Algoritmo para Mejorar el Período Libre de Contención

```
// Variables de configuración
max_intentos = 10 // Número máximo de intentos de transmisión
tiempo_inicial = 100 // Tiempo inicial para el retroceso exponencial en milisegundos
factor_exponencial = 2 // Factor de multiplicación para el retroceso exponencial

// Algoritmo para la transmisión segura en una red ad hoc inalámbrica
algoritmo_transmision(adresse_destino, mensaje):
    intentos = 0
    canal_libre = False

    // Vector de inicialización para mayor seguridad
    iv = generar_vector_inicializacion()

    // Repetir hasta que la transmisión sea exitosa o se alcance el límite de intentos
    mientras intentos < max_intentos y no canal_libre:
        // Escuchar el canal para verificar si está libre
        si canal_esta_libre():
            canal_libre = True
        más:
            // Retroceso exponencial para evitar colisiones
            tiempo_espera = tiempo_inicial * (factor_exponencial ** intentos)
            esperar(tiempo_espera)
            intentos = intentos + 1

    si canal_libre:
        // Transmitir el mensaje con el vector de inicialización
        enviar_mensaje(adresse_destino, mensaje, iv)

        // Esperar confirmación de recepción
        ack = esperar_confirmacion(adresse_destino, tiempo_espera)

        si ack es exitoso:
            devolver "Transmisión exitosa"
        más:
            canal_libre = False
            // Retroceso exponencial
            tiempo_inicial = tiempo_espera // Actualizar el tiempo inicial
    más:
        devolver "Error: No se pudo transmitir el mensaje después de múltiples intentos"
```

Explicación del algoritmo

Retroceso Exponencial: Si el canal está ocupado, el algoritmo espera un tiempo creciente basado en un factor exponencial. Esto reduce la probabilidad de colisiones.

Vector de Inicialización (IV): Se usa para asegurar la transmisión. Es un valor aleatorio que se combina con la transmisión para agregar seguridad.

Confirmación de Recepción (ACK): Después de transmitir, el dispositivo espera una confirmación del destinatario. Si no recibe confirmación, el proceso se reinicia con un tiempo de espera mayor.

Límite de Intentos: Si se excede el número máximo de intentos, el algoritmo devuelve un mensaje de error indicando que la transmisión no fue exitosa.

Esta estructura permite mejorar el período libre de contención en redes ad hoc inalámbricas, utilizando retroceso exponencial y mecanismos de confirmación para minimizar colisiones. La adición de vectores de inicialización aumenta la seguridad de las transmisiones en redes ad hoc.

Parte 1

```
import random
import time

class Channel:
    def __init__(self):
        self.is_occupied = False
        self.last_transmitter = None

    def is_busy(self):
        # Simula si el canal está ocupado
        return random.choice([True, False])

    def occupy(self, node_id):
        self.is_occupied = True
        self.last_transmitter = node_id

    def release(self):
        self.is_occupied = False

    def transmit(self, node_id):
        if not self.is_busy():
            self.occupy(node_id)
            print(f"Node {node_id}: Transmitiendo datos")
            time.sleep(1)  # Simula el tiempo de transmisión
            print(f"Node {node_id}: Transmisión completada")
            self.release()
        else:
            print(f"Node {node_id}: Canal ocupado, intentando nuevamente")

class Node:
    def __init__(self, node_id):
        self.node_id = node_id

    def transmit(self, channel, attempt_limit=5):
        attempt = 0
        while attempt < attempt_limit:
            if channel.is_busy():
                backoff_time = self.exponential_backoff(attempt)
                print(f"Node {self.node_id}: Aplicando backoff por {backoff_time} segundos")
                time.sleep(backoff_time)
                attempt += 1
            else:
                channel.transmit(self.node_id)
                break

    def exponential_backoff(self, attempt):
        return random.uniform(0, 2**attempt)  # Backoff exponencial

# Simulación de múltiples nodos compitiendo por el canal
channel = Channel()
nodes = [Node(1), Node(2), Node(3)]

# Cada nodo intenta transmitir datos
for node in nodes:
  node.transmit(channel)
```

Parte 2

```
# Codificación DSSS
def dsss_encode(data, chip_code):
    encoded = []
    for bit in data:
        # Si el bit es 1, el chip_code se mantiene; si es 0, se invierte
        encoded_bit = [chip * int(bit) for chip in chip_code]
        encoded.extend(encoded_bit)
    return encoded

# Decodificación DSSS
def dsss_decode(encoded_data, chip_code):
    decoded = []
    index = 0
    while index < len(encoded_data):
        # Extrae segmentos del tamaño del chip_code
        segment = encoded_data[index:index + len(chip_code)]
        # Suma los valores y determina el bit decodificado
        decoded_bit = 1 if sum(segment) > 0 else 0
        decoded.append(decoded_bit)
        index += len(chip_code)  # Avanza al siguiente segmento
    return decoded

# Ejemplo de uso
data = "1010"  # Datos originales
chip_code = [1, -1, 1, -1]  # Código de dispersión

# Codificación
encoded_data = dsss_encode(data, chip_code)
print("Datos codificados:", encoded_data)

# Decodificación
decoded_data = dsss_decode(encoded_data, chip_code)
decoded_str = "".join(map(str, decoded_data))
print("Datos decodificados:", decoded_str)
```

