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, and players will draw right after they play their card instead of at the end of the round.

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 2-7, then A, J, H, and K) and is stored in a dictionary.
(s)	One of the four possible suits of the game. Can have a string value of ("Swords", "Coins", "Cups", or "Clubs")
(v)	One of the possible values that a card can be assigned. Can have a value of (2-7) or (A, J, H, K), where (J, H, K) represents the face cards and (A) represents the ace.

Proposition	Description
starting_player(p, t)	Returns True if player (p) is starting the trick (t).
val_is_greater(v_1, v_2)	Returns True if (v_1) is greater than (v_2).
card_is_brisc(c, s)	Returns True if (c) is the same suit as (s), where (s) is the Briscola suit.
card_is_suit_of_round(c, s, t)	Returns True if (c) is the same suit as (s), where (s) is the suit currently leading the round (referred to as the "round suit"), on trick (t).
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, t)	Returns True if (c_1) beats (c_2) in terms of value and suit on trick (t).
player_has_card(c, p, t)	Returns True if player (p) has the card (c) during the trick (t).
player_wins_trick(c, p, t)	Returns True if the player (p) wins the trick (t) with a card (c).
suit_of_round(s, t)	Returns True if a suit (s) is the round suit of the trick (t).
player_plays_card(c, p, t)	Returns True if a player (p) played a card (c) on the trick (t)
starting_player_card(p, c, t)	Returns True if player (p) is the starting player and plays the card (c) on the trick (t).
player_draws_card(c, p, t)	Returns True if player (p) draws a new card (c) on a trick (t).

Constraints

There are three main types of constraints used for this model. They are constraints for **card comparisons**, constraints for **round flow**, and constraints that **limit** the number of times an action can occur.

Card Comparisons:

- For every unique pair of two cards (c_1) and (c_2), both cards can't have the same suit ***and*** the same value (they must be unique):

$$\neg (c_1 \leftrightarrow c_2)$$

- If a card's value (v_1) is greater than another card's value (v_2), and if (v_2) is greater than another card's value (v_3), then (v_1) is also greater than (v_3):

$$(\text{val\_is\_greater}(v_1, v_2) \land \text{val\_is\_greater}(v_2, v_3)) \rightarrow \text{val\_is\_greater}(v_1, v_3)$$

- A card (c_1) beats another card (c_2) on trick (t) *if and only if* one of the following is true:
  - (c_1) and (c_2) are the same suit and value (v_1) (of card (c_1)) is greater than value (v_2) (of card (c_2)):

$$\{\text{Condition 1}\}\ (\text{card\_is\_same\_suit}(c_1, c_2) \land \text{val\_is\_greater}(v_1, v_2))$$

  - (c_1) is apart of the Briscola suit (s) and (c_2) is not the same suit as (c_1):

$$\{\text{Condition 2}\}\ (\text{card\_is\_brisc}(c_1, s) \land \neg\ \text{card\_is\_same\_suit}(c_1, c_2))$$

- (c_1)'s suit (s_1) is apart of the round suit on trick (t) and (c_2) is not apart of the Briscola suit (s) and is not the same suit as card (c_1):

{Condition 3} (card_is_suit_of_round(c_1, s_1, t) ∧ ¬ card_is_brisc(c_2, s) ∧ ¬ card_is_same_suit(c_1, c_2))

- These conditions can then be used in the final form of the constraint:

card_beats_card(c_1, c_2, t)) ↔ {Condition 1} ∨ {Condition 2} ∨ {Condition 3}

Round Flow:

- If a player (p) is not the starting player on trick (t), then the card (c) played by player (p) is not a starting card on trick (t):

¬starting_player(p, t) → ¬starting_player_card(p, c, t)

- If a card (c) played by a player (p) is a starting card on trick (t), that implies player (p) played the card (c) on trick (t):

starting_player_card(p, c, t) → player_plays_card(c, p, t)

- If a card (c) played by a player (p) is a starting card on trick (t), that implies that the suit of the player card (s) is the round suit on trick (t):

starting_player_card(p, c, t) → suit_of_round(s, t)

- If the suit (s) is the round suit of trick (t), then a card (c) that has the suit (s) is a round suit card on trick (t):

suit_of_round(s, t) → card_is_suit_of_round(c, s, t)

