Modelling an Artificial Agent for Liar's Dice with ACT-R

Andrei Miculita (s4161947): UI Designer
Oscar de Vries (s2713748): Modeller
Tommaso Parisotto (s3866033): Programmer

Cognitive Modelling: Complex Behaviour

University of Groningen

November 2, 2020

Contents

This app was made as an assignment for the course: Cognitive Modelling: Complex Behavior at the University of Groningen. The code for this application can be found at `https://github.com/AndreiMiculita/pyliarsdice`.

1 Introduction

This research follows the development of a software application for playing the game of *Liar's Dice* against one or more human-like behaving artificial agents. It was originally designed as an iOS app made with the Swift language and the ACT-R system (Taatgen & Weerd, 2016)[1]. However, due to the coronavirus quarantine measures, the project was redesigned in Python 3[2] to be run on any platform running the Python interpreter and the project's dependencies. The artificial agent acts as an opponent for the human player and makes decisions using a cognitive model based on ACT-R (Anderson, Matessa, & Lebiere, 1997). The goal of the project is to make such an agent play the game in a human-like fashion, as well as making the agent fun to play against. This means it should be an opponent which is not too easy and not too hard to play against. Furthermore, a less challenging opponent (based on random choices) can also be chosen as a basis of comparison, as well as being a nice opponent for inexperienced player to discover the game.

2 The game of Liar's Dice

Before discussing the implementation of the application and the agent's behavior, this section describes the mechanics of the game. Liar's Dice is a multiplayer game in which two or more players compete. There is no limit to the number of players, but we decided to limit our application to 1 to 4 other players, as more players would make it difficult to display all the information on the screen. The idea of the game is that players make bids about the dice that are on the table in total, while only having information about the dice under their own cup and the total number of dice in play. Hence, this is a game of incomplete information, where players make assumptions on the dice other players have concealed under their cups. A game consists of a number of rounds, and in each round the players have to make a bid that is higher than the player before them. A player can decide to not believe the bid of the opponent before, calling that player a 'liar', and then all the other players decide whether they believe the bid or they do not. Whoever is correct can remove one of the dice from their cup. The goal is to be the first player to lose all of their dice.

Many different variations or versions of Liar's Dice exist. Here we describe the version that is implemented in our application.

2.1 Materials needed

The materials needed for the game are:

- **2 or more players**
- **5 dice per player**
- **1 cup per player**

2.2 Description of the game

The game consists of multiple rounds. In every round, all players roll their dice under the cup, such that the dice are only visible to them. Then, one of the players (in the first round, a random

[1]More information can also be found at `http://act-r.psy.cmu.edu/`

[2]See `https://www.python.org/`

player, and in the following rounds, the player who lost the previous round) starts with a bid. In a clockwise direction, every player either raises the bid or calls it a lie.

2.2.1 Action types in a turn

With the exception of the first turn in the round, every player has a choice between two types of actions:

- **Overbid the previous player:** overbidding means that either the number of dice must be higher than the bid of the previous player, or the value of the dice is raised while maintaining the same number of dice as the previous bid.
  Examples:

Correct:	3 × ⚁	→ 4 × ⚁	Higher number of dice
Correct:	3 × ⚁	→ 3 × ⚃	Higher value with the same number of dice
Incorrect:	3 × ⚁	→ 2 × ⚅	Lower number of dice
Incorrect:	3 × ⚂	→ 3 × ⚁	Lower value with the same number of dice

Joker dice: in this version of the game there is a *joker die*, which is every die with value 1 (in our case depicted as a star). This is the most challenging aspect of this version of the game, as a joker die counts as *an instance of all the other die values*, meaning that whenever someone's hand would be:

★ ⚁ ⚂ ⚄ ⚄,

this player would have two 2's, two 3's and three 5's in his hand. This impacts bidding heavily, as it increases chances for every number on the table.

- **Overbidding with Joker dice:** a joker die counts as *double the number of its occurrences* when it is used as a bid. This is because every other value has a double chance of appearing on the table (since a joker also counts towards their number), while the joker itself is only based on the probability of appearing itself.
  Examples:

Correct:	3 × ⚁	→ 2 × ★	Higher number of dice
Correct:	2 × ★	→ 4 × ⚁	Higher value with the same number of dice
Incorrect:	4 × ⚅	→ 2 × ★	Lower value with the same number of dice
Incorrect:	3 × ★	→ 5 × ⚁	Lower number of dice

Bluffing: A bid can be made independently of the roll of dice under the cup. This means bluffing is a substantial part of this game and can be strategically used to cause other players to lie.

- **Calling the previous player a liar:** whenever a player believes that the bid of the previous player is a lie (i.e. the number of dice of the bid is higher than the number of dice on the table), you can call that player a liar.

When a player is calling the previous player a liar, every player has to specify whether he or she believes that the number of dice of the bid is on the table or not, in a clockwise direction. Then, everyone lifts their cup revealing the dice and they are counted and compared to the bid.

If the number of the bid is *equal or less than* the total number of dice of that value, the bid is correct. However, if the number of the bid is *greater than* the total number of the dice of that value, the bid is a lie. All the players who are correct lose one die.

After the round has ended, a new round starts with the remaining number of dice for each player. The player who lost the last round will start the bidding for this new round. For multiple players, this will be either the player who was correctly called a liar, or the player who incorrectly called another player a liar.

2.3 Goal of the game

After every round, all the players who were correct lose a die. The goal is to be the first to lose all of the dice from your cup. In some cases multiple players can reach 0 dice simultaneously, meaning there are multiple winners.

3 Implementation

This section describes the design of the software of this game.

3.1 Program Structure

The application was initially developed in Swift. While we ported the project halfway through its realization in Python, we decided to keep the object-oriented paradigm and the state machine we originally designed for the game.

