Group 10
(Agent 007)

By
Yiğit Kemal Erinç
&
Boran Pekel
&
Onur Yılmaz

Abstract

In this report, Agent 007's strategies will be analyzed and its performance will be compared against winkyagent, agentgg, and simpleshaop. Later, this report will explain the way Agent 007's performance is improved by enhancing its strategies.

Introduction

While the human negotiator undertakes a negotiation task, automated agents can be used simultaneously. They not only can ease some of the efforts necessary during negotiations but can also support people who are less adequate in a negotiation process. Automated negotiators may be used in the future and may completely replace human negotiators as well. Or as another way of use, these agents can be used to practice before they do a task.

Taking into account these possibilities, being able to create an automated agent which has the capabilities of negotiating will have huge benefits. The success and results of these agents change according to certain environments such as Genius and these agents can be made by using many theories or models. We arranged Agent 007 to coordinate the issues and automate the outcome for multilateral negotiations by applying multilateral negotiation strategies when Genius is present. As a result we believe that our agent is a qualified and good contributor to the current negotiation competitions.

Agent 007 Overview

Agent 007 is an automated negotiation agent that aims to utilize its own utility and come to an agreement with other agents. It follows the "Stacked Human Alternating Offers Protocol", SHAOP while communicating with other agents. In this project, we use Genius Web which is a web-based negotiation framework for bilateral and multilateral negotiation. It makes use of a local web server and the agents exchange messages in this network. These messages include but not limited to Offer, Settings, and Accept.

Before going into the details of how every BOA component works, let us mention how the agent works briefly.

```java
@Override
public void notifyChange(Inform info) {
    try {
        if (info instanceof Settings) {
            init((Settings) info);
        } else if (info instanceof ActionDone) {
            Action lastReceivedAction = ((ActionDone) info).getAction();

            if (lastReceivedAction instanceof Offer) {
                onOfferReceived((Offer) lastReceivedAction);
            }
        } else if (info instanceof YourTurn) {
            Action action = chooseAction();
            getConnection().send(action);
            progress = progress.advance();
        } else if (info instanceof Finished) {
            getReporter().log(Level.INFO,  s: "Final ourcome:" + info);
        }
    } catch (Exception e) {
        throw new RuntimeException("Failed to handle info", e);
    }
}
```

Init

Our agent extends DefaultParty, notifyChange method is invoked by the framework every time an action happens and info is passed to this method. This info can be Settings, ActionDone, Offer, YourTurn or Finished. We take different actions based on this info and it is the main loop of our agent. We get the Settings before the negotiation begins and the first if statement is invoked so we do our initializations in this statement.

We do the initializations in this method, getting the references we need and creating the partial profile. We then get the opponent model instance from Opponent Model class and assign it to an instance variable. Then we pass the current partial profile and progress to opponent model. We also get our partial preferences which contain a list of bids, ordered ascending by preference.

opponentModel.calculateMyPreferences line calculates our issue weights and value weights for each issue, based on this partial preference profile. This is

where our preference elicitation happens. It runs only once since our bid list never gets updated.

```
public void init(Settings info) throws IOException, DeploymentException {
    this.me = info.getID();

    this.progress = (ProgressRounds) info.getProgress();

    this.profileint = ProfileConnectionFactory
            .create(info.getProfile().getURI(), getReporter());
    PartialOrdering partialProfile = (PartialOrdering) profileint
            .getProfile();

    this.opponentModel = OpponentModel.getInstance();
    opponentModel.init(partialProfile, (ProgressRounds) progress);

    List<Bid> orderedBids = new SimpleLinearOrdering(profileint.getProfile())
            .getBids();

    opponentModel.calculateMyPreferences(orderedBids);

    getReporter().log(Level.INFO,
            s: "Party " + me + " has finished initialization");
}
```

ActionDone

When one of our opponents completes their action, an ActionDone message is passed to our agent. We get this message and if it is an Offer, we save it to our list of received bids and update our Opponent Model with this new information.

```
public void onOfferReceived(Offer receivedOffer) {
    receivedBids.add(receivedOffer.getBid());

    if (this.receivedBids.size() > 1) {
        int numberOfReceivedBids = receivedBids.size();
        Bid lastBid = receivedBids.get(numberOfReceivedBids - 1);
        Bid previousBid = receivedBids.get(numberOfReceivedBids - 2);
        opponentModel.updateModel(lastBid, previousBid);
    }
}
```

YourTurn

When it is our turn we generate our next bid, compare it with the opponent's last bid and check if we want to accept. Then we either send an accept or our next bid.

```
public Action chooseAction() throws IOException {
    Bid ourNextBid = this.generateBid();

    if (receivedBids.size() > 1) {
        Bid lastReceivedBid = this.receivedBids.get(receivedBids.size() - 1);

        if (isGood(lastReceivedBid, ourNextBid))
            return new Accept(me, lastReceivedBid);
    }

    return new Offer(me, ourNextBid);
}
```

Finished

Genius framework sends us a Finished message when negotiation is done, we log the outcome and exit

Opponent Model

We are using Frequency Counting technique to estimate the opponent's preferences and also each issue's kernel density(weight). by using this technique we are assuming that the more they value an issue, the less willing they will be to change their value.

