EE6150: Stochastic Modelling and Queuing Theory

Python Assignment

Amizhthni P R K, EE21B015

April 19, 2024

Contents

1 Problem Statement

1.1 Tennis Match Analysis with Service Protocols

In a tennis match between players A and B, each rally that begins with a serve by player A is won by player A with probability pa, and each rally that begins with a serve by player B is won by player A with probability pb. The winner of the rally earns a point and becomes the server of the next rally. Player A serves first.You are tasked with analyzing tennis matches between players A and B based on two different service protocols:

1 **Winner Serves Protocol**: The winner of each rally serves for the next rally.

2 **Alternating Service Protocol**: The service alternates between players A and B after each rally.

Write a function to simulate tennis matches under both service protocols and analyze the outcomes. The function should accept the following parameters:

- $\mathbf{p_a}$**_values**: A list of probabilities of player A winning a rally when serving. Each element represents a different value of pa to be tested.
- $\mathbf{p_b}$: The probability of player A winning a rally when not serving (i.e., player B serving).
- **N**: The number of matches to simulate for each value of pa.

1.2 Function Signature

def tennis_match_analysis_with_graph(pa_values: List[float], pb: float, n: int) -> None: pass

1.3 Constraints

- 0 $\leqq p_a, p_b \leqq$1
- n $\geqq$ 1

1.4 Example taken here

- p_a_values = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
- p_b = one from [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
- n = 1000
- tennis_match_analysis_with_graph(pa_values, pb, n)

2 Approach 1 (Simpler) - Without considering Deuce condition

2.1 Rules of Tennis Game considered

- We play n number of matches as specified in the question.
- Whoever wins **5 rallies in a match first** is declared the winner (0, 15, 30, 40, game). There is no notion of number of sets won here, it is simply a number of rallies that lead to a winner of a match.
- We write the simulation for both the specified protocols

2.2 Algorithm

The algorithm is divided into two subparts.

2.2.1 Match Simulation

- The matches for alternating serves and winner serves are simulated separately.
- Both the algorithms run until either A or B reaches a score of 5 points.
- The winner of each rally is calculated from p_a and p_b using *np.random.choice()*.
- Each match is played n times and number of wins of each player is counted and plotted.

2.2.2 Analytic Calculation

- We start the analysis by calculating the probability of A winning each rally.
- For alternating serves it's straightforward, just p_a and p_b alternatively.
- For winner serves the probability of A winning a rally is calculated as follows :
  - The probability of A winning first rally $P(A_1)$ is set as p_a since A serves first.
  - The second rally $P(A_2)$ has two cases namely, A winning and A losing.
  - If A wins the first round probability of A winning second round becomes p_a.
  - If A loses, the probability becomes p_b.
  - Hence we get the probability of A winning second round as

    $$P(A_2) \;=\; P(A_1) * p_a \;+\; (1 - P(A_1)) * p_b \tag{1}$$

    extending this,

    $$P(A_n) \;=\; P(A_{n-1}) * p_a \;+\; (1 - P(A_{n-1})) * p_b \tag{2}$$

- The round ends after a maximum of 9 rallies.
- The probabilities calculated above are then used to calculate the probability of A winning a match $P(A)$.
- This is calculated as

  $$P(A) = P(X \geq 5) = P(5) + P(6) + P(7) + P(8) + P(9) \tag{3}$$

  where, $P(X)$ denotes the probability of A winning X rallies.
- This $P(A)$ is multiplied by n to get expected number of matches A wins in n matches.

The results are then plotted in seperate graphs to compare them.

2.3 Code

```
import matplotlib.pyplot as plt
import numpy as np
from typing import List


def winner_serves_match(pa, pb):
    player_serving = 'A'  # Player A serves first
    score_A = 0
    score_B = 0

    while score_A < 5 and score_B < 5:
        if player_serving == 'A':
            if not np.random.choice(2,1,p=[pa,1-pa])[0]: #A wins if this outputs 0
                score_A += 1 #same player repeats
            else:
                score_B += 1 #score update
                player_serving = 'B' #swap for the winner
        else:
            if not np.random.choice(2,1,p=[pb,1-pb])[0]:  #B wins if this outputs 1
                score_A += 1
                player_serving = 'A'  #swap for the winner
            else:
                score_B += 1 #score update #same player repeats
    ret = 'A'
    if(score_B>score_A):
        ret='B'
    return ret


def alternating_service_match(pa, pb):
    score_A = 0
    score_B = 0
    player_serving = 'A'  # Player A serves first

    while score_A < 5 and score_B < 5:
        if player_serving == 'A':
            if  not np.random.choice(2,1,p=[pa,1-pa])[0]:
                score_A += 1
            else:
                score_B += 1 #score update
        else:
            if  not np.random.choice(2,1,p=[pb,1-pb])[0]:
                score_A += 1
            else:
                score_B += 1 #score update

```

```python
        if player_serving == 'A':  #swap players
            player_serving = 'B'
        elif player_serving == 'B':
            player_serving = 'A'

    ret = 'A'
    if(score_B>score_A):
        ret='B'
    return ret

def simulate_matches(pa_values, pb, n):
    winner_serves_wins = []
    alternating_service_wins = []

    for pa in pa_values:
        winner_serves_wins_count = 0
        alternating_service_wins_count = 0
        for _ in range(n):
            winner = winner_serves_match(pa, pb)
            if winner == 'A':
                winner_serves_wins_count += 1

            winner = alternating_service_match(pa, pb)
            if winner == 'A':
                alternating_service_wins_count += 1

        winner_serves_wins.append(winner_serves_wins_count)
        alternating_service_wins.append(alternating_service_wins_count)

    return winner_serves_wins, alternating_service_wins

def tennis_match_analysis_with_graph(pa_values: List[float], pb: float, n: int):
    winner_serves_wins, alternating_service_wins = simulate_matches(pa_values, pb, n)

    plt.plot(pa_values, winner_serves_wins, label="Winner Serves Protocol",marker='o')
    plt.plot(pa_values, alternating_service_wins, label="Alternating Service Protocol",marker='o')
    plt.xticks(pa_values)
    plt.xlabel('Probability of A winning when A is the server (pa)')
    plt.ylabel('Number of Wins for A')
    plt.title('Simulation of Tennis for pb={}'.format(pb))
    plt.legend()
    plt.savefig('ig{}.png'.format(pb))
    plt.show()

def dp_simul_winner_serve(pa,pb):
    pw=np.zeros(10);
```

