# TDLOG
# Project report

Adrien CANCES, Florentin POUCIN

Monday, February 20, 2020

## 1 Introduction

The goal of our project was to make a decision-support system for Texas hold'em Poker. For the sake of simplicity, we limited our project to two-player games. Knowing some of the cards on the table, what is the probability of winning the game? The system we developed is composed of a calculation program and of a graphical interface program allowing the user to enter the data of the game.

## 2 Encoding the data

One of the biggest challenge in this project was to make the computations fast enough. Indeed, a nice and user-friendly interface is useless if the user has to wait a whole minute to get the probability he asks for. Thus, the encoding of the data had to be carefully chosen. We first opted for a string encoding, in which the hand |A♠|10♥|7♦|7♣|Q♦| was for instance encoded by the string `"14s 10h 7d 7c 12d"`. This encoding was very handy because it is rather transparent: one can immediately "translate" the encoded hand to the real hand by head. But the computations were too long because of several heavy operations such as string concatenations or multiple conversions from string to integers during the computations.

We therefore chose a rawer encoding, which is not very visual but is perfectly adapted to the computations due to its interesting properties, and which we describe in the next paragraphs.

### 2.1 Rank and suit

There are exactly 52 different cards in Texas Holdem. Thus, once we choose an arbitrary numbering of the cards, we can identify them to the set $[\![0, 51]\!]$.
Accordingly, a hand (i.e. a set of five distinct cards) will be identified to a 5-subset of $[\![0, 51]\!]$. We denote $\mathcal{H}$ the sets of 5-subsets of $[\![0, 51]\!]$.

We respectively denote Spade, Heart, Club and Diamond by $s$, $h$, $c$ and $d$, and identify the four suits to 0, 1, 2 and 3.
Finally, we identify the ranks 2, 3, 4, ... , Jack, Queen, King, Ace to $0, 1, 2, ..., 9, 10, 11, 12$.
The following tables sum up these identifications.

| Two | Three | Tour | Five | Six | Seven | Eight | Nine | Ten | Jack | Queen | King | Ace |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

| Spade | Heart | Club | Diamond |
|---|---|---|---|
| 0 | 1 | 2 | 3 |

We identify the card of rank $r \in [\![0, 12]\!]$ and of suit $s \in [\![0, 3]\!]$ to the integer $13s + r$ so that the rank and the suit of a card numbered $c \in [\![0, 51]\!]$ are respectively $r = (c \,\%\, 13)$ and $s = (c \,//\, 13)$, where $a \,\%\, b$ and $a \,//\, b$ respectively denote the rest and the quotient of the euclidian division of $a$ by $b$.
The rank and suit of a card $c$ will be denoted $\mathrm{rk}(c)$ and $\mathrm{st}(c)$.

In the whole computation program, a card is identified to the corresponding integer in $[\![0, 51]\!]$, so the functions only manipulate arrays of integers, which is consequently easier and faster than manipulating strings.

## 2.2 Hand categories

The set of possible hands can be partitioned into exactly nine categories, which are listed in the following table by increasing strength, along with the integers we respectively identify them with. To compute the category of a hand, we use two histograms : the histograms of ranks and the histograms of suits. By analysing these two histograms, it is rather easy to get the category of a hand by a series of boolean tests.

| | |
|---|---|
| high card | 0 |
| pair | 1 |
| double pair | 2 |
| three of a kind | 3 |
| straight | 4 |
| flush | 5 |
| full house | 6 |
| four of a kind | 7 |
| straight flush | 8 |

We denote by $\mathrm{cat}(h) \in [\![0, 8]\!]$ the category of hand $h$.
Let $h_1, h_2$ be two hands. If they belong to different categories, then the hand with the highest category beats the other. If they belong to the the same category, then we need to look further into the cards to rank them. We call features of a hand $h$ and denote $\mathrm{feat}(h)$ the information we use to rank hands that fall into the same category. $\mathrm{feat}(h)$ is tuple of length between one and five (depending on the category of $h$) and with its elements in $[\![0, 12]\!]$. Broadly speaking, it is composed of the ranks of some of the cards in $h$, ordered by decreasing order of importance. For instance, the hand |A♠ |10♥ |7♦ |7♣ |Q♦ | has features $(5, 12, 10, 8)$ because it is a pair of sevens (5), with single cards Ace (12), Queen (10) and ten (8).

