Temas Tratados en el Trabajo Práctico 4

- Representación del Conocimiento y Razonamiento Lógico.
- Estrategias de resolución de hipótesis: Encadenamiento hacia Adelante, Encadenamiento hacia Atrás y Resolución por Contradicción.
- Representación basada en circuitos.

Ejercicios Teóricos

1. ¿Qué es una inferencia?

La inferencia es la derivación lógica de proposiciones que no estaban explícitamente en la base de conocimientos. Se realiza mediante mecanismos de razonamiento (como modus ponens, encadenamiento hacia adelante/atrás, resolución, etc.). Es la base de cómo un sistema inteligente puede responder preguntas o tomar decisiones sin que todo esté "hardcodeado" en la base de datos.

1. ¿Cómo se verifica que un modelo se infiere de la base de conocimientos?

Para verificar si un modelo (o una conclusión) se infiere de la base de conocimientos (BK), se usa el concepto de consecuencia lógica:

-> Se dice que una proposición α se infiere de la base de conocimientos KB (se escribe: KB ⊨ α) si y solo si en todos los modelos donde KB es verdadero, también α es verdadero.

En la práctica, esto se puede verificar de varias formas:

Método semántico (basado en modelos)

- Consiste en comprobar si cada modelo que satisface la BK también satisface la proposición que queremos derivar.
- Es decir: se listan los posibles modelos (asignaciones de verdad) y se chequea.
- Poco práctico en problemas grandes, pero conceptualmente correcto.

Método sintáctico (basado en inferencia)

- En lugar de revisar todos los modelos, se aplican reglas de inferencia (ej.: modus ponens, resolución).
- Si a partir de la BK podemos deducir paso a paso la proposición, entonces está verificada.

Resolución por contradicción

- Una técnica común es suponer que la proposición NO es cierta y ver si eso lleva a una contradicción.
- Si la contradicción aparece, entonces la proposición sí se infiere de la BK

1. Observe la siguiente base de conocimiento:

$R1 : b \land c \to a$

$R2 : d \land e \to b$

$R3 : g \land e \to b$

$R4 : e \to c$

$R5 : d$

$R6 : e$

$R7 : a \land g \to f$

```
    3.1 ¿Cómo se puede probar que $a = True$ a través del
encadenamiento hacia adelante? Este método solamente usa reglas ya
incorporadas a la base de conocimiento para inferir la hipótesis, ¿qué
propiedad debe tener el algoritmo para asegurar que esta inferencia
sea posible?
```

1- De $R5$ y $R6$ sabemos que d y e

2- Con $d \land e$ y $R2$ inferimos b

3- Con e y $R4$ inferimos c

4- Con $b \land c$ y $R1$ inferimos a

Para asegurar que esta inferencia sea posible el algoritmo debe ser completo para cláusulas de Horn. Esto significa que se puede derivar cualquier sentencia que esté implicada.

```
    3.2 ¿Cómo se puede probar que $a = True$ a través del
encadenamiento hacia atrás? Este método asigna un valor de verdad a la
hipótesis y deriva las sentencias de la base de conocimiento, ¿qué
propiedad debe tener el algoritmo para asegurar que esta derivación
sea posible?
```

Objetivo: Mostrar que a

1- Asumimos a, luego, de $R1$ sigue $b \land c$

2- Para b, de $R2$ sigue $d \land e$

3- d y e están satisfechas por $R5$ y $R6$

4- Para e, de $R4$ sigue c

5- Con b y c demostrados, por $R1$ concluímos a

Para asegurar que esta inferencia sea posible el algoritmo debe ser completo para cláusulas de Horn. Esto significa que se puede derivar cualquier sentencia que esté implicada.

```
    3.3 Exprese la base de conocimiento en su Forma Normal Conjuntiva.
A continuación, demuestre por contradicción que $a = True$.
```

Forma Normal Conjuntiva

R1: $\neg b \vee \neg c \vee a$

R2: $\neg d \vee \neg e \vee b$

R3: $\neg g \vee \neg e \vee b$

R4: $\neg e \vee c$

R5: d

R6: e

R7: $\neg a \vee \neg g \vee f$

Para la prueba añadimos una nueva cláusula:

R8: $\neg a$

1- De $R1: \ \neg b \vee \neg c \vee a$ y $R8$ se obtiene $\neg b \vee \neg c$.