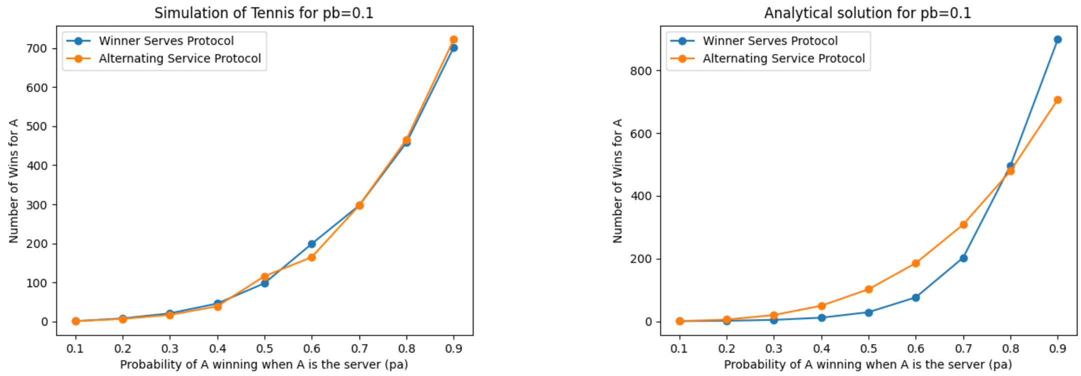
```
    pw[0]=pa
    for round in range(1,10):
        pw[round]=pw[round-1]*pa+(1-pw[round-1])*pb  #dp array initialisation

    p9=0
    for i in range(9): #calculating p5, , A loses 4 matches
        p9t=(1-pw[i])
        for j in range(i,9):
            if(j!=i):
                p9t1=p9t*(1-pw[j])
                for k in range(j,9):
                    if(k!=i and k!=j):
                        p9t2=p9t1*(1-pw[k])
                        for l in range(k,9):
                            if(l!=i and l!=j and l!=k):
                                p9t3=p9t2*(1-pw[l])
                                for m in range(9):
                                    if(m!=i and m!=j and m!=k  and m!=l):
                                        p9t3*=pw[m]
                                p9+=p9t3

    for i in range(9):#calculating p6, A loses 3 matches
        p9t=(1-pw[i])
        for j in range(i,9):
            if(j!=i):
                p9t1=p9t*(1-pw[j])
                for k in range(j,9):
                    if(k!=i and k!=j):
                        p9t2=p9t1*(1-pw[k])
                        for l in range(9):
                            if(l!=i and l!=j and l!=k):
                                p9t2*=(pw[l])
                        p9+=p9t2

    for i in range(9): #calculating p7, A loses 2 matches
        p9t=(1-pw[i])
        for j in range(i,9):
            if(j!=i):
                p9t1=p9t*(1-pw[j])
                for k in range(9):
                    if(k!=i and k!=j):
                        p9t1*=(pw[k])
                p9+=p9t1

    for i in range(9): #calculating p8, A loses only 1 match
        p9t=(1-pw[i])
        for j in range(9):
            if(j!=i):
                p9t*=(pw[j])
```

```
            p9+=p9t

        p9t=1    #calculating p9, A wins all 9 matches
        for i in range(9):
            p9t*=(pw[i])
        p9+=p9t

#Care has been taken to ensure there are no overlaps between the 5
↪   probabilities calculated
    p_final = (p9)
    # print(p_final)
    # print(pw)
    return p_final

def dp_simul_alternate_serve(pa,pb):
    pw=np.zeros(10);
    for round in range(0,10):
        if(round%2):
            pw[round]=pb
        else:
            pw[round]=pa



    p9=0
    for i in range(9):
        p9t=(1-pw[i])
        for j in range(i,9):
            if(j!=i):
                p9t1=p9t*(1-pw[j])
                for k in range(j,9):
                    if(k!=i and k!=j):
                        p9t2=p9t1*(1-pw[k])
                        for l in range(k,9):
                            if(l!=i and l!=j and l!=k):
                                p9t3=p9t2*(1-pw[l])
                                for m in range(9):
                                    if(m!=i and m!=j and m!=k  and m!=l):
                                        p9t3*=pw[m]
                                p9+=p9t3

    for i in range(9):
        p9t=(1-pw[i])
        for j in range(i,9):
            if(j!=i):
                p9t1=p9t*(1-pw[j])
                for k in range(j,9):
                    if(k!=i and k!=j):
                        p9t2=p9t1*(1-pw[k])
                        for l in range(9):
```

```
                            if(l!=i and l!=j and l!=k):
                                p9t2*=(pw[l])
                        p9+=p9t2

    for i in range(9):
        p9t=(1-pw[i])
        for j in range(i,9):
            if(j!=i):
                p9t1=p9t*(1-pw[j])
                for k in range(9):
                    if(k!=i and k!=j):
                        p9t1*=(pw[k])
                p9+=p9t1

    for i in range(9):
        p9t=(1-pw[i])
        for j in range(9):
            if(j!=i):
                p9t*=(pw[j])
        p9+=p9t

    p9t=1
    for i in range(9):
        p9t*=(pw[i])
    p9+=p9t


    p_final = (p9)
    # print(p_final)
    # print(pw)
    return p_final

def final_dp(pa_values,pb,n):
    winner_serves_wins,alternating_service_wins=[],[]
    for pa in pa_values:
        winner_serves_wins.append(dp_simul_winner_serve(pa,pb)*n)
        alternating_service_wins.append(dp_simul_alternate_serve(pa,pb)*n)

    plt.plot(pa_values, winner_serves_wins, label="Winner Serves Protocol",marker='o')
    plt.plot(pa_values, alternating_service_wins, label="Alternating Service Protocol",marker='o')
    plt.xticks(pa_values)
    plt.xlabel('Probability of A winning when A is the server (pa)')
    plt.ylabel('Number of Wins for A')
    plt.title('Analytical solution for pb={}'.format(pb))
    plt.legend()
    plt.savefig('imag{}.png'.format(pb))
    plt.show()

```

Listing 1: Code snippet1-for plotting Simulation and calculated number of wins for A

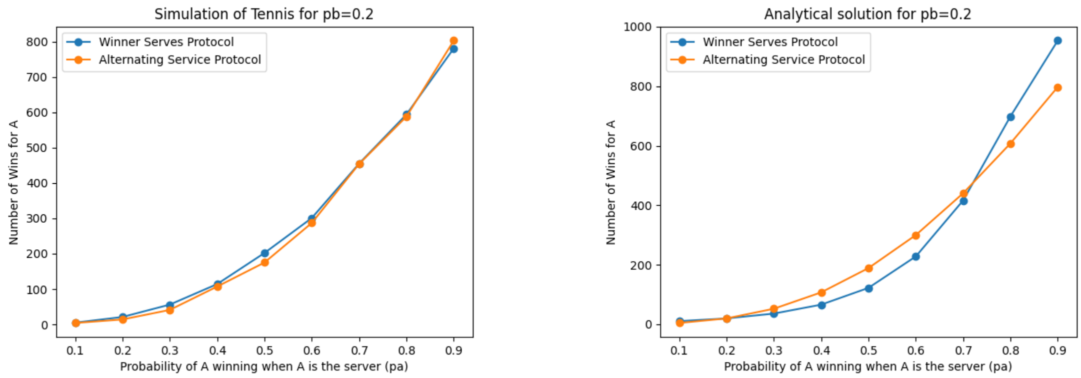
2.4 Results



(a) Simulated wins for A using Random module

(b) Expected number of wins of A as calculated by us

Figure 1: $p_b = 0.1$ for 10 p_a values

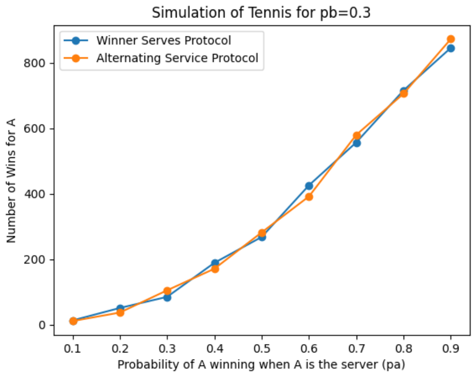


(a) Simulated wins for A using Random module

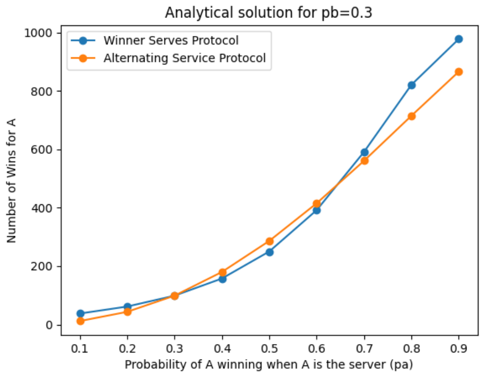
(b) Expected number of wins of A as calculated by us

Figure 2: $p_b = 0.2$ for 10 p_a values

- We see that the our simulation graphs match with the analytical graphs in all cases, especially so for the alternative serving protocol.
- For the p_a=p_b=0.5 case both results give us 500 as the answer, which is intuitively correct.
- The nature of the parabola inverts after p_b=0.5 and we get higher number of wins for lower p_a values. We also get a straight line for p_b=0.5 as expected.

