N REYNAS USANDO RECOCIDO

INTRODUCCIÓN

El problema de las N reinas consiste en colocar N reinas en un tablero de N × N de manera que ninguna de ellas se ataque entre sí. Esto significa que no pueden estar en la misma fila, columna o diagonal. En el caso de las 8 reinas, el objetivo es encontrar una disposición en un tablero de 8 × 8 que cumpla estas restricciones.

Para resolverlo, se utiliza Recocido Simulado, un algoritmo inspirado en el proceso de enfriamiento de los metales. Se basa en la aceptación probabilística de soluciones subóptimas para evitar quedarse atrapado en mínimos locales. La temperatura inicial (T_0) disminuye gradualmente siguiendo la secuencia logarítmica de Boltzmann, lo que reduce la probabilidad de aceptar soluciones peores con el tiempo.

Generacion de la sucion inicial

Permite que el usuario ingrese manual mente una solucion inicial o que el sistema lo pueda generar aleatoreamente, una solucion se representa como una lista de n numeros, donde cada numero indica la fila en la qu esta una reina en la columna correspondiente

```
def generar_solucion_inicial(self):
    opcion = input("¿Deseas ingresar la solución inicial manualmente? (s/n): ").strip().lower()

    if opcion == 's':
        while True:
            try:
                solucion = list(map(int, input(f"Ingrese {self.n} valores separados por espacios (0-{self.n-1}): ").split()))
                if len(solucion) == self.n and all(0 <= x < self.n for x in solucion):
                    return solucion
                else:
                    print(f"Por favor, ingrese {self.n} valores dentro del rango 0-{self.n-1}.")
            except ValueError:
                print("Entrada inválida. Inténtalo de nuevo.")
    else:
        sol_in = [random.randint(0, self.n - 1) for _ in range(self.n)]
        print("Solucion inicial: " + str(sol_in))
        return sol_in
```

Contar Conflictos

```python
def contar_conflictos(self, solucion):
    conflictos = 0
    for i in range(self.n):
        for j in range(i + 1, self.n):
            if solucion[i] == solucion[j] or abs(solucion[i] - solucion[j]) == abs(i - j):
                conflictos += 1
    return conflictos
```

Cuenta cuantas reinas se atacan entre si,
Dos reinas estan en conflicto si estan en la misma fila o en la misma diagonal

Generar un vecino

Genera una solucion vecina modificando la actual, dos opciones de mutacion son el intercambio de dos reinas (swap) y mover una reina a una nueva fila

```
def generar_vecino(self, solucion):
    vecino = solucion[:]
    if random.random() < 0.5:  # 50% de las veces hacemos swap normal
        a, b = random.sample(range(self.n), 2)
        vecino[a], vecino[b] = vecino[b], vecino[a]
    else:  # 50% de las veces cambiamos una reina a una nueva posición
        i = random.randint(0, self.n - 1)
        vecino[i] = random.randint(0, self.n - 1)
    return vecino
```

Actualizar la temperatura

```
def actualizar_temperatura(self):
    #self.temperatura = self.temperatura / (1 + self.k * math.log(1 + self.iteracion))
    self.temperatura *= 0.99
    self.iteracion += 1
```

Reduce gradualmente la temperatura (T=T*0.99)

A medida que la temperatura disminuye, el algoritmo se vuelve mas estricto al aceptar soluciones peores.

```python
    def recocido_simulado(self):
        tiempo_inicio = time.perf_counter()
        solucion_actual = self.generar_solucion_inicial()
        mejor_solucion = solucion_actual[:]
        mejor_puntaje = self.contar_conflictos(mejor_solucion)
        iteraciones = 0

        while self.temperatura > self.min_temperatura and mejor_puntaje > 0:
            vecinos = [self.generar_vecino(solucion_actual) for _ in range(10)]  #generar vecindario


            mejor_vecino = min(vecinos, key=lambda vecino: self.contar_conflictos(vecino))
            mejor_puntaje_vecino = self.contar_conflictos(mejor_vecino)

            if mejor_puntaje_vecino < mejor_puntaje:
                solucion_actual = mejor_vecino[:]
                mejor_solucion = mejor_vecino[:]
                mejor_puntaje = mejor_puntaje_vecino
            else:
                probabilidad = math.exp((mejor_puntaje - mejor_puntaje_vecino) / (self.temperatura + 1e-10))
                if random.random() < probabilidad:
                    solucion_actual = mejor_vecino[:]

            self.actualizar_temperatura()
            iteraciones += 1

        tiempo_fin = time.perf_counter()
        tiempo_ejecucion = tiempo_fin - tiempo_inicio
        return mejor_solucion, mejor_puntaje, iteraciones, tiempo_ejecucion

# Ejecutar el algoritmo
solucionador = NReinasRecocido(n=10)
solucion, conflictos, iteraciones, tiempo = solucionador.recocido_simulado()
```

Recocido Simulado

Comienza con una solucion inicial
Evalua soluciones vecinas y decide si las acepta inmediatamente si son mejores o las acepta con una probabilidad si son peores (basado en la temperatura)

Ejecucion del algoritmo

```
# Ejecutar el algoritmo
solucionador = NReinasRecocido(n=8)
solucion, conflictos, iteraciones, tiempo = solucionador.recocido_simulado()

# Imprimir resultados
print("Solucion encontrada:", solucion)
print("Conflictos restantes:", conflictos)
print("Iteraciones realizadas:", iteraciones)
print(f"Tiempo de ejecucion: {tiempo:.4f} segundos")
```

Crea un objetivo de N reinas, y se ejecuta el algoritmo y obtiene una solucion final, la cantidad de conflictos restantes, el numero de iteraciones y el tiempo de ejecucion

Conclusión

Este codigo usa usa recocido simulado para resolver el problema de las N reinas de manera eficiente, aprovechando la exploración aleatoria y la aceptación de soluciones peores segun la temperatura