Group 4: Briscola

Ananya Kollipara (22kc34)

Arlen Smith (22htl2)

Christian Fiorino (22bqs2)

Jinpeng Deng (22ss117)

Course Modelling Project

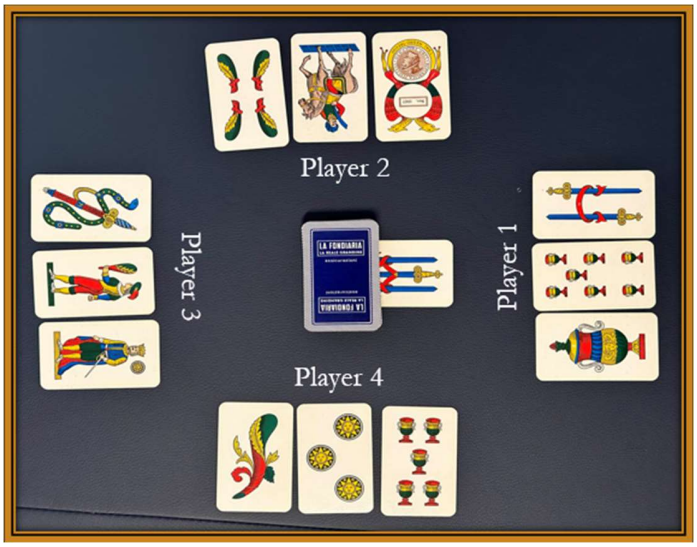
CISC/CMPE 204 – Logic for Computing Science

December 5th, 2024

Project Summary

The game **Briscola** has players play a card from their hand each round. Based on the suit of the card and the value of each card (Ace, King, etc.), a player wins a round if the card they played had the highest value and was the same suit as the first card played. In addition, at the start of the game a suit is chosen to be the "Briscola", which ignores the requirement of the highest card needing to be the same suit as the first card played in order to win the round, but still follows the rules regarding value. The game continues until all cards have been played, with each player drawing one card at the end of each round. In a regular game of Briscola, the player with the most points at the end of all the rounds wins – based on the card values. For this project, the player who wins the most rounds wins the game instead.

We will model if it is possible for Player-1 to win 4 of the 10 possible rounds in a game of four players. The model will be using the game configuration seen below, which shows the players' starting hands and the Briscola suit (swords). The rest of the deck is not predetermined.



Propositions

The propositions will make use of the following variables:

Variable	Description
(p)	A specified player. Can have a value of (1-4) to represent Player-1, Player-2, and so on. Also acts as a dictionary to count the player wins for each player.
(t)	The current trick/round of the game. Can have a value of (1-10) to represent Trick-1, Trick-2, and so on.
(c)	The specified card out of the available 40 cards. Can have a value of (1-40) to represent Card-1, Card-2, and so on. Each card has its own Suit (ranging from "Swords", "Coins", "Cups", and "Clubs") and Value (ranging from 1-7, then J, H, and K) and is stored in a dictionary.
(b)	The Briscola suit of the game. Can have a string value of ("Swords", "Coins", "Cups", or "Clubs")

Proposition	Description
starting_player(p)	Returns True if player (p) is starting the trick.
val_is_greater(c_1, c_2)	Returns True if (c_1)'s value is greater than (c_2)'s value.
card_is_bris(c, b)	Returns True if (c) is the same suit as (b).
card_is_same_suit(c_1, c_2)	Returns True if (c_1) is the same suit as (c_2).
card_beats_card(c_1, c_2)	Returns True if (c_1) beats (c_2) in terms of value and suit. Uses the above propositions to check this.
player_win_trick(p, t, [c_1, c_2, c_3, c_4])	Returns True if the player (p) wins the trick (t) based on the cards [c_1, c_2, c_3, c_4] that were played in the trick.
can_draw_card(p)	Returns True if player (p) can draw one card.
card_in_trick(c)	Returns True if card (c) exists in the trick.

