```
import tkinter as tk

GRID_CELL_SIZE = 50
GAME_PIECE_PADDING = 5
MOVE_DIRS = [(1, 0), (-1, 0), (0, 1), (0, -1), (-1, 1), (1, 1),
(-1, -1), (1, -1)]

class GameManager:
    def __init__(self, board_size):
        self.board = {}
        self.board_size = board_size
        self.selected_piece = None
        for row in range(board_size):
            for col in range(board_size):
                self.board[(row, col)] = 'empty'

        half_board_size = board_size // 2
        for row in range(half_board_size):
            for col in range(half_board_size - row):
                self.board[(row, col)] = 'red'

        for row in range(board_size - 1, half_board_size - 1, -
1):
            for col in range(board_size - 1, (half_board_size -
1) + (board_size - 1 - row), - 1):
                self.board[(row, col)] ='green'
        self.board_display = GameBoard(self.board, self,
board_size)

    def start_move(self, cell):
        self.selected_piece = cell
        possible_moves = MoveGenerator.get_moves(cell,
self.board, self.board_size)
        self.board_display.show_moves(cell, possible_moves)

    def execute_move(self, dest_cell):
        self.board[dest_cell] = self.board[self.selected_piece]
        self.board[self.selected_piece] = "empty"
        self.exit_move()
        self.board_display.update(self.board)
    def exit_move(self):
        self.board_display.exit_move_state()
        self.selected_piece = None

class GameCell:
    def __init__(self, row, col, game_board, manager):
        self.pos = (row, col)
        self.board = game_board
        self.canvas = tk.Canvas(tk_root, width = GRID_CELL_SIZE,
                                        height = GRID_CELL_SIZE,
bg='burlywood1',

highlightbackground='black')
        self.state = "empty"
        self.canvas.grid(row=row+1, column=col+1)
        self.highlighted = False
        self.manager = manager

    def clear(self):
        self.canvas.delete("all")
        self.canvas.unbind('<Button>')
        self.state = "empty"

    def set_red_state(self):
        self.canvas.create_oval(GAME_PIECE_PADDING,
GAME_PIECE_PADDING,
                        GRID_CELL_SIZE - GAME_PIECE_PADDING,
GRID_CELL_SIZE - GAME_PIECE_PADDING,
                                    fill = "white")
        self.state = "red"
        self.canvas.bind('<Button>', lambda event:
manager.start_move(self.pos))

    def set_green_state(self):
        self.canvas.create_oval(GAME_PIECE_PADDING,
GAME_PIECE_PADDING,
                        GRID_CELL_SIZE - GAME_PIECE_PADDING,
GRID_CELL_SIZE - GAME_PIECE_PADDING,
                                    fill = "black")
        self.state = "green"
        self.canvas.bind('<Button>', lambda event:
manager.start_move(self.pos))

    def highlight(self):
        if self.state == "empty":
            self.canvas.create_rectangle(0, 0, GRID_CELL_SIZE,
GRID_CELL_SIZE,
                                        outline="black",
fill="burlywood4")
            self.canvas.bind('<Button>', lambda event:
manager.execute_move(self.pos))
        elif self.state == "red":
            self.canvas.create_oval(GAME_PIECE_PADDING,
GAME_PIECE_PADDING,
                                    GRID_CELL_SIZE -
GAME_PIECE_PADDING, GRID_CELL_SIZE - GAME_PIECE_PADDING,
                                        fill = "white",
outline="red", width=3)
            self.canvas.bind('<Button>', lambda event:
manager.exit_move())
        else:
            self.canvas.create_oval(GAME_PIECE_PADDING,
GAME_PIECE_PADDING,
                                    GRID_CELL_SIZE -
GAME_PIECE_PADDING, GRID_CELL_SIZE - GAME_PIECE_PADDING,
