Predicting Squash Outcomes with a 3D Markov Chain

Oliver Eielson

April 2024

Abstract

This study aims to create a predictive model for squash matches that integrates players' mental states into a 3D Markov chain [1]. Current models fail to incorporate players' mental states in their prediction, which often results in lower performance. Unlike the current models, players' mental states are built directly into the model, which results in more robust predictions. The model employs a 3D Markov chain where the x and y axis model the state of the game while the z-axis models the player's mental state. The resulting model now allows for accurate predictions from any point in the match for any combination of players. Additionally, initial results have validated some yet unproven theories of players' optimal strategies. The model has validated the theory that the optimal strategy for weaker players is usually to devolve the match into chaos in an attempt to force mental factors to be more important than skills. The model improves current prediction capabilities and improves our understanding of how psychological factors play into match outcomes.

Summary of Problem

Predicting the outcome of sports, specifically squash, is a notoriously difficult problem. Many factors need to be considered to model the outcome of a game accurately. Currently, some models exist that attempt to predict a player's skill level based on past results. However, these models routinely fall short as, for various factors, players do not always play at "their skill level." One of the main reasons for this varying performance is players' psychological and mental states. For example, a scared or nervous player will often play worse one match but perform far better with a different mental state. Furthermore, these mental states can change throughout the match as seen with the concept of momentum. The goal of this project is to build a model that can more accurately predict outcomes by accounting for these mental fluctuations.

Literature Review

As squash is a relatively niche sport, there are very few models that have been specifically designed to model squash. Currently, the only feasible model that exists is the US squash rating system[2]. The exact details of the model are secret, but it is loosely based on the ELO rating system. It assigns each player a rating from 0 to around 7 (there is no upper bound) based on previous match performance. However, this model only observes the final match outcome, leaving out the dynamics of the match.

While there are few models that specifically model squash, there are some applicable models designed for tennis. While playing tennis is very different than squash, from a mathematical modeling perspective the two sports are relatively similar. Most papers attempt to create a rating system similar to US squash or use statistical analysis to predict the outcomes of matches. However, very few papers have attempted to model the game to predict the outcome. One such paper by Sébastien Cararo, models tennis games using Markov chains[1]. The model he created works by defining a game as a state machine and served as the initial inspiration for this paper. However, unlike the model proposed in this paper his model assumes a constant probability of winning each point and does not include psychological factors.

[1]Explanation of Squash

Model

The proposed model uses a 3D Markov chain to predict the winner of a squash game. A Markov chain is well suited to this problem because the rules of squash can easily be encoded as a state machine (Figure 1). A standard squash match is played as a best of 5 games, with each game to 11 points. However, like tennis, a player must win by two points to win a match. For simplicity of calculations, we will only model one game and reset the game back to the start if it reaches a tiebreak. Figure 1 below shows the 2D finite state machine that encodes these rules.

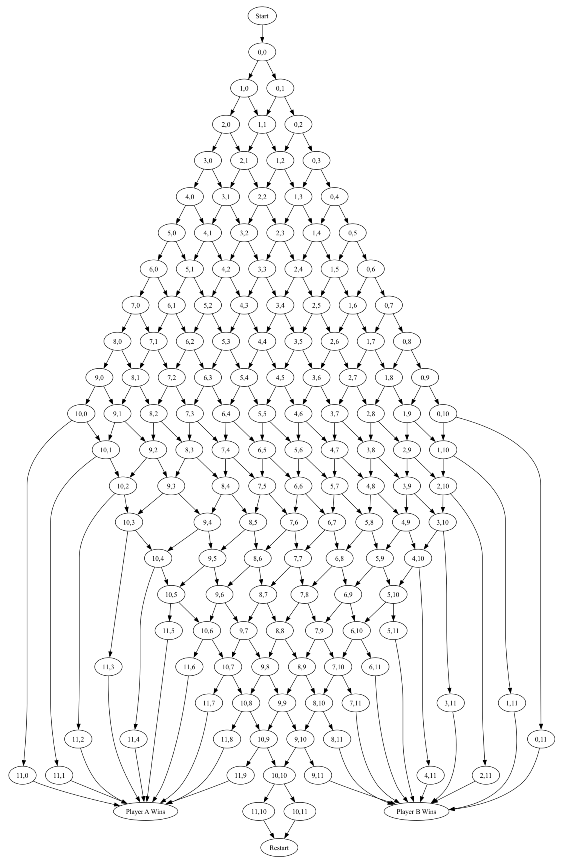


Figure 1: This diagram shows a simple FSA that describes the flow of a Squash match with 11-point scoring.

