Universidad Nacional De Colombia
Facultad de Ciencias

Tarea 1: Muestreo de Gibbs

Juan Sebastián Rueda Segura
Sara Sofía Rivera Sanabria
Santiago Rocha Pachón

Freddy Rolando Hernández Romero
Cadenas de Markov
Bogotá, Colombia
26 de diciembre de 2025

Tabla de contenidos

Muestreo de Gibbs para el modelo Hard-Core

1.1. Resumen

El muestreo de Gibbs es una herramienta que permite estimar una distribución estacionaria de una cadena de Markov asociada a ella. En el caso del modelo Hard-Core, es un método para estimar el número de configuraciones factibles en un espacio de configuración cualquiera, usando el teorema de convergencia y evitando realizar procedimientos computacionalmente inviables. A continuación, se explicará más en detalle el modelo Hard-Core y el muestreo de Gibbs.

1.1.1. El espacio de configuración

Formalmente, dado un grafo $G = (V, E)$ y un conjunto $S \neq \emptyset$ finito, se define el espacio de configuración como S^V, i.e., todas las funciones $f : V \to S$. A cada elemento de este espacio se le denomina configuración, y son usualmente representadas por una $|V|$-tupla con las imágenes de cada vértice, $(f(v_1), f(v_2), ..., f(v_{|V|}))$, donde $v_i \in V$. En el modelo Hard-Core en particular, el espacio tiene las siguientes características:

- El grafo $G = (V, E)$ tiene:
  - $|V| = n \times n$ vértices ($n \in \mathbb{N}^+$).
  - $|E| = 2 \cdot (n-1) \cdot n$ aristas que conectan los vértices de modo que se forme una 'grilla'.
- S tiene dos elementos, por lo que cada vértice puede tomar uno de dos valores distinguibles.

Un ejemplo se muestra en la Figura 1.1.

En suma, es de interés saber cuáles configuraciones del grafo (i.e., asignándole a cada vértice) son **factibles**. Estas son aquellas en las que, dado $S = \{b, w\}$, para todo vértice en V, si el vértice tiene valor asociado b, entonces ninguno de sus vecinos puede tener dicho valor asociado (es decir, todos sus vecinos deben tener valor w). De nuevo, un ejemplo de una configuración factible es la Figura 1.

¿Nota: Sin pérdida de generalidad y siguiendo la mayoría de los textos que incluyen el modelo hard-core, S será el conjunto $\{0, 1\}$, donde 1 se representa como un vértice en negro y 0 como un vértice en blanco.

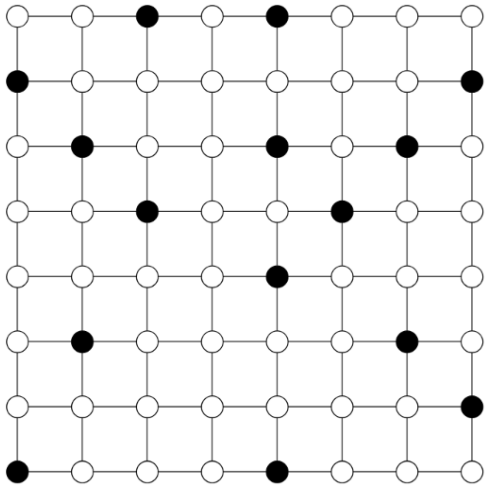

Figura 1.1: Modelo Hard-Core cuyo grafo asociado tiene $8 \times 8 = 64$ vértices y dos posibles 'colores' (elementos de S) para cada vértice: blanco y negro.

1.1.2. El problema

El número de configuraciones posibles crece exponencialmente, pues, como cada vértice puede tener dos valores y hay n^2 de estos, habrán 2^{n^2} configuraciones posibles. Esto hace computacionalmente inviable inspeccionar si cada una de las configuraciones posibles es viable muy rápidamente a medida que se incrementa n.

1.1.3. La solución: una cadena de Markov

Si cualquiera de los valores de S tiene la misma probabilidad de salir, y cada vértice puede ser escogido con la misma probabilidad, entonces cada una de las configuraciones factibles tendrá la misma probabilidad de salir si se asignan aleatorios (pero válidos) valores de S a cada uno de los n^2 vértices. Además, de una configuración factible se puede llegar a otra alterando el valor de S que tome un número de vértices menor a $|V| = n^2$.

Así pues, si se define un procedimiento capaz de ir de una configuración factible a otra tal que no hayan configuraciones inviables absorbentes, entonces se podrá diseñar una cadena de Markov reversible, irreducible y aperiódica cuya distribución estacionaria tenderá a

$$\mu_G(\eta) = \begin{cases} \frac{1}{Z_G} & \text{si } \eta \text{ es una configuración factible.} \\ 0 & \text{en otro caso} \end{cases}$$

que es justamente lo que buscamos.

1.1.4. Primer paso: Muestrear μ_G

El procedimiento mencionado existe y consiste en lo siguiente:

Se realizarán $n \in \mathbb{N}^+$ iteraciones del algoritmo a continuación; en la iteración $1 \leq t \leq n$:

1. Se elige un vértice $v \in V$ al azar (uniformemente).
2. Se lanza una moneda justa ($\mathbb{P}(C) = \mathbb{P}(S) = \frac{1}{2}$).

a) Si sale cara, se pone un 1 en v ($X : n+1(v) = 1$) si todos los vecinos de v en la configuración de X_n son 0. De otro modo, se pone 0 en v ($X : n+1(v) = 0$).

b) Si sale sello, se pone $X : n+1(v) = 0$ (0 en v para la siguiente configuración).

Esto permitirá obtener una muestra de X_n, la cadena de Markov asociada al procedimiento, que es lo que se busca lograr. A continuación, se verá que al hacer n suficientemente grande, tomar suficientes muestras de X_n, y contar el número de partículas que hay en cada muestra, se podrá lograr una 'buena' aproximación del número de partículas que habrían tomando la misma cantidad de muestras de μ_G.

1.2. Implementación del muestreo de Gibbs

Todos los códigos a continuación se encuentran en el Jupyter notebook adjunto. Asímismo, todos han sido realizados con Python y las librerías `numpy` y `matplotlib`.

Para no hacer el documento demasiado largo, se decidió omitir la documentación (al inicio) de las funciones (que sí está en el notebook) y solo conservar los comentarios entre líneas, los cuales pueden presentar ligeras modificaciones para poderlos adaptar al formato del documento.

1.2.1. Funciones auxiliares

Se usó la función `graph_matrix` de la celda 1.1 a lo largo del notebook para poder graficar cualquier configuración, factible o no, para cualquier tamaño de V y de S (siempre y cuando $|V|, |S| \geq 1$).

