The Hottest New NYT Game?

Vamshi Jandhyala

5/18/2025

A nice puzzle by Xavier Durawa for the week of 5/11.

Contents

1 Puzzle

You and your friend are playing a variant of the game Dots and Boxes. In this variation, your board is now a regular octagon. You take turns drawing lines between two non-neighboring vertices with the following rules:

- You can't cross an existing line
- If your line creates a triangular face, you get to claim that face and score +1. If you manage to create two triangular faces, you get both of those faces for +2
- You and your opponent proceed until no more lines can be drawn

At the end, whoever has more points wins.

The questions for this week are:

(i) If both players play randomly, what's the probability that Player 1 wins?

(ii) What is the optimal play for both players?

(iii) Under optimal play, what is the outcome of the game?

2 Solution

Let us create the game tree for the game. Each node of the tree represents the current **state** of the game. The state can be represented by the number of sides of all the polygons into which the original octagon has been sub divided. Each move by a player creates new polygons from the existing set of polygons. At any point in the game, the number of sides of the polygons involved can be $8, 7, 6, 5, 4$ or 3. To arrive at the set of all the valid moves for a player, we first look at the current set of polygons and then see in how many ways each polygon can be subdivided.

2.1 Octagon

For an octagon with 8 vertices labeled 1 through 7:

- Total pairs of vertices: $\binom{8}{2} = 28$.
- Number of edges: 8.
- Non-adjacent pairs: $28 - 8 = 20$ pairs.

The non-adjacent vertex pairs can be classified by the distance between the vertices along the perimeter:

- Distance 2: Vertices like $(1, 3), (2, 4)$, etc. (8 such pairs).
- Distance 3: Vertices like $(1, 4), (2, 5)$, etc. (8 such pairs).
- Distance 4: Vertices like $(1, 5), (2, 6)$, etc. (4 such pairs).

Under the octagon's symmetry:

- All distance-2 pairs are equivalent to each other.
- All distance-3 pairs are equivalent to each other.
- All distance-4 pairs are equivalent to each other.

Therefore, there are 3 distinct ways to select two non-adjacent vertices of an octagon up to symmetry.

2.2 Heptagon

For a heptagon with 7 vertices labeled 1 through 7:

- Total pairs of vertices: $\binom{7}{2} = 21$.
- Number of edges: 7.
- Non-adjacent pairs: $21 - 7 = 14$ pairs.

These non-adjacent pairs can be classified by the distance between vertices along the perimeter:

- Distance 2: Vertices like $(1, 3), (2, 4)$, etc. - 7 such pairs.
- Distance 3: Vertices like $(1, 4), (2, 5)$, etc. - 7 such pairs.
- All distance-2 pairs form one equivalence class (they can all be transformed into each other).
- All distance-3 pairs form another equivalence class.

Therefore, for a heptagon, there are 2 distinct ways to select two non-adjacent vertices up to symmetry.

2.3 Hexagon

A hexagon has 6 vertices.

- Total pairs of vertices: $\binom{6}{2} = 15$.
- Number of edges: 6.
- Non-adjacent pairs: $15 - 6 = 9$ pairs.

These can be classified by distance:

- Distance 2: Vertices like $(1, 3), (2, 4)$, etc. (6 such pairs).
- Distance 3: Vertices like $(1, 4), (2, 5)$, etc. (3 such pairs).

Under hexagon symmetry:

- All distance-2 pairs are equivalent to each other.
- All distance-3 pairs are equivalent to each other.

Therefore, for a hexagon, there are 2 distinct ways to select two non-adjacent vertices of a hexagon up to symmetry.

2.4 Pentagon

A pentagon has 5 vertices.

- Total pairs of vertices: $\binom{5}{2} = 10$.
- Number of edges: 5.
- Non-adjacent pairs: $10 - 5 = 5$ pairs.

Due to the pentagon's symmetry properties, all non-adjacent vertex pairs belong to the same equivalence class. That means they can all be transformed into each other through some combination of rotations and reflections.

Therefore, for a pentagon there is 1 distinct way to select two non-adjacent vertices up to symmetry

2.5 Square

A square has 4 vertices

- Total pairs of vertices: $\binom{4}{2} = 6$.
- Number of edges: 4.
- Non-adjacent pairs: $6 - 4 = 2$ pairs.