This simple state machine will be the basis for the new 3D Markov chain that incorporates players' mental states. The explanation of this new model has been broken up into two sections; An explanation of the graphical structure of the Markov chain and an explanation of the probabilities used to calculate the transition table.

Graph Structure

To incorporate psychological factors and fluctuations into the model, the graph in Figure 1 was augmented. Several versions of the graph in Figure 1 are created and stacked vertically, as seen in Figure 3. In the new 3D graph, each level represents a different mental state. The higher the level, the more player 1 is favored; the lower the level, the more player 2 is favored. This encodes the idea that if Player 1 has a psychological advantage, the probability of them winning a point is higher and vice versa. The levels were then connected together as shown in Figure 2. Each node at each level was connected to all the nodes at the other level. This encoded the idea that a player's mental state can change to any value at any point. The probabilities were set so that the further the level, the less likely the transition was. This was to incorporate the fact that large mental swings are far less likely than subtle changes. Regardless of level, all paths were sent to the same "winning" node. Like in Figure 1, the winning nodes were connected to the start node to make the chain circular. As a game can start at any level the start node was then connected to the (0,0) node at each level.

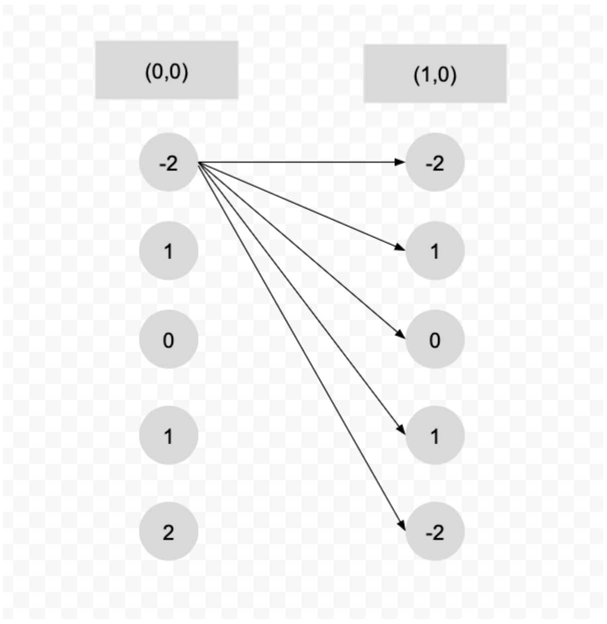


Figure 2: This shows the side view of the 3D model for the transition from the score (0,0) to the score (1,0). If the player wins the point, the mental state can jump from the current level to any other level in the chain.

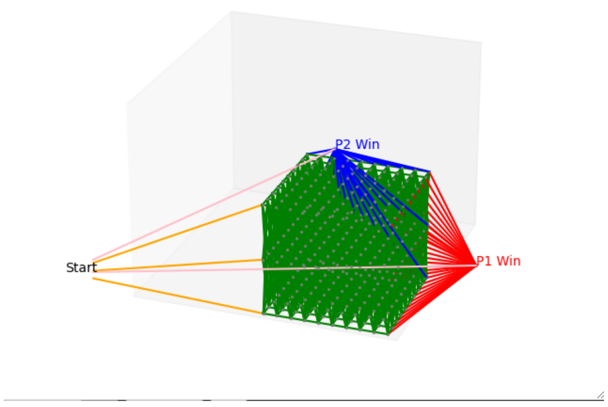


Figure 3: Shows the chain and encodes the match. Each green edge represents a connection between a point in the game, for example ((0,0) → (0,1)). The red and blue edges are connections to the winning nodes, and the orange/pink edges "reset" the game by connecting the winning nodes back to the start.

Equations

This section will outline the formulas used to create the transition probabilities. These probabilities will allow the graph outlined in the previous section to be converted into a transition matrix.

Notation

Below is some explanation for the notation used in equations.

- P1 = P (Probability that player 1 wins a point).
- P2 = (1-p) (Probability that player 2 wins a point).
- M_e is the mental effect and is a decimal value (0-1), describing how much the game is affected by the player's mental state.
- L_C is the current level.
- M_R is the mental range, which describes the number of levels.