To calculate a value's utility for an issue, we divide it by total occurrences of values. For example: Let's say opponent has offered Beer for 120 times and Cocktails for 32 times if the opponent's next bid contains Beer the utility gain from that choice is:

$$U = weightOfDrinks * (numberOfTimesBeverageIsOffered / numberOfBids)$$

In this example, numberOfTimesBeverageIsOffered=120 and number of bids = 152.

The utility contribution of beer choice to this bet is: *weightOfDrinks* * 0.78 in this case.

Next let us look at how we calculate *weightOfDrinks* or any issue weight for that matter. We use the same logic with the previous one but we use the most preferred value for each issue come up with a total sum, here we assume that the opponent will alter the values of least important issues when re-offering.

For example, let's say that Beer was offered 120 times, opponent's most offered music type is Band and it was offered 110 times, their favorite location was Party Room and it was offered 132 times, favorite cleanup is hired help with 72 and printed invitations is 71. We divide the most offered value by the sum of favorite values to normalize them into the 0-1 range.

Weight of drinks: 120 / 505 = 0,2376237624

Weight of music: 110 / 505 = 0,2178217822

Weight of location: 132 / 505 = 0,2613861386

Weight of cleanup: 72 / 505 = 0,1425742574

Weight of invitations: 71 / 505 = 0,1405940594

Notice how they add up to 1.

Opponent's Estimated Utility From a Bid

$U(bid) = \sum_{i=0}^{\#issues} w_i * (\#\ occurrences\ /\ \#\ bids)$ where occurrences refer to the number of occurrences of this value of that issue in bid. #occurrences / # bids is also mentioned as value weight in the code. There is a corresponding number for each value in each issue that indicates how desirable that value is.

Our Estimated Utility From a Bid

$$U(bid) = (\sum_{i=0}^{\#issues} w_i * (\#\ occurrences\ /\ \#\ bids)) / utilityOfBestBid$$

Since our issue and value weights are not explicitly specified to us, we perform Preference Elicitation from a given set of ordered bids. We need a heuristic to evaluate how desirable a bid is for us. To do this, we use the same formula in previous section in a slightly modified way. We calculate our own issue weights and value weights in Preference Elicitation from given set of bids, it is explained in detail in preference elicitation section. Then with these weight values we use the same formula that we used for estimating opponent's utility. But the result of this formula is almost never equal to 1. We wanted to map this value into [0, 1] range so we divide that by utility of the best bid. Best bid is the bid that is at the tail of our list of partial preferences, most desired bid.

Opponent Model Implementation

Opponent Model class is implemented as a Singleton class since there will be only one instance of Opponent Model through the program execution. Many data structures were utilized in the implementation of the Opponent Model. Let us explain what they are used for.

```
private class Weight {
    int count;
    double weight;

    public Weight() {
        this.count = 0;
        this.weight = 0.0;
    }
}
```

A weight class is declared to store count of an issue value and it's weight at the same time.

```
// Map each issue to their weight
private HashMap<String, Double> opponentIssueWeights;
// Map each issue to another map which holds of weights for each value
private HashMap<String, HashMap<String, Weight>> opponentPreferenceMap;
```

We use 2 maps to store opponent's preferences or weights. There are 2 kinds of weights: Issue weights and value weights. Issue weights indicate the importance of issue for example importance of Drinks whereas value weights indicate how preferred each value is in that issue, for example the weight of Beer in drinks.

opponentIssueWeights holds the issue weights and maps each issue name to a double value which is its weight. We do not use the Weight class here because we do not need the counts. We only need the count of most occurring value to calculate issue weight. We get this data from preferenceMaps when we need it.

opponentPreferenceMap maps each issue name to another map, which then maps each value name to its weight. We could use a list of weights instead of the second map but then retrieving the weight of a value would be O(N) where the map implementation provides us O(1) lookup time and better performance.

Calculating Weights

We initialize the weights to 0 when we create the opponent model. Every time we observe the opponent's bid, we update the issue weights and value weights to reflect the new value counts. We parse the bid, increment the count of every value in this bid by iterating over the issues of this bid.

```java
private void updateOpponentPreferences(Bid bid) {
    int numberOfBids = this.opponentBids.size();
    Map<String, Value> issueValues = bid.getIssueValues();

    // Update value weights
    for (String issue : issueValues.keySet()) {
        String value = issueValues.get(issue).toString();
        Weight currentWeight = opponentPreferenceMap.get(issue).get(value);
        currentWeight.count += 1;
        currentWeight.weight = (double) currentWeight.count / numberOfBids;
        opponentPreferenceMap.get(issue).put(value, currentWeight);
    }

    // Update issue weights
    HashMap<String, Integer> maxOccurrences = new HashMap<>();
    for (String issue : opponentIssueWeights.keySet()) {
        HashMap<String, Weight> preferenceMapForIssue = this.opponentPreferenceMap.get(issue);
        maxOccurrences.put(issue, getMostOccurrences(preferenceMapForIssue));
    }

    int sumOfMaxOccurrences = maxOccurrences.values().stream().reduce( identity: 0, Integer::sum);

    for (String issue : opponentIssueWeights.keySet()) {
        double issueWeight = (double) maxOccurrences.get(issue) / sumOfMaxOccurrences;
        this.opponentIssueWeights.put(issue, issueWeight);
    }
}
```

Then we update the issue weights, the formula for issue weights is occurrencesOfMostOccurringElement / sumOfMaxOccurrences where sum of max occurrences is the sum of occurrences of most occurring value for each issue. Calculation was explained in the previous section as well.

