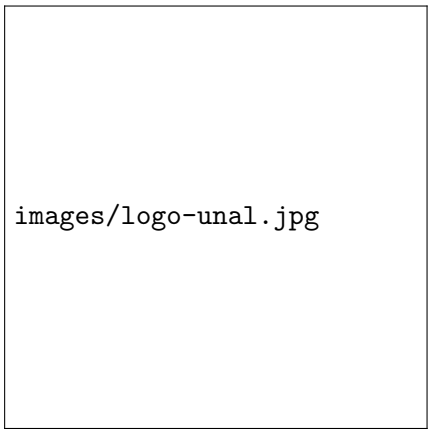
Universidad Nacional De Colombia
Facultad de Ciencias

Tarea 2: Conteo aproximado con MCMC aplicado a los modelos Hard-Core y q-coloraciones

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

Aproximación del número de q-coloraciones

1.1. Links

- Cuaderno de Jupyter con estimaciones:
  Cuaderno de Python.
- Cuaderno de Jupyter en CoCalc para valores exactos usando Sage:
  Cuaderno de Sage

1.2. Resumen

Se demostró en clase (teorema 9.1) que el algoritmo con el que se aproxima, en un grafo $G = (E, V)$ con $|V| = k$, el número de q-coloraciones ($q \in \mathbb{N}, q \geq 2$ son de interés) $Z_{G,q}$ estimando los cocientes Z_i/Z_{i-1} con $i \in \mathbb{N}^+$, donde Z_i es el grafo con las primeros i aristas enumeradas de E; es un Esquema Aleatorizado de Aproximación en Tiempo Polinomial (FPRAS por sus siglas en inglés) para este problema cuando $q > 2d^2$.

Dicho FPRAS requiere a lo más

$$dk \cdot \frac{48d^3k^3}{\epsilon^2} \cdot k \left[\frac{2\log(k) + \log(\epsilon^{-}1) + \log(8)}{\log(\frac{q}{2d^2})} + 1 \right] \tag{1.1}$$

pasos para computar una aproximación $\tilde{Z}_{G,q}$ de $Z_{G,q}$ tal que $Z_{G,q}$ esté en el intervalo $[(1-\epsilon)\tilde{Z}_{G,q}, (1+\epsilon)\tilde{Z}_{G,q}]$.

1.2.1. Problemas

Por un lado, la condición $q > 2d^2$ es muy restrictiva, y aunque se ha logrado mejorar a $q > 2d$ (2) e incluso a $q > \frac{11}{6}d$ (3), para el caso de interés, se debería poder reducir a $q > 4$, y más aún, debería seguir funcionando cuando $q = 3$.

En suma, aunque se tenga certeza de que tras 1.1 pasos el resultado estará en el intervalo mencionado, si, por ejemplo, se toman los parametros $k = 100, d = 4, \epsilon = 0.01$ y $q = 20$, se necesitarían $2.25449 \cdot 10^{19}$ pasos computacionales como mínimo para garantizar la pertenencia del número de q-coloraciones al intervalo, por lo que, al ejecutar el algoritmo en un computador promedio que puede calcular hasta 10^8 operaciones por segundo, tardaría más de 3170 años en dar una estimación.

1.2.2. Objetivo

A pesar de que resultan desalentadoras las cifras de tiempo que requeriría el algoritmo con algunos parámetros, el teorema 9.1 implica que el orden de este algoritmo es $C \cdot k \cdot \log(k)$ para alguna constante $C > 0$, y aunque el algoritmo necesite $i \in \mathbb{N}^+, 1 \leq i \leq |E|$ cocientes para la estimación y $\frac{48d^3k^3}{\epsilon^2}$ muestras para cada cociente, **el número de pasos del muestreador de Gibbs se podría optimizar variando** C.

1.3. Estimación del número de pasos del muestreador de Gibbs

Para saber cuál es la mejor constante C tal que en las estimaciones se hagan $C \cdot k \log(k)$, se intentó utilizar la conclusión de los experimentos realizados en la tarea 1: el número de colores en cada muestra en promedio es uniforme, es decir, en cada muestra deben aparecer $\frac{1}{q}$ colores de cada uno de los q colores.

Sin embargo, como se van a inicializar las configuraciones 'diagonalmente', es decir, poniendo un solo color (distinto al previo) en cada diagonal, resultó que, independientemente del valor de C (siendo el más grande 20 y el más pequeño 10^{-3}), se mantenía la proporción para cada color de $\frac{1}{q}$ usando tres cifras decimales de precisión. Vale la pena mencionar que el experimento se realizó en una rejilla 3×3 con 3 colores.

No obstante, se observó que, tras haber calculado las estimaciones exactas (1.5) y comparando el resultado obtenido con el método que se presentará en 1.4 para distintos valores de C (con los mismos parámetros mencionados en el párrafo anterior), había diferencias significativas con valores menores a 0:

Para el caso de 0.01, las estimaciones estuvieron en el intervalo $[3,000\,,\ 6,000]$ cuando el valor real es 246; para valores entre 0.6 y 0.1, la estimación se ubicaba en el intervalo $[400, 700]$. Luego, al probar con $C = 1$, se evidenció que los valores estaban en el intervalo $[400, 210]$, por lo que, si bien no siempre resultaba siendo la mejor estimación, era considerablemente mejor que valores menores y no era demasiado grande para volverlo infactible con las máquinas y el tiempo que se tenía a disposición.

Por lo tanto, para la estimación del número de q-coloraciones de una rejilla $k \times k$ se usarán $k \log(k)$ iteraciones del muestreador de Gibbs.