- If the suit (s) is not the round suit of trick (t), then a card (c) that has the suit (s) is not a round suit card on trick (t):

¬suit_of_round(s, t) → ¬card_is_suit_of_round(c, s, t)

- If a player (p) does not play the card (c) on trick (t), then the player (p) does not win the trick (t) with the card (c):

¬player_plays_card(c, p, t) → ¬player_wins_trick(c, p, t)

- If a player (p) does not have a card (c) on trick (t), then player (p) does not play that card (c) on trick (t):

¬player_has_card(c, p, t) → ¬player_plays_card(c, p, t)

- If a player (p) wins the trick (t) (with a card (c)), then the player (p) is the starting player of the *next* trick (t + 1):

player_wins_trick(c, p, t) → starting_player(p, t + 1)

- If one player (p_1) plays a card (c_1) on the trick (t) and another player (p_2) plays a different card (c_2) on trick (t), then check if (c_1) beats (c_2) on trick (t). [NOTE: This constraint was added into an "And()" chain command using Bauhaus. The "And()" chain version of this constraint is referred to as "*card_beats_all*" going forward.]
  (The logic with this constraint is that since this proposition is always in an "And()" chain, then if one (`player_plays_card()`) proposition is false, then the statement is vacuously true, meaning the "And()" chain can still hold. It is only ever false if two unique cards are played and if (c_1) does not beat (c_2).)

$$(\text{player\_plays\_card}(c_1, p_1, t) \land \text{player\_plays\_card}(c_2, p_2, t)) \rightarrow \text{card\_beats\_card}(c_1, c_2, t)$$

- If a card (c) does *not* beat every other card played in a trick (t), and a player (p) played card (c) on trick (t), then player (p) does not with trick (t) with their card (c):

$$(\neg \textit{card\_beats\_all} \land \text{player\_plays\_card}(c, p, t)) \rightarrow \neg\text{player\_wins\_trick}(c, p, t)$$

- If a card (c) *does* beat every other card played in a trick, and a player (p) played card (c) on trick (t), then player (p) wins the trick (t) with their card (c):

$$(\textit{card\_beats\_all} \land \text{player\_plays\_card}(c, p, t)) \rightarrow \text{player\_wins\_trick}(c, p, t)$$

Limitations:

- For each player (p), they can only ever play *exactly one* card (c) on a given trick (t). This constraint uses the "exactly one" constraint from Bauhaus:

$$\text{add\_exactly\_one}(\text{player\_plays\_card}(c, p, t))$$

- For a player (p), they can only ever have *exactly one* card (c) that is the starting card of a given trick (t):

$$\text{add\_exactly\_one}(\underline{\text{starting\_player\_card}}(p, c, t))$$

- For each trick (t), there is only *exactly one* suit (s) that is the round suit of the trick:

$$\text{add\_exactly\_one}(\underline{\text{suit\_of\_round}}(s, t))$$

- For each player (p), only *exactly one* of them can be the starting player of a given trick (t):

$$\text{add\_exactly\_one}(\underline{\text{starting\_player}}(p, t))$$

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, the program 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, a new constraint, (`card_is_same_suit(c1, c1)`), was added. 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))
```

Ensuring the Correct Card was Played in a Trick:

Based on the starting hands defined in the summary of this report, the goal was to have the model play four specific cards – one card from each player's hand – and then use what was originally a (`check_trick()`) function to check those four cards, instead of the twelve total cards across the four players' hands. This function was originally implemented with the below code:

```
# Determines if a player plays a card to win a trick or not.
card_win_prop=[]
for prop1 in has_card_prop:
    card1 = prop1.card
    for prop2 in has_card_prop:
        card2 = prop2.card
        if card1 == card2:
            continue
        card_win_prop.append(card_beats_card(card1, card2))
    for player in PLAYERS:
        # NOTE: I had to seperate these constraints to get them to work properly
        # Constaint: If a player doesn't have a card, they can't win the trick with that card
        E.add_constraint((~player_has_card(card1, player, TRICK_NUMBER)) >> ~player_wins_trick(CARDS_IN_PLAY, card1, player, TRICK_NUMBER))
        # Constaint: If a player doesn't have a card, they can't play that card
        E.add_constraint((~player_has_card(card1, player, TRICK_NUMBER)) >> ~player_plays_card(card1, player, TRICK_NUMBER))
        # Constraint: If a player has a card but the card doesn't beat every other card, they don't win the trick that card
        E.add_constraint((~And(card_win_prop) & player_has_card(card1, player, TRICK_NUMBER)) >> ~player_wins_trick(CARDS_IN_PLAY, card1, player, TRICK_NUMBER))
        # Constraint: If a player has a card, it beats other card, and they play the card, they win the trick with that card.
        E.add_constraint((And(card_win_prop) & player_has_card(card1, player, TRICK_NUMBER) & player_plays_card(card1, player, TRICK_NUMBER))
                         >> player_wins_trick(CARDS_IN_PLAY, card1, player, TRICK_NUMBER))
```

It would then output the model below. The issue with this model was that, despite player 3 playing the “J of Clubs” card, the model would still think that they win the trick. This is because since player 3 had the “A of Swords” card in their hand (the strongest card amongst each player’s starting hands), which made the model think player 3 won the trick, even though it wasn’t the card they played.

```
Satisfiable: True
P1's '7 of Cups' is the starting card on trick #1
P1 is the starting player on trick #1
P4 plays the card '3 of Coins' on trick #1
P3 plays the card 'J of Clubs' on trick #1
P2 plays the card 'H of Swords' on trick #1
P1 plays the card '7 of Cups' on trick #1
P3, with their card 'A of Swords', wins the trick #1
```

In order to fix this, the constraints regarding the proposition (`card_beats_card()`) needed to be changed. As the proposition was initially used to compare the cards played in the trick – including if there was only one card played in the round – this caused the problem described above. Thus, the proposition was updated to only be applied when there were at least two played cards in the trick. The implementation of this is seen below, along with an image of the new output that returned the correct winner of the trick:

```
# Determines if a player plays a card to win a trick or not.
card_win_prop=[]
for prop1 in PLAYS_CARD_PROP:
    card1 = prop1.card
    for prop2 in PLAYS_CARD_PROP:
        card2 = prop2.card
        if card1 == card2:
            continue
        card_win_prop.append((player_plays_card(card1, prop1.player, trick_number) & player_plays_card(card2, prop2.player, trick_number)) >> card_beats_card(card1, card2, trick_number))
    for player in PLAYERS.keys():
        # NOTE: I had to seperate these constraints to get them to work properly
        # Constaint: If a player doesn't have a card, they can't win the trick with that card
        E.add_constraint((~player_plays_card(card1, player, trick_number)) >> ~player_wins_trick(CARDS_IN_PLAY, card1, player, trick_number))
        # Constaint: If a player doesn't have a card, they can't play that card
        E.add_constraint((~player_has_card(card1, player, trick_number)) >> ~player_plays_card(card1, player, trick_number))
        # Constraint: If a player has a card but the card doesn't beat every other card, they don't win the trick that card
        E.add_constraint((~And(card_win_prop) & player_plays_card(card1, player, trick_number)) >> ~player_wins_trick(CARDS_IN_PLAY, card1, player, trick_number))
        # Constraint: If a player has a card, it beats other card, and they play the card, they win the trick with that card.
        E.add_constraint((And(card_win_prop) & player_plays_card(card1, player, trick_number))
                         >> player_wins_trick(CARDS_IN_PLAY, card1, player, trick_number))

    # Reset card propositions for each player
    card_win_prop=[]
```

```
Satisfiable: True
P1 is the starting player on trick #1
P1's '2 of Swords' is the starting card on trick #1
P4 plays the card '5 of Cups' on trick #1
P1 plays the card '2 of Swords' on trick #1
P2 plays the card 'A of Coins' on trick #1
P3 plays the card 'J of Clubs' on trick #1
P1, with their card '2 of Swords', wins the trick #1
```

However, it was now found through additional explorations that the model sometimes wouldn’t return a trick winner at all, despite these changes. Luckily, this output helped to identity the issue that was located in what was originally a function named (`update_round_suit()`). To test the

output of this function, a test function originally named (`test_round_suits()`) (which was later renamed to be (`test_player_hands()`)) was created to ensure there were no "Briscola" cards in the trick. Through looking at the tested output, it was eventually found that the round suits were not being set properly to the starting card the model made the player play. This helped to identify a problem with the method used for setting the round suit.

It was found that the program was only setting the suits of a **predetermined** value, which meant that only **one** round suit was being passed in as a value, in essence making it a constant variable when it should be changing every trick. To fix this, a revamp of the (`update_round_suit()`) function was required, which led to the new implementation seen below:

```
def update_round_suit(round_suit):
    start_card_prop=[]
    for prop in HAS_CARD_PROP:
        if (prop.player == start_player.player):
            start_card = starting_player_card(start_player.player, prop.card, trick_number)
            start_card_prop.append(start_card)
            # Constraint: If a starting player plays a certain card, that card is their starting card
            E.add_constraint((start_player & player_plays_card(prop.card, start_player.player, trick_number)) >> start_card)
            # Constraint: A starting card implies its suit is the round suit of the trick
            E.add_constraint(start_card >> suit_of_round(CARDS[prop.card]["suit"], trick_number))



    # Using CARD_DECK instead of CARDS to sightly cut processing time down for this function over time.
    for card1 in CARD_DECK:
        # Constraint: If the suit of a round is a card's suit, then the card is apart of the suit of the round
        E.add_constraint(suit_of_round(CARDS[card1]["suit"], trick_number) >> (card_is_suit_of_round(card1, CARDS[card1]["suit"], trick_number)))
        #TODO: Make the round suits dynamic
        '''
        # Initalizes card suits as round suits or not (this code may need to be tweaked later for more dynamic round suits)
        if CARDS[card1]["suit"] == round_suit:
            # Constraint: If a card is apart of the suit that is the current round suit, it is a round suit card in our round.
            E.add_constraint((card_is_suit_of_round(card1, round_suit, trick_number)))
        else:
            # Constraint: Otherwise, the card is not apart of the current round suit
            E.add_constraint((~card_is_suit_of_round(card1, round_suit, trick_number)))
        '''
        for card2 in CARD_DECK:
            if card1 == card2:
                continue
            # Make all the propositions variables (this is just more consise, since this is a long constraint)
```

In addition, a constraint was added into (`update_round_suit()`) to ensure that a suit that is not the round suit implies that a card of that suit is **not** a round-suit card, which fixed the model and provided the correct output seen below:

```
Satisfiable: True
P1's '2 of Coins' is the starting card on trick #1
P1 is the starting player on trick #1
P2 plays the card 'H of Cups' on trick #1
P1 plays the card '2 of Coins' on trick #1
P4 plays the card '3 of Coins' on trick #1
P3 plays the card 'J of Clubs' on trick #1
P4, with their card '3 of Coins', wins the trick #1 of configuration
'Coins', the current round suit of trick #1.
'Coins' is apart of 'Coins', the current round suit of trick #1.
```

Testing Deck Size, Drawing, and Transitivity:

For the model, it needed to be ensured that the implementation could work if there was a restricted deck size, and that value transitivity worked properly. To test this, a game was run where only cards of the Clubs and Coins suits were present in the deck, with the first half of the deck being each Club card in ascending value order, followed by each Coin card in ascending value order. As the deck was cut in half, this game would consist of 5 tricks instead of the usual 10 tricks. Below is an image of the modified deck.

```
sample_deck2 = ['2 of Clubs', '4 of Clubs', '5 of Clubs', '6 of Clubs','7 of Clubs',
        'J of Clubs', 'H of Clubs','K of Clubs','3 of Clubs', 'A of Clubs', '2 of Coins',
        '4 of Coins', '5 of Coins', '6 of Coins','7 of Coins', 'J of Coins', 'H of Coins',
        'K of Coins','3 of Coins', 'A of Coins']
```

With this modified deck, it was expected that player-1 should win one trick, player-3 should win one trick, and player-4 should win the remaining three tricks. This is because, with our implementation of Briscola, players should draw cards right after they play a card. Based on this rule, it should lead to the above outcomes with the above deck.

Throughout each trick, the output determined that the player with the highest value card was the one that won the trick – proving that the value transitivity was working properly. However, the model determined that player-4 was the winner of all 5 tricks when it should have only won 3 tricks.