3.1.1 Classes

Every component of the game is represented by an object. Each player is represented by their own instance of the *Player* class, which stores the number of dice the player is playing with and a list of integers between 1 and 6 representing their 'hand'. The player class also provides a number of methods to manage the dice that are called throughout the game. The human player is represented as an instance of this class as well and is distinguished from the opponent players by a parameter that indicates the strategy, which is either denoted as 'human', or one of the artificial agents strategies. These are 'random' and 'model', depending if the agent is playing either at random or following directions from the model respectively. The *Game* class has the role of initializing each component and managing the procedures of the game. The class stores a few methods to support the game, resolve actions that require multiple players to interact, and evaluate probabilities that are sent to the model. It is also responsible for the management of the state machine that runs the game and determining its phases.

3.1.2 State Machine

The stages of each round of the game are represented by a state machine with 5 states:

- The **Start** of the round, when players roll the dice and the first player is decided. On the very first round of the game this is determined randomly, but afterwards this will be either the player who was correctly called a liar, or the player who incorrectly called another player a liar.
- During the **Doubting** phase, the players decide if they want to overbid the current bid or call the opponent who made the previous bid a liar. This choice determines which of the following two states the state machine goes into.
- If a player decides to call a liar, the round ends and the game proceeds to the **Resolving Doubt** phase. Here the doubt is resolved, meaning that all the other players must state as well whether they believe the bid or not. All the dice are revealed, and the game class manages the removal of a die of the players who were correct this round.
- If a player believes a bid (or at the first turn of a round), the **Bidding** phase is executed. Here the player states the overbid (or state any bid if it's the first round); if the player is controlled by a model, a request is made to evaluate its response based on the current state of the game.
- The **End** of the game is when at least one of the players has no remaining dice. The winner(s) is/are announced.

A schematic view of the program structure and state machine can be found in the in Figure 1.

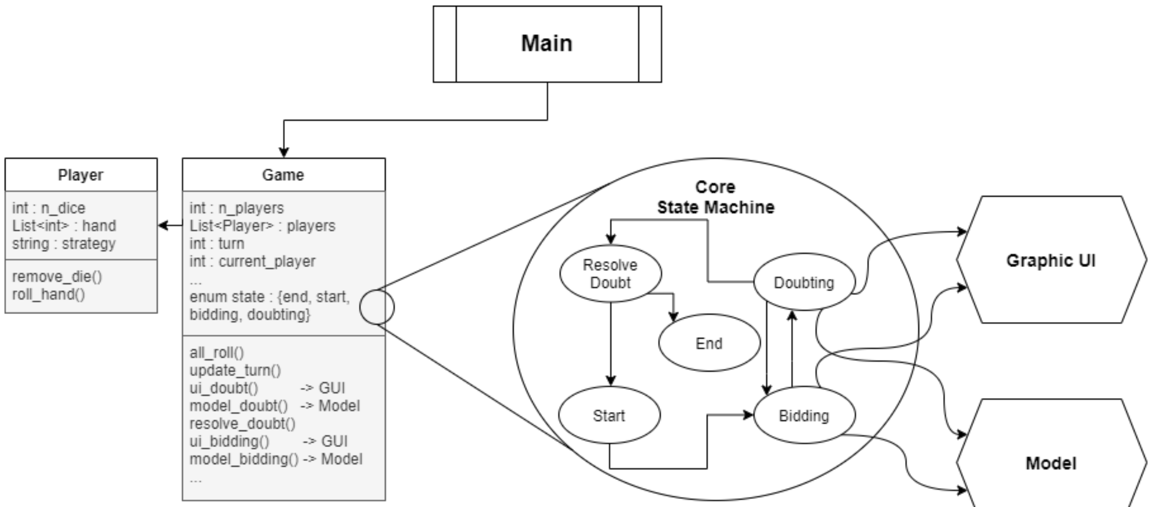


Figure 1: Program Structure with focus on the State Machine

3.2 User Interface

The user interface was developed using the `PySide 2` package, a Python binding of the cross-platform GUI toolkit Qt 5. As the game was initially developed for a command-line interface, while the UI was not yet functional, eventually several changes had to be made in order to have an interface-agnostic communication.

When the game starts, a new QApplication is started, and the start screen is initialized, offering the player the ability to choose the number of opponents and the difficulty (see Figure 2a). The top menu bar is also initialized, allowing the player to start a new game, exit, view the rules, the enemies' reasoning, or the credits.

When a new game is started, a new instance of the *MainWidget* class is initialized and added to the main window (see Figure 2b). This class implements the interface *CommunicationInterface*, which contains all the possible methods relating to information the game may want to display. However, since the UI thread needs to be permanently running, an instance of the Game object must be initialised and played in its own separate thread. This instance takes as an argument a *CommunicationInterface* and an *input queue*, which will serve as the methods of sending and receiving information respectively. Following is a list of detailed descriptions of each.

3.2.1 The Main Widget

The Main Widget contains all the controls associated with a game, and is reinitialized every time the player wants a new game. It contains an instance of the *Game* class, as well as an input queue. It displays all the output visuals and input controls required to play a game. It can also scale itself according to the number of players (see Figure 2c). The human player's side is displayed in the bottom half of the screen, separately from the other players' side. This is where the human player's cup, as well as the controls they can use, are displayed.

When one of the controls - that is, the bid selection group or the believe/call bluff group - is used, an input is added to the input queue. This can be two numbers (for a bid), or a 0/1 (for believing/calling a bluff).