1.4. Estimación con muestreador de Gibbs

Una vez seleccionado el valor de C, se procederá a estimar $Z_{G,q}$. Para ello, se hizo una 'refactorización' considerable del código de la tarea 1 para crear la clase `Config`, una generalización de una configuración de un grafo $G = (V, E)$ conexo pero arbitrario, con todos los métodos usados en la tarea 1 tanto para el modelo hard-core como para q-coloraciones.

El código es demasiado amplio, y al ser una abstracción que los autores usaron para facilitar las implementaciones, no se mostrará.

Sin embargo, para el caso de q-coloraciones, lo novedoso es la función `estimate_n_q_colorations`, que, como su nombre lo indica, sirve para estimar el número de q-coloraciones. El código se puede encontrar en la sección *Gibbs sampler for q-coloration* del cuaderno de Jupyter (no se presenta porque, de nuevo, es demasiado extenso).

La utilización de la función se puede ver en la celda 1.1, en la que se realizan estimaciones con los siguientes parametros [1]:

- $\epsilon = 1, 0.9, 0.8$
- $k = 9, 16, 25, 36, 49, 64$.
- $k\log(k), 4 \cdot k\log(k)$ pasos del muestreador de Gibbs.
- $$\left\lfloor \frac{16 \cdot l}{\frac{2}{3} * \epsilon^2} \right\rfloor$$ muestras, siguiendo el teorema 9.1

```python
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"}
}

final_estimates = []
for e in [1, 0.9, 0.8]:
    for k in range(3, 9):
        for C in [1]:
            for idx in color_palettes:
                    estimates, z_estimate = estimate_n_q_colorations(k=k*k, C=C,
                    epsilon=e, p=2/3, color_dict=color_palettes[idx], verbose=False)
                    final_estimates.append([k*k, idx, e, z_estimate])
                    print([k*k, e, idx, z_estimate])
```

Celda de código 1.1: Conteo aproximado variando k, ϵ, q.

Los resultados no valen la pena enlistarlos (aunque lo están en el cuaderno), pues es probablemente de mayor interés para el lector la comparación con el conteo exacto, que se presentará en 1.6.

1.5. Conteo exacto

El polinomio cromático $P(G, q)$ de un grafo G proporciona el número de formas distintas de colorear los vértices del grafo utilizando exactamente q colores, de manera que vértices adyacentes tengan colores diferentes.

[1] Aunque se menciona que se usarán $4 \cdot k\log(k)$ iteraciones del muestreador de Gibbs, esto solo ocurrirá para $\epsilon = 1.0$. Una nota al respecto se mencionará en la comparación con conteo exacto (1.6).

Para este proyecto se requiere calcular el número exacto de q-coloraciones posibles en rejillas cuadradas de $k \times k$ vértices, donde $k \in \{3, 4, \ldots, 20\}$ y $q \in \{2, 3, \ldots, 15\}$. Dado que las rejillas cuadradas representan grafos con estructura regular altamente conectada, el cálculo del polinomio cromático presenta desafíos computacionales significativos. Esto es necesario resaltarlo.

Se implementaron dos enfoques distintos para el cálculo del polinomio cromático utilizando herramientas computacionales especializadas en teoría de grafos.

1.5.1. Implementación con NetworkX

NetworkX es una biblioteca de Python ampliamente utilizada para el análisis de grafos. La implementación se realizó mediante el procedimiento presentado en la celda 1.2. Como se puede evidenciar, la principal función usada de la librería es `chromatic_polynomial`, cuya documentación se puede encontrar aquí.

```python
import networkx as nx
from networkx.algorithms.polynomials import chromatic_polynomial
import numpy as np
from sympy import Symbol

def exact_q_colorations_count(k: int = 3, q: int = 3) -> int:
    G = nx.grid_2d_graph(k, k)
    x = Symbol('x')
    poly = chromatic_polynomial(G)
    exact_count = int(poly.subs(x, q))
    return exact_count
```

Celda de código 1.2: Función usada para calcular el polinomio cromático de un grafo $G = (V, E)$ con $|V| = k$ y q colores.

Una salida de la función descrita anteriormente se presenta en la celda **??**. El desempeño de esta librería se discutirá en 1.5.3.

```
Número exacto de 2 -coloraciones en rejilla 3x3: 2
Número exacto de 3 -coloraciones en rejilla 3x3: 246
Número exacto de 4 -coloraciones en rejilla 3x3: 9612
Número exacto de 5 -coloraciones en rejilla 3x3: 142820
Número exacto de 6 -coloraciones en rejilla 3x3: 1166910
Número exacto de 7 -coloraciones en rejilla 3x3: 6464682
Número exacto de 8 -coloraciones en rejilla 3x3: 27350456
Número exacto de 9 -coloraciones en rejilla 3x3: 95004072
Número exacto de 10 -coloraciones en rejilla 3x3: 283982490
Número exacto de 11 -coloraciones en rejilla 3x3: 754324670
Número exacto de 12 -coloraciones en rejilla 3x3: 1821684612
Número exacto de 13 -coloraciones en rejilla 3x3: 4067709516
Número exacto de 14 -coloraciones en rejilla 3x3: 8506024982
Número exacto de 15 -coloraciones en rejilla 3x3: 16822697010
```

Celda de código 1.3: Salida de 1.2 para $k = 3$ (rejilla 3×3) y variando q entre 2 y 15.