2- De $R6: \ e$ y $R4: \ \neg e \vee c$ se obtiene c.

3- De (1) y (2) se obtiene $\neg b$.

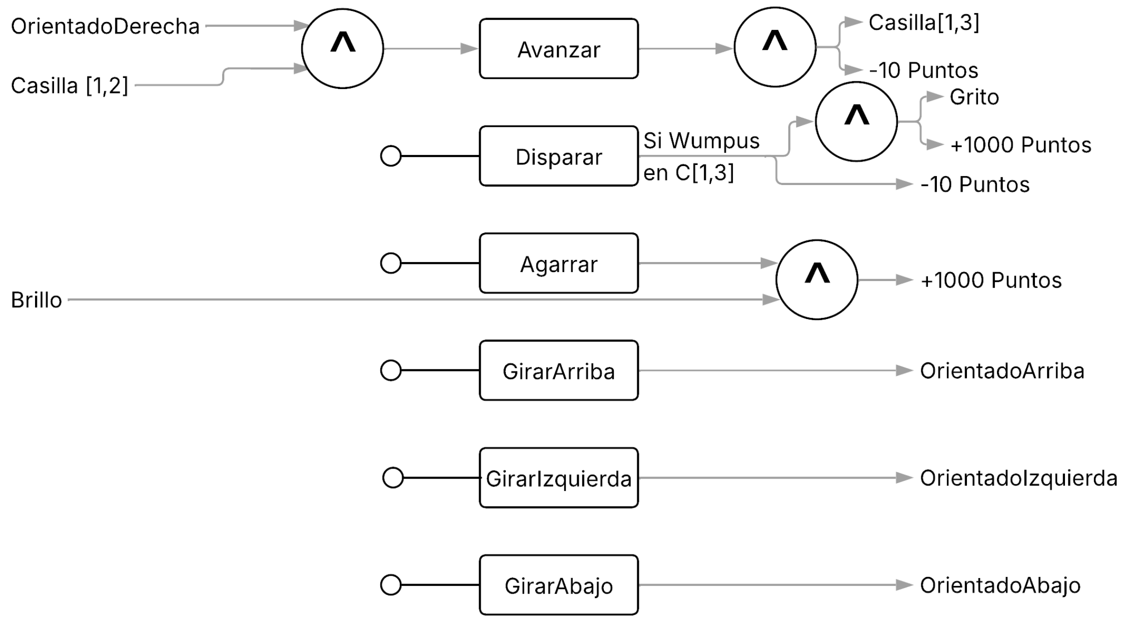
4- De $R5: \ d$ y $R2: \ \neg d \vee \neg e \vee b$ se obtiene $\neg e \vee b$.

- De (4) y $R6: \ e$ se obtiene b.

6- De (3) y (5) se llega a contradicción ($b \wedge \neg b$). Por lo tanto, a.

1. Diseñe con lógica proposicional basada en circuitos las proposiciones *OrientadoDerecha* y *Agente ubicado en la casilla [1,2]* para el mundo de wumpus de 4x4. Dibuje el circuito correspondiente.

Se coloca a continuacion un diseño del agente basado en circuitos:



1. El nonograma es un juego en el cual se posee un tablero en blanco y cada fila y columna presenta información sobre la longitud de un bloque en dicha fila/columna. Además, la leyenda puede indicar más de un número, indicando esto que existen varios bloques de las longitudes mostradas por la leyenda y en el mismo orden, separados por al menos un espacio vacío.

Resuelva el nonograma de la imagen de abajo escribiendo en primer lugar cada regla que puede incorporarse a la base de conocimientos inicial e incorporando cada inferencia que realice.