Bidding Strategy

While bidding, we take into account opponents' strategy along with the time. We are using the previously mentioned utility estimation algorithm to calculate each bid's utility for the opponent and assuming that the higher the utility is, the higher the probability for them to accept this bid.

We are going to maintain 2 parameters for the desired behavior, one being ***selfishness*** and the other one, ***acceptability.*** The values of these parameters depend on the opponent's behavior and time. As time passes, our agent values acceptability more and becomes less selfish. We also employ a tit-for-tat strategy to determine how selfish or acceptable we are going to be. If the opponent is very selfish like a Boulware agent, our acceptableness will increase at a slower rate with respect to time but if it is similar to Conceder

we will get more acceptable at a faster pace and come to an agreement sooner (hopefully).

Our **first bid** will be the best bid for us, then at each round, we will generate all possible bids, then we will sort them according to their bid score and pick the best bet according to this bid score.

Agent 007 v1.0

We define the bid score as:

$$bidScore = \alpha \times ourUtility + \beta \times opponentUtility$$

where α is the selfishness rate and β is the acceptance rate. We calculate our utility and opponent utility with the help of our Opponent Model.

Now we are going to take a look at how we calculate our selfishness (α) and acceptability (β). Acceptability is a function of the opponent's ***niceness*** and time. We calculate it as follows:

$$acceptability = \log_2(time * 10) * niceness^2/10$$

We multiply the time by 10 since Genius framework provides us time between 0 and 1 and we want to take the log. We select log because we want a slow growth, we multiply it with niceness then divide by 10 in order to map it to a percentage range.

Let us explain how we calculate opponent niceness as well. We calculate niceness by calculating the ratio between the opponent's most recent offer's utility for them and the previous one.

$$niceness = U(receivedOffer) / U(previousOffer)$$

This gives us a number greater than one if the opponent lowered their utility and they have conceded, then we will be increasing our acceptability, otherwise, our acceptability will decrease.

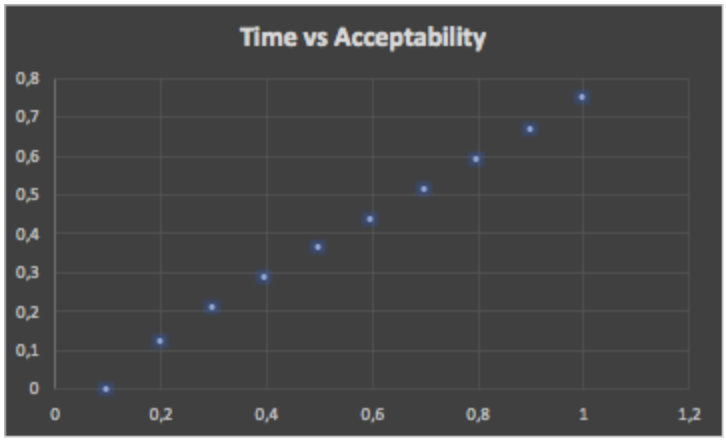
After we define acceptability, selfishness is really simple to define.

$$selfishness = 1 - acceptability$$

You can see the hypothetical acceptability values vs an opponent that is nice, similar to one with a concession strategy and another one that is not so nice.

Acceptability rate vs nice opponent that decreases its utility over time:

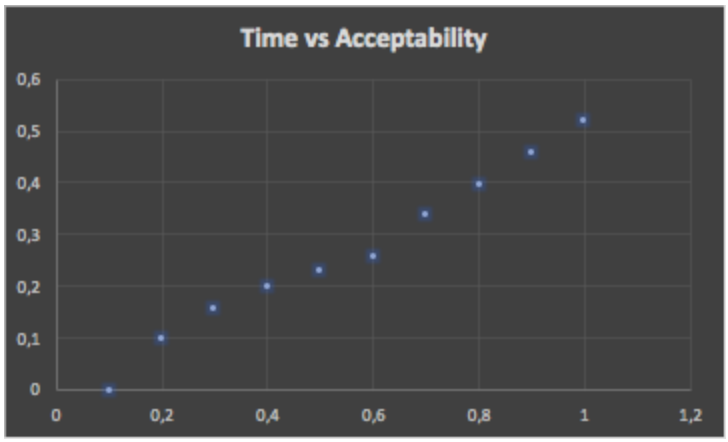
Time	Time * 10	Niceness	Acceptability	Selfishness
0,1	1	1,05	0	1
0,2	2	1,1	0,121	0,879
0,3	3	1,15	0,209611291	0,790388709
0,4	4	1,2	0,288	0,712
0,5	5	1,25	0,362801265	0,637198735
0,6	6	1,3	0,436858663	0,563141337
0,7	7	1,35	0,511640435	0,488359565
0,8	8	1,4	0,588	0,412
0,9	9	1,45	0,666476732	0,333523268
1	10	1,5	0,747433821	0,252566179



