IN4010-12 Artificial Intelligence Techniques - Group 18

Koen Wendel
4170555
Delft University of Technology

Yapkan Choi
4025067
Delft University of Technology

Job Zoon
4393899
Delft University of Technology

1 INTRODUCTION

For the course *IN4010-12 Artificial Intelligence Techniques* lectured at the TU Delft, this paper is written to elaborate on a negotiation party suitable for automated bilateral negotiation on discrete domains, designed by Group 18.

The negotiation party is introduced step-by-step according to the BOA framework [2]. Furthermore, the methods to account for discounted domains and domains with preference uncertainty are discussed.

Performance is quantified by pitting it against other negotiation parties which are known to perform well across multiple domains.

Finally, weaknesses of the party and further recommendations are discussed.

2 NEGOTIATION STRATEGY

In this section, the negotiation strategy of our agent is explained. The negotiation strategy is a combination of a time dependent strategy and the offer-proposing function described in [7], which is part of PhoenixParty [7], a negotiation agent that was used in the ANAC 2015 competition [5]. The different components of the negotiation strategy are split according to the BOA framework.

The following subsections will go over the negotiation strategies, consisting of the four BOA components and the adaptations necessary for discounted domains and when there is preference uncertainty.

2.1 Acceptance Strategy

In our acceptance strategy, there are two possibilities for our agent to accept the offer of the opposing agent. The first possibility is based on AC_Next [3]. Our agent will accept an offer, if the utility of the opponent's offer is (1) higher than the offer our agent is going to propose and (2) is equivalent or higher than the best offer that we have received from the opponent. We added the second condition because we never want to accept an offer from the opposition with a lower utility than it is willing to offer. So, if we already saw that the opponent is willing to offer higher than the current offer, we will not be accepting it.

The second possibility uses a time-dependent function. The closer the negotiation gets to the deadline, the more willing our agent is to accept lower offers. The function is shown in equation 1.

$$a = m - (m - o) \times t^2 \tag{1}$$

In equation 1, a is the utility of the offer at which we are willing to accept, t is the percentage of time that has passed, m is our best offer in terms of utility and o is the utility of the opponent's best offer. When the bid of the opponent is equivalent or higher than a, we will accept it.

There are two parameters (α and β) added to our acceptance strategy to make it possible to adjust the strategy. α is used to define a bottom line for an offer that is acceptable. When the negotiation session reaches the last β percent of rounds ($1 - t \leq \beta$), this bottom line offer value is used instead of the acceptable offer value in equation 1.

The time-dependent utility function for this case is shown in equation 2. Where a is the utility of the offer at which we are willing to accept, m is our best offer in terms of utility and o is the utility of the opponent's best offer. The default value of α is 1.4 and the default value of β is 0.02.

$$a = \begin{cases} \frac{m}{\alpha} & \text{if } \frac{m}{\alpha} > o \\ o & \text{otherwise} \end{cases} \tag{2}$$

2.2 Bidding Strategy and Opponent Model Strategy

For our bidding and opponent model strategy, it was decided to implement the offer-proposing function described in [7], which is part of PhoenixParty [7], a negotiation agent that was used in the ANAC 2015 competition [5]. This approach tries to find offers that are as close to the Pareto frontier as possible. It does this with the approximation algorithm Distance-based Pareto Frontier Approximation (DPFA), which is proposed in their work.

DPFA is based on the assumption that the opponent is also searching for Pareto optimal deals. This assumption is represented by the opponent's first bid, best bid and last bid. These three bids are the reference bids, which are used to compare and find offers from a set of available bids for our party. Proposing an offer that has small distance to the three reference bids, increases the chance that the opponent accepts the offer and it should lead to an outcome that is close to the Pareto frontier.

This specific strategy was chosen, because it complements the time-dependent utility threshold function of our acceptance strategy. In DPFA, a different threshold function based on Gaussian process regression [7], is used to determine the minimum utility that is considered for the available bids. We used the function in equation 1 to determine the utility threshold. Furthermore, there are several other differences between our implementation and the proposed approach in [7], due to absence of some details in their work.

One difference is that our opening bid is the bid with the highest utility for itself, because reference bids are not always available at the start of the negotiation session.

Offers after the opening bid are determined using DPFA. First, all available bids with a utility above the value from our utility threshold function (equation 1) are obtained. Secondly, the three reference bids are collected: the opponent's first, best and last bid. The opponent's first bid is often their best offer for them. The

opponent's bid with the highest utility for our party should be close to the Pareto frontier. By keeping track of the opponent's last bids, our agent can propose bids that are close to the Pareto frontier, assuming that the opponent is also looking for them. All three reference bids can have different weights, to ensure that our party can account for random bids from the opponent.