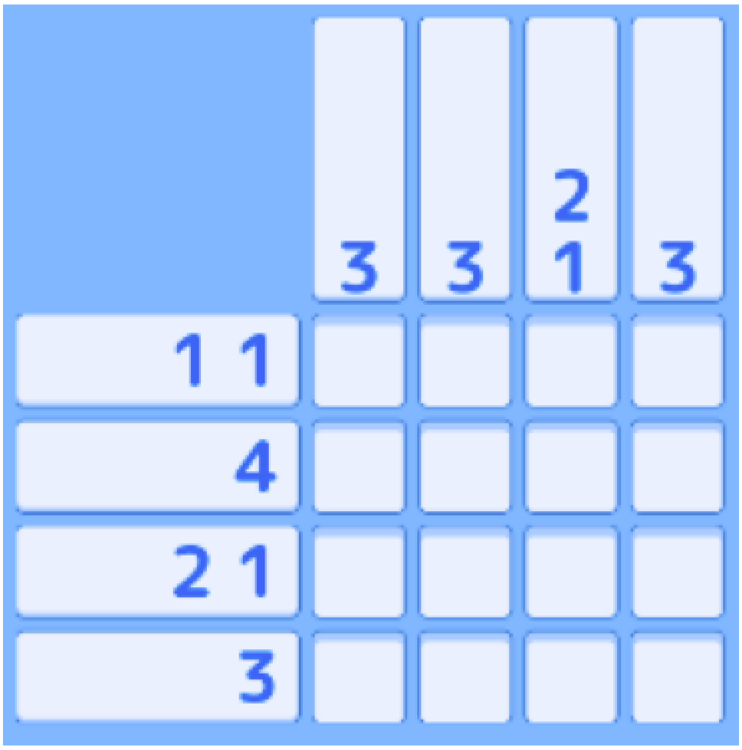
In [ ]:

```
import requests
from PIL import Image
from io import BytesIO
import matplotlib.pyplot as plt

# URL directa de Google Drive
url = "https://drive.google.com/uc?export=view&id=1SKiXvrI_TX-U4sbw60TYSRmaNYyFixmI"

# Descargar la imagen
response = requests.get(url)
img = Image.open(BytesIO(response.content))

# Mostrar la imagen
plt.imshow(img)
plt.axis('off')  # Ocultar ejes
plt.show()
```



BC inicial:

Filas:

F1=C11 ^ C12 v C12 ^ C14 v C11 ^ C14

F2=C21 ^ C22 ^ C23 ^ C24

F3=C31 ^ C32 ^ C34

F4=C41 ^ C42 ^ C43 v C42 ^ C43 ^ C44

Columnas:

X1=C11 ^ C21 ^ C31 v C21 ^ C31 ^ C41

X2=C12 ^ C22 ^ C32 v C22 ^ C32 ^ C42

X3=C13 ^ C23 ^ C43

X4=C14 ^ C24 ^ C34 v C24 ^ C34 ^ C44

Inferencias

C21, C22, C23, C24 a partir de F2

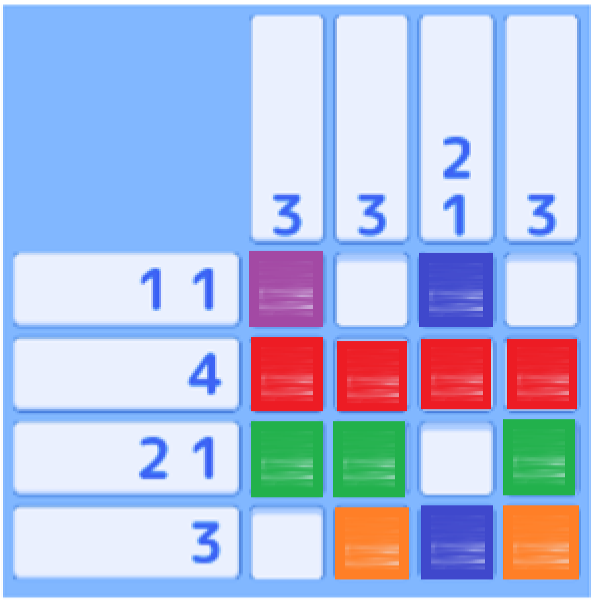
C31, C32, C34 a partir de F3

C13, C43 a partir de X3

C11 a partir de C13 y F1

!C41 a partir de C11, C21, C31 y X1

C42, C44 a partir de !C41 y F4



Ejercicios de Implementación

1. Implementar un motor de inferencia con encadenamiento hacia adelante. Pruébelo con las proposiciones del ejercicio 3.

In [21]:

```
# Motor de encadenamiento hacia adelante

from sympy import symbols, And, Implies, true

def encadenamiento_adelante_sympy(reglas_sympy, hechos_iniciales, metas=None, max_it
    """
    Dispara reglas de Horn hacia adelante

    Devuelve (hechos, traza, origen, reglas_norm):
      - hechos: set de símbolos verdaderos al finalizar
      - traza: lista de strings con la secuencia de inferencias
      - origen: dict literal -> (nombre_regla, set_antecedentes)
      - reglas_norm: lista de reglas normalizadas (dicts) para inspección/imprimir
    """
    reglas = convertir_reglas_sympy(reglas_sympy)   # convierto a (nombre, anteceden
    hechos = set(hechos_iniciales)
    traza = []
    origen = {}            # p -> (nombre_regla, antecedentes), es para imprimirlo más
    reglas_disparadas = set()  # índices de reglas ya disparadas al menos una vez
    iteraciones = 0

    for h in hechos:
        origen.setdefault(h, None)

    while True:
        hubo_cambio = False

        for i, r in enumerate(reglas):
            # si la regla ya disparó y no puede aportar nada nuevo, la salteo.
            # igual la reviso por si se incorporaron hechos y su consecuente aún no
            if r["antecedentes"].issubset(hechos):
                c = r["consecuente"]
                if c not in hechos:
                    hechos.add(c)
                    origen[c] = (r["nombre"], set(r["antecedentes"]))
                    traza.append(f'{r["nombre"]}: {conjunto_a_str(r["antecedentes"])
```

```python
                    hubo_cambio = True
                reglas_disparadas.add(i)

                # si alcanzamos metas, podemos cortar
                if metas and set(metas).issubset(hechos):
                    return hechos, traza, origen, reglas

        iteraciones += 1
        if max_iter is not None and iteraciones >= max_iter:
            break
        if not hubo_cambio:
            break

    return hechos, traza, origen, reglas


def convertir_reglas_sympy(reglas_sympy):
    """
    Toma una lista de Implies(antecedente, consecuente)
    Devuelve lista de dicts: {"nombre", "antecedentes": set, "consecuente": símbolo}
    """
    reglas = []
    contador = 1
    for item in reglas_sympy:
        if isinstance(item, tuple) and len(item) == 2:
            nombre, expr = item
        else:
            nombre, expr = None, item