Let us denote $h_1 \prec h_2$ the relation "$h_1$ loses against $h_2$", or equivalently "$h_2$ wins against $h_1$". Using lexicographical order, which we will also denote $\prec$, we have the following property :

$$h_1 \prec h_2 \quad \Longleftrightarrow \quad \text{cat}(h_1) \oplus \text{feat}(h_1) \prec \text{cat}(h_2) \oplus \text{feat}(h_2) \tag{1}$$

It should be noted that some hands, despite being disjoint, have exactly the same value, in which case the game is a draw.

### 2.3 Computing the value of the best hand among seven cards

At the end of the game, the score of a player is the value of the best hand that can be extracted from the five cards on the table and the two cards he has. To be precise, the best hand is not necessarily unique but the value is.

To compute the value of the best hand among seven cards, we run through all the possible hands that are contained in the seven cards. There are exactly 21 hands so the computation is rather quick.

Instead of a for loop, we use a list comprehension to get the list of respective concatenated category and features of the 21 hands. We then use the built-in maximum function to extract the best one. Indeed, since Python uses lexicographical order, property (1) ensures that the maximum value in the list corresponds to the best hand.

```
cat_feats = [category_and_features(hand) for hand
             in list(it.combinations(seven_cards, 5))]
return max(cat_feats)
```

Figure 1: Segment of code determining the value of the best hand with a list comprehension

The `itertools` module was very convenient to code this function, as it allows to get the list of k-combination of a set with the function `combination`.

## 3 Probability calculations

During two-player Texas Holdem games, exactly nine cards intervene: the two cards of of the first player $J_1$, the two cards of the second player $J_2$, and the five cards successively displayed on the table.

We will not look at the mathematics into details because it is not the goal of the project, but we need to define what exactly we are computing. Throughout the project, we consider discrete uniform probability, which means that all final situations have an equal chance of happening.

Suppose we know some of the cards on the table (but not necessarily all of them). Let $I$ be the information we know. There are a finite number of possibilities for what the unknown cards of the table are. Let $S_I$ be the set of these possible situations. We call "probability that player $J_k$

wins knowing information $I$" the quantity $P_k(I) = \frac{1}{|S_I|} \sum_{s \in S_I} g_s$, where $g_s$ is equal to 1 if $J_k$ wins in situation $s$, 0 if $J_k$ loses, and 0.5 if the game is a draw.

Note that given this definition, we obviously have $P_1(I) + P_2(I) = 1$.

An easy way to compute $P_1(I)$ is to set a variable v to 0, explore all possible situations $s \in S_I$ and for each of them, compute $g_s$ (that is, determine who wins) and add it to v. It suffices to divide v by the number of situations explored to get the wanted value.

### 3.1 Exact calculations

The algorithm described above gives the exact value of $P_1(I)$, but exploring all possible cases can be extremely long depending on the number of cards we know. The following table shows the number of possible situations that have to be explored for different information $I$.

| $I$ : cards unknown | $\|S_I\|$ : number of possible situations |
|---|---|
| none | 1 |
| river | 44 |
| turn and river | 990 |
| opponent cards | 990 |
| opponent cards and river | $45540 \approx 5.10^4$ |
| opponent cards, turn and river | $1070190 \approx 10^6$ |
| flop, turn and river | $1712304 \approx 2.10^6$ |
| opponent cards, flop, turn and river | $20975724000 \approx 2.10^{10}$ |

For $S_I < 1000$, we can easily do the computations in a couple of seconds. For $S_I = 45540$, we can do the computations in a bit more than twenty seconds, which is alright but not fast enough for our goal. For $S_I \approx 10^6$, the calculations take about eight minutes, which is way too long. Finally, for $S_I \approx 2.10^{10}$, the calculations would take about 106 days, way over the limit!
This is why we restricted ourselves to the first seven cases of the table.

### 3.2 Approximate calculations

The idea for approximate calculations is to only explore a fraction of the possible situations. Say we explore the set $T_I \subseteq S_I$, then the estimated value will be $\frac{1}{|T_I|} \sum_{s \in T_I} g_s$.
But choosing a good set $T_I$ is extremely difficult, because it has to be representative of all the situations in $S_I$. This is why we chose to do the following instead:

This algorithm uses Monte Carlo technique. The idea is to set a proportion $p$ of situations to explore, compute the set $S_I$, and successively draw $\lfloor p \, \mathrm{Card}(S_I) \rfloor$ times a random situation $s$ in $S_I$ with replacement.

