Tarea 3 Cadenas de Markov

Autores:

- Daniel Alejandro García Hernández
- David Camilo Cortes Salazar

En este notebook se encuentra una implementación de algunos resultados vistos en clase para el Gibbs Sampler aplicado a Hard-core y q-colorings.

Las librerías necesarias para ejectuar el código son:

In [1]:
```
import numpy as np
import random
import matplotlib.pyplot as plt
import pandas as pd
from copy import deepcopy
from pprint import pprint
```

Modelo de Ising usando MCMC y Algoritmo de Propp - Wilson

Muestras del modelo de Ising con inverso de temperatura usando MCMC

Iniciamos creando la grilla del módelo. Para esto, creamos una matriz kxk de dimensión 2, apoyándonos en la libreria Numpy. Las dimensiones del grafo cumplen $10 \leq k \leq 20$.

Por otro lado, la energía del modelo, bajo campo libre y sin interacciones de orden superior, viene dada por el Hamiltoniano $H(\eta)$, que se expresa como:

$$H(\eta) = -\sum_{x \sim y} \eta_x \eta_y$$

D0onde η es la configuracion actual del sistema y η_x, η_y son la posicion de dos nodos de η tal que x y y estan conectados.

Con esto en mente creamos una funcion que calcula la energia del sistema dada una configuracion η del modelo.

In [2]:
```
def calcular_energia(G):
    """
    Calcula la energia de la configuracion dada. Primero se suman enlaces horizontales, luego verticales
    """
    energia = 0
    k = len(G)

    ## Suma sobre enlaces horizontales
    for ii in range(k):
        for jj in range(k-1):
            energia += G[ii,jj] * G[ii,jj+1]

    ## Suma sobre enlaces verticales
    for ii in range(k-1):
        for jj in range(k):
            energia += G[ii,jj] * G[ii+1,jj]

    return -1 * energia


G = np.random.choice([-1, 1], size=(4, 4))
print("Grilla del modelo de Ising:")
print(G)
```

```
Grilla del modelo de Ising:
[[-1 -1 -1 -1]
 [ 1 -1  1 -1]
 [-1  1  1 -1]
 [-1 -1  1  1]]
```

Ahora usemos un Gibbs sampler para implementar el método de Montecarlo en nuestro problema. Este funcionará de la siguiente manera:

- Seleccione un vertice al azar \~x de la configuración actual η.
- Cambie el valor del vertice escogido. Con esto se obtiene una nueva configuración $\hat{\eta}$ del sistema.
- Calcule la diferencia de energía de la configuración η.

- Calcule la probabilidad de cambio de η utilizando la probabilidad de distribución del vértice $\sim x$:

$$\frac{e^{2\beta(k_+(\sim x,\eta)-k_-(\sim x,\eta))}}{e^{2\beta(k_+(\sim x,\eta)-k_-(\sim x,\eta))}+1}$$

- Genere un numero aleatorio uniforme u entre 0 y 1
- Si $u < \Delta\pi$, acepta la nueva configuracion. Si no, mantenemos con la configuracion η.
- Repetimos el algoritmo hasta completar el número de iteraciones propuestas.

Note que el calculo de la energía deja de aparecer directamenete en el algoritmo, y aparece la diferencia entre la acumulación de cargas positivas y negativas alrededor de un vertice dado.

A continuación, creamos una nueva función para el cálculo de esta acumulación y otra para la probabiliadad de cambio:

In [3]:
```
def calcular_factor(G, x, y):
    """
    Calcula la diferencia de la energia al cambiar el valor de spín en un nodo (electrón) dado
    """
    energia = 0
    k = len(G)

    if x != 0:
        energia += G[x-1, y]
    if x != k-1:
        energia += G[x+1, y]
    if y != 0:
        energia += G[x, y-1]
    if y != k-1:
        energia += G[x, y+1]
    return energia


def calcular_probabilidad(beta, factor):
    """
    Calcula la probabilidad de cambio de la nueva configuración
    """
    e = np.exp(2 * beta * factor)
    return e / (e+1)
```

Y con esto creamos el Gibbs sampler para nuestro modelo:

In [4]:
```
def gibbs_sampler_mcmc(G, beta, iteraciones):

    k = len(G)

    for _ in range(iteraciones):

        # Selecciona spin aleatorio
        x = random.randint(0, k-1)
        y = random.randint(0, k-1)

        # Calcula la energia de la configuracion actual y siguiente segun el spin dado
        diferencia_energia = calcular_factor(G, x, y)

        # Halla la probabilidad de transición
        prob_cambio = calcular_probabilidad(beta, diferencia_energia)

        # Decidir si hacer o no transición
        if np.random.rand() < prob_cambio:
            G[x, y] = 1
        else:
            G[x, y] = -1

    return G
```

Ahora solo nos queda probar el algoritmo con los valores de beta y pasos del gibbs sampler que definimos al inicio para obtener el numero de muestras deseadas.

Adicionalmente, creamos una lista con los β que vamos a evaluar y otra con los diferentes números de pasos con los que ejecturemos el algoritmo. Adicional a los β sugeridos, vamos a incluir β_c, el parámetro crítico de Onsager.

In [5]:
```
onsager_critical_value = 0.5 * np.log(1 + np.sqrt(2))

k = 10
beta_list_mcmcm = [0, 0.1, 0.2, 0.3, 0.4, onsager_critical_value, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
steps_list = [10**3, 10**4, 10**5]
n_muestras = 100
```

```
configuraciones_mcmc = {}

# Loop principal
for step in steps_list: # Iterar sobre los posibles números de pasos

    configuraciones_mcmc[step] = {key: [] for key in beta_list_mcmcm}

    for beta in beta_list_mcmcm: # Iterar sobre los posibles beta

        for _ in range(n_muestras): # Numero de muestras a guardar
            G = np.random.choice([-1, 1], size=(k, k))  # Configuración inicial aleatoria
            configuraciones_mcmc[step][beta].append(gibbs_sampler_mcmc(G, beta, step).copy())  # Guarda la confi
        print(f"Guardado {n_muestras} muestras con X = {step} para beta = {beta}")
```

```
Guardado 100 muestras con X = 1000 para beta = 0
Guardado 100 muestras con X = 1000 para beta = 0.1
Guardado 100 muestras con X = 1000 para beta = 0.2
Guardado 100 muestras con X = 1000 para beta = 0.3
Guardado 100 muestras con X = 1000 para beta = 0.4
Guardado 100 muestras con X = 1000 para beta = 0.44068679350977147
Guardado 100 muestras con X = 1000 para beta = 0.5
Guardado 100 muestras con X = 1000 para beta = 0.6
Guardado 100 muestras con X = 1000 para beta = 0.7
Guardado 100 muestras con X = 1000 para beta = 0.8
Guardado 100 muestras con X = 1000 para beta = 0.9
Guardado 100 muestras con X = 1000 para beta = 1
Guardado 100 muestras con X = 10000 para beta = 0
Guardado 100 muestras con X = 10000 para beta = 0.1
Guardado 100 muestras con X = 10000 para beta = 0.2
Guardado 100 muestras con X = 10000 para beta = 0.3
Guardado 100 muestras con X = 10000 para beta = 0.4
Guardado 100 muestras con X = 10000 para beta = 0.44068679350977147
Guardado 100 muestras con X = 10000 para beta = 0.5
Guardado 100 muestras con X = 10000 para beta = 0.6
Guardado 100 muestras con X = 10000 para beta = 0.7
Guardado 100 muestras con X = 10000 para beta = 0.8
Guardado 100 muestras con X = 10000 para beta = 0.9
Guardado 100 muestras con X = 10000 para beta = 1
Guardado 100 muestras con X = 100000 para beta = 0
Guardado 100 muestras con X = 100000 para beta = 0.1
Guardado 100 muestras con X = 100000 para beta = 0.2
Guardado 100 muestras con X = 100000 para beta = 0.3
Guardado 100 muestras con X = 100000 para beta = 0.4
Guardado 100 muestras con X = 100000 para beta = 0.44068679350977147
Guardado 100 muestras con X = 100000 para beta = 0.5
Guardado 100 muestras con X = 100000 para beta = 0.6
Guardado 100 muestras con X = 100000 para beta = 0.7
Guardado 100 muestras con X = 100000 para beta = 0.8
Guardado 100 muestras con X = 100000 para beta = 0.9
Guardado 100 muestras con X = 100000 para beta = 1
```