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

def graph_matrix(M: list[list], color_dict: dict) -> None:
    #*Convert array-like M to np array
    M = np.asarray(M)
    if M.ndim != 2:
        raise ValueError("M must be a 2D array")
    rows, cols = M.shape

    fig, ax = plt.subplots(figsize=(cols, rows))

    #*Set up coordinate system (fitting grid)
    ax.set_xlim(-0.5, cols - 0.5)
    ax.set_ylim(rows - 0.5, -0.5)
    ax.set_xticks(np.arange(0, cols))
    ax.set_yticks(np.arange(0, rows))

    #*Hide tick marks and labels
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.tick_params(length=0)

    #*Add grid lines
    ax.grid(which="major", color="black", linewidth=1)

    #*Calculate size of the circles (particles) depending on the size of M
    cell_width_pts = (fig.get_figwidth() * 72) / max(cols, 1)
    cell_height_pts = (fig.get_figheight() * 72) / max(rows, 1)
    node_size = (0.35 * min(cell_width_pts, cell_height_pts)) ** 2

    #*Plot circles
    for row in range(rows):
        for col in range(cols):
            value = M[row, col]
            if isinstance(value, np.generic):
                value = value.item()
            facecolor = color_dict[value]
            ax.scatter(col, row, s=node_size, c=facecolor, edgecolors="black",
            zorder=3)

    ax.set_aspect("equal") #*square proportions
    plt.show()

```

Celda de código 1.1: `graph_matrix`

1.2.2. Funciones para hacer el muestreo

La principal función usada fue `gibbs_sampler_hcm` (1.2), que se encuentra en la celda 1.2. Esta permite tomar una muestra, tras `n` iteraciones del algoritmo descrito en 1.1.4, de un Hard-Core model cuyo grafo asociado tenga una cuadrícula de `k` $\times$ `k`. Si bien está planteada para un caso más general en el que S puede tener más de dos elementos, conserva la lógica del algoritmo presentado en 1.1.4, comprimiendo el 2.a) y el 2.b) en 3).

En suma, `gibbs_sampler_hcm` (1.2) usa `graph_matrix` (1.1) para imprimir la configuración final (la muestra) y la configuración tras cada `show_each` pasos, es decir, si `show_each` $= p$, entonces se imprimirán las configuraciones $X_p, X_{2p}, ..., X_{ip}$, donde $i \in \mathbb{N}$ es tal que $ip \leq n < (i+1)p$.

Por otro lado, para ejecutar el algoritmo, `gibbs_sampler_hcm` (1.2) se ayuda de `determine_feasible_values` y de `get_neighbors_values` presentadas en la celda 1.3. La primera determina qué valores de S puede tomar el vértice previamente escogido (en el caso de interés, si puede tomar o no el valor de 1), mientras que la segunda permite obtener los valores de todos los vértices vecinos a algún vértice.

```python
def gibbs_sampler_hcm(k: int, initial_config: list[list], S: list = ['w', 'b'],
n: int = 10000, show_each: int = 1000, verbose=False) -> list[list]:
    X: np.array = initial_config.copy()
    S_set: set = set(S)
    if len(set(S)) != len(S):
        raise Exception(f"S, {S}, has at least a pair of non-distinct elements.")
    for i in range(1, n + 1):
        #*1) Pick a vertex at random (but uniformly)
        rand1: int = random.randint(0, k*k - 1)
        vertex: tuple = (rand1 // k, rand1 % k)
        # print("Vertex:", vertex)

        #*2) Pick a random color from the uniform distribution of the
        #*possible colors vertex can take
        neighbors_vals: list = get_neighbors_values(vertex, X)
        feasible_values: list = determine_feasible_values(neighbors_vals, S)
        if S[0] not in feasible_values:
            feasible_values.append(S[0]) #*since 0 or white can always appear
        if len(feasible_values) == 0:
            if verbose:
                print(f"Neighbors have all {len(S)} values")
        else:
            rand2: int = random.randint(0, len(feasible_values) - 1)
            #*3) Update the chosen vertex according to the distribution
            #*and leave the rest unchanged
            X[vertex[0]][vertex[1]] = feasible_values[rand2]

        if show_each != 0:
            if i % show_each == 0:
                print(f"Configuration at {i} out of {n}")
                graph_matrix(X, {"b": "black", "w": "white"})

    return X
```

Celda de código 1.2: `gibbs_sampler_hcm`, la función principal para el muestreo del modelo Hard-Core

```python
def determine_feasible_values(neighbors_values: list, S: list) -> list:
    neighs: set = list(set(neighbors_values))
    return [val for val in S if val not in neighs]

def get_neighbors_values(vertex: tuple, current_config: np.array) -> tuple:
    row: int = vertex[0]
    col: int = vertex[1]
    neighbors: list = []
    #*Left
    try:
        neighbors.append(current_config[row][col-1])
    except IndexError:
        pass
    #*Top
    try:
        neighbors.append(current_config[row-1][col])
    except IndexError:
        pass
    #*Right
    try:
        neighbors.append(current_config[row][col+1])
    except IndexError:
        pass
    #*Botttom
    try:
        neighbors.append(current_config[row+1][col])
    except IndexError:
        pass

    return neighbors
```

Celda de código 1.3: `determine_feasible_values` y `get_neighbors_values`, funciones auxiliares de `gibbs_sampler_hcm` (1.2).

1.3. Estimación del número de partículas

1.3.1. Funciones necesarias

Teniendo el algoritmo para muestrear ya implementado, solo resta poder contar cuántas partículas hay en cada configuración, tomar un número arbitrario de muestras (no solo uno) y graficar la frecuencia del número de partículas por cada conjunto de muestras tomadas.

Lo primero se logra con `count_n_of_x` (1.4), que permite contar cuántos elementos de una configuración son `x`, donde `x` $\in$ `S` (para el caso de interés, x será 1 y $S = \{0, 1\}$).

```python
def count_n_of_x(config: list[list], x: any, S: list = ['w', 'b']) -> int:
    """
    Count the numbers of `x` in `config`, where `x` is an element of `S` and
    config is a lists of lists of elements of `S`
    """
    if x not in S:
        raise ValueError(f"x: {x} is not an element of S: {S}")

    count: int = 0
    for l in config:
        for y in l:
            if y == x:
                count+=1

    return count
```

Celda de código 1.4: `count_n_of_x`.

Lo segundo lo posibilita `build_record` (1.5), que permite construir un registro o historial de cuantas muestras se quiera de X_n, con n fijo.

