Queen's UNIVERSITY

GRP_9: Winning UNO

Joanne Mabior (23YFC1)

Kanika Poonia (22LC53)

Michelle Shi (22TJ18)

Thu Thu Tun (22BBN6)

Course Modelling Project

CISC/CMPE 204

Logic for Computing Science

December 5, 2024

Abstract

This project models a simulation of a well-known card game, UNO. In this version of UNO, the player starts with 7 cards in their hand and must play one of these cards depending on the previously played card (i.e. the top card). This is determined through the colour and number or icon of the top card. A card is playable if one of the following holds:

- The colour of the card matches the colour of the top card
- The number/icon of the card matches the number/icon of the top card
- They have a wildcard, in which case they can place it down whenever they want.

Otherwise, the player must draw another card from the deck.

Additionally, UNO has certain action cards:

- **Skip**: if a player places down a Skip card, then the next player's turn is skipped.
- **Reverse**: if a player places down a Reverse card, then the order in which the people play reverses, and it becomes the previous player's turn again.
- **Draw 2**: if a player places down a Draw 2 card, then the next player must draw 2 cards from the deck and misses their turn. **However**, if the next player also has a Draw 2 card, then they can place this card instead of drawing to accumulate the number of Draw 2 cards to the next player (i.e. the next player will have to draw 4 cards).
- **Draw 4 (Wild)**: similar rules apply here; however, not only does the next player have to draw 4 cards from the deck, but the current player can also change the current colour of the game.
- **Wild card**: if a player places down a "normal" wild card, then they can change the current colour of the game.

The objective of this game is to get rid of all of one's cards first, which is something the code will determine. However, in our model, we will deem the sequence of play satisfiable and the game ended as long as the player loses all of their cards at the end.

If further clarification on the rules is needed, a more detailed overview of the game can be found here: https://en.wikipedia.org/wiki/Uno_(card_game)

Propositions

Below is a list of variables that we will use and their assigned meanings.

Variable Name	**Representation**
A	The player has cards in hand.
C	A matching colour of the previous card exists in hand. Note that wildcards always have a matching colour with the previous card.
N	A matching number or icon of the previous card exists in hand.
R	The previous card is a "Reverse" card.
S	The previous card is a "Skip" card.
D2	The previous card is a "Draw 2" card.
D4	The previous card is a (wild) "Draw 4" card.
W	The player has a colour-changing wild card in hand.

Next, we have a list of propositions and their corresponding logic translation.

Proposition	**Meaning**	**Logic Translation**
can_play(A, C, N)	True if the player has a matching colour or number/icon card as the previous card.	$A \land (C \lor N)$
can_play_multiple(N)	True if the player has multiple cards of the same number in hand.	N
must_draw_single(A, C, N)	True if the player has to draw a single card.	$A \land \neg (C \lor N)$
must_draw_multiple(A, C, N)	True if the player has to draw multiple cards.	$A \land (D2 \lor D4) \land \neg (C \lor N)$

can_accumulate_d2(A, D2, N)	True if the previous card was D2 and player has a D2 in hand (i.e. has the same icon as previous card).	$A \wedge D2 \wedge N$
can_accumulate_d4(A, D4, N)	True if the previous card was D4 and player has a D4 in hand (i.e. has the same icon as previous card).	$A \wedge D4 \wedge N$
game_ended(A)	True if the player has no cards in hand left.	$\neg A$

Constraints

The following is a list of constraints used in our model and their English interpretation.

Constraint	**Interpretation**
$\neg$ can_play(A, C, N) $\rightarrow$ must_draw_single(C, N)	If the player does not have any playable cards at their disposal, they must draw a card from the pile.
can_play(A, C, N) $\rightarrow$ $\neg$ game_ended(a)	If the player can still play, then the game hasn't ended.
$R \vee S \rightarrow \neg$ can_play(A, C, N)	If the previous card played was a reverse or skip card, then the player cannot play.
$D2 \wedge \neg$ can_accumulate_d2(A, D2, N) $\rightarrow$ must_draw_multiple(A, C, N)	If the player does not have any playable cards after an accumulation of Draw 2 cards were played, then the player must draw the total number of cards shown.
$D4 \wedge \neg$ can_accumulate_d4(A, D4, N) $\rightarrow$ must_draw_multiple(A, C, N)	If the player does not have any playable cards after an accumulation of Draw 4 cards were played, then the player must draw the total number of cards shown.

