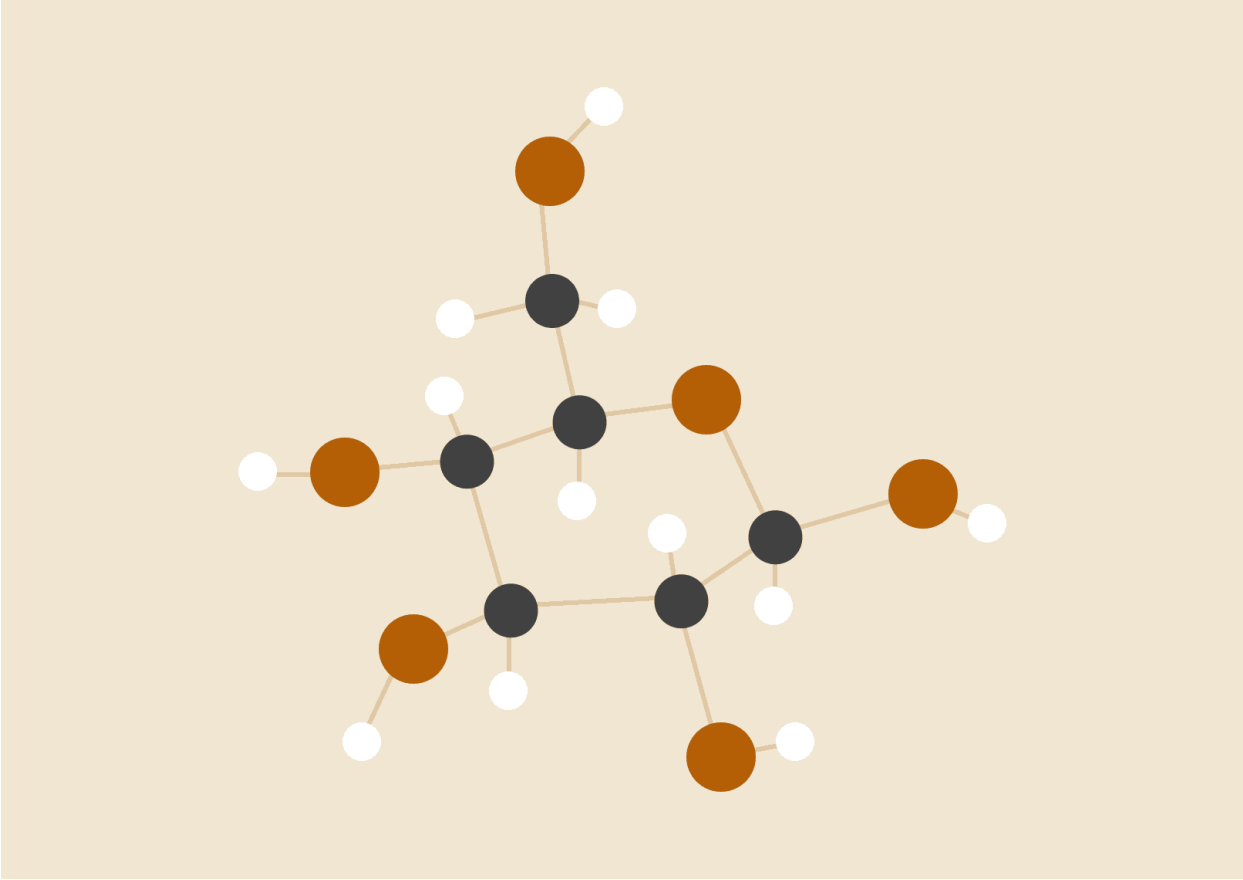
PYTHON PROJECT REPORT



Volety Sriram Panchamukhi

23EEB0A15

EEE-A

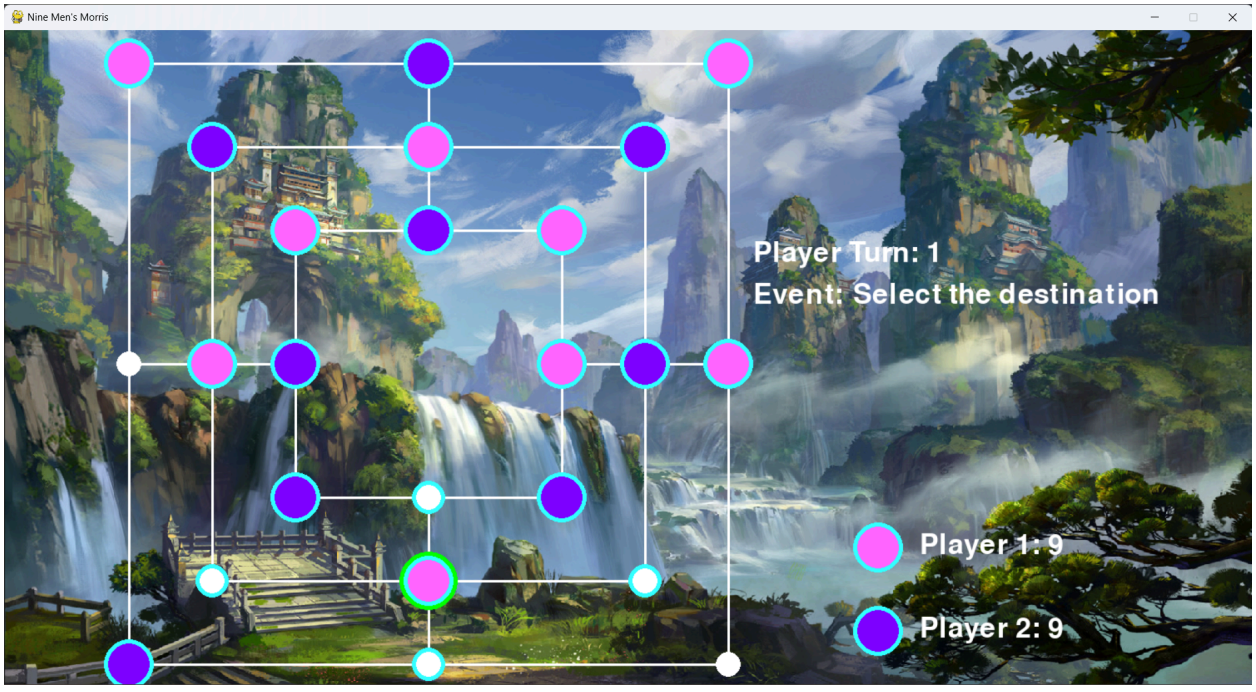
Submission Date: 6/9/24

CONTENTS

INTRODUCTION

The main aim of my project is to make a python program to play Nine Men’s Morris in a player vs player model.This Nine Men's Morris project in Python, built with Pygame, brings a classic strategy game to life. Featuring an intuitive GUI, players can place, move, and capture pieces while the program tracks occupancy. The project aims to deliver an engaging and visually captivating experience. The rules of the game along with the functions used to execute them will be explained along this report.

GAME MAP



GAME RULES

MAP DETAILS

The map for the game Nine Men’s Morris is a fairly simple one. It consists of 3 concentric squares of varying side lengths whose midpoints of sides are connected by two vertical and two horizontal lines connecting from inner to square to outer square on both sides.

Players can place their pieces along the vertices of squares or along the intersection points between the squares and lines.

The main aim is to get your 3 pieces in a straight line forming a mill.

PLAYING RULES

The game mainly consists of 2 parts - placing and moving. Each player gets nine pieces to place on the given map. After placing all of your pieces is done, players can move their pieces along the map to adjacent empty points taking turns.

During any phase, if a player forms a mill, he can remove an opponent's piece of his choice. Opponent’s pieces that are already in a mill are immune to this effect if there are other pieces that are not in mills. After a mill is formed, a player can break the mill by moving one of its pieces and forming a mill again, he gains the power to remove an opponent's piece once again.

The game ends if the number of pieces of a player becomes less than 2 or if there are no empty spaces for a player to move his pieces.

MILESTONES

COMPLETELY ACHIEVED

1. Placing of pieces and removal pieces is being executed perfectly according to the rules of the game.
2. Player Turn and Event to be done are shown on screen so players can clearly understand whose move has to be played and what kind of move should be done now.
3. During the moving phase, when a piece is selected to move, all of its possible movements are highlighted.
4. Mill made during a turn can remove the piece only once, to use it again, a new mill has to be formed.

