T2 Modelos estocásticos

Vicente Pareja

May 2023

1 Introducción

En este trabajo, se analiza el movimiento de las piezas en un tablero de ajedrez. Las tres partes planteadas se abordan mediante el cálculo de matrices de transición, simulaciones numéricas y visualizaciones gráficas.

2 Parte 1: Calcular la matriz de transición, elevarla a 10 y evaluar en $(0, 64)$.

2.1 Caballo:

Se calcula la matriz de transición del caballo en un tablero de ajedrez y se eleva 10. Luego, evaluamos el resultado en la casilla $0, 64$. El resultado obtenido es 0.01009.

```
n = 8
m = 8

def knight_transition_matrix(n, m):
    directions = [(1, 2), (2, 1), (-1, 2), (-2, 1),
                  (1, -2), (2, -1), (-1, -2), (-2, -1)]

    matrix = np.zeros((n * m, n * m))

    for i in range(n):
        for j in range(m):
            current = i * m + j
            possible_moves = 0

            for dx, dy in directions:
                x, y = i + dx, j + dy
                if 0 <= x < n and 0 <= y < m:
                    possible_moves += 1
```

```
            for dx, dy in directions:
                x, y = i + dx, j + dy
                if 0 <= x < n and 0 <= y < m:
                    neighbor = x * m + y
                    # No olvidar divir por la cantidad de movimientos
                    matrix[current, neighbor] = 1 / possible_moves

    return matrix

knight_matrix = knight_transition_matrix(n, m)
P_power_n = matrix_power_n(knight_matrix, num_steps)
print_row_as_board(P_power_n[nodo_inicio], n, m)
```

Esto significa que habiendo comenzado en el nodo 0 (casilla (1,1)) luego de 10 movimientos aleatorios, el caballo tiene un 1.009% de terminar en el nodo 64 (casilla (8,8)).

2.2 Alfil:

Se calcula la matriz de transición del alfil en un tablero de ajedrez y la elevamos a la potencia de 10. Luego, evaluamos el resultado en la casilla $0, 64$. El resultado obtenido es 0.02662.

```
n = 8
m = 8

def bishop_transition_matrix(n, m):
    matrix = np.zeros((n * m, n * m))

    for i in range(n):
        for j in range(m):
            current = i * m + j
            possible_moves = 0

            # Contar movimientos posibles
            for x in range(n):
                for y in range(m):
                    if abs(x - i) == abs(y - j) and (x, y) != (i, j):
                        possible_moves += 1

            # Llenar la matriz con las probabilidades
            for x in range(n):
                for y in range(m):
                    if abs(x - i) == abs(y - j) and (x, y) != (i, j):
```

```
                        neighbor = x * m + y
                        matrix[current, neighbor] = 1 / possible_moves

    return matrix

bishop_matrix = bishop_transition_matrix(n, m)
P_power_n = matrix_power_n(bishop_matrix, num_steps)
print_row_as_board(P_power_n[nodo_inicio], n, m)
```

De manera equivalente, ello impla que comenzando en el nodo 0 (casilla (1,1)) luego de 10 movimientos aleatorios, el alfil tiene un 2.662% de terminar en el nodo 64 (casilla (8,8)).

2.3 Torre:

Se calcula la matriz de transición de la torre en un tablero de ajedrez y la elevamos a la potencia de 10. Luego, evaluamos el resultado en la casilla $0, 64$. El resultado obtenido es 0.01562.

```
n = 8
m = 8

def rook_transition_matrix(n, m):
    matrix = np.zeros((n * m, n * m))

    for i in range(n):
        for j in range(m):
            current = i * m + j
            # movimientos verticales y horizontales
            possible_moves = (n - 1) + (m - 1)

            for x in range(n):
                if x != i:
                    neighbor = x * m + j
                    # Se divide por la cantidad de posibilidades
                    matrix[current, neighbor] = 1 / possible_moves

            for y in range(m):
                if y != j:
                    neighbor = i * m + y
                    # Se divide por la cantidad de posibilidades
                    matrix[current, neighbor] = 1 / possible_moves

    return matrix
```

```
rook_matrix = rook_transition_matrix(n, m)
P_power_n = matrix_power_n(rook_matrix, num_steps)
print_row_as_board(P_power_n[nodo_inicio], n, m)
```

De manera equivalente, ello impla que comenzando en el nodo 0 (casilla (1,1)) luego de 10 movimientos aleatorios, la torre tiene un 1.562% de terminar en el nodo 64 (casilla (8,8)).

