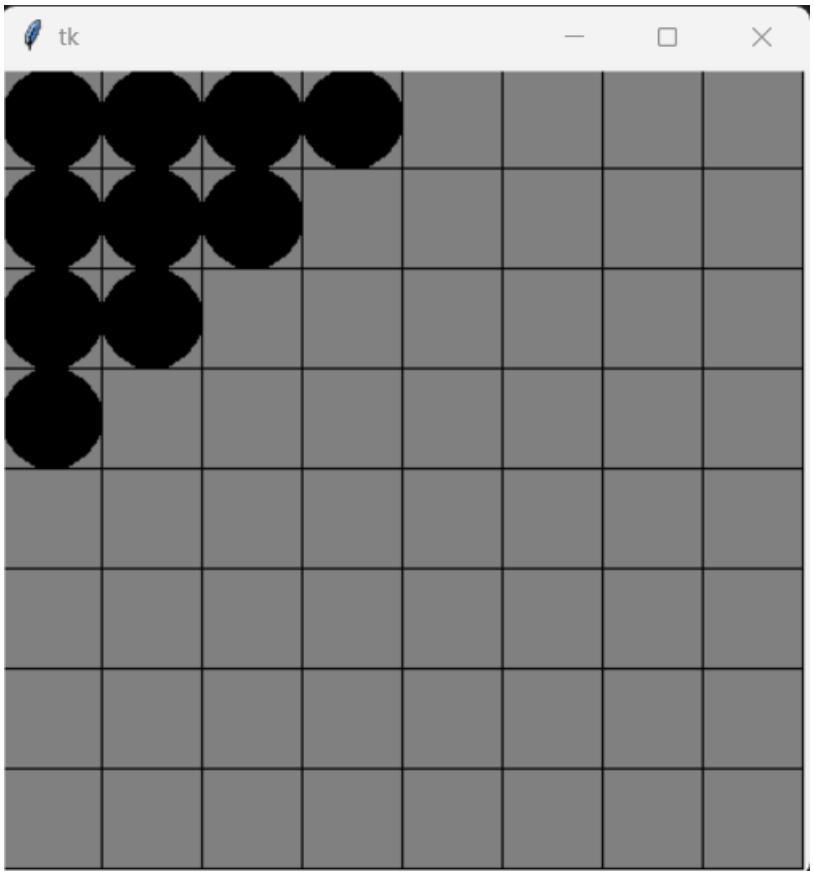
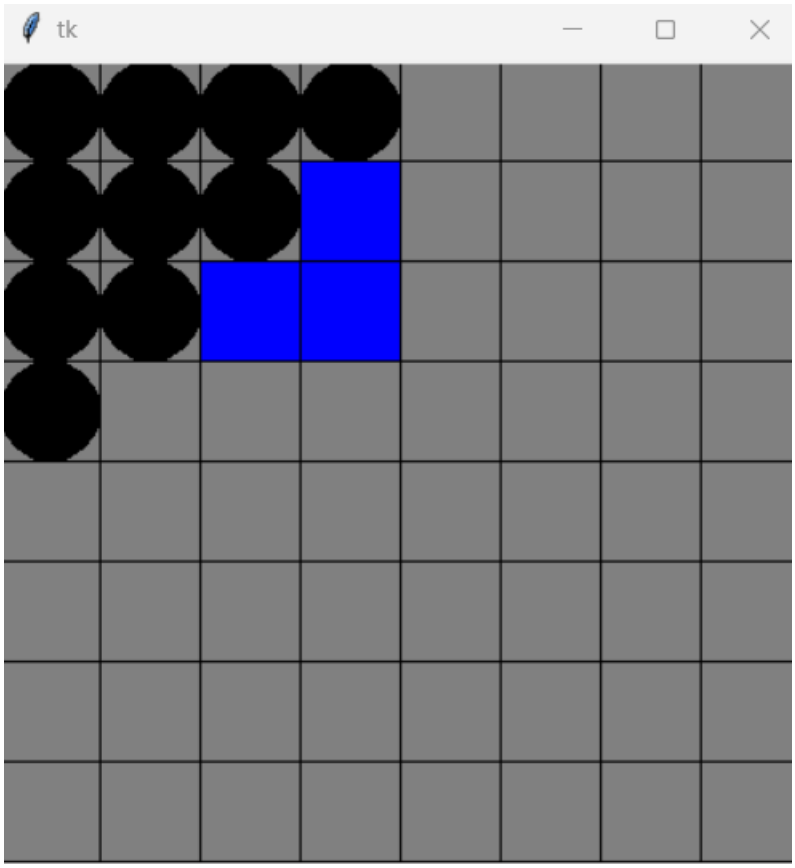
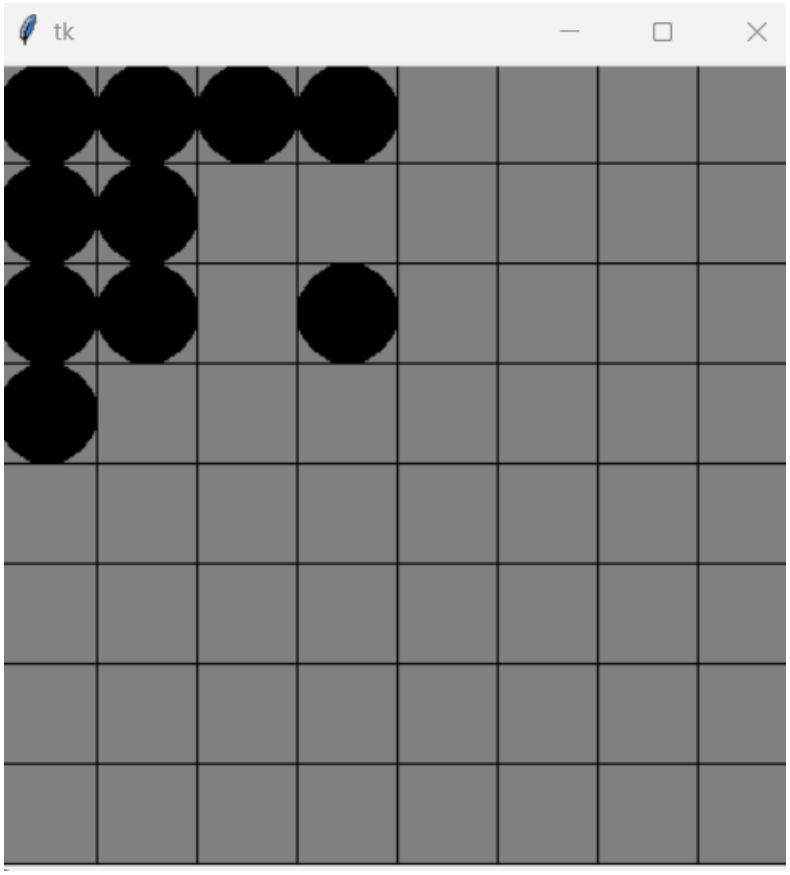
AI-HW9

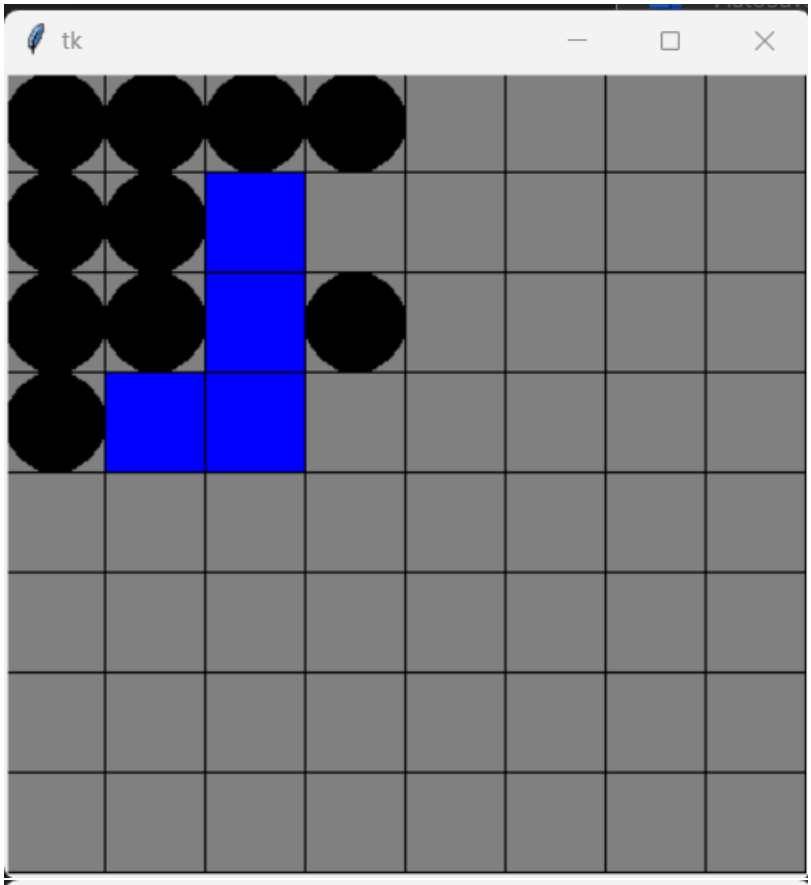
