Multi-Agent Systems Project

Panagiotis Chatzichristodoulou and Kirill Tumanov

December 5, 2013

0.1 Introduction

0.1.1 Abstract

This paper addresses the implementation of an intelligent negotiation agent. Furthermore, it discusses and analyses the results obtained from the experiments that were performed. The agent development was based on the *Genius* software platform, released by Delft University in 2009. Genius introduces an environment where agents can negotiate within different domains - discrete or continuous. Genius is the standard in the agent negotiation domain platforms. Furthermore, in accordance to the *ANAC'2013* guidelines, the agent is based on the *BOA* (Bidding Strategy, Opponent Model, Acceptance Strategy) framework [2]. The work was performed as a practical part of the Multi-agent Systems course at Maastricht University.

0.1.2 Framework

Agents in the Genius platform can either be standalone, or belong to the BOA framework. A single-entity agent can include all the required modules in itself or use the BOA components instead. Some additional helper classes can exist outside of the main agent class. A BOA agent implements the four models needed independently, and its modular architecture makes the agent more versatile and extendable. The Fig. 1 depicts how the BOA components are integrated.

Figure 1: Scheme of a BOA framework
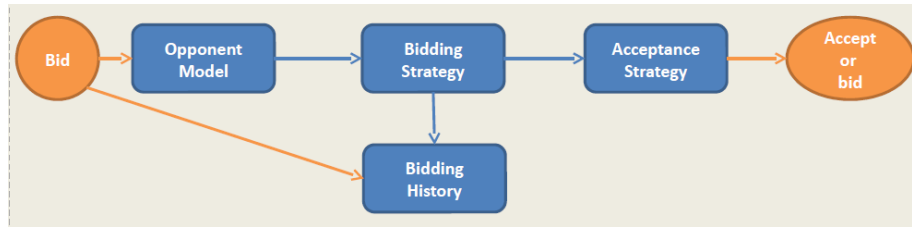
0.1.3 Agent summary

The agent described in this paper was named NegoAgent. It behaves as a self-interested negotiator, meaning that it seeks to maximize its utility without taking into account the opponent's score, or the social welfare. However, it takes into account the discount factor δ, and keeps track of the negotiation time in order to receive the highest utility possible. The agent is not going to accept not reasonable offers - it believes that "no deal is better than a bad deal", but when possible it tries to achieve a "win-win" compromise with an opponent. Overall, the agent is not tied to any specific domain or opponent behavior type, on contrary it tries to be adaptive to these challenges to its benefit.

A description of the BOA components used for agent development is given in subsequent sections, followed by modeling results and discussion.

0.2 Bids Offering Strategy

Agent's bid offering strategy was implemented as time-dependent (TD). The NegoAgent_TDOffering.class requires two user parameters: P_{min} and P_{max} which specify the lower and upper bounds for the bids agent is to propose. In the current version of the agent these values were kept equal to 0 and 0.99 respectfully. No experimentation was done on changing these values, it may be the case that some alterations in specific negotiation domains are beneficial, however this topic is left for an additional research.

Offering strategy introduces two methods used in the agent: `isNash()` and `countUniqueBids()`. First one is a simple implementation of the two-factor (due to the number of negotiating parties) maximization algorithm. An idea behind it is in finding a Nash equilibrium point, there none of the parties can obtain a higher utility at no cost to any other party. This particular implementation does not claim to be optimal, but it produced sufficiently good results for the agent at this stage of development. The pseudo-code of the `isNash()` method is given as follows:

```
double nashsum;
boolean isNash(){
    bid = getOpponentLastBid();
    if (bid != null) {
        temp = bid.getMyUtility() + bid.getOppenentUtility();
        if (temp < nashsum && bid.getOppenentUtility() < bid.getMyUtility())
            return true;
        nashsum = temp;
    }
    return false;
}
```

Here `nashsum` stores a previously found maximum combined utility result to compare the last opponent bid against. When `nashsum` becomes less than the last bid combined utility, then Nash is found and true is returned. An additional limit in the opponent's and agent's utilities is set to ensure, that Nash bid was proposed by the opponent, and not by the agent itself.