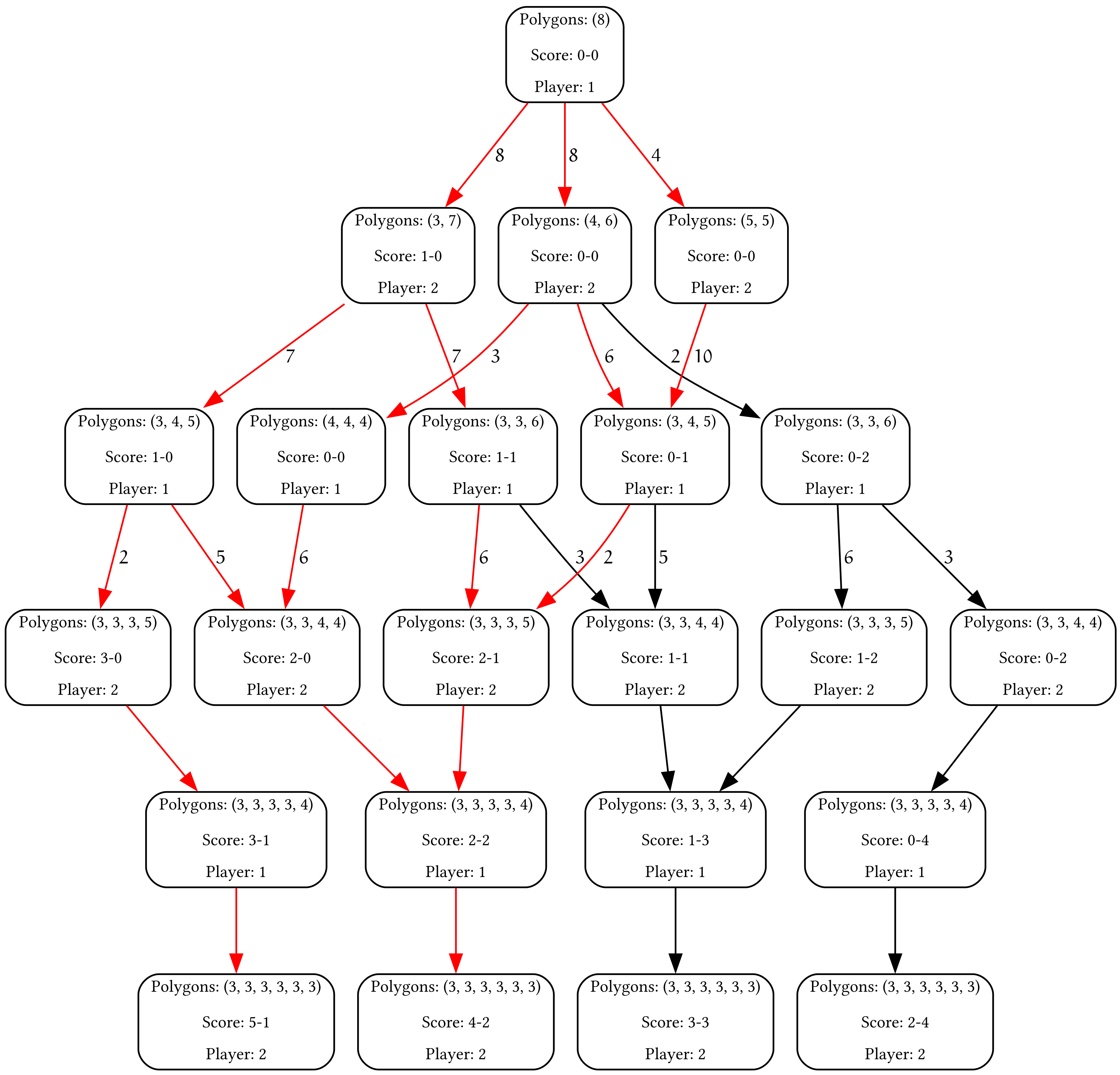
These are the diagonal pairs: $(1, 3)$ and $(2, 4)$.

Under square symmetry, all diagonal pairs are equivalent.

Therefore, for a square there is only $\mathbf{1}$ distinct way to select two non-adjacent vertices of a square up to symmetry.

3 Game tree

Armed with the above information, we can now create the following game tree. The number on the edge tells us the number of ways to get from the source node to the target node. The winning paths for player 1 are highlighted in red.



