

Human-Agent Negotiation

IAGO Framework

Automated Negotiating Agents Competition (ANAC2020)

Hadasa Nechemia 315552414

Michal Lerner 317801132

Efrat Edri 203678305

Batel Avitan 205691108

Supervisor: Dr. Galit Haim

Table of Contents

1. PROJECT DESCRIPTION	6
2. RELATED WORK	7
3. FUNCTIONAL DESCRIPTION / REQUIREMENTS	8
4. ARCHITECTURE	9
SENDING AND RECEIVING EMOTIONS	9
SENDING AND RECEIVING OFFERS	15
SENDING AND RECEIVING MESSAGES	17
STIMULATING ACTIVE BEHAVIOR	21
5. WORK PLAN	24
6. CLIENT-SIDE	25
ABOUT THE GAME BOARD	25
ABOUT EMOTICONS AND AVATARS	26
ABOUT THE TRADE TABLE	26
ABOUT EXPRESSING PREFERENCES	27
7. SERVER-SIDE	28
7.1 IAGO API:	28



1. Project description

The Human-Agent Negotiation (HAN) competition explores the strategies, nuances, and difficulties in creating realistic and efficient agents whose primary purpose is to negotiate with humans.

It is part of the Automated Negotiating Agent Competition (ANAC) which is an international tournament that has been running since 2010 to bring together researchers from the negotiation community. ANAC provides a unique benchmark for evaluating practical negotiation strategies in multi-issue domains.

Human participants will compete against each submitted agent in three back-to-back negotiations.

An IAGO (Interactive Arbitration Guide Online) agent is a computer player that humans can negotiate with. IAGO agents are designed to act much like humans in many ways and often use emotional tactics to try to one-up their opponents.

Each agent's preferences will be unknown to the opposing side at the beginning of the three negotiations. In this way, agents that do a good job of learning the opponent's preferences will likely outperform agents that do not.

IAGO supports a wide variety of features that have been shown to be critical to realistic human agent communication. These include partial offers, preference elicitation statements, a customizable set of natural language argumentation phrases, and an expressive virtual human agent. The interface also features a full conversational history, the ability to feature non-linear utility structures, and a customizable graphic interface. IAGO exposes a simple API that allows games and agents to be customized with a single template Java class.

It is a platform developed by Dr. Johnathan Mell and Jonathan Gratch at the University of Southern California.

Dr. Johnathan Mell is a specialist in human-computer interaction, negotiation, and artificial intelligence. His academic research focuses on computational negotiation and designing AIs that act in social ways with humans.



2. Related Work

We made use of the following web pages to develop our agent. They all helped us understand better the aim of the game, the given code, and the way to get to the best negotiation.

1. <http://web.tuat.ac.jp/~katfuji/ANAC2020/#overview>

General instructions of the ANAC2020 Competition.

2. <https://myiago.com/IAGO/designTutorial.html>

Agent design tutorial. Explanation of how to build an agent.

3. https://iagoonline.com/IAGOs_fav_jim/?id=test&condition=apva

Demo of an agent and board explanation

4. https://www.youtube.com/watch?v=WFs_4YdK5UM

Video with demo game.

5. http://web.tuat.ac.jp/~katfuji/ANAC2020/cfp/ham_cfp.pdf

Special rules for 2020.

6. <https://myiago.com/IAGO/>

File documentation including support pages, installation instructions, etc.

7. https://myiago.com/IAGO_Doc/

Given code: methods and class explanation.

8. https://www.academia.edu/33744905/The_Misrepresentation_Game_How_to_win_at_negotiation_while_seeming_like_a_nice_guy

Gratch, J., Nazari, Z., & Johnson, E. (2016). The Misrepresentation Game: How to win at negotiation while seeming like a nice guy. In Proceedings of the 2016 International Conference on Autonomous Agents and Multiagent Systems.

from this text we understood the importance of our agent being a “nice guy” and from then on, we stuck to that behavior.

9. <https://myiago.com/about/>

The Likeability-Success Tradeoff: Results of the 2nd Annual Human-Agent Automated Negotiating Agents Competition paper.



3. Functional Description / Requirements

Our agent design consists in being a “nice guy”, as it is proven to be a good strategy in IAGO agents (see related papers #8).

On the one hand, our agent will take control of the negotiation. It will always talk first and it will make offers first but always in a nice way.

On the other hand, it is waiting for fair answers and offers from the human agent. Every action we make, we tell it to the other player, that is another way of being nice and fair.

As a first step, we will always ask for the item the player wants the least. From then on, we will continue depending on the following two things:

The item is the same as mine or the item is different from mine.