1.5.2. Implementación con SageMath

SageMath es un sistema de álgebra computacional que integra múltiples bibliotecas matemáticas. La implementación (1.4), aprovecha sus capacidades optimizadas para cálculos simbólicos.

```
def exact_q_colorations_sage(k: int = 3, q: int = 3) -> int:

    G = graphs.GridGraph([k, k])
    chromatic_poly = G.chromatic_polynomial()
    exact_count = chromatic_poly(q)
    return int(exact_count)
```

Celda de código 1.4: Función (escrita en el lenguaje Sage, basado en Python) usada para calcular el polinomio cromático de un grafo $G = (V, E)$ con $|V| = k$ y q colores.

A continuación se presenta la salída de código de **??** para grafos con 9 (1.5), 16 (1.6) y 25 (1.7) vértices, variando q entre 2 y 15, tal como en 1.5.1.

```
Número exacto de 2 -coloraciones en rejilla 3x3: 2
Número exacto de 3 -coloraciones en rejilla 3x3: 246
Número exacto de 4 -coloraciones en rejilla 3x3: 9612
Número exacto de 5 -coloraciones en rejilla 3x3: 142820
Número exacto de 6 -coloraciones en rejilla 3x3: 1166910
Número exacto de 7 -coloraciones en rejilla 3x3: 6464682
Número exacto de 8 -coloraciones en rejilla 3x3: 27350456
Número exacto de 9 -coloraciones en rejilla 3x3: 95004072
Número exacto de 10 -coloraciones en rejilla 3x3: 283982490
Número exacto de 11 -coloraciones en rejilla 3x3: 754324670
Número exacto de 12 -coloraciones en rejilla 3x3: 1821684612
Número exacto de 13 -coloraciones en rejilla 3x3: 4067709516
Número exacto de 14 -coloraciones en rejilla 3x3: 8506024982
Número exacto de 15 -coloraciones en rejilla 3x3: 16822697010
```

Celda de código 1.5: Salida de 1.4 para un grafo con 9 vértices variando el número de colores entre 2 y 15.

```
Número exacto de 2 -coloraciones en rejilla 4x4: 2
Número exacto de 3 -coloraciones en rejilla 4x4: 7812
Número exacto de 4 -coloraciones en rejilla 4x4: 6000732
Número exacto de 5 -coloraciones en rejilla 4x4: 828850160
Número exacto de 6 -coloraciones en rejilla 4x4: 38128724910
Número exacto de 7 -coloraciones en rejilla 4x4: 856858754052
Número exacto de 8 -coloraciones en rejilla 4x4: 11722360851992
Número exacto de 9 -coloraciones en rejilla 4x4: 111647093496192
Número exacto de 10 -coloraciones en rejilla 4x4: 807567269568570
Número exacto de 11 -coloraciones en rejilla 4x4: 4707230299664420
Número exacto de 12 -coloraciones en rejilla 4x4: 23062698161984052
Número exacto de 13 -coloraciones en rejilla 4x4: 97963534144477872
Número exacto de 14 -coloraciones en rejilla 4x4: 369313246327400102
Número exacto de 15 -coloraciones en rejilla 4x4: 1258250118125770980
```

Celda de código 1.6: Salida de 1.4 para un grafo con 16 vértices variando el número de colores entre 2 y 15.

```
Número exacto de 2 -coloraciones en rejilla 5x5: 2
Número exacto de 3 -coloraciones en rejilla 5x5: 580986
Número exacto de 4 -coloraciones en rejilla 5x5: 20442892764
Número exacto de 5 -coloraciones en rejilla 5x5: 50820390410180
Número exacto de 6 -coloraciones en rejilla 5x5: 21977869327169310
Número exacto de 7 -coloraciones en rejilla 5x5: 3031776844080257742
Número exacto de 8 -coloraciones en rejilla 5x5: 189586772101347563096
Número exacto de 9 -coloraciones en rejilla 5x5: 6660747304179593671944
Número exacto de 10 -coloraciones en rejilla 5x5: 151086899096935604867610
Número exacto de 11 -coloraciones en rejilla 5x5: 2432514752747773742479970
Número exacto de 12 -coloraciones en rejilla 5x5: 29730874656484222810604532
Número exacto de 13 -coloraciones en rejilla 5x5: 2898142783768863657886861 56
Número exacto de 14 -coloraciones en rejilla 5x5: 233870655888100652735427 8774
Número exacto de 15 -coloraciones en rejilla 5x5: 160799411495544322585745007 90
```

Celda de código 1.7: Salida de 1.4 para un grafo con 25 vértices variando el número de colores entre 2 y 15.

1.5.3. Resultados

Los resultados obtenidos revelan diferencias significativas en el desempeño de ambas herramientas según el tamaño de la rejilla analizada; el cuadro 1.1 presenta los tiempos de ejecución observados utilizando ambos códigos:

Tamaño ($k \times k$)	Vértices	Tiempo de ejecución (NetworkX)	Tiempo de ejecución (Sage)
3×3	9	$<$ 1 segundo	$<$ 1 segundo
4×4	16	$\approx$ 4 minutos	$<$ 5 segundos
5×5	25	No completado	$<$ 10 segundos
6×6	36	No completado	$>$ 1 dia (no completado)

Cuadro 1.1: Tiempos de ejecución usando NetworkX y Sage dependiendo de la cantidad de vértices (para todos los q colores entre 2 y 15).

Por un lado, la biblioteca NetworkX demostró un rendimiento aceptable únicamente para la rejilla más pequeña evaluada. El incremento en el tiempo de cálculo de segundos a minutos entre las rejillas de 3×3 y 4×4 indica un crecimiento exponencial en la complejidad computacional que hace inviable el uso de esta herramienta para rejillas mayores.