Notamos que, en términos de tiempo de cómputo, el Gibss sampler tiene tiempo constante para todo β, y escala linealmente con el valor del número de pasos. Esto hace que hacer muestreo del sistema en equilibrio sea demorado, pues se hace necesario un gran número de pasos

In [6]:
```
print("\nVisualizacion de algunas de las configuraciones de las muestras:\n")
print("Beta = 0")
pprint(configuraciones_mcmc[100000][0][3])

print()
print("Beta = 1")
pprint(configuraciones_mcmc[100000][1][2])
```

```
Visualizacion de algunas de las configuraciones de las muestras:

Beta = 0
array([[-1,  1, -1,  1, -1,  1,  1, -1, -1, -1],
       [-1,  1,  1, -1, -1,  1,  1, -1,  1,  1],
       [ 1, -1, -1,  1,  1, -1,  1, -1,  1, -1],
       [-1, -1, -1, -1,  1, -1,  1,  1,  1, -1],
       [ 1, -1,  1,  1, -1, -1,  1,  1,  1, -1],
       [-1,  1,  1, -1, -1, -1, -1, -1,  1,  1],
       [ 1,  1, -1, -1,  1,  1,  1,  1, -1,  1],
       [ 1,  1,  1, -1,  1, -1,  1, -1, -1, -1],
       [-1, -1,  1, -1, -1, -1, -1,  1,  1,  1],
       [-1, -1, -1,  1, -1, -1, -1, -1, -1, -1]])

Beta = 1
array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])
```

Cuando imprimimos algunas de estas configuraciones, vemos que:

- Para $\beta = 0$ (altas temperaturas), la organización de spins parece ser de tipo i.i.d.: aproximadamente la mitad está en estado +1, y otra mitad en estado -1, sin ningún tipo de preferencia espacial.
- Para $\beta = 1$ (bajas temperaturas), la organización de spins es a) todos en estado +1, o b) todos en estado -1.

Esto está de acuerdo con lo predicho por la teoría.

Muestras del modelo de Ising con inverso de temperatura usando Prop-Wilson

Usaremos los mismos valores tomados anteriormente para hacer una comparación mas adelante entre las muestras obtenidas con el algoritmo de de Propp-Wilson y MCMC.

Las funciones de acumulación y probabilidad son las mismas, aquí cambia el método de aplicar el Gibbs sampler. Usaremos sadwiching para la implementacion de el algoritmo de Propp-Wilson sobre las cofiguraciones η_{min} (Todos -1) y η_{max} (Todos 1). Y para esto, nos hace falta:

- Una secuencia creciente de números enteros positivos: $N = \{N_1, N_2, \dots\}$.
- Una secuencia de números aleatorios uniformemente distribuidos en $[0, 1]$: $U = \{U_0, U_{-1}, \dots\}$.
- Dos secuencias de coordenadas aleatorias uniformemente distribuidos en $[0, k-1]$: $\hat{U} = \{\hat{U}_0, \hat{U}_{-1}, \dots\}$ y $\hat{U} = \{\hat{U}_0^{'}, \hat{U}_{-1}^{'}, \dots\}$.

La secuencia N nos servira para seleccionar los elementos de las sucesiones aleatorias. Tomaremos $N_n = 2^n$.

La secuencia U dicatará los valores a comparar con las probabilidades de cambio de los sistemas η_{min} y η_{max}.

Las secuencias $\hat{U}$ y $\hat{U}^{'}$ serán las encargadas de seleccionar las cordenadas $\hat{x}, \hat{y}$ de los vértices a evaluar.

Para esto, creemos la siguiente clase:

In [7]:
```
class Random_sequence:
    def __init__(self, k):
        self.k = k
        self.N = 0 # indice de la sucesion de numeros creciente (N)
        self.U_float = np.zeros((0,), dtype=float)
        self.U_int = np.zeros((0,2), dtype=int)

    def update_random(self, next_N):
        self.U_float.resize((next_N,))
        self.U_int.resize((next_N,2))
        self.U_float[self.N:] = np.random.random((next_N - self.N,))
        self.U_int[self.N:] = np.random.randint(0, self.k, size=(next_N - self.N,2))
        self.N = next_N
```

Esta clase funciona nos asegura generar números aleatorios nuevos en cada paso del Gibbs sampler y seleccionar los correspondientes según la secuencia dada. Además, estos valores se guardan en los arreglos creados y nos permiten ver el registro historico de elementos aleatorios seleccionados.

Ahora podemos crear nuestro Gibbs Sampler usando Propp-Wilson:

In [8]:
```
def gibbs_sampler_pw(k, beta):

    N = 1 # indice de la sucesion de numeros creciente
    rand_sequence = Random_sequence(k) # Clase de generacion y memoria de numeros aleatorios usados
    rand_sequence.update_random(N) # genera primeros numeros aleatorios

    #coalecencia
    coales_time = 0

    # Configuraciones extremas
    G_max = np.ones((k, k))
    G_min = -np.ones((k, k))

    while True:

        top_size = rand_sequence.U_float.size

        for i in range(top_size):
            coales_time += 1
            m = top_size - i - 1 # indice de la secuencia aleatoria

            x = rand_sequence.U_int[m, 0] # Numero aleatorio dado por la secuencia de enteros
            y = rand_sequence.U_int[m, 1] # Numero aleatorio dado por la secuencia de enteros

            # Sandwiching

            delta_energia_min = calcular_factor(G_min, x, y)
            delta_energia_max = calcular_factor(G_max, x, y)

            prob_cambio_max = calcular_probabilidad(beta, delta_energia_max)
            prob_cambio_min = calcular_probabilidad(beta, delta_energia_min)

            rand_to_prob = rand_sequence.U_float[m] # Numero aleatorio dado por la secuencia de flotantes

            # Condicion de Cambio G1\n",
            if rand_to_prob < prob_cambio_max:
                G_max[x, y] = 1
            else:
                G_max[x, y] = -1

            # Condicion de Cambio G2\n",
            if rand_to_prob < prob_cambio_min:
                G_min[x, y] = 1
            else:
                G_min[x, y] = -1

        # Verifica coalecencia\n",
        if np.array_equal(G_min, G_max):
            return G_min, coales_time
        else:
            N = max(N+1, N*2) # sucesion de potencias de 2
            rand_sequence.update_random(N) # genera nuevos numeros aleatorios
```

Ahora generamos las muestras deseadas con los valores de beta dados.

Es necesario notar que, según la nota 43 del capítulo 11 del libro de Haggstrom, el tiempo de simulación para el algoritmo de Prop-Wilson aumenta exponencialmente para valores de Beta crítico que superen el valor crítico de Onsager $\beta_c = \frac{1}{2}\log(1+\sqrt{2}) \approx 0.441$. Por esta razón, solo vamos tomar muestras con este algoritmo para valores de $\beta \leq 0.6$. Experimentalmente intentamos hallar muestras para valores $\beta > 0.6$, pero con $\beta = 0.6$ el tiempo de ejecución fue enorme.

- $\beta = 0.1, 0.2, 0.3, 0.4$ se simulan en cuestión de pocos segundos.
- $\beta = 0.5$ se simula en cuestión de ≈ 45 segundos.
- $\beta = 0.6$ tomó 23 minutos.

Teniendo en cuenta el aumento exponencial mencionado en el libro, no pudimos seguir simulando.

In [9]:
```
configuraciones_pw = {}
tiempos_coalescencia = {}

k = 10
beta_list_pw = [0, 0.1, 0.2, 0.3, 0.4, onsager_critical_value, 0.5, 0.6]
n_muestras = 100


# Loop principal\n",
for beta in beta_list_pw: # Beta a utilizar
    configuraciones_pw[beta] = []
    tiempos_coalescencia[beta] = []
    for _ in range(n_muestras): # Numero de muestras a guardar
        data = gibbs_sampler_pw(k, beta)
```

```
        configuraciones_pw[beta].append(data[0].copy())  # Guarda la configuracion final
        tiempos_coalescencia[beta].append(data[1])  # Guarda el tiempo de coalecencia
    print(f"Guardado {n_muestras} muestras para beta = {beta}")
```

```
Guardado 100 muestras para beta = 0
Guardado 100 muestras para beta = 0.1
Guardado 100 muestras para beta = 0.2
Guardado 100 muestras para beta = 0.3
Guardado 100 muestras para beta = 0.4
Guardado 100 muestras para beta = 0.44068679350977147
Guardado 100 muestras para beta = 0.5
Guardado 100 muestras para beta = 0.6
```

Notamos que, en términos de tiempo de cómputo, Propp-Wilson es extremadamente eficiente siempre que $\beta \leq \beta_c$. Una vez se pasa este umbral, el tiempo de cómputo aumenta enormemente, y para β muy grande, Propp-Wilson se vuelve un algoritmo muy ineficiente. Por lo tanto, es situaciones reales, lo ideal sería usar Propp-Wilson para $\beta \leq \beta_c$, y a partir de este punto empezar a usar el algoritmo de Monte Carlo descrito anteriormente

Estimacion valor esperado de magnetización en modelo de Ising

Con las muestras que tenemos, vamos a comparar la magnetización del sistema para las muestras generadas con MCMCM y Propp-Wilson.