For each available bid, the distance (rating) to the reference bids is computed with equation 7 from [7]. Bids with higher ratings will be closer to the reference bids. An important component of DPFA is keeping track which values the opponent offers each round, represented by ω. These are used to model the opponent's preferences for certain values, in order to apply different weighting to them during computing of the ratings. This component is explained in more detail in the next subsection about our opponent model.

The function to calculate the rating of an available bid is shown in equation 3. Where a is a bid from the set of available bids. r_i is the i-th reference bid, with $i = 1, 2, 3$ being the first, best and last bid respectively. γ_i is the weight for the i-th reference bid, and these can be changed as BOA parameters. The default values for γ, $[1.0, 0.8, 0.3]$, are taken from [7]. a and r_i are both vectors, with each element corresponding to one value of an issue (e.g. Food: Catering = 1, Handmade Food = 2). The vector difference of a and r_i is then element-wise multiplied (Hadamard product) with ω, the weight vector of the issue values in a.

$$\text{rating}(a) = -\sum_{i=1}^{3} \gamma_i \|\omega \circ (a - r_i)\|_2 \tag{3}$$

Lastly, after computing all the ratings, a bid is chosen randomly from the available bids. Bids with higher rating (closer to zero) will have higher probability to be drawn. Their work [7] does not provide further details, therefore it was decided to bias the drawing using exponentiation in our implementation. This prevents that the agent always proposes the same bid, thus increasing exploration and the chances of the opponent accepting an offer that is close to the Pareto frontier.

The function to draw a rating is shown in equation 4. Where l and h refer to the lowest rating and highest rating that were computed respectively. $U(0, 1)$ is a uniformly distributed random number between 0 and 1, which gets raised to the power of b. b represents the amount for which the random number gets biased towards the maximum computed rating, when b is between 0 and 1. The default b is 0.25, but this can be changed as a BOA parameter. The bid with a rating that is closest to this drawn rating is chosen as our next offer. If there are multiple bids with the same rating, it picks one randomly.

$$\text{rating}(l, h, b) = l + (h - l) \times U(0, 1)^b \tag{4}$$

Our party with the implemented bidding and opponent model strategy can propose offers that increase the utility for itself relative to its previous bid. While our party has a utility threshold function, it can propose bids that are higher than the minimum utility, depending on the ratings of the available bids. Due to having some stochastic behaviour when offering a bid, it is not guaranteed that subsequent bids always have increasing utility.

2.3 Opponent Model

Our opponent model consists of keeping track which values the opponent offers each round, represented by ω. ω is initialized with one for each value of every issue in the domain. It was decided to model the opponent's preferences for certain values by comparing the offers to the first bid of the opponent. This is based on the assumption that the first bid of the opponent is usually the most important for them.

Each round, ω is updated: for values that are both in the current offer and first offer of the opponent, the corresponding values in ω are incremented by one. The ω values are used to apply different weightings to the issues of the domain during the computation of the ratings of our potential offers (equation 3).

2.4 Discounted Domains

Our original strategy does not perform well on discounted domains. The foundation of our strategy is waiting for the opponent to give us enough information and only move into the opponent's direction slowly. In undiscounted domains, this works well, and we usually get a deal in the last couple of rounds, but this does not work well for discounted domains. That is why some adaptations of our strategy are necessary in discounted domains.

Both our bidding and acceptance strategy use a threshold. The acceptance strategy will accept a bid if it is above this threshold, the bidding strategy looks in the subspace of bids above this threshold. In our usual strategy, this number decreases really slowly, usually ending with the best offer of the opponent.

Now if our agent encounters a discounted domain, this threshold should be way lower at the start, both for the acceptance strategy and the bidding strategy, because it is crucial to find an agreement earlier in the negotiation. How low this number is, should depend on the discount rate. For a discount factor of 0.9, it is not very important to adjust our original strategy; the loss of the discount is minimal. But for a discount factor of 0.05, the threshold should be really low, because every round will decrease the discounted utility dramatically. This means we needed to find a balance between the discount rate and our threshold function.