Por otro lado, Los resultados con SageMath mostraron un comportamiento considerablemente superior para rejillas pequeñas, es decir, hasta 5×5 en tiempos razonables, proporcionando además las evaluaciones para todos los valores de q entre 2 y 15 de forma instantánea una vez obtenido el polinomio. Sin embargo, para la rejilla de 6×6 vértices, el cálculo no finalizó incluso después de varias horas de ejecución, como se puede observar en la siguiente imagen.

Figura 1.1: Figura

1.5.4. Comparación

A pesar de que solo se obtuvieron valores exactos hasta un grafo con 25 vértices entre las dos alternativas, se encontró que en la base de datos de la OEIS (1), hay valores para coloraciones de rejillas cuadradas de mayor tamaño que se presentan a continuación y se usarán para hacer la comparación con las estimaciones que sí admiten grafos con 36 o más vértices.

$k\backslash q$	2	3	4	5	6	7
6	2	101596896	3.80×10^{14}	3.29×10^{19}	2.23×10^{23}	2.86×10^{26}
7	2	41869995708	3.86×10^{19}	2.25×10^{26}	4.01×10^{31}	7.22×10^{35}
8	2	4.07×10^{13}	2.13×10^{25}	1.63×10^{34}	1.27×10^{41}	4.86×10^{46}

$k\backslash q$	8	9	10
6	1.16×10^{29}	2.02×10^{31}	1.86×10^{33}
7	2.66×10^{39}	3.1×10^{42}	1.51×10^{45}
8	2.32×10^{51}	2.42×10^{55}	8.02×10^{58}

$q\backslash k$	6	7	8
5	6	7	
3×3	9	$<$ 1 segundo	$<$ 1 segundo
4×4	16	$\approx$ 4 minutos	$<$ 5 segundos
5×5	25	No completado	$<$ 10 segundos
6×6	36	No completado	$>$ 1 dia (no completado)

Cuadro 1.2: Tiempos de ejecución usando NetworkX y Sage dependiendo de la cantidad de vértices (para todos los q colores entre 2 y 15).

1.5.5. Análisis

Los resultados obtenidos evidencian las limitaciones inherentes al cálculo exacto del polinomio cromático en grafos de tamaño moderado. La complejidad computacional de este problema es conocida por ser NP-difícil, lo cual se manifiesta claramente en el crecimiento exponencial de

los tiempos de ejecución observados. De aqui se aprecia mejor la importancia de los calculos estimados.

SageMath demostró un desempeño superior por sus optimizaciones específicas para cálculos simbólicos. Sin embargo, ambas herramientas enfrentan el mismo límite fundamental impuesto por la complejidad intrínseca del problema.

La diferencia de rendimiento entre calcular la rejilla de 4×4 en aproximadamente cuatro minutos con NetworkX versus menos de cinco segundos con SageMath representa una mejora de casi dos órdenes de magnitud. **No obstante**, esta ventaja no se traduce en la capacidad de resolver instancias significativamente mayores del problema, en este caso rejillas mayores a 6×6.

Vale la pena, sin embargo, mencionar que el límite que tiene Sage en parte se debe a la herramienta que se utilizó para correrlo: CoCalc, pues se está utilizando el plan gratuito de esta plataforma, y este tiene restricciones considerables en temas de potencia de la CPU y de almacenamiento asignados. En nuestra experiencia en la materia de Álgebra abstracta, es la mejor manera de correr códigos de este lenguaje, pero posiblemente tendría mejores rendimientos en otras plataformas o localmente.

1.6. Comparación entre valores exactos y estimados

Al comparar los valores exactos con los estimados, se calculó que el error medio entre los 3 valores de ϵ fue de $63.07\,\%$. A continuación, se muestran dos gráficas de las estimaciones de las rejillas 4×4 (1.2) y 8×8 (1.3).

Figura 1.2: Número de q-coloraciones estimado y exacto para una rejilla 4×4 usando $C = 1$.

Figura 1.3: Número de q-coloraciones estimado y exacto para una rejilla 8×8 usando $C = 1$.

Esto quiere decir que la elección de la constante $C = 1$ ciertamente no fue acertada. Posiblemente, esto se deba a la divergencia del método propuesto para estimarla y el error relativo con el valor exacto de la rejilla 3×3 con 3 colores, que llevó a iterar visualmente los valores de C en vez de usar un método más robusto. Sin embargo, se probó con $C = 4$ y $\epsilon = 1$ y el resultado tampoco fue bueno: tuvo un error de medio de $60.49\,\%$. Algunas gráficas con esta elección se pueden ver a continuación:

Figura 1.4: Número de q-coloraciones estimado y exacto para una rejilla 4×4 usando $C = 4$.

Figura 1.5: Número de q-coloraciones estimado y exacto para una rejilla 8×8 usando $C = 4$.

Por lo tanto, una elección como $C = 20$ -como se hará en el caso del modelo Hard-Core- y tomando valores de ϵ como 0.6 o 0.4, se hubiera podido, potencialmente, alcanzar un error relativo de no más de una decena porcentual. Asímismo, cuando se cuente con el conocimiento de paralelismo computacional, se podrían usar técnicas que ponen a trabajar a múltiples núcleos

del procesador, algo que no ocurrió en ninguna de las dos máquinas con las que se ejecutó (solo utilizaba uno), para hacer más factible más cómputos con parámetros más precisos.

Aproximación del Número de Configuraciones del Modelo Hard-Core

---

2.1. Introducción

Similarmente al problema de las q-coloraciones, hallar el número de posibles configuraciones factibles para el modelo Hard Core es muy complicado, puesto que el conjunto de posibles estados es de 2^{k^2} en una rejilla de tamaño k x k. Como el Hard Core no es un caso particular del problema de las q-coloraciones, no podemos aplicar el Teorema 9.1 para obtener un número esperado de pasos del Gibbs Sampler ni del número de muestras necesarias para determinar cuándo hemos llegado a la cifra aproximada de configuraciones válidas. Es por esto que vamos a experimentar para hallar unos parámetros aceptables.