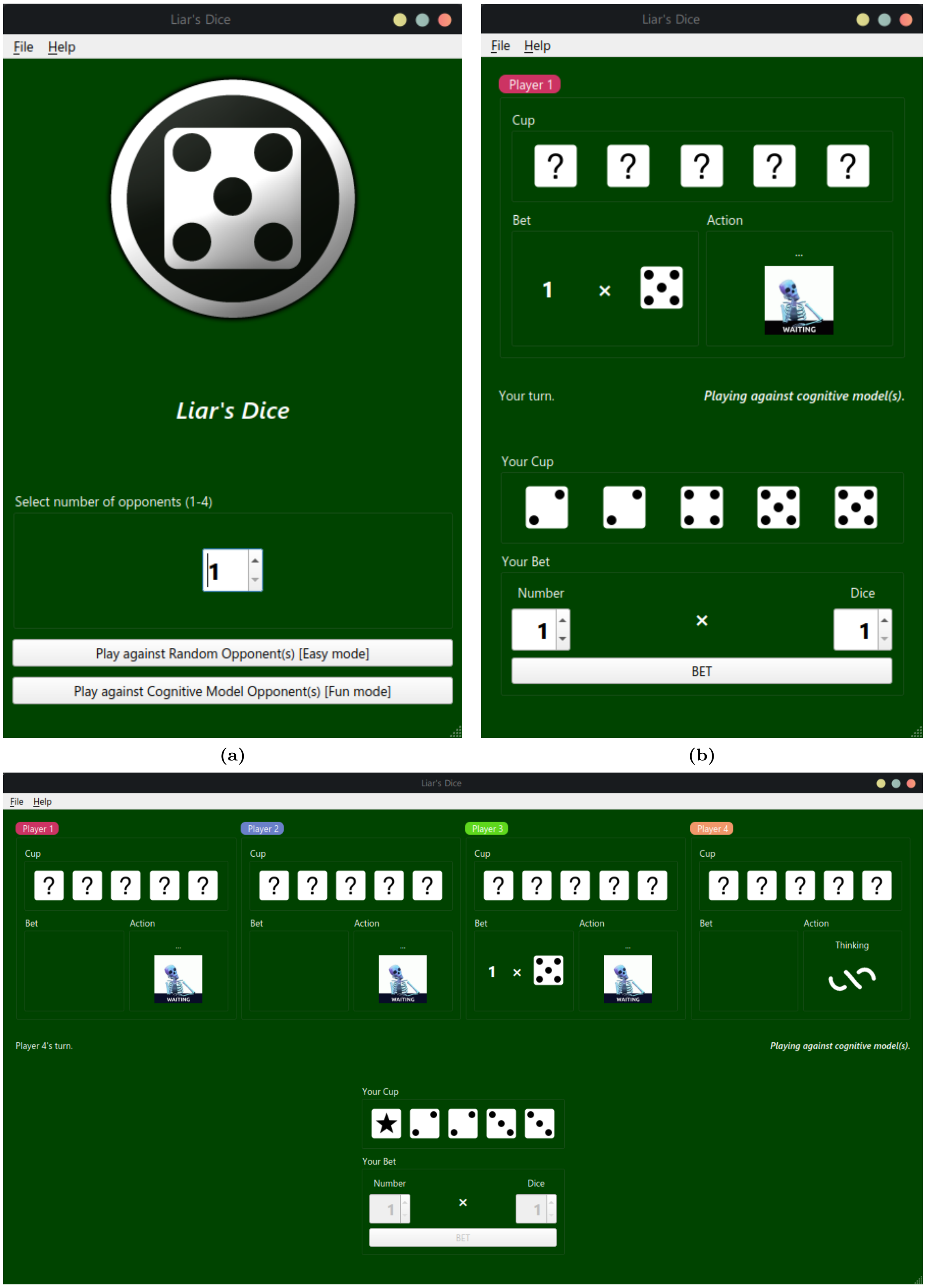
The other players' (i.e. enemies') side is displayed in the top half of the screen. For each enemy, their cup, bets, and actions are shown. When the player must not know the contents of the enemies' cups, the dice are shown as question marks.

When one of the exit controls (either the $\times$ button in the window border or the File $\rightarrow$ Exit option) is activated, a -1 is put into the running Main Widget's input queue (see Section 3.2.3 for an explanation).

3.2.2 Game to Main Widget: the *CommunicationInterface*

The *CommunicationInterface* is an interface that is agnostic with regards to the communication method. In our case, it was implemented by a graphical application widget. However, since it is completely agnostic and contains no functionality of its own, it could also be implemented entirely in the command line, or other types of communication.

It is not enough for the game thread to call a function in the *CommunicationInterface* (represented by the *MainWidget*) directly to send information to the main (UI) thread. In fact, it is not possible to call any function in another thread. This is where the *invoke_in_main_thread* function becomes useful: given a function and its arguments, it will *post* that function to the application: send it into a queue, to be handled by the UI when it can (Qt5, 2020).



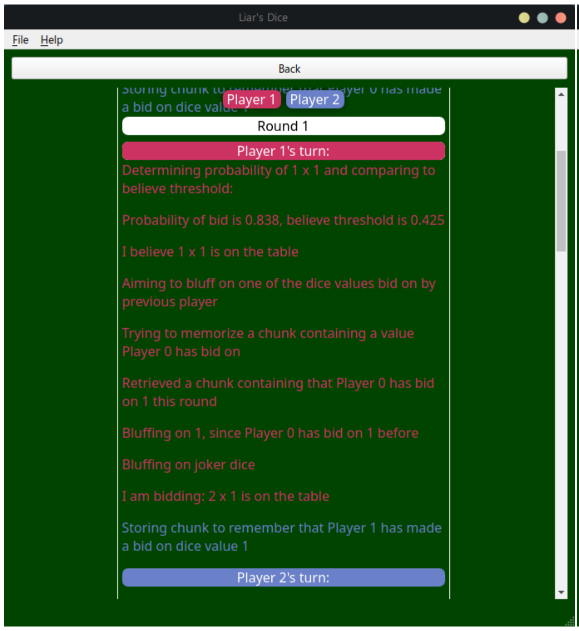
Your turn.

Playing against cognitive model(s).

Player 4's turn.

Playing against cognitive model(s).

Figure 2: This Figure shows screenshots of the Start screen (a), as well as the game being played against 1 (b) and 4 (c) opponents.