Para esto, primero tenemos que calcular la magnetización de cada una de las muestras encontradas. Para esto, usamos la fórmula:

$$M(\eta) = \left| \frac{1}{|V_k|} \sum_{x \in V_k} \eta_x \right|$$

En nuestro caso tenemos $|V_k| = k^2$.

Por otra parte, el valor absoluto se añade por una cuestión física. Suponga que se tienen 3 configuraciones con las siguientes características:

1. Todos los spins hacia arriba (+1).
2. Todos los sipins hacia abajo (-1).
3. De forma aleatoria y desordenada, la mitad de los spins están hacia arriba (+1), y la otra mitad hacia abajo (-1).

El tercer caso representa un sistema completamente desmagnetizado, pues los spins no siguen ningún patrón coherente. Por su parte, las dos primeras configuraciones representan un estado con alta magnetización, dado que los spins se encuentran perfectamente ordenados.

Esto justifica la existencia del valor esperado, pues tiene en cuenta la simetría física del sistema presente entre la configuración 1. y 2.

Finalmente, otro argumento físico a favor de esta valor absoluto es el comportamiento del sistema visto anteriormente para distintos valores de β:

- Para $\beta = 0$ (altas temperaturas), la organización de spins parece ser de tipo i.i.d.: aproximadamente la mitad está en estado +1, y otra mitad en estado -1, sin ningún tipo de preferencia espacial.
- Para $\beta = 1$ (bajas temperaturas), la organización de spins es a) todos en estado +1, o b) todos en estado -1.

Así, tenemos la siguiente función:

In [10]:
```
def calcular_magnetizacion(k, configuraciones, beta_list):
    magnetizacion_list = {}
    for beta in beta_list:
        magnetizacion_list[beta] = []
        for config in configuraciones[beta]:
            magnetizacion_list[beta].append(abs(np.sum(config)/(k**2)))
    return magnetizacion_list

magnetismo_mcmc_list = {} # Lista por numero de pasos de magnetizacion en muestras de MCMC
for step in steps_list:
    magnetismo_mcmc_list[step] = calcular_magnetizacion(k, configuraciones_mcmc[step], beta_list_mcmcm) # Magne
    print(f"Hallada la magnetizacion de MCMCM para step {step}")

magnetismo_pw = calcular_magnetizacion(k, configuraciones_pw, beta_list_pw) # Magnetizacion muestras Prop-Wilso
print(f"Hallada la magnetizacion de Propp-Wilson")
```

```
Hallada la magnetizacion de MCMCM para step 1000
Hallada la magnetizacion de MCMCM para step 10000
Hallada la magnetizacion de MCMCM para step 100000
Hallada la magnetizacion de Propp-Wilson
```

Ahora calculamos el valor esperado de la magnetización para configuración mediante la siguiente expresión:

$$E[M(\eta)] \approx \frac{1}{X}\sum_{k=1}^{X} M(\eta_k)$$

donde X es el numero de muestras, que en nuestro caso es 100. Así, definimos la siguiente función:

In [11]:
```
def valor_esperado(magnetismo, beta_list):
    vesperado_list = {}
    for beta in beta_list:
        vesperado_list[beta] = np.average(magnetismo[beta])

    return vesperado_list

vesperado_mcmc_list = {} # Lista por numero de pasos de valor esperado en mustras de mcmc
for step in steps_list:
    vesperado_mcmc_list[step] = valor_esperado(magnetismo_mcmc_list[step], beta_list_mcmcm) # Valor esperado de


vesperado_pw = valor_esperado(magnetismo_pw, beta_list_pw) # Valor esperado de magnetizacion en P-W

for step in steps_list:
    print(f"Valor esperado usando MCMC con X = {step}:")
    pprint(vesperado_mcmc_list[step])
    print()

print("Valor esperado usando PW: ")
pprint(vesperado_pw)
```

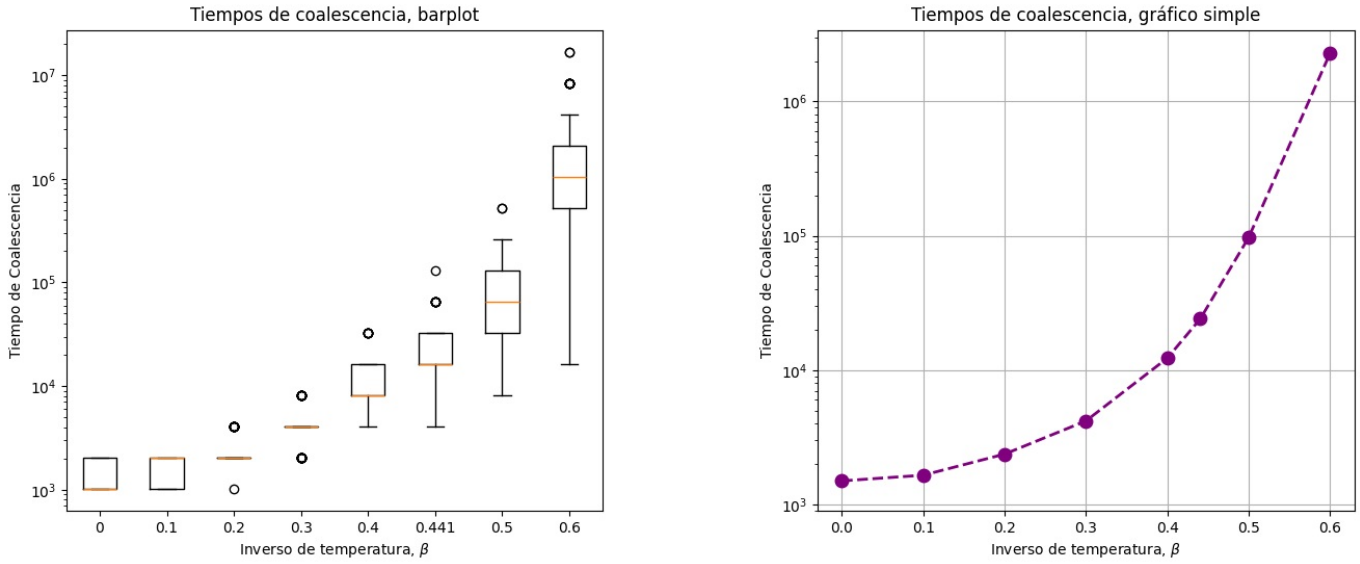
```
Valor esperado usando MCMC con X = 1000:
{0: 0.08559999999999998,
 0.1: 0.0928,
 0.2: 0.1326,
 0.3: 0.17699999999999996,
 0.4: 0.262,
 0.44068679350977147: 0.28,
 0.5: 0.32480000000000003,
 0.6: 0.412,
 0.7: 0.3889999999999999,
 0.8: 0.4166,
 0.9: 0.4224,
 1: 0.4278}

Valor esperado usando MCMC con X = 10000:
{0: 0.0748,
 0.1: 0.09659999999999998,
 0.2: 0.12200000000000001,
 0.3: 0.18580000000000002,
 0.4: 0.32500000000000007,
 0.44068679350977147: 0.3986,
 0.5: 0.6322000000000001,
 0.6: 0.7806000000000001,
 0.7: 0.85,
 0.8: 0.8930000000000001,
 0.9: 0.8850000000000001,
 1: 0.8504}

Valor esperado usando MCMC con X = 100000:
{0: 0.0814,
 0.1: 0.1026,
 0.2: 0.12420000000000002,
 0.3: 0.19,
 0.4: 0.2846,
 0.44068679350977147: 0.48840000000000006,
 0.5: 0.6448,
 0.6: 0.8825999999999999,
 0.7: 0.9539999999999997,
 0.8: 0.9808000000000001,
 0.9: 0.9914000000000001,
 1: 0.9948}

Valor esperado usando PW:
{0: 0.09119999999999999,
 0.1: 0.09559999999999996,
 0.2: 0.125,
 0.3: 0.20379999999999998,
 0.4: 0.3486,
 0.44068679350977147: 0.49,
 0.5: 0.6616000000000001,
 0.6: 0.8866}
```