PARTIALLY ACHIEVED:

1. Instructions for 'Event' in case of mouse-click at an invalid place is not being managed effectively (code logic is correct but instructions related to that are not being displayed on the game window).
2. After winning the game, I wanted to keep the game window open for 10 seconds to see the board and get it closed automatically but it is not occurring for some reason.

FUTURE IMPROVEMENTS

1. A player vs AI model can also be designed.
2. A mute toggle button can also be added.

3. Undo function can also be added.
4. Movement of pieces can be animated.

IMPLEMENTATION DETAILS

1. Structure of a Point

There are a total of 24 points on the board. A class is defined for the points on the board in which

i) [.x] and [.y] : A set of coordinates(x,y) is used to mention its position on screen

ii) [.occupied] : Its occupation status if a piece is present or not

iii)[.connections] : A list which contains the points which are connected to it in the map

iv) [.player] : If it is occupied by Player 1 or Player 2 or None

A few basic functions are also defined on these points such as

occupy : Used to place a piece of the player given as input at a given point

vacate : Used to remove a piece of the player given as input at the given point

```python
# Point class for storing details
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.occupied = False
        self.player = None
        self.connections = []

    def occupy(self, player):
        self.occupied = True
        self.player = player

    def vacate(self):
        self.occupied = False
        self.player = None
```

2. Defining Variables and Others

Variables are used for different functionalities in my project which are:

1. Colors for GUI
2. Tracking number of pieces for each player
3. Tracking game phase and event execution
4. Using sets for effective handling of mills of each player
5. Defining board dimension parameters for drawing the board using GUI
6. Defining list of points, possibility of connections between them and possibility of mill formation between the points to readily detect them

3. Functions in Code

1. my_board() :

This function is being used for drawing the map on the game window and it also returns the list of points which are used as an input in every other function.

```
def my_board():  2 usages
    v_margin = 40
    o_size = 720
    m_size = 520
    s_size = 320
    gap = 100

    # Define 24 points
    points = [
        Point(h_margin, v_margin), Point(h_margin + o_size // 2, v_margin), Point(h_margin + o_size, v_margin),
        Point(h_margin + gap, v_margin + gap), Point(h_margin + gap + m_size // 2, v_margin + gap),
        Point(h_margin + gap + m_size, v_margin + gap), Point(h_margin + gap * 2, v_margin + gap * 2),
        Point(h_margin + gap * 2 + s_size // 2, v_margin + gap * 2),
        Point(h_margin + gap * 2 + s_size, v_margin + gap * 2),
        Point(h_margin, v_margin + o_size // 2), Point(h_margin + gap, v_margin + o_size // 2),
        Point(h_margin + gap * 2, v_margin + o_size // 2), Point(h_margin + gap * 2 + s_size, v_margin + o_size // 2),
        Point(h_margin + gap * 3 + s_size, v_margin + o_size // 2),
        Point(h_margin + gap * 4 + s_size, v_margin + o_size // 2),
        Point(h_margin + gap * 2, v_margin + gap * 2 + s_size),
        Point(h_margin + gap * 2 + s_size // 2, v_margin + gap * 2 + s_size),
        Point(h_margin + gap * 2 + s_size, v_margin + gap * 2 + s_size), Point(h_margin + gap, v_margin + gap + m_size),
        Point(h_margin + gap + m_size // 2, v_margin + gap + m_size),
        Point(h_margin + gap + m_size, v_margin + gap + m_size),
        Point(h_margin, v_margin + o_size), Point(h_margin + o_size // 2, v_margin + o_size),
        Point(h_margin + o_size, v_margin + o_size)
    ]
```

```
# Draw points and connections on the screen
connections = [
    (0, 1), (0, 9), (1, 4), (1, 2), (2, 14), (3, 4), (3, 10), (4, 5), (4, 7), (5, 13),
    (6, 7), (6, 11), (7, 8), (8, 12), (9, 10), (9, 21), (10, 11), (10, 18), (11, 15),
    (12, 13), (12, 17), (13, 14), (13, 20), (14, 23), (15, 16), (16, 17), (16, 19),
    (18, 19), (19, 20), (19, 22), (21, 22), (22, 23)
]

for con in connections:
    pygame.draw.line(screen, White, start_pos: (points[con[0]].x, points[con[0]].y), end_pos: (points[con[1]].x, points[con[1]].y), width: 3)
    points[con[0]].connections.append(points[con[1]])
    points[con[1]].connections.append(points[con[0]])

pygame.draw.circle(screen, P_Colour, center: (1050, 620), radius: 30)
pygame.draw.circle(screen, P1_Colour, center: (1050, 620), radius: 25)
pygame.draw.circle(screen, P_Colour, center: (1050, 720), radius: 30)
pygame.draw.circle(screen, P2_Colour, center: (1050, 720), radius: 25)

return points,connections
```

