Multi-Agent Systems Project

Panagiotis Chatzichristodoulou, Kirill Tumanov

December 2, 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.

0.1.2 Project Analysis

The agents created are based on the negotiation platform *Genius* [2], released by Delft University in 2009. It introduces an environment where agents can negotiate within different domains - discrete or continuous. Genius is the standard in the agent negotiation domain platform. The work was performed as a practical part of the Multi-agent Systems course at Maastricht University.

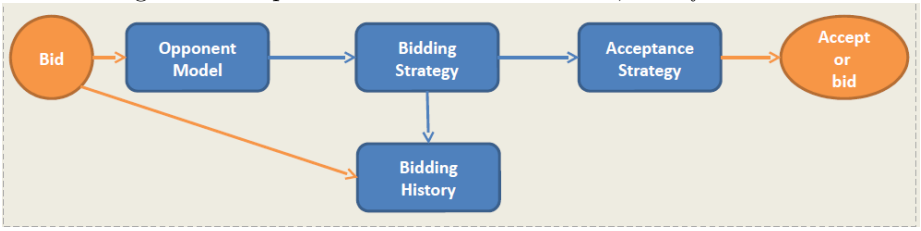
0.1.3 Agent models

Each agent can consist of 4 different models:

- Bidding Strategy
- Acceptance Strategy
- Opponent Model Strategy
- Opponent Model

The agents in the Genius platform can either be standalone, or belonging to the BOA framework. A standalone agent is usually a single Java file having all the models there. Some helper classes can exist outside of the main agent class. A BOA agent implements the four needed models independently , and that makes a BOA agent more versatile and easier to implement, due to the modularity of its architecture. In our implementation we chose a BOA agent and used the genius platform to create our final agent that consists of those four parts. The figure below depicts how the four different models cooperate to output a bid or an acceptance decision.

Figure 1: Cooperation between four models,*BOA framework*



0.1.4 Agent type

The agent that is implemented in this paper, is self interested . That is an agent that seeks to optimize its utility without taking into account the opponent's utility, or the social welfare. Furthermore, it takes into account the discount factor, and will not negotiate for too long. That way, it will receive as high utility as possible most of the time.

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 supposed to be 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`:

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

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. 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:

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

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

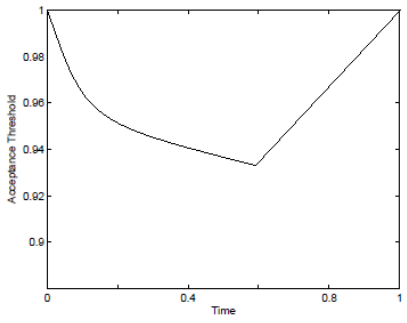
So as time increases, ℓ slowly decreases, and as time reaches λ, ℓ is set to $Utility^{max} \cdot discount^{time}$. Graphical representation of the process is presented on Fig. 1. Here $a < 1$ is a user specified constant value, and notable are values of b and c which define the curvature of nonlinear 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.

Figure 2: Change of ℓ over time



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 it's 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.

```
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 3.

Finally define ω in equation 3. 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 $(0 \dots 1)$.

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

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

```
}
Reject();
```

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

0.4 Opponent Model

An opponent model is based on the Hard-Headed Opponent Model of the Genius-BOA framework.

Besides from the basic model, there are two extra components in the model. [3]

0.4.1 Simple Frequency

The Simple frequency model only updates weights when the bid change often. That means that the weights of the model are only changed when the opponents bids pass a certain numerical threshold.

0.4.2 Distance of the opponent

The opponent model also calculates a threshold that is used in the Acceptance Strategy and is based on the distance of the opponent.

The basic algorithm of the distance of the opponent is:

DOUS = $\mathrm{u}^m - firstOppBid$

calculate mean and variance foreach past bid where :

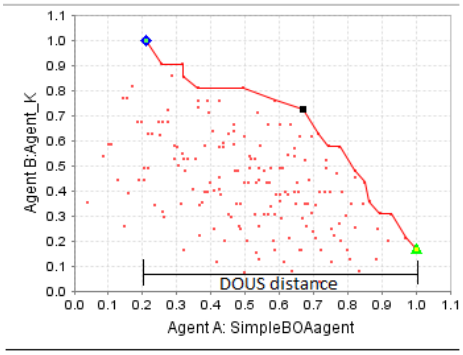
$firstOppBid < bidUtil < u^m$

use the $(mean + dous)/2$ *as a threshold value if* $time < .5$

$if(time < .5)$*return threshold else return* 0

The visualization of the DOUS distance would be:

Figure 3: The distance between the initial utilities of the agents



The DOUS value is versatile, as it is not domain dependent. As it is presented here, it can be implemented only on negotiations of two agents, but it can

be expanded to multiple agent negotiations. The threshold is outputted is weak threshold. That is, if the value is smaller than the threshold, the bid is rejected. If the value is higher, we compare it with the normal threshold computed in the Acceptance Strategy. This threshold ensures that the agents will reject points that are far from the Nash equilibrium. After some time in the negotiation, the threshold is set to zero, because the agent needs to be more concessive as the negotiation approaches the deadline.

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 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:

- Bidding Strategy
- Acceptance Strategy
- Opponent Model
- Opponent Model Strategy

In order to bind all the modules together, a class that extended the BOA agent and passes each of the modules with it's 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 very-strong willed agent. It will accept bids faster when the discount factor is big, and will negotiate for a long time when the discount is small. Its acceptance strategy with the weak threshold enable it to accept offers that are at least close to Nash equilibrium, and as time approaches 1, it accepts more reasonable bids. Its bidding formula makes the agent extremely efficient at obtaining the best utility possible from strong willed opponents. The figures below show the results against a very good 2013 agent.

0.8 Conclusion and further work

Building a negotiating agent is a non-trivial matter, as the negotiation domains are practically infinite, as are the strategies that can be used. As the agent negotiation is a research field that has only been introduced recently, and as there are strategies that behave well given different domains there is a lot of research to be done, until we reach a universal strategy, or a strategy that works very well in all of the domains.

Bibliography

[1] Hao J., Leung H.: ABiNeS: An Adaptive Bilateral Negotiating Strategy over Multiple Items

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

[3] Something