2.4 Reina:

Calculamos la matriz de transición de la reina en un tablero de ajedrez y la elevamos a la potencia de 10. Luego, evaluamos el resultado en la casilla $0, 64$. El resultado obtenido es 0.01442.

```
n = 8
m = 8

def queen_transition_matrix(n, m):
    queen_matrix = np.zeros((n * m, n * m))

    for i in range(n):
        for j in range(m):
            current_position = i * m + j
            possible_moves = 0

            # Contar movimientos verticales y horizontales
            for k in range(n):
                if k != i:
                    possible_moves += 1

            for l1 in range(m):
                if l1 != j:
                    possible_moves += 1

            # Contar movimientos diagonales
            for k in range(n):
                for l1 in range(m):
                    if k != i and l1 != j and abs(k - i) == abs(l1 - j):
                        possible_moves += 1

            # Llenar la matriz con las probabilidades
            for k in range(n):
                if k != i:
                    new_position = k * m + j
```

```
                queen_matrix[current_position,
                             new_position] = 1 / possible_moves

            for l1 in range(m):
                if l1 != j:
                    new_position = i * m + l1
                    queen_matrix[current_position,
                                 new_position] = 1 / possible_moves

            for k in range(n):
                for l1 in range(m):
                    if k != i and l1 != j and abs(k - i) == abs(l1 - j):
                        new_position = k * m + l1
                        queen_matrix[current_position,
                                     new_position] = 1 / possible_moves

    return queen_matrix

queen_matrix = queen_transition_matrix(n, m)
P_power_n = matrix_power_n(queen_matrix, num_steps)
print_row_as_board(P_power_n[nodo_inicio], n, m)
```

De manera equivalente, ello implica que comenzando en el nodo 0 (casilla (1,1)) luego de 10 movimientos aleatorios, la reina tiene un 1.442% de terminar en el nodo 64 (casilla (8,8)).

2.5 Rey:

Se calcula la matriz de transición del rey en un tablero de ajedrez y la elevamos a la potencia de 10. Luego, evaluamos el resultado en la casilla $0, 64$. El resultado obtenido es 0.00010.

```
n = 8
m = 8

def king_transition_matrix(n, m):
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1),
                  (1, 1), (1, -1), (-1, 1), (-1, -1)]

    matrix = np.zeros((n * m, n * m))

    for i in range(n):
        for j in range(m):
            current = i * m + j
```

```
            possible_moves = 0

            # Contar movimientos posibles
            for dx, dy in directions:
                x, y = i + dx, j + dy
                if 0 <= x < n and 0 <= y < m:
                    possible_moves += 1

            # Llenar la matriz con las probabilidades
            for dx, dy in directions:
                x, y = i + dx, j + dy
                if 0 <= x < n and 0 <= y < m:
                    neighbor = x * m + y
                    matrix[current, neighbor] = 1 / possible_moves

    return matrix

king_matrix = king_transition_matrix(n, m)
P_power_n = matrix_power_n(king_matrix, num_steps)
print_row_as_board(P_power_n[nodo_inicio], n, m)
```

De manera equivalente, ello implica que comenzando en el nodo 0 (casilla (1,1)) luego de 10 movimientos aleatorios, el caballo tiene un 0.010% de terminar en el nodo 64 (casilla (8,8)).

3 Parte 2: Realizar una simulación numérica para calcular la probabilidad de que la pieza termine en la casilla 64 habiendo comenzado en la 1.