Storing chunk to remember that Player 0 has made a bid on dice value 1

Round 1

Player 1's turn:

Determining probability of 1 x 1 and comparing to believe threshold:

Probability of bid is 0.838, believe threshold is 0.425

I believe 1 x 1 is on the table

Aiming to bluff on one of the dice values bid on by previous player

Trying to memorize a chunk containing a value Player 0 has bid on

Retrieved a chunk containing that Player 0 has bid on 1 this round

Bluffing on 1, since Player 0 has bid on 1 before

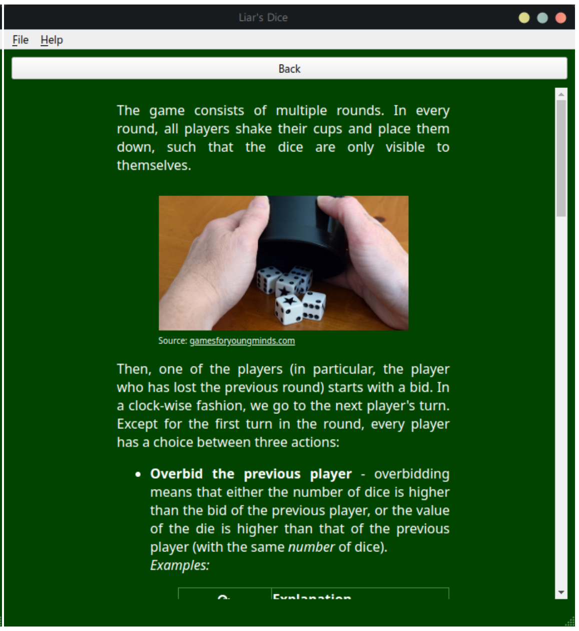
Bluffing on joker dice

I am bidding: 2 x 1 is on the table

Storing chunk to remember that Player 1 has made a bid on dice value 1

Player 2's turn:

(a)



The game consists of multiple rounds. In every round, all players shake their cups and place them down, such that the dice are only visible to themselves.

Then, one of the players (in particular, the player who has lost the previous round) starts with a bid. In a clock-wise fashion, we go to the next player's turn. Except for the first turn in the round, every player has a choice between three actions:

- **Overbid the previous player** - overbidding means that either the number of dice is higher than the bid of the previous player, or the value of the die is higher than that of the previous player (with the same *number* of dice). *Examples:*

(b)

Figure 3: Screenshots of auxiliary views. On the left, (a) shows the tab containing the reasoning of the cognitive model opponents. Each color represents the reasoning of a particular player. On the right, (b) shows the 'how to play' tab, containing detailed instructions for playing the game.

3.2.3 Main Widget to Game: the *input queue*

For the *Game*, receiving information, on the other hand, is much simpler. When an input is required, it can be obtained by popping from the *input queue* given as an argument to the *Game*. It is up to the UI thread to actually put objects into that queue. This required minimal modifications to the previous command-line interface, and brought minimal changes to program behavior, as waiting for a blocking queue is similar to waiting for standard input. One difference is that the GUI, as opposed to the CLI, allows the player to end the game at any time. Therefore, the game must be able to stop itself even when it is waiting to receive a gameplay-related input. The solution is simple: every time one such input is expected, it is checked, and if it is -1, the game will stop itself gracefully.

3.2.4 Game to Reasoning: the *reasoning file*

Another widget, separate from the *Main Widget*, is the *Reasoning* widget (shortcut F2). This can be used by the *Game* thread to display information about the players' internal reasoning (see Figure 3a). The way it works is by using a reasoning file, a file-like string buffer, which is written into by the *Game* thread at the different stages of the rounds. This file contains formatted HTML text, colored depending on the player it is referring to. Every time the player wishes to switch to this *Reasoning* widget, it will be populated with the most recent contents of the reasoning file. It also contains a "Back" button which allows the player to return to the Main Widget at any time.

3.2.5 Instructions: *How to play*

Last but not least, the ”How to play” widget (shortcut F1) will display information about the rules of the game and how to use the interface. It is a web engine view based on Chromium, as this format allows for simple and rapid design in HTML/CSS. A screenshot can be found in Figure 3b. Note that a later section in this report contains extensive instructions on how to use the app.

3.2.6 Dynamics

To graphically illustrate the dynamics of the UI-Game communication, a simplified overview can be seen in Figure 4. The Game is started in its own thread, it never communicates with the Main Widget in the main thread directly. Instead, the queues are used. The player models are queried for actions based on the current state of the game and their own current state. Once they decide on an action, it is sent back to the game, which will update itself, update the UI, and write the model's reasoning into a file.

At any moment during the game, and without losing the current state of gameplay, the human player can choose to switch to auxiliary tabs, such as the ”How to play” tab or the ”Reasoning” tab. These can prove useful when a player is stuck because they do not yet know the rules clearly, or when they wish to better understand how the cognitive models work.

For the sake of clarity, the Game thread sleeps when an agent is thinking, when dice are being counted at the end of a round, and when players are rolling dice. In reality, the outcomes of these events can be calculated almost instantly, but it would make the game difficult to understand for a human player, and prevent them from seeing possibly important information. The durations of the sleeps was calculated empirically. When a model thinks, this duration is based on the number of chunks it needs to retrieve from memory (see also Section 3.3.3). When the dice are counted at the end of a round, as well as when dice are rolled, the duration is based on the total number of dice visible on the screen - since it would take a human a longer time to count a larger amount of dice.

A large part of the application is animated. In particular, the different tabs of the application translate horizontally when switched. Dice also animate appropriately when they are thrown, counted or removed. There are also animations for describing enemy player's actions.

