Artificial Intelligence Techniques

Negotiation Agent Design

Group 4

4004868 Tung Phan
4409159 Francesco Corsini
1369326 Dirk Meijer

January 18, 2015

Contents

1 Introduction

Negotiation is a complex problem and humans are often not the best negotiators. Emotion and the limited processing capabilities of the human brain can prevent us from getting the best results in negotiations. This makes it an interesting area for AIs. A good negotiation agent can aid humans in negotiation, since they are not limited in the same way humans are.
The first step is knowing your own **utility**, a quantification of your preferences within the negotiation domain. This allows us to make offers that are agreeable to ourselves and also inspect offers made by other agents, to base our decision of rejection or acceptation of said offer on. With just this information, it is possible to create a functional agent, although a rather simple one. Such an agent would only make and accept bids that are agreeable to itself. A major issue with this approach is that we don't know in what direction to continue the negotiation, since we only know our own preferences. This means that this strategy might never converge to a solution, depending on the parameters of the scenario.
A good second step, would be to observe the other agents' behavior, to work out what their utilities are. This allows us to find the solution that gives us the highest utility, **within** the solutions we expect to be acceptable to the other parties. To this end we can think of many different ways to estimate whether a bid will be agreeable to another party, and many different ways to generate bids.

The assignment was twofold. Firstly we had to define a negotiation domain for three parties, in which different degrees of conflict can exist. The degrees of conflict were specified as collaborative, moderate and competitive. This will be discussed in section 2.
Secondly we were asked to program a negotiation agent in the Java programming language and with the GENIUS negotiation environment. The agent has to work regardless of the used scenario and has to incorporate the preferences of other agents in its decision making process. This will be discussed in section 3.
Finally, we will show experimental results in section 4 and discuss these results and the performance of our agent in section 5.

2 Domain

The domain had to exist of multiple issues, each with discrete values. Within the domain, we have created three different scenarios that correspond to different degrees of conflict.

Background: A specific land zone has turned out to be a perfect place to build a new neighborhood. Up until now it has been used by the farmer to make his cattle go around. The owner of the land is the municipality.

Parties:

- Farmer

- Construction Company
- Municipality

Issues:

- Segmentation of the land
  - S1: Split 33%, 33%, 33%
  - S2: 100% to Farmer
  - S3: 100% to Construction Company
  - S4: 100% to Municipality
  - S5: 50% to Farmer, 50% to Construction Company
  - S6: 50% to Farmer, 50% to Municipality
  - S7: 50% to Construction Company, 50% to Municipality
- Building a water canal
  - W1: Big canal
  - W2: Medium-sized canal
  - W3: Small canal
  - W4: No canal
- Part of the land reserved for a park
  - P1: Big park
  - P2: Medium-size park
  - P3: Small park
  - P4: No park
- Building functionality
  - F1: Factories
  - F2: Housing
  - F3: Shops
  - F4: Farms

Scenario 1: Competitive The municipality is selling the land. Both the farmer and the construction company want to buy it. But the municipality still wants to retain a zone for welfare structures.
It's going to be difficult for the parties to find an outcome that is agreeable to everyone.

Party	Issue	Preference	Weight
Farmer	Segmentation	S2>S5=S6>S1>S3=S4=S7	0.4
	Water Canal	W2>W3>W1>W4	0.1
	Park	Don't Care	0.0
	Functionality	F4>F2>F3>F1	0.5
Construction Company	Segmentation	S3>S5=S7>S1>S2=S4=S6	0.4
	Water Canal	W2>W3>W4>W1	0.2
	Park	P4>P3>P2>P1	0.1
	Functionality	F3>F2>F1>F4	0.3
Municipality	Segmentation	S1>S5=S6=S7>S2=S3=S4	0.1
	Water Canal	W2>W1>W3>W4	0.3
	Park	P1>P2>P3>P4	0.3
	Functionality	F1>F4>F3>F2	0.2

Scenario 2: Moderate Conflict The municipality is selling the land. Both the farmer and the construction company want to buy it. The municipality wants to support independent farmers, so its preference is more in alignment with the farmer than with the construction company.
This basically reduces to a two-party negotiation, since the preferences for the farmer and the municipality are almost perfectly aligned.