3.1 Caballo:

Utilizamos una simulación numérica con 1000000 de casos para calcular la probabilidad de que el caballo, después de 10 movimientos, esté en la casilla 64 habiendo partido de la casilla 1. Al sacar el promedio de 3 de estas simulaciones la probabilidad obtenida es: 0.01015 o de un 1.015%. Otorgando un error del $0,59\%$ respecto al velor teórico obtenido en la parte 1.

```
class nodo:

# Resto de la clase

def movimientos_caballo(self):
```

```
        saltos = []
        x = self.x
        y = self.y

        if x - 2 >= 0 and y - 1 >= 0:
            saltos.append((x - 2, y - 1))

        if x - 2 >= 0 and y + 1 < self.tamañoy:
            saltos.append((x - 2, y + 1))

        if x - 1 >= 0 and y - 2 >= 0:
            saltos.append((x - 1, y - 2))

        if x - 1 >= 0 and y + 2 < self.tamañoy:
            saltos.append((x - 1, y + 2))

        if x + 1 < self.tamañox and y - 2 >= 0:
            saltos.append((x + 1, y - 2))

        if x + 1 < self.tamañox and y + 2 < self.tamañoy:
            saltos.append((x + 1, y + 2))

        if x + 2 < self.tamañox and y - 1 >= 0:
            saltos.append((x + 2, y - 1))

        if x + 2 < self.tamañox and y + 1 < self.tamañoy:
            saltos.append((x + 2, y + 1))

        return saltos

def simular_camino_caballo(self):
        x0 = self.casilla_inicial[0]
        y0 = self.casilla_inicial[1]
        nodo_actual = self.tablero.tablero[y0][x0]

        for i in range(self.T):
            nodo_actual.veces_pisado = nodo_actual.veces_pisado + 1
            salto = nodo_actual.mover_caballo()
            x = salto[0]
            y = salto[1]
            nodo_actual = self.tablero.tablero[y][x]

        return nodo_actual

    def probabilidad_camino(cantidad_caminos, casilla_inicial, casilla_final, cantidad_turno
    tablero = Tablero(8, 8)
```

```
    camino = Camino(tablero, casilla_inicial, cantidad_turnos)

    favorables = 0

    if pieza.rstrip() == "caballo":
        for i in range(cantidad_caminos):
            final = camino.simular_camino_caballo()
            xf = final.x
            yf = final.y

            if (xf, yf) == casilla_final:
                favorables += 1

        return favorables / cantidad_caminos

if __name__ == '__main__':
    p = probabilidad_camino(100000, (0, 0), (7, 7), 10, "caballo")
    p2 = probabilidad_camino(100000, (0, 0), (7, 7), 10, "caballo")
    p3 = probabilidad_camino(100000, (0, 0), (7, 7), 10, "caballo")

    print(f"La probabilidad es : {(p + p2 + p3)/3}")
```

3.2 Alfil:

Al realizar el mismo procedimiento, se obtuvo una probabilidad de 0.02669 lo cuál genera una discrepancia del 0.263% con el valor teórico obtenido en la parte 1.

```
class nodo
# resto de la clase

    def movimientos_alfil(self):
        movimientos = []
        x = self.x
        y = self.y

        # Movimientos en diagonal hacia arriba-izquierda
        i, j = x - 1, y - 1
        while i >= 0 and j >= 0:
            movimientos.append((i, j))
            i -= 1
            j -= 1

        # Movimientos en diagonal hacia abajo-izquierda
        i, j = x - 1, y + 1
        while i >= 0 and j < self.tamañoy:
```

```
            movimientos.append((i, j))
            i -= 1
            j += 1

        # Movimientos en diagonal hacia arriba-derecha
        i, j = x + 1, y - 1
        while i < self.tamañox and j >= 0:
            movimientos.append((i, j))
            i += 1
            j -= 1

        # Movimientos en diagonal hacia abajo-derecha
        i, j = x + 1, y + 1
        while i < self.tamañox and j < self.tamañoy:
            movimientos.append((i, j))
            i += 1
            j += 1

        return movimientos

def probabilidad_camino(cantidad_caminos, casilla_inicial, casilla_final, cantidad_turnos, p
    tablero = Tablero(8, 8)
    camino = Camino(tablero, casilla_inicial, cantidad_turnos)

    favorables = 0

    elif pieza.rstrip() == "alfil":
            for i in range(cantidad_caminos):
                final = camino.simular_camino_alfil()
                xf = final.x
                yf = final.y

                if (xf, yf) == casilla_final:
                    favorables += 1

            return favorables / cantidad_caminos

    if __name__ == '__main__':
    p = probabilidad_camino(1000000, (0, 0), (7, 7), 10, "alfil")
    p2 = probabilidad_camino(1000000, (0, 0), (7, 7), 10, "alfil")
    p3 = probabilidad_camino(1000000, (0, 0), (7, 7), 10, "alfil")

    print(f"La probabilidad es : {(p + p2 + p3)/3}")
```