Start Node → (0,0,m)

$$\mathbb{P}(0,0,m) = \frac{1}{M_r}$$

The start node is not a part of the game structure but was added to model the differing mental states going into a match. For example, say a player is about to play a long-time rival and is very nervous. His mental state from point 0-0 might favor the opponent. We assume that we are equally likely to start a match at any level.

Win Nodes → Start Node

$$\mathbb{P}(StartNode) = 1$$

The probability of going to the start node is 100%, as there are no other nodes to go to. This transition is only included in the chain to make the graph circular. This allows the model to reset itself after each game.

Game Nodes → Win Nodes

$$\mathbb{P}(P1_{Wins}) = P1 + M_e \times (L_c - \frac{1 + M_r}{2})$$
$$\mathbb{P}(P2_{Wins}) = P2 - M_e \times (L_c - \frac{1 + M_r}{2})$$

The probability of either player winning at any point is P1 or P2. However, these values are augmented by the current level and mental effect. Thus, to calculate the new probability, we calculate the current level, multiply it by the mental effect, and then add or subtract it from the original probability. The result is the probability of transitioning to the new node.

Game Nodes → Game Nodes

$$\text{P(x+1,y)} = \frac{P1 + M_e \times (L_c - \frac{1+M_r}{2}) \times \frac{1}{1+|L_c-L_n|}}{\sum_{M_r} \frac{1}{1+|L_c-L_n|}},$$
$$\text{P(x,y+1)} = \frac{P2 - M_e \times (L_c - \frac{1+M_r}{2}) \times \frac{1}{1+|L_c-L_n|}}{\sum_{M_r} \frac{1}{1+|L_c-L_n|}}.$$

Going from one game node to another is the most complicated case. This is because we must calculate the probability of either player winning and then calculate the probability of changing levels. To calculate the probability of either player winning, we use the same formula as in the last section. Then, to calculate changing levels, we use the formula $\frac{1}{1+|L_c-L_n|}$, which lowers the probability of changing levels the further away the next level is. We then normalize the probabilities to ensure that they sum to 1 in order to ensure the transition table remains stochastic.

Calculations

Once the graph and transition probabilities have been computed, the match's outcomes can be computed. The probability of being in any state can be calculated by finding the eigenvectors of the transition matrix and normalizing them. The probabilities for the nodes "P1 Win" and "P2 Win" are normalized and used as the final prediction.

Assumptions

- Initial Probabilities: We assume the base probabilities of each player winning a remains constant. For example, we assume that P1 does not improve their skill level during the match or developed a strategy mid-game that improves their odds of winning. Additionally, this assumes that neither player gets hurt during the match, which negatively affects their skill level.
- Game Reset: We disregard any concept of tie break as the dynamics remain the same as in the regular match. Additionally, since tie breaks can go infinitely, and Markov chains need to be finite removing the tie break simplified calculations.

- Mental Effect: The mental state has a symmetric effect on the probability of winning a point; it can either positively or negatively impact the player's performance. Thus, if a player's mental state positively increases his chance of winning by 5%, then it hurts the opponent's chance by 5%. Future versions of the model would benefit from removing this assumption and instead develop a model that is not symmetric.
- Mental State Dynamics: The mental state of a player at any given point is influenced by the game's progress and can transition between different levels, reflecting changes in player confidence or stress. Furthermore, we assume that it is more likely that a player's mental state changes by small increments and not massive jumps. This assumption closely matches presumed real-world probabilities. It is rare that a player goes from losing mentally to winning in one point, but it does happen sometimes.

Random Walk Example

Using the transition probabilities and graph structure, an example random walk was defined. The random walk does not help with calculations but provides an example of how the model works.

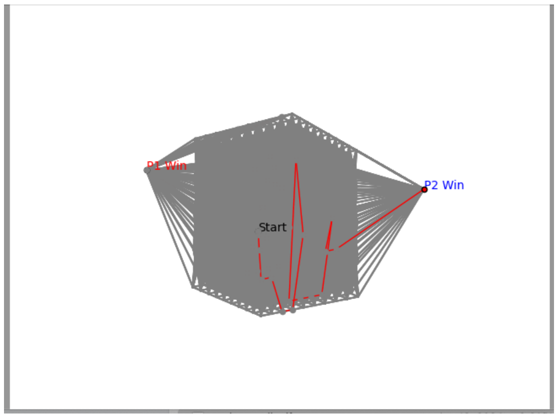


Figure 4: Click here if the animation does not load.