The second method `countUniqueBids()` is implemented to find and count all the opponent's non-equal bids as well as calculate a total sum of all proposed bids. A pseudo-code of the method is given below.

```
uniquebids;
double bidsum;
countUniqueBids() {
    int count = 0;
```

```
    bid = getOpponentLastBid();
    if (getOpponentBidHistory().size() == 1) {
        uniquebids.add(bid.getMyUtility());
        bidsum += bid.getMyUtility();
    }
    else {
        for (uniquebid : uniquebids) {
            if (uniquebid != bid.getMyUtility()) {
                count ++;
            }
        }
        if (count == uniquebids.size()) {
            uniquebids.add(bid.getMyUtility());
            bidsum += bid.getMyUtility();
        }
    }
}
```

This method is called every time an opponent proposes a new bid. As a result a list of `uniquebids` representing agent's utility values, `bidsum` value of all bids and `count` are stored and used for further calculations.

The main procedure of the agent's bidding strategy is `determineNextBid()` which relies on a time-dependent function `p(t)`. For this work we introduce a novel TD function, results of which are determined by the values received from the methods described above. The following is a pseudo-code of the `p(t)` implementation.

```
int bidsThreshold ;
double timeThreshold;
double p(double t) {
    countUniqueBids();
    double ft = calculateF(t);
    double time = getTime();
    if (getNumberOfPossibleBids() < bidsThreshold) {
        if (time > timeThreshold) {
            ft += getOpponentLastBid.getMyUtil()/(1 - time);
        }
    }
    pt = Pmin + (Pmax - Pmin) * (1 - ft);
    return pt;
}
```

Here `pt` is used only for adjustment of the ft result based on the P_{min} and P_{max} bid bounds discussed at the beginning of the section. The most interesting part

is a calculation of `ft`:

$$f(t) = \begin{cases} df(\sin(-n \cdot e^{t \cdot df}) + (\frac{1}{df} - 1)) \cdot \log(t \cdot df + 1) \cdot \frac{sum}{ubs} t & \text{if ubs>1 and !isNash(),} \\ \frac{df}{2}(\sin(-n \cdot e^{t \cdot \frac{df}{2}}) + (\frac{2}{df} - 1)) \cdot \log(t \cdot \frac{df}{2} + 1) \cdot \frac{sum}{ubs} t & \text{if ubs>1,} \\ df(\sin(-n \cdot e^{t \cdot df}) + (\frac{1}{df} - 1)) \cdot \log(t \cdot df + 1) & \text{otherwise.} \end{cases}$$

(1)

where df - a division factor in a range $\{0 \dots 1\}$, n - number of issues a negotiation is about, t - a normalized representation of time, sum - a total sum of all opponent's bids at the moment of t (afore-referred as a `bidsum`), ubs - number of unique opponent's bids at the moment of t (a size of an afore-referred list of `uniquebids`).

The function $f(t)$ was designed so that it allows modification of bid offering dependent on the negotiation status. If opponent proposes more non-equal bids of a decent utility for an agent, then agents offers new bids more cautious - simulation of waiting for the best opponent's offer. On contrary, when opponent proposes a few non-equal bids, agent tries to provoke an opponent for cooperation, by offering more new bids. In addition, a fact of Nash equilibrium reach by an opponent is treated as a trigger to loose agent's own offering, and to start biding more actively. It is done in order to prevent an opponent from unnecessary utility losses, whilst it is likely that agent's own score is not going to be lowered. Finally, increase of t during the negotiation process unveils more new bids to offer, and n ensures that the offering will be domain-dependent.

Referring back to the pseudo-code of `p(t)`, in case of a very limited utility space (`getNumberOfPossibleBids() < bidsThreshold`) and negotiation time running out (`time > timeThreshold`), agent begins to concede by increasing the $f(t)$ value proportionally to the remaining time $(1 - t)$. This allows to reach an agreement, instead of loosing all the utility. However, note that when opponent follows a non-cooperative strategy, this approach will not be of help.

0.3 Acceptance Strategy

An acceptance strategy of the agent was based on the ABiNeS agent's [1] acceptance strategy and on a time dependent sigmoid function.

0.3.1 ABiNeS

This module introduces an acceptance threshold ℓ. The value of ℓ corresponds to the agents concession degree and is modified during the negotiation. This change is based on the previous results of the negotiation as well as on the domain.