Acceptability rate vs a boulware opponent that is not willing to decrease its utility over time:

Time	Time * 10	Niceness	Acceptability	Selfishness
0,1	1	1,00	0	1

0,2	2	1	0,1	0,9
0,3	3	1	0,15849625	0,84150375
0,4	4	1	0,2	0,8
0,5	5	1	0,232192809	0,767807191
0,6	6	1	0,25849625	0,74150375
0,7	7	1,1	0,339689946	0,660310054
0,8	8	1,15	0,39675	0,60325
0,9	9	1,2	0,4564692	0,5435308
1	10	1,25	0,519051265	0,480948735



Notice how slower the acceptability growth is vs a selfish opponent.

Agent 007 v2.0

Our first assumptions were presented in v1.0, after implementing the agent and testing against different agents, we have realized that our agent is caring too much about opponent's behaviour and too little about time. We have revised the formula in a way that time will be more important.

$$acceptability = \log_2(time * 10) * niceness^2/10$$

Formula is now:

$$acceptability = \log_2(time * 10) * niceness / numberOfRounds$$

We were taking the square of niceness to reflect on opponents more, it is no longer the case. We were also dividing by 10 but we noticed that acceptability not be in 0-1 range if it is more than 10 rounds so we now divide by numberOfRounds instead.

Bidding Strategy Implementation

In this section we will explain the code that implements the logic that is discussed in the previous section. We maintain the selfishness, acceptability and opponentNiceness variables in the Opponent Model.

```
private double selfishness = 1;
private double acceptability = 0;
private double opponentNiceness = 0;
```

We update their values in the updateModel function which runs every time we get a new offer. We update the opponent's niceness and acceptability to reflect this change.

```
/**
 * Runs when we receive a new bid
 * Updates preference maps of opponent, niceness score and acceptability
 * @param bid: Recently received (last) bid
 * @param previousBid: Second last bid
 */
public void updateModel(Bid bid, Bid previousBid) {
    if (opponentBids == null) {  // We are just gettin started
        this.opponentBids = new ArrayList<>();
        this.opponentBids.add(previousBid);
        this.opponentBids.add(bid);
        updateOpponentPreferences(previousBid);
        updateOpponentPreferences(bid);
    } else {
        this.opponentBids.add(bid);
        updateOpponentPreferences(bid);
    }

    this.opponentNiceness = getOpponentUtility(bid) / getOpponentUtility(previousBid);

    int round = this.progressRounds.getCurrentRound();
    int numberOfRounds = this.progressRounds.getTotalRounds();

    int time = (int) Math.floor(((double) round / numberOfRounds) * 10);

    this.acceptability = (log2(time) / opponentNiceness) / 10;

    this.selfishness = 1 - acceptability;

    this.reporter.log(Level.INFO,  s: "time: " + time);
    reporter.log(Level.INFO,  s: "niceness: " + opponentNiceness);
    reporter.log(Level.INFO,  s: "selfishness: " + selfishness);
    reporter.log(Level.INFO,  s: "acceptability: " + acceptability);
}
```

We also update the Opponent Model's time (Progress Rounds) information every time it is our turn in notifyChange method.

```
else if (info instanceof YourTurn) {
  Action action = chooseAction();
  getConnection().send(action);
  progress = progress.advance();
  opponentModel.setProgressRounds(this.progress);
```

Then we generate bids in generateBid function which generates all the possible bids in bid space then sorts the bids according to their bid score, recall that bid score for a bid is:

$$bidScore = \alpha \times ourUtility + \beta \times opponentUtility$$

Where α is selfishness and β is acceptability. Then we pick the first bid in this sorted list.

```
public Bid generateBid() throws IOException {
    AllBidsList bidspace = new AllBidsList(profileint.getProfile().getDomain());
    BigInteger numberOfBids = bidspace.size();
    ArrayList<Bid> possibleBids = new ArrayList<>();

    for (int i = 0; i < numberOfBids.intValue(); i++) {
        possibleBids.add(bidspace.get(i));
    }

    // sort all offers by = selfishness × ourUtility + acceptability × opponentUtility
    possibleBids.sort(Comparator.comparingDouble(this::getBidScore));

    return possibleBids.get(0);
}

private double getBidScore(Bid bid) {
    double acceptability = opponentModel.getAcceptability();
    double selfishness = opponentModel.getSelfishness();

    double ourEstimatedUtility = opponentModel.getMyUtility(bid);
    double opponentEstimatedUtility = opponentModel.getOpponentUtility(bid);

    return selfishness * ourEstimatedUtility + acceptability * opponentEstimatedUtility;
}
```

Acceptance Strategy

We are going to accept when the opponent's bid is better than our upcoming bid. As soon as we get the opponent's response, we generate our next bid, then we compare the bids. We chose this as our acceptance strategy because we are employing a tit-for-tat like strategy in bidding and our bids are getting lower in utility for us and more acceptable for the opponent so this is similar to time-based concession. It is also good to accept and stop when opponent's bid is better than our next bid since we are giving up on utility on each round in order to come to an agreement, this makes sure that we are not giving up on utility for more rounds needlessly and it is best to stop when we can agree.