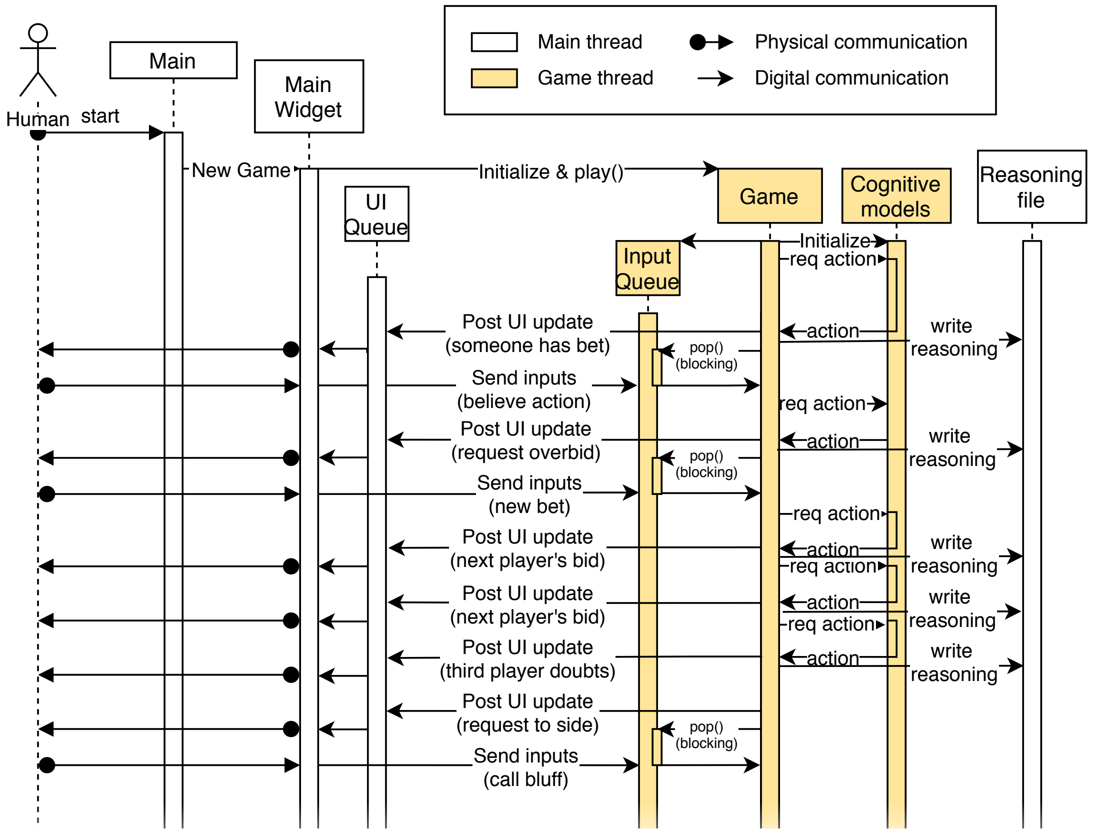


Figure 4: Sequence diagram showing a simplified view of the communication between various classes in the program (digital communication), as well as physical communication with the human user on the left side. In this example, a game with 3 cognitive model opponents is illustrated, in which the opponent right before the human is randomly selected to bid first. Note that this is a game that continues even after the sequence shown. `req` stands for "request". Simplifications include: objects are shown as belonging to threads, when in reality it is all of their methods that would more accurately belong to threads; the start screen widget is omitted, and instead the Main Widget is shown as being launched from the main process; the input queue, shown here as initialized in the Game, is actually initialized in the Main Widget and passed to the Game as a constructor argument; the Cognitive models, to the right of the Game, are condensed into one.

3.3 (Cognitive) Model strategies

In this section, we discuss the strategies of the opponents a human player will face in this game. There are two difficulties: one strategy works by means of random chances of believing bids and random bidding. Another works by means of a more sophisticated playstyle, which incorporates an ACT-R model.

3.3.1 Random Model

The random strategy executes the parts in its turn on the basis of random probabilities, making it the 'easy' opponent. The decision making is quite straightforward.

- Random strategy players will believe bids of players with an 80% chance, and will otherwise call them a liar.
- During the bidding phase, a probability of $\frac{1}{6}$ exists that the random strategy bids on the first possible number of ★ that overbids the previous bid. Otherwise, they will choose a random value (non-joker). If bidding on this value with the same number of dice as the previous bid is possible, they will do so. If this is not possible, they will increment the number of the bid by 1 and choose a new value randomly. In the case of overbidding a bid on joker dice, random strategy players will choose the first possible number of dice of the determined value.

3.3.2 ACT-R Cognitive Model

This section discusses the implementation of the cognitive model strategy.

For opponents playing according to the 'model' strategy, an ACT-R model is initialized at the beginning of a round. During any round, each model stores chunks with the bids of other players, whenever they make a bid. The chunks consist of the following two slots:

1. The player who made the bid
2. The value the player has bid on

An example would be that Player 2 has bid 4 × ⚄. The stored chunk would look as follows: Player 2 has bid on the value 5. The number of the bid (which is 4 in this example) is not stored, since no reasoning is done using the number. After a round is finished and a new round is started, models initialize a new ACT-R model, thereby no information is memorized between rounds.

In the following part, the strategy of the cognitive model in the different phases of a regular (non-first) turn will be discussed.

- **Doubting Phase:** The model has to assess the bid of the previous player, and determine whether to call the player a liar or believe the bid. The model firstly determines the probability of the bid, which is done by calculating the chance that at least the number of dice of the bid of a given value is available on the table. Formula (1) describes how this probability is determined:

$$P(x) = \sum_{k=i}^{N} \binom{N}{k} p^k (1-p)^{N-k}, \tag{1}$$

in which

- x: At least i times a dice value,
- i: The number of dice in the current bid minus the number of dice of that value the model has in its hand,
- N: The number of unknown dice $\rightarrow$ The total number of dice remaining in the game minus the number of dice in the model's cup,
- p: 1/6 for joker dice and 1/3 for non-joker dice (since joker dice also count toward the total of non-joker dice, doubling their probability).