After some testing, we figured out that it was only beneficial to use a different strategy for discount factors lower than 0.7. For factors higher than 0.7 we were giving up more utility in the early parts of the negotiation than we would have lost by the discount. For discount factors between 0.45 and 0.7, we replace our threshold function with our minimumOffer variable. This variable is calculated using our optimal offer and the opponent's best offer (see our acceptance strategy). This means that we have a bigger space to search within for offers and that we accept lower offers, but not too low. The reason that we do not go lower here is that the discount factor is still not too low to wait a bit for an agreement to happen.

Finally, if the discount factor is lower than 0.45, we use a different threshold. We still use the minimumOffer variable, but the threshold is lowered even more according to formula 5.

$$\text{threshold} = \text{minimumOffer} - (0.45 - \text{discountFactor}) \tag{5}$$

This formula results in finding an agreement way sooner, but also in lower utilities. This is still beneficial, since a 0.5 utility in

the first 10% of rounds is better in a discounted domain than a 0.1 utility after 90% of the rounds.

2.5 Preference Uncertainty

To cope with negotiation scenarios where the utility function (i.e. preference) is only known through a limited number of ordered bids, a method is devised to transform those bids back to the original utility function. This method revolves around that the frequency of which the values in the bids are chosen are indicative for both the value and issue weights.

First the problem is defined clearly for N bids and I issues:

$$\begin{bmatrix} V_{1,1}(1) & \cdots & V_{1,I}(1) \\ \vdots & \ddots & \vdots \\ V_{n,1}(N) & \cdots & V_{n,I}(N) \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_I \end{bmatrix} = \begin{bmatrix} U(1) \\ \vdots \\ U(N) \end{bmatrix} \tag{6}$$

With $V_i(n)$ as the value weight of issue i of a bid, w_i as the weight of the issue and $U(n)$ the resulting utility.

For the given bids in equation 6, only the *ordering* of bid 1 to N and the *index* of the values for each issue are known. In other words: we know that a certain combination of values multiplied with a certain set of weights for all bids result in the given ordering.

To start solving equation 6, a linearly spaced vector U^e ranging from 0 to 1 is defined with length N. Now $S_{v,i}$ is defined as the summation of the times value v occurs in issue i in all bids, multiplied by the estimated utility of the bid the value occurs in. This is summarised in equation 7.

$$S_{v,i} = \sum_{n=1}^{N} \{V_i(n) = v\} \cdot U^e(n) \tag{7}$$

For all issues and values, equation 8 results in a $v \times i$ matrix, with $S_{v,i}$ being higher if it occurs more often and in higher ranked bids. Therefore, normalizing the columns of S with the maximum column values gives an estimation of the value weights:

$$V^e_{v,i} = \frac{S_{v,i}}{\max S_{*,i}} \tag{8}$$

Assuming that the highest weighted value for the highest weighted issue appears more often in higher ranked bids, the same principle can be applied to estimate the issue weights:

$$w^e_i = \frac{\max S^*_{*,i}}{\sum_{i=1}^{I} \max S^*_{*,i}} \tag{9}$$

An important note is that S^* is similar to S, but calculated using a different U^e, which now is linear spaced between -1 and 1 to penalize values that occur in low ranked bids.

The negotiation strategies can now use the estimated w^e and V^e to calculate the utility of the bids that we receive or plan to offer.

3 IMPLEMENTATION

Our agent and negotiation strategies are implemented with the BOA framework in the negotiation environment GENIUS [6]. This section gives a high-level description of the party and its structure, including the main Java methods that are used in the BOA components. The four BOA components all have their respective Java classes (Group18_AS, Group18_BS, Group18_OM and Group18_OMS), while the UtilityFunctionEstimate class provides methods for dealing with preference uncertainty.

3.1 Acceptance Strategy

The Group18_AS class holds our acceptance strategy for both undiscounted and discounted domains. The class has conditional statements that deal with discounted domains and preference uncertainty (with the UtilityFunctionEstimate class).

- init()
  Apart from the standard initiation function, this function also initializes the parameters a and b. If the parameters are not given, the standard values are used.

- determineAcceptability()
  In this function, the method getDiscountFactor() is used to check if the agent is in a discounted domain. Then the function determineAcceptabilityAction() is called to find the appropriate action for the (un)discounted domain.

- determineAcceptabilityAction()
  This is the main function to determine whether to accept an offer or not. Many variables are pulled out of the negotiationSession variable and used to calculate if the offer of the opponent is above out threshold. The functions findMinimumOffer(), findAcceptableOffer() and findAcceptableOfferDiscounted() are used to calculate different numbers that are used for the threshold. In the end, the return Action is decided based on those numbers.