Preference Elicitation Strategy

In version 1, we planned to use frequency counting with same logic that is applied to opponent's preferences. We are still using a similar approach but we modified it to be more reasonable. We are doing the frequency counting from given set of bids but we also take into account, where those values appear because we have the sorted list of bids, which means the later a value appears, more desired it is. Instead of incrementing the count by 1, we increment it with the index of the bid it appears for example if it is in 3rd bid in list, we increase it's count by 3. Then divide by total, which is n * (n + 1) / 2 where n is size of the list.

We have noticed that this reflects our preferences better. We calculate this when negotiation is starting.

```java
/**
 *
 * @param bidOrdering: a sorted list of bids, ascending
 * Fills myPreferenceMap and myIssueWeights
 */
// TODO: As we go down the list, occurrences must be more valuable (count * smth), mention in report
public void calculateMyPreferences(List<Bid> bidOrdering) {
    int numberOfBids = bidOrdering.size();

    // Calculate value weights for each issue

    int[] positionBonus = new int[numberOfBids];
    Arrays.setAll(positionBonus, i -> i + 1);
    for (int i = 0; i < bidOrdering.size(); i++) {
        Bid bid = bidOrdering.get(i);
        for (String issue : bid.getIssues()) {
            HashMap<String, Weight> valueWeights = this.opponentPreferenceMap.get(issue);

            String issueValue = bid.getValue(issue).toString();
            Weight currentWeight = valueWeights.get(issueValue);
            currentWeight.count += positionBonus[i];

            // Total weight to be distributed is n * (n + 1) / 2
            int totalWeight = (numberOfBids * (numberOfBids + 1)) / 2;
            currentWeight.weight = (double) currentWeight.count / totalWeight;
        }
    }

    // Calculate issue weights
    HashMap<String, Integer> maxOccurrences = new HashMap<>();
    for (String issue : myIssueWeights.keySet()) {
        HashMap<String, Weight> preferenceMapForIssue = this.myPreferenceMap.get(issue);
        maxOccurrences.put(issue, getMostOccurrences(preferenceMapForIssue));
    }

    int sumOfMaxOccurrences = maxOccurrences.values().stream().reduce( identity: 0, Integer::sum);

    for (String issue : myIssueWeights.keySet()) {
        double issueWeight = (double) maxOccurrences.get(issue) / sumOfMaxOccurrences;
        this.myIssueWeights.put(issue, issueWeight);
    }

    // Calculate the utility of best bid
    Bid bestBid = bidOrdering.get(bidOrdering.size() - 1);
    this.utilityOfBestBid = calculateBidUtility(bestBid, myPreferenceMap, myIssueWeights);
}
```

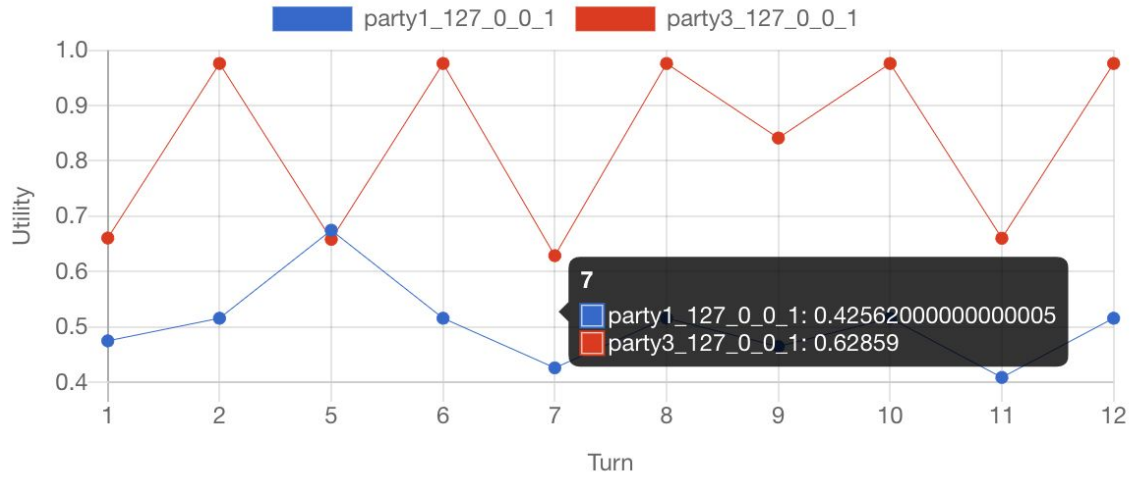
In this portion of the code we calculate our value weights, issue weights and the best bid's utility.