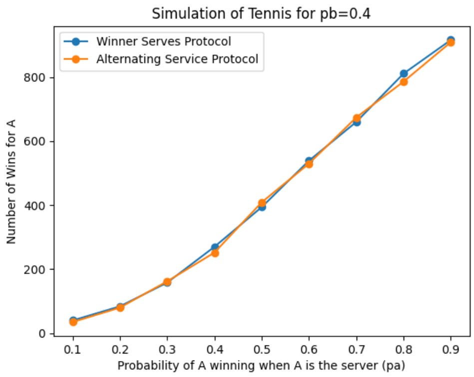


(a) Simulated wins for A using Random module

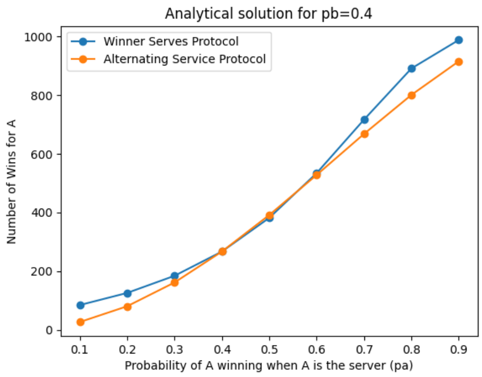


(b) Expected number of wins of A as calculated by us

Figure 3: $p_b = 0.3$ for 10 p_a values

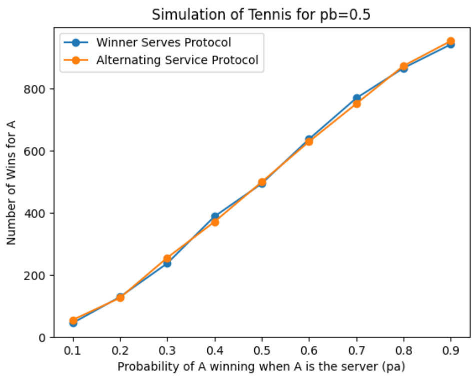


(a) Simulated wins for A using Random module

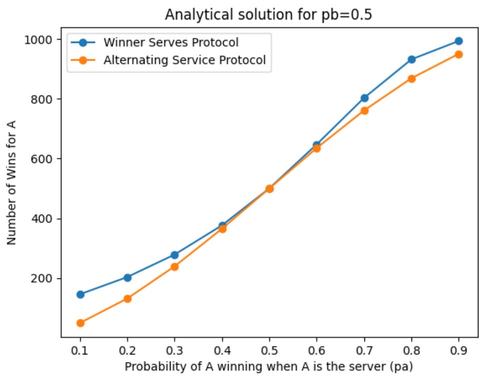


(b) Expected number of wins of A as calculated by us

Figure 4: $p_b = 0.4$ for 10 p_a values

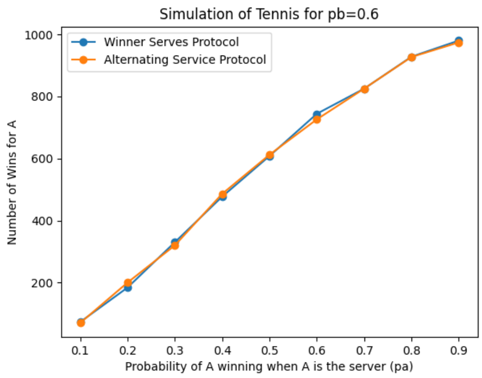


(a) Simulated wins for A using Random module

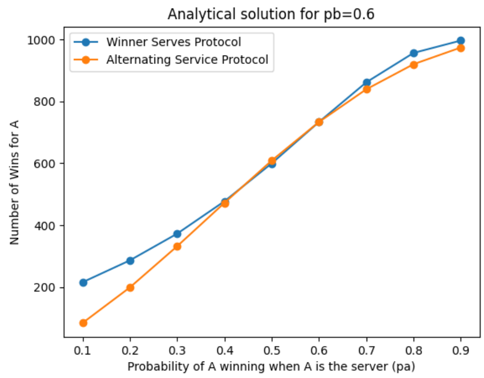


(b) Expected number of wins of A as calculated by us

Figure 5: $p_b = 0.5$ for 10 p_a values

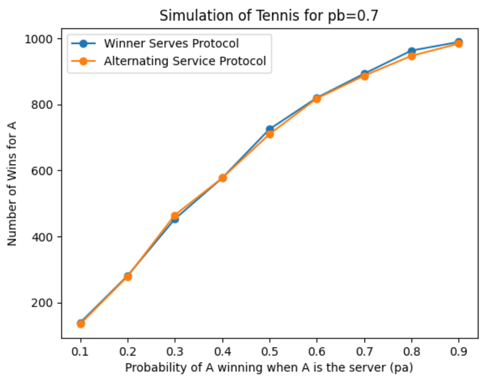


(a) Simulated wins for A using Random module

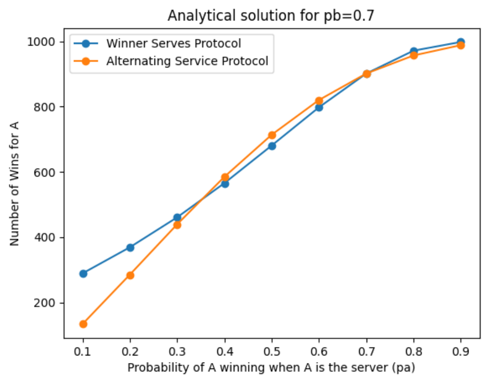


(b) Expected number of wins of A as calculated by us

Figure 6: $p_b = 0.6$ for 10 p_a values

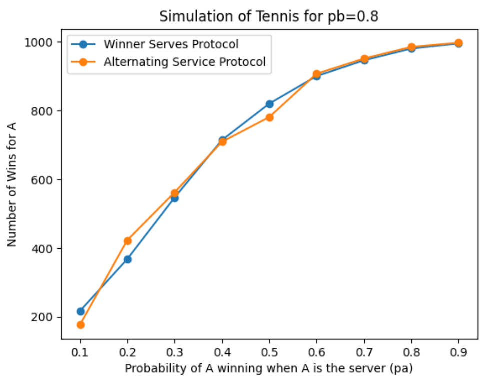


(a) Simulated wins for A using Random module

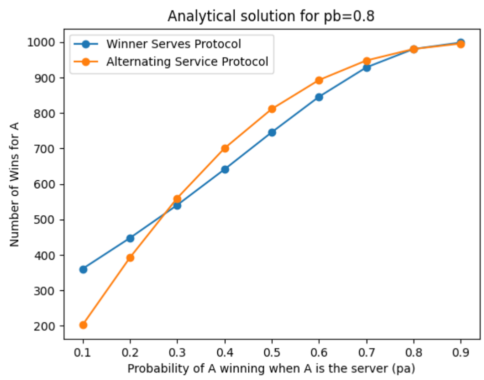


(b) Expected number of wins of A as calculated by us

Figure 7: $p_b = 0.7$ for 10 p_a values

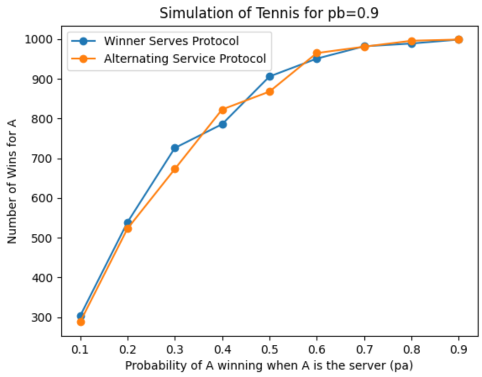


(a) Simulated wins for A using Random module

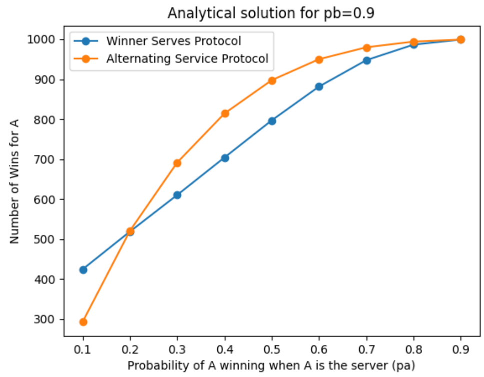


(b) Expected number of wins of A as calculated by us

Figure 8: $p_b = 0.8$ for 10 p_a values

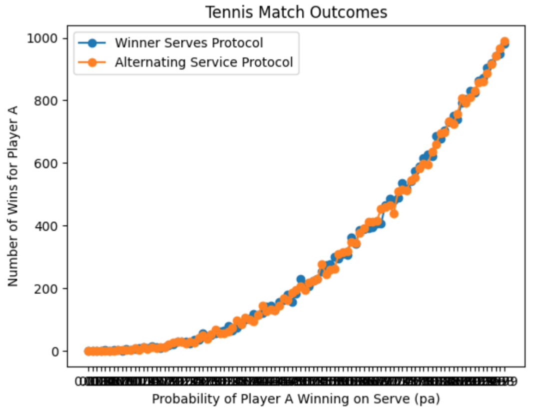


(a) Simulated wins for A using Random module

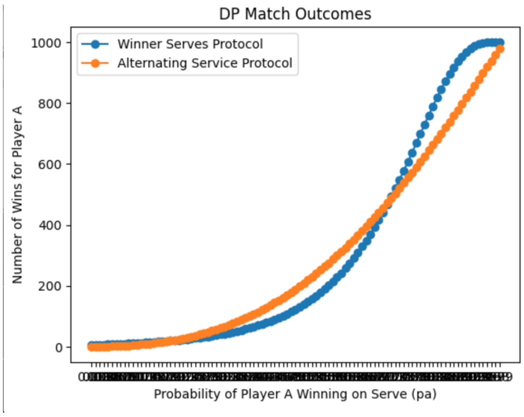


(b) Expected number of wins of A as calculated by us

Figure 9: $p_b = 0.9$ for 10 p_a values

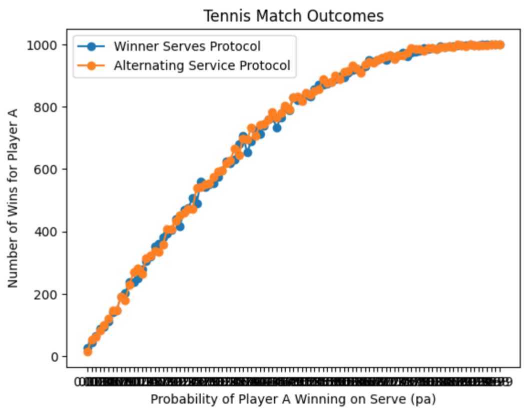


(a) Simulated wins for A using Random module

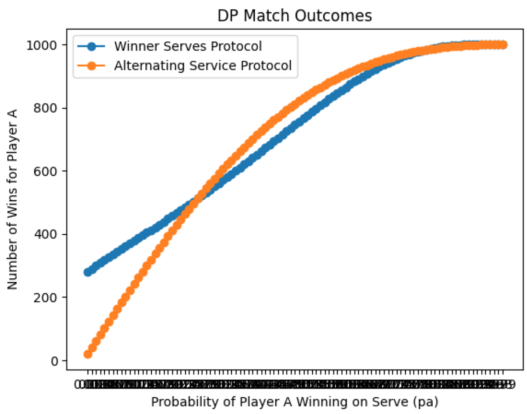


(b) Expected number of wins of A as calculated by us

Figure 10: $p_b = 0.2$ for 100 uniformly distributed p_a values



(a) Simulated wins for A using Random module



(b) Expected number of wins of A as calculated by us

Figure 11: $p_b = 0.8$ for 100 uniformly distributed p_a values. Note that both these parabolas are inverted due to change in pb value. The expected wins graph is clearly the denoised or smooth version of the simulation. This noise can be accounted for by the random module

3 Approach 2 (More Complex) - Considering Deuce condition

3.1 Rules of Tennis Game considered

- We play n number of matches as specified in the question.
- Whoever wins **5 rallies in a match first** is declared the winner (0, 15, 30, 40, game).
- Additionally, when A and B reach the same score (40-40), a situation called the "DEUCE", one of them is expected to establish a two point lead in order to win. This code takes into account this additional complexity.
- We write the simulation for both the specified protocols.

3.2 Algorithm

- The simulation is mostly similar to the simulation for the game without deuces with one crucial difference.
- There is a additional condition which keeps the loop running while the score difference is less than 2.
- To prevent infinite looping the code breaks and outputs the higher scoring player when their scores reach 500, but the probability of this happening is infinitesimally small.
- The analytical solution is presented in the form of a **Markov chain** diagram in section 3.5.

3.3 Code

```
import matplotlib.pyplot as plt