Reporte

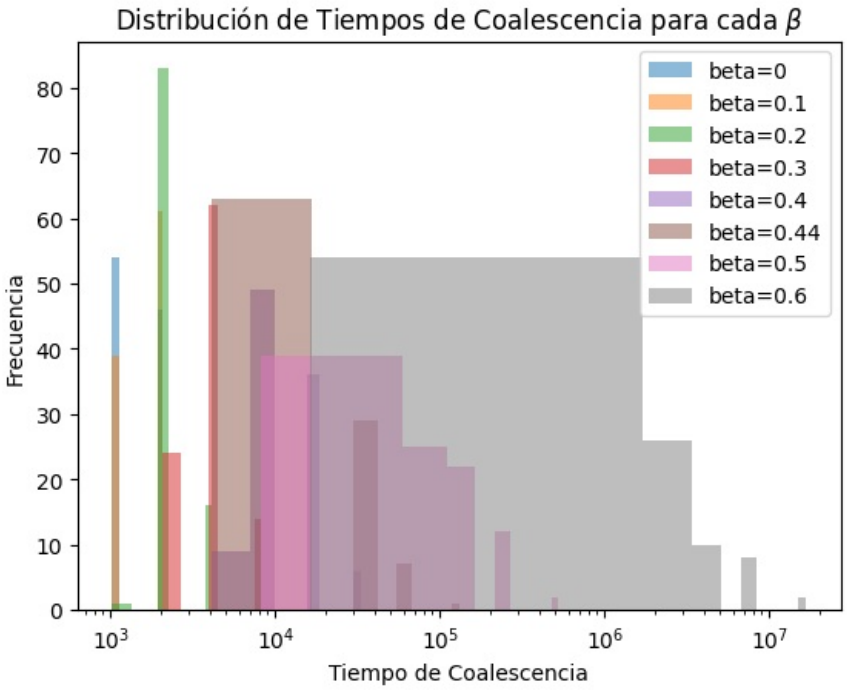
Tiempo de coalescencia

Graficamos los tiempos de Coalescencia en Propp-Wilson para cada valor de β

In [21]:
```
time_avg_list = []
time_std_list = []

for beta in beta_list_pw:
    time_avg_list.append(np.average(tiempos_coalescencia[beta]))
    time_std_list.append(np.std(tiempos_coalescencia[beta]))


## Begin plot
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))
plt.subplots_adjust(wspace=0.4)


## Barplot de la izquierda
aux_beta_list_pw = list(beta_list_pw) # Betas de PW aproximados a 3 cifras decimales
for ii, beta in enumerate(aux_beta_list_pw):
    aux_beta_list_pw[ii] = (round(beta, 3))

ax1.set_title('Tiempos de coalescencia, barplot')
ax1.boxplot([tiempos_coalescencia[beta] for beta in beta_list_pw], tick_labels=aux_beta_list_pw)
ax1.set_yscale('log')
ax1.set_xlabel(r'Inverso de temperatura, $\beta$')
ax1.set_ylabel('Tiempo de Coalescencia')


## Gráfico simple de la derecha
ax2.plot(beta_list_pw, time_avg_list, color='purple', marker='o', linestyle='--', markersize=9, linewidth=2)
ax2.set_title('Tiempos de coalescencia, gráfico simple')
ax2.set_yscale('log')
ax2.set_xlabel(r'Inverso de temperatura, $\beta$')
ax2.set_ylabel('Tiempo de Coalescencia')
ax2.set_ylim([0.9*1E3, np.max(time_avg_list)*1.5])
ax2.grid()


plt.show()
```



In [27]:
```
fig, ax = plt.subplots()

for beta, times in tiempos_coalescencia.items():
    ax.hist(times, bins=10, alpha=0.5, label=f'beta={round(beta, 2)}')

ax.set_title(r'Distribución de Tiempos de Coalescencia para cada $\beta$')
ax.set_xlabel('Tiempo de Coalescencia')
ax.set_ylabel('Frecuencia')

ax.set_xscale('log')

ax.legend()

plt.show()
```

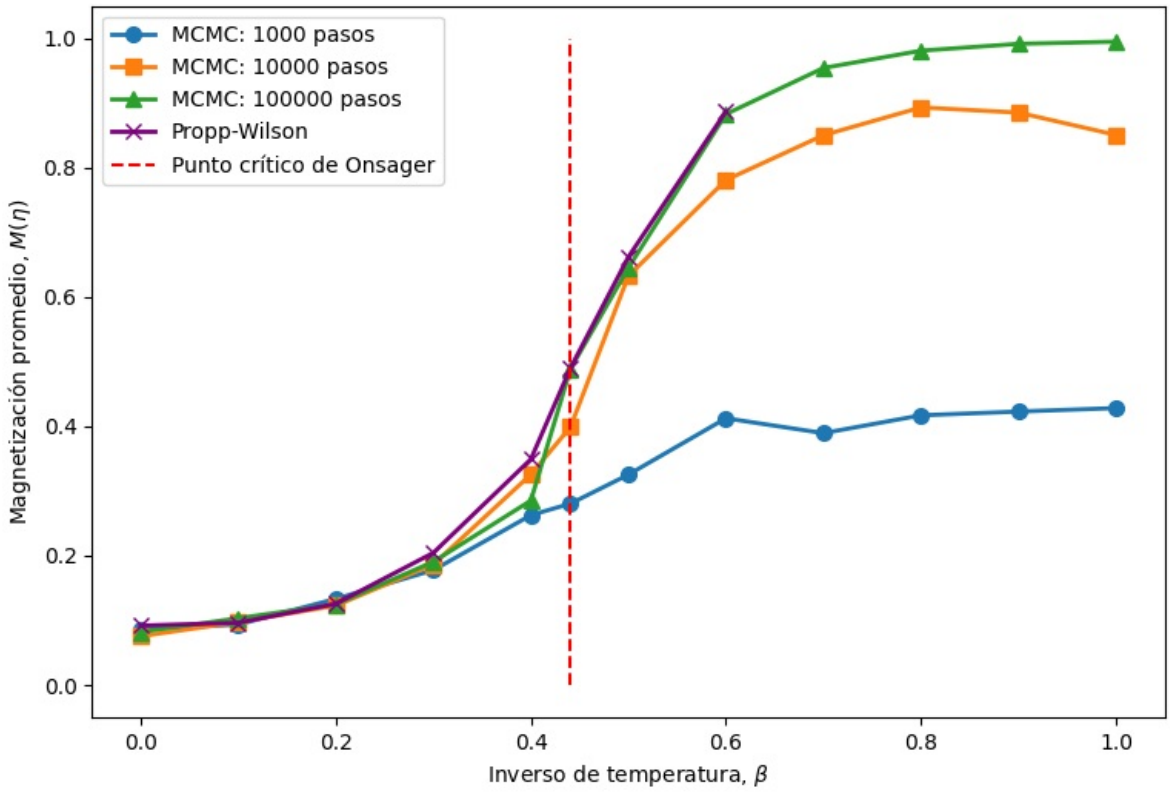


Notamos que el tiempo de coalescencia crece de forma exponencial y aumentan la dispersión en su distribución de frecuencia una vez se supera el valor de β_c.

Siguiendo esta tendencia se espera que, por ejemplo, para $\beta = 0.7$ el tiempo de coalescencia sea del orden de 4×10^7. Con esto en mente, y dado que la simulación de $\beta = 0.6$ tomó 23 minutos, se espera que la de $\beta = 0.7$ tome $\sim$ 400 minutos = 6.7 horas.