```python
def build_record(k: int, initial_config: list[list], S: list = ['w', 'b'],
particle_id: str = 'b', num_sim: int = 200, n: int = n,
color_dict: dict = {'b': "black", 'w': "white"}, verbose = False) -> None:
    if particle_id not in S:
        raise ValueError(f"Particle ID {particle_id} is not in S: {S}")

    record: list[list[list]] = []
    Xn: list[list] = []
    for i in range(num_sim):
        Xn = (gibbs_sampler_hcm(k, initial_config, S, n, n+1))
        count: int = count_n_of_x(Xn, particle_id, S)
        if verbose:
            print(f"Count of {particle_id} at iteration {i}: {count}")
        record.append(count)

    return record
```

Celda de código 1.5: `build_record`.

Finalmente, `build_histogram_from_record` permite construir un histograma de la frecuencia

del número de partículas a partir de un registro construido con `build_record` (1.5).

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

def build_histogram_from_record(record: list, k: int, n: int,
particle_id: str = 'b', color_dict = {'b': "black", 'w': "white"},
ax=None, show=True, label=None, color=None, alpha=0.6) -> None:
    #*Style and figure
    plt.style.use('seaborn-darkgrid' if 'seaborn-darkgrid'
                  in plt.style.available else 'default')
    if ax is None:
        fig, ax = plt.subplots(figsize=(10, 6))

    if len(record) == 0:
        return
    #*Get histogram data (so that the distribution of the bars is
    #(done uniformly)
    unique_values = sorted(set(record))
    counts = [record.count(val) for val in unique_values]

    #*Create bars with small gaps between them
    bar_width = 0.95
    bars = ax.bar(unique_values, counts, width=bar_width,
                  color=color or 'steelblue', alpha=alpha,
                  edgecolor='navy', linewidth=0.5,
                  label=label or f"n={n}")
    for bar, count in zip(bars, counts):
        ax.text(bar.get_x() + bar.get_width()/2.,
        bar.get_height() + max(counts)*0.01,
        f'{count}', ha='center', va='bottom',
        fontweight='bold', fontsize=8)
    mean_particles = np.mean(record)
    std_particles = np.std(record)
    max_particles = max(record)
    min_particles = min(record)
    #*Add titles and labels
    if show:
        ax.set_xlabel(f'Number of {color_dict[particle_id]} particles',
                      fontsize=12, fontweight='bold')
        ax.set_ylabel('Frequency', fontsize=12,
                      fontweight='bold')
        ax.set_title(f'Distribution of Particles in Hard-Core Model\n'
                     f'Grid: {k}×{k}  -  Iterations (n in X_n): {n}
                     -  Simulations: {len(record)}',
                     fontsize=14, fontweight='bold', pad=20)

    #*Set integer ticks on x-axis
        ax.set_xticks(unique_values)
        ax.set_xticklabels(unique_values)

    #*Add grid for better readability
        ax.grid(True, alpha=0.3, linestyle='--')
        ax.set_axisbelow(True)

    #*Add statistics text box
    stats_text = f"""Mean: {mean_particles:.2f}\nStd: {std_particles:.2f}\nRange:
                     [{min_particles}, {max_particles}]"""
    ax.text(0.02, 0.98, stats_text, transform=ax.transAxes,
        verticalalignment='top', bbox=dict(boxstyle='round', facecolor='wheat',
                                           alpha=0.8), fontsize=10)
    ax.legend()
```

Celda de código 1.6: `build_histogram_from_record`.

```
    # Adjust layout and show
        plt.tight_layout()
        plt.show()

    # Print summary statistics
    print(f"\nHistogram Summary for different number of iterations:")
    print(f" -TOTAL ITERATIONS: {n}")
    print(f" -Total simulations: {len(record)}")
    print(f" -Particle range: {min_particles} to {max_particles}")
    print(f" -Mean particles: {mean_particles:.2f} ± {std_particles:.2f}")
    print(f" -Most frequent count: {max(unique_values, key=record.count)}
            (appears {max(counts)} times)")
```

Celda de código 1.7: `build_histogram_from_record`.

1.3.2. Experimento

Para estimar el número de partículas "típico"que tiene una configuración factible de un modelo Hard-Core con $|V| = k \times k$, se generaron dos histogramas superpuestos para $k \in \mathbb{N}, 3 \leq k \leq 20$, donde un histograma cuenta el número de partículas de muestras de $X_{10,000}$ (azul) y el otro de muestras de $X_{100,000}$ (naranja). Esto se hizo usando `build_histogram_from_record` como en la celda 1.8.

A continuación, se presentarán las gráficas obtenidas con el código 1.8, cabe recalcar que en el documento solo se enseñarán algunas. Las gráficas en su totalidad se pueden hallar en el cuaderno de Jupyter.

```python
Ks = range(3, 21)
ns = [1000, 10000]
num_sims = [1000]
S = ['w', 'b']
particle_id = 'b'

colors = ['steelblue', 'darkorange']

for k in Ks:
    fig, ax = plt.subplots(figsize=(8, 5))

    for color, n in zip(colors, ns):
        for num_sim in num_sims:
            X0 = [k * ['w'] for _ in range(k)]
            record = build_record(k, X0, S, particle_id, num_sim=num_sim, n=n)
            build_histogram_from_record(record, k, n, particle_id, ax=ax, show=False,
                                        label=f"n={n}", color=color, alpha=0.6)

    ax.set_title(f"Distribución de partículas para k = {k}")
    ax.set_xlabel("Número de partículas")
    ax.set_ylabel("Frecuencia")
    ax.legend(title="Iteraciones n")
    ax.grid(True, alpha=0.3, linestyle='--')
    plt.tight_layout()
    plt.show()
```

Celda de código 1.8: `build_histogram_from_record`.