The idea is that the player who doesn't care much for that item, will be cooperative and fair with us. We will also let the player know we are being nice, so he will be nice to us too.

- When human's least wanted item is different than mine:
We will offer to take it all and give him ours.
- When human's least wanted item is the same than mine:
We offer to take more of that item.

As always, there might be some edge cases and they are taken into consideration.

- The player does not answer our questions or offers, in that case, we will make an uncooperative offer to make him answer and we will go again to the beginning, asking again for the least wanted item until he answers or until timeout.
- Irrational behavior or answers from the player which are considered as outliers.



4. Architecture

IAGO agents respond to “Events” that occur when the user takes certain actions.

We can divide the action into three main parts: Emotions, Offers, and Messages.

- **Sending and Receiving Emotions**

While emotions may be the most subtle part of designing an agent, it can have a huge impact!

Aggressive agents may be able to make gains in the short term, but will that work in repeated negotiations? We want our agent to be able to react to and send emotions.

When we receive an emotion from the human user, it is because they have clicked on one of the four emoticon buttons on the bottom of their screen.

We will receive a SEND_EXPRESSION Event that contains a description of the emotion.

Often, we may want to respond with a message reacting to their event, or with a similar emotion yourself (mirroring), or even both! To do this, we created our own SEND_EXPRESSION Event and SEND_MESSAGE Event (explained below).

To send an expression, we used the appropriate constructor as described here. As an agent designer, there is more flexibility in emotions, we have access to EIGHT emotions, and we can choose the duration that they will appear on the agent avatar. So we created an Event and added it to our List of responses.

NOTE: The concept **BATNA** which appears in the following code stands for **Best Alternative to a Negotiated Agreement**.

Here is the relevant code in the **Core agent** that deals with responding to an incoming expression with an expression of our own (and a message).

The following code here is from : **RepeatedFavorMessage**

```
class RepeatedFavorMessage extends IAGOCoreMessage implements MessagePolicy
```

```
    Constructor for a positive message.
```

```
    The resulting agent can be either withholding or open, lying or honest.
```



- * @param isWithholding a boolean representing whether an agent is withholding (true if yes, false if no)
- * @param lying a boolean representing whether an agent will tell a BATNA lie
- * @param lb an enum representing how the agent talks about favors

```
private String getEmotionResponse(History history, GameSpec game, Event e) {

    if (e.getType() != Event.EventClass.SEND_EXPRESSION)
        throw new UnsupportedOperationException("The last event wasn't an expression");
        if(e.getMessage().equals("sad") || e.getMessage().equals("angry"))
            return "What's wrong?";
        else if(e.getMessage().equals("happy"))
            return "Well, at least you're happy!";
        else if(e.getMessage().equals("surprised"))
            return "Can you tell me what, did I surprise you?";
        else if(e.getMessage().equals("neutral"))
            return null;
        return "I don't know what face you just made!";
    }
}
```

Return suitable state according to getMessage of user

Description from interface: **MessagePolicy**

Gives the message response to a particular event in a history in the form of an event.

Specified by: **getVerboseMessageResponse**

ePrime - the particular event to respond to

Returns: an event representing the response

```
public Event getVerboseMessageResponse(History history, GameSpec game, Event ePrime)
```

What to do when a player sends an expression? React to it with text and our own expression.

```
if (ePrime.getType() == Event.EventClass.SEND_EXPRESSION )
{
    String str = getEmotionResponse(history, game, ePrime);
```



```

        Event resp = new Event(agentID, Event.EventClass.SEND_MESSAGE,
        Event.SubClass.GENERIC_POS, str, delay);
        return resp;
    }
    else if (ePrime.getType() == Event.EventClass.SEND_EXPRESSION)
    {
        return null; // Disables responding to emotions
    }

```

We determine the incoming type of Event. Since it is an Expression Event:

Event resp;

Then : return resp;

We've chosen to create two response events of our own. These events are then added to the "resp" list (defined earlier) and are executed in order. For more on chaining events, see Part 5.