3.3 Torre:

Al realizar el mismo procedimiento, se obtuvo una probabilidad de 0.01564 lo cuál genera una discrepancia del 0.12% con el valor teórico obtenido en la parte 1.

```
class nodo:
    # Resto del código

    def movimientos_torre(self):
    movimientos = []
    x = self.x
    y = self.y

    # Movimientos horizontales
    for i in range(self.tamañox):
        if i != x:
            movimientos.append((i, y))

    # Movimientos verticales
    for j in range(self.tamañoy):
        if j != y:
            movimientos.append((x, j))

    return movimientos

def probabilidad_camino(cantidad_caminos, casilla_inicial, casilla_final, cantidad_turnos, p
    tablero = Tablero(8, 8)
    camino = Camino(tablero, casilla_inicial, cantidad_turnos)

    favorables = 0

    elif pieza.rstrip() == "torre":
        for i in range(cantidad_caminos):
            final = camino.simular_camino_torre()
            xf = final.x
            yf = final.y

            if (xf, yf) == casilla_final:
                favorables += 1

        return favorables / cantidad_caminos

if __name__ == '__main__':
    p = probabilidad_camino(1000000, (0, 0), (7, 7), 10, "torre")
    p2 = probabilidad_camino(1000000, (0, 0), (7, 7), 10, "torre")
```

```
    p3 = probabilidad_camino(1000000, (0, 0), (7, 7), 10, "torre")

    print(f"La probabilidad es : {(p + p2 + p3)/3}")
```

3.4 Reina:

Al realizar el mismo procedimiento, se obtuvo una probabilidad de 0.014464 lo cuál genera una discrepancia del 0.277% con el valor teórico obtenido en la parte 1.

```
class nodo:

    #resto de la clase

    def movimientos_reina(self):
        movimientos = []
        x = self.x
        y = self.y

        # Movimientos horizontales
        for i in range(self.tamañox):
            if i != x:
                movimientos.append((i, y))

        # Movimientos verticales
        for j in range(self.tamañoy):
            if j != y:
                movimientos.append((x, j))

        # Movimientos en diagonal hacia arriba-izquierda
        i, j = x - 1, y - 1
        while i >= 0 and j >= 0:
            movimientos.append((i, j))
            i -= 1
            j -= 1

        # Movimientos en diagonal hacia abajo-izquierda
        i, j = x - 1, y + 1
        while i >= 0 and j < self.tamañoy:
            movimientos.append((i, j))
            i -= 1
            j += 1
```

```
        # Movimientos en diagonal hacia arriba-derecha
        i, j = x + 1, y - 1
        while i < self.tamañox and j >= 0:
            movimientos.append((i, j))
            i += 1
            j -= 1

        # Movimientos en diagonal hacia abajo-derecha
        i, j = x + 1, y + 1
        while i < self.tamañox and j < self.tamañoy:
            movimientos.append((i, j))
            i += 1
            j += 1

        return movimientos

def probabilidad_camino(cantidad_caminos, casilla_inicial, casilla_final, cantidad_turnos, p
    tablero = Tablero(8, 8)
    camino = Camino(tablero, casilla_inicial, cantidad_turnos)

    favorables = 0


    elif pieza.rstrip() == "reina":
        for i in range(cantidad_caminos):
            final = camino.simular_camino_reina()
            xf = final.x
            yf = final.y

            if (xf, yf) == casilla_final:
                favorables += 1

        return favorables / cantidad_caminos


    if __name__ == '__main__':
    p = probabilidad_camino(1000000, (0, 0), (7, 7), 10, "reina")
    p2 = probabilidad_camino(1000000, (0, 0), (7, 7), 10, "reina")
    p3 = probabilidad_camino(1000000, (0, 0), (7, 7), 10, "reina")

    print(f"La probabilidad es : {(p + p2 + p3)/3}")
```