2. piece_place() :

This function is used in the placing phase and takes player_turn and coordinates as input. Returns true if the point was empty and the player could successfully occupy it.

```
def piece_place(player, points, x_mouse, y_mouse):  1 usage
    for poi in points:
        if not poi.occupied and is_near(x_mouse, y_mouse, poi.x, poi.y):
            poi.occupy(player)
            return True
    return False
```

3. move_piece() :

This function is used in the moving phase and takes player_turn ,from_point and to_point as inputs. Returns true if the player could successfully move his piece from from_point to to_point. This event takes place across two mouse click events. 1st time for the from_point coordinates and 2nd time for the to_point coordinates.

```
def piece_move(player, points, from_point, to_point):  1 usage
    if from_point.occupied and from_point.player == player:
        if to_point in from_point.connections and not to_point.occupied:
            to_point.occupy(player)
            from_point.vacate()
            return True
    return False
```

4. mill_detector() :

This function is used to detect if new mills are formed after a piece is placed or moved. It collects the list of all mills formed now and compares it with the list of mills formed previously. If both the lists don’t match, a new mill is formed. The player_turn and the list of points,mills are taken as input and it returns if a new mill is formed and the list of immune pieces of the player.

```python
def mill_detector(player, points, mills):  3 usages
    immune = []
    global player1_mills, player2_mills, last_move_mills, miller, processed_mills, broken_mills
    new_mill_formed = False
    current_player_mills = set()

    # Find all current mills for the player
    for mil in mills:
        if points[mil[0]].player == points[mil[1]].player == points[mil[2]].player == player:
            mill_tuple = tuple(sorted([id(points[mil[0]]), id(points[mil[1]]), id(points[mil[2]])]))
            current_player_mills.add(mill_tuple)
            immune.extend([points[mil[0]], points[mil[1]], points[mil[2]]])

            # Check if this is a new mill or a reformed mill
            if player == 1:
                if mill_tuple not in player1_mills or mill_tuple in broken_mills:
                    player1_mills.add(mill_tuple)
                    if mill_tuple not in processed_mills or mill_tuple in broken_mills:
                        new_mill_formed = True
                        last_move_mills.add(mill_tuple)
                        broken_mills.discard(mill_tuple)  # Remove from broken mills if it was there
            else:
                if mill_tuple not in player2_mills or mill_tuple in broken_mills:
                    player2_mills.add(mill_tuple)
                    if mill_tuple not in processed_mills or mill_tuple in broken_mills:
                        new_mill_formed = True
                        last_move_mills.add(mill_tuple)


                        broken_mills.discard(mill_tuple)  # Remove from broken mills if it was there

    return immune, new_mill_formed
```

5. draw_mills() :

This function is used to highlight the mills formed every turn. This is kept outside the main loop just for code neatness.

```
def draw_mills(screen, points):  1 usage
    # Draw Player 1's mills in Red
    for mill_tuple in player1_mills:
        mill_points = []
        for point in points:
            if id(point) in mill_tuple:
                mill_points.append(point)
        if len(mill_points) == 3:
            pygame.draw.line(screen, P1_Colour,
                             start_pos: (mill_points[0].x, mill_points[0].y),
                             end_pos: (mill_points[2].x, mill_points[2].y),  width: 10)

    # Draw Player 2's mills in Blue
    for mill_tuple in player2_mills:
        mill_points = []
        for point in points:
            if id(point) in mill_tuple:
                mill_points.append(point)
        if len(mill_points) == 3:
            pygame.draw.line(screen, P2_Colour,
                             start_pos: (mill_points[0].x, mill_points[0].y),
                             end_pos: (mill_points[2].x, mill_points[2].y),  width: 10)
```

6. check_broken_mills() :