**Input:** I, p
**Output:** R
compute $S_I$;
$N_I \longleftarrow |S_I|$;
$N \longleftarrow \lfloor pN_I \rfloor$;
**for** $i = 1, \ldots, N$ **do**
    pick $s \in S_I$ uniformly;
    compute $g_s$;
    $v \longleftarrow v + g_s$;
$R \longleftarrow v/N$;

The module `random` and `itertools` were extremely useful to implement this code.
$S_I$ is easily computed with a list comprehension, which is faster than a for loop and lightens the code. In the next figure is an example of one of the list comprehensions we used.

```
possible_rivers_hole_cards = [(r, list(hc)) for r in CARDS for hc in CARD_PAIRS
                              if not r in hc and not hc[0] in known_cards
                              and not hc[1] in known_cards]
```

Figure 2: List comprehension for computing the set of possible situations

Drawing the situations successively with replacements can be done in a single line with the function `rd.choices(population, k = N)` from the Python `random` module, which returns a $k$ sized list of elements from the list `population` with replacement. It suffices to take the list of indices $[\![0, N_I - 1]\!]$ of $S_I$ for `population`, which means we draw the indices of the situations rather than the situations.

### 3.3 General computation function

Of course, the way we compute the exact or estimated probability $P_1(I)$ depends on which cards of the table we know. We therefore have seven different computation function, which correspond to the seven first lines of the table of the $(I, |S_I|)$ couples.

To conveniently adapt the computation program to the interface program, we wrote a general function which takes in argument an array of nine integers corresponding to the nine cards of table in a given order, and computes the wanted probability. The array passed in argument contains 52 or 53 for the cards that are not known, so the function knows which of the nines values correspond to known cards and which do not. The function can therefore know which of the seven specific function it has to use for the calculations.

# 4 Graphical interface

The graphical interface has to enable the user to chose the cards he knows and to visualise the game situation. If the conditions of computation are satisfied, he has to be able to launch the program and get his winning probabilities depending on the situation described. We wanted to make a visual interface very simple to use, in which the user can select cards by clicking. In order to do that, we used `tkinter` package on python.

## 4.1 Main interface

The main interface represents the game situation described by the user as shown in the Figure 1 below. It is composed of 9 cards : the 2 player cards, the 2 opponent cards, the 3 flop cards, the turn card and the river card. For the implementation of all graphical interface, we have created two classes : `Card` and `Interface`.

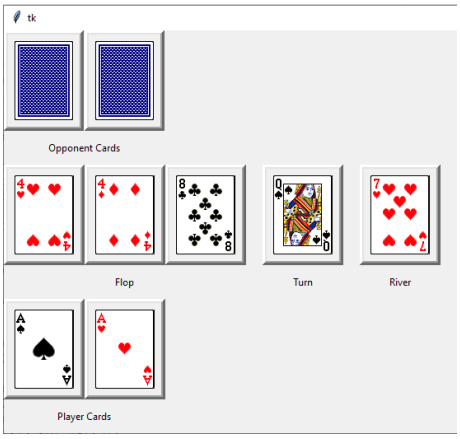


Figure 3: Situation in which the user does not know his opponent cards

**`Card class :`** contains the value of the card, the interface to which it belongs and its position in this interface (row and column). A function display seeks the picture path of the card depending on its value and displays it on a button. There are two different buttons whether the card belongs to the main interface or to the card selector (subsection below).

**`Interface class :`** contains the nine cards describing the game situation. These cards can be face-down, if the user doesn't know the value of the card, or face-up if this card is known. Each card is represented as a button with its own command : clicking on this button closes the interface window, opens the card selector, and, after the user has chosen a card, opens a new interface with the new value of the card corresponding to the initial button.

The problem of button command is one difficulty that we have encountered. In fact, in the main interface each of the 9 buttons has his own command, changing only the value of his own card. In addition, the module `Button` of `tkinter` enables to link a function to a button only if this function do not take any argument (impossible to take a "type of card" argument). The problem was we did not want to create nine different functions doing almost the same and to respond to this, we used **`lambda`** functions and class attributes as illustrated in the last lines of the following code.

```
    def button_card (self, id_button):
        self.im = PhotoImage(master=self.window, file=self.path)
        if (self.val==53):
            self.bout = Button(self.window, image=self.im, bd='5')
        else:
            self.bout = Button(self.window, image=self.im, command =
lambda : self.interface.set_card(id_button), bd='5')
```

Figure 4: Main interface `button_card` function

## 4.2 Card Selector

The card selector is called by every button from the main interface. It enables the user to pick a card among the 52 playing cards of the pack. The cards already present in the main interface cannot be picked and appear face-down on the card selector, however the user can select these face-down card to cancel his previous selection.