- findMinimumOffer()
  The minimumOffer variable value is calculated in the function using the best offer of the opponent, our best offer and parameter a.

- findAcceptableOffer()
  In this function, the acceptableOffer variable value is calculated using equation 1.

- findAcceptableOfferDiscounted()
  This function is called when the domain is discounted. It calculates the acceptableOffer variable value using the formula presented earlier in this paper.

3.2 Bidding Strategy

The Group18_BS class takes care of determining the opening bid and the next bids, in conjunction with the opponent model and opponent model strategy. The class has conditional statements that deal with discounted domains and preference uncertainty (with the UtilityFunctionEstimate class).

- init()
  When the bidding strategy is initialized, a SortedOutcomeSpace is created to easily generate bids within a certain range or to get the bid with maximum utility.

- determineOpeningBid()
  The opening bid is the bid with maximum utility

- determineNextBid()
  Our algorithm to determine the next bid. First the opponent model is updated. Secondly, the minimum utility for the next bid is calculated with findLowerBound(). Thirdly, all bids that satisfy the previous calculated minimum utility are retrieved. Finally, the algorithm returns the bid chosen from our opponent model strategy.

- findLowerBound()
  Computes the mimimum utility for the next bid, which depends on the time left, the best offer of the opponent and the best possible offer (equation 1).

- getAvailableBids()
  All bids that satisfy the minimum utility are retrieved from the SortedOutcomeSpace.

3.3 Opponent Model

The Group18_OM class keeps track and updates the issue and value weights of the opponent. Every update also normalizes the weights.

- init()
  When the opponent model is initialized, the utility space is copied from the negotiation session as an AdditiveUtilitySpace. Furthermore, the value and issue weights of the opponent are initialized with ones, using the method initializeOmega().

- updateModel()
  This method updates the opponent model and is called from the bidding strategy component. The last bid of the opponent is compared with the first bid of the opponent. For each value of the last bid, if the value is the same as in the first bid, the not-normalized value is incremented by one and then normalized again.

- initializeOmega()
  The value and issue weights of the opponent are initialized with ones. This is implemented with the getEvaluators() and setEvaluation() methods from the AdditiveUtilitySpace and Evaluator classes respectively.

3.4 Opponent Model Strategy

The Group18_OMS class returns the next bid to be offered to the bidding strategy BOA component, given a list of bids that satisfy the minimum utility. The class has conditional statements that deal with preference uncertainty (with the UtilityFunctionEstimate class).

- init()
  When the opponent model strategy is initialized, an additiveUtilitySpace is created to to easily get the values weights for every bid. This method also initializes the parameters gamma and bias. If the parameters are not given, the standard values are used.

- getBid()
  This method is called from the bidding strategy to get the next bid. It gets the three reference bids with getReferenceBids(). The ratings are computed with computeRating() for all bids from the bidding strategy that satisfy the minimum utility. Finally, one bid is drawn using drawBidFollowRating().

- computeRating()
  This method computes the rating for a bid, while comparing with the three reference bids (equation 3).

- drawBidFollowRating()
  This method chooses a bid randomly, where bids with higher rating have higher probability to be chosen (equation 4).

- getReferenceBids()
  Returns a list of the three reference bids: first, best and last bid of the opponent.

- euclideanDistance()
  This method calculates the euclidean distance between two values, weighted by the issue weights.

3.5 Preference Uncertainty

The UtilityFunctionEstimate class is a helper class that is used when preference uncertainty is enabled. It estimates the value and issue weights, given the utility space of the domain and a ranked list of bids.

- UtilityFunctionEstimate()
  Constructor method that receives a domain utility space and a ranked list of bids. The value and issue weights are estimated with estimateValueWeights() and estimateIssueWeights() respectively. Then, setWeightsOfUtilitySpace() is called to set the weights of the utility space to the estimated weights.

- estimateValueWeights()
  Calls initializeMatrix() to initalize an empty value-issue matrix. Creates a linearly spaced vector from 0 to 1 with linspace(). Then the value weights are calculated with the equations in section 2.5.

- estimateIssueWeights()
  Calls initializeMatrix() to initalize an empty value-issue matrix. Creates a linearly spaced vector from -1 to 1 with linspace(). Then the issue weights are calculated with the equations in section 2.5.

- setWeightsOfUtilitySpace()
  After the value and issue weights are estimated, they are

added to the utility space with the getEvaluators() and setEvaluation() methods from the AdditiveUtilitySpace and Evaluator classes.