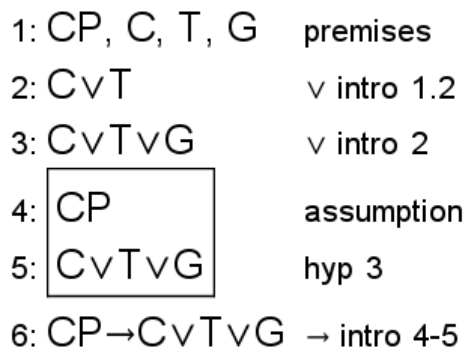
Jape Proofs

There were a few conflicts when it came to using our predetermined propositions in Japе, so the following changes have been made to accommodate:

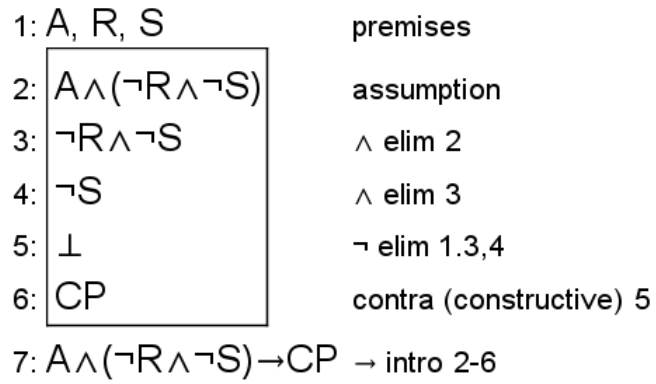
- N for card number becomes T in Jape
- W for wild card becomes G in Jape
- can_play(A, C, N) becomes CP in Jape
- must_draw_multiple(A, C, N) becomes DM in Jape
- can_accumulate_d2(A, D2, N) becomes A2 in Jape
- can_accumulate_d4(A, D4, N) becomes A4 in Jape

And with that, here are our three Jape proofs:

Proof 1: This proof demonstrates the requirements for a player to play. If the player can play a card, this implies that they either have a matching colour card, matching number/icon card, or a wildcard.



Proof 2: This proof describes the restrictions of a player's ability to play a card. As long as they still have cards in their hand and the previous card is not a reverse and is not a skip either, the player can play a card.



Proof 3: This proof represents one of our above constraints; if the player cannot place down a Draw 2/4 card to deflect the accumulation of Draw 2/4 cards in the pile, they cannot play their turn and instead must draw from the deck however many cards the pile has accumulated to.

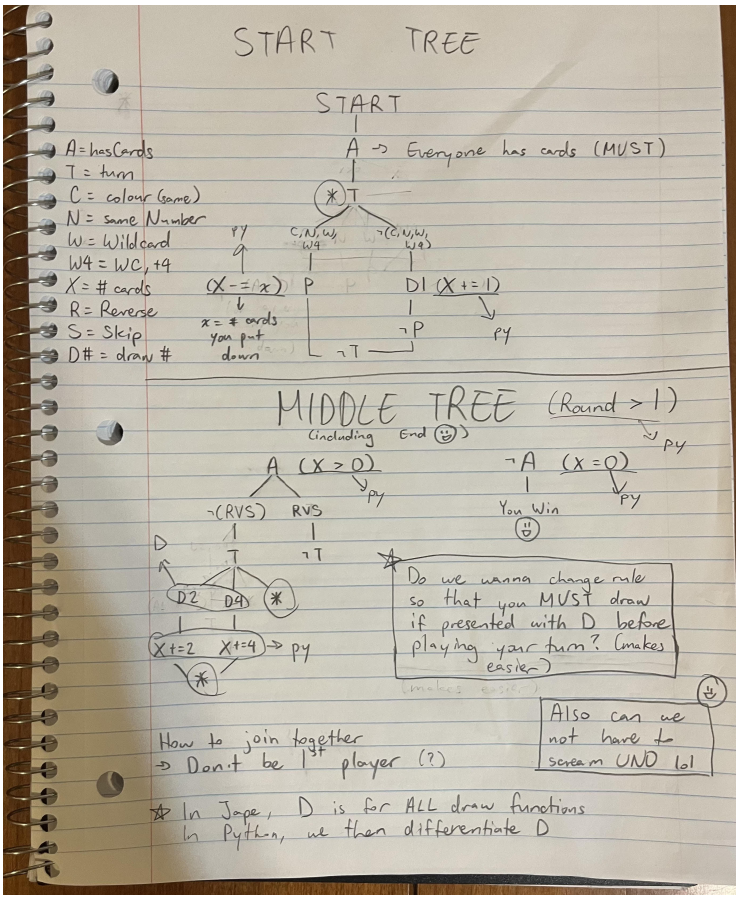
1:	D2, A2, D4, A4, DM	premises
2:	(D2∧¬A2)∨(D4∧¬A4)	assumption
3:	D2∧¬A2	assumption
4:	¬A2	∧ elim 3
5:	⊥	¬ elim 1.2,4
6:	¬CP	contra (constructive) 5
7:	D4∧¬A4	assumption
8:	¬A4	∧ elim 7
9:	⊥	¬ elim 1.4,8
10:	¬CP	contra (constructive) 9
11:	¬CP	∨ elim 2,3-6,7-10
12:	¬CP∧DM	∧ intro 11,1.5
13:	(D2∧¬A2)∨(D4∧¬A4)→¬CP∧DM	→ intro 2-12