Formula (1) is always applied to the unknown dice, since those are the only dice necessary to include in the calculation. For instance, when the bid is $10 \times$ ⚃, and the model has 5 dice, of which $3 \times$ ⚃, while 20 dice are remaining. The model calculates the probability of there being *at least* $7 \times$ ⚃ in remaining 15 dice.

This returns a probability which has to be weighed against a threshold, in order to determine whether the model believes the bid. A belief threshold is therefore drawn from a normal distribution with ($\mu = 1/4$ and $\sigma = 1/12$). Such a dynamic threshold seemed more suitable than a static threshold, as that would result in deterministic behavior. One the one hand, deterministic behavior is unrealistic, as in the game human players tend to both believe and not believe the same bid in different situations, given the same knowledge. In other words, the behavior human players show with regards to bidding does not seem to allow a static belief threshold. On the other hand, deterministic behavior seems exploitable, given that a human player can learn the bids that are handled in the deterministic way by the model. Following, if the probability of the bid is equal to or higher than the belief threshold, the model believes the bid and goes into the bidding phase. Otherwise, it will call the previous player a liar, and the penalty phase (in which the doubt is resolved) is started.

- **Bidding Phase:** In this phase, the model has to determine a bid, which the next player has to assess. Here, there is a possibility of 33% that the model will bluff. If so, the model will choose between the player before it and the player after it, and then try to remember one of the values this player has bid on. Remembering the value happens by retrieving the chunk with the highest activation, which can be a different chunk depending on the structure of the round. Most likely, the model retrieves the value last bid on by that player (especially early in a round). However, in longer rounds with many bids, a chunk different than the most recently stored chunk might have the highest activation. In that case, the model has a higher probability to remember the most frequently stored chunk.

  Furthermore, sometimes the model does not remember the value on which a player has bid. In games with a higher number of players, more bids are stored and the model's times are increased more often, causing lower chunk activation values. As a result, the models remember the bids less often than with fewer players. Therefore, for the models to have an acceptable memory rate (between 50% and 75%) in most of the games, they have a number of tries to remember a chunk, depending on the number of opponents. Then, if the model is able to remember a value, it will make a bid on the retrieved value. Otherwise, it will bluff on a randomly chosen value.

  When the model does not bluff, it will determine the most common values present in its hand and make a bid on one of these values (randomly chosen).

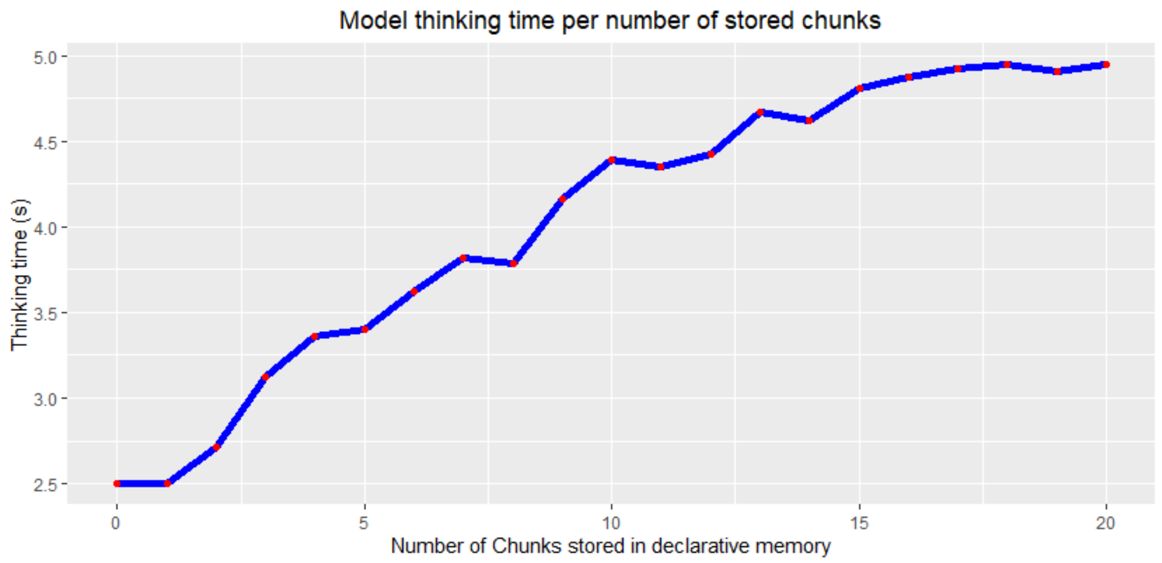


Figure 5: This Figure shows an example graph of Formula (2), depicting approximate thinking times of a model for different numbers of chunks stored in the declarative memory.

3.3.3 Simulating model thinking times

In order for game to be well-paced, the models most implement a form of thinking time. Without these thinking times, models can execute their turn in an instance, leaving no time for the human player to see and memorize bets of the models. Hence, in the UI models will take time to 'think' about their choices in the turn. In order to make the thinking time cognitively plausible, it was decided to determine the thinking time on the basis of the number of chunks in the declarative memory of the model. The thinking time is determined as by the following formula:

$$t(x) = max((log(2 * x) + b), 2.5), \tag{2}$$

in which x is the number of chunks stored in the memory of a model, and b is a random float between 1 and 1.5. The time is floored at a value of 2.5 seconds, since this value seemed like a reasonable minimal thinking time. Moreover, by the use of the log function, this formula does not exceed thinking times of 5-6 seconds, given the maximally possible number of chunks stored in a game, avoiding too long waiting times. As a result, the user will have reasonable thinking times for each phase in a round, allowing it to memorize bets accordingly.

Figure 5 shows approximate thinking times of a model, as determined by Formula (2). Thinking times in the plot are an approximation, since b in is a randomly determined float, hence thinking times per stored number of chunks fluctuate in a 0.5-second range.

Furthermore, since ACT-R models in Python require manual incrementing of their time, the determined thinking times are added to model (instead of our original idea, in which time was added as a randomly determined real number). As mentioned earlier, in order to have a reasonable recall rate of chunks (which highly depends on the model time), models have a number of tries to remember a chunk.