import numpy as np


def winner_serves_match(pa, pb):
    player_serving = 'A'  # Player A serves first
    score_A = 0
    score_B = 0

    while (score_A < 5 and score_B < 5) or abs(score_A - score_B)<2:#the
    ↪ second condition here checks for deuce
        if player_serving == 'A':
            if not np.random.choice(2,1,p=[pa,1-pa])[0]: #we choose 0 with
            ↪ pa,
                score_A += 1 #score update
            else:
                score_B += 1
                player_serving = 'B' #swap for winner
        else:
            if not np.random.choice(2,1,p=[pb,1-pb])[0]:
                score_A += 1
                player_serving = 'A'
            else:
```

```
                score_B += 1 #score update

        ret = 'A'
        if(score_B>score_A):
            ret='B'

        if(score_A>500):
            return ret
    return ret


def alternating_service_match(pa, pb):
    score_A = 0
    score_B = 0
    player_serving = 'A'  # Player A serves first

    while (score_A < 5 and score_B < 5) or abs(score_A-score_B)<2:
        if player_serving == 'A':
            if  not np.random.choice(2,1,p=[pa,1-pa])[0]:
                score_A += 1
            else:
                score_B += 1  #score update
        else:
            if  not np.random.choice(2,1,p=[pb,1-pb])[0]:
                score_A += 1
            else:
                score_B += 1  #score update

        if player_serving == 'A': #Swapping after every match
            player_serving = 'B'
        elif player_serving == 'B':
            player_serving = 'A'
        '''if(score_A>10):
            print("Score A : ",score_A)
            print("Score B : ",score_B)'''
        ret = 'A'
        if(score_B>score_A):
            ret='B'

        if(score_A>500):
            return ret
    return ret

def simulate_matches(pa_values, pb, n):
    winner_serves_wins = []
    alternating_service_wins = []

    for pa in pa_values:
        winner_serves_wins_count = 0
        alternating_service_wins_count = 0
```

```
        for _ in range(n):
            winner = winner_serves_match(pa, pb)
            if winner == 'A':
                winner_serves_wins_count += 1

            winner = alternating_service_match(pa, pb)
            if winner == 'A':
                alternating_service_wins_count += 1
        '''if(score_A>20):
            print("Score A : ",score_A)
            print("Score B : ",score_B)'''
        winner_serves_wins.append(winner_serves_wins_count)
        alternating_service_wins.append(alternating_service_wins_count)

    return winner_serves_wins, alternating_service_wins

def tennis_match_analysis_with_graph(pa_values, pb, n):
    winner_serves_wins, alternating_service_wins =
    ↪  simulate_matches(pa_values, pb, n)

    plt.plot(pa_values, winner_serves_wins, label="Winner Serves
    ↪  Protocol",marker='o')
    plt.plot(pa_values, alternating_service_wins, label="Alternating
    ↪  Service Protocol",marker='o')
    plt.xticks(pa_values)
    plt.xlabel('Probability of A winning when A is the server (pa)')
    plt.ylabel('Number of Wins for A')
    plt.title('Simulation of Tennis for pb={}'.format(pb))
    plt.legend()
    plt.savefig('o{}.png'.format(pb))
    plt.show()

```

Listing 2: Code snippet2-for plotting just the simulations for the Deuce condition.

