Guide to Writing a Negotiation Agent

W.Pasman, 30 oct 2007

Introduction

For second half of the course AI Techniques (IN4010TU) you have to implement an agent that will negotiate in your behalf. This document describes how you can install the required environment, work with the provided agents, and write, compile, and run such an agent yourself.

Installing the Environment

The negotiation environment has been tested extensively on Mac OSX 10.9 and on Windows XP.
It should run on any machine running java 1.5.0 or higher [sun07]. This includes Solaris, Linux, and more recent versions of Mac OSX and Microsoft Windows.
To install the environment, the file `negotiator.zip` can be downloaded from Blackboard. It contains the following files:

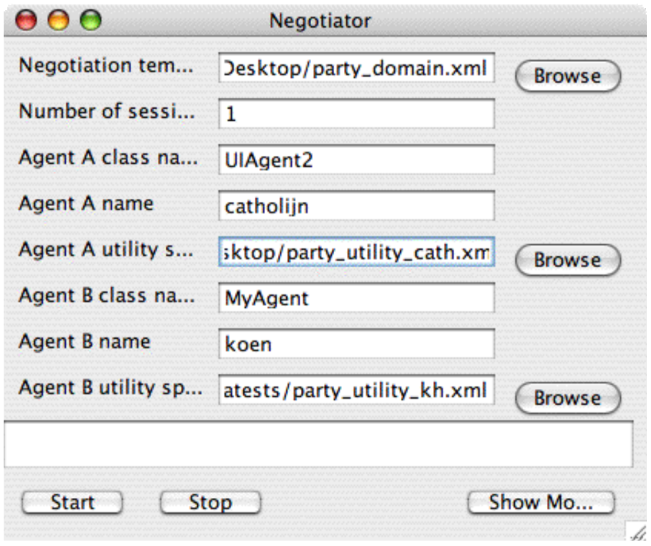
CHECKTHIS-PRELIMINARY

- **negotiation.pdf**, containing a broad discussion of this assignment
- **guide.pdf**, containing this document
- **negotiator.jar**, containing the negotiator environment
- **editor.jar**, containing the domain- and utility space editor
- templates, containing the inherit and party domain space and a few sample utility spaces.
- The **SimpleAgent.java** and **SimpleAgent.class** file
- The **BayesianAgent.class** file

In case you need more information about JAVA programming, please use the following link: http://java.sun.com/docs/books/tutorial/index.html. The Java API definitions can be found on http://java.sun.com/j2se/1.5.0/docs/api/index.html.

Running a Negotiation

To run a negotiation, double click the negotiator.jar file. The set-up screen pops up (Figure 1).


Negotiator

Negotiation tem... Desktop/party_domain.xml

Number of sessi... 1

Agent A class na... UIAgent2

Agent A name catholijn

Agent A utility s... sktop/party_utility_cath.xm

Agent B class na... MyAgent

Agent B name koen

Agent B utility sp... atests/party_utility_kh.xml

Figure 1. The negotiation set-up screen.

You need to set up the following items:

- The negotiation template. This is an xml file holding a decription of the domain, plus the default values for all other fields in the set-up screen. You can edit the xml file by hand to change the default values. To run a demo, select the party_domain.xml file using the Browse button.
- Number of sessions: the number of negotiation sessions that you want to run.
- Agent A/B class name: the name of the negotiation agent that party A/B in the negotiation wants to use. You can use the provided agents negotiator.agents.UIAgent2, negotiator.agents.SimpleAgent2, and negotiator.agents.BayesianAgent, or use your own agent. UIAgent2 allows you to manually control party A/B in the negotiation. SimpleAgent2 is an agent that automatically handles the negotiation for party A/B. BayesianAgent also automatically handles the negotiation, but uses a more advanced algorithm that tries to estimate the opponent's utility space.
- Agent A/B name: the plain name of the agent, used mainly when printing information about the progress and asking input.
- Agent A/B utility space: the xml file containing the utility space of agent A/B. Usually this is a file with a name like party_utility_A/B.xml.