- initializeMatrix()
  An empty value-issue matrix is constructed with a nested HashMap datastructure: Map<issueNumber, Map<valueString, 0.0»
- computeWeightedFrequency()
  Method to compute weighted frequency of the weights. This is called from estimateValueWeights() and estimateIssueWeights().
- linspace()
  Generate a linearly spaced vector.
- getUtilityEstimate()
  AFter estimating the value and issue weights, the utility of a bid can be estimated with this method.
- getUtilitySpace()
  Returns the utility space with the estimated value and issue weights.

4 EVALUATION

4.1 Experimental Setup

To test our agent, the command line application of GENIUS was used to be able to perform and analyze many negotiations in a short time. We did different tests against different agents in different domains and used the outputs of the negotiations in Excel to gather the results. All of the negotiations are done in the Stacked Alternating Offers Protocol with a deadline of 180 rounds.

The results that are presented, are both the utility of our agent and the utility of the opponent agent in every round, the number of times that we reached an agreement, the number of times we had an agreement that was Pareto optimal and the number of times that we reached a Nash product agreement. The utility that is presented, is an average of the time the agent played with domain 1 and the time the agent played with domain 2 in one tournament. This means that if the utility is around 0.4, it usually means that there was no agreement one of the times.

4.2 Results

4.2.1 General tests. First, we tested our complete agent against TheNegotiator [4], Gahboninho [1] and HardHeaded [8] in all possible combinations of the first 7 profiles of the party domain. There are 21 possible combinations in total per opponent per side. Three opponents with two sides means 6 combinations for our agent, which makes the total amount of negotiations 126. The results are presented in the figures 1, 2, 3 and 4.

We can see that our agent performs really well against TheNegotiator: we always get a deal and the utility of us is always higher than the utility of TheNegotiator. But the agent performs worse against the other two agents: both against Gahboninho and HardHeaded it seems hard to get to an agreement, especially against HardHeaded. This can be explained by the fact that our agent also

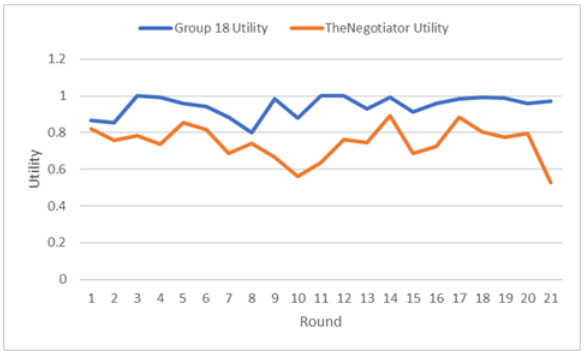

Figure 1: The utility results of our agent against TheNegotiator on all combinations of the party domain

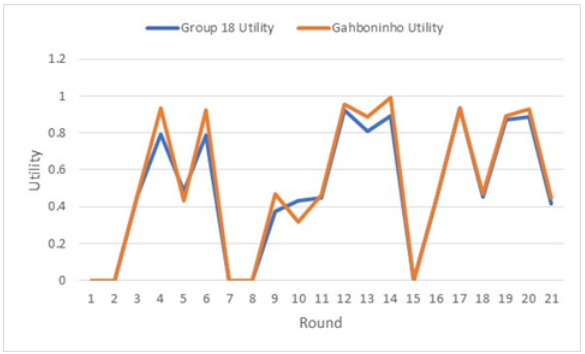

Figure 2: The utility results of our agent against Gahboninho on all combinations of the party domain

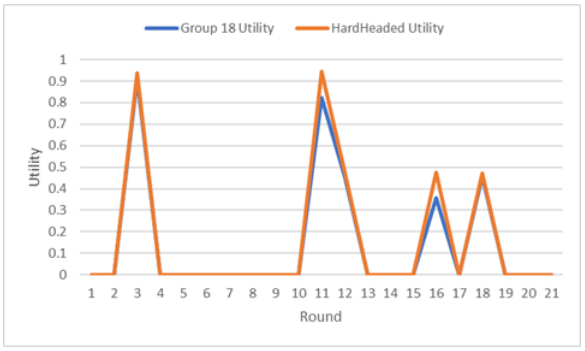

Figure 3: The utility results of our agent against HardHeaded on all combinations of the party domain

does not give in much in the early stages of the negotiation, which makes it tricky to still get a deal at the end. When there is an agreement against one of those two agents, the utility is really close to each other.