```
----------------------------------------------------------------
Players are out of cards, so the game is over.
----------------------------------------------------------------
Final Results:
P1 wins: 0
P2 wins: 0
P3 wins: 0
P4 wins: 5

The winner of this game is P4!
```

This output revealed that the proper deck drawing rule had not been implemented correctly, as it turned out that player-1 was drawing first every time. To fix this, the part of the implementation that draws the cards for each player was adjusted, as seen below. Where the top image is the implementation before it was adjusted, and the bottom image is the implementation after it was adjusted:

```
# Code that allows players to draw cards
for player1 in PLAYERS.keys():
    if len(CARD_DECK) != 0:
        drawn_card = CARD_DECK.pop(0)
        HAS_CARD_PROP.append(player_has_card(drawn_card, player1, (trick_number + 1)))
        # Constraint: Each player has the card that they have in their hand
        E.add_constraint(player_has_card(drawn_card, player1, (trick_number + 1)))
        E.add_constraint(player_draws_card(drawn_card, player1, trick_number))
        for player2 in PLAYERS.keys():
            if player1 == player2:
                continue
            # Constraint: A second player does not have a card another player already has in their hand
            E.add_constraint(~player_has_card(drawn_card, player2, (trick_number + 1)))

    start_player_prop.append(starting_player(player1, (trick_number + 1)))
```

```
start_player_i = 0
# Code that allows players to draw cards
for player in PLAYERS.keys():
    # First, find the starting player index
    if player == start_player.player:
        break
    else:
        start_player_i += 1

# Use a for loop, since we can't use indexes on dictionaries
for i in range(4):
    player1 = f"P{start_player_i + 1}"
    if len(CARD_DECK) != 0:
        drawn_card = CARD_DECK.pop(0)
        HAS_CARD_PROP.append(player_has_card(drawn_card, player1, (trick_number + 1)))
        # Constraint: Each player has the card that they have in their hand
        E.add_constraint(player_has_card(drawn_card, player1, (trick_number + 1)))
        E.add_constraint(player_draws_card(drawn_card, player1, trick_number))
        for player2 in PLAYERS.keys():
            if player1 == player2:
                continue
            # Constraint: A second player does not have a card another player already has in their hand
            E.add_constraint(~player_has_card(drawn_card, player2, (trick_number + 1)))

    start_player_prop.append(starting_player(player1, (trick_number + 1)))
    # Make sure that start_player_i is in range
    start_player_i = (start_player_i + 1) % 4
```

Re-running the model with the same test cases as before now gave the expected output for who won, as seen below:

```
------------------------------------------------------------------
Players are out of cards, so the game is over.
------------------------------------------------------------------
Final Results:
P1 wins: 1
P2 wins: 0
P3 wins: 1
P4 wins: 3
```

Jape Proof Ideas

The Briscola Suit is the Highest Priority Suit:

In a trick where the first three players played non-Briscola Suit cards, and the last player plays a Briscola Suit card, then the last player wins the trick, regardless of any of the cards' values.

The following assumptions are used to implement the premises and conclusion into the JAPE proof:

- For all played cards (x), the played card (PC(x)) is either apart of the Briscola suit (B(x)) or is not apart of the Briscola suit (¬B(x)). These are represented by the following two conjunctions:

$$(\forall x.(PC(x) \wedge \neg B(x))) \qquad (\forall x.(PC(x) \wedge B(x)))$$

- The initial configuration of the four played cards is then established below:

$$(\text{actual i1})\ (\text{actual i2})\ (\text{actual i3})\ (\text{actual i4})$$

- Then, the initial configuration of the game is established, where three of the four players played non-Briscola Suit cards, and the last player played a Briscola Suit card, which determines the winner using the predicate (PW(y)), where (y) is the player that won:

$$(\ (PC(i1) \wedge \neg B(i1)) \wedge (PC(i2) \wedge \neg B(i2)) \wedge (PC(i3) \wedge \neg B(i3)) \wedge (PC(i4) \wedge B(i4))\ ) \rightarrow PW(i4)$$

- We then want to prove that the last player – player-4 – was the one that won the round. So, the conclusion is:

$$PW(i4)$$

The full JAPE proof is seen below:

1:	actual i1, actual i2, actual i3, actual i4	premises
2:	∀x.(PC(x)∧¬B(x)), ∀x.(PC(x)∧B(x))	premises
3:	((PC(i1)∧¬B(i1))∧(PC(i2)∧¬B(i2))∧(PC(i3)∧¬B(i3))∧(PC(i4)∧B(i4))→PW(i4))	premise
4:	PC(i4)∧B(i4)	∀ elim 2.2,1.4
5:	PC(i3)∧¬B(i3)	∀ elim 2.1,1.3
6:	PC(i2)∧¬B(i2)	∀ elim 2.1,1.2
7:	PC(i1)∧¬B(i1)	∀ elim 2.1,1.1
8:	(PC(i1)∧¬B(i1))∧(PC(i2)∧¬B(i2))	∧ intro 7,6
9:	(PC(i1)∧¬B(i1))∧(PC(i2)∧¬B(i2))∧(PC(i3)∧¬B(i3))	∧ intro 8,5
10:	(PC(i1)∧¬B(i1))∧(PC(i2)∧¬B(i2))∧(PC(i3)∧¬B(i3))∧(PC(i4)∧B(i4))	∧ intro 9,4
11:	PW(i4)	→ elim 3,10

Determining the Winner of a Trick:

If only **one** of the four players can win a trick, and three of those players lost, then the only other player must have won the trick.

The following assumptions are used to implement the premises and conclusion into the JAPE proof:

- The three players losing the trick are represented by the following conjunction:

$$(\neg P2 \wedge \neg P3 \wedge \neg P4)$$

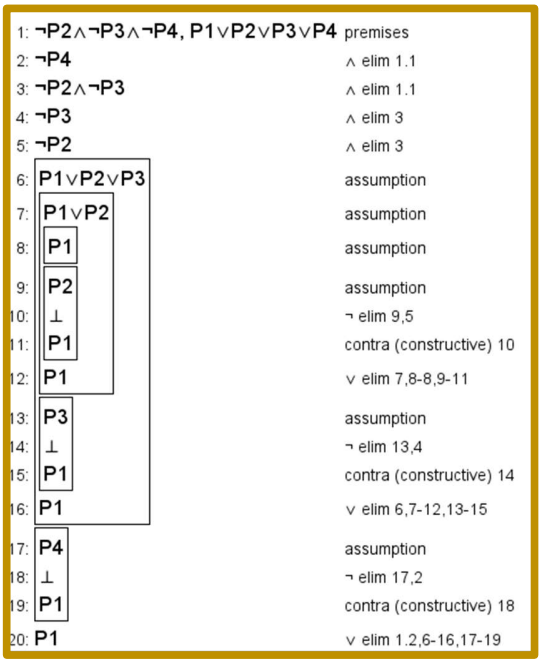
- Having one of the players win the trick is represented by the following disjunction:

$$(P1 \vee P2 \vee P3 \vee P4)$$

- We want to prove that the only player that didn't lose was the one that won the round. This means we want to prove the below conclusion:

$$(P1)$$

The full JAPE proof is seen below:



Players only play their Strongest or Weakest Card:

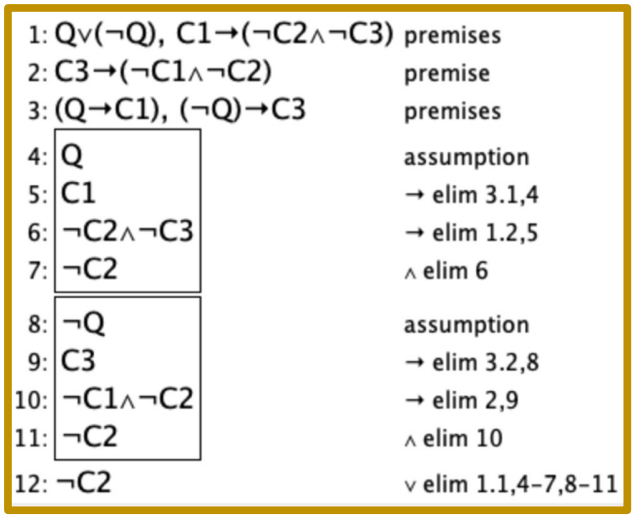
In Briscola, players can play exactly one card in a trick. In order for a player (p) – who is finishing the trick – to win the trick, their card must be stronger than the current card winning the trick. If (p) can win the trick, they will play their strongest card (C1). If not, they will play their weakest card (C3). Thus, there is no situation where (p) plays their card (C2) that isn't the strongest or weakest card in their hand.

The following assumptions are used to implement the premises and conclusion into the JAPE proof:

- "If the player (p) has a card that can beat the card currently winning the trick" is represented by the proposition (Q).