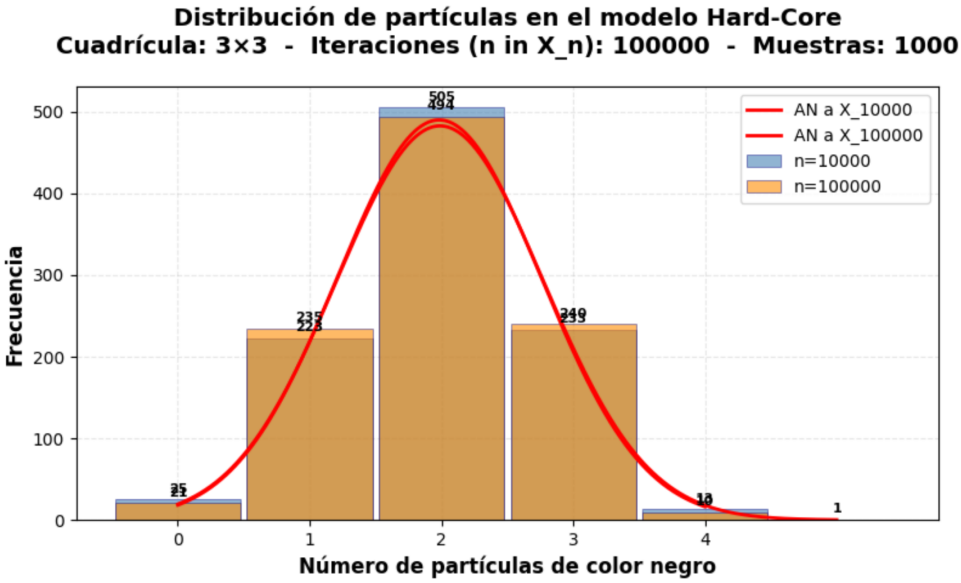

Figura 1.2: Histograma de frecuencias que muestra el número de partículas que tiene una configuración factible en una rejilla de $3 \times 3 = 9$ vértices luego de correr $10,000$ y $100,000$ iteraciones.

Para la rejilla mas pequeña, tenemos que para $10,000$ iteraciones, el numero promedio de particulas es de 1.99 ± 0.79, mientras que para $100,000$ iteraciones es de 1.98 ± 0.77.

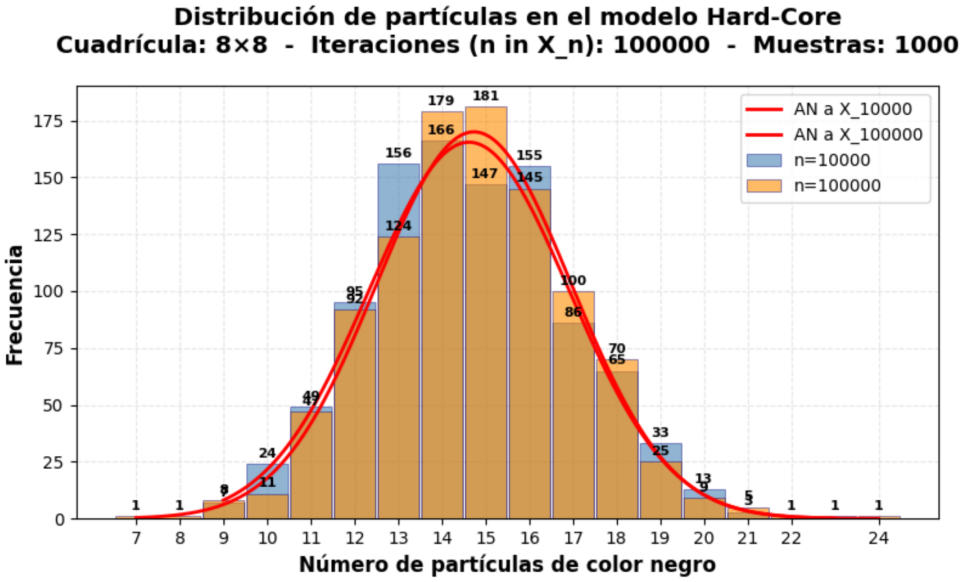

Figura 1.3: Histograma de frecuencias que muestra el número de partículas que tiene una configuración factible en una rejilla de $8 \times 8 = 64$ vértices luego de correr $10,000$ y $100,000$ iteraciones.

En el caso en el que la cuadrícula tenga 64 vértices, empezamos a notar algunos cambios substanciales entre las frecuencias para los diferentes números de iteraciones. El número medio de partículas y la desviación estándar también empiezan a diferir por una cantidad ciertamente mayor que en el caso de $k = 3$, pues para $10,000$ iteraciones, es 14.62 ± 2.29, pero tras $100,000$ iteraciones, es 14.74 ± 2.23

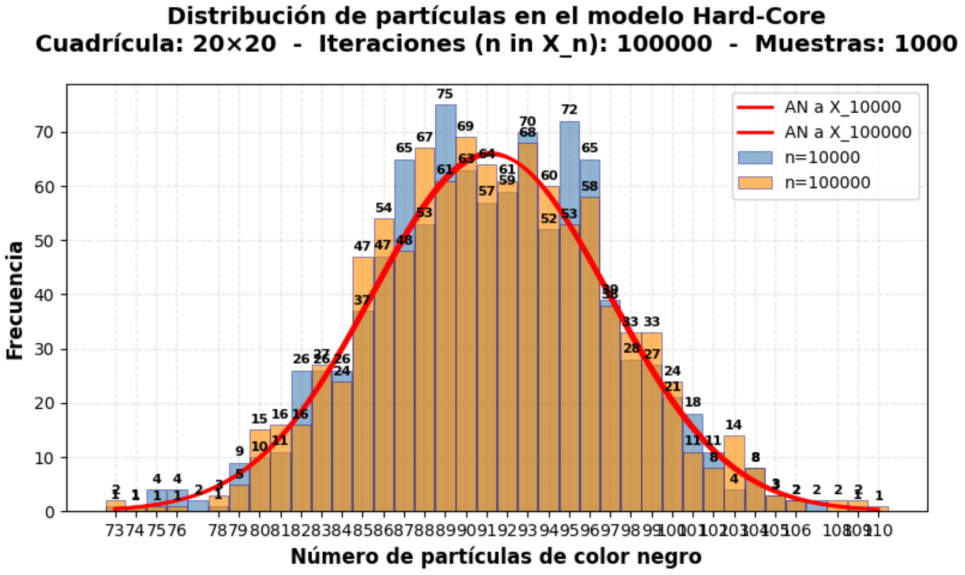

Figura 1.4: Histograma de frecuencias que muestra el número de partículas que tiene una configuración factible en una rejilla de $20 \times 20 = 400$ vértices luego de correr $10,000$ y $100,000$ iteraciones.

Para nuestro último experimento con 400 vértices, es visible el drástico cambio que a lo largo de la variación de k se ha acumulado entre las $10,000$ iteraciones en comparación con las $100,000$ iteraciones. A pesar de esto, se mantiene congruencia con el número típico de partículas: para

$10,000$ iteraciones, 91.02 ± 5.61 partículas, y para $100,000$ iteraciones, 91.45 ± 5.59 partículas.