The agent is build as a self-interested agent and will be more concessive as the deadline of $t = 1$ approaches. The acceptance threshold should always be higher than the utility agent can obtain at deadline. Therefore, the threshold should never be lower than the $Utility_{max} \cdot discount_{time-1}$, where $Utility_{max}$ is the maximum utility the agent can receive without taking into account the

discount factor. In contrast, if it takes the agent too long to reach an agreement, it may receive very low utility due to the influence of a discount factor even if the agreed outcome proposes a high utility.

To address the issue factor λ is introduced. It is used to balance between exploring and exploiting the negotiating partner. More formally, when time is smaller than λ, it should be modified to gradually converge to $Utility_{max} \cdot discount_{time-1}$. So the ℓ is defined as follows:

$$\ell = \begin{cases} u^m - (u^m - u^m d^t)(t/\lambda)^a & \text{if } t < \lambda, \\ u^m d^t & \text{if } t \geq \lambda. \end{cases} \tag{2}$$

$$\lambda = \begin{cases} \lambda = \lambda_0 + (1 - \lambda_0)^b & \text{if } t == 0, \\ \lambda = \lambda + w(1 - \lambda_0)\sigma^{tc} & \text{if } 0 < t \leq 1. \end{cases} \tag{3}$$

So as time increases, ℓ slowly decreases, and as time reaches λ, ℓ is set to $Utility^{max} \cdot discount^{time}$. Graphical representation of the process is presented on Fig. 1. Here $a < 1$ is a user specified constant value, and notable are values of b and c which define the curvature of nonlinear piece, specifically they set a lower bound of an acceptance threshold. This fact is crucial for the strategy's implementation, as both b and c values should be domain-dependent in order to adequately reflect the setup features and prevent early acceptance of non-desirable offers from an opponent. Generally $b > 1$ and $c > 1$. In this work they were defined as follows:

$$\begin{cases} b = ol \cdot \theta_1 \cdot (of - mf) \cdot (1 - t) \\ c = ol \cdot \theta_2 \cdot (of - mf) \cdot (1 - t), \text{and} \end{cases} \tag{4}$$

$$\begin{cases} b = b \cdot (1 - usmin) \\ c = c \cdot (1 - usmin), \end{cases} \tag{5}$$

where ol - an opponent's last bid utility, of and mf - opponent's and the agent's first bid utilities respectfully, and $\theta_{1,2}$ - coefficients with values > 1. Formula 5 is used for b and c values normalization within a given domain. $usmin$ - is an agent's minimum possible utility in a given utility space.

An acceptance condition is a function of the history of the previous negotiations, the current acceptance threshold, and the outcome of the negotiation at time t. The agent will accept a proposal π if it receives utility bigger than his current threshold, or his next-bid to propose utility. However this classic approach is not flexible enough when dealing with a strong opponent. An agent to be able to accept reasonable π also need to be willing to concede more, then necessary. To do so agent needs to take into account its opponent's behavior, namely, identify how actively is an opponent bidding in order to reach an agreement. For this purpose method `countUniqueBidsAtPeriod()` was introduced. It is to calculate a number of unique bids within a given time frame. The method's pseudo-code is shown below.

Figure 2: Change of ℓ over time
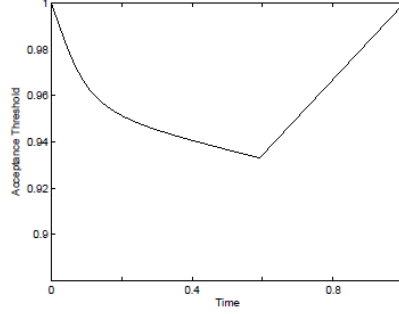

```
countUniqueBidsAtPeriod(double t1, double t2) {
    if (getOpponentBids().filterByTime(t1, t2).size() != 0) {
        for (bidInPeriod : bidsInPeriod) {
            ub.add(bid);
        }
        ub.removeDuplicates();
        return ub.size();
    }
    return 0;
}
```

A resulting number of bids is used as a definition of σ^t in equation 4.

Finally define ω in equation 4. It's purpose is in weighting the effect of σ^t on λ, dependent on the behavior of the opponent. In this work ω was calculated as:

$$\omega = \frac{getUniqueBidsCount()}{getOpponentBidHistory().size()}, \tag{6}$$

where divider is received from the Bidding Strategy class. Therefore ω controls a decrease of the λ and of the acceptance threshold as a consequence.