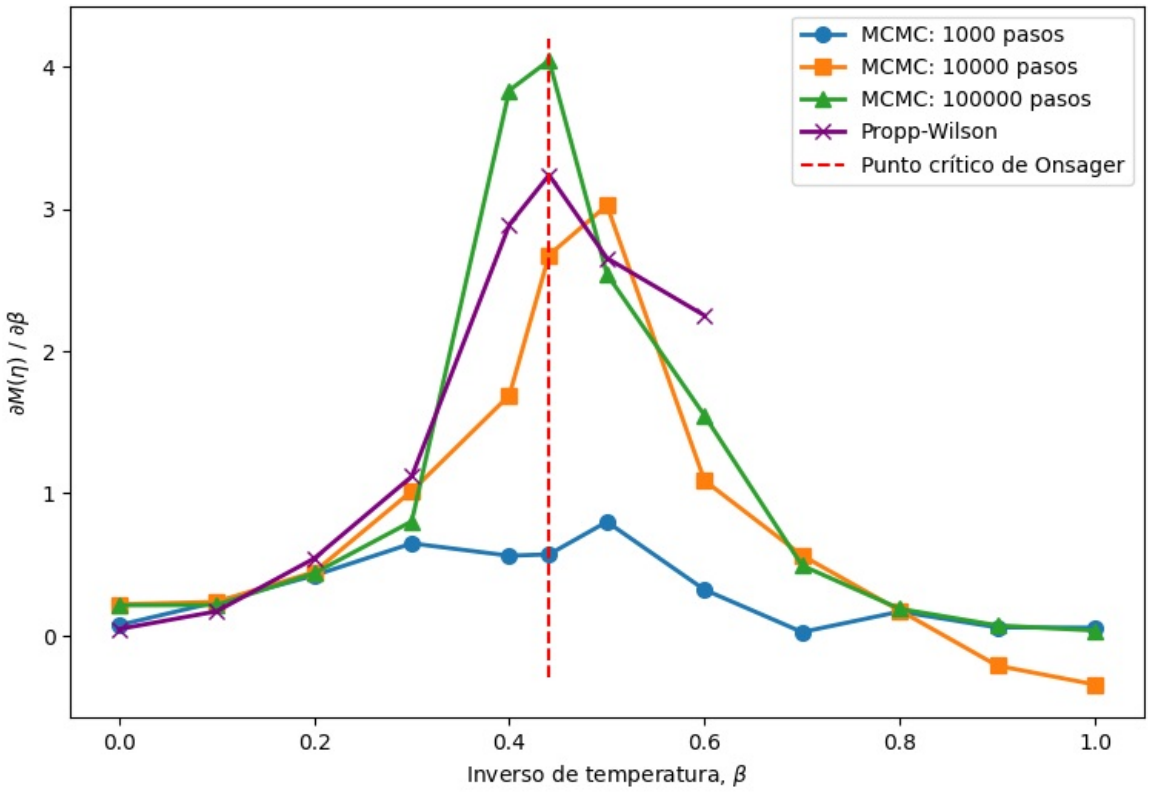
Transición de fase

In [28]:
```
marker_list = ['o', 's', '^', 'd']


plt.figure(figsize=(9, 6))

## Plot de MCMC
for ii, step in enumerate(steps_list):
    magnetizaciones = list(vesperado_mcmc_list[step].values())
    plt.plot(beta_list_mcmcm, magnetizaciones, label=f'MCMC: {step} pasos', marker=marker_list[ii], linestyle='


## Plot de Propp-Wilson
plt.plot(beta_list_pw, list(vesperado_pw.values()), label='Propp-Wilson', color='purple', marker='x', linestyle


## Onsager critical
plt.vlines(onsager_critical_value, 0, 1, linestyles='dashed', label='Punto crítico de Onsager', color='red')



## Propiedades de la gráfica
plt.xlabel(r'Inverso de temperatura, $\beta$')
plt.ylabel(r'Magnetización promedio, $M ( \eta ) $')
plt.legend()
#plt.grid()

plt.show()
```



Una forma comúnmente empleada para identificar transiciones de fase, así como sus características, es graficar la derivada del parámetro (ver, por ejemplo), en la cual se espera un pico muy notorio en el valor crítico donde se da la transición de fase. Así, grafiquemos la derivada numérica de la magnetización del sistema respecto a β.

In [36]:
```
marker_list = ['o', 's', '^', 'd']


plt.figure(figsize=(9, 6))

## Plot de MCMC
for ii, step in enumerate(steps_list):
    x = beta_list_mcmcm
    y = list(vesperado_mcmc_list[step].values())
    dy_dx = np.gradient(y, x)

    plt.plot(x, dy_dx, label=f'MCMC: {step} pasos', marker=marker_list[ii], linestyle='-', markersize=7, linewi


## Plot de Propp-Wilson
x = beta_list_pw
y = list(vesperado_pw.values())
dy_dx = np.gradient(y, x)

plt.plot(x, dy_dx, label='Propp-Wilson', color='purple', marker='x', linestyle='-', markersize=7, linewidth=2)


## Onsager critical
plt.vlines(onsager_critical_value, -0.3, 4.2, linestyles='dashed', label='Punto crítico de Onsager', color='red



## Propiedades de la gráfica
plt.xlabel(r'Inverso de temperatura, $\beta$')
plt.ylabel(r'$\partial M ( \eta ) \text{ / }  \partial \beta $')
plt.legend()
#plt.grid()

plt.show()
```



Finalmente, mencionamos algunas conclusiones:

- El algoritmo de Propp-Wilson con sandwiching, es mucho más computacionalmente eficiente que el algoritmo de Monte Carlo con Metropolis para valores de $\beta \leq \beta_c$. Sin embargo, a pesar de permitir muestreo perfecto, decae exponencialmente en términso de eficacia computacional cuando $\beta > \beta_c$.
- Los resultados de la simulación de Monte Carlo con 10^5 pasos son muy parecidos a los obtenidos con Propp-Wilson, lo que suguiere que este número de pasos provee muestras del sistema similares a los obtenidos con simulación perfecta, es decir, de la distribución estacionaria del sistema.
- La gráfica de la magnetización promedio en función de β exhibe indicios de una transición de fase alrededor del punto crítico de Onsager.
- En la gráfica de la derivada de la magnetización respecto a β para MCMCM con 10^5 pasos y Propp-Wilson se oberva un fuerte pico en β_c, lo que suguiere que la magnetización en el modelo de Ising atraviesa una transición de fase, con valor crítico $\beta_c = \frac{1}{2}\log(1+\sqrt{2}) \approx 0.441$, el cual coincide con el valor crítico de Onsager.

Simulated Annealing

Camino mas corto de la hormiga empezando en (0,0), pero sin retornar a su colonia

Empezamos creando funciones con las siguientes funcionalidades:

- Carga y lectura de los datos de las coordenadas de las colonias desde un archivo .csv
- Cálculo de la distancia euclidiana entre los puntos $(x_1, y_1), (x_2, y_2)$, definida como $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Se puede cambiar para usar otras métricas de distancia
- Cálculo de la distancia total que hace la hormiga al completar el recorrido por todas las colonias
- Generación de un recorrido aleatorio que empiece en la colonia de origen (punto $(0, 0)$).

In [2]:
```
def cargar_puntos_csv(archivo):
    return pd.read_csv(archivo, header=None).values

def calcular_distancia_puntos(punto_1, punto_2):
    delta_x = punto_1[0] - punto_2[0]
    delta_y = punto_1[1] - punto_2[1]

    return (delta_x**2 + delta_y**2)**0.5

def calcular_distancia_total(recorrido):

    total = 0
```

```
    for i in range(len(recorrido)-1):
        total += calcular_distancia_puntos(recorrido[i], recorrido[i + 1])

    return total

def recorrido_aleatorio(puntos):

    recorrido = list(range(1, len(puntos)))  # Excluir el punto (0,0) del shuffle
    random.shuffle(recorrido)

    recorrido.insert(0, 0)  # Agregar el punto (0,0) al inicio

    return np.array([puntos[x] for x in recorrido])
```

Para nuestro algoritmo, la probabilidad de aceptación está dada por la siguiente fórmula:

$$p = \min\left(e^{\frac{f(x)-f(y)}{T}}, 1\right)$$

donde $f(x)$ es la distancia total del mejor recorrido encontrado, $f(y)$ es la distancia total del recorrido candidato y T es la temperatura actual del sistema. Así, definimos la siguiente función:

In [3]:
```
def calcular_aceptacion(mejor_distancia, distancia, temp):
    return np.exp((mejor_distancia-distancia)/temp)
```

Y con esto, creamos la función que ejecuta el simulated annealing teniendo en cuenta que a los puntos escogidos aleatoriamente los cambiara de lugar para crear el recorrido candidato. Cabe resaltar que para asegurar que el punto (0,0) sea el primero, se debe prohibir que se elija este nodo al generar los vertices aleatorios al seleccionar:

In [4]:
```
def simulated_annealing(puntos, t_inicial, temp_final, enfriamiento, iteraciones, retorna=False):
    distancias = []
    acceptance_probabilities = []
    temperaturas = []

    recorrido_inicial = recorrido_aleatorio(puntos)
    distancia_inicial = calcular_distancia_total(recorrido_inicial)
    distancias.append(distancia_inicial)




    temp = t_inicial

    if retorna:
        rango_valido = len(puntos)-1
        recorrido_inicial = np.concatenate((recorrido_inicial, np.array([[0,0]])))
    else:
        rango_valido = len(puntos)


    recorrido = deepcopy(recorrido_inicial)
    mejor_distancia = calcular_distancia_total(recorrido)



    while temp > temp_final:

        for _ in range(iteraciones):

            i, j = np.random.randint(1, rango_valido, size=2)

            aux = deepcopy(recorrido[i])
            recorrido[i] = recorrido[j]
            recorrido[j] = aux

            nueva_distancia = calcular_distancia_total(recorrido)

            if nueva_distancia < mejor_distancia:
                mejor_distancia = nueva_distancia
                acceptance_prob = 1

            else:
                acceptance_prob = calcular_aceptacion(mejor_distancia, nueva_distancia, temp)
                if np.random.rand() < acceptance_prob:
                    mejor_distancia = nueva_distancia
                else:
                    aux = deepcopy(recorrido[i])
                    recorrido[i] = recorrido[j]
```

```
                    recorrido[j] = aux

            distancias.append(mejor_distancia)
            acceptance_probabilities.append(acceptance_prob)
            temperaturas.append(temp)


        temp *= enfriamiento
    return recorrido, mejor_distancia, recorrido_inicial, distancia_inicial, distancias, acceptance_probabiliti
```

Ahora veamos cuál es el mejor recorrido y su distancia con el siguiente esquema de temperatura:

- Temperatura Inicial: 100
- Temperatura Final: 0.0001
- Factor de enfriamiento: 0.99999
- Iteraciones por temperatura: 2

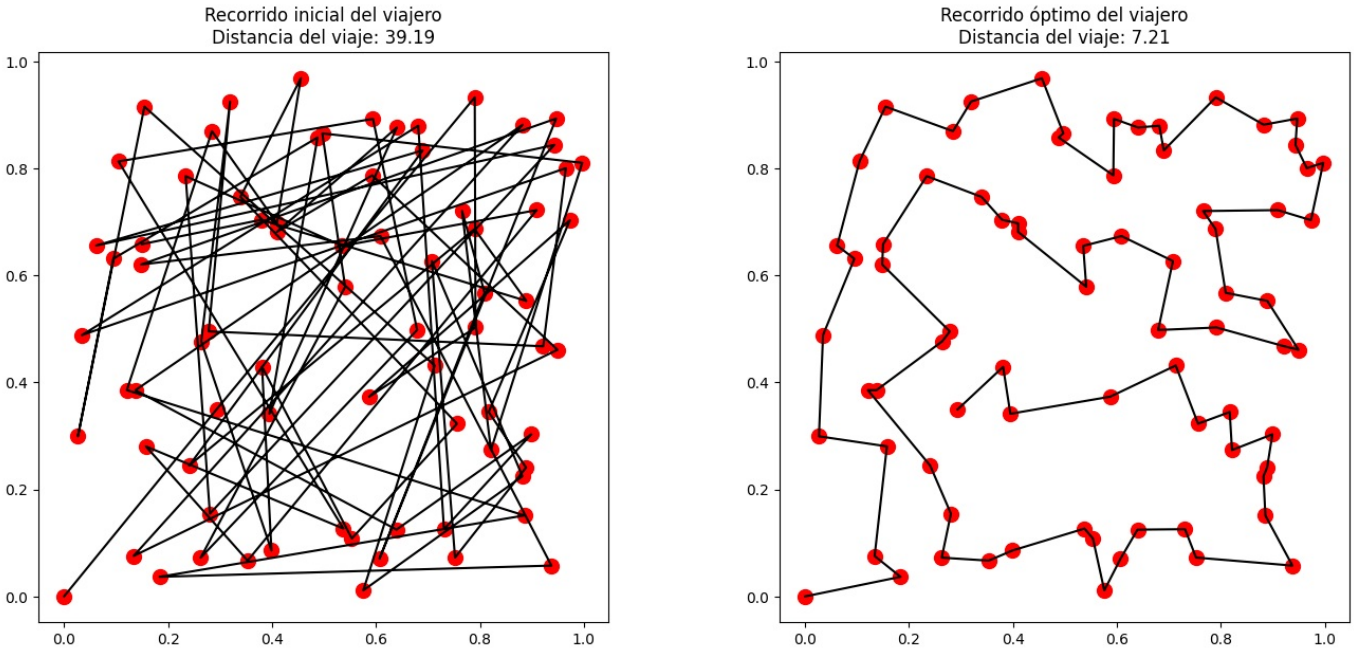
In [473…
```
data = cargar_puntos_csv("data.csv")

temp_inicial = 100
temp_final = 0.0001
enfriamiento = 0.99999

iteraciones = 2

recorrido, distancia, recorrido_inicial, distancia_inicial, all_distancias, acceptance_probabilities, temperatu

#print("Mejor recorrido:", recorrido)
print("Distancia total:", distancia)
```

```
Distancia total: 7.205793897989737
```

Veamos los resultados obtenidos.

Empecemos visualizando el recorrido encontrado:

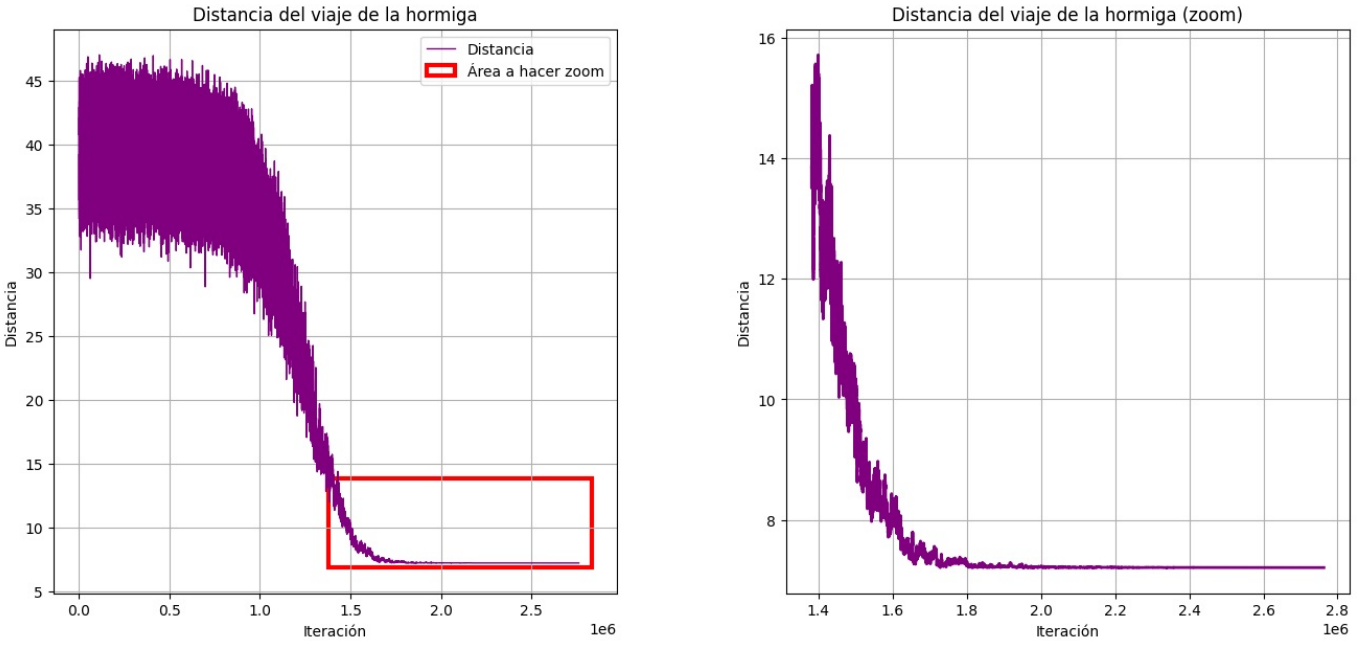
In [474…
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 7))
plt.subplots_adjust(wspace=0.3)



ax1.plot(recorrido_inicial[[x for x in range(len(data))], 0], recorrido_inicial[[x for x in range(len(data))],
ax1.scatter(data[:, 0], data[:, 1], color='red', label='Puntos', s=100)
ax1.set_title(f'Recorrido inicial del viajero\nDistancia del viaje: {distancia_inicial:.2f}')



ax2.plot(recorrido[[x for x in range(len(data))], 0], recorrido[[x for x in range(len(data))], 1], color='black
ax2.scatter(data[:, 0], data[:, 1],  color='red', label='Puntos', s=100)
ax2.set_title(f'Recorrido óptimo del viajero\nDistancia del viaje: {distancia:.2f}')


plt.show()
```



Ahora veamos como evolucionó la distancia recorrida por la hormiga (función de coste) a lo largo de la simulación

In [475…