En conclusión, el experimento evidencia que, conforme aumenta el tamaño de la rejilla y el número de iteraciones, la distribución del número de partículas se vuelve más estable y concentrada alrededor de un valor promedio bien definido. Para rejillas pequeñas, las diferencias entre 10,000 y 100,000 iteraciones son mínimas, lo que indica una rápida convergencia del sistema. A medida que k crece, la distribución se ensancha y adquiere una forma aproximadamente normal, reflejando una mayor variabilidad en el número de partículas que puede tener una configuración factible. Aun asi, las medias entre ambos numeros de iteraciones se mantienen muy próximas.

Muestreo de Gibbs para q-coloraciones

2.1. Resumen

Como se explicó antes, podemos utilizar el muestreo de Gibbs para estimar una distribución estacionaria de una cadena de Markov asociada a esta. El problema de las q-coloraciones es un caso más general al modelo Hard-Core, ya que se puede ver como un problema con solo 2 colores.

2.1.1. El espacio de estados

Es completamente análogo al descrito anterionrmente con el modelo Hard-Core exceptuando que S será el conjunto de q colores para rellenar la grilla. Un ejemplo se muestra en la figura 2.1.

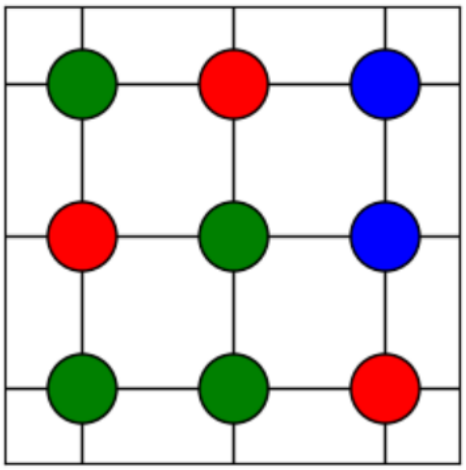

Figura 2.1: 3-coloración en una grilla 3x3

2.1.2. El problema

Al igual que en Hard-Core estamos interesados en conocer el número de configuraciones factibles para una q-coloración. Estas son aquellas donde para todo vértice, su color es diferente al color de los vértices conectados con este. Notemos entonces que la figura anterior se trata de una configuración no factible.

Como hallamos una cadena de Markov que converge a la distribución estacionaria μ_G, entonces podemos averiguar cuál es el número de configuraciones factibles si dejamos correr la cadena un buen tiempo ya que no sabemos cuál es el N mínimo para alcanzar la convergencia.

2.1.3. Muestrear μ_G

Se realizarán $n \in \mathbb{N}^+$ iteraciones del algoritmo a continuación; en la iteración $1 \leq t \leq n$:

1. Se elige un vértice $v \in V$ al azar (uniformemente).
2. Se añaden los colores de los vértices adyacentes (vecinos) al escogido en el paso 1.
3. Se toma el complemento a esa lista respecto a la lista de colores, para obtener los que serían válidos para poner en el vértice escogido.
4. De la lista de colores válidos se escoge uno uniformemente y se le asigna a v.

Esto permitirá obtener una muestra de X_n, la cadena de Markov asociada al procedimiento, que es lo que se busca lograr, ya que garantiza que para muchas iteraciones se puedan obtener q-coloraciones factibles. A continuación, se verá que al hacer n suficientemente grande, tomar suficientes muestras de X_n, y contar el número de partículas que hay en cada muestra, se podrá lograr una 'buena' aproximación del número de partículas que habrían tomando la misma cantidad de muestras de μ_G.

2.2. Implementación

2.2.1. Función auxiliar

Esta función halla la lista de colores válidos para poner en un vértice escogido. Añade el color de los vecinos a una lista y retorna el complemento respecto a la lista de colores total.

```python
def valid_colors(x, y, grid: np.ndarray, colors) -> list:
    # ! Realized might be better to use set
    neighbors_colors = []
    grid_size = grid.shape[0]
    if x > 0:
        neighbors_colors.append(grid[x - 1, y])
    if x < grid_size - 1:
        neighbors_colors.append(grid[x + 1, y])
    if y > 0:
        neighbors_colors.append(grid[x, y - 1])
    if y < grid_size - 1:
        neighbors_colors.append(grid[x, y + 1])

    valid_color = [c for c in colors if c not in neighbors_colors]

    return valid_color
```

Celda de código 2.1: `valid_colors`.

2.2.2. Funciones principales

Como se mencionó en clase, podemos generar muestras sin importar la configuración inicial de la grilla, ya que por el Teorema de convergencia, eventualmente convergerá a la distribución estacionaria. Es por esto que planteamos generar tres posibles configuraciones iniciales:

Configuración inicial "vacía"

Se trata de inicializar la grilla con un solo color (un ejemplo se puede ver en la figura 2.2), en este caso blanco (el código se presenta en la celda 2.2).

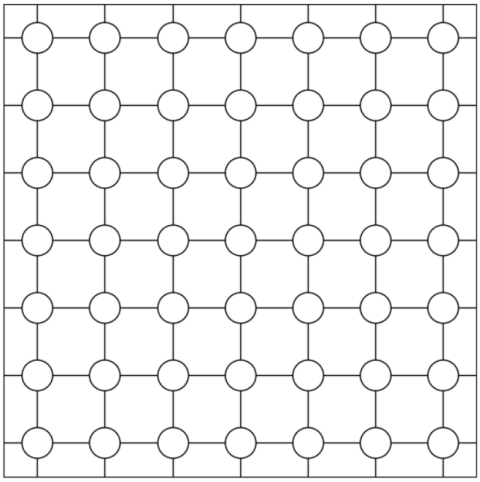

Figura 2.2: Configuración inicial "vacía" en una grilla 7x7

```python
def gibbs_sampler_q_coloration_blank(grid_size: int, color_dict: dict,
iterations: int, show_graph: bool = True):
    grid = np.zeros((grid_size, grid_size), dtype=int)

    colors = list(color_dict.keys())
    if show_graph:
        graph_matrix(grid, color_dict)

    for _ in range(iterations):
        (x, y) = np.random.randint(0, grid_size, size=2)
        valid_color = valid_colors(x, y, grid, colors)
        if valid_color:
            grid[x, y] = np.random.choice(valid_color)
    if show_graph:
        graph_matrix(grid, color_dict)
    return grid
```

Celda de código 2.2: `gibbs_sampler_q_coloration_blank`.