In addition to the described above threshold a *"pressure"* threshold was defined. It was found as shown:

$$pressureThreshold = onbu \cdot (1 - p) \cdot e^{1-onbu}, \tag{7}$$

where $onbu$ - is an opponent next bid's utility received from the Opponent Model class, and p - a user specified pressure constant in a range $(0 \ldots 1)$.

0.3.2 Sigmoid Function

The acceptance threshold is implemented so as to make the agent as hard-headed as possible. This makes the agent behave non optimally against opponents with the same strategy as his. For this reason, a sigmoid function

`isLeapBigEnough()` is introduced. The pseudo code of the function is the following:

```
boolean isLeapBigEnough(double bidUtil,double prevBidUtil,
          double threshold){
   difference = bidUtil-prevBidUtil;
   return difference*getTimeMultiplier()>threshold;
}
```

The function returns the difference of the two latest bids of the opponent multiplied by the `TimeMultiplier`. The `TimeMultiplier` is the value of time passed in the sigmoid domain. That means that the value of the multiplier will increase fast until approximately the middle of the negotiation, and the difference value will not be discounted as much after that. If the opponent leaps enough towards the agent trying to reach an agreement, the `getleapthreshold` function will exploit the intention of the opponent. The threshold is based on the function:

$$\tau = \begin{cases} \xi & \text{if } t < 2, \\ \xi \cdot \dfrac{1}{\sqrt[3]{(1 + e^{(-x)})}} & \text{if } 2 < t \le 3.5 \\ \xi \cdot \dfrac{1}{\sqrt[3]{(1 + e^{(x)})}} + \alpha & \text{if } t > 3.5 \end{cases} \tag{8}$$

The time is passed to the sigmoid domain and ranges between $\{-5 \ldots 5\}$ whilst ξ and α are constants that modify the speed by which threshold decreases over time. As are defined here, the graphical representation of the threshold multiplier is shown in Fig. 3 The threshold multiplier is taken from piecewise of

Figure 3: The distance between the initial utilities of the agents
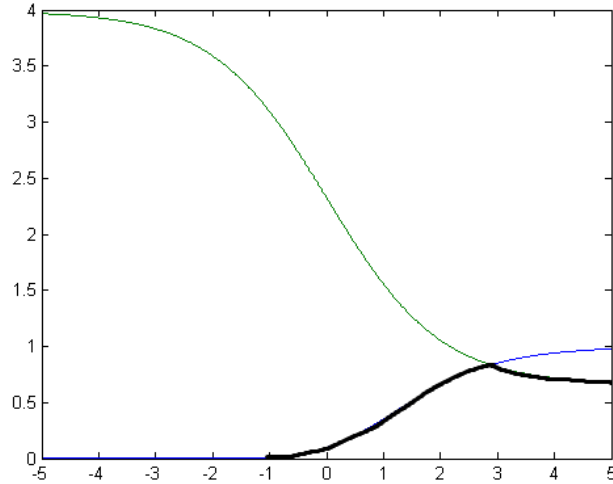
the lower parts of the two functions, and it modifies ξ according to its value. Although having this threshold makes the agent agree on bids that it would not just with the acceptance threshold , having this threshold increases the average utility the agent receives by 5-10%.

0.3.3 Acceptance formula

Finally the agent's Acceptance Strategy is comprised of the following statement represented in pseudo-code below:

```
if(lastOpponentBidUtil <= weakThreshold * delta) {
    Reject();
}
if(isLeapBigEnough() && lastOpponentBidUtil > getTau()){
    Accept();
}
if(lastOpponentBidUtil >= acceptanceThreshold
    or lastOpponentBidUtil >= pressureThreshold) {
    Accept();
}
Reject();
```

Here `weakThreshold` is provided by the Opponent Model, which is described in the following section.

0.4 Opponent Model

The opponent model is based on the `Hard-Headed Model` of the `Genius-BOA` framework. The basic difference from the `Hard-Headed Model` lies in the way the weights are modified. The weights are updated only if the number of unique bids an opponent has made in the recent past is above a certain value. That is, the model is updated only if the opponent has made a lot of different bids; this happens because if the weights were updated taking into consideration the number same bids, the model will converge to wrong weights.

Besides from the basic model and its basic modification, one extra component was implemented. [3]

0.4.1 Distance of the opponent

The opponent model also calculates a threshold that is based on the *Distance Of The Opponent in Our Utility Space*−`DOUS` [3] value. This value is used in the Acceptance Strategy and it provides the agent with a threshold that helps reject bids that are far from Nash, in high discount environments.