3.4 Results

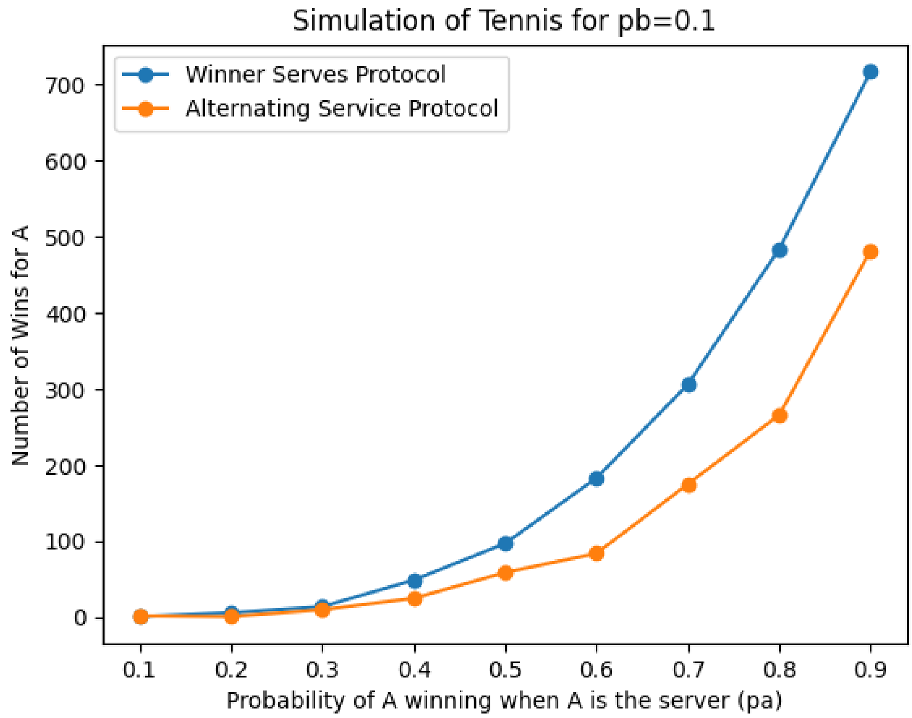


Figure 12: The simulated number of winnings for A, considering Deuce in the matches for pb=0.1

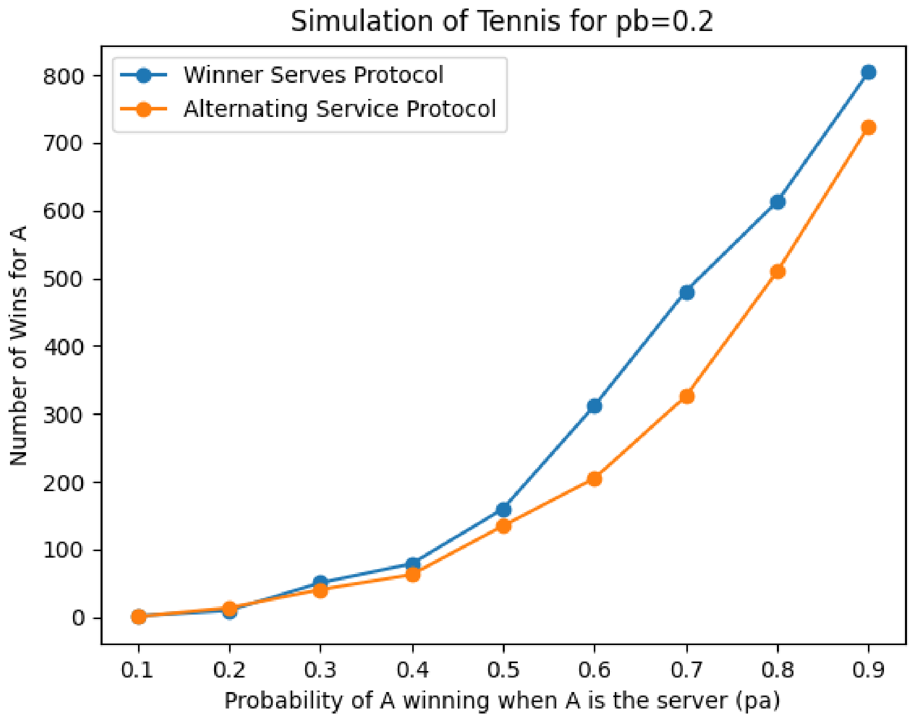


Figure 13: The simulated number of winnings for A, considering Deuce in the matches for pb=0.2

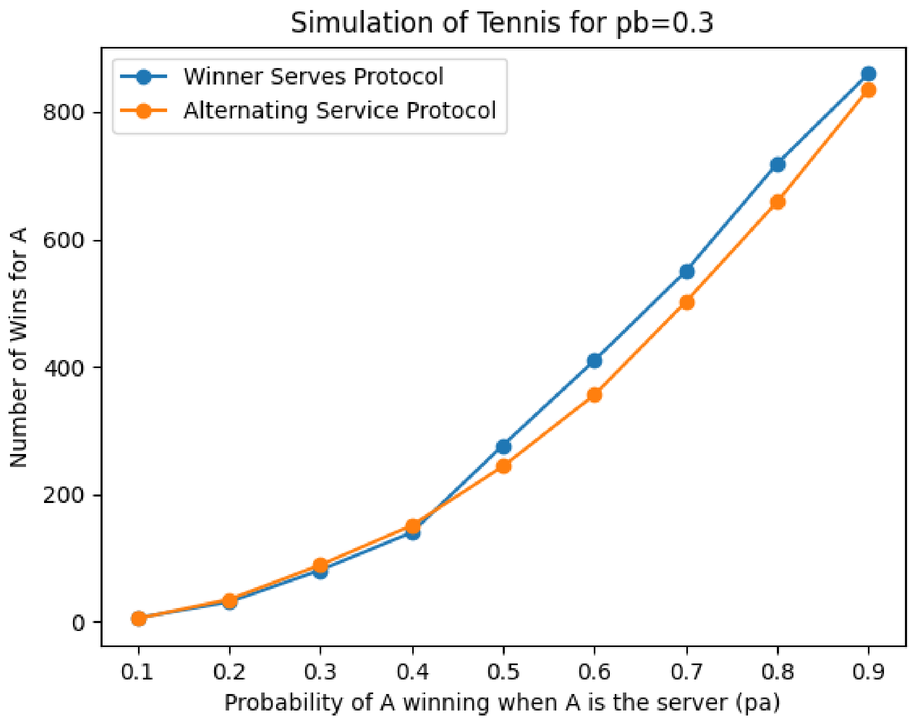


Figure 14: The simulated number of winnings for A, considering Deuce in the matches for pb=0.3

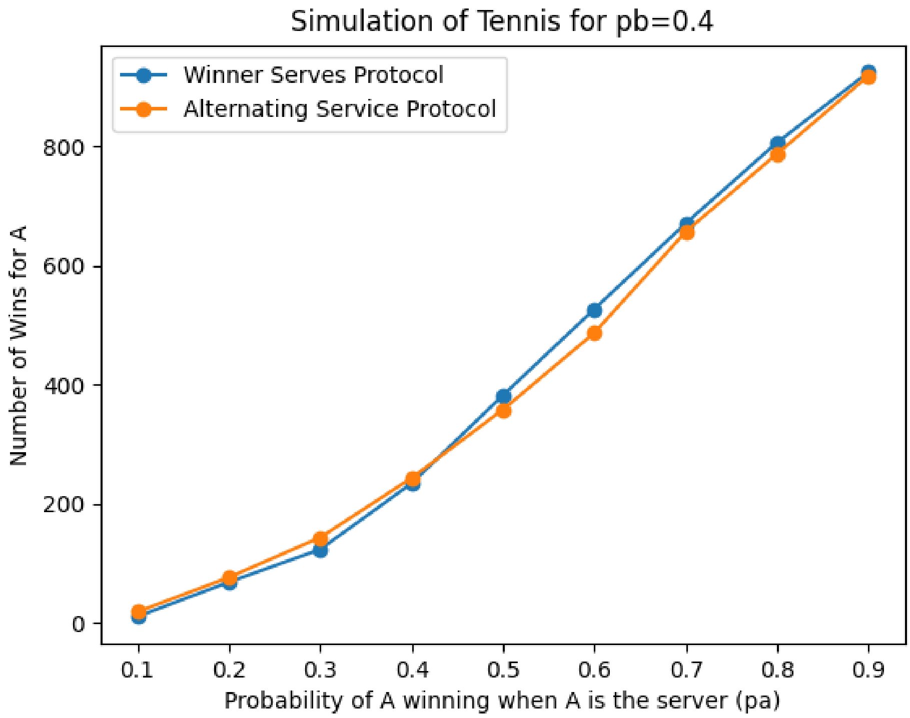


Figure 15: The simulated number of winnings for A, considering Deuce in the matches for pb=0.4