Configuración inicial aleatoria

Se trata de inicializar la grilla donde cada vértice tiene un color aleatorio. Muy posiblemente no sea una q-coloración factible (2.3).

```
def gibbs_sampler_q_coloration_random_start(grid_size: int, colors: list,
iterations: int):
    grid = np.random.choice(colors, size=(grid_size, grid_size))
    # Inicializar con colores aleatorios, puede que no sea válida la configuración

    for _ in range(iterations):
        (x, y) = np.random.randint(0, grid_size, size=2)
        valid_color = valid_colors(x, y, grid, colors)
        if valid_color:
            grid[x, y] = np.random.choice(valid_color)

    return grid
```

Celda de código 2.3: `gibbs_sampler_q_coloration_random_start`.

Configuración inicial diagonal

Se inicializa con los primeros n colores dispuestos de manera diagonal (celda 2.4), como se puede ver en la siguiente figura (2.3):

```
def gibbs_sampler_q_coloration_diagonal_colors(grid_size: int, color_dict: dict,
iterations: int, show_graph: bool = True) -> np.ndarray:
    colors = list(color_dict.keys())
    grid = np.zeros((grid_size, grid_size), dtype=int)
    for i in range(grid_size):
        for j in range(grid_size):
            grid[i, j] = colors[(i + j) % len(colors)]

    if show_graph:
        graph_matrix(grid, color_dict)

    for _ in range(iterations):
        (x, y) = np.random.randint(0, grid_size, size=2)
        valid_color = valid_colors(x, y, grid, colors)
        if valid_color:
            grid[x, y] = np.random.choice(valid_color)

    if show_graph:
        graph_matrix(grid, color_dict)
    return grid
```

Celda de código 2.4: `gibbs_sampler_q_coloration_random_start`.

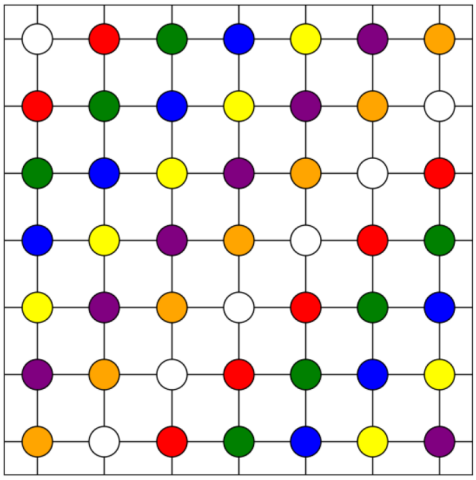

Figura 2.3: 7-coloración (factible) en una grilla 7x7

2.2.3. Experimento

A continuación se realizó un experimento para estimar el número de partículas "típico"de cada color que tiene una configuración factible de un modelo de q-coloraciones con $|V| = k \times k$. Al igual que en el experimento pasado, se generaron dos histogramas superpuestos para $k \in \mathbb{N}, 3 \leq k \leq 20$, y estos se agrupaban segun $q \in \mathbb{N}, 2 \leq 1 \leq 10$ donde un histograma cuenta el número de partículas de cada color de muestras de $X_{1,000}$ (azul) y el otro de muestras de $X_{10,000}$ (naranja). Esto se hizo usando `build_histogram_from_record_q_coloration`.

Analogamente al modelo hard-core se tiene una funcion `build_record_q_coloration`, que permite construir un registro o historial de cuantas muestras se quiera de X_n, con n fijo.

```python
def build_record_q_coloration(k: int, color_dict: dict, n: int, num_sim: int = 100,
                              sampler: str = "diagonal",
                              verbose: bool = False) -> list[dict]:
    record = []
    for i in range(num_sim):
        if sampler == "random":
            grid = gibbs_sampler_q_coloration_random_start(k,
            list(color_dict.keys()), n)
        elif sampler == "diagonal":
            grid = gibbs_sampler_q_coloration_diagonal_colors(k,
            color_dict, n, show_graph=False)
        elif sampler == "blank":
            grid = gibbs_sampler_q_coloration_blank(k,
            color_dict, n, show_graph=False)
        else:
            raise ValueError("Invalid sampler. Choose from 'random', 'diagonal',
            or 'blank'.")
        counts = count_number_of_particles(grid, color_dict)
        record.append(counts)
        if verbose:
            print(f"Simulation {i+1}/{num_sim}: {counts}")

    return record
```

Celda de código 2.5: `build_record_q_coloration`.

En este caso, `build_histogram_from_record_q_coloration` (celdas 2.6 y 2.7) construye un histograma de la frecuencia del número de partículas de cada color en una q-coloracion a partir de un registro construido previamente con `build_record_q_coloration`.

```python
def build_histogram_from_record_q_coloration(record: list[dict], k: int, n: int,
color_dict: dict, sampler: str = "diagonal", ax=None, show=True, label=None,
color=None, alpha=0.6) -> None:
    import matplotlib.pyplot as plt
    import numpy as np

    plt.style.use('seaborn-darkgrid' if 'seaborn-darkgrid' in plt.style.available
    else 'default')
    if ax is None:
        fig, ax = plt.subplots(figsize=(10, 6))

    if len(record) == 0:
        return

    color_names = list(color_dict.values())

    #calc mean counts for each color across all simulations
    mean_counts = []
    for color_name in color_names:
        counts = [sim_counts.get(color_name, 0) for sim_counts in record]
        mean_counts.append(np.mean(counts))

    x_pos = np.arange(len(color_names))
    bar_width = 0.95
    bars = ax.bar(x_pos, mean_counts, width=bar_width, color=color or 'steelblue',
    alpha=alpha, edgecolor='navy', linewidth=0.5, label=label or f"n={n}")

    #value labels on bars
    for bar, count in zip(bars, mean_counts):
        ax.text(bar.get_x() + bar.get_width()/2.,
        bar.get_height() + max(mean_counts)*0.01, f'{int(count)}',
        ha='center', va='bottom', fontweight='bold', fontsize=8)

    #titles and labels only if show=True
    if show:
        ax.set_xlabel('Color', fontsize=12, fontweight='bold')
        ax.set_ylabel('Frecuencia', fontsize=12, fontweight='bold')
        ax.set_title(f'Distribución de colores para k = {k}', fontsize=14,
        fontweight='bold', pad=20)

        ax.grid(True, alpha=0.3, linestyle='--')
        ax.set_axisbelow(True)

        ax.legend(title="Iteraciones n")
        plt.tight_layout()
        plt.show()
```

Celda de código 2.6: Parte 1 del código de `build_histogram_from_record_q_coloration`.

```
    ax.set_xticks(x_pos)
    ax.set_xticklabels([name.capitalize() for name in color_names])

    print(f" -Grid size: {k}×{k} ({k*k} cells)")
    print("Mean cells per color:")
    for color_name, mean_val in zip(color_names, mean_counts):
        print(f"  {color_name.capitalize()}: {mean_val:.2f}")
```

Celda de código 2.7: Parte 2 del código de `build_histogram_from_record_q_coloration`.