Discussion

Different simulations were run using the model described above and the code in the appendix. These simulations yielded two main results:

Chaos is a Ladder

While not a novel discovery, the model confirms a long-held theory that the weaker player's optimal strategy is to try to devolve the match into chaos. This was discovered by calculating the "game shape." A simulation was run for every combination of P and mental effect, resulting in Figure 5.

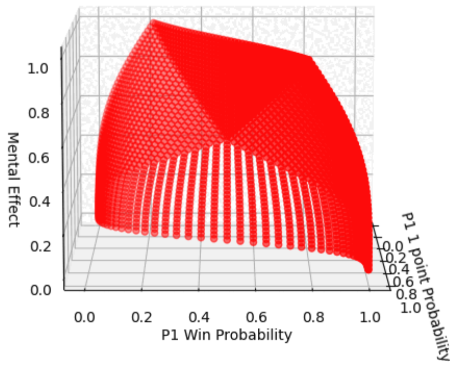
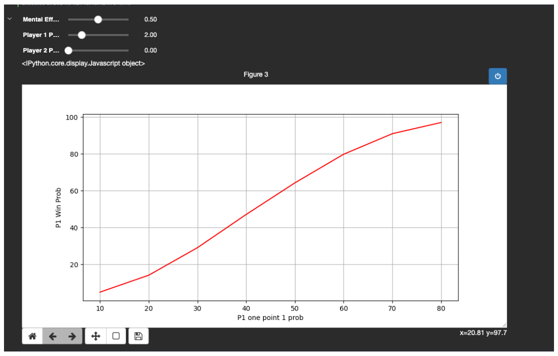


Figure 5: The X-axis is the probability that P1 wins a point, and the Y-axis is the probability that P1 wins the entire game. The Z-axis is the mental effect. If the animation does not work click here.

From the graph, it was observed that when the P1 probability was close to 0 or 1, the mental effect had a much larger effect on the outcome. For example, when P1 is 0.1 (P1 has a 10% chance of winning any point) and the mental effect is 0.1, player 1 only has a 1.148% chance of winning the game. However, if the mental effect is raised to .99, P1 now has a 17.5% chance of winning the game. Importantly, the chance of P1 winning the game is now higher than their chance of winning an individual point. However, when P1 and P2 are both 0.5, the mental effect has no effect on the percent chance of either player winning. This confirms the theory that the weaker player's optimal strategy is to try to devolve the match into chaos.

Predictions

As the model is very modular, it can be used to predict match outcomes. The figure below shows an interactive graph developed to predict match results in real-time quickly.



Mental Eff... 0.50
Player 1 P... 2.00
Player 2 P... 0.00
<IPython.core.display.Javascript object>

Figure 6: This shows the random walk.

The sliders allow the user to change the mental effect, and score of the match. The widget will them output a graph showing the chance of winning the match based on P1 chance of winning a point. This application of the model has the most real-world application as there are many times when predicting the outcome of a match is extremely useful. The original plan was the compare these results against real world results however, since US Squash hid all there data behind a ridiculous paywall no comparison could be made.

Model Weaknesses

Model Inputs

George E. P. Box famously said, "All models are wrong, but some are useful," and this is no exception. One of the model's larger weaknesses is its inputs. The model assumes that values for the percent chance of player 1 winning a point and the Mental effects are accurate, while in reality, it is extremely hard to estimate them. This problem was considered when designing the model; however, since there is no available match data, there was no way to overcome it other than just assuming a correct estimate.

Model Symmetry

Another weakness of the model is its symmetry. The model assumes that each player's mental state affects the match to the same degree. Some players are mentally stronger than others and thus have less of a chance of jumping from one level to another. This is often seen in professional sports, where some players are very stoic, and others are all over the place. However, the model assumes that each player jumps to new mental levels with the same probability. Future versions of the model should address this problem.

Model Validation

One weakness is that the model has not been validated against real world results. The original plan with the paper was to download a dataset of match results from US Squash. This would allow the new model to be compared with other models and determine the models accuracy. However, in another outrageous attempt to squeeze every penny out of their users US Squash recently hid the data behind an paywall. Future versions of the model would benefit from more rigorous validation.

Conclusion

In conclusion this paper proposes a novel model that uses a 3D Markov Chain to predict the outcome of squash matches. Unlike past models this model incorporates psychological factors into the Markov chain. Despite showing many promising results the model was unable to be verified due to a lack of data. However ultimately this model helped improve our understand of how mental factors effect the outcome of squash matches.

