NegoAgent

Panagiotis Chatzichristodoulou and Kirill Tumanov

December 1, 2013

This paper addresses implementation of an intelligent negotiation agent as well as discussion and analysis of experiments performed are result obtained. Agent was develop based on the *Genius* software platform in accordance with the *ANAC'2013* guidelines and within the *BOA* (Bidding Strategy, Opponent Model, Acceptance Strategy) framework. This work was performed as a practical part of the Multiagent Systems course at Maastricht University.

0.1 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`:

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

where df - a division factor in a range $\{0 \dots 1\}$, n - number of issues negotiation about, t - representation of time in a range $\{0 \dots 1\}$, 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.

0.2 Acceptance Strategy

0.2.1 Summary

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 $Utility_{max} * discount_{time-1}$, 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.

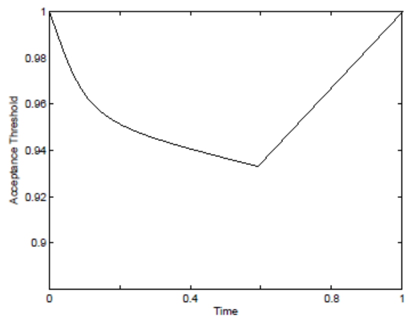
To address the issue factor λ is introduced. It is used to balance between exploring and exploiting the negotiating partner. More formally, when time is smaller than λ, it should be modified to gradually converge to $Utility_{max} * discount_{time-1}$. So the ℓ is defined as follows:

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

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

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 notable is that values of b and c define the curvature of non-linear 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 $c > 1$.

Figure 1: Change of ℓ over time



0.2.2 Acceptance Model

Now that we have defined the environment, the 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 he receives utility bigger than his current threshold, or his next-bid to propose utility.

0.3 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. [2]

0.3.1 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.

0.3.2 Distance of the opponent

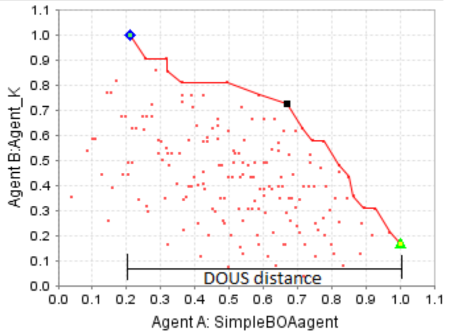
The opponent model also calculates a threshold that is used in the Acceptance Strategy and is based on the distance of the opponent.

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 2: 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.

0.4 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 );
    }
}
```

0.5 Conclusion

Our agent beats them all!

Bibliography

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

[2] Something