Each of these events are "core" functionality. In conclusion, no matter what kind of Expression the user sends, and why, this agent will always respond with an Expression and a Message. Which Expression and Message are used are determined by the respective Policies that are called here.

class **RepeatedFavorExpression** extends **IAGOCoreExpression** implements ExpressionPolicy/
Specified by:
getExpression in interface ExpressionPolicy

getExpression(History history)

Causes the VH to show an expression when a triggering event occurs, given a history

Parameters:

history - the history of the negotiation

//

@Override

```

    public String getExpression(History history)
    {

```

```

        Event last = history.getUserHistory().getLast();
    }

```




```

if(last.getType().equals(Event.EventClass.SEND_EXPRESSION)){
    if(last.getMessage().equals("sad") || last.getMessage().equals("neutral"))
        return "sad";
    else if(last.getMessage().equals("happy"))
        return "happy";
    else if(last.getMessage().equals("surprised"))
        return "neutral";
    else if(last.getMessage().equals("angry"))
        return "sad";
    else
        return null;
}

```

Every type like CONFUSION , FAVOR_ACCEPT , FAVOR_REJECT are SubClasses

```
public static enum Event.SubClass extends java.lang.Enum<Event.SubClass>
```

Return an appropriate message according to SubClass :

```

else if (last.getType().equals(Event.EventClass.SEND_MESSAGE))
{
    Event.SubClass type = last.getSubClass();
    switch(type)
    {
        case BATNA_INFO:
            return "happy";
        case BATNA_REQUEST:
            return null;
        case CONFUSION:
            return "sad";
        case FAVOR_ACCEPT:
            return "happy";
        case FAVOR_REJECT:
            return "sad";
        case FAVOR_REQUEST:
            return null;
        case FAVOR_RETURN:
            return "happy";
    }
}

```



```

        case GENERIC_NEG:
            return "sad";
        ...
        default:
            return null;
    }

```

As you can see, when we get the Expression from this Policy, it responds to sad or angry player Expressions with angry Expressions of its own.

Similarly, it "responds" with a neutral expression to happy or surprised player Expressions.

Note that responding with a neutral Expression isn't strictly necessary, since the agent image will already be showing a neutral expression by default.

IAGOCoreExpression Direct Known Subclasses: **RepeatedFavorExpression**

```

public abstract class IAGOCoreExpression implements ExpressionPolicy
{
    protected abstract String getSemiFairEmotion();
    protected abstract String getFairEmotion();
    protected abstract String getUnfairEmotion();
}

```

The following code is from: **abstract class IAGOCoreVH extends GeneralVH**

```

private IAGOCoreExpression expression;
* @param expression Every core agent needs an expression extending CoreExpression.

Public IAGOCoreVH(String name, GameSpec game, Session session, IAGOCoreBehavior
behavior, IAGOCoreExpression expression, IAGOCoreMessage messages)
{...
    this.expression = expression;
...}

getEventResponse(Event e)

```



Agents work by responding to various events.

```
@Override
public LinkedList<Event> getEventResponse(Event e){
    LinkedList<Event> resp = new LinkedList<Event>(); ....
```

What to do when a player sends an expression: react to it with text and our own expression:

```
if(e.getType().equals(Event.EventClass.SEND_EXPRESSION))
{
    String expr = expression.getExpression(getHistory());

    if (expr != null)
    {
        System.out.println(expr);
        Event e1 = new Event(this.getID(), Event.EventClass.SEND_EXPRESSION, expr, 1000,
(int) (500*game.getMultiplier()));

        resp.add(e1);
    }

    Event e0 = messages.getVerboseMessageResponse(getHistory(), game, e);
    else if (e0 != null)
    {
        resp.add(e0);
    }

    return resp;}
}
```

Since it is an Expression Event, we created response events of our own. These events are then added to the "resp" list (defined earlier) and are executed .

For example checking on **totalFair**:

```
Event eExpr = new Event(this.getID(), Event.EventClass.SEND_EXPRESSION,
expression.getUnfairEmotion(), 1000, (int) (700*game.getMultiplier()));
if (eExpr != null)
```



```

{
    resp.add(eExpr);
}
Event e0 = new Event(this.getID(), Event.EventClass.SEND_MESSAGE, Event.SubClass.OFFER_REJECT,
messages.getVHRejectLang(getHistory(), game), (int) (1000*game.getMultiplier()));
    resp.add(e0);

```

- **Sending and Receiving Offers**

Offers are the core of a negotiation. In IAGO, offers are made by specifying the location of objects on a 3 by X board, where X is the number of distinct items. Each of the 3 rows of the board represents items that are currently assigned to your agent, to the human opponent, or are "undecided" by being in the middle of the table. Whenever you craft an offer, you'll have to specify the location of ALL items. When you do this, the IAGO platform will also automatically create a message honestly describing the offer.

For example, if you choose to split the 3 apples with 1 on each side and 1 in the middle, but choose to ignore the oranges entirely, the offer and the automatic message will reflect that: "How about I get one apple, and you get one apple?" (or similar). Note that incorrectly specifying the number of items can cause an exception, so be sure to debug carefully!