Appendix

Game Class

Below is the Game class that can be used to predict results and display the graphs.

```python
import numpy as np
from matplotlib import pyplot as plt, animation
import networkx as nx
from matplotlib.animation import FuncAnimation


class Game:

    def __init__(self, player1, player2, mental_effect, gameRange, mentalRange, draw_graph=False, P1_start=0,
                 P2_start=0):
        self.P1 = player1
        self.P2 = player2
        self.mental_effect = mental_effect
        self.draw_graph = draw_graph
        self.gameRange = gameRange
        self.mentalRange = mentalRange
        self.graph = {}
        self.P1_start = P1_start
        self.P2_start = P2_start

        self.nodes = []

        self.P1_win_relative = 0
        self.P2_win_relative = 0

        self.P1_win = 0
        self.P2_win = 0
        assert self.gameRange > 0, "Game range must be greater than 0"
        assert self.mentalRange > 0, "Mental range must be greater than 0"
        assert self.mental_effect > 0, "Mental effect for P1 must be greater than 0"
        assert self.P1 > 0, "P1 must be greater than 0"
        assert self.P2 > 0, "P2 must be greater than 0"
        assert self.P1 + self.P2 == 1, "P1 + P2 must be equal to 1"

    def create_graph(self):

        for m in range(self.mentalRange):
            self.graph[(-1, -1, self.mentalRange / 2), (self.P1_start, self.P2_start, m)] = (1 / self.mentalRange)

        for A in range(self.P1_start, self.gameRange):  # all of A's possible points
            for B in range(self.P2_start, self.gameRange):
                for z in range(self.mentalRange):  # all of the possible mental states
                    mental_modifier = self.mental_effect * ((z + 1) - ((1 + self.mentalRange) / 2))
                    prob_A = max(0, min(1, self.P1 + mental_modifier))
                    prob_B = max(0, min(1, self.P2 - mental_modifier))
                    assert prob_A >= 0, "Prob A is negative A = " + str(prob_A)
                    assert prob_B >= 0, "Prob B is negative B = " + str(prob_A)

                    total_jump_prob_A = 0
                    total_jump_prob_B = 0
                    for m in range(self.mentalRange):
                        total_jump_prob_A += 1 / (1 + abs(z - m))
                        total_jump_prob_B += 1 / (1 + abs(z - m))

                    for m in range(self.mentalRange):  # jumps to all the other mental states
                        jump_A = (prob_A * (1 / (1 + abs(z - m)))) / total_jump_prob_A
                        jump_B = (prob_B * (1 / (1 + abs(z - m)))) / total_jump_prob_B

                        if A == self.gameRange - 1:
                            self.graph[(A, B, z), (15, 0, self.mentalRange / 2)] = prob_A
                        else:
                            self.graph[(A, B, z), (A + 1, B, m)] = jump_A

                        if B == self.gameRange - 1:
                            self.graph[(A, B, z), (0, 15, self.mentalRange / 2)] = prob_B
                        else:
                            self.graph[(A, B, z), (A, B + 1, m)] = jump_B

        self.graph[(15, 0, self.mentalRange / 2), (-1, -1, self.mentalRange / 2)] = 1
        self.graph[(0, 15, self.mentalRange / 2), (-1, -1, self.mentalRange / 2)] = 1

    def createMatrix(self):
        self.nodes = list(set([x for y in self.graph.keys() for x in y]))
        matrix = np.zeros((len(self.nodes), len(self.nodes)))

        for key, value in self.graph.items():
            matrix[self.nodes.index(key[1])][self.nodes.index(key[0])] = value

        # check square
        assert matrix.shape[0] == matrix.shape[1], "Matrix is not square"
        # Check for non-Neg
        assert np.all(matrix >= 0), "Matrix has negative values"
        # Check for sum of cols (close enough)
        assert np.all(np.sum(matrix, axis=0) > 0.99), "Sum of cols is not 1"

        return matrix

    def calculate_eigenvector(self):
        matrix = self.createMatrix()
        eigenvalues, eigenvectors = np.linalg.eig(matrix)
        assert 0 == np.argmin(np.abs(eigenvalues - 1)), "no 1-eigenvector "

        eigenvector = np.real(eigenvectors[:, 0])
        normalized_eigenvector = eigenvector / np.sum(eigenvector)
        # get the index of the winning nodes
        IndexP1 = self.nodes.index((15, 0, self.mentalRange / 2))
        IndexP2 = self.nodes.index((0, 15, self.mentalRange / 2))

        self.P1_win_relative = normalized_eigenvector[IndexP1]
        self.P2_win_relative = normalized_eigenvector[IndexP2]

        self.P1_win = self.P1_win_relative / (self.P1_win_relative + self.P2_win_relative)
        self.P2_win = self.P2_win_relative / (self.P1_win_relative + self.P2_win_relative)

    def print_results(self):
        print("P1 wins relative: ", self.P1_win_relative)
        print("P2 wins relative: ", self.P2_win_relative)
        print("P1 wins: ", self.P1_win)
        print("P2 wins: ", self.P2_win)

    def draw_graph123(self, animate=False):
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3D')

        # turn off the backgrounds
        ax.xaxis.set_pane_color((1.0, 1.0, 1.0, 0.0))
        ax.yaxis.set_pane_color((1.0, 1.0, 1.0, 0.0))
        ax.zaxis.set_pane_color((1.0, 1.0, 1.0, 0.0))

