# DETAILED TEXT DESCRIPTIONS OF HOW TO HANDLE THE SCENARIOS

1. Determining how to store the board
   1. The board will be stored when the Class “HasamiShogiGame” is called.
2. Initializing the board
   1. The board is initialized in the init method of the class “HasamiShogiGame”.
3. Determining how to track which player's turn it is to play right now
   1. Since Black goes first (1), an init variable named \_move\_tracker will be initialized to 1. The \_move\_tracker then increments by 1 based on successful moves by each player. Black will always move when \_move\_tracker is “odd”. Red will always move when \_move\_tracker is “even”.
4. Determining how to validate piece movement
   1. Piece movements will depend on several things:
      1. If the current active player is starting the move {get\_active\_player()}
      2. If the starting location for the move belongs to the current active player
      3. If the move is horizontal or vertical
         1. A move is horizontal if starting location’s row IS the same as the ending location’s row AND starting location’s column IS NOT the same as the ending location’s column.
         2. A move is vertical if starting location’s row IS NOT the same as the ending location’s row AND starting location’s column IS the same as the ending location’s column.
         3. Anything else is an invalid move
      4. If the move is within the board’s dimensions 9x9 (a1:i9)
      5. If the movement is not obstructed by any other pieces along the way using {get\_square\_occupant()}
5. Determining when pieces have been captured
   1. Opposing pieces can only be captured by the active player’s move (active player cannot lose their own piece on their own turn).
   2. If there’s an opposing piece adjacent to the move’s end location, check to determine if the active player has another piece on the opposing piece’s side (sandwich).
      1. If so, capture the piece and increase the num\_of\_captured\_piece(opposing\_color) by 1
   3. If the move was to one of 8 “double team” corner positions (a2, b1, a8, b9, h1, i2, i8, h9). A piece of the opponent would need to be in the respective corner (a1, a9, i0, i9). If there’s a piece there, then check if active player has the 2nd “double team” position filled.
6. Determining when the game has ended
   1. While loop will be used to keep the movement going for the game.
   2. get\_game\_state() will be used to keep the while loop on-going.
   3. At the end of each successful move, get\_num\_captured\_pieces(“BLACK”) and get\_num\_captured\_pieces(“RED”) will be called. Those are used to update the game\_state if either call returns a value greater than 7.

# Steven Tran

# 2021-12-02

# CS-162 Final Project: Hasami Shogi (Variant 1)

# Description: In the Japanese Board Game of Hasami Shogi, players take turns moving their piece (either red or black). Pieces move like rooks in the game of chess, horizontally or vertically. The goal is to at capture least 8 of the 9 opponent’s pieces. To capture a piece, the opposing player's piece(s) must be "sandwiched" between two of the active player's pieces. Or be trapped in the corner by two of the active player’s pieces.

**class GamePiece():**

"""Initial game pieces (either RED or BLACK). Still TBD if want to use."""

**def \_\_init\_\_(self):**

pass

**class HasamiShogiGame():**

"""Initializes a game of Hasami Shogi (Japanese Rook-Like Capture Game).

The board will be stored in this init of \_game\_board where it is first initalized with 'RED' and 'BLACK' pieces at the applicable starting positions."""

**def \_\_init\_\_(self):**

"""Initializes the \_game\_board.

Moves will be tracked using the \_move\_tracker which will increment by one based on successful moves. This may also be used later at the end to determine how many moves it took for the game to complete.

\_game\_state: will be used with a while loop to determine the current state of the game. After each move, determines if the game is still in progress. This will be achieved by calling get\_num\_captured\_pieces function to determine if either RED or BLACK has captured at least 8 pieces. If so, the \_game\_state flag will result with the winner.

Otherwise it will remain in the 'UNFINISHED state. """

self.\_top\_label = " 1 2 3 4 5 6 7 8 9"

self.\_side\_label = ['a','b','c','d','e','f','g','h','i']

self.\_game\_board = [["R", "R", "R", "R", "R", "R", "R", "R", "R"],

["\_", "\_", "\_", "\_", "\_", "\_", "\_", "\_", "\_"],

["\_", "\_", "\_", "\_", "\_", "\_", "\_", "\_", "\_"],

["\_", "\_", "\_", "\_", "\_", "\_", "\_", "\_", "\_"],

["\_", "\_", "\_", "\_", "\_", "\_", "\_", "\_", "\_"],

["\_", "\_", "\_", "\_", "\_", "\_", "\_", "\_", "\_"],

["\_", "\_", "\_", "\_", "\_", "\_", "\_", "\_", "\_"],

["\_", "\_", "\_", "\_", "\_", "\_", "\_", "\_", "\_"],

["B", "B", "B", "B", "B", "B", "B", "B", "B"]]

self.\_move\_tracker = 1 # Tracks the number of moves performed in the game

self.\_game\_state = "UNFINISHED" # Will be used for while loop to determine if the game is still on-going.

**def display\_game(self):**

"""Displays the current state of the game. """

pass

**def set\_red(self, space):**

"""Sets a red piece in a specified location."""

pass

**def set\_black(self, space):**

"""Sets a black piece in a specified location."""

pass

**def set\_empty(self, space):**

"""Sets a space to empty in a specified location (typically starting location of a move)."""

pass

**def get\_game\_state(self):**

"""Function determines the current progress of the game. If there's a current winner or if its still "UNFINISHED"".