Been unable to complete HW due to illness.

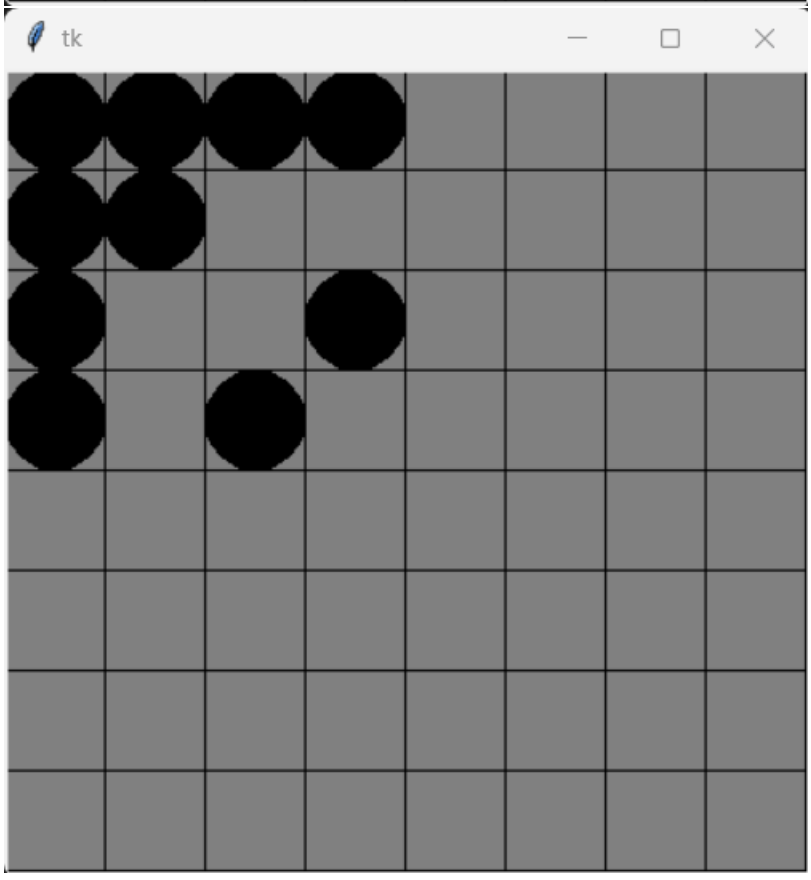
The general Idea is to have the board as a canvas and draw pieces objects onto that board. Later those pieces will interact with each other in order to generate the kind of possible moves of jumping over other pieces.