3.5 Rey:

Al realizar el mismo procedimiento, se obtuvo una probabilidad de 0.0001 lo cuál genera una discrepancia del 0% con el valor teórico (Obtenido en la parte 1).

```
class nodo:

    #Resto de la clase
    def movimientos_rey(self):
        movimientos = []
        x = self.x
        y = self.y

        # Las direcciones posibles del rey: arriba, abajo, izquierda, derecha y diagonales
        direcciones = [
            (-1, -1), (-1, 0), (-1, 1),
            (0, -1),           (0, 1),
            (1, -1),  (1, 0),  (1, 1),
        ]

        for dx, dy in direcciones:
            i, j = x + dx, y + dy
            if 0 <= i < self.tamañox and 0 <= j < self.tamañoy:
                movimientos.append((i, j))

        return movimientos

def probabilidad_camino(cantidad_caminos, casilla_inicial, casilla_final, cantidad_turnos, p
    tablero = Tablero(8, 8)
    camino = Camino(tablero, casilla_inicial, cantidad_turnos)

    favorables = 0

    elif pieza.rstrip() == "rey":
        for i in range(cantidad_caminos):
            final = camino.simular_camino_rey()
            xf = final.x
            yf = final.y

            if (xf, yf) == casilla_final:
                favorables += 1

        return favorables / cantidad_caminos

    if __name__ == '__main__':
```

```
p = probabilidad_camino(1000000, (0, 0), (7, 7), 10, "rey")
p2 = probabilidad_camino(1000000, (0, 0), (7, 7), 10, "rey")
p3 = probabilidad_camino(1000000, (0, 0), (7, 7), 10, "rey")

print(f"La probabilidad es : {(p + p2 + p3)/3}")
```

4 Parte 3: Realizar una simulación numérica de un camino en el cuál la pieza salta de forma aleatoria muchas veces (Se calculó para 1 millón de saltos). Luego, se grafica un mapa de calor de las casillas más visitadas.

4.1 Caballo:

A continuación, se presenta un mapa de calor que muestra las casillas donde el caballo pasa más veces durante su simulación de camino.

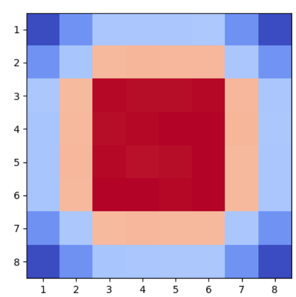

Figure 1: Mapa de calor de las casillas más visitadas por el caballo.

Al contrastarlo con su matriz de probabilidad elevada a un número grande (100000) y evaluar la primera fila (Qué expresa las probabilidades del nodo de llegada habiendo comenzado en el nodo 1 después de 100000 saltos) se obtiene este mapa:

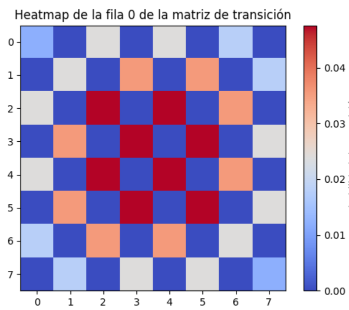

Figure 2: Probabilidad caballo

Estos dos mapas de calor deberías ser iguales. Pero son radicalmente distintos. Esto ocurre debido a que en las casillas pares el caballo siempre está en un color y en las impares en el otro.