- "If (Q) is true, then (p) will play their strongest card" is represented by $(Q \rightarrow C1)$.
- "If (Q) is false, then (p) will play their weakest card" is represented by $(\neg Q \rightarrow C3)$.

- "If (p) plays their strongest card, they won't play their other two cards" is represented by (C1→(¬C2∧¬C3))
- "If (p) plays their weakest card, they won't play their other two cards" is represented by (C3→(¬C1∧¬C2))
- We want to prove that the player's card that isn't their strongest or weakest card will never be played. Thus, we want to prove (¬C2).

The full JAPE proof is seen below:



First-Order Extension

In a predicate logic setting we can easily convert each proposition into a predicate format:

- $startingPlayer(p, t)$ – Player (p) starts trick (t).
- $valIsGreater(v_1, v_2)$ – Value (v_1) is greater than value (v_2).
- $cardIsBrisc(c, s)$ – Card (c) is the same suit as Briscola suit (s).
- $cardIsSuitOfRound(c, s, t)$ – (c) is the same suit as (s), where (s) is the round suit on trick (t).
- $cardIsSameSuit(c_1, c_2)$ – Card (c_1) is the same suit as card (c_2).
- $cardBeatsCard(c_1, c_2, t)$ – Card (c_1) beats card (c_2) in terms of value and suit on trick (t).
- $playerHasCard(c, p, t)$ – Player (p) has the card (c) during the trick (t).
- $playerWinsTrick(p, t, c)$ – Player (p) wins the trick (t) with a card (c).

- $suitOfRound(s,t)$ – Checks if the suit (s) is the round suit of the trick (t).
- $playerPlaysCard(c,p,t)$ – Player (p) played a card (c) on the trick (t)
- $startingPlayerCard(c,t)$ - Card (c) is the starting card on the trick (t).

We can then implement some of the constraints into first-order logic using some of our predicates defined above with the appropriate quantifiers. Some examples of these first-order constraints are seen below:

Card Values are Transitive:

If a card's value (v_1) is greater than another card's value (v_2). Then, if (v_2) is greater than another card's value (v_3), then (v_1) is greater than (v_3).

$$\exists v_1.\exists v_2.\exists v_3.((valIsGreater(v_1,v_2) \wedge valIsGreater(v_2,v_3)) \rightarrow valIsGreater(v_1,v_3))$$

Conditions for Cards Beating Each Other:

A card (c_1) beats another card (c_2) on all tricks (t) *if and only if* **one** of three conditions are met. This constraint can then be broken down into these three conditions:

- (c_1) and (c_2) are the same suit and the value (v_1) (of card (c_1)) is greater than the value (v_2) (of card (c_2)):

$$\{1\} \quad \exists c_1.\exists c_2.\exists v_1.\exists v_2.\left(cardIsSameSuit(c_1,c_2) \wedge valIsGreater(v_1,v_2)\right)$$

- (c_1) is apart of the Briscola suit (s) and (c_2) is not the same suit as (c_1):

$$\{2\} \quad \exists c_1.\exists c_2.\exists s.\left(cardIsBrisc(c_1,s) \wedge \neg\, cardIsSameSuit(c_1,c_2)\right)$$

- (c_1)'s suit (s_1) is apart of the round suit on trick (t) and (c_2) is not apart of the Briscola suit (s) and is not the same suit as card (c_1):

$$\{3\} \quad \exists c_1.\exists c_2.\forall t.\exists s.\exists s_1.\left(cardIsSuitOfRound(c_1,s_1,t) \wedge \neg cardIsBrisc(c_2,s) \wedge \neg cardIsSameSuit(c_1,c_2)\right)$$

These three conditions can then be put into the final constraint seen below:

$$\exists c_1.\exists c_2.\forall t.cardBeatsCard(c_1,c_2,t) \leftrightarrow \{1\} \vee \{2\} \vee \{3\}$$

Players Can't Play a Card They Don't Have:

For all players (p), if the player doesn't have a card (c), then they can't play this card during any trick (t):

$$\forall t.\forall p.\forall c.(\neg playerHasCard(c,p,t) \rightarrow \neg playerPlaysCard(c,p,t))$$