Overall, as shown in figure 4, only in around 60-70% of the negotiations, we get an agreement. When we do get an agreement, more than 75% of them are Pareto optimal. This makes sense since our agent really tries to figure out the opponent's utility function

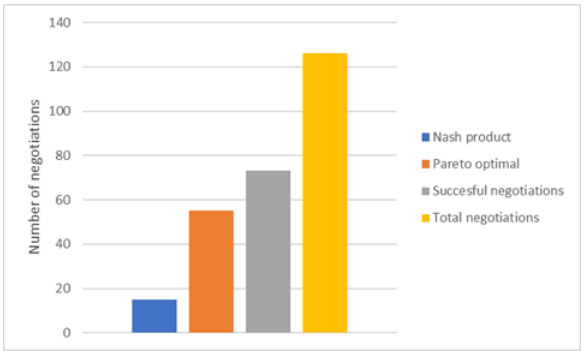

Figure 4: The total number of negotiations on all combinations of the party domain, as well as the number of agreements, the number of Pareto optimal agreements and the number of Nash product agreements

while not giving in too much on our own utility. There are a couple of negotiations where a Nash outcome is reached, but this does not occur often. Reaching a Nash outcome is not one of the goals of our agent, so this makes sense.

4.2.2 Testing with different Acceptance Strategies. For this test, we are using different acceptance strategies to check if our acceptance strategy works better than the others. We test on party domain 3 and party domain 4 against the same agents that we tested against in the previous test. The results, as well as the different Acceptance Strategies used, are shown in figure 5.

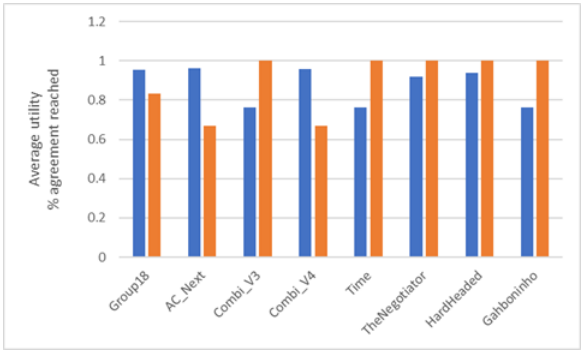

Figure 5: For each acceptance strategy: the average utility and the percentage of agreements reached.

What we can see in this figure is that our average utility is among the highest, but the percentage of agreements reached is a bit lower than for some other agents. This might mean that our acceptance strategy is a bit too stubborn, it could be useful to adjust the parameters alpha and beta in our acceptance strategy to alter this. It still performs really well, most of the acceptance strategies with a higher agreement rate have a lower average utility. There are two acceptance strategies that are potentially better than ours based on this experiment: the ones of TheNegotiator and HardHeaded. With both of those strategies, we lose a couple more negotiations, but we do get more agreements.

4.2.3 Testing with a different Opponent Model. We replaced our own Opponent Model with the HardHeaded Frequency Opponent Model. We tested on party domain 3 and 4 against the same three opponents again. The results are presented in figure 6.

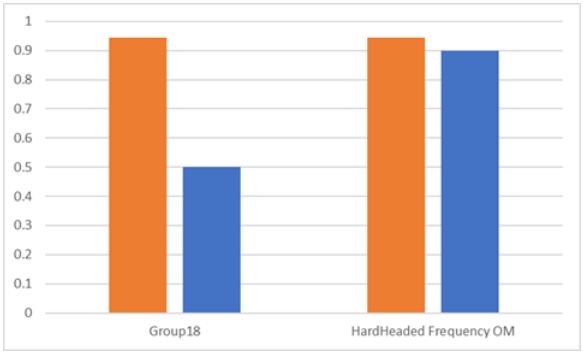

Figure 6: For our own Opponent Model and the HardHeaded Frequency Model: the average utility and the percentage of times an agreement was reached

The average utilities of both the Opponent Models are comparable, but we do see that using our Opponent Model results in much less agreements in comparison to the HardHeaded Frequency Model. This could be caused by the fact that the HardHeaded Frequency Model uses a learning rate parameter, whereas our agent does not, which can cause our agent to be less accurate when using the issue weights to calculate rating of a bid.

4.2.4 Discounted Domains. Our agent was tested against the three other agents, The Negotiator, Gahboninho and HardHeaded, on two discounted domains to see if our agents performs under discounts. We tested on the Travel-C and Ultimatum domains. The results of this test are shown in figure 7, as well as the results on the (undiscounted) party domain as a reference.