When a mill is broken and formed again to remove an opponent's piece, we have to remove the mill out of the saved mills list onc the mill is broken. This is the main purpose of the function.

```python
def check_broken_mills(points):  2 usages
    """Check and remove any mills that are no longer valid"""
    global player1_mills, player2_mills, broken_mills

    # Check Player 1's mills
    invalid_mills_p1 = set()
    for mill_tuple in player1_mills:
        mill_points = []
        for point in points:
            if id(point) in mill_tuple:
                mill_points.append(point)
        if len(mill_points) != 3 or any(p.player != 1 for p in mill_points):
            invalid_mills_p1.add(mill_tuple)
            broken_mills.add(mill_tuple)  # Add to broken mills when a mill is broken
    player1_mills -= invalid_mills_p1

    # Check Player 2's mills
    invalid_mills_p2 = set()
    for mill_tuple in player2_mills:
        mill_points = []
        for point in points:
            if id(point) in mill_tuple:
                mill_points.append(point)
        if len(mill_points) != 3 or any(p.player != 2 for p in mill_points):
            invalid_mills_p2.add(mill_tuple)
            broken_mills.add(mill_tuple)  # Add to broken mills when a mill is broken
    player2_mills -= invalid_mills_p2
```

7. remove_opponent_piece() :

This function is called after the mill is formed, and it ensures that when an opponent's piece is removed, all the edge case rules are considered and removal occurs properly. It takes the immune list of opponent's pieces from mill_detector() and works accordingly following the game rules.

```
def remove_opponent_piece(player, points, piece_dead_1, piece_dead_2, x_mouse, y_mouse):  2 usages
    global processed_mills, miller, broken_mills
    immune, _ = mill_detector(3 - player, points, mills)
    all_pieces_in_mill = True

    # Check if all remaining opponent pieces are in mills
    for poi in points:
        if poi.occupied and poi.player == 3 - player and poi not in immune:
            all_pieces_in_mill = False
            break

    for poi in points:
        if poi.occupied and poi.player == 3 - player and is_near(x_mouse, y_mouse, poi.x, poi.y):
            if poi not in immune or all_pieces_in_mill:
                if 3 - player == 1:
                    piece_dead_1 += 1
                else:
                    piece_dead_2 += 1
                poi.vacate()
                processed_mills.update(last_move_mills)
                last_move_mills.clear()
                miller = False
                check_broken_mills(points)  # Check for newly broken mills
                return piece_dead_1, piece_dead_2, True
    return piece_dead_1, piece_dead_2, False
```

8. check_win() :

This function is used to check if any player has won after every move. It checks both the cases of a player’s immobility and number of pieces alive.

```python
def check_win(player,game_phase, piece_count_1, piece_count_2, piece_dead_1, piece_dead_2,connections,points):  1 usage
    if game_phase == "moving":
        if (piece_count_1 - piece_dead_1) < 3:
            return 2  # Player 2 wins
        elif (piece_count_2 - piece_dead_2) < 3:
            return 1  # Player 1 wins
        j=True
        for poi in points:
            if poi.player == player and poi.occupied :
                for con in poi.connections:
                    if con.occupied == False:
                        j=False
                        break
            if not j:
                break
        if j:
            return 3-player
    return None
```

9. draw_highlights() :

This function is used during the moving phase. When a point is selected to be moved, this function displays all the possible places to which the piece can move.

```python
def draw_highlights(screen, selected_point, points, game_phase):  1 usage
    if selected_point and game_phase == "moving":
        # Highlight selected piece
        pygame.draw.circle(screen,  color: (0, 255, 0),  center: (selected_point.x, selected_point.y),  radius: 35)
        # Highlight possible moves
        for con in selected_point.connections:
            if not con.occupied:
                pygame.draw.circle(screen, P_Colour,  center: (con.x, con.y),  radius: 20)
```

CONCLUSION

In conclusion, developing the Nine Men's Morris game allowed me to integrate classic gameplay mechanics with unique visual elements, enriching the user experience. This project enhanced my skills in Python and Pygame, especially in creating interactive game logic and a thematic interface. Overcoming challenges in board design, piece movement, and win condition handling deepened my problem-solving abilities. While the game meets core functionality, future improvements could include animations and sound effects for added immersion. Overall, this project provided valuable insights into game development, and I’m eager to explore further enhancements.