The Core Agent included in the source download has two example Behavior Policies. The Conceding Behavior attempts to take the vast majority of the objects and then slowly relinquishes them, while the Fair Behavior leaves most items in the middle and splits them between players two at a time. Look at both Behaviors for examples on how you might create your agent. We'll go over a piece of the Conceding Behavior here:

```

protected Offer getRejectOfferFollowup(History history)
{

    //start from where we currently have conceded
    Offer propose = new Offer(game.getNumIssues());
    for(int issue = 0; issue < game.getNumIssues(); issue++)

```



```

        propose.setItem(issue, concession.getItem(issue));
int[] free = new int[game.getNumIssues()];
int totalFree = 0;
    for(int issue = 0; issue < game.getNumIssues(); issue++)
    {
        free[issue] = concession.getItem(issue)[1];
        totalFree += concession.getItem(issue)[1];
    }
    if(totalFree > 0) //concede free items
    {
        for (int issue = 0; issue < game.getNumIssues(); issue++)
        {
            propose.setItem(issue, new int[] {concession.getItem(issue)[0],
            0,concession.getItem(issue)[2] + free[issue]});
        }
    }
    else
        return null;
    concession = propose;
    return propose;
}

```

In this snippet, the agent determines what the last offer was that was proposed (the variable: "concession"). It then prepares a new offer ("propose"). It begins by reading the number of items that have not been allocated yet (are free, and in the middle of the table). It does this by reading the [1] position of the last proposal table. Then, if there are any free items, it concedes them all to the opponent. Otherwise, there is no concession! The Conceding Behavior has other parts where concessions occur--feel free to check it out.

IMPORTANT: Offers are non-binding in IAGO. Accepting or rejecting an offer has no bearing on the status of items on the game board, nor does it "lock-in" offers. Locking-in of offers may only be done with the FORMAL_ACCEPT Event.

FORMAL_ACCEPT events indicate that the current state of the board is acceptable to the party that sent the Event. Changing the board will invalidate previous FORMAL_ACCEPTs. If both players



send FORMAL_ACCEPTs, then the negotiation is over. FORMAL_ACCEPTs are ignored if the game board is not fully assigned (i.e., there are still items in the middle row).

- **Sending and Receiving Messages**

Messages and discussion are an important part of negotiation, the information gathered when talking with your negotiation partner may have a large bearing on strategy! In IAGO, messages are divided into three broad types: preferences, offer responses, and patterns. There is no formal difference between each of the types, they are all represented by the SEND_MESSAGE event. Preferences are the most precise type of message. Preferences are either statements or questions sent by the user to the agent that express some mathematical relation between issues. For example, the user may say "I like apples more than oranges", or "Do you like pears least?". If your SEND_MESSAGE event contains a non-null Preference object, you can use this class to decode the preference and respond accordingly. The Core Agent responds to questions and statements with truthful answers about its own preferences, but you do not have to (last year's competition winner lied!)

Offer responses are a simple accept or reject statement. They are triggered by two buttons on the left-side of the screen, near the offer table. They send a message indicating acceptance or rejection of an offer. Acceptances and rejections are standard messages but include a Message Code of 100 (for rejections) or 101 (for acceptances).

Pattern responses are accessed through the menu on the right side of the screen, like preferences. But instead of using the relational GUI, pattern responses are simply pre-scripted sentences that the human player may say. Each has a Message Code corresponding to its place in the list in the GameSpec class, to make parsing the incoming messages easier. The Core Agent responds to all messages with some kind of response - look at the switch statements in the Expression and Message Policies.

Let's take a brief look at how the Policy deals with users expressing preferences:

```
Preference p = ePrime.getPreference();
```



```

if (p != null) //a preference was expressed
{
    Relation myRelation;
    if (p.getRelation() == Relation.BEST)

    {
        return "I like " + this.findVHItem(1, game) + " the best!";
    }

    else if (p.getRelation() == Relation.WORST)
    {
        return "I like " + this.findVHItem(game.getNumIssues(), game) + " the least.";
    }

    else
    {
        if(p.getIssue1() == -1 || p.getIssue2() == -1)
            return "Can you be a little more specific? Saying \"something\" is a little confusing.";
        int value1 = game.getSimpleVHPoints().get(game.getIssuePluralNames()[p.getIssue1()]);
        int value2 = game.getSimpleVHPoints().get(game.getIssuePluralNames()[p.getIssue2()]);
        if(value1 > value2)
            myRelation = Relation.GREATER_THAN;
        else if (value2 > value1)
            myRelation = Relation.LESS_THAN;
        else
            myRelation = Relation.EQUAL;
        return prefToEnglish(new Preference(p.getIssue1(),
        p.getIssue2(), myRelation, false), game);
    }
}

```

Here, the agent looks at the last Event ("ePrime"). If this Event had a Preference encoded in it, that means the player just expressed something about their preferences, like: "I like apples best".

