```
import numpy as np
from dataclasses import dataclass, field
from typing import Dict, List, Tuple, Set, Optional, Any, Callable
from enum import Enum
import json
from datetime import datetime

@dataclass
class ValorVerdadeParaconsistente:
    """
    Valor de verdade paraconsistente: (verdade, contradição)
    v ∈ [0,1]: grau de verdade
    c ∈ [0,1]: grau de contradição
    """
    verdade: float  # v ∈ [0,1]
    contradicao: float  # c ∈ [0,1]

    def __post_init__(self):
        self.verdade = np.clip(self.verdade, 0, 1)
        self.contradicao = np.clip(self.contradicao, 0, 1)

    def is_consistente(self, threshold=0.1) -> bool:
        return self.contradicao < threshold

    def is_contraditorio(self, threshold=0.5) -> bool:
        return self.contradicao > threshold

    def is_nao_trivial(self) -> bool:
        """Sistema permanece não-trivial mesmo com contradições"""
        return not (self.verdade == 1.0 and self.contradicao == 1.0)

@dataclass
class Conhecimento:
    """Unidade de conhecimento epistêmico"""
    id: str
    conteudo: str
    valor_verdade: ValorVerdadeParaconsistente
    contexto: str
    fonte: str
    timestamp: Any = field(default_factory=lambda: np.datetime64('now'))
    relacoes: Set[str] = field(default_factory=set)

class ConflictResolver:
    """
    Módulo de Resolução de Conflitos "Escolha de Sofia"
    Implementa estratégias paraconsistentes para resolver conflitos epistêmicos
    """

    def __init__(self):
        self.historico_resolucoes = []
        self.estrategias = {
```

```
            'reconciliação': self.reconciliacao,
            'síntese': self.sintese_paraconsistente,
            'contextualização': self.contextualizacao,
            'hierarquização': self.hierarquizacao
        }

    def reconciliacao(self, A: ValorVerdadeParaconsistente, B: ValorVerdadeParaconsistente) -> Dict:
        """
        Estratégia de reconciliação: Encontra um ponto médio não-trivial
        """
        # Aplicar operador paraconsistente
        sintese = self.operador_sintese(A, B)

        # Calcular distância entre as verdades
        distancia = abs(A.verdade - B.verdade)

        # Verificar se a síntese é não-trivial
        if sintese.is_nao_trivial():
            return {
                'estrategia': 'reconciliação',
                'sucesso': True,
                'resultado': sintese,
                'distancia': distancia,
                'confianca': 1 - distancia
            }
        return {
            'estrategia': 'reconciliação',
            'sucesso': False,
            'resultado': sintese,
            'distancia': 1.0,
            'confianca': 0.0
        }

    def sintese_paraconsistente(self, A: ValorVerdadeParaconsistente, B:
ValorVerdadeParaconsistente) -> Dict:
        """
        Estratégia de síntese paraconsistente: Cria uma nova verdade que incorpora ambas
        """
        # Aplicar operador paraconsistente
        sintese = self.operador_sintese(A, B)

        # Verificar se a síntese é consistente
        if sintese.is_consistente():
            return {
                'estrategia': 'síntese',
                'sucesso': True,
                'resultado': sintese,
                'confianca': sintese.verdade
            }

        # Se não consistente, tentar reduzir a contradição
        nova_contradicao = max(A.contradicao, B.contradicao) * 0.8
```

```
    sintese_ajustada = ValorVerdadeParaconsistente(sintese.verdade, nova_contradicao)

    return {
        'estrategia': 'síntese',
        'sucesso': sintese_ajustada.is_consistente(),
        'resultado': sintese_ajustada,
        'confianca': sintese_ajustada.verdade * (1 - sintese_ajustada.contradicao)
    }

def contextualizacao(self, A: Conhecimento, B: Conhecimento) -> Dict:
    """
    Estratégia de contextualização: Mantém ambas verdades em contextos distintos
    """
    # Verificar se os contextos são diferentes
    if A.contexto != B.contexto:
        return {
            'estrategia': 'contextualização',
            'sucesso': True,
            'resultado': {
                'conhecimento_A': A,
                'conhecimento_B': B,
                'contextos_distintos': True
            },
            'confianca': 1.0
        }

    # Se contextos são iguais, tentar diferenciar
    novo_contexto_A = f"{A.contexto}_variant_A"
    novo_contexto_B = f"{B.contexto}_variant_B"

    A_ajustado = Conhecimento(
        id=A.id,
        conteudo=A.conteudo,
        valor_verdade=A.valor_verdade,
        contexto=novo_contexto_A,
        fonte=A.fonte,
        relacoes=A.relacoes