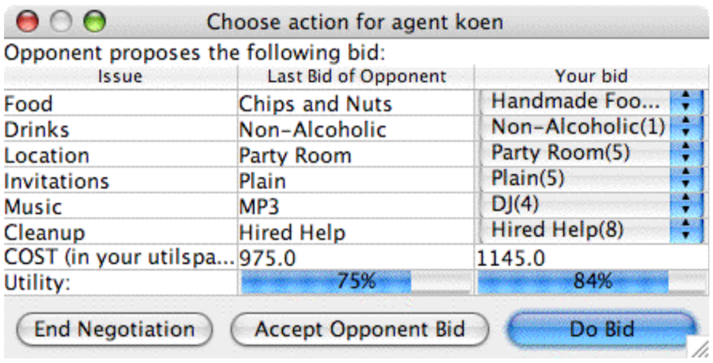
The text field just above the start button will show real-time feedback on the ongoing negotiation.
After these fields have been set appropriately, you can press the Start button to run the negotiation sessions. Note, if you press "Show Model", you will end up in the utility space editor, editing the utility space of Agent A as discussed in XXXXX.

Using an Automatic Agent

If your selected an automatic negotiation agent, there will not appear any window while that agent has the turn. The agent should pose its bid within a reasonable time. After the agent made its bid, the other agent is given the turn. If both agents are automatic, no windows will appear at all during the entire negotiation, and only the output windows will show the ongoing negotiation results.

Using the UIAgent2

If you selected the UIAgent2 for Agent A/B class name, the window as shown in Figure 2 will pop up every time input from negotiation agent A/B is needed.


Choose action for agent koen

Opponent proposes the following bid:

Issue	Last Bid of Opponent	Your bid
Food	Chips and Nuts	Handmade Foo...
Drinks	Non-Alcoholic	Non-Alcoholic(1)
Location	Party Room	Party Room(5)
Invitations	Plain	Plain(5)
Music	MP3	DJ(4)
Cleanup	Hired Help	Hired Help(8)
COST (in your utilspa...	975.0	1145.0
Utility:	75%	84%

Figure 2. The GUI of UIAgent2.

This GUI has three main components: the text field at the top level, the table showing the last bid and a possible next bid, and a row of buttons at the bottom.
The text field shows some text about the current negotiation state.
The table has three columns:

- The left column shows the names of the issues in the domain
- The center column shows the evaluation values for the issues as proposed in the last bid of the opponent (or "-" if this is the first round)
- The right column shows the current picked values for the issues. You can edit the current pick by clicking on the fields, which will open the combo boxes in the fields (Figure 3).

Figure 3. Combo box opens after clicking on a field, allowing change of the picked values for the next bid.

The last two rows of the table show the cost and utility of the last opponent's bid and your current bid. The cost field will turn red if you exceed the maximum cost of € 1200. The utility is shown as a percentage, and also as a bar of matching size. These values or course are computed according to your utility space, as that determines your score, and also because you have no access to the opponent's utility space.
The lower three buttons allow you to submit the next bid as set in the right column, accept the opponent's last bid.

About the Negotiation

This section discusses the various details concerning a negotiation: the protocol, the time restrictions, and the results display.

Actions

In an negotiation, the two parties take turns in doing the next negotiation action. An action are objects in the negotiator.actions directory, and can be one of the following:

- Accept(Agent agent). This action indicates that agent (which usually has the value 'this') accepts the opponent's last bid
- Offer(Agent agent,Bid bid). This action indicates that the agent proposes a new bid.
- EndNegotiation(Agent agent). This action indicates that the agent terminates the entire negotiation, resulting in the lowest possible score of zero for both agents.

Negotiation Protocol

The negotiation protocol determines the overall order of actions during a negotiation. Agents are obliged to stick to this protocol, and deviations from the protocol are caught and penaltized. This section discusses the details of the protocol for this assignment.

Agent A and B take turns in the negotiation. One of the two agents is picked at random to start. When it is the turn of agent X (X being A or B), the method agentX.ReceiveMessage(action) is called, where action is the previous action that was done by the opponent, followed by a call to agentX.chooseAction() which is supposed to return the next action of agent X.

If the action returned is an Offer, the turn taking goes into the next round.