Performance Analysis

Performance of our agent has been evaluated by negotiating with 3 different agents that support the SHAOP protocol: SimpleShaop, AgentGG and WinkyAgent.

SimpleShaop

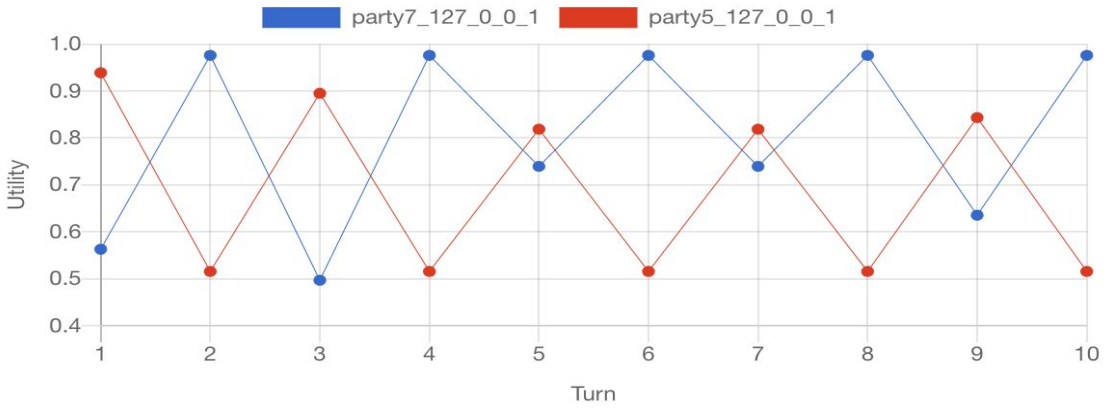
Our selfishness values start with 1.0 as expected and after round 2, it starts to decrease if opponent concedes (acts nice). If the niceness is greater than 1 it means the opponent has been nice to us and trying to come to an agreement, then we decrease our selfishness as well. Our acceptability increases with time as well, if the opponent acts nice. The changes in selfishness and acceptability are observed as expected. Unfortunately we fail to come to an agreement.