Party	Issue	Preference	Weight
Farmer	Segmentation	S2>S1>S5=S6>S3=S4=S7	0.4
	Water Canal	W2>W3>W1>W4	0.2
	Park	Don't Care	0.0
	Functionality	F4>F2>F3>F1	0.4
Construction Company	Segmentation	S3>S5=S7>S1>S2=S4=S6	0.3
	Water Canal	W4>W3>W2>W1	0.1
	Park	P4>P3>P2>P1	0.1
	Functionality	F3>F2>F1>F4	0.5
Municipality	Segmentation	S2>S1>S6>S5>S3=S4=S7	0.2
	Water Canal	W1>W2>W3>W4	0.2
	Park	P1>P2>P3>P4	0.2
	Functionality	F4>F3>F1>F2	0.4

Scenario 3: Collaborative The municipality is selling the land, they don't really care what happens to it, as long as they no longer have to maintain it. The farmer wants to buy the land and build a canal, to irrigate his other land. The construction company does not necessarily want to buy the land, but they don't want it to turn into a big park, and they want to build houses, shops or factories on the land.
Since the three parties all care about very different things, they should be able to find an outcome for which everyone has a high utility.

Party	Issue	Preference	Weight
Farmer	Segmentation	S2>S5=S6>S1>S3=S4=S7	0.2
	Water Canal	W1>W2>W3>W4	0.8
	Park	Don't Care	0.0
	Functionality	Don't Care	0.0
Construction Company	Segmentation	Don't Care	0.0
	Water Canal	Don't Care	0.0
	Park	P4>P3>P2>P1	0.3
	Functionality	F1=F2=F3>F4	0.7
Municipality	Segmentation	S2=S3=S5>S1>S6=S7>S4	1.0
	Water Canal	Don't Care	0.0
	Park	Don't Care	0.0
	Functionality	Don't Care	0.0

3 Agent Design

3.1 Strategy

The strategy our agent uses is twofold, on one hand we have to decide how to generate bids, on the other hand we have to choose which bids to accept. We decided that the bid generation is more important and should therefore be more complex. The bid generation is based on our expectation of how the other parties will respond to our bid, while keeping our own utility a priority.
The acceptation strategy only looks at our utility for the bid and accepts it if it gives us a utility higher than a time-dependent threshold value. The threshold is lowered each turn, because we are willing to compromise more as we near the negotiation deadline.

3.1.1 Bid Generation

After having looked at several strategies, we decided to go with a pseudo-random strategy. The major advantage of this is that we might reach solutions in the solution space that are unreachable to deterministic agents.
The agent starts out with its best possible bid, which is probably not agreeable to other parties. The agents estimation of the other parties' weights and utilities improves with every message they receive. (see section **??**) and with this information and our knowledge about our own weights and utilities, we incrementally modify our previous bid.
One issue is modified at the time. To select which issue, we look at all our weights for different issues and the estimated weights our opponents have for these issues. We take a pseudo-random approach, taking an issue at random, but first weighing every issue with

$$W_i = \left[\epsilon \times \frac{1/W_{i,0}}{\sum_j 1/W_{j,0}}\right] + \left[(1-\epsilon) \times \sum_{p \neq 0} \frac{W_{i,p}}{P-1}\right] \quad (1)$$

- W_i is the total weight of the issue i for this method.
- ϵ is a number in the range `[0,1]` that becomes smaller as we approach our deadline.
- $W_{i,p}$ is the weight party p assigns to the issue, where $p = 0$ is the agent itself.
- P is the total number of parties.