        ant = expr.args[0]
        cons = expr.args[1]

        if ant is true:
            antecedentes = set()
        elif isinstance(ant, And):
            antecedentes = set(ant.args)
        else:
            antecedentes = {ant}

        reglas.append({
            "nombre": nombre if nombre else f"R{contador}",
            "antecedentes": antecedentes,
            "consecuente": cons
        })
        contador += 1

    return reglas


def conjunto_a_str(s):
    if not s:
        return "∅"
    return " ∧ ".join(sorted((str(x) for x in s)))


def imprimir_traza(traza):
    print("Traza de inferencias:")
    for paso in traza:
        print(" -", paso)


def reconstruir_prueba(meta, origen):
    """
    Devuelve pasos [(nombre_regla, antecedentes, consecuente), ...] en orden lógico
```

```
    para derivar 'meta'. Si 'meta' es hecho base, devuelve lista vacía.
    """
    visitados = set()
    pasos = []

    def expandir(lit):
        if lit in visitados:
            return
        visitados.add(lit)
        just = origen.get(lit, None)
        if just is None:
            return
        nombre_regla, antecedentes = just
        # primero me aseguro de los antecedentes
        for a in sorted(antecedentes, key=lambda x: str(x)):
            expandir(a)
        # luego agrego este paso
        pasos.append((nombre_regla, antecedentes, lit))

    expandir(meta)
    return pasos


def imprimir_prueba(meta, origen):
    pasos = reconstruir_prueba(meta, origen)
    if meta not in origen:
        print(f"No se puede reconstruir la prueba de '{meta}' (no fue derivado).")
        return
    if origen[meta] is None:
        print(f"'{meta}' ya era un hecho base.")
        return

    for i, (nombre, antecedentes, consecuente) in enumerate(pasos, 1):
        print(f"{i}- {nombre}: {conjunto_a_str(antecedentes)} ⇒ {consecuente}")


# ---------------------------
# Ejemplo de uso
# ---------------------------
if __name__ == "__main__":
    # símbolos proposicionales
    a, b, c, d, e, f, g = symbols('a b c d e f g')

    # Reglas de Horn
    reglas_sympy = [
        ("R1", Implies(And(b, c), a)),
        ("R2", Implies(And(d, e), b)),
        ("R3", Implies(And(g, e), b)),
        ("R4", Implies(e, c)),
        ("R7", Implies(And(a, g), f)),
    ]

    # Hechos iniciales
    hechos_iniciales = {d, e}

    metas = {a}

    hechos, traza, origen, reglas_norm = encadenamiento_adelante_sympy(
        reglas_sympy, hechos_iniciales, metas=metas
    )

    print("Hechos derivados:", ", ".join(sorted(str(x) for x in hechos)))
    print()
    imprimir_traza(traza)
```

```
    print("\nPrueba de la meta 'a':")
    imprimir_prueba(a, origen)
```

```
Hechos derivados: a, b, c, d, e

Traza de inferencias:
 - R2: d ∧ e ⇒ b
 - R4: e ⇒ c
 - R1: b ∧ c ⇒ a

Prueba de la meta 'a':
1- R2: d ∧ e ⇒ b
2- R4: e ⇒ c
3- R1: b ∧ c ⇒ a
```