In all other cases, the turn taking stops and the final score (utility of the last bid) is determined for each of the agents, as follows:

- the action returned is an Accept. This action is possible only if the opponent actually did a bid. The last bid of the opponent is taken, and the utility of that bid is determined in the utility spaces of agent A and B. The opponent is informed of this accept via the ReceiveMessage function (but now without the subsequent chooseAction).
- the action returned is an EndNegotiation. The score of **both** agents is set to 0.

So far for the protocol. If an agent does not follow this protocol, for instance by sending another action that is not one of the above or by crashing, that agent will get a utility of 0, and the opponent will be given the utility of his last bid (or 1 if he did not yet do a bid). If the agent deviates grossly from the protocol in such a way that it endangers the negotiator simulator itself, it may be disqualified entirely.

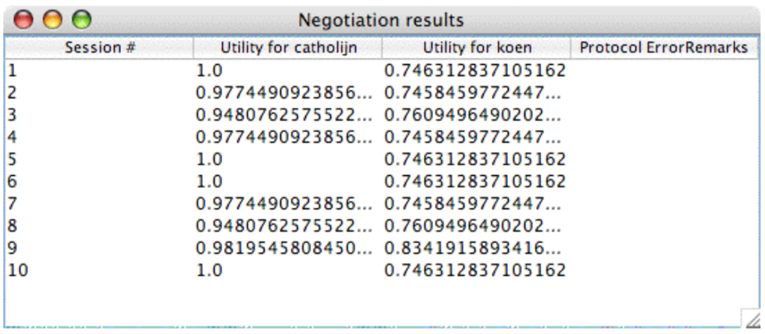
Time available for a Session

Normally, each negotiation session is allowed to last at most 120 seconds. After this time, the negotiation will be terminated by killing the negotiation agents, and the utility of both parties will be 0. Only if one of the agents is a GUI agent requiring user input, the deadline is set to 1200 seconds.

Notice that manipulation of the available time (by delaying the response of your agent) can be an important factor in the negotiation results, and one improvement for the example agent would be to be more careful about this.

Negotiation Results

The results of the negotiation are shown in the Negotiation Results screen (Figure 4). This screen shows four columns: the session number, the final utility for agent A and B, and the Protol ErrorRemarks column. The utilities are the utilities of the final, accepted bid, as measured in the utility spaces of agent A and B (see the section Utility of a Bid). The Protocol ErrorRemarks button shows extra information in case one of the agents ignored the negotiation protocol, causing an non-standard scoring of the result (see the section Negotiation Protocol).



Negotiation results

Session #	Utility for catholijn	Utility for koen	Protocol ErrorRemarks
1	1.0	0.746312837105162	
2	0.9774490923856...	0.7458459772447...	
3	0.9480762575522...	0.7609496490202...	
4	0.9774490923856...	0.7458459772447...	
5	1.0	0.746312837105162	
6	1.0	0.746312837105162	
7	0.9774490923856...	0.7458459772447...	
8	0.9480762575522...	0.7609496490202...	
9	0.9819545808450...	0.8341915893416...	
10	1.0	0.746312837105162	

Figure 4. Example of Negotiation results screen. Here Agent A has the name 'catholijn', and Agent B 'koen'.

Utility of a Bid

A bid is a set of chosen values $v_1 \ldots v_N$ for each of the N issues. Each of these values has been assigned an evaluation value eval(v_i) by you, using the utility space editor, and also there were fixed costs cost(v_i) associated with each value.
The utility of a bid is (currently) computed by UtilitySpace.getUtility. The utility is the weighted sum of the normalized evaluation values, under the assumption that the cost is below the maximum cost of 1200. If the maximum cost is exceeded, the utility is zero.

$$
\begin{aligned}
Utility(v_1 ... v_N) &= \begin{cases} U(v_1 ... v_N), \ if \ \ CostSum(v_1 ... v_N) < 1200 \\ 0, \ if \ \ Cost(v_1 ... v_N) > 1200 \end{cases} \\
U(v_1 ... v_N) &= \sum_{i=1}^{N} w_i \frac{eval(v_i)}{\max(eval(v_k))} \qquad (1) \\
CostSum(v_1 ... v_N) &= \sum_{i=1}^{N} \mathrm{cost}(v_i)
\end{aligned}
$$

Thus, the utility function is more or less fixed, you can only adjust the weight values. This is done this way to allow a comparison of the results of the final negotiation results.