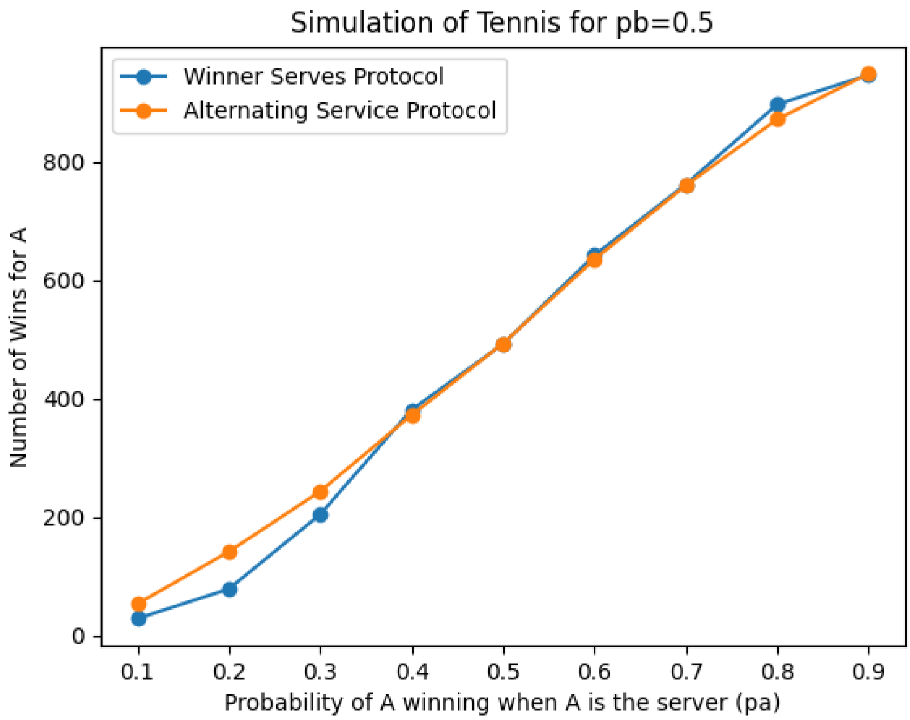


Figure 16: The simulated number of winnings for A, considering Deuce in the matches for pb=0.5

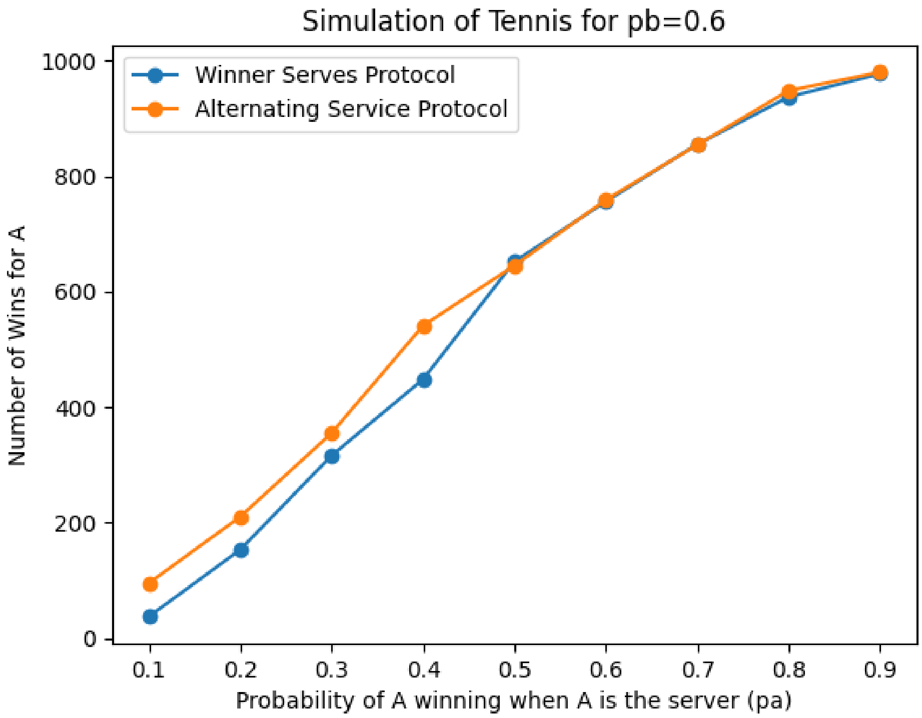


Figure 17: The simulated number of winnings for A, considering Deuce in the matches for pb=0.6

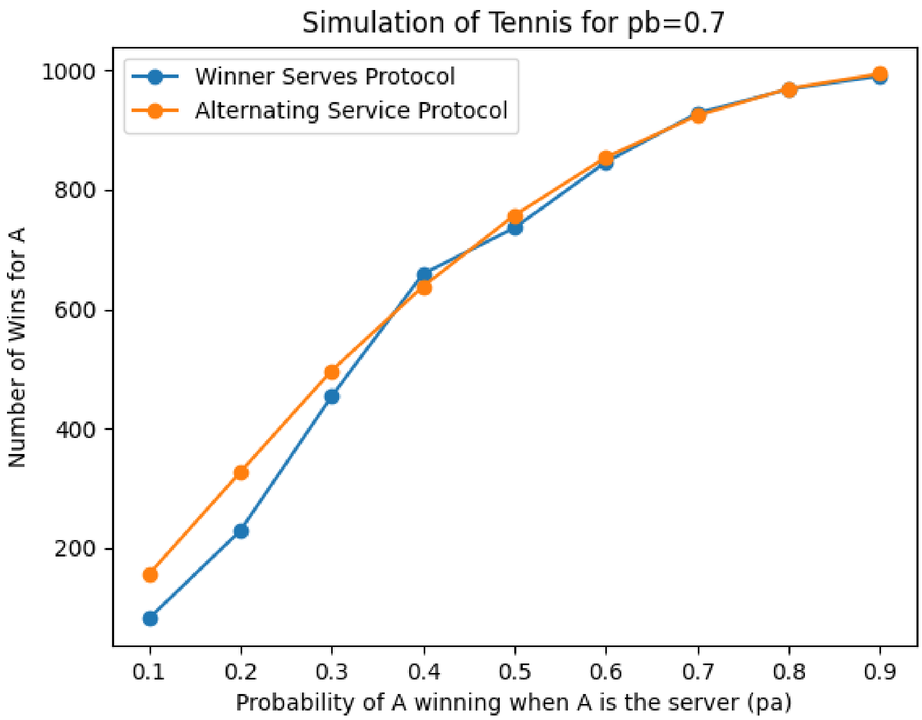


Figure 18: The simulated number of winnings for A, considering Deuce in the matches for pb=0.7

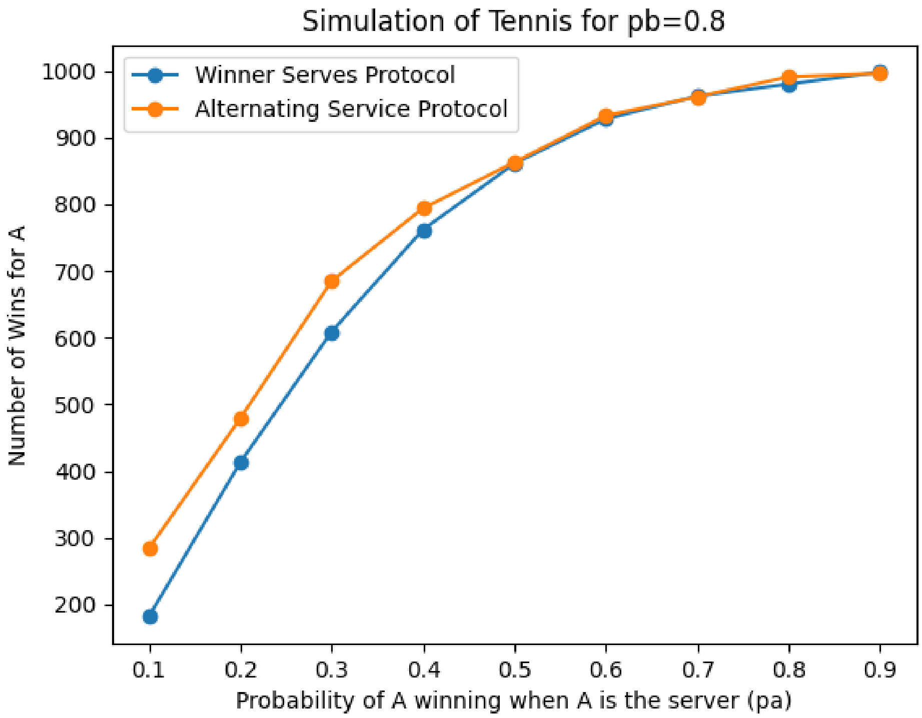


Figure 19: The simulated number of winnings for A, considering Deuce in the matches for pb=0.8