```
import matplotlib.patches as patches


n_steps = len(all_distancias)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 7))
plt.subplots_adjust(wspace=0.3)



left_side = n_steps//2
bot_side = all_distancias[-1]*0.95
ancho = (n_steps - n_steps//2)*1.05
alto = -(all_distancias[-1] - all_distancias[n_steps//2])*1.05


rect = patches.Rectangle((left_side, bot_side), ancho, alto, linewidth=3, edgecolor='r', facecolor='none', label

ax1.plot(range(n_steps), all_distancias, label='Distancia', linewidth=1, color='purple')
ax1.add_patch(rect) # Rectangulo rojo para contextualizar la Figura con zoom
ax1.set_title('Distancia del viaje de la hormiga')
ax1.set_xlabel("Iteración")
ax1.set_ylabel("Distancia")
ax1.legend()
ax1.grid()



ax2.plot(range(n_steps//2, n_steps), all_distancias[n_steps//2::], linewidth=2, color='purple')
ax2.set_title(f'Distancia del viaje de la hormiga (zoom)')
ax2.set_xlabel("Iteración")
ax2.set_ylabel("Distancia")
ax2.grid()


plt.show()
```

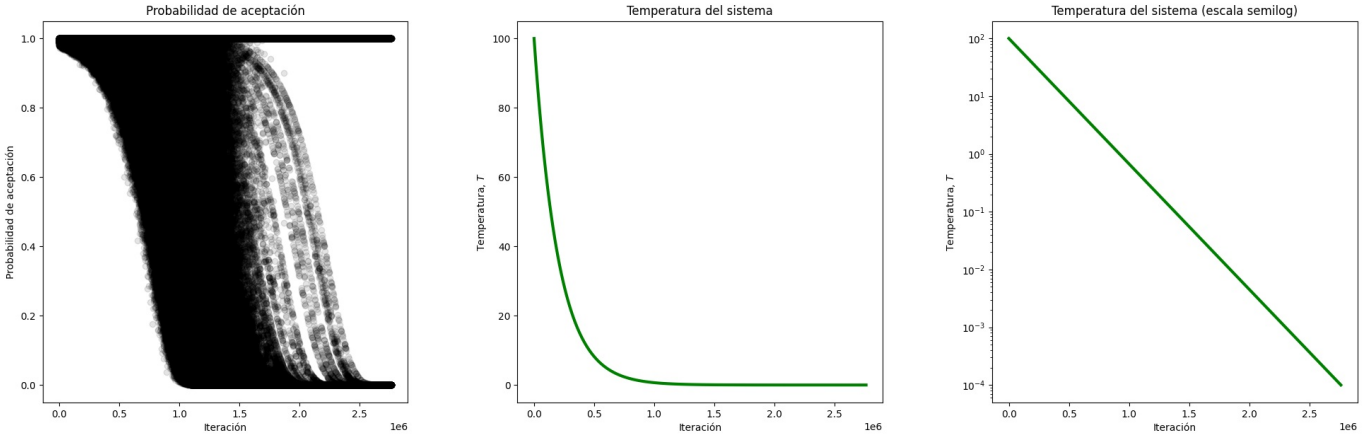


Finalmente, veamos como evolucionó la probabilidad de aceptación y la temperatura del sistema. Dado que en cada iteración la temperatura disminuía por un factor constante, el comportamiento es exponencial, como se verifica en la tercera gráfica

In [476…

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(24, 7))
plt.subplots_adjust(wspace=0.3)



ax1.scatter(range(n_steps -1), acceptance_probabilities, color='black', alpha=0.1)
ax1.set_title(f'Probabilidad de aceptación')
ax1.set_ylabel(r'Probabilidad de aceptación')
ax1.set_xlabel(r'Iteración')


ax2.plot(range(n_steps -1), temperaturas, linewidth=3, color='green')
ax2.set_title(f'Temperatura del sistema')
ax2.set_ylabel(r'Temperatura, $T$')
ax2.set_xlabel(r'Iteración')
#ax2.grid()
```

```
ax3.plot(range(n_steps -1), temperaturas, linewidth=3, color='green')
ax3.set_title(f'Temperatura del sistema (escala semilog)')
ax3.set_ylabel(r'Temperatura, $T$')
ax3.set_xlabel(r'Iteración')
ax3.set_yscale('log')



plt.show()
```



Notamos una formación muy curiosa de patrones en la gráfica de la probabilidad de aceptación. Estos pueden deberse a que el sistema encuentra un nuevo mínimo con configuración radicalmente diferente a la anterior que conlleva a una sucesión de varias mejoras sustanciales; es decir, el sistema se encontraba atrapado en un mínimo local.

Camino más corto empezando en (0,0) y retornando a su colonia

La función que creamos antes para el Annealing funciona también en este caso. El unico cambio que se debe realizar es añadir al recorrido el punto $(0, 0)$ al final prohibir que sea elegido al seleccionar los vertices a intercambiar.

Ahora veamos como se comporta el algoritmo si se necesita terminar en el punto de inicio:

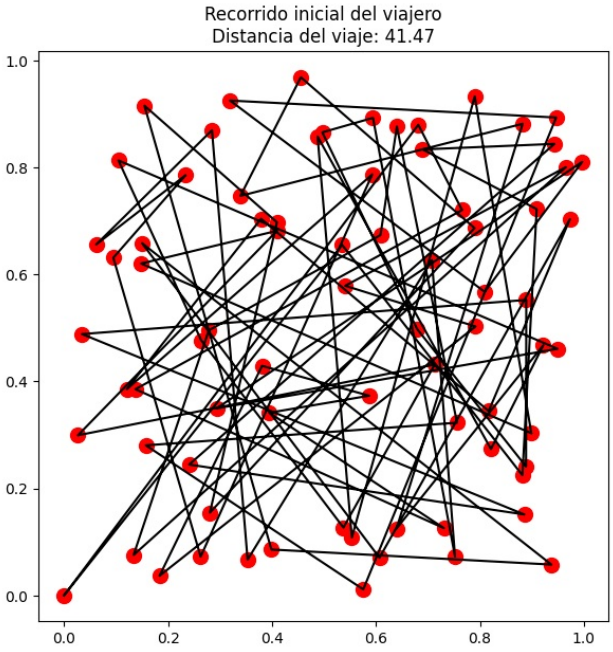
In [5]:
```
data = cargar_puntos_csv("data.csv")

temp_inicial = 100
temp_final = 0.0001
enfriamiento = 0.999995

iteraciones = 4


recorrido_ret, distancia_ret, recorrido_inicial_ret, distancia_inicial_ret, all_distancias_ret, acceptance_proba

#print("Mejor recorrido:", recorrido_ret)
print("Distancia total:", distancia_ret)
```

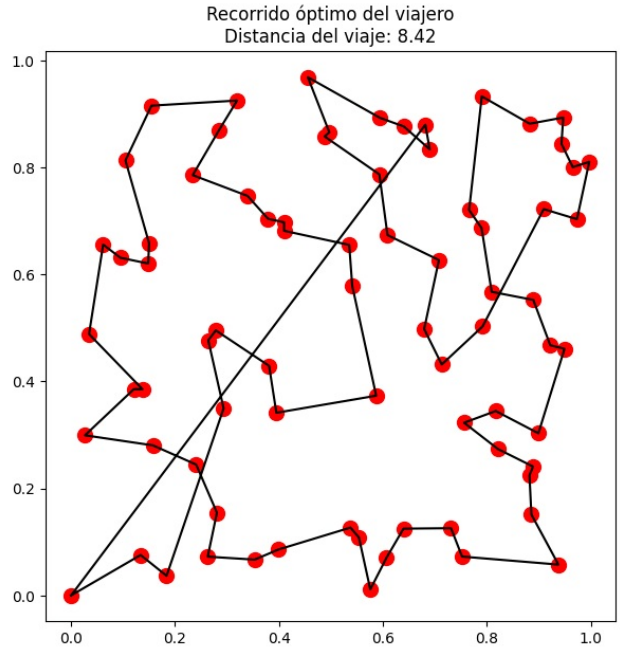
```
Distancia total: 8.417462412667364
```

Veamos los resultados obtenidos.

