name='Boulware',
11                                      party2_name='Conceder',
12                                      domain_name='Job',
13                                      utility_space_name1='AdditiveUtilitySpace',
14                                      utility_space_name2='AdditiveUtilitySpace')
15 bilateral_session.start_session()
```

Fig. 31. A module that uses NegoSim as library and run a negotiation session. first_preference and second_preference was defined as a string.

You need to import *BilateralSession* class from *core.BilateralSession* (line 1 in Fig. 31). In line 4, an instance of *BilateralSession* is created. The *init* method of *BilateralSession* class has 11 arguments which are detailed in table 10:

Table 10. Arguments of class *BilateralSession* and their descriptions.

Argument	description
protocol_name	In this argument the protocol name that you want to run the negotiation under its rules should be determined. This variable should be a string, e.g. if we want use the SOAP protocol, then we have to send 'SOAP' as protocol_name. If we send 'SOAP' as protocol_name, it means there is a module with name *SOAP.py* in path *core/protocols*, line 4 in Fig. 31.
analysis_man_name	In this argument the analysis entity that you want to analyze the negotiation with it should be determined. You can choose a module from path *core.analysis*. For example if you want to choose module Analysis_man0.py then you should send string ' Analysis_man0', line 5 in Fig. 31.
deadline	In this argument you have to send a string as deadline of the negotiation, e.g. if you want your negotiation deadline would be 5 second/millisecond then send string '5' (see line 6 in Fig. 31).
deadline_type	This argument could receive strings 's' or 'm' as second or millisecond respectively, line 7 in Fig. 31.
first_preference	First preference of a domain for the first agent. This argument could receive a string or an object: a) A string: You can choose a preference that there is in

	path *core.Domains*, line 8 in Fig. 31. b) An object: you can create a preference object, see line 4 and line 10 in Fig. 32. **Please Note:** both first preference and second preference should belong same domain. In Fig. 31 both first and second preferences belongs Job domain.
second_preference	First preference of a domain for the first agent. This argument could receive a string or an object: a) A string: You can choose a preference that exists in path *core.Domains*, line 9 in Fig. 31. b) An object: you can create a preference object, see line 4 and line 10 in Fig. 32. **Please Note:** both first preference and second preference should belong same domain. In Fig. 24 both first and second preferences belongs Job domain.
party1_name	The first agent's name should be sent here. You should choose an agent from path *NegoSim/Agents*. E.g. if you choose Boulware.py, then you should send it as a string 'Boulware', line 10 in Fig. 31.
party2_name	The second agent's name should be sent here. You should choose an agent from path *NegoSim/Agents*. E.g. if you choose Boulware.py, then you should send it as a string 'Conceder', line 11 in Fig. 31.
domain_name	In here you have to send a domain name which you have selected your preferences from that. In Fig. 31, since the preferences have been selected from Job domain, you have to send it as a string 'Job', line 12 in Fig. 31. Domain names are defined as folder which contains preferences. There are some predefine domains in path *NegoSim/Domains*: Job, laptop.
utility_space_name1	**Please Note:** NegoSim does not force the developer to use the same utility space for both negotiators, so each agent can have their own utility space. This argument receives the utility space for the first agent. You can choose the utility space from path *NegoSim/utility_spaces*. E.g. if you choose AdditiveUtilitySpace.py, you have to send it as a string ' *AdditiveUtilitySpace*'.
utility_space_name2	**Please Note:** NegoSim does not force the developer to use the same utility space for both negotiators, so each agent can have their own utility space. This argument receives the utility space for the second agent. You can choose the utility space from path *NegoSim/utility_spaces*. E.g. if you choose AdditiveUtilitySpace.py, you have to send it as a string '*AdditiveUtilitySpace*'.

Class *BilateralSession* makes a choice for the users in order to send a preference; users can send it as a string like line 8 and line 9 in Fig. 31 or send it as an instance of *Preference* class (line 4 and line 10 in Fig. 32). After creation of an instance of class *BilateralSession*, you should call *start_session* method, line 15 in Fig. 31.

Fig. 32 shows another implementation that uses an instance of preferences.

```
1  from core.BilateralSession import BilateralSession
2  from core.Preference import Preference
3
4  preference1 = Preference('Job', 'Jobs_util1.xml')
5
6  bilateral_session = BilateralSession(protocol_name='SOAP',
7                                       analysis_man_name='Analysis_man0',
8                                       deadline='5',
9                                       deadline_type='s',
10                                      first_preference=preference1,
11                                      second_preference='Jobs_util2.xml',
12                                      party1_name='Boulware',
13                                      party2_name='Conceder',
14                                      domain_name='Job',
15                                      utility_space_name1='AdditiveUtilitySpace',
16                                      utility_space_name2='AdditiveUtilitySpace')
17 bilateral_session.start_session()
```

Fig. 32. A module that uses NegoSim as a library and runs a negotiation session. first_preference and second_preference were defined as an instance of class Preference and a string, respectively.