Parte 3:

```
import random

# Función para ajustar el backoff según la tasa de éxito
def adjust_contention_window(success_rate):
    if success_rate < 0.5:
        return increase_backoff()
    else:
        return decrease_backoff()

# Aumenta el tiempo de backoff
def increase_backoff():
    backoff_factor = random.randint(2, 4) # Incremento aleatorio para simular variabilidad
    print("Incrementando el tiempo de backoff")
    return backoff_factor

# Disminuye el tiempo de backoff
def decrease_backoff():
    backoff_factor = random.randint(1, 2) # Decremento aleatorio para simular ajuste
    print("Disminuyendo el tiempo de backoff")
    return backoff_factor

# Función que ajusta la ventana de contención basándose en la tasa de éxito
def contention_window_adjustment():
    success_rate = calculate_success_rate() # Simula la tasa de éxito
    backoff_adjustment = adjust_contention_window(success_rate) # Ajusta el backoff
    return backoff_adjustment

# Simula el cálculo de la tasa de éxito
def calculate_success_rate():
    # Devuelve un valor aleatorio entre 0 y 1 para simular la tasa de éxito
    return random.random()

# Ejemplo de uso para el ajuste dinámico
contention_window_adjustment()
```

Parte 4

```
import random
import time

# Clase para el canal compartido
class Channel:
    def__init_(self):
        self.is_occupied = False
        self.last_transmitter = None

    def is_busy(self):
```

```
        # Simula si el canal está ocupado
        return random.choice([True, False])

    def occupy(self, node_id):
        self.is_occupied = True
        self.last_transmitter = node_id

    def release(self):
        self.is_occupied = False

# Clase para representar un nodo con CSMA/CA
class Node:
    def __init_(self, node_id):
        self.node_id = node_id

    def transmit(self, channel, attempt_limit=5):
        attempt = 0
        while attempt < attempt_limit:
            if channel.is_busy():
                backoff_time = self.exponential_backoff(attempt)
                print(f"Node {self.node_id}: Aplicando backoff por {backoff_time} segundos")
                time.sleep(backoff_time)
                attempt += 1
            else:
                channel.occupy(self.node_id)
                print(f"Node {self.node_id}: Transmitiendo datos")
                time.sleep(1) # Simula el tiempo de transmisión
                print(f"Node {self.node_id}: Transmisión completada")
                channel.release()
                break

    def exponential_backoff(self, attempt):
        # Backoff exponencial
        return random.uniform(0, 2**attempt)

# Codificación y decodificación DSSS
def dsss_encode(data, chip_code):
    encoded = []
    for bit in data:
        encoded_bit = [chip * int(bit) for chip in chip_code]
        encoded.extend(encoded_bit)
    return encoded

def dsss_decode(encoded_data, chip_code):
    decoded = []
    index = 0
    while index < len(encoded_data):
        segment = encoded_data[index:index + len(chip_code)]
```

```
        decoded_bit = 1 if sum(segment) > 0 else 0
        decoded.append(decoded_bit)
        index += len(chip_code)
    return decoded

# Ajuste dinámico del período de contención
def adjust_contention_window(success_rate):
    if success_rate < 0.5:
        return increase_backoff()
    else:
        return decrease_backoff()

def increase_backoff():
    print("Incrementando el tiempo de backoff")
    return random.randint(2, 4)

def decrease_backoff():
    print("Disminuyendo el tiempo de backoff")
    return random.randint(1, 2)

def calculate_success_rate():
    # Simula la tasa de éxito
    return random.random()

# Ejemplo de uso
channel = Channel()  # Creamos el canal compartido
nodes = [Node(1), Node(2), Node(3)]  # Tres nodos en la simulación

# Codificamos datos utilizando DSSS
chip_code = [1, -1, 1, -1]  # Código de dispersión
data = "1010"
encoded_data = dsss_encode(data, chip_code)
decoded_data = dsss_decode(encoded_data, chip_code)

print("Datos codificados:", encoded_data)
print("Datos decodificados:", "".join(map(str, decoded_data)))

# Simulamos transmisiones de nodos con ajuste de contención
for node in nodes:
    success_rate = calculate_success_rate() # Simulamos una tasa de éxito aleatoria
    adjust_contention_window(success_rate) # Ajustamos el backoff
    node.transmit(channel)  # El nodo intenta transmitir datos
```