Writing a Negotiation Agent

This section discusses how you create an automatic negotiation agent. To explain explain this, the SimpleAgent.java code as provided in the installation package will be discussed.

In the first place, a negotiation agent has to extend the negotiator.agents.Agent class. By extending this class, your agent gets access to the following fields:

- utilitySpace, an instance of the UtilitySpace that you specified in the negotiation set-up screen (Figure 1).
- startTime, an instance of Date specifying the time at which the negotiation started
- totalTime, an Integer specifying the number of seconds in which this negotiation session has to be completed.

Furthermore you have access to the function getName() that will return a String containing the name of your agent.

To implement your agent, you have to override only two classes:

- public void **ReceiveMessage**(Action opponentAction)
- public Action **chooseAction**()

ReceiveMessage

The ReceiveMessage(Action opponentAction) informs you that the opponent just did the opponentAction. The opponentAction may be null if you are the first to place a bid, or an Offer containing the bid of the opponent. It may also be an Accept or StopNegotiation action.

The chooseAction() asks you to return an Action to make the next step in the negotiation.

In the SimpleAgent2 code, the following code is available for ReceiveMessage (Figure 5). This will be the typical code for automatic negotiation agents.

```
        public void ReceiveMessage(Action opponentAction) {
          actionOfPartner = opponentAction;
    }
```

Figure 5. Example code for ReceiveMessage

ChooseAction

Figure 6 shows the example code for the chooseAction method. For safety, all code was wrapped in a try-catch block, because if our code would accidentally contain a bug we still want to return a good action (failure to do so is a protocol error – see Negotiation Protocol – and results in a score of 0 for us!).

The sample code works as follows. If we are the first to place a bid, we place a random bid with sufficient utility (see the .java file for the gory details on that). Else, we determine the probability to accept the bid, depending on the utility of the offered bid and the remaining time. Finally, we randomly accept or pose a new random bid.

```
public Action chooseAction()
    {
        Action action = null;
        try {
            if(actionOfPartner==null)
                  action = chooseRandomBidAction();
            if(actionOfPartner instanceof  Offer)
            {
                Bid partnerBid = ((Offer)actionOfPartner).getBid();
                double offeredutil=
                  utilitySpace.getUtility(partnerBid);
                double time=((new Date()).getTime()-
                  startTime.getTime())/(1000.*totalTime);
                double P=Paccept(offeredutil,time);
                if (P>Math.random()) action = new Accept(this);
                else action = chooseRandomBidAction();
            }
            Thread.sleep(1000); // just for fun
        } catch (Exception e) {
            System.out.println("Exception in
                  ChooseAction:"+e.getMessage());
            action=new Accept(this);
        }
        return action;
    }
```

Figure 6. Example code for chooseAction

The Paccept function is

$$P_{accept} = \frac{u - 2ut + 2\left(t - 1 + \sqrt{(t-1)^2 + u(2t-1)}\right)}{2t-1} \qquad (2)$$

Where u is the utility of the bid made by the opponent (as measured in our utility space), and t is the current time as a fraction of the total available time.
Figure 7 shows how this function behaves depending on the utility and remaining time. It can take quite some time to figure out a formula that suits the requirements, but it is at the heart of the example agent.

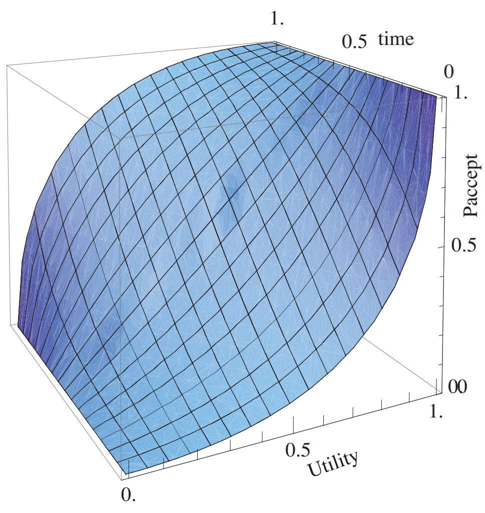


Figure 7. Paccept value as function of the utility and time (as a fraction of the total available time)

The UtilitySpace and its functions should be considered as 'given', XXXXX mantis issue.