Polygons: (8) Score: 0-0 Player: 1
8
8
4
Polygons: (3, 7) Score: 1-0 Player: 2
Polygons: (4, 6) Score: 0-0 Player: 2
Polygons: (5, 5) Score: 0-0 Player: 2
7
7
3
6
2
10
Polygons: (3, 4, 5) Score: 1-0 Player: 1
Polygons: (4, 4, 4) Score: 0-0 Player: 1
Polygons: (3, 3, 6) Score: 1-1 Player: 1
Polygons: (3, 4, 5) Score: 0-1 Player: 1
Polygons: (3, 3, 6) Score: 0-2 Player: 1
2
5
6
6
3
2
5
6
3
Polygons: (3, 3, 3, 5) Score: 3-0 Player: 2
Polygons: (3, 3, 4, 4) Score: 2-0 Player: 2
Polygons: (3, 3, 3, 5) Score: 2-1 Player: 2
Polygons: (3, 3, 4, 4) Score: 1-1 Player: 2
Polygons: (3, 3, 3, 5) Score: 1-2 Player: 2
Polygons: (3, 3, 4, 4) Score: 0-2 Player: 2
Polygons: (3, 3, 3, 3, 4) Score: 3-1 Player: 1
Polygons: (3, 3, 3, 3, 4) Score: 2-2 Player: 1
Polygons: (3, 3, 3, 3, 4) Score: 1-3 Player: 1
Polygons: (3, 3, 3, 3, 4) Score: 0-4 Player: 1
Polygons: (3, 3, 3, 3, 3, 3) Score: 5-1 Player: 2
Polygons: (3, 3, 3, 3, 3, 3) Score: 4-2 Player: 2
Polygons: (3, 3, 3, 3, 3, 3) Score: 3-3 Player: 2
Polygons: (3, 3, 3, 3, 3, 3) Score: 2-4 Player: 2

Figure 1: **Full Game Tree**

The probability of player 1 winning when both players play at random is given by

$$\frac{8}{20}\cdot\frac{7}{14}+\frac{8}{20}\cdot\frac{3}{11}+\frac{8}{20}\cdot\frac{7}{14}\cdot\frac{6}{9}+\frac{8}{20}\cdot\frac{6}{11}\cdot\frac{2}{7}+\frac{4}{20}\cdot 1\cdot\frac{2}{7}=\frac{59}{105}=56.19\%. \tag{3.1}$$

The probability of player 2 winning when both players play at random is given by

$$\frac{8}{20}\cdot\frac{2}{11}\cdot\frac{3}{9}=\frac{4}{165}=2.42\%. \tag{3.2}$$

The optimal play is as follows which ends with a score of $4-2$.