                                        fill = "black",
outline="green", width=3)
            self.canvas.bind('<Button>', lambda event:
manager.exit_move())
        self.highlighted = True


    def unbind_click(self):
        self.canvas.unbind('<Button>')

class GameBoard:
    def __init__(self, board, manager, board_size):
        self.display_board = {}
        self.manager = manager
        col_labels = [chr(ord('a') + num) for num in
range(board_size)]
        for index, label in enumerate(col_labels):
            tk.Label(tk_root, text=label).grid(row=0,
column=index + 1)
        for index in range(board_size):
            tk.Label(tk_root, text=index + 1).grid(row=index +
1, column=0)
        for cell, state in board.items():
            row, col = cell
            self.display_board[cell] = GameCell(row, col, self,
self.manager)
            if state == 'red':
                self.display_board[cell].set_red_state()
            elif state == 'green':
                self.display_board[cell].set_green_state()


    def exit_move_state(self):
        for cell in self.display_board.values():
            state = cell.state
            if cell.highlighted:
                cell.highlighted = False
                cell.clear()
            if state == "red":
                cell.set_red_state()
            elif state == "green":
                cell.set_green_state()

    def show_moves(self, piece, moves):
        for cell, canvas in self.display_board.items():
            if cell in moves or cell == piece:
                canvas.highlight()
            else:
                canvas.unbind_click()

    def update(self, new_board):
        for cell, canvas in self.display_board.items():
            if new_board[cell] == "empty":
                canvas.clear()
            elif new_board[cell] == "red":
                canvas.set_red_state()
            else:
                canvas.set_green_state()

class MoveGenerator:
    def get_moves(cell, game_board, board_size):
        move_stack = [[cell, [cell]]]
        valid_moves = []

        while move_stack:
            curr_cell, path = move_stack.pop()
            row, col = curr_cell

            for row_change, col_change in MOVE_DIRS:
                move = (row + row_change, col + col_change)
                if MoveGenerator.valid_cell(move[0], move[1],
board_size) and move not in path:
                    if game_board[move] == "red" or
game_board[move] == "green":
                        jump_move = (move[0] + row_change,
move[1] + col_change)
                        if
MoveGenerator.valid_cell(jump_move[0], jump_move[1], board_size)
and game_board[jump_move] == "empty" and jump_move not in path:
                            valid_moves.append(jump_move)
                            if
MoveGenerator.check_for_surrounding_piece(jump_move, game_board,
board_size):

move_stack.append([(jump_move[0], jump_move[1]), path +
[jump_move]])
                    elif curr_cell == cell:
                        valid_moves.append(move)

        return valid_moves

    def check_for_surrounding_piece(cell, board, board_size):
        row, col = cell
        for row_change, col_change in MOVE_DIRS:
            adj_cell = (row + row_change, col + col_change)
            if MoveGenerator.valid_cell(adj_cell[0], 
adj_cell[1], board_size) and \
            (board[(adj_cell)] == "red" or board[adj_cell] == 
"green"):
                return True
        return False

    def valid_cell(row, col, board_size):
        return row < board_size and row > -1 and col < 
board_size and col > -1


tk_root = tk.Tk()

tk_root.title("Halma")
manager = GameManager(8)
tk_root.mainloop()
```