Model Exploration

Our model has underwent many major changes as we revised and figured out the optimal way to go about this project.

During the beginning stages of our model, we decided to consider multiple players, steps they take throughout the game, how every player draws or puts down a card. Through this, we then decided upon a list of propositions and constraints that mirror the original gameplay.

We came up with a tree model to connect all our variables together to figure out how our game will play out, from start to middle to finish. Through this, we figured out unnecessary variables and potential complications for our project. This tree drawing can be found below.



Later, after getting feedback from the proposal, we decided to shift our model's perspective to follow an individual player's gameplay for simplicity rather than an entire group, so that we can make specific propositions and constraints based on how the player reacts to the game.

As a "demonstration" or "something to go off of", we played a game of UNO and recorded the moves of the winning person and the cards played immediately before her.

Once we really got the hang of what our main objective was, it was time to weed out the unnecessary variables, introduce new ones, and come up with new, better propositions and constraints.

Fast forward a week later, we realized that there was still a lot of work to do. We started off by completely revamping our list of propositions, clarifying the difference between our variables and function-like propositions that took in said variables as "parameters". Doing so greatly decreased the difficulty of creating comprehensible constraints and coding our model.

Additionally, our code initially lacked propositions to represent the cards

played previously. Without this information, it was hard to figure out how well the constraints captured the interactions between the current and previous cards. So we introduced propositions specifically for the previously played cards to better track their color and number/icon, like so:

```
@constraint.exactly_one(E)
@proposition(E)
class PrevColorPropositions():
    def __init__(self, data):
        self.data = data

    def _prop_name(self):
        return f"PrevColor.{self.data}"

# Previous card is either red, blue, green or yellow in color.
PCCR = PrevColorPropositions("PCCR")
PCCB = PrevColorPropositions("PCCB")
PCCG = PrevColorPropositions("PCCG")
PCCY = PrevColorPropositions("PCCY")

# Previous Number proposition to check if the cards played previously are a specific number/icon or not.
@constraint.exactly_one(E)
@proposition(E)
class PrevNumberPropositions():
    def __init__(self, data):
        self.data = data

    def _prop_name(self):
        return f"PrevNumber.{self.data}"
# Previous card has either 0-9, skip, draw4, draw2, wild
PCN0 = PrevNumberPropositions("PCN0") # Card has number 0
PCN1 = PrevNumberPropositions("PCN1") # Card has number 1
PCN2 = PrevNumberPropositions("PCN2") # Card has number 2
PCN3 = PrevNumberPropositions("PCN3") # Card has number 3
PCN4 = PrevNumberPropositions("PCN4") # Card has number 4
PCN5 = PrevNumberPropositions("PCN5") # Card has number 5
PCN6 = PrevNumberPropositions("PCN6")
PCN7 = PrevNumberPropositions("PCN7") # Card has number 7
PCN8 = PrevNumberPropositions("PCN8") # Card has number 8
PCN9 = PrevNumberPropositions("PCN9") # Card has number 9
PCNS = PrevNumberPropositions("PCNS") # Card is a skip card
PCND4 = PrevNumberPropositions("PCND4") # Card is a Draw4 card
PCND2 = PrevNumberPropositions("PCND2") # Card is a Draw2 card
PCNW = PrevNumberPropositions("PCNW") # Card is a wild card
```