The basic algorithm of the DOUS value and its threshold calculation is:

```
double getDousThreshold(horizon){
```

```
    myUtility = negotiationSession.getMyMaxUtil();
    dous      = myUtility - getOpponentLastBid().getMyUtil();
    mean      = getOpponentBidsMean(horizon);
    variance  = getOpponentBidVar(horizon,mean);
    time      = getTime();
    threshold = (mean+dous)/2;
    if(time > timeLimit){
        threshold+=(1-(time-0.9))*variance;
    }else{
        threshold += variance;
    }
    if(time > delta){
        return 0;
    }
    return threshold;
}
```

The `horizon` parameter, defines the number of past bids the agent takes into account when calculating the `DOUS` [3] distance. The smaller the horizon, the more short-sighted the agent's threshold will be. If the horizon is given a big value, the fluctuations of the biding values will not be imprinted with the same precision to the output value, and this may result in a small threshold. Usually, a `horizon` between 10-100 captures the fluctuations of the opponents bids efficiently. A parameter `NegotiatedTooLong` is introduced, and is set equal to the discount factor. That means that whenever the discount factor is a small number, and the utility gets high discount over time, the threshold will be kept for a small amount of time. On the other hand, if the discount factor is a big number, the threshold will be kept for a bigger amount of time. The DOUS value is versatile, as it is not domain dependent. As it is presented here, it can be implemented only on negotiations between two agents, but it can be expanded to multiple agent negotiations creating a DOUS distance for every opposing agent. The threshold that is outputted is a `weakThreshold`. That is, if the value of a bid an agent offers to the Nego Agent is smaller than the `weakThreshold`, the bid is rejected. If the value is higher, we compare it with the strong threshold computed in the Acceptance Strategy.This ensures that the agents will reject points that are far from the Nash equilibrium. The amount of time this threshold holds is dependent on the discount factor of the negotiation. This restriction is set because the agent needs to be more concessive as the negotiation approaches the deadline. If the discount factor is 1, and no depreciation exists, this threshold stands for the whole negotiation. The visualization of the DOUS distance would be:

0.5   Opponent Model Strategy

This module was adapted from a Best Bid Strategy, included in *Genius*. The idea behind it is in keeping a small history of opponent's bids for determination

Figure 4: The distance between the initial utilities of the agents
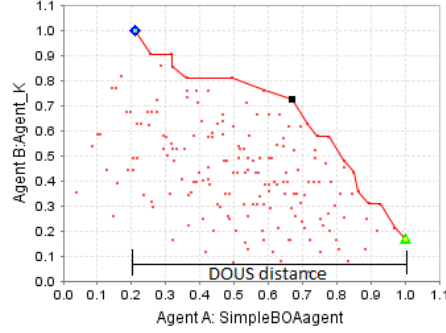
of the best bid. An experimentation with a history size showed that keeping of the considerable amount of opponent's bids has a negative impact on the agent's performance, thus the history size was set to 10 independent of the domain.

The most basic OMS was selected to show the strengths and weaknesses of the NegoAgent in it's current state regardless or other variables introduced by more advanced models. However it is likely that a more tuned model is to increase the agent's performance, and thus this leaves a space for further investigation.

0.6 Binding the Modules

It is useful to have a single-entity agent instead of several BOA components. In particular, *Genius* allows single-entity agents run separate sessions against each other, and *ANAC* competition uses this kind of agents in the tournament series. A single-entity agent may consist of numerous parts, however the one described in the paper is comprised of four main BOA components, described in the previous sections.

In order to bind all the modules together, a class that extended the BOA agent and passes each of the modules with its parameters as parts of the agent should be created. Then it is necessary to override an `agentSetup()` method of the class. The pseudo-code of the overriden class is:

```
NegoAgent extends BOAagent {
    @Override
    agentSetup(){
        OpponentModel om = new OpponentsModel(negotiationSession);
        OMStrategy oms = new BidStrategy(negotiationSession , om);
        OfferingStrategy offering = new NegoAgent_TDOffering(nego-
        tiationSession, om, oms, Pmax, Pmin);
        AcceptanceStrategy ac =  new AStrategy(negotiationSession,
        offering, a , b, c);
```

```
        setDecoupledComponents (ac, offering, om, oms );
    }
}
```