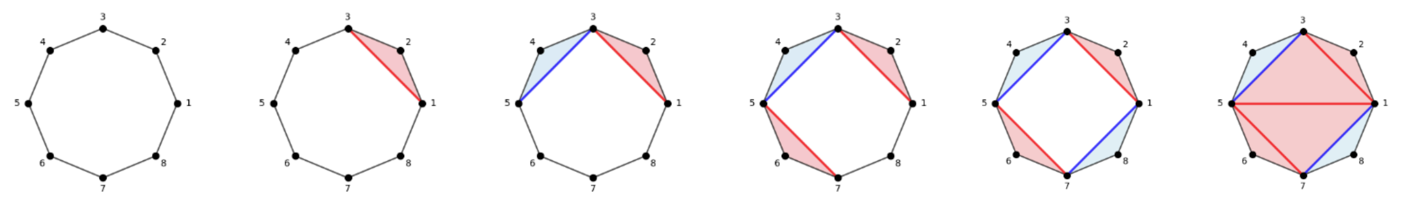
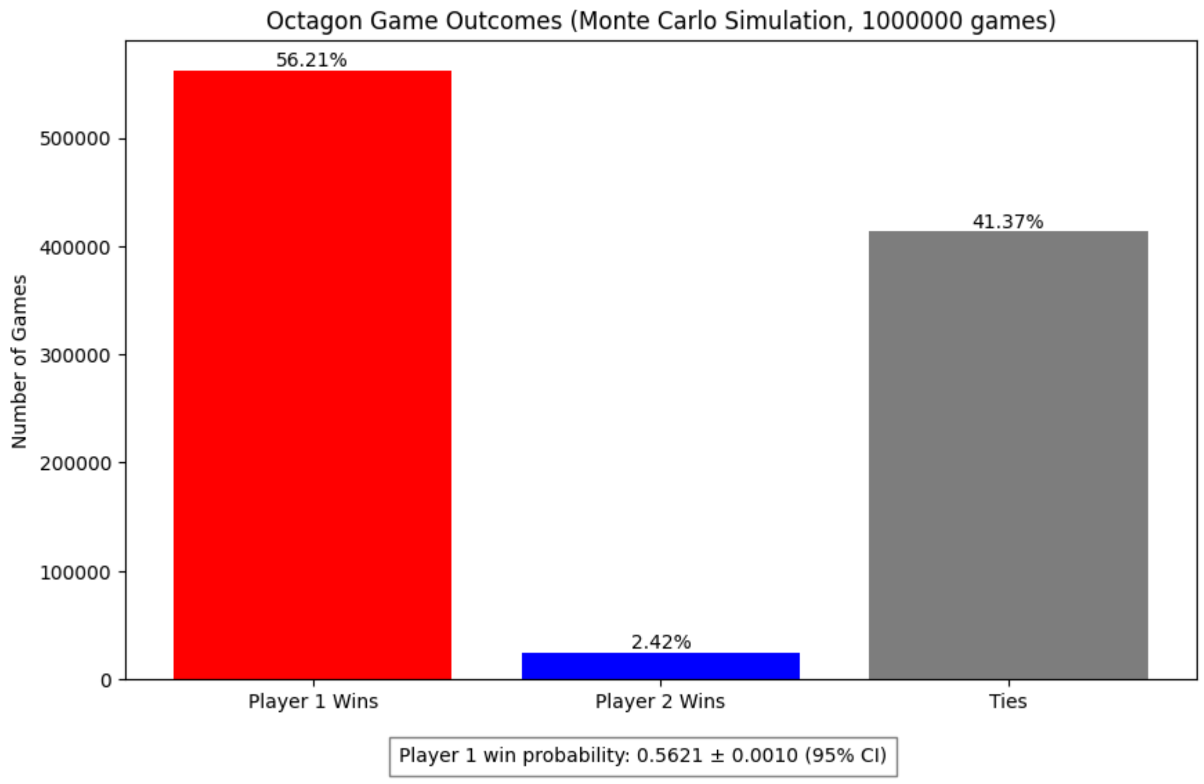


Figure 2: **Optimal play, Player 1 moves in red and Player 2 moves in blue.**

4 Monte Carlo Simulation

Here are the results of the Monte-Carlo simulation of both the players making random moves.