1. Implementar un motor de inferencia con encadenamiento hacia atrás. Pruébelo con las proposiciones del ejercicio 3.

In [24]:

```
# Motor de encadenamiento hacia atrás

from sympy import symbols, And, Implies

def encadenamiento_atras_sympy(reglas_sympy, hechos_iniciales, metas=None, max_pasos
    # convierto reglas simpy a una estructura con nombre, antecedentes, consecuente
    reglas = convertir_reglas(reglas_sympy)

    # índice por consecuente para encontrar rápido qué reglas prueban un objetivo
    # indice[consecuente] -> [reglas que lo prueban]
    indice = {}
    for r in reglas:
        indice.setdefault(r["consecuente"], []).append(r)

    hechos = set(hechos_iniciales)
    traza = []
    origen = {}         # literal -> (nombre_regla, set_antecedentes)
    memo_exito = {}     # literal -> True/False (si ya se determinó su demostrabilida
    en_curso = set()    # para detectar ciclos
    pasos = 0

    def probar(objetivo):
        # si ya es un hecho, está probado
        if objetivo in hechos:
            return True

        # uso memo para no repetir trabajo
        if objetivo in memo_exito:
            return memo_exito[objetivo]

        # evito ciclos
        if objetivo in en_curso:
            memo_exito[objetivo] = False
            return False

        en_curso.add(objetivo)
        exito = False

        # intento probar todos los antecedentes recursivamente
        for r in indice.get(objetivo, []):
            puede = True
            for a in sorted(r["antecedentes"], key=lambda x: str(x)):
                if not probar(a):
                    puede = False
                    break
```

```python
            if puede:
                # registro hecho, justificación y traza
                hechos.add(objetivo)
                origen[objetivo] = (r["nombre"], set(r["antecedentes"]))
                traza.append(f'{r["nombre"]}: {conjunto_a_str(r["antecedentes"])} ⇒
                exito = True
                # corto con la primera regla que funcione
                break

        en_curso.discard(objetivo)
        memo_exito[objetivo] = exito

        # corto por tope de pasos si se pidió
        nonlocal pasos
        if exito:
            pasos += 1
            if max_pasos is not None and pasos >= max_pasos:
                return True

        return exito

    # si hay metas, intento probar cada una; si no, intento probar todas las cabecer
    if metas:
        for m in metas:
            probar(m)
    else:
        for r in reglas:
            probar(r["consecuente"])

    return hechos, traza, origen, reglas


def convertir_reglas(reglas_sympy):
    # toma Implies(And(...), cons) o Implies(lit, cons) y devuelve lista de dicts
    reglas = []
    for i, expr in enumerate(reglas_sympy, 1):
        ant = expr.args[0]
        cons = expr.args[1]
        if isinstance(ant, And):
            antecedentes = set(ant.args)
        else:
            antecedentes = {ant}
        reglas.append({
            "nombre": f"R{i}",
            "antecedentes": antecedentes,
            "consecuente": cons
        })
    return reglas


def conjunto_a_str(s):
    # imprime símbolos ordenados por nombre unidos con " ∧ "
    if not s:
        return "∅"
    return " ∧ ".join(sorted((str(x) for x in s)))


def construir_camino_hacia_atras_sympy(meta, origen, reglas, hechos_iniciales):
    """
    Devuelve líneas de texto con el camino seguido hacia atrás.
    """
    if meta not in origen and meta not in hechos_iniciales:
        return ["No se pudo justificar la meta: " + str(meta)]
```

```python
    lineas = []
    n = 1

    if meta in origen:
        nombre_meta, ants_meta = origen[meta]
        lineas.append(f"{n}- Asumimos {meta}, luego, de {nombre_meta} sigue {conjunt
        n += 1
    else:
        lineas.append(f"{n}- {meta} ya es un hecho de partida")
        return lineas

    # armo un índice de reglas base vacías (por si el usuario las usa)
    reglas_base = {}
    for r in reglas:
        if not r["antecedentes"]:
            reglas_base[r["consecuente"]] = r["nombre"]

    # recorro antecedentes de la meta
    for sub in sorted(ants_meta, key=lambda x: str(x)):
        if sub in origen:
            nombre_sub, ants_sub = origen[sub]
            lineas.append(f"{n}- Para {sub}, de {nombre_sub} sigue {conjunto_a_str(a
            n += 1

            # indico cuáles ya están dados
            base_lits = [x for x in sorted(ants_sub, key=lambda x: str(x)) if (x in
            if base_lits:
                # si hay nombres de reglas base, cítalos, si no, decí "hechos de par
                etiquetas = [reglas_base[x] for x in base_lits if x in reglas_base]
                if etiquetas:
                    etiquetas_orden = ", ".join(sorted(etiquetas))
                    lineas.append(f"{n}- {conjunto_a_str(base_lits)} están satisfech
                else:
                    lineas.append(f"{n}- {conjunto_a_str(base_lits)} están satisfech
                n += 1
        else:
            # ya era hecho de partida
            lineas.append(f"{n}- {sub} ya es un hecho de partida")
            n += 1

    lineas.append(f"{n}- Con {conjunto_a_str(ants_meta)} demostrados, por {nombre_me
    return lineas


# ---------------------------
# Ejemplo de uso
# ---------------------------
if __name__ == "__main__":
    # símbolos
    a, b, c, d, e, f, g = symbols('a b c d e f g')

    # reglas en SymPy
    reglas_sympy = [
        Implies(And(b, c), a),    # R1
        Implies(And(d, e), b),    # R2
        Implies(And(g, e), b),    # R3
        Implies(e, c),            # R4
        Implies(And(a, g), f),    # R5
    ]

    hechos_iniciales = {d, e}
    metas = {a}

    hechos, traza, origen, reglas = encadenamiento_atras_sympy(reglas_sympy, hechos_
```

```
    print("Hechos derivados:", sorted([str(x) for x in hechos]))
    print("\nTraza de justificación:")
    for t in traza:
        print(" -", t)

    print("\nCamino hacia atrás:")
    for linea in construir_camino_hacia_atras_sympy(a, origen, reglas, hechos_inicia
        print(linea)
```

```
Hechos derivados: ['a', 'b', 'c', 'd', 'e']

Traza de justificación:
 - R2: d ∧ e ⇒ b
 - R4: e ⇒ c
 - R1: b ∧ c ⇒ a

Camino hacia atrás:
1- Asumimos a, luego, de R1 sigue b ∧ c
2- Para b, de R2 sigue d ∧ e
3- d ∧ e están satisfechas por los hechos de partida
4- Para c, de R4 sigue e
5- e están satisfechas por los hechos de partida
6- Con b ∧ c demostrados, por R1 concluimos a
```

1. Implementar un motor de inferencia por contradicción que detecte si el conjunto de proposiciones del ejercicio 3 es inconsistente.