Nótese observar esto al evaluar la matriz de probabilidad den la segunda fila:

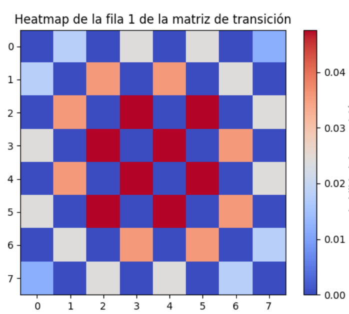

Figure 3: Probabilidad caballo

Para lograr el verdadero valor teórico es necesario promediar cualquier par de filas, donde una es una casilla de inicio negra y la otra blanca. Aquí el resultado:

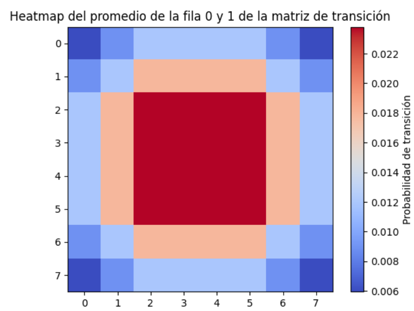

Figure 4: Mapa de calor de la matriz de transición promedio desde la casilla 0 y la 1.

Es indistingible con el mapa de calor obtenido con la simluación numérica.

Código de la implementación:

```
def crear_tablero_a_pintar(tablero):
    tablero_a_pintar = []

    for fila in tablero.tablero:
        fila_a_pintar = []
        for celda in fila:
            num = celda.veces_pisado
            fila_a_pintar.append(num)
        tablero_a_pintar.append(fila_a_pintar)
    return tablero_a_pintar


def escalar_a_rango_01(lista_de_listas):
    # Convertir la lista de listas en un numpy array
    array_original = np.array(lista_de_listas)

    # Encontrar los valores mínimo y máximo del array
    min_val = np.min(array_original)
    max_val = np.max(array_original)

    # Escalar los valores al rango [0, 1]
    array_escalado = (array_original - min_val) / (max_val - min_val)

    return array_escalado
```

```
def crear_mapa_calor(tablero):
    fig, ax = plt.subplots()
    im = ax.imshow(tablero, cmap='coolwarm', interpolation='nearest')

    # Personalizar el gráfico
    ax.set_xticks(np.arange(tablero.shape[1]))
    ax.set_yticks(np.arange(tablero.shape[0]))
    ax.set_xticklabels(range(1, tablero.shape[1] + 1))
    ax.set_yticklabels(range(1, tablero.shape[0] + 1))

    plt.show()

def main(inicial, n):
    tablero = Tablero(8, 8)
    camino = Camino(tablero, inicial, n)
    camino.simular_camino_caballo()
    tablero_a_pintar = crear_tablero_a_pintar(tablero)
    array_escalado = escalar_a_rango_01(tablero_a_pintar)
    crear_mapa_calor(array_escalado)

if __name__ == '__main__':

    # Ver mapa de calor de simulación de camino. editar main para ver otras piezas.
    main((0, 0), 1000000)
```

4.2 Alfil:

De forma equivalente, se muestran los resultados.

Mapa de calor generado por la simulación:

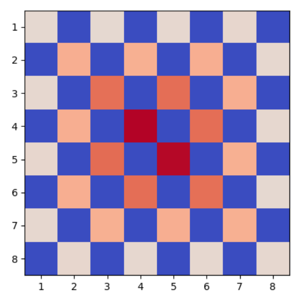

Figure 5: Simulación Alfil

Mapa de calor obtenido a travéz de la fila 1 de la enésima matriz:

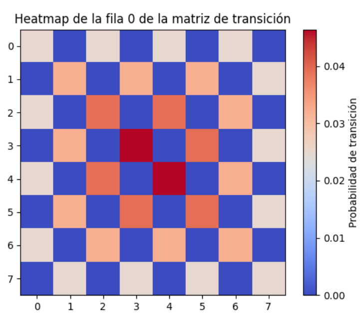

Figure 6: Probabilidad Alfil

Indistingibles. El código de implementación es equivalente para todas las piezas.