tk


tk






```
from tkinter import *
from tkinter import ttk
```

```
SQUARE_SIZE = 50
BOARD_SIZE = 8
CAMP_SIZE = 4
OFFSETS = [-1, 0, 1]
PIECES_DICT = {}

root = Tk()
selected_piece = None
previous_piece =None


class GameBoard(Canvas):
    def __init__(self, root):
        super().__init__(root, width=SQUARE_SIZE*BOARD_SIZE,
height=SQUARE_SIZE*BOARD_SIZE)
        self.grid()
        self.squares = []
        for row_num in range(BOARD_SIZE):
            row = []
            for col_num in range(BOARD_SIZE):
                color = "grey"
                square_id = self.create_rectangle(col_num * SQUARE_SIZE,
row_num * SQUARE_SIZE,
                                                  (col_num + 1) * SQUARE_SIZE,
(row_num + 1) * SQUARE_SIZE, fill=color)
                row.append(square_id)
                self.tag_bind(square_id, "<Button-1>", self.move)
            self.squares.append(row)

    def move(self, event):
        global selected_piece
        global previous_piece
        if selected_piece:
            col_num, row_num = event.x // SQUARE_SIZE, event.y // SQUARE_SIZE
            selected_piece.col_num, selected_piece.row_num = col_num, row_num
            selected_piece.redraw()
            selected_piece.reset_moves_shown()
            previous_piece = selected_piece
            selected_piece = None


class Pawn:
    def __init__(self, board, row_num, col_num, color):
        self.board = board
        self.row_num = row_num
        self.col_num = col_num
        self.color = color
        self.update_adjacent()
```

```
        self.oval_id = self.board.create_oval(self.col_num * SQUARE_SIZE,
self.row_num * SQUARE_SIZE,
                                              (self.col_num + 1) *
SQUARE_SIZE, (self.row_num + 1) * SQUARE_SIZE, fill=self.color)
        self.board.tag_bind(self.oval_id, "<Button-1>", self.select_pawn)

    def select_pawn(self, event):
        global selected_piece
        selected_piece = self
        self.update_adjacent()
        update_pieces_dict()
        self.show_possible_moves()

    def redraw(self):
        self.board.coords(self.oval_id, self.col_num * SQUARE_SIZE,
self.row_num * SQUARE_SIZE,
                          (self.col_num + 1) * SQUARE_SIZE, (self.row_num + 1)
* SQUARE_SIZE)

    def show_possible_moves(self):
        for adjacent_x, adjacent_y in self.adjacent:
            if (adjacent_x, adjacent_y) not in PIECES_DICT.keys() and 0 <=
adjacent_x < BOARD_SIZE and 0 <= adjacent_y < BOARD_SIZE:
                self.board.itemconfig(self.board.squares[adjacent_x][adjacent_
y], fill="blue")

    def reset_moves_shown(self):
        for row_num in range(BOARD_SIZE):
            for col_num in range(BOARD_SIZE):
                if (row_num, col_num) not in PIECES_DICT.keys():
                    self.board.itemconfig(self.board.squares[row_num][col_num]
, fill="grey")

    def update_adjacent(self):
        self.adjacent = [
            (self.row_num + row_offset, self.col_num + col_offset)
            for row_offset in OFFSETS
            for col_offset in OFFSETS
            if row_offset != 0 or col_offset != 0
        ]

    def calculate_possible_moves(self):
        pass
        #If there is piece adjacent
        #check if there is spot on other side
        #if so add to possible moves list or dict
        #recursivley call this function again from the new possible position
```

```
def update_pieces_dict():
    keys_to_remove = [
        key for key, piece in PIECES_DICT.items()
        if piece.col_num == selected_piece.col_num and piece.row_num ==
selected_piece.row_num
    ]
    for key in keys_to_remove:
        del PIECES_DICT[key]

    if previous_piece:
        PIECES_DICT[(previous_piece.row_num, previous_piece.col_num)] =
previous_piece




board = GameBoard(root)

for y_col in range(BOARD_SIZE):
    for x_col in range(BOARD_SIZE):
        if y_col + x_col < CAMP_SIZE:
            PIECES_DICT[(y_col, x_col)] = Pawn(board, y_col, x_col, "black")

root.mainloop()
```