Constraints

Constraint	Description
player_win_trick(p_1, t, [c_1, c_2, c_3, c_4]) ∧ starting_player(p_2)	When checking if player (p_1) won the trick (t), we also check if (p_2) is the starting player (where it is possible for $p_1 = p_2$) to determine the order the cards were played.
¬ ($c_1 \Leftrightarrow c_2$)	Two cards (c_1) and (c_2) cannot be the same (they must be unique). (And by proxy, they can't have the same suit ***and*** the same value).
(val_is_greater(c_1, c_2) ∧ val_is_greater(c_2, c_3)) → val_is_greater(c_1, c_3)	If a card's value (c_1) is greater than another card's value (c_2). Then, if (c_2)'s value is greater than another card's value (c_3), then (c_1)'s value is greater than (c_3)'s value.
(player_win_trick(p_1, t, [c_1, c_2, c_3, c_4]) ∨ player_win_trick(p_2, t, [c_1, c_2, c_3, c_4]) ∨ player_win_trick(p_3, t, [c_1, c_2, c_3, c_4]) ∨ player_win_trick(p_4, t, [c_1, c_2, c_3, c_4])) → (can_draw_card(p_1) ∧ can_draw_card(p_2) ∧ can_draw_card(p_3) ∧ can_draw_card(p_4))	If any of the four players have won a trick, this means a trick has concluded, and thus all for players draw one card.

Model Exploration

Ensuring there are no Non-Unique Suit Comparisons:

After trying to make the code that is used for getting each card to beat each card working as intended, I ran into a small problem. Despite cards needing to be unique, the solver would compare the suit of (c_1) to the suit of (c_1), essentially comparing a single card (c_1) to itself.

Although this would be technically true, this wouldn't make sense in logic, since all cards are unique. The code that was causing this issue is seen below:

```
# Initalize all cards with the same suit (this method will automatically make cards suits symmetric as well)
same_suit_propositions=[]
for card1 in CARDS:
    for card2 in CARDS:
        if card1 == card2:
            continue
        if CARDS[card1]["suit"] == CARDS[card2]["suit"]:
            # If the suits of the cards match, they are the same suit.
            E.add_constraint(card_is_same_suit(card1, card2))
        else:
            # Otherwise, they are not the same suit.
            E.add_constraint(~card_is_same_suit(card1, card2))

#TODO: Cards are still making themselves the same suit as themselves, this shouldnt happen.
```

In order to fix this issue, I added a new (`card_is_same_suit(c₁, c₂)`) constraint when initializing the suits to tell the model that a card cannot be the same suit as itself. This update can be seen in the below image of the code:

```
# Initalize all cards with the same suit (this method will automatically make cards suits symmetric as well)
for card1 in CARDS:
    # Constraint: Cards cannot be the same suit as themselves, as that wouldn't make sense
    E.add_constraint(~card_is_same_suit(card1, card1))
    for card2 in CARDS:
        if card1 == card2:
            continue
        if CARDS[card1]["suit"] == CARDS[card2]["suit"]:
            # If the suits of the cards match, they are the same suit.
            E.add_constraint(card_is_same_suit(card1, card2))
        else:
            # Otherwise, they are not the same suit.
            E.add_constraint(~card_is_same_suit(card1, card2))
```

Jape Proof Ideas

We haven't started on fully implementing the JAPE proofs yet but below are some potential ideas we can look into solving, with the possible premises of the JAPE proofs being in curly brackets ({}), and the possible conclusions of the JAPE proofs being in square brackets ([]):

1. This proof compares the winner and losers of a single trick: {if player-1 wins the trick}, [then player-2, player-3, and player-4 could not have won].
2. This proof checks for if the game is over: {((player-1 has won 4 tricks) or (round 10 has ended)) then stop the game}, [stop the game].
3. This proof checks to make sure that all players don't have more than 3 cards: {(player-1 and player-2 and player-3, and player-4) have 3 or 2 or 1 card(s) in their hand}, [then none of the players have 4 cards in their hand].

Requested Feedback

1. Do you have any advice on how we could have a proposition that checks if the trick has ended? Are we allowed to use other propositions as arguments for this proposition?
2. Do you have any advice for coming up with additional ideas for the JAPE proofs?

3. Do you have any advice on how we set what cards the player's have? Should it be as a proposition, or as a variable?
4. Are we allowed to use True or False *variables* instead of propositions, or do we have to use propositions to represent True or False values?

First-Order Extension

Describe how you might extend your model to a predicate logic setting, including how both the propositions and constraints would be updated. ***There is no need to implement this extension!***

[Haven't started on Predicate Logic yet].

Useful Notation

∧ ∨ ¬ → ∀ ∃