This Policy determines the Relation that the player used and tries to truthfully mirror a similar statement about the agent's preferences. In the case that the Relation was a binary one (meaning



>, <, or =), the agent determines which issues were discussed, then compares their values according to its own utility (`game.getSimpleVHPoints`).

Details of the relevant departments for sending a message:

Interface MessagePolicy

- ❖ `getProposalLang(History history, GameSpec game) :`
Returns the language to use when making a proposal, given a history.
- ❖ `getAcceptLang(History history, GameSpec game) :`
Returns the language to use when the VH offer is accepted, given a history.
- ❖ `getRejectLang(History history, GameSpec game) :`
Returns the language to use when the VH offer is rejected, given a history.
- ❖ `getVHAcceptLang(History history, GameSpec game) :`
Returns the language to use when the Player offer is accepted, given a history.
- ❖ `getVHRejectLang(History history, GameSpec game) :`
Returns the language to use when the Player offer is rejected, given a history.
- ❖ `getMessageResponse(History history, GameSpec game) :`
Gives the message response when given a history.
- ❖ `Event getVerboseMessageResponse(History history, GameSpec game, Event e) :`
Gives the message response to a particular event in a history in the form of an event.

public abstract class IAGOCoreMessage implements MessagePolicy

- ❖ `protected boolean getLying(GameSpec game) :`
Returns F - means that the player is not a liar.
- ❖ `protected String getProposalLangChange() :` Returns a message for a change proposal.
- ❖ `protected String getProposalLangFirst() :` Returns message for first proposal.
- ❖ `protected String getProposalLangRej() :` Returns a message rejecting the proposal.
- ❖ `protected String getProposalLangRepeat() :` Returns a message for a response of
- ❖ another proposal.



protected static String prefToEnglish(Preference preference, GameSpec game) :

Returns a message that is translated from a phrase to a text.

```
{
    String ans = "";
    if (preference.isQuery())
        ans += "Do you like ";
    else
        ans += "I like ";
    if (preference.getIssue1() >= 0)
        ans += game.getIssuePluralNames()[preference.getIssue1()] + " ";
    else
        ans += "something ";
    switch (preference.getRelation())
    {
        case GREATER_THAN:
            ans += "more than ";
            if (preference.getIssue2() >= 0)
                ans += game.getIssuePluralNames()[preference.getIssue2()];
            else
                ans += "something else.";
            break;
        case LESS_THAN:
            ans += "less than ";
            if (preference.getIssue2() >= 0)
                ans += game.getIssuePluralNames()[preference.getIssue2()];
            else
                ans += "something else.";
            break;
        case BEST:
            ans += "the best";
            break;
        case WORST:
            ans += "the least";
            break;
        case EQUAL:
            ans += "the same as ";
    }
}
```



```

        ans += game.getIssuePluralNames()[preference.getIssue2()];
        break;
    }

    ans += preference.isQuery() ? "?" : ".";

    return ans; }

```

public class RepeatedFavorMessage extends IAGOCoreMessage implements MessagePolicy:

protected Event getFavorBehavior(History history, GameSpec game, Event e) :

Returns an event in response to a proposal based on history

SEND_MESSAGE - allows for an agent to send or receive a string to be displayed in the chat log.

```

{
    if (lb != LedgerBehavior.NONE && utils.isImportantGame())
        return new Event(agentID, Event.EventClass.SEND_MESSAGE, Event.SubClass.FAVOR_REQUEST,
            "Excuse me, but there is a super-valuable item here for me! If you accept my favor request, I'll promise
            to pay you back in a future round!", (int) (1000*game.getMultiplier()));

    else if (lb != LedgerBehavior.NONE && utils.getLedger() > 0)
        return new Event(agentID, Event.EventClass.SEND_MESSAGE,
            Event.SubClass.FAVOR_REQUEST, "Excuse me, but you still owe me a favor. Accept my favor request,
            so you can pay me back!", (int) (1000*game.getMultiplier()));

    else if (lb != LedgerBehavior.NONE && lb != LedgerBehavior.BETRAYING &&
        utils.getLedger() < 0)
    {
        utils.modifyVerbalLedger(1);
        return new Event(agentID, Event.EventClass.SEND_MESSAGE,
            Event.SubClass.FAVOR_ACCEPT, "I think I still owe you a favor! Let me just pay that
            back for you.", (int) (1000*game.getMultiplier()));
    }
    return null;
}

```

public RepeatedFavorMessage(boolean isWithholding, boolean lying,

RepeatedFavorBehavior.LedgerBehavior lb) :



Constructor for a positive message. The resulting agent can be either withholding or open, lying or honest.