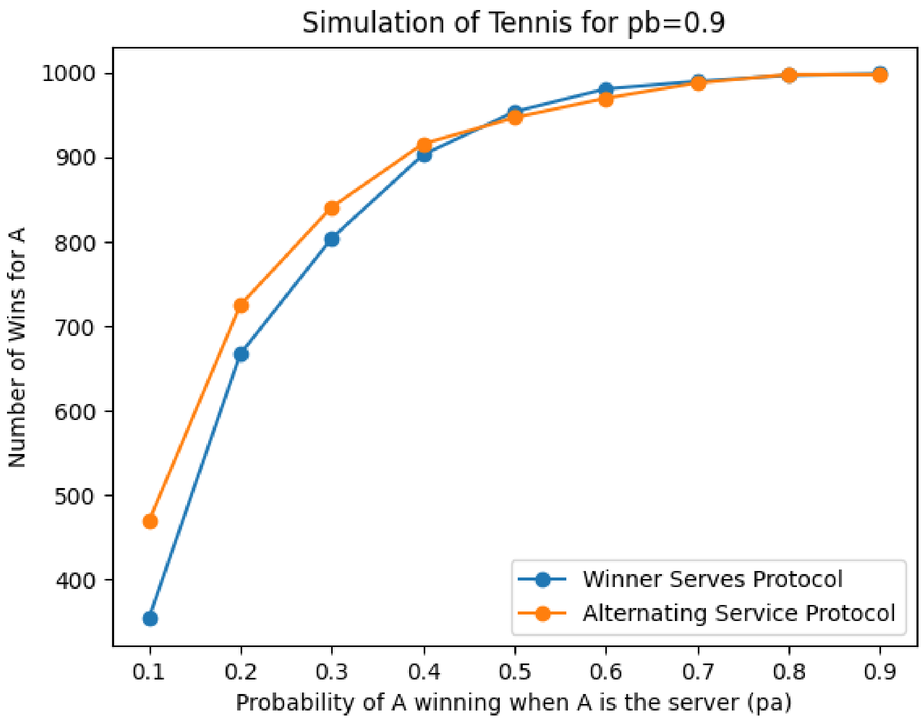


Figure 20: The simulated number of winnings for A, considering Deuce in the matches for pb=0.9

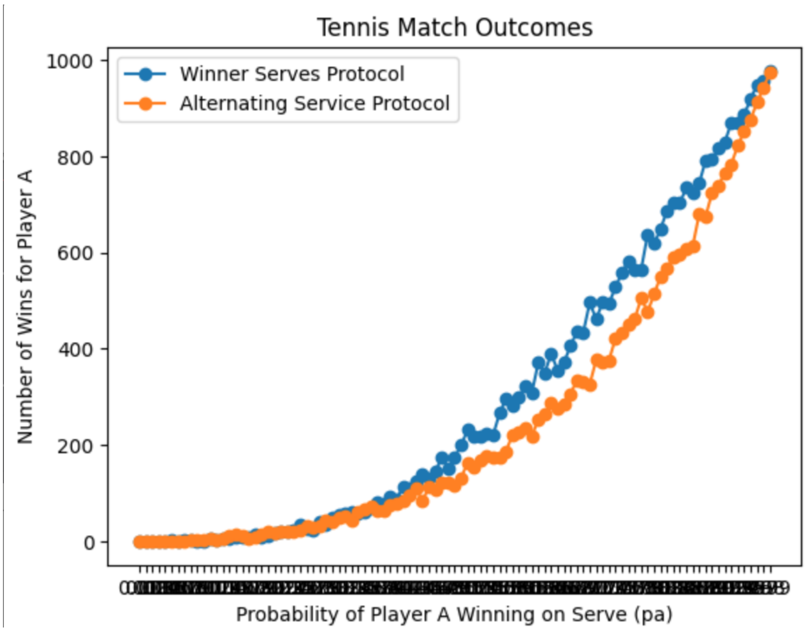


Figure 21: The simulated number of winnings for A, considering Deuce in the matches for pb=0.2, but for 100 uniformly spaced values of pa.

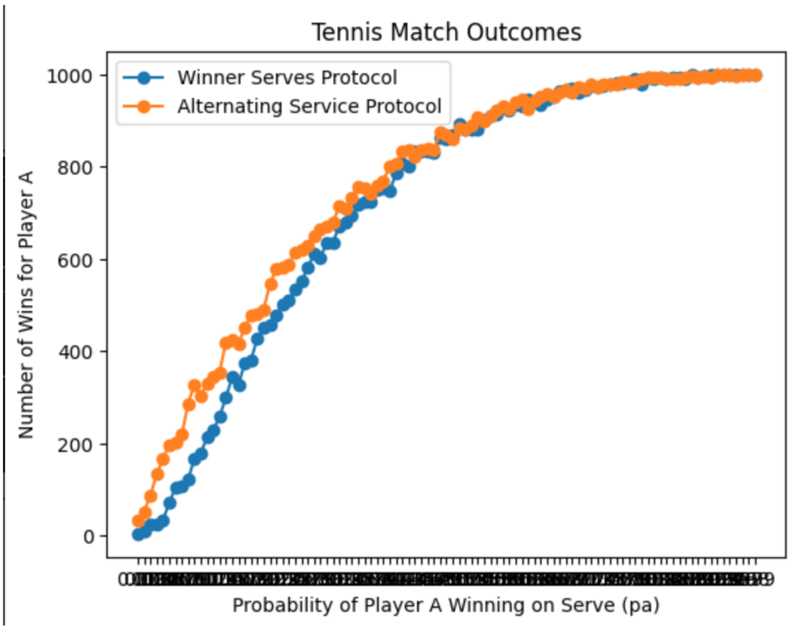


Figure 22: The simulated number of winnings for A, considering Deuce in the matches for pb=0.8, but for 100 uniformly spaced values of pa. Note that the shape of the parabola inverts compared to the previous parabola for pb=0.2

3.5 Analytical Deuce Solution

The analytical solution to the game is presented in the form of a Markov chain diagram in an attempt to form a solution.