4 Evaluation of the game

The game was tested by a number of participants. One participant liked the game, but had only played the game a few times before and added that some of the model turns were a bit too quick. Partly because of this feedback, we decided to increment the model thinking time with the number of chunks (on a logarithmic scale). As a result, early (less interesting) turns were executed in approximately the same time as before, while the final turns (in which the bids get closer to the actual values) allow for more thinking time. Not only is this nice for less experienced players, but this is also more cognitively plausible (since comparing your possible bids to more memory chunks requires more thinking time).

A second participant had played the game many times before, and played the game in a more finished state. He first played against the random models and easily beat them. However, he had more trouble beating the cognitive model opponents, but argued that this was a nice challenge for more experienced players. In the end, he said the game looked good and was nice to play, and maybe it would be nice to make it a bit more colorful. After this we tried to implement some more colors in the game.

A third participant had a bit of trouble seeing some of the information on the screen, since he had bad sight. After this we decided to increase the font size of some of the important text labels in the game.

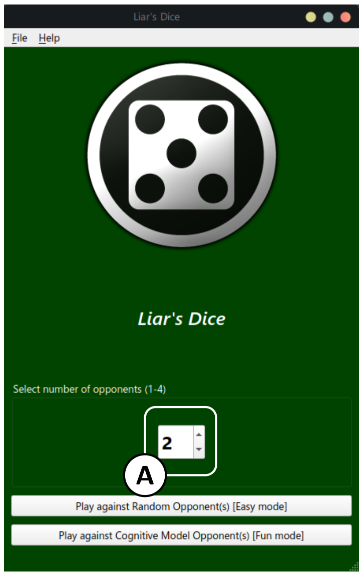
4.1 Possible improvements

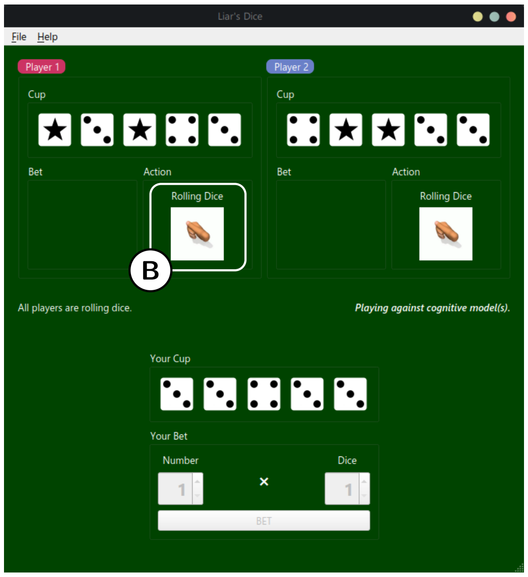
- *Improve model reasoning from memory*: Currently, the reasoning from memory of the models happens mostly in the bidding part. Also, the models now determine whether they believe a bid on the basis of probability calculations, which in principle is what experienced human players do. However, the determined probability is now weighed against a threshold drawn from a normal distribution. An improvement would be for the models to store chunks about bluffing behavior of other players, which could influence whether the models believe those players' bids or not. On top of that, the values bid on by players in a round could also be used as an effect on the threshold. In a round where a model stores many chunks containing the bid value '5', it could be such that the model more easily believes a bid made on ⚄, in contrast to solely basing this on probability calculations.

- *Real multiplayer*: The game could be rewritten into a client-server architecture, to allow users to host servers locally and play with others in the LAN. With multiple human opponents in a game, along with our AI opponent models, gameplay could become much more exciting.

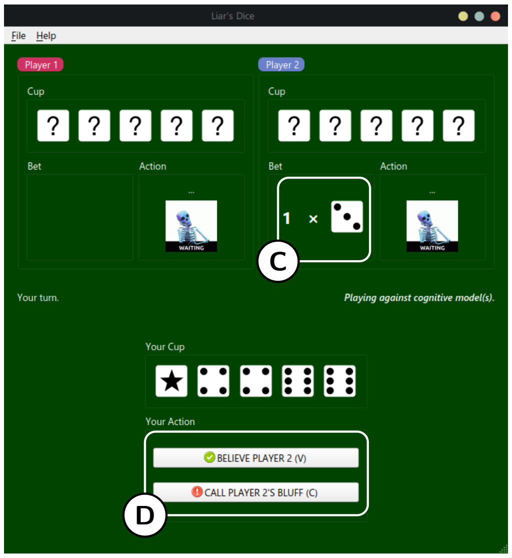
- *Allow other versions of the game to be played*: For example, an alternative version in which you keep your die when you are correct, and try to be the last player with dice left.

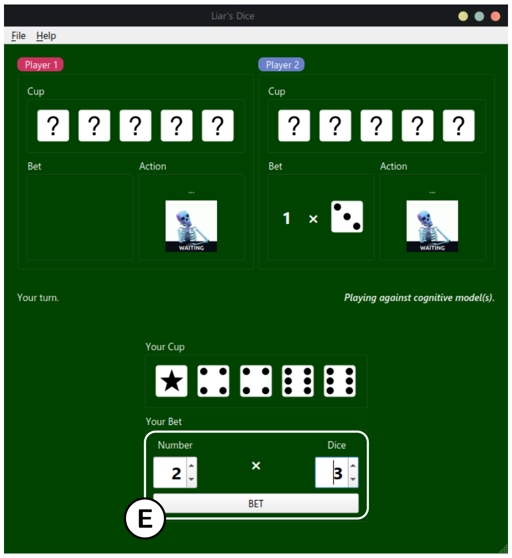
5 Instructions on how to use the App


Liar's Dice
File Help
Liar's Dice
Select number of opponents (1-4)
2
A
Play against Random Opponent(s) [Easy mode]
Play against Cognitive Model Opponent(s) [Fun mode]