2.2. Conteo exacto

Aunque pudiera parecer que el problema de hallar el número de q-coloraciones factibles de un grafo fuese mucho más difícil a priori que el de contar las factibles de Hard Core, para el primero existe una fórmula cerrada, como se vio anteriormente, pero para el segundo sorprendentemente no.

Es por esto que haremos una implementación muy básica (fuerza bruta) para hallar el número exacto de configuraciones factibles. Si bien esto no se pedía en el entregable, nos va a servir para poder saber si los resultados obtenidos usando el conteo aproximado están cerca.

La idea es hallar el número en binario al que corresponde cada posible configuración de la rejilla (hay 2^{k^2} en total) y revisar si es factible o no. Se puede correr hasta $k = 6$ en tiempos razonables pues la complejidad aumenta exponencialmente. El código a utilizar fue:

```python
import itertools

def vecinos(i, j, n):
    dirs = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for di, dj in dirs:
        ni, nj = i + di, j + dj
        if 0 <= ni < n and 0 <= nj < n:
            yield ni, nj

def es_factible(config, n):
    """Verifica si una configuración (tupla de 0s y 1s) cumple el modelo hard-core."""
    for i in range(n):
        for j in range(n):
            if config[i * n + j] == 1:
                # Si hay un ocupado, ninguno de sus vecinos puede estar ocupado
                for ni, nj in vecinos(i, j, n):
                    if config[ni * n + nj] == 1:
                        return False
    return True

def contar_configuraciones(n):
    """Cuenta todas las configuraciones factibles para un lattice n x n."""
    total_factibles = 0
    for config in itertools.product([0, 1], repeat=n * n):
        if es_factible(config, n):
            total_factibles += 1
    return total_factibles

for n in range(1, 5):
    print(f"n = {n}, configuraciones factibles = {contar_configuraciones(n)}")

```

Celda de código 2.1: Conteo exacto de configuraciones factibles de Hard Core

Y los resultados obtenidos fueron:

k	Configuraciones obtenidas
1	2
2	7
3	63
4	1234
5	55447

Cuadro 2.1: Número de configuraciones factibles en una rejilla $k \times k$

Buscando en la OEIS (1) (The On-Line Encyclopedia of Integer Sequences) vemos que corresponde a la secuencia A006506. Más valores serían

k	Configuraciones obtenidas
6	5598861
7	1280128950
8	660647962955

Cuadro 2.2: Número de configuraciones factibles en una rejilla $k \times k$

2.3. Implementaciones

Si bien comentamos que no tenemos garantías sobre el número de pasos o de simulaciones, el algoritmo que presentamos para hallar q-coloraciones será casi idéntico al que usaremos para hallar las del Hard Core.

2.3.1. Implementación del Muestreo de Gibbs

Debemos actualizar las condiciones para revisar si una configuración es factible en el Gibbs Sampler. Se muestra en detalle en el siguiente bloque de código:

```python
def gibbs_sampler_hard_core(self, iterations: int) -> "Config_HardCore":
        """
        Pick a random initial configuration and perform Gibbs' sampling to
        color the grid.
        The Gibbs sampler picks a random cell and reassigns it a color
        that does not violate the hard-core constraints.
        """

        def _valid_colors(x_idx, config: Config) -> list:
            try:
                neighbors = config.E[x_idx]
                not_valid_colors: list[str] = [self.V[neighbor] for neighbor
                in neighbors]
                # print("Not valid colors:", not_valid_colors)
                return ["w"] if 'b' in not_valid_colors else ['b', 'w']
            except BaseException as e:
                # print("Exception:", e)
                return ['b', 'w']

        result = self
        for _ in range(iterations):
            v_idx = np.random.randint(0, len(result.V))
            rand_val = np.random.rand()
            valid_color = _valid_colors(v_idx, self)
            if rand_val < 0.5:
                if valid_color == ['b', 'w']:
                    result.V[v_idx] = 'b'
                else:
                    result.V[v_idx] = 'w'
            else:
                result.V[v_idx] = 'w'


        return result
```