Returns:

UNFINISHED (str): when red/black captures < 8

RED\_WON (str): when black captures >= 8

BLACK\_WON (str): when red captures >= 8

"""

pass

**def get\_active\_player(self):**

"""Determines the active player of the game init based on

number of turns partaken.

Returns:

BLACK (str): if move\_tracker is odd

RED (str): if move\_tracker is even

"""

if self.\_move\_tracker % 2 == 1:

return "BLACK"

else:

return "RED"

**def get\_num\_captured\_pieces(self, color = None):**

"""Returns the number of captured pieces for the specified color. Used to determine if there's a winner after each move."""

pass

**def move\_to\_index(self, move):**

"""Converts the move to an index for the \_game\_board's list parameters allowing for rows of a-i and columns 1-9.

Returns:

HORIZONTAL (str): move to make is horizontal

VERTICAL (str): move to make is vertical

FALSE: Illegal move request (i.e. diagonal, non-existent space)

"""

pass

**def get\_square\_occupant(self, square = ""):**

"""Determines if the square is occupied or not.

Returns:

"RED": if square is occupied by a "R" piece

"BLACK": if square is occupied by a "B" piece

"NONE"": square is empty; occupied by a "\_"

"""

pass

**def move\_type(self, start\_loc, end\_loc):**

"""After checking the start and end locations are valid, this function

determines what kind of movement direction is being requested.

Args:

start\_loc (str): a location with an active piece

end\_loc (str): a valid location to move an active piece

Returns:

HORIZONTAL: move to make is horizontal

VERTICAL: move to make is vertical

None: Illegal move request (i.e. diagonal, non-existent space)

"""

pass

**def horizontal\_move(self, start\_loc, end\_loc):**

"""Checks to see if there are any pieces between the start\_loc to the end\_loc. If the start location's column is less than the end location's column, then the movement is right (i.e. d1 to d7). If the start location's column is greater than the end location's row, then the movement is left (i.e. d7 to d1).

Args:

start\_loc (str): a location with an active piece

end\_loc (str): a valid location to move an active piece

Returns:

True: move is valid; sets the move.

False: move is not valid (something obstructs)

"""

pass

**def vertical\_move(self, start\_loc, end\_loc):**

"""Checks to see if there are any pieces between the start\_loc to the end\_loc. If the start location's row is less than the end location's row, then the movement is up (i.e. i1 to h1). If the start location's row is greater than the end location's row, then the movement is down (i.e. a1 to f1).

Args:

start\_loc (str): a location with an active piece

end\_loc (str): a valid location to move an active piece

Returns:

True: move is valid; sets the move.

False: move is not valid (something obstructs)

"""

pass

**def corner\_capture(self, start\_loc, end\_loc):**

"""If the current move is nearby an opponent's corner piece, check to see if another active player's piece resides on the nearby corner.

Note: Corner captures can only occur if the active player moves to one 8 tiles (a2, b1, a8, b9, h1, i2, i8, h9). A piece of the opposite would need to be in the respective corner (a1, a9, i0, i9)

Args:

start\_loc (str): a location with an active piece

end\_loc (str): a valid location to move an active piece

Returns:

True: Captures the corner piece. Updates the active player, and move\_tracker

False: if corner capture condition is not met

"""

pass

**def horizontal\_capture(self, start\_loc, end\_loc):**

"""After a valid move check, determines if the current move is

horizontal to an opponent's piece. If so check recursively on the opposing side if either NONE, RED, or BLACK piece is on the side. If None, capture will not occur. If color matches active player, capture the piece and score. If color matches opposing player, check again recursively.

Args:

start\_loc (str): a location with an active piece

end\_loc (str): a valid location to move an active piece

Returns:

True: Captures the opponent's piece(s). Updates number of captured pieces, and move\_tracker

False: if horizontal capture condition is not met

"""

pass

**def vertical\_capture(self, start\_loc, end\_loc):**

""""""After a valid move check, determines if the current move is vertical to an opponent's piece. If so check recursively on the opposing side if either NONE, RED, or BLACK piece is on the side. If None, capture will not occur. If color matches active player, capture the piece and score. If color matches opposing player, check again recursively.

Args:

start\_loc (str): a location with an active piece

end\_loc (str): a valid location to move an active piece

Returns:

True: Captures the opponent's piece(s). Updates number of captured pieces, and move\_tracker

False: if vertical capture condition is not met

"""

pass

**def next\_turn(self):**

"""Increases the move tracker after a successful turn."""

self.\_move\_tracker += 1

**def make\_move(self, start\_loc, end\_loc):**

"""Moves a piece from the start\_loc to the end\_loc as long as it is a valid move.

Checks for:

\* Valid Turn (if its RED or BLACK's turn)

\* Valid Move (if the start\_loc can actually reach end\_loc without any obstructions)

\* Type of Move (horizontal versus vertical)

\* Capturing conditions:

- Horizontal Captures (left and right)

- Vertical Captures (above and below)

- Corner Captures

Args:

start\_loc (str): a location with an active piece

end\_loc (str): a valid location to move an active piece

Returns:

True: updated location with the piece. Also updates start\_loc to an empty location again. Updates the active player, and move\_tracker

False: if the current move is blocked by any pieces

"""

pass