As we start out with a large ϵ, we change the issues that we care about the least more often, as $1/W_{i,0}$ is largest for those issues. As time goes on, ϵ increases, and the weights for this formula shift more towards what the other parties find important issues. The denominator of both fractions are to normalize the sum of W_i to one.
Once we have selected which issue to change, the process of changing it takes a similarly pseudo-random approach. We generate two possible options for the change. The first option is the value for the issue that gives **us** the highest utility (except for the previously selected value, because we need to change the selected issue)

The second option is selected by generating a change that is "best" for the other parties combined.

$$v_0 = \arg\max_v \{u_0(i = v)\} \tag{2}$$

$$v_1 = \arg\max_v \left\{ \sum_{p \neq 0} W_{i,p} \times u_p(i = v) \right\} \tag{3}$$

- i is the issue selected to be changed.
- v is the value that issue i takes.
- $W_{i,p}$ is the weight that party p assigns to issue i
- $u_p(i = v)$ is the utility that party p assigns to setting issue i to value v.

Now that we have selected two possible alterations to our bid, we weigh these again before selecting one randomly. The weight for each option is as follows

$$V_{i,v_0} = W_{i,0} - \phi \tag{4}$$

$$V_{i,v_1} = 1 - W_{i,0} + \phi \tag{5}$$

- $V_{i,v}$ is the weight assigned to changing issue i into value v.
- $W_{i,0}$ is the weight this agent assigns to issue i.
- ϕ is a small number in the range `[-1,1]`.

This means that if we care a lot about the issue we're about to change, ($W_{i,0}$ is large) there is a high probability that pick our own choice of change. While if we do not care about issue i a lot, we tend to pick the option that benefits the other parties.
The parameter ϕ changes over time, starting out small and increasing as we near the deadline, making it more likely that we compromise and pick an option that benefits other parties.

3.1.2 Acceptation

An opponents bid is accepted if our utility for that bid meets a certain threshold. The threshold starts at starting value T_0 and descreases, reaching our **reservation value** T_∞ at the deadline. The reservation value is the value below which we are not accepting any bid. Any bid we accept or generate will gain us a utility higher than the reservation value, making sure that we don't compromise too heavily.
The decrease is not linear, however. Near the start of the negotiation, we don't want to compromise too much, because we don't have a lot of information on our opponents, and we might miss out if we lower our threshold too quickly. The formula for the threshold is

$$T = T_0 - (T_0 - T_\infty) \times \left(\frac{r}{R}\right)^p \tag{6}$$

- T is the current threshold.
- T_0 is the starting threshold.

- T_∞ is our reservation value.
- r is the current round.
- R is the total number of rounds.
- p is a power that scales the speed of the descent. $p = 1$ is a linear descent, higher values of p make the agent wait before compromising too much.

This acceptation strategy is of course rather simple, but it makes sure that we accept bids that are reasonable enough that we could make them ourselves.

3.2 Implementation

3.2.1 Group4.java

This is the agent itself. The agent

- keeps track of incoming data from other agents through the `receiveMessage` method,
- chooses whether to accept a bid or generate and offer a new bid in the `chooseAction` method,
- creates new bids in the `BidGenerator` class, following the previously described strategy.

```
@Override
public void receiveMessage(Object sender, Action action)
```

This function is the core of the update of our predictive model. Every time an action is recieved, depending on that action (Offer, Acceptance) our model of the others parties is updated. Moreover, this functions register the bids from other parties to be processed later in the chooseAction.

```
@Override
public Action chooseAction(List<Class> validActions)
```

In the chooseAction funtion the agent itself chose what to do next. It analyses the bid previusly recieved and decide if it is worth to be accepted. If not, it generates a new bid thanks to the use of the BidGenerator

3.2.2 Party.java

The Party class is a model of the other parties that our agent has. It consists of weights for each issue, and utilities for each option. The model is updated every time the opponent accepts a bid, rejects a bid and makes a bid.

```
public void updateWithBid(Bid bid,Action action)
```

When an opponent takes any action, our model of that opponent is updated. Making a new bid is treated as accepting that new bid, so there are two cases to be distinguished: Either the opponents finds a bid agreeable or they don't.

```
public Double estimateUtility(Bid bid)
```

This method turns all the IssueModels that are part of this Party model into an estimate of utility for a particular bid. This helps us foresee how agreeable bids are to opponents, and make more efficient bids.