* param isWithholding: a boolean representing whether an agent is withholding (true if yes, false if no)

* param lying: a boolean representing whether an agent will tell a BATNA lie

* param lb: an enum representing how the agent talks about favors

- ❖ `public String getWaitingLang(History history, GameSpec game) :`
Returns a message when there is no response from the agent for a few seconds.
- ❖ `protected String getEndOfTimeResponse() :`
Returns a message with a warning that time is running out.
- ❖ `protected String getSemiFairResponse():`
Returns message that the proposal is almost fair.
- ❖ `protected String getContradictionResponse(String drop):`
Returns a message that the proposal contradicts the previous proposal.

What to do when the player sends a message (including offer acceptances and rejections):

```
public abstract class IAGOCoreVH extends GeneralVH
{ ...
    param messages: Every core agent needs a message extending CoreMessage.
    private IAGOCoreMessage messages;
    ...
    public LinkedList<Event> getEventResponse(Event e)
    if(e.getType().equals(Event.EventClass.SEND_MESSAGE))
    {
        Preference p;
        getPreference() - Provides the preference stored in the Event
        if (e.getPreference() == null)
        { p = null; }
        else
        {
```



```

        p = new Preference(e.getPreference().getIssue1(), e.getPreference().getIssue2(),
        e.getPreference().getRelation(), e.getPreference().isQuery());
    }
    a preference was expressed
    if (p != null && !p.isQuery())
    {
        utils.addPref(p);
        if(utils.reconcileContradictions())
        {
            we simply drop the oldest expressed preference until we are reconciled.
            LinkedList<String> dropped = new LinkedList<String>();
            dropped.add(IAGOCoreMessage.prefToEnglish(utils.dequeuePref(), game));
            int overflowCount = 0;
            while(utils.reconcileContradictions() && overflowCount < 5)
            { dropped.add(IAGOCoreMessage.prefToEnglish(
                utils.dequeuePref(), game));
            overflowCount++; }
            String drop = "";
            for (String s: dropped)
                drop += "\"" + s + "\", and ";
            remove last 'and'
            drop = drop.substring(0, drop.length() - 6);
        }
        ....
        if (!((COMBehavior) this.behavior).getWasSecondOfferMade())
        { if (((COMBehavior) behavior).getFirstOfferGenerosity() &&
            ((COMBehavior) behavior).getWasFirstOfferMade())
            {
                this means both LW items are the same, and we took more of it
                String str = "It seems that our least wanted item is the
                same! As a favor to you, I took more of this item in this round.";
                Event e4 = new Event(this.getID(),
                Event.EventClass.SEND_MESSAGE, str,
                (int) (1 * 2000 * game.getMultiplier()));
                resp.add(e4);
            }
        }
    }

```



```

else if (((COMBehavior) behavior).getWasFirstOfferMade())
{
    String str = "As a favor to you, I will take your least
wanted item, and give you mine.";
    Event e4 = new Event(this.getID(),
    Event.EventClass.SEND_MESSAGE, str,
    (int) (1 * 2000 * game.getMultiplier()));
    resp.add(e4);
}
}

else
{
    String str = "What do you say? we are trying to take your
least valuable item again.";
    Event e4 = new Event(this.getID(), Event.EventClass.SEND_MESSAGE, str,
    (int) (1 * 2000 * game.getMultiplier()));
    resp.add(e4);
}

Event e5 = new Event(this.getID(), Event.EventClass.SEND_MESSAGE,
    Event.SubClass.NONE, messages.getProposalLangFirst(),
    (int) (2000 * game.getMultiplier()));
    resp.add(e5);

...}

```

- **Stimulating Active Behavior**

Because of the Event system, IAGO agents are, by nature, passive. They cannot take unprompted action. The *delay* parameter tries to solve this problem. Each delay will cause the agent to pause before taking that action.

For example, we first send an expression and only then a message because an agent, like humans, can't do two things at the same time. Between them we waited for one "delay".

```

Event resp;
\
    if (value != -1) {
        resp = new Event(agentID, Event.EventClass.SEND_MESSAGE, sc, value, str, delay);
    }
}

```



```

    } else {
        resp = new Event(agentID, Event.EventClass.SEND_MESSAGE, sc, str, delay);
        if (relation != null) {
            resp.encodePreferenceData(new Preference(issue1, issue2, relation, isQuery));
        }
        return resp;
    }
}

```

The method `getDelay` which appears in the Event Class is in charge of this.

```
public int getDelay()
```

Provides the length of time to wait before executing the Event, in milliseconds.