Por ultimo, se definen los parametros para correr el experimento (2.8 y 2.9). Seguido a esto, se presentarán algunas de las gráficas obtenidas (2.4). Cabe recalcar que en el documento solo se enseñarán algunas. Las gráficas en su totalidad se pueden hallar en el cuaderno de Jupyter.

```
color_palettes = {
    2: {1: "red", 2: "green"},
    3: {1: "red", 2: "green", 3: "blue"},
    4: {1: "red", 2: "green", 3: "blue", 4: "yellow"},
    5: {1: "red", 2: "green", 3: "blue", 4: "yellow", 5: "purple"},
    6: {1: "red", 2: "green", 3: "blue", 4: "yellow", 5: "purple", 6: "orange"},
    7: {1: "red", 2: "green", 3: "blue", 4: "yellow", 5: "purple", 6: "orange",
    7: "cyan"},
    8: {1: "red", 2: "green", 3: "blue", 4: "yellow", 5: "purple", 6: "orange",
    7: "cyan", 8: "magenta"},
    9: {1: "red", 2: "green", 3: "blue", 4: "yellow", 5: "purple", 6: "orange",
    7: "cyan", 8: "magenta", 9: "pink"},
    10: {1: "red", 2: "green", 3: "blue", 4: "yellow", 5: "purple", 6: "orange",
    7: "cyan", 8: "magenta", 9: "pink", 10: "brown"}
}

Ks = range(3, 21)
num_sim = 1000
ns = [100, 1000]
bar_colors = ['steelblue', 'darkorange']
sampler = "diagonal"
```

Celda de código 2.8: Definición de las paletas de colores para las 9 q-coloraciones usadas y de los parámetros para `build_histogram_from_record_q_coloration`.

```python
# Loop over each q (color count)
for q in range(2, 11):
    colors_dict = color_palettes[q]

    print(f"\n{'='*60}")
    print(f"PROCESANDO q={q} COLORES")
    print(f"{'='*60}")

    nrows = 6
    ncols = 3
    fig, axes = plt.subplots(nrows, ncols, figsize=(16, 20))
    axes = axes.flatten()

    for idx, k in enumerate(Ks):
        ax = axes[idx]

        for bar_color, n in zip(bar_colors, ns):
            print(f"\n--- Corriendo k={k}, n={n} ---")

            record = build_record_q_coloration(k, colors_dict, n, num_sim,
            sampler=sampler)
            build_histogram_from_record_q_coloration(
                record, k, n, colors_dict, sampler=sampler,
                ax=ax, show=False, label=f"n={n}", color=bar_color, alpha=0.6
            )

        ax.set_title(f"Distribución de colores (q={q}, k={k})", fontsize=13,
        fontweight='bold')
        ax.set_xlabel("Color", fontsize=11, fontweight='bold')
        ax.set_ylabel("Frecuencia", fontsize=11, fontweight='bold')
        ax.legend(title="Iteraciones n")
        ax.grid(True, alpha=0.3, linestyle='--')

    # Hide unused subplots (if any)
    for ax in axes[len(Ks):]:
        ax.set_visible(False)

    fig.suptitle(f"Distribución de colores para q={q}", fontsize=16,
    fontweight='bold')
    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()

    print(f"\n{'='*60}\n")
```

Celda de código 2.9: Ciclos de ejecución variando el número de colores (q) y el tamaño de la cuadrícula (k^2) en los parámetros de `build_histogram_from_record_q_coloration`.