Overview
For this deliverable, four classes were developed to enable the Halma functionality:

GameManager

- The main “controller” class, takes in a board size in its __init__ method, creates a dictionary to logically manage the board, then creates a GameBoard object for displaying the game with the initial conditions and piece placements.
- Methods:
    - start_move: triggers the start of the move, gets the possible moves for the given piece and displays them on the board
    - execute_move: “performs” the move by moving it on the board and reflects this in the display
    - exit_move: used to exit the move “state”, whether the move is cancelled or executed

GameCell

- Represents individual cells on the display board, used for tkinter displaying/action binding/etc. Takes in a row and col denoting the cell position, the display GameBoard, and the GameManager.
- Methods:
    - clear: removes all shapes, unbinds actions, and sets cell state to empty
    - set_red_state: denotes the cell as filled with a “red” player piece, draws, sets state, and binds actions accordingly
    - set_green_state: same as above, but for “green”.
    - highlight: depending on the cell’s state (i.e., if its empty or has a piece), highlights it for display during a move
    - unbind_click: unbinds actions from clicking on the cell

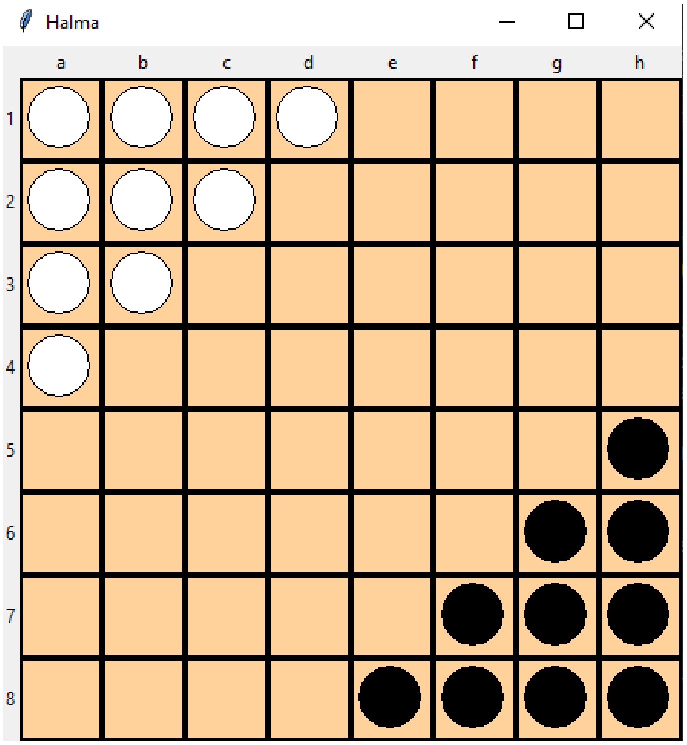
GameBoard

- The “display” board for the game, takes in the “logical” board from the GameManager and contains methods for updating and manipulating the board depending on the game state. Takes in the logical board and the GameManager, and initializes a board full of GameCells matched to each position in the board.
- Methods:
    - exit_move_state: used from exit_move and execute_move in the GameManager; visually reflects the exiting of the move state in the GameBoard
    - show_moves: used from start_move in GameManager, highlights the piece being moved and the possible spaces it can move to
    - update: used when moving pieces in execute_move, updates the GameBoard to reflect the logical board given from GameManager

MoveGenerator

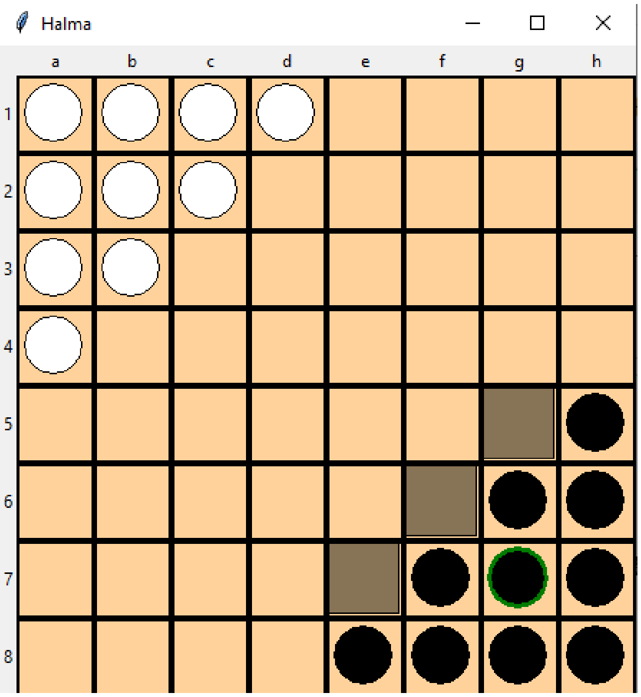
- A function class used for getting possible moves from a given piece's position
- Methods:
    - get_moves: uses a move stack to determine all of the next possible moves for the given cell and board state, also considering multiple jumps
    - check_for_surrounding_piece: used by get_moves, used after a jump to determine if a subsequent jump is possible.
    - valid_cell: utility function for determining if a cell is valid given the board size

Initial Board:

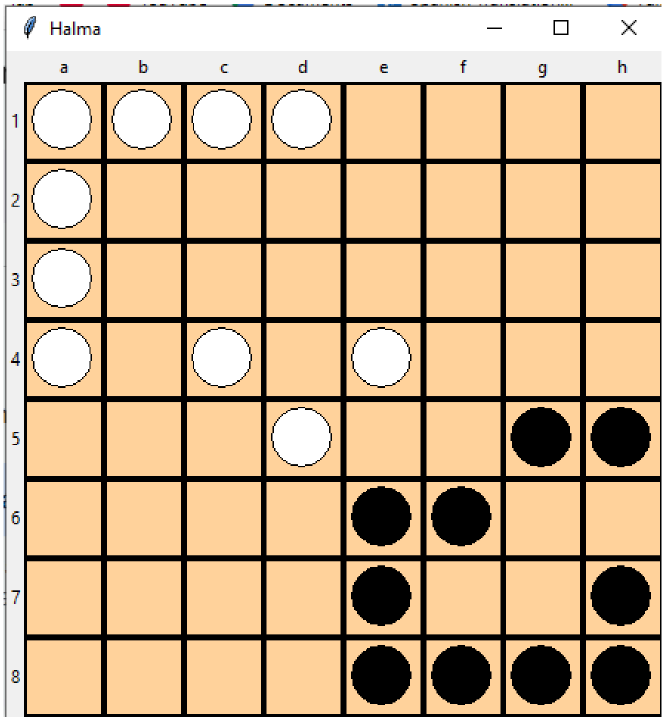


	a	b	c	d	e	f	g	h
1	○	○	○	○				
2	○	○	○					
3	○	○						
4	○							
5								●
6							●	●
7						●	●	●
8					●	●	●	●

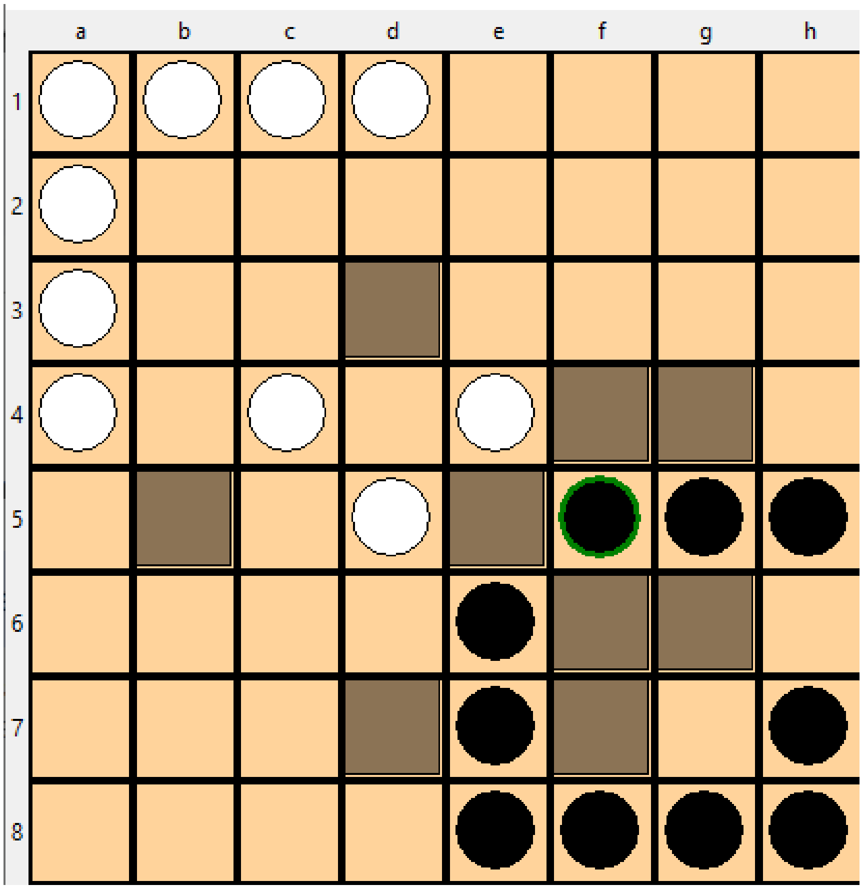
Board after selecting piece 7g



Board after moving some pieces from both players



Showing subsequent jumps (piece in 5f can do the jump 5f → 3d → 5b)



	a	b	c	d	e	f	g	h
1	○	○	○	○				
2	○							
3	○			▨				
4	○		○		○	▨	▨	
5		▨		○	▨	●	●	●
6					●	▨	▨	
7				▨	●	▨		●
8					●	●	●	●