Empecemos visualizando el recorrido encontrado:

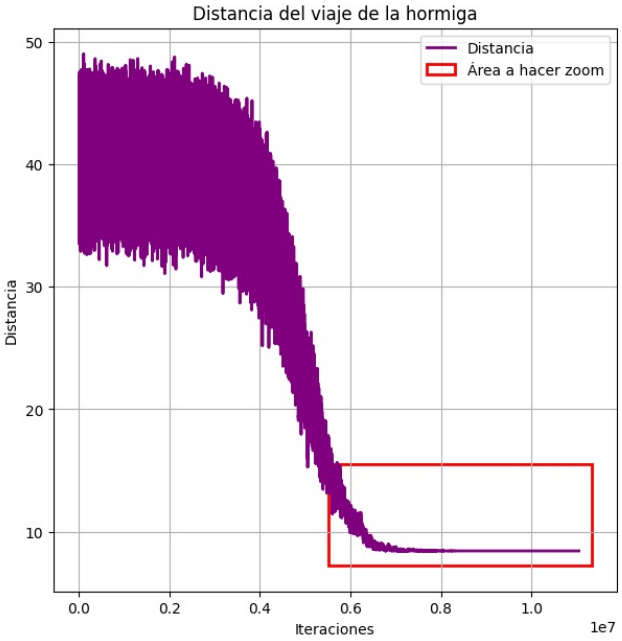
In [6]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 7))
plt.subplots_adjust(wspace=0.3)



ax1.plot(recorrido_inicial_ret[[x for x in range(len(data)+1)], 0], recorrido_inicial_ret[[x for x in range(len
ax1.scatter(data[:, 0], data[:, 1], color='red', label='Puntos', s=100)
ax1.set_title(f'Recorrido inicial del viajero\nDistancia del viaje: {distancia_inicial_ret:.2f}')



ax2.plot(recorrido_ret[[x for x in range(len(data)+1)], 0], recorrido_ret[[x for x in range(len(data)+1)], 1],
ax2.scatter(data[:, 0], data[:, 1], color='red', label='Puntos', s=100)
ax2.set_title(f'Recorrido óptimo del viajero\nDistancia del viaje: {distancia_ret:.2f}')


plt.show()
```

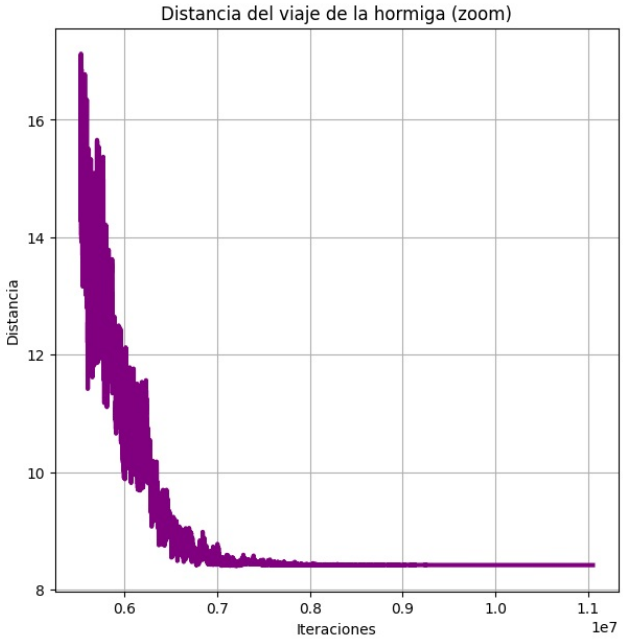




Ahora veamos como evolucionó la distancia recorrida por la hormiga (función de coste) a lo largo de la simulación

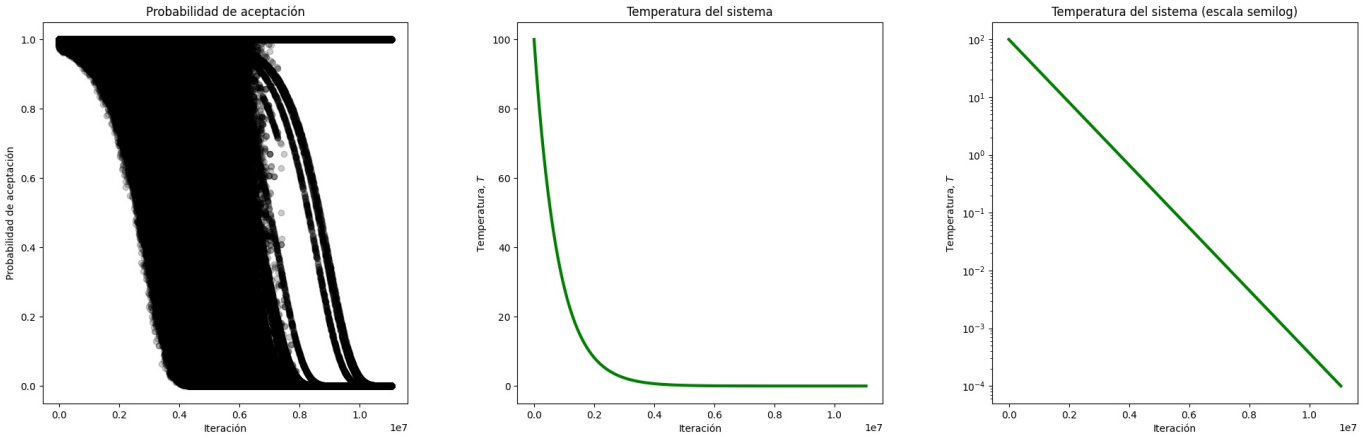
In [9]:
```
import matplotlib.patches as patches


n_steps = len(all_distancias_ret)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 7))
plt.subplots_adjust(wspace=0.3)



left_side = n_steps//2
top_side = all_distancias_ret[n_steps//2]*0.95
ancho = (n_steps - n_steps//2)*1.05
alto = (all_distancias_ret[-1] - all_distancias_ret[n_steps//2])*1.05


rect = patches.Rectangle((left_side, top_side), ancho, alto, linewidth=2, edgecolor='r', facecolor='none', label

ax1.plot(range(n_steps), all_distancias_ret, label='Distancia', linewidth=2, color='purple')
ax1.add_patch(rect) # Rectangulo rojo para contextualizar la Figura con zoom
ax1.set_title('Distancia del viaje de la hormiga')
ax1.set_xlabel("Iteraciones")
ax1.set_ylabel("Distancia")
ax1.legend(loc='upper right')
ax1.grid()



ax2.plot(range(n_steps//2, n_steps), all_distancias_ret[n_steps//2::], linewidth=3, color='purple')
ax2.set_title(f'Distancia del viaje de la hormiga (zoom)')
ax2.set_xlabel("Iteraciones")
ax2.set_ylabel("Distancia")
ax2.grid()


plt.show()
```





Finalmente, veamos como evolucionó la probabilidad de aceptación y la temperatura del sistema.

In [10]:
```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(24, 7))
plt.subplots_adjust(wspace=0.3)



ax1.scatter(range(n_steps -1), acceptance_probabilities_ret, color='black', alpha=0.2)
ax1.set_title(f'Probabilidad de aceptación')
ax1.set_ylabel(r'Probabilidad de aceptación')
ax1.set_xlabel(r'Iteración')


ax2.plot(range(n_steps -1), temperaturas_ret, linewidth=3, color='green')
ax2.set_title(f'Temperatura del sistema')
ax2.set_ylabel(r'Temperatura, $T$')
ax2.set_xlabel(r'Iteración')
#ax2.grid()



ax3.plot(range(n_steps -1), temperaturas_ret, linewidth=3, color='green')
ax3.set_title(f'Temperatura del sistema (escala semilog)')
ax3.set_ylabel(r'Temperatura, $T$')
ax3.set_xlabel(r'Iteración')
ax3.set_yscale('log')



plt.show()
```



Nuevamente, notamos los patrones mencionados anteriormente para el caso en el que la hormiga no tiene que retornar a (0,0)

Como conclusión, pudimos encontrar soluciones aproximadas al problema del Traveling Businessman mediante simulated annealing, con y sin retorno al origen, utilizando una temperatura con decaimiento exponencial. Se evidenciaron patrones en la probabilidad de aceptación que pueden ser indicios del viaje del algoritmo a través de mínimos locales. Finalmente, notar que el desempeño computacional del algoritmo depende de encontrar valores adecuados de la temperatura inicial, final y de su factor de decaimiento, proceso en el cual gastamos gran parte del periodo de simulación.