Celda de código 2.2: Hard Core Gibbs Sampler

2.3.2. Implementación del conteo aproximado

Una vez tenemos este algoritmo, podemos proceder con la implementación del número estimado de configuraciones factibles. Por cada simulación partimos de una configuración inicial factible (la que empieza toda la rejilla con color blanco, esto para evitar llamar la función $isFeasible()$ varias veces), se hace el muestreo de Gibbs y a la hora de añadir la nueva arista, verificamos que los colores de los vértices que une esta arista no sean ambos negros, esto para garantizar que se trata de una configuración factible en Z_{i+1} rápidamente.

```python
def estimate_n_hard_core_configurations(k: int = 9, num_simulations: int = 100,
    k_root: int = int(k**(1/2))
    total_edges: int = 2*k_root*(k_root - 1)
    estimates: list[float] = [2**k] #*add Z_0

    gibbs_iterations = num_gibbs_steps * k**1/2

    if verbose:
        print("Gibbs iterations per simulation:", gibbs_iterations)

    E_list: list[str] = []
    for c in range(k_root):
        for b in range(k_root - 1):
            E_list.append((c*k_root + b, c*k_root + b + 1))
    for c in range(k_root - 1):
        for b in range(k_root):
            E_list.append((c * k_root + b, (c + 1) * k_root + b))

    if verbose:
        print("Len E:", len(E_list))

    feasible_count: int = 0
    total_count: int = 0
    current_next_E: dict = {} #*E_{i+1}
    for i in range(0, total_edges):
        feasible_count = 0
        total_count = 0
        base_conf: Config_HardCore = Config_HardCore([], current_next_E)
        base_conf = base_conf.white_init(V_size=k)
        try:
            current_next_E[E_list[i][0]].append(E_list[i][1])
        except KeyError:
            current_next_E[E_list[i][0]] = [E_list[i][1]]
        try:
            current_next_E[E_list[i][1]].append(E_list[i][0])
        except KeyError:
            current_next_E[E_list[i][1]] = [E_list[i][0]]
        if verbose:
            print("E:", current_next_E)

        for m in range(num_simulations):
            conf: Config_HardCore = base_conf.white_init(V_size=k)
            conf = conf.gibbs_sampler_hard_core(int(gibbs_iterations))

            if (conf.V[E_list[i][0]] == 'w' and conf.V[E_list[i][1]] == 'w')
            or (conf.V[E_list[i][0]] == 'b' and conf.V[E_list[i][1]] == 'w')
            or (conf.V[E_list[i][0]] == 'w' and conf.V[E_list[i][1]] == 'b'):
                feasible_count += 1

            total_count += 1
        if verbose:
            print(f"{feasible_count} / {total_count}, for Z_{i+1} / Z_{i}")
        estimates.append(feasible_count / total_count)

    prod_estimate: float = 1
    for estimate in estimates:
        prod_estimate *= estimate

    return [estimates, prod_estimate]
```


Celda de código 2.3: Configuraciones aproximadas del Hard Core

2.4. Experimentación

Primero veamos cómo se comporta el algoritmo para rejillas pequeñas, con un número de iteraciones fijo (100) y de pasos del muestreador de Gibbs (1000).

k	Configuraciones obtenidas
1	2
2	7.452
3	67.67
4	1111.78
5	54716.50

Cuadro 2.3: Número de configuraciones factibles obtenidas con 100 muestras y 1000 pasos

2.4.1. Estimación de número de simulaciones y de pasos del Sampler

De la tabla anterior observamos que son valores bastante cercanos a los obtenidos mediante la función de fuerza bruta, lo que indica que el algoritmo da una buena estimación. Sin embargo, como no tenemos garantías del número de iteraciones ni de muestras, veamos qué pasa si las variamos para las rejillas 2×2, 3×3 y 4×4, usando el siguiente código:

```python
def estimate_hard_core_multiple_runs(k, simul, gstep, num_runs, verbose=False):
    factibles = []

    for _ in range(num_runs):
        new_value = estimate_n_hard_core_configurations(k, simul, gstep, verbose)[1]
        factibles.append(new_value)

    avg = np.average(factibles)
    std = np.std(factibles)

    return avg, std


configs_number = []
configs_uncert = []
def report_run(k, num_runs, simulations_arr = [10, 20, 50], gibbs_steps_arr):

    for simul in simulations_arr:
        configs_number_arr = []
        configs_uncert_arr = []

        for gstep in gibbs_steps_arr:
            avg, std = estimate_hard_core_multiple_runs(k, simul, gstep, num_runs)
            print(f'Dimensión: {k**(1/2)} x {k**(1/2)}')
            print(f'Numero de simulaciones: {simul}')
            print(f'Numero de pasos de Gibbs sampler: {gstep}')
            print(f'Numero de configuraciones factibles : {avg:.1f}',
            " con desviación estándar de", f'{std:.1f}')
            print()
            configs_number_arr.append(avg)
            configs_uncert_arr.append(std)

        configs_number.append(configs_number_arr)
        configs_uncert.append(configs_uncert_arr)
```

Celda de código 2.4: Variación de parámetros para el sampler

Se encarga de variar el número de iteraciones y de simulaciones, correrlo *num_runs* veces y sacar la media y la varianza del resultado obtenido, para poder estimar qué tanto varía el resultado por cada corrida del algoritmo.

$k \times k$	Número de Simluaciones	Pasos del Gibbs Sampler	Media	Desviación estándar
2 x 2	10	500	6.7	1.3
2 x 2	10	1000	7.1	1.6
2 x 2	10	5000	7.3	1.3
2 x 2	20	500	6.9	1.2
2 x 2	20	1000	7.2	0.9
2 x 2	20	5000	7.4	1.8
2 x 2	50	500	7.2	1.0
2 x 2	50	1000	7.1	1.1
2 x 2	50	5000	7.2	0.2
Promedio			7.12	1.15

Cuadro 2.4: Media y desviación estándar para diferentes simulaciones y pasos en una rejilla 2x2.

$k \times k$	Número de Simulaciones	Pasos del Gibbs Sampler	Media	Desviación estándar
3 x 3	10	500	65.5	33.3
3 x 3	10	1000	77.9	37.7
3 x 3	10	5000	42.8	29.2
3 x 3	20	500	62.6	14.9
3 x 3	20	1000	59.6	8.7
3 x 3	20	5000	55.1	10.3
3 x 3	50	500	60.5	9.3
3 x 3	50	1000	55.6	9.8
3 x 3	50	5000	65.1	11.9
Promedio			60.52	17.23

Cuadro 2.5: Media y desviación estándar para diferentes simulaciones y pasos en el caso 3×3.

