Homework 10: Hanabi

Introduction to AI

This is a substantial homework, but it is one that students enjoy thinking about.

You have more time than usual to work on this, so make sure you start early!

You may optionally work with one partner on this assignment. If you do so, your code files should clearly indicate who worked on the project. Only one of you needs to upload files to D2L.

# Overview

For this assignment, you will build a **cooperative** AI to play the card game Hanabi. The game will be entirely automated, so one copy of your AI will play with another.

I will provide code to enforce the basic rules of Hanabi, including limited information about your own hand. Leave all the .java files other than Player and CardKnowledge alone. You can edit those two files and add additional Java files where desired. All your code must be in Java.

You will be evaluated in large part on the performance of your AI. I will simulate several games, tracking your average score across those runs. For full credit, you should average at least 15 points. For each additional full point that you score, you will receive one point of extra credit on your next exam. (For the record, the current average to beat is 20.1.)

I have extra copies of Hanabi available in my office. You are welcome to check out a copy to practice strategy.

# Hanabi Rules Summary

Hanabi is a cooperative card game played with a deck of fifty cards. The deck is divided into five suits by color: Red, Green, Blue, Yellow, and White. In each suit, there are cards of various numerical rank: three 1s, two 2s, two 3s, two 4s, and one 5.

The objective of Hanabi is to build five piles of cards, one for each of the five suits, with each pile counting from 1 up to 5 in order; these piles of cards are called the *tableau*. Barring a complete tableau, the players want to complete as much as possible. Each player is dealt a hand of five cards, which he plays onto the shared number piles.

The central twist of the game is that players hold their cards facing away from themselves; thus, no player ever knows for sure what cards he has available. Players *can* see their partners’ hands, but they are forbidden from commenting on them except as detailed below. On his turn, a player **must take one** of the following three actions:

* He can play a card from his hand to the tableau. To do so, the player touches one of his cards in hand, without looking at it, announces “Playing,” and lays it down face-up. If the card can be legally added to the tableau, it is; otherwise, the players expend one of their three *fuses*. If they run out of fuses, the game ends in defeat. (Fuses are shared between players, so they have three fuses between the two of them.)
  + *Example: The tableau currently shows the following cards on top of their respective stacks: Green 3, Blue 2, Red 5, White 3. (The Yellow 1 has not yet been played, and so there is no yellow pile.) A play of Green 4 or Blue 3 is legal and would be placed on top of the appropriate pile. A play of Green 3 or Blue 4 is illegal and would expend a fuse.*
  + If a player legally plays a 5, he also recovers a hint.
* He can identify all cards in a partner’s hand of a particular number or color. He could, for instance, indicate all 3s or all Blue cards in the hand. A player giving hints can only give one hint, and he must indicate all cards of the chosen number or color. Unfortunately, the players must spend a hint token in order to offer a hint, and they begin with only eight hint tokens. (Like with fuses, the eight hint tokens are shared between the players.)
  + A player who discards a card for his action (below) may recover a hint token, again to a maximum of 8.
  + A player who legally plays a 5 to the tableau may recover a hint token, again to a maximum of 8.
* He can discard a card from his hand, again without looking at it until after declaring “Discard.” Doing so allows the players to recover one hint token (again, to a maximum of 8). Discards are common knowledge, and players may sort through the discard pile at any time.

At the end of any turn in which a player’s hand size dropped below 5, he must draw back up to 5 cards if able. The game ends when one of three conditions are met:

* If the tableau is ever completed (all five suits at 5), the players win with a score of 25 (the maximum possible).
* If the players spend all their fuse tokens, they immediately lose, with a score of 0.
* If the draw deck runs out, all players (including the one who drew the last card of the deck) receive one more turn. Then the game ends, and players score one point for each card showing in the tableau. For example, if the tableau was Green 3, Red 2, Blue 5, White 4, Yellow 4, the players would score 18 points.

# Code Base

**Driver** contains main. It creates and runs **Hanabi** objects, which are instances of a Hanabi game. The Hanabi.parseAndHandleResponse method is the bulk of this class, taking a string command, validating it, and executing the given command. Valid strings are described later.

Hanabi objects separate the game state into member variables hidden to the agent—cards in the hands and deck—and a **Board** object, which contains game state that the agents can see: the number of fuses, number of cards in the deck, number of hints, the tableau, and the discards.

**Hand, Colors, and Card** are basic data classes to represent their respective objects.

Aside from simple modifications to Driver to test different cases, do not edit any of the above classes or turn those in with your project. I will use my version of those classes to test your project.

That leaves the Player class and the CardKnowledge class. **The Player class** is the starting point for your code...

You **must implement each of the methods provided in the starter code for Player.java**. These are the only communication your Player should have with the game. Specifically...

* The game will call the ask method to get your player’s next move. The comments for the ask method specify valid return values to indicate the move you want to make. This is the only information that should be sent out from your Player. Determine the next move using information that your player has gathered so far, its knowledge base about the world.
* The game will call one of several tellXYZ methods to tell your agent the results from moves (sometimes your move and sometimes your partner’s move). Your player should use that information to update its knowledge about the game, so it can make a good move the next time it is asked.

Determining the following is the interesting and challenging part of the project!

1. What knowledge to maintain about the game and what data structures to use to store that information. The provided **CardKnowledge class** might be useful in your implementation, but you are not required to use it. You can ignore it, use it as is, or modify it as you see fit. If you use it or modify it, include it with your submitted code.
2. How to update that knowledge, both when a “tell” method is called to tell you the result of a move, and when you make a move that has known result (so no “tell” method will be called)
3. How to use that knowledge to determine the next move to make (i.e., when “ask” is called)

To do these, you will likely want to create some more classes to add to your project. However, **using static variables or other methods for communicating from your agent to the game or the other agent is strictly forbidden and will result in a failing grade. The only information to/from your agent goes through the provided tell/ask methods.**

# Tips and Hints

Play a couple games of Hanabi to figure out strategies that you think might work or eliminate strategies that are unlikely to work.

You don’t need a very sophisticated algorithm to earn full credit. **Start with a simple strategy**, evaluate it, then improve if necessary.

Each card in the game must be in one of the following locations: in your hand, in your partner’s hand, in the deck, in the discard pile, or in the tableau. You can use this fact to reason about where different cards are.

Once you have a basic strategy implemented, **if** you need to improve your score, consider the following...

Since your agent is playing the game with another instance of the same agent, you know when your partner will be making certain moves. That is, WHEN your partner gives a hint can convey information, not just WHAT they hint. For example, if my strategy is to always hint a color when my partner has a 3 of that color, and that is the only time I hint a color, then my partner can deduce that if I hinted a color, they must have a 3 of that color. (This isn’t a good strategy, just an example.)  
As a more complex example, suppose that I hint yellow cards if my partner has any (and they don’t already know about them); otherwise, I hint 2’s if they have any that they don’t already know about; otherwise, I discard. Then if my partner sees me discard (and there are hints available on the Board), he or she can determine that they have already been told about any yellows or 2’s that they have.

If you do something like this, I encourage you to draw a flow chart or finite state machine to illustrate when your agent will make the different moves. You can use that to figure out “If my partner did this move, then certain things must be true and other things must not be true...”

# Handin

Zip your Player.java file, CardKnowledge.java file (if you used it) and any other Java files you created, and submit those on D2L. I’ll insert them into my project and evaluate the performance of your agent.