You may want to override the init function, to catch the sessionNumber and sessionTotalNumber. See the MyAgent2.java file.

Your agent has to negotiate on its own, and is not allowed to communicate with a human user Therefore, do not override the isUIAgent function in automatic negotiation agents:.

Compiling an Agent

To compile the agent, you put YourAgent.java code in the directory containing the negotiator.jar file, and use the command line

```
    javac -cp negotiator.jar YourAgent.java
```

The resulting YourAgent.class ifle then be loaded into the negotiator simulator by specifying the class path to YourAgent.

Data structures

This section discusses the datastructures that are relevant when writing a negotiation agent. Only the functions that fetch information out of the available datastructures will be discussed, as modification of these datastructures seems not very useful in this context.

Short Introduction

The central datastructures in the negotiation are the bid and the utilityspace. Both work in a domain.
Figure 8 shows an overview of the Domain and utility space datastructures and their relations.

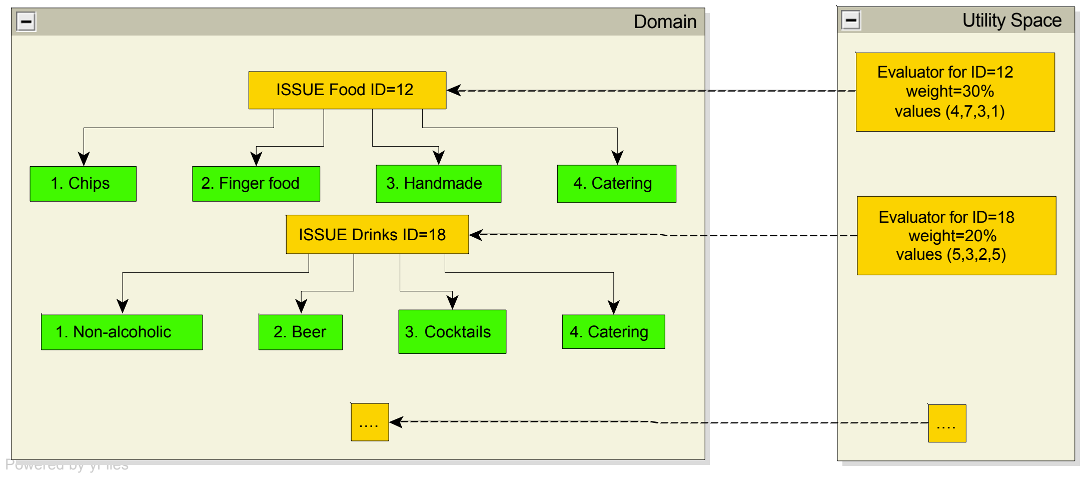


Figure 8. Overview of the datastructures and relations.

The **domain** describes which issues are the subject of the negotiation. To give a concrete example of a domain, in the party domain the domain is the list of issues, where the **issues** are food, drink, location, invitations, music and cleanup. Issues are being referred to via the **issue ID**, a unique number for each issue. The domain description also describes the possible **values** that all the issues can take. In the party domain, all issues can have only discrete values, e.g. the cleanup issue can have the discrete values 'water and soap', 'specialized materials', 'special equipment' and 'hired help'. Therefore these issues are an instance of IssueDiscrete. There are other variants of the Issue but discussion of them falls out of the scope of this short discussion.
Issues are an instantiation of a more general **Objective** class. The objective class itself is not relevant except that some functions return an Objective, and the returned object then has to be cast to an Issue or IssueDiscrete as needed.
The **utilityspace** provides all the information enabling the computation of the utility of some bid. It is implemented as a list of evaluators, one evaluator for every issue in the domain. Because the issues in the party domain are all discrete issues, the evaluators are all EvaluatorDiscrete objects.
The **Evaluator** object contains a weight, and for each value that the issue can have it gives an evaluation-value and a cost value.
The **weight** indicates the relative importance of an issue. The sum of the weights of all issues is 1.0.

