API Specification – Rummy Player

Todd Dole, CSCI-4332 Artificial Intelligence – Spring 2025

Revision 1.0

Preliminaries:

For assignments in this course, each student will need to make a github account and use git to submit your code to a public github repository. You will then submit a link to your github repository with the completed code through Canvas.

I highly recommend using an IDE that has git integrated, such as Jetbrains Pycharm, vscode or Visual Studio. (Pycharm is my personal preference.)

We will test and run code for this project on the CSCI department's AI workstation (ai.hsutx.edu). You will need a SSH client such as Mobaxterm (https://mobaxterm.mobatek.net/) to connect to the server. You will also need to be on the HSU network to connect.

Architecture:

For this assignment, we will use a microservices model running multiple processes in Python using FastAPI and Uvicorn. All of the interprocess communication has already been written for you, and the FastAPI approach allows us to exchange data without needing to use json directly—the library will automatically pack and unpack data for us. Your job will be to implement the coding logic to respond appropriately to play the game successfully. Each client will only be engaged in a single game at a time—one game will fully complete before the next one begins. So there should be no need to implement thread safe code in your knowledge tracking or game play approach. For more information about FastAPI and Uvicorn, see https://fastapi.tiangolo.com/

Overview and Setup:

When a player launches, it will notify the server that it is available for games by making an api call (see pre-existing code.) In debug mode, instead of notifying the server it is ready for games, it will notify the server it is ready for testing. At that point, the player process will expose its own API on a designated port (each student will be given their own port range to use.) The server will then process testing and/or games by making api calls to each connected client.

Data is passed in json format, using Python string data. In all data, cards are designated by a single character representing the card value (23456789TJQKA) and a single character representing the card suit (CDHS) so that the 2 of clubs would be represented “2C”, the ten of hearts would be represented “TH” and the Ace of Spades would be represented “AS”. Individual cards will be separated by spaces, so for instance a set of melded 2’s could be represented as “2S 2C 2H” and a run of clubs could be represented “TC JC QC KC”.

Meld sets will be numbered by the server in the order played, resetting for each hand and numbered starting at 0. For instance, if a player plays the first meld of the game as “2S 2C 2H” the server could designate it to other players as “meld(0): 2S 2C 2H”. If a later player wants to then lay off a card on that meld, they would use “meld(0)” to designate where the card will be played.

API Endpoints:

All Player API Endpoints use the POST method and will be passed a JSON structure with game state information.

/start-2p-game/

This endpoint is called when a game begins, either in test mode or in regular game play mode. The data passed includes a game id, the name of the opponent, and a string containing the hand (a set of 10 cards separated by spaces.) For example:

```
{ “game_id”: “1”,
  “opponent”: “DoleTest”,
 “hand”: “2C 7C 4D QD AD 7H 9H JS QS AS” }
```

API must return {“status”: “OK”} once finished processing data.

/start-2p-hand/

This endpoint is called when a new hand begins, continuing the current game. The data passed includes only the new hand.

API must return {“status”: “OK”} once finished processing data.

/update-2p-game/

This endpoint is called to update the player on other players' moves and game status. Data includes the game_id and an event string containing one or more lines (separated by \n), each line representing one player's action or game update.

Your code will need to process the event string and record/store what information you would like to save (at the very least, anything which updates your own hand.) (Using regex would be useful here.)

Here are some representative examples of event string lines (using DoleTest for the opponent player name, and mvr2070 for the current client name). This is not an exhaustive list—I recommend paying close attention to what the server passes when testing.

"DoleTest draws"

"mvr2070 draws 2C" – when you draw, either from discard or deck

"DoleTest takes 2C" – when the player takes the 2 of Clubs from the top of discard pile.

"DoleTest plays meld(0): 2C 2S 2D"

"DoleTest laysoff meld(0): 2H" -- when a player adds to an existing meld pile

"DoleTest discards 7H"

"Discard Shuffled" – When the discard is shuffled and returned to stock pile. (The most recent card added to discard remains on the discard pile.)

"Hand Ends: DoleTest 170 mvr2070 195" (Hand is over, DoleTest has 170 points and mvr2070 has 195 points)

"Game Ends: DoleTest 390 * mvr2070 510" (Game is over, DoleTest has 390 points and mvr2070 has 510 points. * occurs after the winner's score, to designate winner in case of tie.

API must return {"status": "OK"} once finished processing data.

/draw/

This is called when it is your turn to draw. Data passed includes update_info identical in structure to what is passed to /update-2p-game/ (see above including event string examples).

API must return either {“play”: “draw discard”} or {“play”: “draw stock”} indicating whether you wish to draw a card from the stock pile or discard pile.

/lay-down/

This is called when it is your turn to lay down one or more cards. Data passed includes update_info identical in structure to what is passed to /update-2p-game/ (see above including event string examples).

API must return {“play”: “...”} with the “...” containing a list of cards to be played. Examples:

“meld 2C 3C 4C” plays the 2 of clubs, 3 of clubs, and 4 of clubs as meld

“layoff meld(4) 5C” adds the 5 of clubs to meld pile 4.

“discard 7H” discards the 7H

Multiple sets/cards should be combined into one string, for example:

{ “play”: “meld 2C 3C 4C meld 9H 9S 9C layoff meld(1) AS discard JH”}

/shutdown

This uses the GET method, and is only called in debug mode when testing is complete. It will cause your API to exit. You can add additional code to save data here before the program exits if you wish.

Starter Code:

Available at https://github.com/todddole/RummyPlayer

Note that an improved version of this will be released by Wednesday with a very basic player implemented. The game server will also be up and running by Wednesday, on ai.hsutx.edu.

Game / Contest Rules:

We will be playing standard Rummy with no wild cards/jokers. 52 cards in the deck, 10 cards in each player's hand.

Discarding is optional when a player goes out. All other turns must end with a player discarding, even when melding or laying off.

Players can lay off cards (add to another player's meld) before melding themselves.

Technical:

Each time your API is called, your program has at most 2 seconds to respond. If you exceed this time limit, the game will be forfeited.

If a client returns invalid output, or at any point attempts to make an illegal play, the game will be forfeited. A few examples: Discarding a card that's not in your hand. Playing invalid meld. Adding cards to a non-existent meld stack.