        plt.tight_layout()
        ax.grid(False)
        ax.xaxis.line.set_visible(False)
        ax.yaxis.line.set_visible(False)
        ax.zaxis.line.set_visible(False)
        ax.set_xticks([])
        ax.set_yticks([])
```

```python
        ax.set_zticks([])

        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.set_zticklabels([])

        ax.set_zlim(0, 5)

        for key, value in self.graph.items():

            if key[1][0] == 15 and key[1][1] == 0:
                ax.plot([key[0][0], key[1][0]], [key[0][1], key[1][1]], [key[0][2], key[1][2]], 'red', linewidth=value)
            elif key[1][0] == 0 and key[1][1] == 15:
                ax.plot([key[0][0], key[1][0]], [key[0][1], key[1][1]], [key[0][2], key[1][2]], 'blue',
                        linewidth=value, )
            elif key[0][0] == 15 and key[0][1] == 0:
                ax.plot([key[0][0], key[1][0]], [key[0][1], key[1][1]], [key[0][2], key[1][2]], 'pink', linewidth=value)
            elif key[0][0] == 0 and key[0][1] == 15:
                ax.plot([key[0][0], key[1][0]], [key[0][1], key[1][1]], [key[0][2], key[1][2]], 'pink', linewidth=value)
            elif key[0][0] == -10 and key[0][1] == -10:
                ax.plot([key[0][0] + 9, key[1][0]], [key[0][1], key[1][1]], [key[0][2], key[1][2]], 'orange',
                        linewidth=value)
            else:
                ax.plot([key[0][0], key[1][0]], [key[0][1], key[1][1]], [key[0][2], key[1][2]], 'green',
                        linewidth=value)

            ax.scatter(key[0][0], key[0][1], key[0][2], color='grey', alpha=0.2, s=2, )

        ax.text(15, 0, self.mentalRange / 2, "P1 Win", color='red')
        ax.text(0, 15, self.mentalRange / 2, "P2 Win", color='blue')
        ax.text(-1, -1, self.mentalRange / 2, "Start", color='black')
        ax.scatter(-1, -1, self.mentalRange / 2, color='black', )
        ax.scatter(0, 15, self.mentalRange / 2, color='black', )
        ax.scatter(15, 0, self.mentalRange / 2, color='black', )

        def update(frame):
            # Update the azimuthal angle and elevation to rotate the view
            ax.view_init(elev=10, azim=frame)
            print(f"Animating frame {frame}/{360}")
            return fig,

        if animate:
            anim = FuncAnimation(fig, update, frames=np.arange(0, 360, 2), interval=50, blit=True, )
            anim.save('game_chain.gif', dpi=80, writer='imagemagick', )
        else:
            plt.show()

    def generate_random_walk(self, start_index, num_steps=50):

        current_index = self.nodes.index(start_index)
        path = [current_index]
        for _ in range(num_steps):
            probabilities = self.createMatrix()[:, current_index]
            if np.sum(probabilities) == 0:
                break
            next_index = np.random.choice(len(probabilities), p=probabilities)
            path.append(next_index)
            current_index = next_index

        node_path = [self.nodes[i] for i in path]
        return node_path

    def animate_walk(self):
        start_node = (-1, -1, self.mentalRange / 2)  # Adjust as necessary for your graph
        path = self.generate_random_walk(start_node, num_steps=200)  # Generate the walk

        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3D')
        ax.xaxis.set_pane_color((1.0, 1.0, 1.0, 0.0))
        ax.yaxis.set_pane_color((1.0, 1.0, 1.0, 0.0))
        ax.zaxis.set_pane_color((1.0, 1.0, 1.0, 0.0))

        ax.text(15, 0, self.mentalRange / 2, "P1 Win", color='red')
        ax.text(0, 15, self.mentalRange / 2, "P2 Win", color='blue')
        ax.text(-1, -1, self.mentalRange / 2, "Start", color='black')
        ax.scatter(-1, -1, self.mentalRange / 2, color='black', )
        ax.scatter(0, 15, self.mentalRange / 2, color='black', )
        ax.scatter(15, 0, self.mentalRange / 2, color='black', )

        plt.tight_layout()
        ax.grid(False)
        ax.xaxis.line.set_visible(False)
        ax.yaxis.line.set_visible(False)
        ax.zaxis.line.set_visible(False)
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_zticks([])

        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.set_zticklabels([])

        # Plot all graph connections lightly
        for key, value in self.graph.items():
            ax.plot([key[0][0], key[1][0]], [key[0][1], key[1][1]], [key[0][2], key[1][2]],
    'grey', alpha=1)

        # Prepare to animate the walk

        def update(num):
            node = path[num]
            node_old = path[num - 1]

            if node_old[0] == -1 and node_old[1] == -1:
                for key, value in self.graph.items():
                    ax.plot([key[0][0], key[1][0]], [key[0][1], key[1][1]], [key[0][2], key
    [1][2]], 'grey', alpha=1)

            ax.scatter(node_old[0], node_old[1], node_old[2], color='grey')
            ax.plot([node_old[0], node[0]], [node_old[1], node[1]], [node_old[2], node[2]],
    'red', linewidth=1)
            ax.scatter(node[0], node[1], node[2], color='red', s=5)
            ax.view_init(elev=-20, azim=num % 360)
            return fig,

        ani = FuncAnimation(fig, update, frames=len(path), blit=True, interval=200, repeat=
    False)
        ani.save('random_walk.gif', writer='imagemagick', fps=2)
        plt.show()
```