The **evaluation-value** gives the evaluation, or utility, associated with each value that an issue can take. For instance, for the cleanup issue above, the evaluation values could be 2 for 'water and soap', 5 for 'specialized materials', etc.
The exact formula for computation of the utilities is given in equation 1.
The **bid** is a set of values for each of the issues in the domain.

The following sections discuss the methods of the relevant objects in more detail. For a number of data structures the constructor is not given, as the readily provided objects should be used.

Domain

`Objective getObjective(int ID)` ID is the ID of the objective returns the objective with the given ID.
`Bid getRandomBid()` returns a bid with randomly set values.
`ArrayList<Issue> getIssues()` get all issues as an arraylist.

UtilitySpace

`Evaluator getEvaluator(int index)` index is the IDnumber of the issue. returns an evaluator for the objective or issue.
`double getUtility(Bid bid) throws Exception` computes the utility of a given bid in your utility space. This is in fact the weighted sum of getEvaluation(). Can throw an exception if there is a problem with the computations, for instance if the utility space contains illegal weights or evaluators.
`double getEvaluation(int pIssueIndex, Bid bid) throws Exception` Computes the utility of one of the issues (pIssueIndex) in a bid. This just translates to a call to EvaluatorDiscrete.getEvaluation.
`boolean constraintsViolated(Bid bid)` Check that the constraints are not violated in this bid. returns true if the bid violates the constraints. In the party domain, the only constraint is that getCost(bid)≤1200.
`Bid getMaxUtilityBid() throws Exception` returns a bid with the maximum utility value attainable in this utility space. Throws an exception if there is no bid at all in this utility space
`double getWeight(int issuesID)` returns the weight of the issue with given issueID
`Objective getIssue(int index) {` does exactly the same as Domain.getIssue
`Domain getDomain()` returns the domain
`Double getCost(Bid bid) throws Exception` returns the cost of the given bid. throws if cost can not be computed for some reason.

EvaluatorDiscrete

```
double getWeight()
```
returns the weight for this evaluator, a value between 0 and 1.

```
Integer getValue(ValueDiscrete alternativeP)
```
returns the non-normalized evaluation-value
may return null if the alternativeP is not a value, or when the value has not been set (which should not occur in the party domain)

```
Integer getEvalMax() throws Exception
```
returns the largest evaluation value available.
Throws if there are no evaluation values available.

```
Double getEvaluation(UtilitySpace uspace, Bid ID,
      int index) throws Exception
```
returns the normalized evaluation value ($eval(bid\ for\ issue\ ID)/\max(eval(v_k))$ in equation 1)

```
Double getEvaluation(ValueDiscrete alt) throws Exception
```
returns the normalized evaluation value of evaluationvalue altP
($eval(alt)/\max(eval(v_k))$ in equation 1)

```
Double normalize(Integer EvalValue) throws Exception
```
returns $EvalValue/\max(eval(v_k)$
throws if the EvalValue does not exist, or if the normalization fails.

```
Double getCost(Value value)
```
returns the cost for given value, or null if no cost available for value.

```
double getMaxCost()
```
returns the maximum of the costs for this evaluator.

```
Double getCost(UtilitySpace uspace, Bid bid, int index)
      throws Exception
```
returns the cost of given bid
May throw if the bid is incomplete or utilityspace has problems

```
Value getMaxValue()
```
returns the first value that has the maximum evaluation-value.

```
Value getMinValue()
```
returns the value that has the smallest evaluation-value

Bid

fDomain
variable holding the domain

```
Bid(Domain domainP, HashMap<Integer,Value> bidP)
      throws Exception
```
constructor for a Bid.
The bidP is a hashmap that maps issue IDs to Values (which are Strings).
Will throw exception if not all issues in the domain are assigned a value.

```
Value getValue(int issueNr) throws Exception
```
returns the picked value for a given issue ID number.

```
void setValue(int issueId, Value pValue)
```
sets the value for issue with issueId to pValue.

```
boolean equals(Bid pBid)
```
returns true if bids are equal

```
HashMap<Integer, Value> getValues()
```
returns the entire set of bid values

References

[sun07] Sun Developer Network, JDK 6 Update 3. http://java.sun.com/javase/downloads/index.jsp