$k \times k$	Número de Simulaciones	Pasos del Gibbs Sampler	Media	Desviación estándar
4 x 4	10	500	1405.0	757.9
4 x 4	10	1000	2179.2	1157.3
4 x 4	10	2000	1195.4	673.5
4 x 4	20	500	1107.4	542.3
4 x 4	20	1000	1314.6	760.3
4 x 4	20	2000	1126.7	461.7
4 x 4	50	500	1025.2	316.4
4 x 4	50	1000	1153.5	311.2
Promedio			1313.75	622.58

Cuadro 2.6: Media y desviación estándar para diferentes simulaciones y pasos en el caso 4×4.

Acá se corrió 5 veces el Gibbs Sampler dado el tamaño las simulaciones y los pasos para sacar la media y la desviación estándar. De estas tablas podemos inferir que el número de simulaciones es significativo, pero a partir de las 10 y 20. Esto tiene sentido por el Teorema Central del Límite, donde se dice que con un número de muestras mayor o igual a 30 podemos estimar bien.

Para el número de pasos del Gibbs sampler notamos que entre haya un mayor número, la desviación estándar disminuye, luego es importante que se haga el número suficiente de pasos para que se obtengan resultados más precisos.

Esto puede ser observado en las siguientes gráficas, donde se puede ver el error obtenido.

Figura 2.1: Rejilla 2x2

Figura 2.2: Rejilla 3x3

Figura 2.3: Rejilla 4x4

2.4.2. Estimación de la constante C

Como en la sección 1.2, podríamos intentar hacer un estimado para saber cuál puede ser una constante que acompañe al factor $k\log(k)$ para que nos de un buen número de simulaciones. Procediendo de manera similar, veamos cómo se comporta el número de partículas para saber si se aproxima al resultado obtenido en la tarea 1, de aproximadamente 23

C	**Media de partículas**	**Desviación estándar**	**Porcentaje**
20	2.570	0.886	0.286
10	2.400	0.883	0.267
1	2.340	1.041	0.260
0.1	2.130	0.879	0.237
0.01	3.470	1.228	0.386

Cuadro 2.7: Resumen del porcentaje de partículas para diferentes valores de C.

Es decir, no podemos utilizar C como 0.01 ya que se aleja al valor esperado. Sin embargo como vimos en anteriores secciones, entre mayor sea el número de iteraciones más se disminuye la desviación estándar, por lo que con $C = 20$ podemos obtener resultados satisfactorios.

Con $C = 20$, tenemos que:

Dimensión	Pasos Gibbs	Configuraciones
2x2	40.0	6.3
3x3	95.09775	54.6
4x4	160.0	1525.4
5x5	232.19281	76079.3
6x6	310.19550	6087479.6
7x7	393.02969	694790374.7
8x8	480.0	1096931362136.8
9x9	570.58650	579117328461176.1
10x10	664.38562	3391324208907807232.0
11x11	761.07496	29080627771920185556992.0
12x12	860.39100	275773265023135350517661696.0
13x13	962.11433	1222724543961935304095885688320.0
14x14	1066.05938	463578449569656223879647008840797388.0
15x15	1172.06718	27874711897936813552121326032107864147886 08.0

Cuadro 2.8: Número estimado de configuraciones factibles y pasos de Gibbs usados por dimensión.

2.4.3. Conteo aproximado vs Conteo exacto

Teniendo estos datos, decidimos correr 2 veces para cada tamaño del grafo desde 2×2 hasta 10×10 con 30 simulaciones y $10000 \times k$ iteraciones, para que entre mayor sea la rejilla más preciso sea, al tener más iteraciones. Sin embargo, en aproximadamente 7 horas solo pudo concluir hasta el tamaño 7×7, así que por cuestiones de tiempo, nos tocó hacer la simulación hasta ese tamaño. Los resultados obtenidos son los siguientes:

Dimensión $k \times k$	# de Simulaciones	Pasos del Gibbs Sampler	Media	Desviación estándar
2 x 2	30	20000	6.7	0.3
3 x 3	30	30000	72.8	6.8
4 x 4	30	40000	1444.8	406.6
5 x 5	30	50000	64295.9	3028.8
6 x 6	30	60000	5525527.3	327075.2
7 x 7	30	70000	1443558047.5	692371490.3

Cuadro 2.9: Número de configuraciones factibles aproximadas mediante MCMC para distintos tamaños de rejilla.

Visualizados:

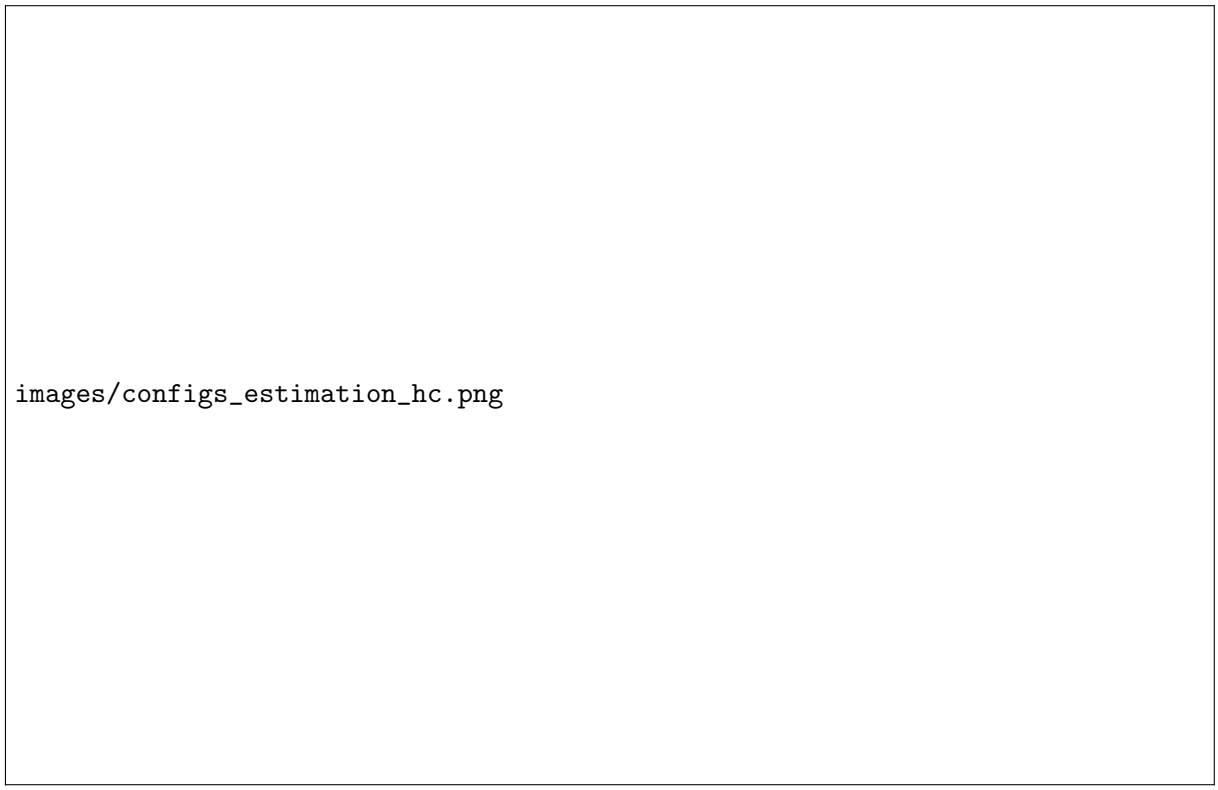


Figura 2.4: Estimaciones vs tamaño

Sin embargo queríamos tener una estimación para valores grandes de la rejilla por lo que completamos los valores de 8×8 hasta 17×17 con 10000 iteraciones para obtener un estimado.

Dimensión	**Configuraciones**
2x2	6.7
3x3	72.8
4x4	1444.8
5x5	64295.9
6x6	5525527.3
7x7	1443558047.5
8x8	951286414073.8
9x9	189030653966917.5
10x10	691599757640581120.0
11x11	2373482165990338330624.0
12x12	2819661581716950925233368.0
13x13	4311425132096001172574641848320.0
14x14	1331661159409247708564178020702617600.0
15x15	13795976302719700716009855382427361345536.0
16x16	434244581903792939216808427676316219608557158 4.0
17x17	365616305778792262207318430094127515908747925913 6000.0

Cuadro 2.10: Número estimado de configuraciones factibles para distintas dimensiones de la red.

Figura 2.5: Estimaciones vs tamaño

Por lo que, si bien entre más iteraciones pongamos, mejor la exactitud, vemos que igual se comporta relativamente similar a los valores previos.

Para finalizar, vamos a comparar con los valores exactos conocidos (hasta 15x15):

Figura 2.6: Estimaciones vs tamaño

Ahora, para el número de iteraciones siendo calculado con $C = 20$, también obtenemos resultados similares:

Figura 2.7: Estimaciones vs tamaño

2.5. Conclusiones

A pesar de no tener las garantías del Teorema 9.1, experimentalmente hemos comprobado que el número de pasos de Gibbs Sampler necesarios también se comporta como un $C \times k \log(k)$ donde vimos que con $C = 20$ se obtienen resultados muy satisfactorios, ya que con menos simulaciones hemos obtenido una precisión bastante óptima a los valores exactos computados con fuerza bruta o hallándolos en la secuencia A006506.

Nos queda de lección que el número de pasos hallados con la fórmula exacta es una cota bastante alta y que experimentalmente no da más utilidad puesto que cuando se hizo la comparativa en la gráfica 2.5, esas se hicieron con demasiadas iteraciones para terminar teniendo un comportamiento muy similar al hallado en la gráfica 2.7.

Anexo

A continuación se muestra una tabla con el porcentaje de partículas promedio en grafos $G = (V, E)$ donde $|V| = k \times k$:

$k \times k$	Porcentaje de $X_{1,000}$	Porcentaje de $X_{10,000}$	Porcentaje promedio
9	0.221	0.220	0.221
16	0.241	0.240	0.241
25	0.227	0.231	0.229
36	0.231	0.234	0.233
49	0.231	0.231	0.231
64	0.228	0.230	0.229
81	0.229	0.230	0.230
100	0.229	0.230	0.230
121	0.229	0.229	0.229
144	0.227	0.229	0.228
169	0.228	0.228	0.228
196	0.229	0.228	0.228
225	0.228	0.228	0.228
256	0.228	0.228	0.228
289	0.228	0.228	0.228
324	0.227	0.228	0.228
361	0.227	0.228	0.228
400	0.228	0.228	0.228
Promedio	0.229	0.229	0.229

Cuadro 3.1: Porcentaje (i.e., $\frac{\mu}{k^2}$ donde μ es la media de partículas para cada k) de la media de partículas para una rejilla $k \times k$ en el modelo hard-core tras $1,000$ y $10,000$ iteraciones, y el promedio de ambas (es decir, su suma dividida entre 2).

Bibliografía y referencias

1. The On-Line Encyclopedia of Integer Sequences (OEIS). Colorings of grid graphs. url: link. (acceso: 12.11.2025).
2. Mark Jerrum. A very simple algorithm for estimating the number of k-colourings of a low-degree graph. url: link. (acceso: 12.11.2025).
3. Sarah Miracle and Dana Randall. Algorithms to approximately count and sample conforming colorings of graphs. url: link. (acceso: 12.11.2025).