4.1 Python Code

```
import networkx as nx
import matplotlib.pyplot as plt
from collections import deque
import numpy as np
from matplotlib.patches import Polygon
import matplotlib.patches as mpatches
import random

class OctagonGame:
    """
    A comprehensive class for the Dots and Boxes variant with an octagon.
    Includes game simulation, game tree generation, optimization, and visualizations.
    """

    def __init__(self):
        # Create a regular octagon
        self.num_vertices = 8
        self.angles = np.linspace(0, 2*np.pi, self.num_vertices+1)[:-1]  # 8 angles
        self.vertices = np.array([np.cos(self.angles), np.sin(self.angles)]).T

        # Initialize game state
        self.score1 = 0
        self.score2 = 0
        self.player = 1  # Player 1 starts
        # Track lines and triangles
        self.lines = []  # (v1, v2, player) tuples
        self.triangles = []  # (v1, v2, v3, player) tuples

        # Initial edges of the octagon
        self.edges = [(i, (i+1) % self.num_vertices) for i in
range(self.num_vertices)]

        # Track regions (initially just the octagon)
        self.regions = [list(range(self.num_vertices))]  # List of vertices forming
each region

        # Game history
        self.history = []  # List of (lines, triangles, score1, score2, player)
tuples

        # Current polygon state (for compatibility with game tree)
        self.polygon_state = ((8,), 0, 0, 1)  # (polygons, score1, score2, player)

        # Game tree cache
        self.game_tree = None
        self.minimax_memo = {}
        self.best_moves = {}

    def reset(self):
        """Reset the game to initial state."""
        self.score1 = 0
        self.score2 = 0
        self.player = 1
        self.lines = []
        self.triangles = []
        self.regions = [list(range(self.num_vertices))]
        self.history = []
        self.polygon_state = ((8,), 0, 0, 1)
        # Don't reset game tree cache

    #--------------------- Core Game Logic ---------------------#

    def get_valid_moves(self):
        """Get all valid moves (pairs of vertices that can be connected)."""
        valid_moves = []

        # Check each region
        for region in self.regions:
            if len(region) < 4:
                continue  # Need at least 4 vertices to draw a non-adjacent line

            # Check all pairs of vertices in this region
            for i, v1 in enumerate(region):
                for j in range(i+2, len(region)):  # Skip adjacent vertices
                    v2 = region[j]
```

```python
                    # Skip if they're adjacent in the region
                    if j == i + 1 or (i == 0 and j == len(region) - 1):
                        continue

                    # Check if this would be a valid move
                    if self.is_valid_move(v1, v2):
                        valid_moves.append((v1, v2))

        return valid_moves

    def is_valid_move(self, v1, v2):
        """Check if connecting vertices v1 and v2 is a valid move."""
        # Ensure vertices are in the same region
        same_region = False
        for region in self.regions:
            if v1 in region and v2 in region:
                same_region = True
                break

        if not same_region:
            return False

        # Check if line already exists
        for line_v1, line_v2, _ in self.lines:
            if (v1 == line_v1 and v2 == line_v2) or (v1 == line_v2 and v2 ==
line_v1):
                return False

        # Check if line intersects any existing lines
        for line_v1, line_v2, _ in self.lines:
            if self.line_segments_intersect(v1, v2, line_v1, line_v2):
                return False

        return True

    def line_segments_intersect(self, v1, v2, v3, v4):
        """Check if line segments (v1,v2) and (v3,v4) intersect."""
        # Skip if they share an endpoint
        if v1 == v3 or v1 == v4 or v2 == v3 or v2 == v4:
            return False

        p1 = self.vertices[v1]
        p2 = self.vertices[v2]
        p3 = self.vertices[v3]
        p4 = self.vertices[v4]

        def ccw(a, b, c):
            """Check if three points are counter-clockwise oriented."""
            return (c[1] - a[1]) * (b[0] - a[0]) > (b[1] - a[1]) * (c[0] - a[0])

        # Check if line segments intersect
        return ccw(p1, p3, p4) != ccw(p2, p3, p4) and ccw(p1, p2, p3) != ccw(p1, p2,
p4)
```

```
    def make_move(self, v1, v2):
        """Make a move by connecting vertices v1 and v2."""
        if not self.is_valid_move(v1, v2):
            return False

        # Save current state for history
        self.history.append((
            self.lines.copy(),
            self.triangles.copy(),
            self.score1,
            self.score2,
            self.player
        ))

        # Add the line
        self.lines.append((v1, v2, self.player))

        # Find which region contains both vertices
        target_region = None
        target_index = -1
        for i, region in enumerate(self.regions):
            if v1 in region and v2 in region:
                target_region = region
                target_index = i
                break

        if target_region is None:
            return False  # Shouldn't happen if is_valid_move returned True

        # Split the region
        # Find the indices of v1 and v2 in the region
        i1 = target_region.index(v1)
        i2 = target_region.index(v2)

        # Ensure i1 < i2
        if i1 > i2:
            i1, i2 = i2, i1
            v1, v2 = v2, v1

        # Split into two regions
        region1 = target_region[i1:i2+1]  # Vertices from v1 to v2
        region2 = target_region[i2:] + target_region[:i1+1]  # Remaining vertices +