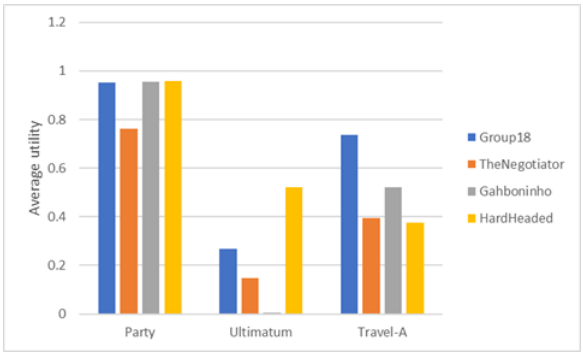

Figure 7: For three domains, the average discounted utility of our agent and the average discounted utility of the other agents against our agent are shown.

The figure shows that our agent performs quite well on the two discounted domains. The discounted utilities are obviously lower than that of the party domain, but our agent finishes first

and second in terms of utility, only losing to HardHeaded on the Ultimatum domain. The utility on the Ultimatum domain could be higher, but the domain really does not suit our agent well with a discount factor of 0.05. Our agent performs better when it has many rounds available and that is not the case in this domain.

4.2.5 Preference Uncertainty. In section 2.5, the preference estimation functionality was introduced. This part discusses the results, both in utility estimation and in performance in a domain with preference uncertainty.

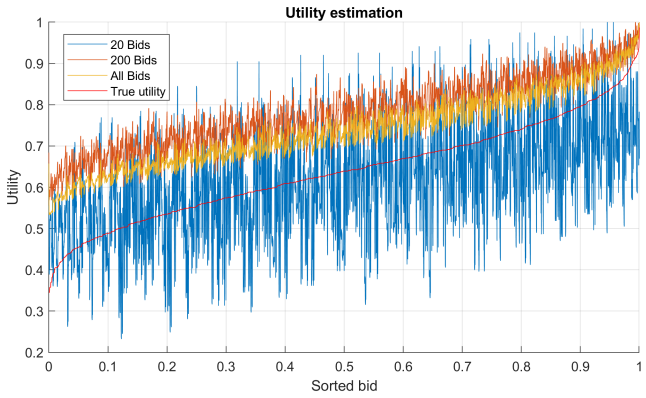

Figure 8: Utility estimation for 20, 200 and all possible bids (3073) on the party domain

In figure 8, estimations of the utility in the party domain are presented for the given ranked list of 20, 200 and all bids. It can be seen that the estimated value and issue weights together result in an utility function for all bids, that reasonably approximates the shape of the true utility function. However, it overestimates the utility. This is most likely caused by the assumption of a linear utility function, when defining the U^e arrays. Also, it can be seen that the estimated utility function is not a monotonically increasing function. This violation also appears when only the provided ordered bids are evaluated. This information could be used to further improve the estimation of the true value and issue weights.

Evaluating the quality of the estimation of the issue and value weights, reveal that the estimation of the issue weights is better by up to a factor seven (table 1). For future improvements, this information can be used to improve the estimation of V^e, by adding more weight to the estimation of w^e.

	20 Bids		*200 Bids*		All Bids	
	V^e	w^e	V^e	w^e	V^e	w^e
Mean error	0.25	0.05	0.16	0.03	0.15	0.02
Standard deviation	0.07	0.02	0.04	0.01	0.02	0.01

Table 1: Error data for N=100

The estimation method was tested on the party domain against TheNegotiator, using the first two preference profiles.The average utility over 100 repeated sessions is shown in figure 9 for 20, 200, all bids and without preference uncertainty. This shows that the

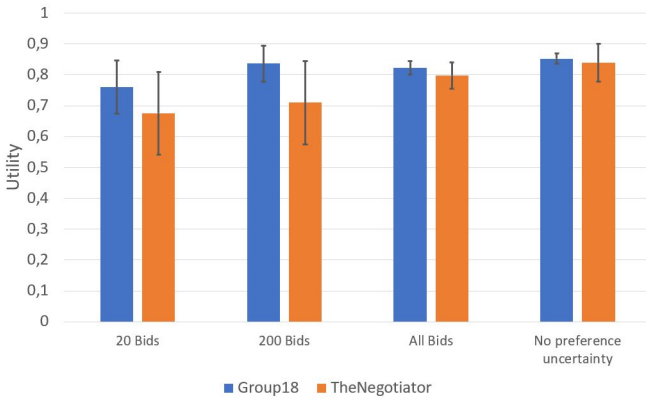

Figure 9: Utility in domains with preference uncertainty

achieved utility by our party is roughly similar to the utility without preference uncertainty, albeit with a higher variance when having fewer bids available.