    )

    B_ajustado = Conhecimento(
        id=B.id,
        conteudo=B.conteudo,
        valor_verdade=B.valor_verdade,
        contexto=novo_contexto_B,
        fonte=B.fonte,
        relacoes=B.relacoes
    )

    return {
        'estrategia': 'contextualização',
        'sucesso': True,
        'resultado': {
```

```
            'conhecimento_A': A_ajustado,
            'conhecimento_B': B_ajustado,
            'contextos_distintos': True
        },
        'confianca': 0.9
    }

def hierarquizacao(self, A: Conhecimento, B: Conhecimento) -> Dict:
    """
    Estratégia de hierarquização: Estabelece uma relação de precedência
    """
    # Usar timestamp para determinar precedência
    if A.timestamp < B.timestamp:
        dominante = B
        subordinado = A
    else:
        dominante = A
        subordinado = B

    return {
        'estrategia': 'hierarquização',
        'sucesso': True,
        'resultado': {
            'dominante': dominante,
            'subordinado': subordinado,
            'relacao': 'precedência temporal'
        },
        'confianca': dominante.valor_verdade.verdade
    }

def operador_sintese(self, A: ValorVerdadeParaconsistente, B: ValorVerdadeParaconsistente) ->
ValorVerdadeParaconsistente:
    """Operador ⊕: Síntese paraconsistente"""
    v_sintese = (A.verdade + B.verdade) / 2
    c_sintese = (A.contradicao + B.contradicao) / 2
    interacao = abs(A.verdade - B.verdade)
    c_sintese = min(1.0, c_sintese + interacao * 0.1)
    return ValorVerdadeParaconsistente(v_sintese, c_sintese)

def resolver_conflito(self, conhecimento_A: Conhecimento, conhecimento_B: Conhecimento,
estrategia: Optional[str] = None) -> Dict:
    """
    Resolve conflito entre dois conhecimentos usando estratégia especificada
    """
    # Detectar se há contradição real
    if not (conhecimento_A.valor_verdade.is_contraditorio() or
conhecimento_B.valor_verdade.is_contraditorio()):
        return {
            'conflito': False,
            'mensagem': 'Sem contradição detectada'
        }
```

```
        # Selecionar estratégia
        if estrategia and estrategia in self.estrategias:
            resolver = self.estrategias[estrategia]
        else:
            # Escolher estratégia automaticamente
            resolver = self._escolher_estrategia(conhecimento_A, conhecimento_B)

        # Aplicar estratégia
        resultado = resolver(conhecimento_A, conhecimento_B)

        # Registrar no histórico
        self.historico_resolucoes.append({
            'conhecimento_A': conhecimento_A.id,
            'conhecimento_B': conhecimento_B.id,
            'estrategia': resultado.get('estrategia', 'automática'),
            'sucesso': resultado.get('sucesso', False),
            'timestamp': str(datetime.now())
        })

        return resultado

    def _escolher_estrategia(self, A: Conhecimento, B: Conhecimento) -> Callable:
        """
        Escolhe estratégia automaticamente com base nas características do conflito
        """
        # Se contextos são diferentes, usar contextualização
        if A.contexto != B.contexto:
            return self.contextualizacao

        # Se uma das verdades é claramente mais consistente
        if A.valor_verdade.is_consistente() and not B.valor_verdade.is_consistente():
            return self.hierarquizacao
        if B.valor_verdade.is_consistente() and not A.valor_verdade.is_consistente():
            return self.hierarquizacao

        # Caso contrário, tentar síntese paraconsistente
        return self.sintese_paraconsistente

    def get_historico(self) -> List[Dict]:
        """Retorna histórico de resoluções de conflito"""
        return self.historico_resolucoes

    def get_estrategias(self) -> List[str]:
        """Retorna estratégias disponíveis"""
        return list(self.estrategias.keys())
```

Para usar este módulo, você pode fazer o seguinte:

Salve o código acima em um arquivo chamado conflict_resolver.py
Importe-o em seu projeto principal com: from conflict_resolver import ConflictResolver, Conhecimento, ValorVerdadeParaconsistente
Crie uma instância do ConflictResolver: resolver = ConflictResolver()

Use os métodos para resolver conflitos entre conhecimentos
O módulo inclui:

Todas as classes e funções necessárias
Documentação completa
Estratégias de resolução de conflitos
Histórico de resoluções
Funções auxiliares para escolha automática de estratégias
Este é um módulo autônomo que pode ser integrado ao sistema ELEDONTE ou usado independentemente para resolver conflitos epistêmicos em qualquer sistema que utilize lógica paraconsistente.