AgentGG

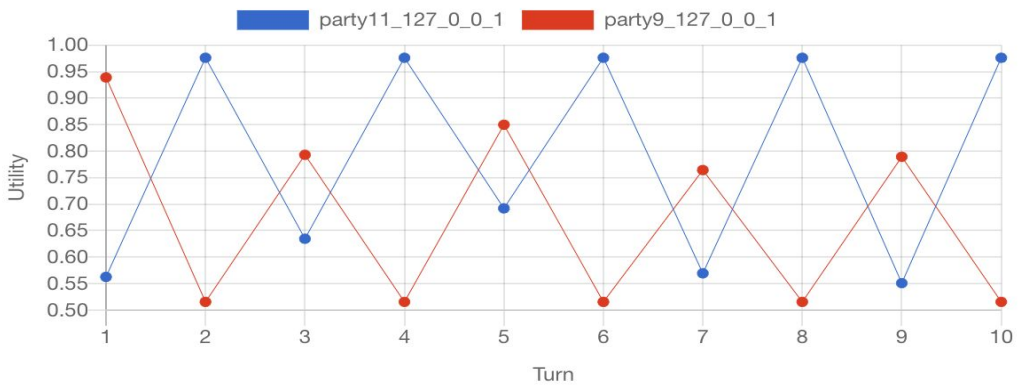
When the AgentGG increases their utility, we increase ours in the next step as well., when they decrease we also decrease. We are checking the previous offers so there is 1 round delay.

```
04-Jun-2020 00:53:46.063 INFO [http-nio-8081-exec-7] tudelft.utilities.logging.ReportToLogger.log time: 1
04-Jun-2020 00:53:46.064 INFO [http-nio-8081-exec-7] tudelft.utilities.logging.ReportToLogger.log niceness: 4.940072202166065
04-Jun-2020 00:53:46.064 INFO [http-nio-8081-exec-7] tudelft.utilities.logging.ReportToLogger.log selfishness: 1.0
04-Jun-2020 00:53:46.064 INFO [http-nio-8081-exec-7] tudelft.utilities.logging.ReportToLogger.log acceptability: 0.0
04-Jun-2020 00:53:46.065 INFO [http-nio-8081-exec-2] tudelft.utilities.logging.ReportToLogger.log high threshold: 1.05
04-Jun-2020 00:53:46.065 INFO [http-nio-8081-exec-2] tudelft.utilities.logging.ReportToLogger.log low threshold: 0.95
04-Jun-2020 00:53:46.066 INFO [http-nio-8081-exec-2] tudelft.utilities.logging.ReportToLogger.log estimated nash: 0.0
04-Jun-2020 00:53:46.066 INFO [http-nio-8081-exec-2] tudelft.utilities.logging.ReportToLogger.log reservation: 0.09137396694214878
04-Jun-2020 00:53:46.077 INFO [http-nio-8081-exec-5] tudelft.utilities.logging.ReportToLogger.log time: 1
04-Jun-2020 00:53:46.077 INFO [http-nio-8081-exec-5] tudelft.utilities.logging.ReportToLogger.log niceness: 0.28646594562993277
04-Jun-2020 00:53:46.077 INFO [http-nio-8081-exec-5] tudelft.utilities.logging.ReportToLogger.log selfishness: 1.0
04-Jun-2020 00:53:46.078 INFO [http-nio-8081-exec-5] tudelft.utilities.logging.ReportToLogger.log acceptability: 0.0
04-Jun-2020 00:53:46.078 INFO [http-nio-8081-exec-1] tudelft.utilities.logging.ReportToLogger.log here
04-Jun-2020 00:53:46.101 INFO [http-nio-8081-exec-7] tudelft.utilities.logging.ReportToLogger.log time: 2
04-Jun-2020 00:53:46.101 INFO [http-nio-8081-exec-7] tudelft.utilities.logging.ReportToLogger.log niceness: 3.019872423945044
04-Jun-2020 00:53:46.102 INFO [http-nio-8081-exec-7] tudelft.utilities.logging.ReportToLogger.log selfishness: 0.9668860183605492
04-Jun-2020 00:53:46.102 INFO [http-nio-8081-exec-7] tudelft.utilities.logging.ReportToLogger.log acceptability: 0.03311398163945081
04-Jun-2020 00:53:46.104 INFO [http-nio-8081-exec-10] tudelft.utilities.logging.ReportToLogger.log high threshold: 1.0
04-Jun-2020 00:53:46.106 INFO [http-nio-8081-exec-10] tudelft.utilities.logging.ReportToLogger.log low threshold: 0.9
04-Jun-2020 00:53:46.106 INFO [http-nio-8081-exec-10] tudelft.utilities.logging.ReportToLogger.log estimated nash: 0.7213782207097715
04-Jun-2020 00:53:46.106 INFO [http-nio-8081-exec-10] tudelft.utilities.logging.ReportToLogger.log reservation: 0.09137396694214878
04-Jun-2020 00:53:46.177 INFO [http-nio-8081-exec-9] tudelft.utilities.logging.ReportToLogger.log time: 2
04-Jun-2020 00:53:46.177 INFO [http-nio-8081-exec-9] tudelft.utilities.logging.ReportToLogger.log niceness: 0.604889245954321
04-Jun-2020 00:53:46.177 INFO [http-nio-8081-exec-9] tudelft.utilities.logging.ReportToLogger.log selfishness: 0.8346804796599878
04-Jun-2020 00:53:46.178 INFO [http-nio-8081-exec-9] tudelft.utilities.logging.ReportToLogger.log acceptability: 0.16531952034001215
04-Jun-2020 00:53:46.178 INFO [http-nio-8081-exec-9] tudelft.utilities.logging.ReportToLogger.log here
04-Jun-2020 00:53:46.204 INFO [http-nio-8081-exec-3] tudelft.utilities.logging.ReportToLogger.log time: 3
04-Jun-2020 00:53:46.204 INFO [http-nio-8081-exec-3] tudelft.utilities.logging.ReportToLogger.log niceness: 1.5651641194299348
04-Jun-2020 00:53:46.204 INFO [http-nio-8081-exec-3] tudelft.utilities.logging.ReportToLogger.log selfishness: 0.8987350603655269
04-Jun-2020 00:53:46.204 INFO [http-nio-8081-exec-3] tudelft.utilities.logging.ReportToLogger.log acceptability: 0.10126493963447311
```