3.2.3 IssueModel.java

This is a sub-part of the Party model, a Party has an IssueModel for each issue in the domain. The IssueModel keeps track of the bids that were rejected and accepted, in order to estimate the weight of the issue, and it iteratively updates the utility with every new bid that comes in.

```
public void updateUtility(ValueDiscrete option, double
    change)
```

If a bid was accepted or rejected, every IssueModel is notified which option for said issue was accepted or rejected. The IssueModel then alters the utility for that option by a small number ($\pm$`change`, order of magnitude 0.1). Then all the utilities are normalized, so that they are in the range `[0,1]`.
The weight of the issue is updated by looking at the variation in how often different options were accepted and rejected. The rationale behind this is that is only certain options are accepted, (the variation is large) this issue will be important, because a wrong option is a deal breaker for the party. On the other hand if the variation is small, all options are rejected and accepted almost as often, the party will not care for this issue very much.
To cut down on processing time, we decided against calculating the actual variation or standard deviation. Instead, we have a counter for each option for the issue, which gets a +1 for an accepted bid and a -1 for a rejected bid. Then we calculate the spread (max - min) and take that as an estimate for the relative variation. The weights for every issue are normalized in the Party class such that the sum equals one.

3.2.4 BidGenerator.java

BidGenerator is a support class that manage the creation of the bids following the previuoly described strategy. The main public mothods caled are:

```
public Bid generateBestOverallBid()
public Bid generateBestBid()
```

These two functions are what the Group4 class call in order to generate the bid.
The first one is called the first time you have to produce a bid, getting the best utility bid for you.
The second is called every other time in order to produce the next bid. It calls all the support functions in order to create the bid. It will be described in more details later

Please note that in order to produce the bid, our strategy is followed. The other parties preferences are weighted and combined together in other to produce a overall "opponent", resulting in the others parties are seen as only one. This has done in order to semplify things by dealing with all the others as one. Obviously, this overall

opponent is created by following the model we slowly built about the preferences of the opponents.
Let's take a closer look at the most important support function in this class:

```
private int getWeightedRandomIssueIndex()
```

This is one of the core function of our strategy. Following a series of steps it return the index of the weighted choice of the weighted inverse random issues.
Brief explanation: first it gets the weighted sum of all the values of all the other parties regarding an issue(weighted based on the issue weight), then normalize the values by dividing them but they overall sum, in order to have them as a percentage; then it inverse them. The same is done with the personal preferences of the agent itself. With both the preferences of the others and ours translated into an inverse percentage regarding the issues, we combine them together following a parameter that changes over time.
At start, this parameter is 75% our preference and 25% their preference, but this parameters slowly change due to multiple factors. TODO

```
private ValueDiscrete getTheirBestValue(int issueNumber)
```

This support function is called when the issue to change the bid on has been decided. In order to understand what the other parties (all of them takes as one entity) want, this function is called. It returns the other parties best value choice based on utility from the model. What it does: given the issue, it sum up all the values utilities for each party given each discrete value. The sum is weighted, meaning that parties that care more about that specific issue have a bigger weight. after this, it only takes the value with the highest sum. This results in the best overall value for the other parties for that specific issue.

```
public Bid generateBestBid()
```

This function is the main function of the generating bid process. // It first call the getWeightedRandomIssueIndex() to decide which issue to change in the next bid. As explained previously, the more the issue is important to us the less it is likely to be chosen; this is even applied to the other parties models, giving higher chance to pick those which the others care less. Then, a value for the chosen issue has to be taken. This is done by first finding our favourite value, then by calling getTheirBestValue() to get the other parties favourite. Then a weighted random dice is thrown to see which value will be selected. The weight depends on how much the agent care about that specific issue, and on how close the deadline is. If the value chosen has already appeared in the last proposed bid, the picking of the value is called again.

4 Test Results

TODO

5 Conclusions and Discussion

TODO