In Fig. 32, line 2, class *Preference* was imported from *core.Preference*. In line 4, an instance of class *Preference* was created (section 4 detailed how you can create an instance of Preference in different ways). In line 10, the created instance of Preference was sent as the *first_preference* argument. Now, you can run this module.

In this way of run negotiation session, the analysis data are saved at *NegoSim/Run/logs*.

All exchanged offers, results of negotiation are shown in the console, but if you want to save all details, you need to make your particular session and tournament analysis entity, see section 7.

Run a negotiation tournament

To use NegoSim as a library you can make a folder inside of NegoSim folder and then create a python module (e.g. see *NegoSim/Run/ run_tournament.py*) and write a piece of code like Fig. 33:

```
1  from core.BilateralTournament import BilateralTournament
2
3  bilateral_tournament = BilateralTournament(protocol_name='SOAP',
4                                             analysis_man_name='Analysis_man0',
5                                             Tournament_analysis_name='AnalysisTournament1',
6                                             deadline='3',
7                                             deadline_type='s',
8                                             agent_names=['Boulware'],
9                                             opponent_names=['Conceder'],
10                                            domain_names=['laptop'],
11                                            tournament_repetition='1',
12                                            utility_space_names=['AdditiveUtilitySpace'])
13 bilateral_tournament.start_tournament()
```

Fig. 33. A module that uses NegoSim as a library and runs a Tournament.

In Fig. 33, line 1 class *BilateralTournament* was imported from *core.BilateralTournament*. In line 3, an instance of class *BilateralTournament* was created. Class *BilateralTournament* has 10 arguments which are detailed in table 11.

Table 11. Arguments of *BilateralTournament* and their description.

Method	Description
protocol_name	In this argument the protocol name that you want to run the negotiation under its rules should be determined. This variable should be a string, e.g. if we want use the SOAP protocol, then we have to send 'SOAP' as protocol_name. If we send 'SOAP' as protocol_name, it means there is a module with name *SOAP.py* in path *core/protocols*, line 3 in Fig. 33.
analysis_man_name	In this argument the analysis entity that you want analyze the negotiation session with it should be determined. You can choose a module from path *core.analysis*. For example, if you want to choose module Analysis_man0.py then you should send string ' Analysis_man0', line 4 in Fig. 33.
tournament_analysis_name	This argument receives tournament analysis entity which was defined in path *NegoSim\AnalysisTournament*. E.g. if you want to select module *AnalysisTournament1.py*, you have to send it as string ' AnalysisTournament1', line 5 in Fig. 33.
deadline	In this argument you have to send a string as deadline of the negotiation, e.g. if you want your negotiation deadline would be 3 second/millisecond then send string '3' (see line 6 in Fig. 33).
deadline_type	This argument could receive strings 's' or 'm' as second or millisecond respectively, line 7 in Fig. 33.

agent_names	This argument receives a list of agents that should negotiate against opponent agents, line 8 in Fig. 33. NegoSim analyzes the result and performance of these agents.
opponent_names	This argument receives a list of agents as opponents that should negotiate against agents, line 9 in Fig. 33.
domain_names	This argument receives a list of domain names. NegoSim iterates over all preferences of these domains, line 10, Fig. 33.
tournament_repetition	This argument receives a string that determines how many times the tournament should be repeated. E.g. if you want the tournament would be repeated 2 times, you should send it a string '2'. NegoSim averages on the analysis data.
utility_space_names	**Please Note:** NegoSim does not force the developer to use the same utility space for both negotiators, so each agent can have their own utility space. This argument receives a list of utility spaces. You can choose the utility space from path *NegoSim/utility_spaces*. E.g. if you choose AdditiveUtilitySpace.py, you have to send it as a string '*AdditiveUtilitySpace*'. The class *BilateralTournament* iterates over list of utility spaces.

After the creation of an instance of *BilateralTournament*, you have to call the *start_tournament* method, line 13 in Fig. 33. Now, you can run this module.

In this way of running a tournament, the analysis data, for sessions are saved at *NegoSim/Run/logs* and for tournament are saved at *NegoSim/Run/TournamentLogs*.

All exchanged offers, results of negotiation and tournament are shown in the console, but if you want to save all details, you need to make your particular session and tournament analysis entity, see section 7.

References

1. Aydoğan, Reyhan, et al. "Alternating offers protocols for multilateral negotiation." Modern approaches to agent-based complex automated negotiation. Springer, Cham, 2017. 153-167.
2. Baarslag, Tim. Exploring the strategy space of negotiating agents: A framework for bidding, learning and accepting in automated negotiation. Springer, 2016.
3. Baarslag, Tim, Koen Hindriks, and Catholijn Jonker. "Acceptance conditions in automated negotiation." Complex Automated Negotiations: Theories, Models, and Software Competitions. Springer, Berlin, Heidelberg, 2013. 95-111.

4. Tunalı, Okan, Reyhan Aydoğan, and Victor Sanchez-Anguix. "Rethinking frequency opponent modeling in automated negotiation." International Conference on Principles and Practice of Multi-Agent Systems. Springer, Cham, 2017.
5. pandastable, https://pandastable.readthedocs.io/en/latest/.