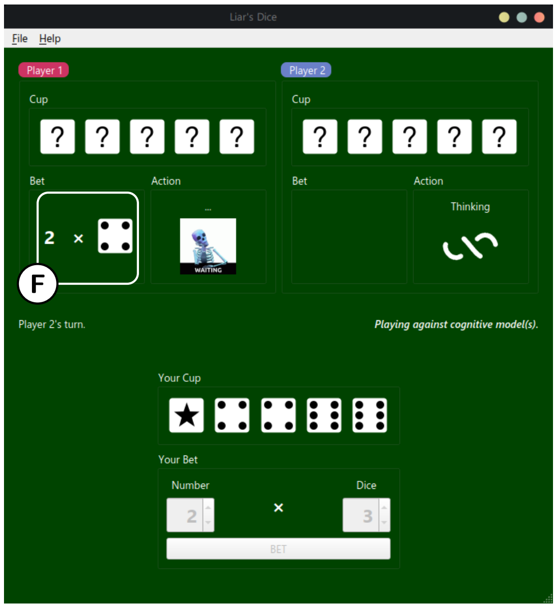


First, select the number of opponents (A) and the preferred difficulty. Then wait for all the players to roll dice (B).



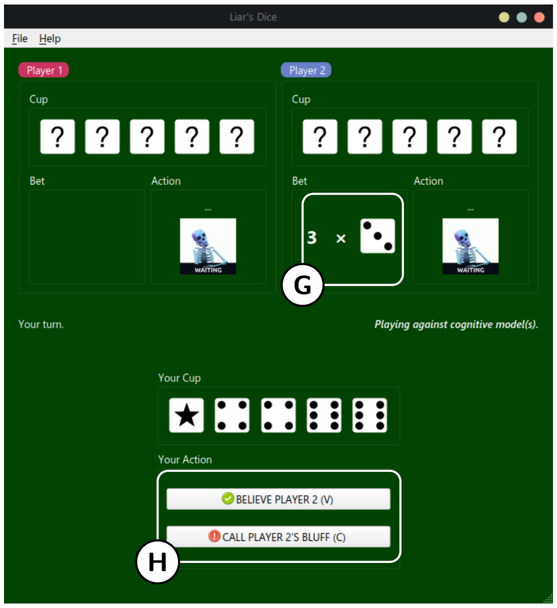


Supposing another opponent is selected to bid first (C), you will be asked whether you believe the bid (D). Suppose you believe (press the first button). Then, you will be asked to overbid, which you can do through the interface (E).



Player 2's turn.

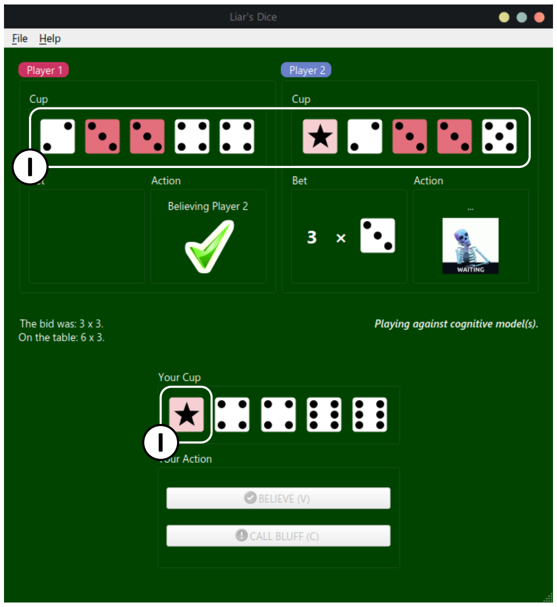
Playing against cognitive model(s).



Your turn.

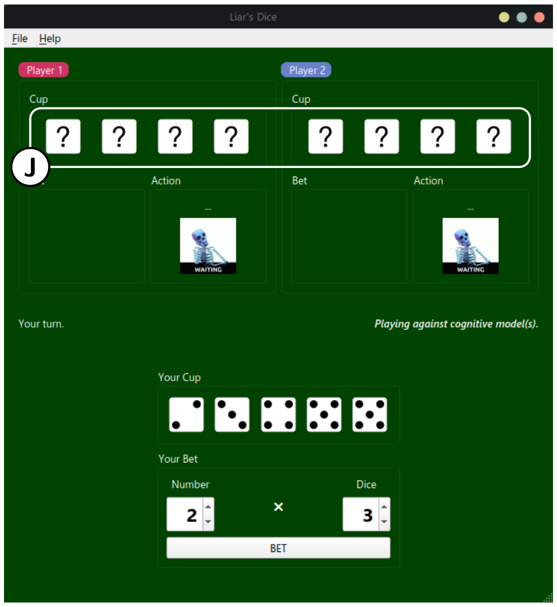
Playing against cognitive model(s).

The other players will bid, in a clockwise manner (F,G). Suppose this round, you choose to doubt Player 2's bid - press the second button (H).



The bid was: 3 x 3.
On the table: 6 x 3.

Playing against cognitive model(s).



Your turn.

Playing against cognitive model(s).

The cups are lifted, and the dice matching the bid are flashing red (I). In our case, the 3's and the Joker dice match the bid. As the animation cannot be displayed in this report, some of the highlights are not very visible. A new round starts, and the correct players have lost a die each (J).

References

Anderson, J. R., Matessa, M., & Lebiere, C. (1997, December). ACT-R: A Theory of Higher Level Cognition and Its Relation to Visual Attention. *Hum.-Comput. Interact.*, *12*(4), 439–462. Retrieved from `https://doi.org/10.1207/s15327051hci1204_5` doi: 10.1207/s15327051hci1204_5

Qt5. (2020). Documentation of QCoreApplication::postEvent. Retrieved from `https://doc.qt.io/qt-5/qcoreapplication.html#postEvent`

Taatgen, N., & Weerd, H. D. (2016). Using a Cognitive Architecture in Educational and Recreational Games: How to Incorporate a Model in your App..