5 FUTURE PERSPECTIVES

An alternative for one-to-one negotiation is to use a trusted mediator to which each party declares its private preferences. If the goal of the negotiation is to find an optimal outcome for both parties, introducing a mediator is a better option. Parties do not know the real preferences of the opponent; it is very hard to propose bids that are fair for both parties. With this goal in mind, some circumstances in which it is particularly suitable to use a mediator would be domains with a large outcome space or negotiations where a limited amount of time is available. Both situations are difficult for an agent to accurately model the preferences of the opponent.

The parties in the negotiation environment have limited capabilities to achieve a negotiated deal. There are some additional capabilities that would help an agent to achieve a better deal, which can be implemented in the negotiation environment.

First of all, the parties in the negotiation environment are currently limited to one set of strategies (e.g. acceptance, bidding opponent modelling). If an agent would be able to run and evaluate multiple strategies in parallel, the agent can be more flexible in their decisions by choosing the best strategy at a given time.

Secondly, enabling the agent to learn from past negotiation would be another helpful capability to improve negotiation strength. This would be especially useful in negotiation tournaments, where learning from negotiation outcomes, allows agents to adapt to domains and agent matchups over time. This increases the chance that the agent achieves a higher average utility during the tournament.

6 CONCLUSION

Building a negotiation agent over the past months was an interesting project for us to work on. The fact that we could use our own creativity and imagination to build the different components of the agent, while also taking inspiration from many other agents, made it an enjoyable assignment. We decided to build an agent that is searching for a Pareto optimal solution, while not giving

in too much. Our goal was to create a tough but reasonable agent. Using our own creativity to create different functions of time in the Acceptance and Bidding strategies, while also taking inspiration from other agents in the Opponent model created an agent just like that, which can compete and win against many agents included in the GENIUS negotiation environment.

If one would want to use our agent in a real-life negotiation, not many changes need to be made. The different issues and values of the negotiation should be digitalized by the person who is negotiating. This might be hard, because people usually do not think about the importance of issues and values as numbers. Then, a negotiation needs to be started with our agent and the CounterOfferHumanNegotiationParty. The offers of the opponent can be inserted in that party, and the actions that our agent does in return can help someone figure out their strategy in this negotiation. In that way, the code we wrote can actually be used to win negotiations against people in a structured way, while also figuring out there preferences.

REFERENCES

[1] Mai Ben Adar, Nadav Sofy, and Avshalom Elimelech. 2013. Gahboninho: Strategy for balancing pressure and compromise in automated negotiation. In *Complex Automated Negotiations: Theories, Models, and Software Competitions*. Springer, 205–208.

[2] Tim Baarslag. 2014. *What to bid and when to stop.* Ph.D. Dissertation. Delft University of Technology.

[3] Tim Baarslag, Koen Hindriks, Mark Hendrikx, Alexander Dirkzwager, and Catholijn Jonker. 2014. Decoupling negotiating agents to explore the space of negotiation strategies. In *Novel Insights in Agent-based Complex Automated Negotiation*. Springer, 61–83.

[4] ASY Dirkzwager, MJC Hendrikx, and JR De Ruiter. 2013. TheNegotiator: A dynamic strategy for bilateral negotiations with time-based discounts. In *Complex Automated Negotiations: Theories, Models, and Software Competitions*. Springer, 217–221.

[5] Katsuhide Fujita, Reyhan Aydoğan, Tim Baarslag, Koen Hindriks, Takayuki Ito, and Catholijn Jonker. 2017. The sixth automated negotiating agents competition (ANAC 2015). In *Modern Approaches to Agent-based Complex Automated Negotiation*. Springer, 139–151.

[6] Koen Hindriks, Catholijn M Jonker, Sarit Kraus, Raz Lin, and Dmytro Tykhonov. 2009. Genius: negotiation environment for heterogeneous agents. In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*. International Foundation for Autonomous Agents and Multiagent Systems, 1397–1398.

[7] Max WY Lam and Ho-fung Leung. 2017. Phoenix: A Threshold Function Based Negotiation Strategy Using Gaussian Process Regression and Distance-Based Pareto Frontier Approximation. In *Modern Approaches to Agent-based Complex Automated Negotiation*. Springer, 201–212.

[8] Thijs van Krimpen, Daphne Looije, and Siamak Hajizadeh. 2013. Hardheaded. In *Complex Automated Negotiations: Theories, Models, and Software Competitions*. Springer, 223–227.