Returns: The length of time to wait, in milliseconds.

Another example from IAGICoreVH which asks the agent to wait 90 seconds before it continues:

```

// At 90 second, computer agent will send prompt for user to talk about preferences

if (e.getMessage().equals("90") && this.getID() == History.OPPONENT_ID)
{
    String str = "By the way, will you tell me a little about your preferences?";
    Event e1 = new Event(this.getID(), Event.EventClass.SEND_MESSAGE,
Event.SubClass.PREF_REQUEST, str, (int) (1000*game.getMultiplier()));

    e1.setFlushable(false);
    resp.add(e1);
}
return resp;
}

```

Finally, a note about the `OFFER_IN_PROGRESS` Event. This Event is used in two senses. The first is to use it as a signal. Whenever a human user begins moving items on the board to craft an offer, an `OFFER_IN_PROGRESS` Event is triggered. Your agent can choose to respect this by refraining from interrupting the user and sending its own offers or messages. No one likes a pushy person! On the other hand, you could choose to design an agent that DOES interrupt, especially if it thinks the incoming OFFER is sure to be an unpleasant one.



You may also decide to send your own OFFER_IN_PROGRESS Event to the user. Once triggered, this Event will cause the human player's screen to show flashing dots in the text log, to indicate that you are busy. These dots will automatically disappear once your agent takes another action. Let's look at one of the longer sequences of Events in the Core Agent:

```
Event eExpr = new Event(History.VH_ID, Event.EventClass.SEND_EXPRESSION,
    expression.getUnfairEmotion(), 2000, 0);
    resp.add(eExpr);

Event e0 = new Event(History.VH_ID, Event.EventClass.SEND_MESSAGE,
    messages.getVHRejectLang(getHistory(), game), 0);
    resp.add(e0);

Event e3 = new Event(History.VH_ID, Event.EventClass.SEND_OFFER,
    behavior.getNextOffer(getHistory()), 700);
if(e3.getOffer() != null)
{
    Event e1 = new Event(History.VH_ID, Event.EventClass.OFFER_IN_PROGRESS, 0);
    resp.add(e1);

    Event e2 = new Event(History.VH_ID, Event.EventClass.SEND_MESSAGE,
        messages.getProposalLang(getHistory(), game), 3000);
    resp.add(e2);

    this.lastOfferSent = e3.getOffer();
    resp.add(e3);
}
```

Here, the player has just sent an offer that the agent has judged to be less than fair. In response, the agent first waits for 2 seconds, then creates an Expression as dictated by its Expression Policy. It sends that simultaneously with language from its Message Policy indicating rejection. Then, 700ms later, it computes its next offer. Assuming the agent returns an offer, it then signals the GUI to activate the Offer In Progress dots, sends the offer, and some proposal text, following a 3 second delay.

NOTE ABOUT CODE ORGANIZATION: The files we've added got a name starting with the letters COM which stand for College of Management according to the Code Organization rules.



5. Work plan

Sending and Receiving Emotions - Efrat Edri

Sending and Receiving Offers - Batel Avitan

Sending and Receiving Messages - Hadassa Nechemia

Simulating Active Behavior & Project Book - Michal Lerner

6. Client-side

There is no client side of IAGO beyond a web browser. All content is delivered through JavaScript, HTML5, and CSS. This is what makes it easy to deploy through online subject recruitment tools like Amazon's MTurk—no client-side installation is required.