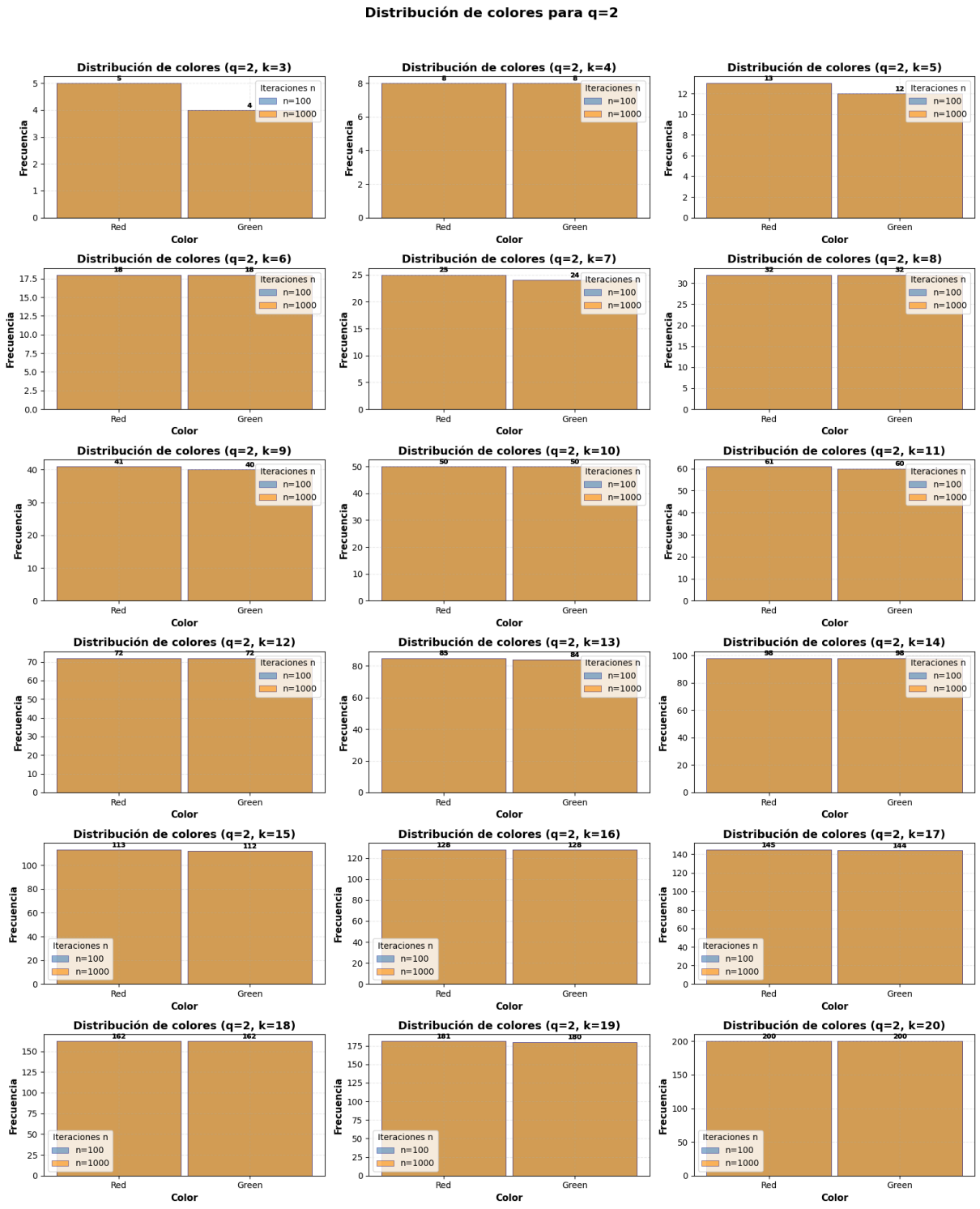

Figura 2.4: Histograma de frecuencias que muestra el número de partículas de cada color que tiene una configuración factible en una rejilla de 9 a 400 vértices y $q = 2$ colores, luego de correr $1,000$ y $10,000$ iteraciones.

Procesando 2 colores, notamos que para los diferentes valores de k, el numero tipico de particulas de cada color es identico independientemente de si se toman 1000 o 10000 iteraciones. Es entonces

posible intuir que la distribución entre los dos colores tiende rápidamente a un equilibrio casi perfecto conforme crece k.

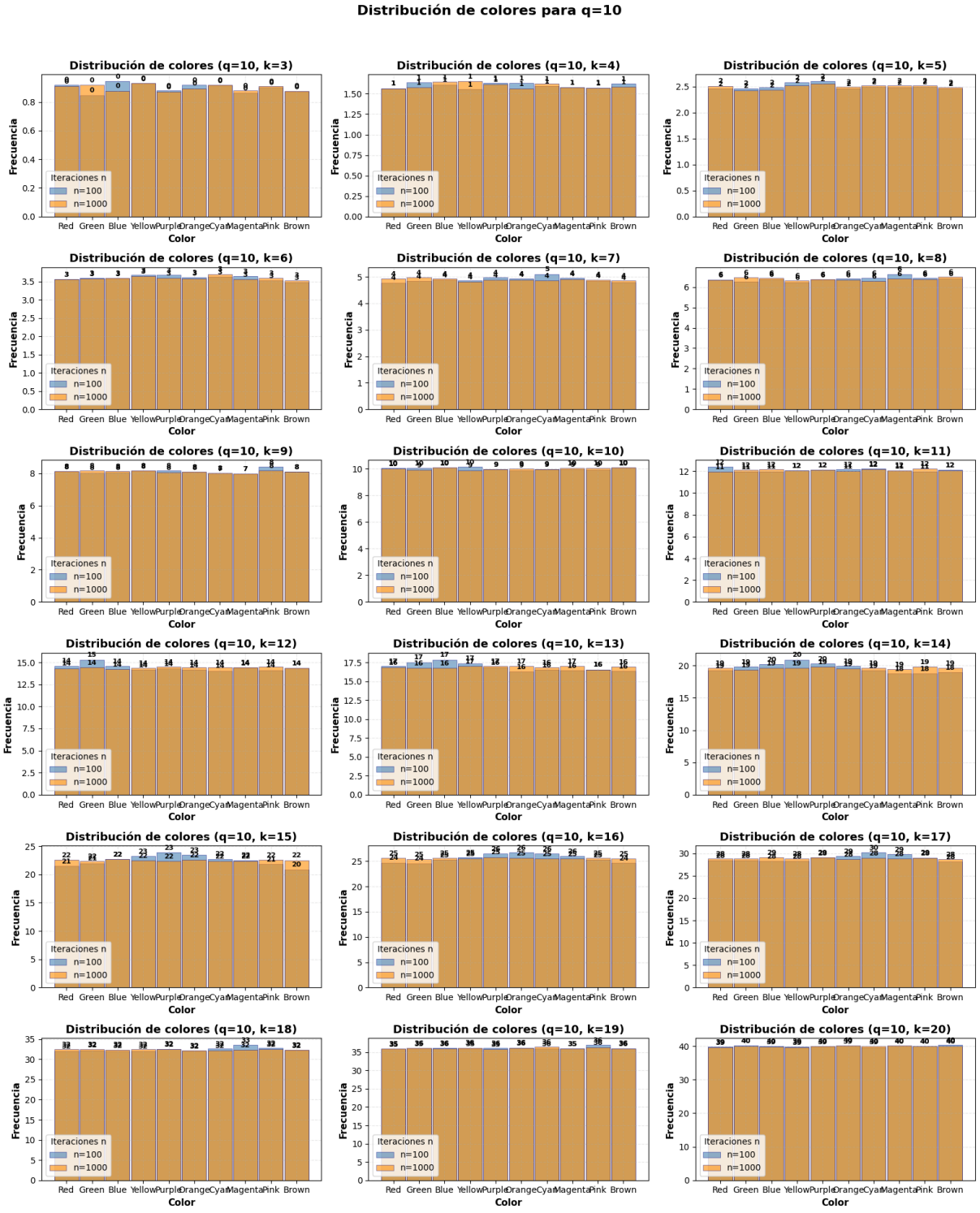

Figura 2.5: Histograma de frecuencias que muestra el número de partículas de cada color que tiene una configuración factible en una rejilla de 9 a 400 vértices y $q = 10$ colores, luego de correr $1,000$ y $10,000$ iteraciones.

Por último, con 10 colores observamos que no ha cambiado mucho el comportamiento de las frecuencias. El experimento muestra que, a medida que aumenta el tamaño de la rejilla, k, y el número de iteraciones, n, las distribuciones de colores en las configuraciones factibles tienden a estabilizarse y volverse uniformes. Las frecuencias de cada color se igualan progresivamente, indicando que el modelo distribuye los colores de forma equitativa conforme crece el número de vértices y se incrementa el muestreo. Esto sugiere que el modelo alcanza o "converge" a un régimen estacionario en el que todas las coloraciones factibles son igualmente probables, que es justamente lo que se estaba buscando.

Por otro lado, se debe mencionar que, aunque con $1,000$ y $10,000$ iteraciones la ejecución de la celda 2.9 con los parámetros de la 2.8 se tardó alrededor de 26 minutos en un computador, cuando se intentó correr en otra máquina con similares especificaciones pero con $10,000$ y $100,000$ iteraciones, iba 44 minutos y no había terminado la simulación para $k = 3$, $q = 2$ y $100,000$ iteraciones, así que se decidió presentar los resultados de las figuras 2.4 y 2.5.