4.3 Torre:

Equivalentemente. Simulación del camino:

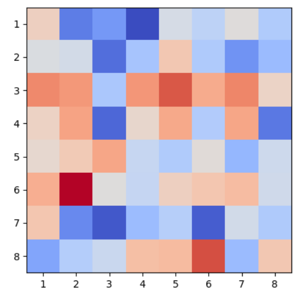

Figure 7: Simulación Torre.

fila 1 de la enésima matriz de la torre:

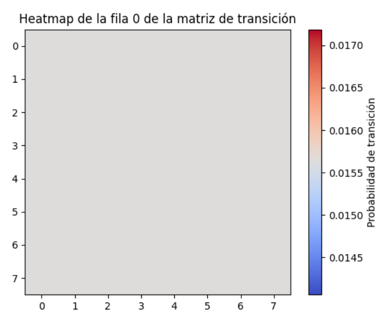

Figure 8: Probabilidad Torre

En este caso, vemos que todas las casillas tienen la misma probabilidad. Lo que explica el patrón aleatorio observado en la simulación. Se conjetura que al aumentar fuertemente la cantidad de pasos se comenzará a ver nun patrón más homogéneo.

Simulación camino de 100000000 de movimientos:

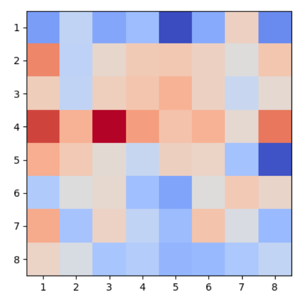

Figure 9: Simulación Torre 2

No se observa cambio al aumentar en 2 órdenes de magnitud la cantidad de movimientos.

4.4 Reina:

De manera equivalente, se obtiene el mapa de calor de la reina:

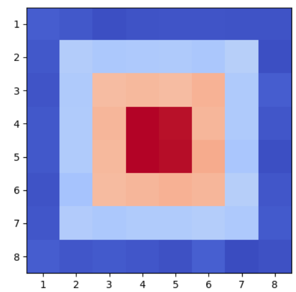

Figure 10: Simulación Reina.

Y se compara con sus valores de la fila 1 de la enésima matriz de transición.

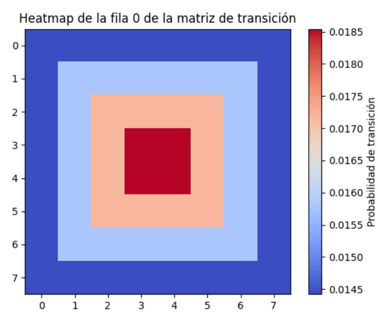

Figure 11: Probabilidad Reina.

4.5 Rey:

Por último, se observa la simulación del rey:

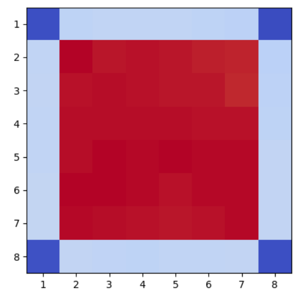

Figure 12: Simulación Rey.

Y se compara con la fila 1 de la enésima matriz.

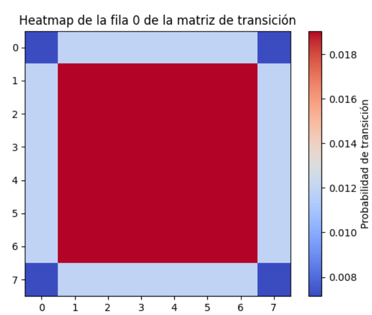

Figure 13: Probabilidad Rey.

5 Conclusión

En resumen, se observa que los valores numéricos tienen un bajo error respecto a los valores esperados con el análisis de la matriz de transición.

Respecto a los mapas de calor, se observa que la casillas más importantes para cada pieza son las que están conectadas con una mayor cantidad casillas. Es decir, donde hay más movimientos posibles. Si se analiza la cantidad de estos, se puede deducir que tan importante es.

Por esta razón, vemos que las casillas centrales son más importantes para todas las piezas, exceptuando la torre, ya que esta es la única que tiene la misma cantidad de movimietos posibles independientemente de su posición.