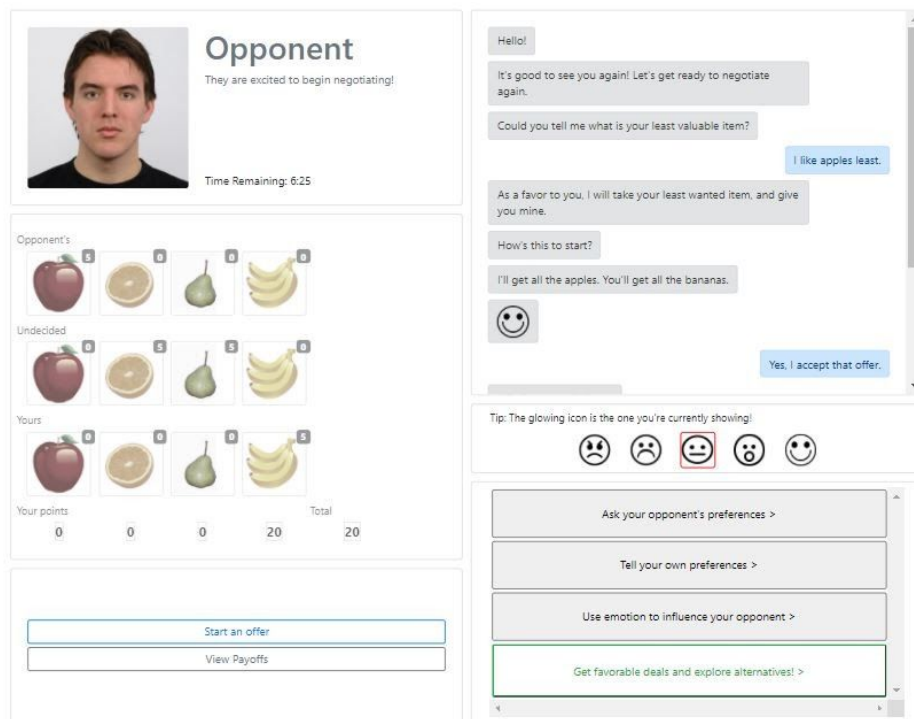
The IAGO Negotiation platform provides a front-facing GUI for the human-participants (see Fig. below). In particular, this feature allows subjects to be recruited using online platforms, such as Amazon's Mechanical Turk (MTurk). Additionally, IAGO provides the features necessary for simulating the characteristics of human negotiation. These include an expanded set of channels for communication between both sides of negotiation, such as by sending text, expressing preferences, and transmitting emotions. Text is transmitted through a set of pre-selected utterances, and emotions are transmitted by selecting from a variety of prototypical “emojis” within the interface. These channels are in addition to the traditional methods supported by agent-agent negotiation platforms, such as exchanging offers. IAGO also allows offers to be sent that do not involve all the items in the negotiation (“partial offers”). These features of IAGO mean that it provides a platform to address the basic features that intelligent negotiating agents require. It provides information that allows for robust user modeling and allows multiple channels for communicating in different ways. IAGO provides information that agents require to reason about their own preferences and allows them to pursue a number of more complex strategies that require specific features (such as partial offers).



- **About the Game Board**

The chat log is on the right, and a picture of your partner on the left. In the bottom half, there is a trade table and buttons. In the game, you can send messages and questions to your opponent. You can also move items around on the game board, and send offers.

Everything you do will appear in the chat log on the right side of the screen so you can look it over. In the game, your agent can send messages and questions to your opponent. It can also move items around on the game board, and send offers.



- **About Emoticons and Avatars**

The buttons you see below can be used to send emoticons in chat. Your avatar will change facial expressions when you send an emoticon.









- **About the Trade Table**

With the trade table your agent can send offers to your partner. You cannot interact with this interface and must watch your agent do so for you. Your agent can click any item to pick it up, then click again to place it. Nothing is sent until clicked "Send Offer".

It can also accept or reject partial offers that your partner sends it. These offers aren't binding, but are helpful in building towards a full offer.

Pressing "Formal Accept" is only possible if ALL items are either on its side or your partner's. If they both agree, the game is finished.

Opponent's	 0	 0	 0	 5
Undecided	 0	 0	 0	 0
Yours	 5	 5	 5	 0
Your points	40	20	15	0
Total	75			



- **About Expressing Preferences**

During the negotiation you can express your own preferences for items and ask your opponent specific questions about their preferences. The available symbols are: "less than", "more than", "equal", "least" and "best"



7. Server-Side

In order to run IAGO, the best way is to host it on a web server, such as Apache Tomcat.

Apache Tomcat (called "Tomcat" for short) is an open-source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and WebSocket technologies. Tomcat provides a "pure Java" HTTP web server environment in which Java code can run.

Tomcat is developed and maintained by an open community of developers under the auspices of the Apache Software Foundation, released under the Apache License 2.0 license.

In our project we also used Servlet technology.

Servlet is a Java-based server-side technology, which is one of the Java Enterprise Edition application programming interfaces.

The Java Servlet specification defines how a software component written in the Java language and running on the server side can respond to requests received from the client (for example, GET and POST HTTP requests).

Servlets are Java classes, which in principle can communicate over any server-client protocol, but the most common use of Servlets is with the HTTP protocol.

Thus, the word "Servlet" alone is usually used in the context of "HTTP Servlet". A servile is an object that receives a request and creates a response based on that request.

Typically, the content created by Servlets are HTML pages, but these can also be data in other forms, such as JSON, XML, and more.

Servlets can store state information within session variables, across multiple server transactions, by using cookies or URL rewriting.

7.1 IAGO API:

IAGO is a platform and programming API. It serves as a testbed for Human-Agent negotiation specifically. IAGO is a web-based servlet hosting system that provides data collection and recording services, a human-usable HTML5 UI, and an API for designing human-like agents.