Jypter Note Book

Simple Prediction

```python
g = Game(player1=0.1, player2=0.9, mental_effect=.99, gameRange=10, mentalRange=3,
draw_graph=False, P1_start=0, P2_start=0)
g.create_graph()
```

```
    g.draw_graph123(animate=False)
    g.calculate_eigenvector()

    print(f'Player 1 Win Probability: {g.P1_win}')
    print(f'Player 2 Win Probability: {g.P2_win}')
```

Interactive Plot

```

def update_plot(mental_effect, point1, point2):
    fig, ax = plt.subplots(figsize=(10, 5))
    x_values = []
    y_values = []
    max_prob = 10
    for p in range(1, max_prob - 1):
        g = Game(player1=p / max_prob, player2= 1 - (p / max_prob), mental_effect=mental_effect, gameRange=10,
                 mentalRange=3, draw_graph=False, P1_start=int(point1), P2_start=int(point2))
        g.create_graph()
        g.calculate_eigenvector()
        x_values.append((p / max_prob) * 100)
        y_values.append(g.P1_win * 100)

    ax.plot(x_values, y_values, color='red')

    # Adding labels and legend
    ax.set_xlabel('P1 one point 1 prob')
    ax.set_ylabel('P1 Win Prob')
    plt.grid(True)
    plt.show()


mental_slider = FloatSlider(value=0.5, min=0.01, max=1.0, step=0.01, description='Mental Effect:')
point1_slider = FloatSlider(value=0, min=0, max=9, step=1, description='Player 1 Points')
point2_slider = FloatSlider(value=0, min=0, max=9, step=1, description='Player 2 Points')

interactive_plot = interactive(update_plot, mental_effect=mental_slider, point1=point1_slider, point2=point2_slider)
interactive_plot
    print(f'Player 2 Win Probability: {g.P2_win}')
```

References

[1] Sébastien Cararo. *Modeling a tennis match with Markov Chains*. Jan. 2021. URL: `https://medium.com/analytics-vidhya/modelizing-a-tennis-match-with-markov-chains-b59ca2b5f5bf`.

[2] US Squash. *UNIVERSAL SQUASH RATING (USR)*. Sept. 2020. URL: `https://ussquash.org/ratings/`.