LOTR and UNO

David Adamashvili
dadamashvili3@gatech.edu

1 LORD OF THE RINGS

Let us make the problem more straightforward, since the wording is very broad. We have three items on the left side of a river bank, Frodo, Gollum and the Ring, henceforth abbreviated as F, G and R. The river can transport one or none of these items. Also, F and G cannot be alone in the same side of the river, unless the boat is also there. Same goes for R and G. Since Sam will always be on the boat, I will equate the boat and Sam and refer to both as the boat B. The boat is initially on the left bank.

1.1 State representation

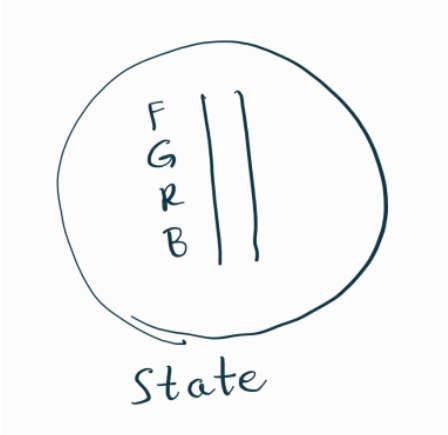

Figure 1 - **State**

Each state (node) of the semantic network will consist of information about the location of Frodo, Gollum, the Ring and Sam/boat. We will visually represent this information with the river and the items on each of its banks. Figure 1 shows the initial state of the problem.

1.2 State transition

State transitions will consist of moving items from the bank where the boat currently resides. When we write "Move B, R", this means that the items B and R will be moved from the bank where the boat is to the other bank.

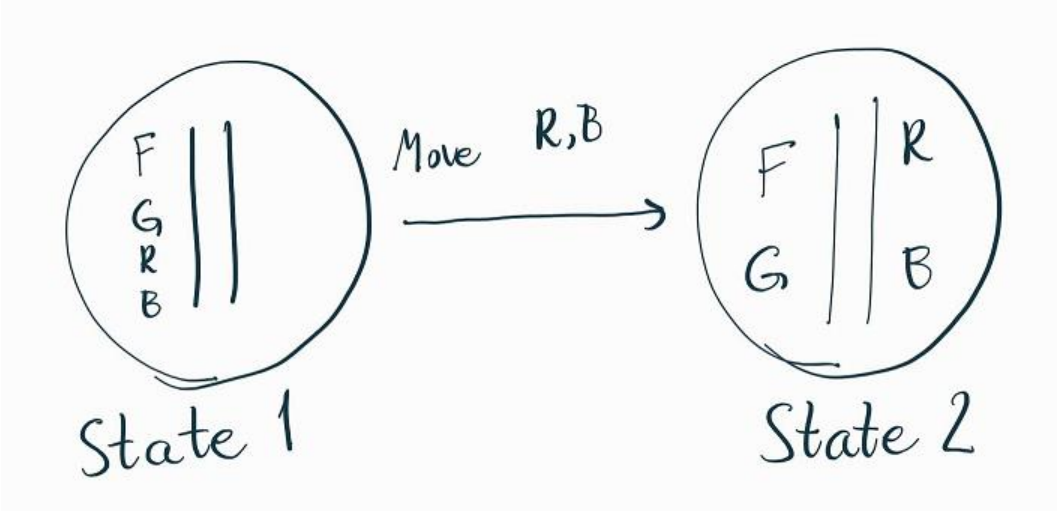

Figure 2 - **Transition**

1.3 Semantic network

Our network will be smart enough as to not do the same transition twice in a row, as this will always be redundant. The initial state is in the top left of Figure 3. The destination state is green. If the generated states end up being invalid, they will be crossed out in the diagram.

States will be invalid if they have been already visited or if R ends up alone with either F or G on the same side of the river bank.

Keeping this in mind, let us look at the entire semantic network for this problem:

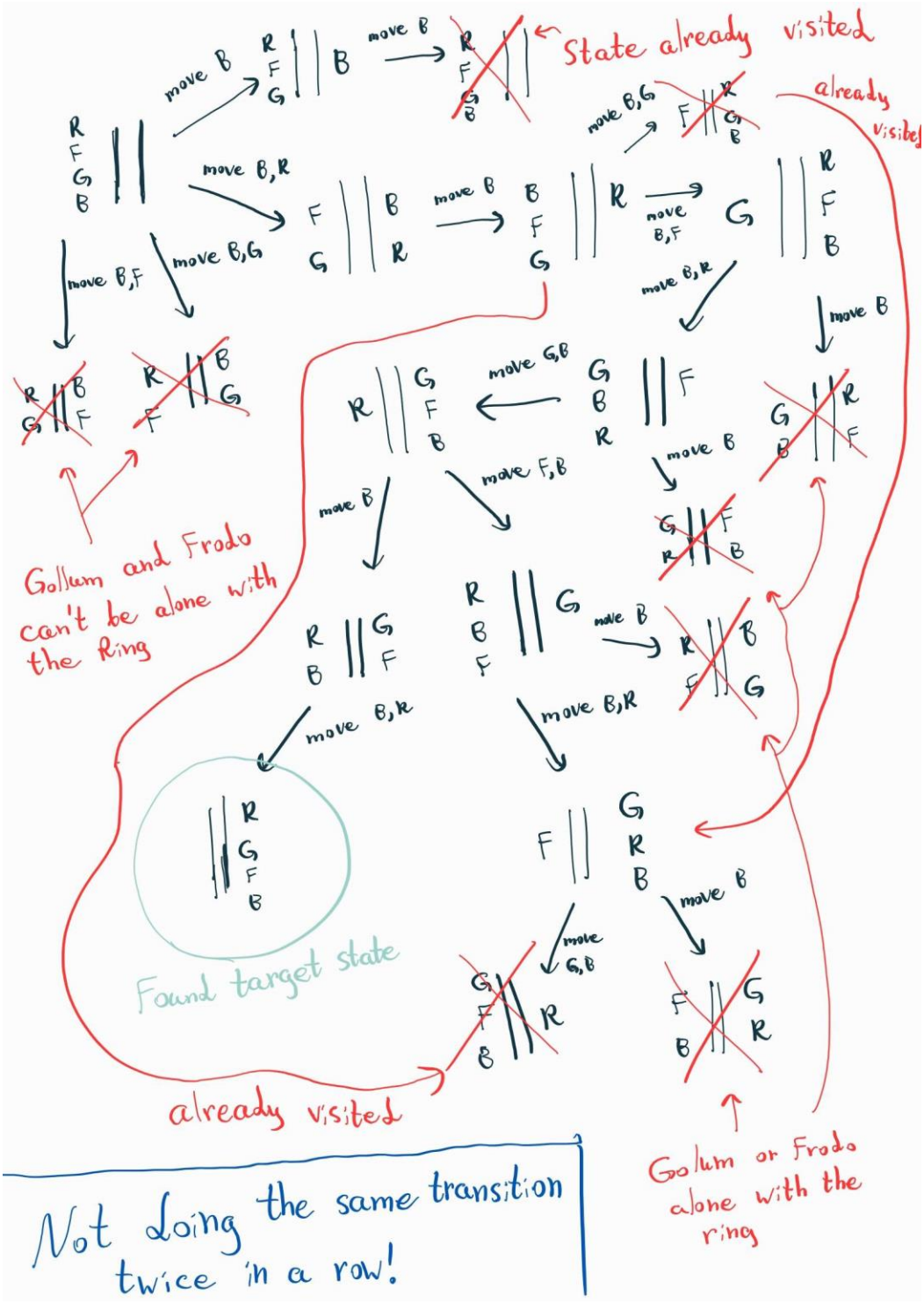

Figure 3 - **Semantic network**

2 UNO

We are required to design an intelligent agent that will play the famous card game UNO. While far from optimal, the agent that I have designed should be far better than just randomly choosing cards to play. We assume that the agent is already intelligent enough to memorize every move made by every player.

An improvement that comes to mind is as follows:

We know that there are 25 of each color and 8 of each number (except 0). We could use this to create a Bayesian model and make it optimal in an information theoretic context. This is not something that I will do for the homework assignment, but should be kept in mind anyway. As in, what is the probability that the next player will have a certain number or a color given the log of the game.

2.1 Pseudocode explanation

The **play** function is an abstraction over playing a card. The **draw** function is an abstraction over drawing a card. Both of them mutate the **PlayedColorsOrderedMap** and **PlayedNumbersOrderedMap** maps. These two are frequency maps that we can iterate from the largest to the smallest. They are basically balanced trees with iterators (imagine **ordered_map** from **C++ STL**). At the beginning of the game, they contain the agent's hand and the top card of the deck. For example, if the game begins and the agent is holding the following hand:

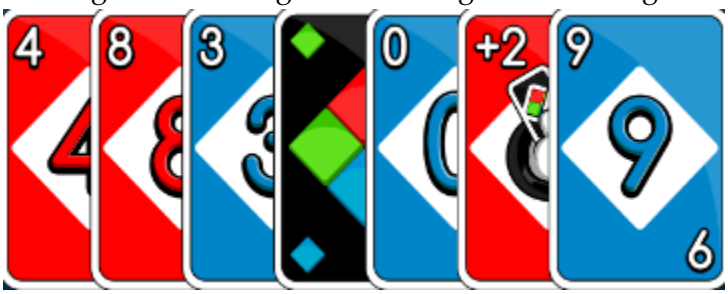

Figure 4 - **Potential initial hand**

And the deck looks like this:

Figure 5 - **Potential initial deck**

Then the initial values of the maps would be:

PlayedColorsOrderedMap = {red: 4, blue: 3, green: 0, yellow: 1}

PlayedNumbersOrderedMap = {0:1, 1:0, 2:0, 3:1, 4:1, 5:0, 6:0, 7:0, 8:2, 9:1}

Whenever the cards get reshuffled, anyone draws or plays a card, these maps get updated automatically. I will not be implementing this in my pseudocode.

ChooseColor and **ChooseNumber** are functions that try to choose a good color/number to play. They run through the maps from the highest frequency to the lowest frequency, and if they find a color or a number in the agent's hand, they return the first one they come across.

Agent.cards is the cards the agent has in its hand. **Deck.top()** is the top card of the deck (the one that's visible).

2.2 Agent explanation and strategy

Keep in mind that, compared to an average human, an agent can memorize cards that have been played in the past, and combined with the cards in its own hand, deduce the least likely colors/numbers the opponents would have.

The agent has been designed to use three key observations about the game state.

1. The game log to find a card in the agent's hand that will make other players draw. We will keep this information in the frequency maps and along with the next point in the list, we will use them to choose a card to play.
2. The total number of each color of card is 25, and the total number of each card number is 8 (except for 0, which there are 4 of). We can use these numbers to calculate the ratio of a given color/number that the opponents cannot have in hand. Whichever ratio is greater is the one that we will try to play.
3. That it is sometimes better to hang onto a +4 wild card. A heuristic that I am using is that, we should play a +4 wild card only when the next player has less than 4 cards in their hand. This is more useful when there are more than 2 players playing the game. The color we choose, again, depends on the log.

So, the strategy is as follows: first we try to play the +4 wildcard by checking if the next player has less than 4 cards in hand. If they do, play it and choose the color based on **ChooseColor**. If null, it will choose randomly. Then we try playing the normal wildcard using the same method. Afterwards we look for any special card (+2, reverse, skip…) in the agent's hand that is the same color or type as the one on top of the deck and play it. If there is no such special card, then we look for a normal numbered card.

What if we have a tie? As in, we have a card with the same color and a different card with the same number as the top card. This is when the number and color ratios come into play. We again look for a candidate number using **ChooseColor** and a candidate number using **ChooseNumber**, and divide the first one by 25 and the second one by either 8 or 4 (depending on whether the top card is a zero or not). This will give us the ratio of each that cannot be in the opponents' hands. We will play whichever is greater.

If all of the previous fail, draw a card.

3 APPENDICES

3.1 Pseudocode

```
# increasing order. whenever a card is played, this is reflected in the sets
PlayedColorsOrderedMap = {}
PlayedNumbersOrderedMap = {}

ChooseColor():
        # iterate in decreasing order
        for color in PlayedColorsOrderedSet:
                if color in Agent.cards.colors:
                        return color

        return null

ChooseNumber():
        # iterate in decreasing order
        for number in PlayedNumbersOrderedSet:
                if number in Agent.cards.numbers:
                        return number

        return null
```

Figure 6 - **Helper data structures and functions**

```
# game loop
while(GameInProgress and AgentTurn):
        if Agent.cards.length == 1:
                # shout uno
                Uno()

        WildPlus4 = Agent.cards.findWildPlus4()
        if WildPlus4 and Agent.NextPlayer.cards.length < 4:
                color = ChooseColor()
                if color != null:
                        play(WildPlus4,color)
                        continue

        Wild = Agent.cards.findWild()
        if Wild:
                color = ChooseColor()
                if color != null:
                        play(Wild,color)
                        continue

        # if we have a special card of the same color as the top card, play it
        # doesn't matter which one
        SpecialCard = Agent.cards.findSpecial(Deck.top.color,Deck.top.SpecialType)
        if SpecialCard != null:
                play(SpecialCard)
                continue

        # choose either a number or a color
        CandidateColor = ChooseColor()
        CandidateNumber = ChooseNumber()

        if CandidateColor == null and CandidateNumber == null:
                draw()
                continue

        TotalEachColor = 25
        TotalEachNumber = CandidateNumber != 0 ? 8 : 4

        if PlayedColorsOrderedMap[CandidateColor] / TotalEachColor >= PlayedNumbersOrderedMap[CandidateNumber] / TotalEachNumber :
                play(CandidateColor)
                continue
        else:
                play(CandidateNumber)
                continue
```

Figure 7 - **Game loop**