v1

        # Replace the original region with the two new ones
        self.regions.pop(target_index)
        self.regions.append(region1)
        self.regions.append(region2)

        # Check if triangles were formed
        triangles_formed = 0
        if len(region1) == 3:
            self.triangles.append((region1[0], region1[1], region1[2], self.player))
            triangles_formed += 1
        if len(region2) == 3:
            self.triangles.append((region2[0], region2[1], region2[2], self.player))
            triangles_formed += 1

        # Update score
        if self.player == 1:
            self.score1 += triangles_formed
        else:
            self.score2 += triangles_formed

        # Update polygon state for game tree compatibility
        # Get the sizes of all regions
        polygon_sizes = tuple(sorted([len(r) for r in self.regions]))

        # Switch player
        self.player = 3 - self.player  # 1 -> 2, 2 -> 1

        # Update polygon state with new player
        self.polygon_state = (polygon_sizes, self.score1, self.score2, self.player)

        return True

    def random_move(self):
        """Make a random valid move."""
        valid_moves = self.get_valid_moves()
        if not valid_moves:
            return False  # No valid moves

        v1, v2 = random.choice(valid_moves)
        return self.make_move(v1, v2)

    def play_game(self):
        """Play a complete game with random moves."""
        self.reset()
        game_over = False
        while not game_over:
            # Try to make a random move
            if not self.random_move():
                game_over = True

        # Add final state to history
        self.history.append((
            self.lines.copy(),
            self.triangles.copy(),
            self.score1,
            self.score2,
            self.player
        ))

        # Return the final scores
        return self.score1, self.score2

    def is_game_over(self):
        """Check if the game is over (no more lines can be drawn)."""
        return len(self.get_valid_moves()) == 0


    def run_monte_carlo_simulation(self, num_games=1000, verbose=True):
        """Run a Monte Carlo simulation of the octagon game."""
        if verbose:
            print(f"Running Monte Carlo simulation with {num_games} games...")

        # Track results
        player1_wins = 0
        player2_wins = 0
        ties = 0

        # Run the simulations
        for i in range(num_games):
            # Display progress every 10% if verbose
            if verbose and i % (num_games // 10) == 0:
                print(f"Progress: {i / num_games * 100:.1f}%")

            # Play a game
            score1, score2 = self.play_game()

            # Record the result
            if score1 > score2:
                player1_wins += 1
            elif score2 > score1:
                player2_wins += 1
            else:
                ties += 1

        # Calculate probabilities
        p1_win_prob = player1_wins / num_games
        p2_win_prob = player2_wins / num_games
        tie_prob = ties / num_games

        # Print results if verbose
        if verbose:
            print("\nResults:")
            print(f"Player 1 wins: {player1_wins} games ({p1_win_prob:.4f} or
{p1_win_prob*100:.2f}%)")
            print(f"Player 2 wins: {player2_wins} games ({p2_win_prob:.4f} or
{p2_win_prob*100:.2f}%)")
            print(f"Ties: {ties} games ({tie_prob:.4f} or {tie_prob*100:.2f}%)")

            # Calculate 95% confidence interval for Player 1 win probability
            std_dev = np.sqrt(p1_win_prob * (1 - p1_win_prob) / num_games)
            margin_of_error = 1.96 * std_dev

            print(f"\n95% confidence interval for Player 1 win probability: "
                  f"{p1_win_prob:.4f} ± {margin_of_error:.4f} "
                  f"({p1_win_prob - margin_of_error:.4f} to {p1_win_prob + 
margin_of_error:.4f})")

        # Visualize results
        labels = ['Player 1 Wins', 'Player 2 Wins', 'Ties']
        values = [player1_wins, player2_wins, ties]
        colors = ['red', 'blue', 'gray']

        plt.figure(figsize=(10, 6))
        bars = plt.bar(labels, values, color=colors)

        # Add percentage labels on top of bars
        for bar in bars:
            height = bar.get_height()
            percentage = height / num_games * 100
            plt.text(bar.get_x() + bar.get_width()/2., height + 0.1,
                    f'{percentage:.2f}%', ha='center', va='bottom')

        plt.ylabel('Number of Games')
        plt.title(f'Octagon Game Outcomes (Monte Carlo Simulation, {num_games} 
games)')

        # Add confidence interval information to the plot
        std_dev = np.sqrt(p1_win_prob * (1 - p1_win_prob) / num_games)
        margin_of_error = 1.96 * std_dev

        plt.figtext(0.5, 0.01,
                    f"Player 1 win probability: {p1_win_prob:.4f} ± 
{margin_of_error:.4f} (95% CI)",
                    ha="center", fontsize=10, bbox={"facecolor":"white", "alpha":0.5, 
"pad":5})

        if verbose:
            plt.savefig("octagon_monte_carlo_results.png", dpi=300, 
bbox_inches='tight')
            print("Results visualization saved as 'octagon_monte_carlo_results.png'")

        return {
            'player1_wins': player1_wins,
            'player2_wins': player2_wins,
            'ties': ties,
            'p1_win_prob': p1_win_prob,
            'p2_win_prob': p2_win_prob,
            'tie_prob': tie_prob,
            'plot': plt
        }

def main():
    import matplotlib.pyplot as plt
    # Create an instance of the game
    game = OctagonGame()
    game.run_monte_carlo_simulation(10000)

if __name__ == "__main__":
    main()
```