Furthermore, our model used to consider all previous cards and current cards simultaneously, treating every card's properties as active constraints at the same time. While we originally thought this was a good idea because it addressed everything, we realized that it was too restricting on our model. To fix this, we simply checked all cards against one previous card at a time. By doing this, the number of solutions increased, which was what we were hoping for!

```
temporary_constraints = []

for card in previous_cards:
    # Adding color constraints if valid
    if card['color'] in color_constraints:
        constraint = color_constraints[card['color']]
        E.add_constraint(constraint)
        temporary_constraints.append(constraint)

    # Adding number constraints if valid
    if card['number'] in number_constraints:
        constraint = number_constraints[card['number']]
        E.add_constraint(constraint)
        temporary_constraints.append(constraint)

can_play_card(temporary_constraints, temporary_constraints)

for constraint in temporary_constraints:
    E.clear_constraints()
```

We also took into account the card features. We originally left out the notion that each card could only have one colour and one number. Without this restriction, the model allowed unrealistic scenarios where a card could simultaneously represent multiple colors or numbers, leading to invalid gameplay interpretations. Obviously, we didn't want that, so we implemented the constraint.exactly_one() feature.

```
@constraint.exactly_one(E)
@proposition(E)
class ColorPropositions():
    def __init__(self, data):
        self.data = data

    def _prop_name(self):
        return f"Color.{self.data}"

@constraint.exactly_one(E)
@proposition(E)
class NumberPropositions():
    def __init__(self, data):
        self.data = data

    def _prop_name(self):
        return f"Number.{self.data}"

@constraint.exactly_one(E)
@proposition(E)
class PrevColorPropositions():
    def __init__(self, data):
        self.data = data

    def _prop_name(self):
        return f"PrevColor.{self.data}"

@constraint.exactly_one(E)
@proposition(E)
class PrevNumberPropositions():
    def __init__(self, data):
        self.data = data

    def _prop_name(self):
        return f"PrevNumber.{self.data}"
```

Later on, we decided to change up the way we addressed the Draw 2 and Draw 4 cards. Initially, our model included functions like draw_2_was_played() and draw_4_was_played() to check if the previous card was a Draw 2 or Draw 4. However, as we refined the model, we realized that these functions were redundant because we already had propositions explicitly describing whether a Draw 2 or Draw 4 had been played.

Finally, we removed redundant constraints. We did this by experimenting with commenting out individual constraints and removing those that didn't have an effect on the final number of solutions.

First-Order Extension

The way we decided to approach this is to first redefine a bunch of propositions as predicates, with a focus on two new variables: x as a current player's card and y as the previous played card.

Below is a list of predicates that we defined, along with their meanings.

Predicate	**Meaning**
$\mathrm{A}(x)$	x is a card in hand.
$\mathrm{C}(x, y)$	x has the same colour as y.
$\mathrm{N}(x, y)$	x has the same number/icon as y.
$\mathrm{S}(y)$	y skips the next player.
$\mathrm{R}(y)$	y reverses the order of the play.
$\mathrm{P}(x)$	x is a playable card.
$\mathrm{D}(x)$	x has to be drawn.

From here, we came up with a few formulas to demonstrate the relationships between these predicates.

Formula 1: For all previous cards y, there exists a playable card x that implies a matching colour or number/icon between x and y.

$$\forall y \, \exists x \, (P(x) \rightarrow (C(x, y) \lor N(x, y)))$$

Formula 2: For all previous cards y, those that are skip or reverse cards imply unplayable cards for all current cards x.

$$\forall y \, (S(y) \lor R(y)) \rightarrow \forall x \, (\neg P(x))$$

Formula 3: If all cards in hand x are unplayable, then the player must draw another card from the deck.

$$\forall x \, ((A(x) \land \neg P(x)) \rightarrow D(x))$$

Formula 4: Every time a player draws a card x, they cannot play their turn.

$$\forall x (D(x) \rightarrow \neg P(x))$$

Formula 5: For all current cards x, the nonexistence of such x in the player's hand implies that the player has won.

$$\forall x (\neg A(x) \rightarrow \textbf{Win})$$