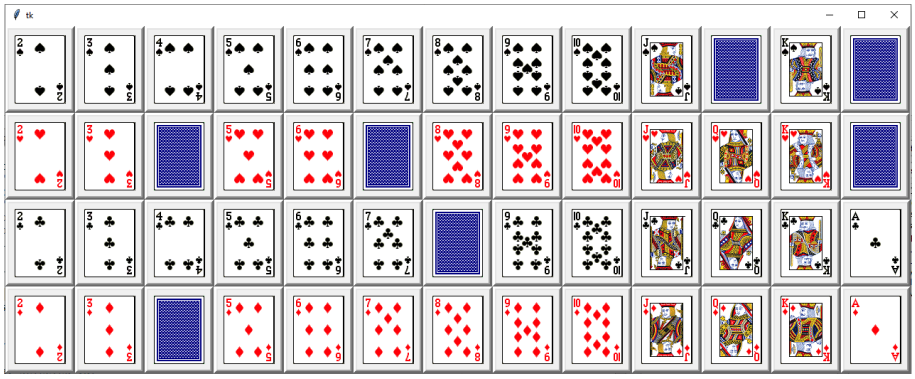

Figure 5: Card Selector in which the cards of Figure 1 are impossible to select

The problem of button command is also a difficulty that we have encountered for the card selector, but this time there were 52 different cards. Then, we used card attribute `self.value` and list by

comprehension to reduce the number of functions and to accelerate the program. In the following code we can see this list by comprehension composed of 52 cards and the `for` loop displaying them, face-up or face-down, with the function `card.button_val`.

```
card_list = [Card(i*13+j, window, interface, i, j) for i in range(4) for j in range(13)]
for i in range(52):
    if(i in selected_card):
        card_list[i].set_val(52)
    card_list[i].button_val()
```

Figure 6: Part of code of the Card Selector display

## 4.3 Connection with the computation

To enable the user to compute his winning probabilities, there is a computation button at the right of the nine cards. The command is available only if the user has selected the two player cards and at least the three flop cards or the two opponent cards. In addition, there are some cards that the user cannot select depending on the situation. For example, the user cannot select the river card or the turn card if he did not select the three flop cards before. In this case, these cards appear in grey on the interface and there is no more command linked to their button (`if(self.val==53)` condition in the Figure 4). This is illustrates by the figure just below.

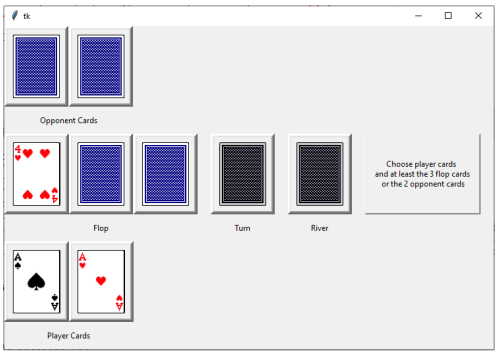


Figure 7: The user cannot launch computation because cards are missing

If the user has selected enough cards, he can launch computation by clicking on the button at the right. This button is linked with the computation probabilities functions described in the previous part and displays the winning probabilities of the two players.

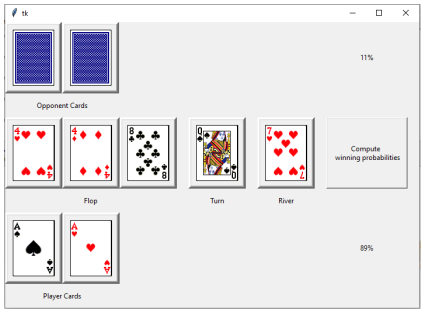


Figure 8: Computation when the user does not know the opponent cards

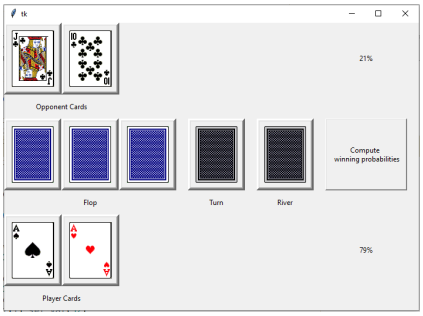


Figure 9: Computation when the user only knows player and opponent cards

## 5 Conclusion

In the end, we have made a Python project completing most of our objectives. The algorithms concerning the encoding of cards and poker rules are totally functional. The computation of probabilities could be faster with an optimization of the code or by replacing some part of the Python code with C language. However, the calculation is honest and precise enough for its use. The interface is quiet simple but it is efficient, simple to use and visual. It could be interesting to beautify the display and maybe create an application with Java but this would be another project.