0.7   Results

The agent implemented is a strong willed agent. It will accept bids fast when
the discount factor is big, otherwise it will negotiate for the biggest part of the
negotiation session. The Acceptance Strategy with the weak threshold restricts
the agent from accepting offers that are not at least close to Nash equilibrium.
As $t \rightarrow 1$, the weak threshold is dropped, and the acceptance threshold is
reduced, making the agent capable of accepting more reasonable bids. Its TD
offering makes the agent extremely efficient at obtaining the best utility possible
from strong willed opponents. Table 1 shows the performance tests results of a
proposed NegoAgent versus several existed agents ranging from most basic to
the most advanced ones.

Table 1: NegoAgent's negotiation results

Opponent	Role (B)	Rounds	u, A	u, B	δu, A	δu, B	t
Simple	Mary	41	1	0.4500	0.9755	0.4390	0.1148
	Sam	73	1	0.5250	0.9591	0.5035	0.1932
Meta	Itex	6517	0.9268	0.3073	0.9268	0.3073	0.9822
	Cypress	7475	0	0	0	0	1
TitfotTat	CarA2	1406	0.9641	0.8936	0.7269	0.6737	0.9818
	CarA1	1536	0.9800	0.7642	0.7360	0.5739	0.9953
Negotiator	SmartP2	1145	0.9585	0.6635	0.8466	0.5861	0.8911
	SmartP1	1061	0.9776	0.5540	0.8605	0.4877	0.9158
CUHKA	NiceOrDie2	3681	0.2990	0.2990	0.2243	0.2243	0.9990
	NiceOrDie1	3437	0.2990	0.2990	0.2243	0.2243	0.9986

u - utility of agent, δu - discounted utility. Opponents negotiated for role B

Here you played too much whith the ˜, I deleted some, but then I got bored...
In the first negotiation outcome, `NegoAgent` completely dominates over a
`SimpleAgent` in the domain Sam and Mary. An agreement is reached very fast,
and the `NegoAgent` gathers the max amount of utility it can gather in this
domain.

In the second session, `NegoAgent` negotiates in the ItexVsCypress domain
against the `MetaAgent`. `MetaAgent` is not a normal agent; it contains an agent
repository that consists of the agents that competed in the ANAC'2012, and
uses machine learning to decide which agent fits best each negotiation domain.

It competed in the ANAC'2013 and took the second place. `MetaAgent` is completely dominated on the first session, whereas on the second, no agreement is reached.

In the third session, `NegoAgent` is opposed by `NiceTitForTat` agent in the car buyer-car seller domain. `NiceTitForTat` loses, but manages to keep the utility it gains from the session relatively high. In contrast to the `SimpleAgent`, the negotiation on both sessions took a lot of time and almost ended without reaching a settlement.

In the fourth session, `NegoAgent` competes against `Negotiator` agent in the Smartphone buying domain. `Negotiator` also looses against `NegoAgent` and, in contrast to `NiceTitForTat`, it does not manage to keep its utility high enough.

In the final session, `NegoAgent` is opposed by `CUHKA` agent in the NiceOrDie domain. `CUHKA` agent is a very intelligent self-interested agent, and the agreement is reached on the nash point of this domain. It must be noted that this domain had only 3 possible bids that could be offered, and the agreement is accepted on the final offer of our agent.

0.8   Conclusion

Building a negotiating agent is a non-trivial matter, as the negotiation domains and issue sets are practically infinite, so are the strategies that can be used. Negotiation agent design is an active research field that has been introduced recently, thus very limited practical experience is shared yet. The proposed NegoAgent tries to incorporate some of the Multi-Agent Systems concepts in it's negotiation behavior. Some domain/opponent specific cases might be treated separately, but it was decided to as general and versatile as possible. Overall results the NegoAgent shows against competitive opponents are promising, and the agent is believed to be a solid base for further research.

Bibliography

[1] Hao J., Leung H.: ABiNeS: An Adaptive Bilateral Negotiating Strategy over Multiple Items, should be added info about the place of publication, (2013)

[2] Genius platform at TU Delft, `http://www.mmi.tudelft.nl/genius/` Retrieval date: 01.12.13

[3] Fourth Automated Negotiating Agent Competition (2013).: `http://www.itolab.nitech.ac.jp/ANAC2013/` Retrieval date: 01.12.13