Winky Agent

```
04-Jun-2020 00:55:30.294 INFO [http-nio-8081-exec-8] tudelft.utilities.logging.ReportToLogger.log time: 1
04-Jun-2020 00:55:30.294 INFO [http-nio-8081-exec-8] tudelft.utilities.logging.ReportToLogger.log niceness: 7.825237728090465
04-Jun-2020 00:55:30.295 INFO [http-nio-8081-exec-8] tudelft.utilities.logging.ReportToLogger.log selfishness: 1.0
04-Jun-2020 00:55:30.295 INFO [http-nio-8081-exec-8] tudelft.utilities.logging.ReportToLogger.log acceptability: 0.0
04-Jun-2020 00:55:30.296 INFO [http-nio-8081-exec-2] tudelft.utilities.logging.ReportToLogger.log 1generateBid: 0.8295411235488259
04-Jun-2020 00:55:30.298 INFO [http-nio-8081-exec-4] tudelft.utilities.logging.ReportToLogger.log time: 1
04-Jun-2020 00:55:30.298 INFO [http-nio-8081-exec-4] tudelft.utilities.logging.ReportToLogger.log niceness: 0.5039268383016305
04-Jun-2020 00:55:30.298 INFO [http-nio-8081-exec-4] tudelft.utilities.logging.ReportToLogger.log selfishness: 1.0
04-Jun-2020 00:55:30.298 INFO [http-nio-8081-exec-4] tudelft.utilities.logging.ReportToLogger.log acceptability: 0.0
04-Jun-2020 00:55:30.299 INFO [http-nio-8081-exec-4] tudelft.utilities.logging.ReportToLogger.log here
04-Jun-2020 00:55:30.316 INFO [http-nio-8081-exec-10] tudelft.utilities.logging.ReportToLogger.log time: 2
04-Jun-2020 00:55:30.316 INFO [http-nio-8081-exec-10] tudelft.utilities.logging.ReportToLogger.log niceness: 1.884323693641137
04-Jun-2020 00:55:30.316 INFO [http-nio-8081-exec-10] tudelft.utilities.logging.ReportToLogger.log selfishness: 0.9469305616983635
04-Jun-2020 00:55:30.316 INFO [http-nio-8081-exec-10] tudelft.utilities.logging.ReportToLogger.log acceptability: 0.05306943830163644
04-Jun-2020 00:55:30.316 INFO [http-nio-8081-exec-8] tudelft.utilities.logging.ReportToLogger.log 2generateBid: 0.908730349637613
04-Jun-2020 00:55:30.318 INFO [http-nio-8081-exec-9] tudelft.utilities.logging.ReportToLogger.log time: 2
04-Jun-2020 00:55:30.318 INFO [http-nio-8081-exec-9] tudelft.utilities.logging.ReportToLogger.log niceness: 0.4703628307987067
04-Jun-2020 00:55:30.318 INFO [http-nio-8081-exec-9] tudelft.utilities.logging.ReportToLogger.log selfishness: 0.78739816700611
04-Jun-2020 00:55:30.318 INFO [http-nio-8081-exec-9] tudelft.utilities.logging.ReportToLogger.log acceptability: 0.21260183299389004
04-Jun-2020 00:55:30.319 INFO [http-nio-8081-exec-9] tudelft.utilities.logging.ReportToLogger.log here
04-Jun-2020 00:55:30.335 INFO [http-nio-8081-exec-5] tudelft.utilities.logging.ReportToLogger.log time: 3
04-Jun-2020 00:55:30.335 INFO [http-nio-8081-exec-5] tudelft.utilities.logging.ReportToLogger.log niceness: 2.0368031259939112
04-Jun-2020 00:55:30.335 INFO [http-nio-8081-exec-5] tudelft.utilities.logging.ReportToLogger.log selfishness: 0.922183814405345
04-Jun-2020 00:55:30.335 INFO [http-nio-8081-exec-5] tudelft.utilities.logging.ReportToLogger.log acceptability: 0.07781618559465499
04-Jun-2020 00:55:30.337 INFO [http-nio-8081-exec-10] tudelft.utilities.logging.ReportToLogger.log 3generateBid: 0.8394868451908084
04-Jun-2020 00:55:30.341 INFO [http-nio-8081-exec-1] tudelft.utilities.logging.ReportToLogger.log time: 3
04-Jun-2020 00:55:30.341 INFO [http-nio-8081-exec-1] tudelft.utilities.logging.ReportToLogger.log niceness: 0.43791215837681313
04-Jun-2020 00:55:30.341 INFO [http-nio-8081-exec-1] tudelft.utilities.logging.ReportToLogger.log selfishness: 0.6380638284636679
04-Jun-2020 00:55:30.341 INFO [http-nio-8081-exec-1] tudelft.utilities.logging.ReportToLogger.log acceptability: 0.361936171536332
```



WinkyAgent is similar to AgentGG, we believe both agents are performing tit for tat and our agent is also similar so they generate this zig-zag like graph.

Conclusion

Our group purpose has been achieved based on the process. The agent is set to operate and negotiate with different agents under various conditions. Our agent named The Agent 007 is an adaptive agent. It can adapt complex strategies for different agents. It sees whether an agent is accepting or refraining from conceding. Besides, its adaptive nature is one of the innovative implementations between the agents and this makes it a bit unique. The agent deals very justly among the different agents for the opponents. Besides, it provides the supreme utility to everyone. It aims for win-win situations.

Our experience as a team was considerably good at developing steps of the this agent for a tournament. This agent instructed us numerous information about artificial intelligence and remarkable machine learning techniques. The most crucial part of creating this agent was making strategies or rather coming up with new and different strategies for new and different circumstances. This was something that involved opponent modeling, and we had to learn opponent modeling significantly.

Agent 007 has the skill of being prepared in real-life scenarios in various fields, such as it could be used for future negotiations within different parties. It means that this could be a little device where one can enter the bids, the weights, or more choices to get the maximum utility. The negotiations with

the help of this tiny device would encourage human talks and boost the negotiation speed.

Besides being a useful tool, we do not believe that automated negotiation will replace human negotiators in many fields anytime soon. It is minimal, lacks emotions, and deals with preference uncertainty in a very mechanical and simplified way. Also, it isn't easy to get people to input their preferences to the program since we are not good at coming up with weight values or things like that. We believe that people would not accept the agent's negotiation result when it is not as good as they want the result to be, and deny the consequence if they were to negotiate with their agents. We also think that Genius (especially Web) is very far from being a complete framework with proper documentation, and it is hard to work with, in its current state.