In [26]:

```
from sympy import symbols, And, Or, Not, Implies, to_cnf
from sympy.logic.inference import satisfiable

def es_inconsistente_sympy(formulas, mostrar_cnf=True):
    """
    Devuelve (inconsistente_bool, modelo, clausulas)
      - inconsistente_bool: True si no existe modelo (contradicción)
      - modelo: dict con una valuación si es satisfacible; False si no es satisfacib
      - clausulas: lista de sets de literales como strings
    """
    expr = And(*formulas)
    modelo = satisfiable(expr)  # False si es UNSAT, dict si es SAT
    inconsistente = (modelo is False)

    clausulas = []
    if mostrar_cnf:
        cnf = to_cnf(expr)
        clausulas = cnf_a_listas(cnf)

    return inconsistente, modelo, clausulas


def cnf_a_listas(expr_cnf):
    """
    Convierte una expresión CNF de SymPy en lista de cláusulas,
    cada cláusula como set de strings ("a", "¬b", ...).
    """
    # CNF es And(c1, c2, ...), cada ci es Or(lits...) o un literal suelto
    if expr_cnf.func is And:
        partes = list(expr_cnf.args)
    else:
        partes = [expr_cnf]

    clausulas = []
    for c in partes:
```

```python
        if c.func is Or:
            lits = list(c.args)
        else:
            lits = [c]
        clausulas.append({literal_a_str(l) for l in lits})
    return clausulas


def literal_a_str(lit):
    # lit puede ser un símbolo (a) o Not(a)
    if lit.func.__name__ == 'Not':
        arg = lit.args[0]
        return "¬" + str(arg)
    return str(lit)


def imprimir_cnf(clausulas):
    for i, c in enumerate(clausulas, 1):
        # ordenar por variable base y luego por signo para estética
        def clave(x):
            xb = x[1:] if x.startswith("¬") else x
            s = 1 if x.startswith("¬") else 0
            return (xb, s)
        orden = " ∨ ".join(sorted(c, key=clave))
        print(f"C{i}: {orden}")


# ---------------------------
# Ejemplo de uso (mismo ejercicio)
# ---------------------------
if __name__ == "__main__":
    # símbolos proposicionales
    a, b, c, d, e, f, g = symbols('a b c d e f g')

    # Reglas de Horn + hechos (en lógica clásica)
    formulas = [
        Implies(And(b, c), a),    # R1
        Implies(And(d, e), b),    # R2
        Implies(And(g, e), b),    # R3
        Implies(e, c),            # R4
        d,                        # R5 (hecho)
        e,                        # R6 (hecho)
        Implies(And(a, g), f),    # R7 (no afecta a la refutación de ¬a)
        Not(a)                    # NA: negación de la tesis para prueba por contradi
    ]

    inconsistente, modelo, clausulas = es_inconsistente_sympy(formulas, mostrar_cnf=

    print("CNF (cláusulas):")
    imprimir_cnf(clausulas)

    print("\n¿Conjunto inconsistente?:", inconsistente)
    if inconsistente:
        print("Conclusión: Por contradicción, a es verdadero")
    else:
        print("Conjunto consistente. Un modelo que satisface las fórmulas es:")
        # modelo puede tener valores parciales; mostramos algunos
        llaves = sorted(str(k) for k in modelo.keys())
        for k in llaves:
            print(f"  {k} = {modelo[eval(k)]}")
```

```
CNF (cláusulas):
C1: d
C2: e
```

```
C3: ¬a
C4: c ∨ ¬e
C5: a ∨ ¬b ∨ ¬c
C6: b ∨ ¬d ∨ ¬e
C7: b ∨ ¬e ∨ ¬g
C8: ¬a ∨ f ∨ ¬g

¿Conjunto inconsistente?: True
Conclusión: Por contradicción, a es verdadero
```

Bibliografía

Russell, S. & Norvig, P. (2004) *Inteligencia Artificial: Un Enfoque Moderno*. Pearson Educación S.A. (2a Ed.) Madrid, España

Poole, D. & Mackworth, A. (2023) *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press (3a Ed.) Vancouver, Canada