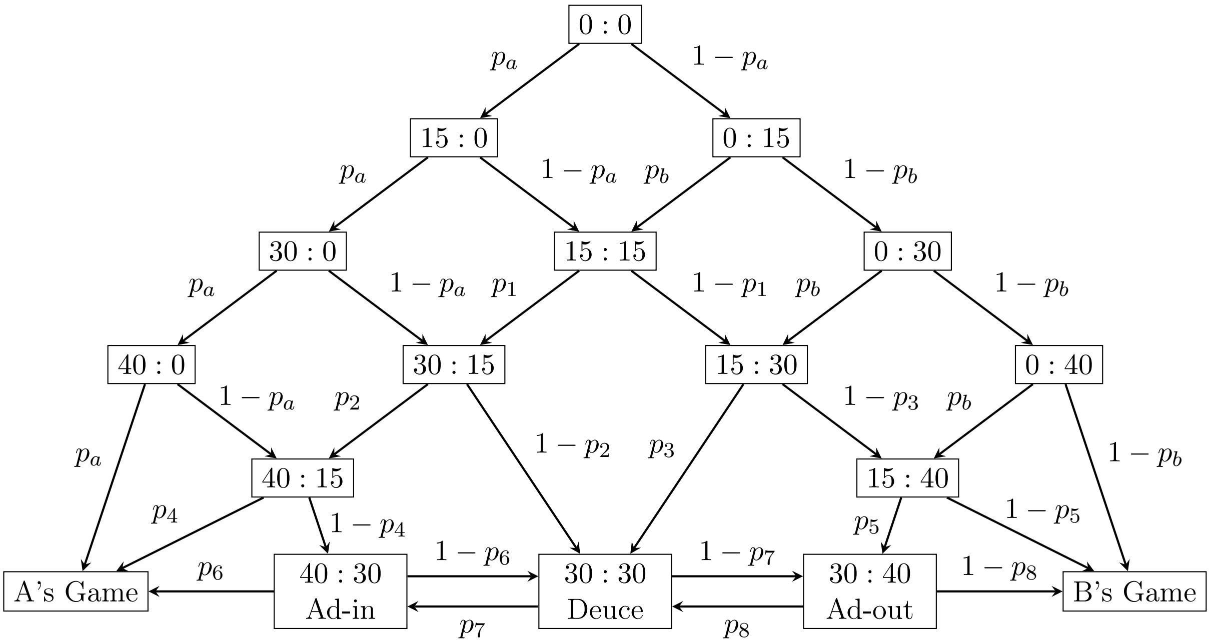


Figure 23: Winner Serves Game

In the figure,

$$p_1 = (1 - p_a) * (p_b) + (p_b) * (p_a) \qquad p_2 = (1 - p_a) * (p_b) + (p_1) * (p_a)$$

$$p_3 = (1 - p_1) * (p_b) + (p_b) * (p_a) \qquad p_4 = (1 - p_a) * (p_b) + (p_2) * (p_a)$$

$$p_5 = (1 - p_3) * (p_b) + (p_b) * (p_a) \qquad p_6 = (1 - p_4) * (p_b) + (p_7) * (p_a)$$

$$p_7 = (1 - p_2) * (p_b) + (p_6) * (p_b) + (p_8) * (p_a) + (p_3) * (p_a)$$

$$p_8 = (1 - p_7) * (p_b) + (p_5) * (p_a)$$

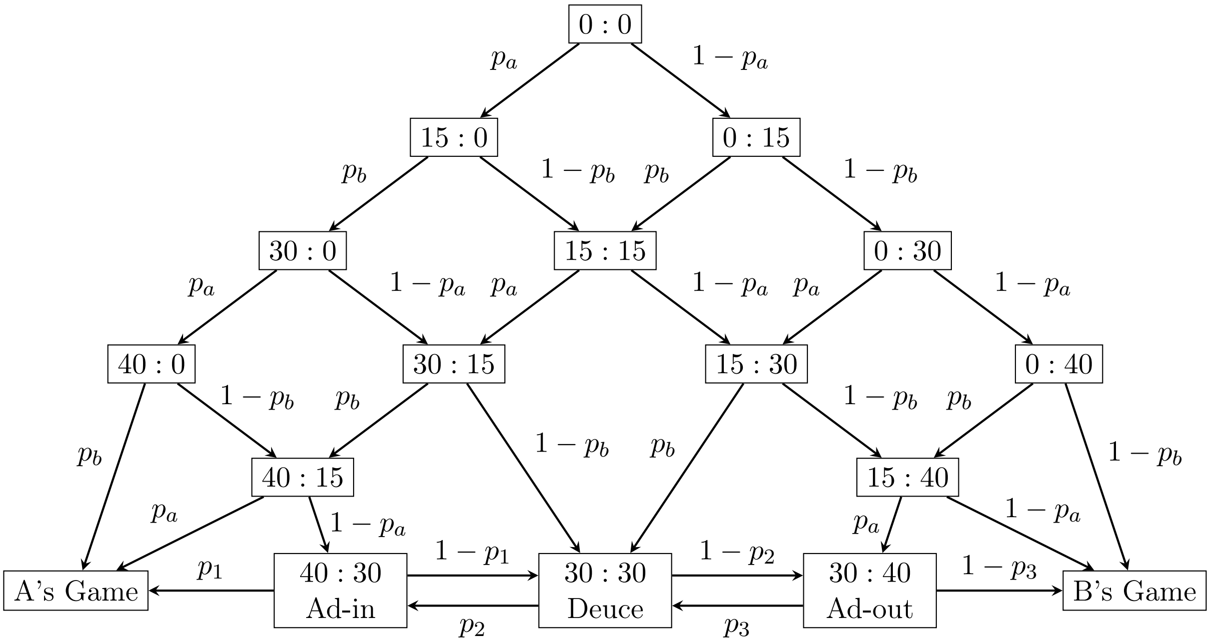


Figure 24: Alternate Serves Game

In the figure,

$$p_1 = (1 - p_a) * (p_b) + (p_2) * (p_b) \qquad p_3 = (1 - p_2) * (p_b) + (p_b) * (p_a)$$

$$p_2 = (1 - p_1) * (p_a) + (1 - p_b) * (p_a) + (p_b) * (p_a) + (p_3) * (p_a)$$

- The above Markov chains can be solved by taking the value at [0:0] to be 1 and applying the transitions consecutively to get the final probability of A winning the game.
- A sample half-solved chain is shown below to illustrate the method to solve them.
- The empty cells can be filled by applying the corresponding transtions to the states and summing them up.
- This helps us find the probability of reaching all of the states in the bottom row of the markov chain. We model a random walk for the 5 states in the bottom of the Markov Chain. We then multiply the probabilities of reaching one of the 4 states (Advantage in, out, Deuce and B's game) and the probability of reaching A's game from each of these 4 states (from the random walk earlier) and add these products to get the final probability of reaching A's game. This calculation gets messy because of the fact that we have different success and failure probabilities at each stage, and ends up with us dealing with messy polynomials. I however tried this for p_a=p_b=0.5 and it matched with the simulation.

Github Link of Assignment: `https://github.com/hizhitman/EE6150-Assignment`

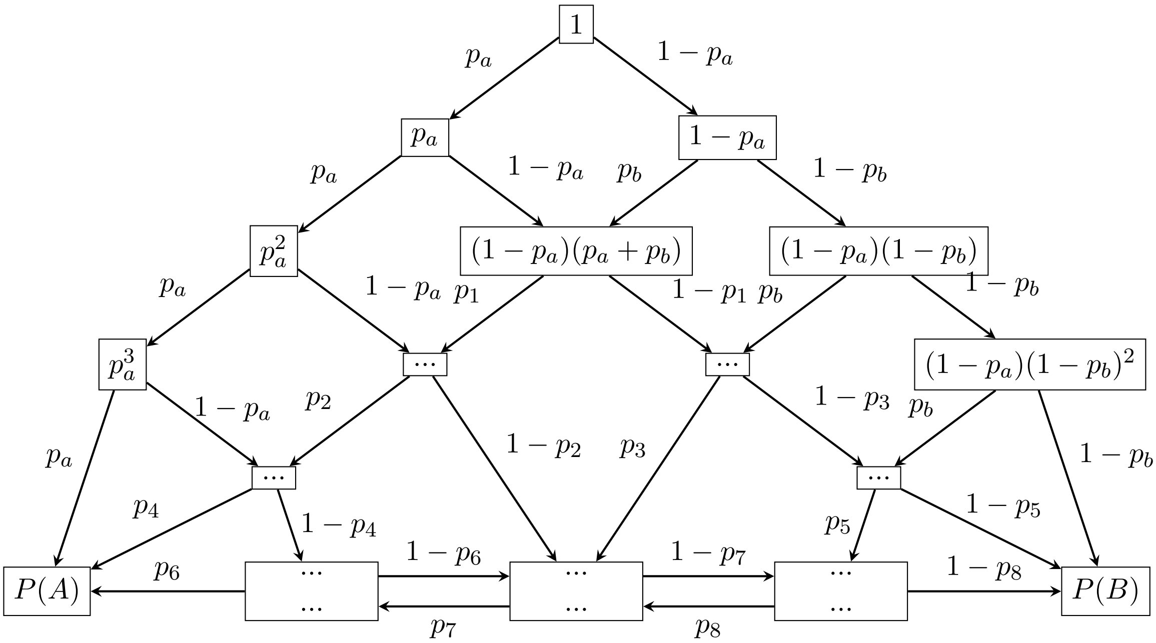


Figure 25: Sample Solution