Trabajo Práctico 4

September 4, 2025

1 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.

2 Ejercicios Teóricos

2.1 ¿Qué es una inferencia?

La inferencia es el proceso de usar un modelo de IA ya entrenado para hacer predicciones o tomar decisiones sobre datos nuevos que no ha visto antes.

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

Verificar que la inferencia de un modelo de IA se base en su conocimiento es, en esencia, confirmar que sus predicciones son correctas y fiables. Este proceso no se hace mirando dentro del modelo como una "caja transparente", sino evaluando su desempeño con datos que nunca ha visto. Después de entrenar un modelo, se utilizan conjuntos de datos de validación y prueba que contienen preguntas con respuestas conocidas. Si el modelo hace inferencias precisas sobre estos nuevos datos, se considera que ha aprendido a generalizar de su base de conocimiento en lugar de simplemente memorizar. Para cuantificar este rendimiento, se usan métricas como la precisión, la exactitud y la matriz de confusión, que muestran con claridad dónde el modelo acierta y dónde falla.

2.3 Observe la siguiente base de conocimiento:

$R1 : b \land c \rightarrow a$

$R2 : d \land e \rightarrow b$

$R3 : g \land e \rightarrow b$

$R4 : e \rightarrow c$

$R5 : d$

$R6 : e$

$R7 : a \land g \rightarrow f$

De $R5$ y $R6$ se sabe que tanto d como e son tautologias, se podría decir que es un axioma de la Base de Conocimientos. A partir de esto, se pueden desarrollar las demás reglas.

2.3.1 ¿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?

Con el encadenamiento hacia adelante podemos probar $a =$ True con a partir de las cosas que sabemos, en este caso $R5$ y $R6$ las cuales son tautologias. Así, se razona en el orden $R2$ y $R4$ primero, luego $R1$:

$R2$: $(d \wedge e \rightarrow b)$ entonces $b =$ True

- Ya que d y e siempre son verdaderas.

$R4$: $(e \rightarrow c)$ entonces $e =$ True

- Ya que c siempre es verdadera

Entonces, puedo plantear:

$R1$: $(b \wedge c \rightarrow a)$ entonces $a =$ True

Por lo tanto, la hipótesis a puede inferirse correctamente a partir de las reglas y hechos de la base de conocimiento.

La propiedad que debe tener el algoritmo para asegurar que esta inferencia sea posible es la completitud: que todo enunciado que se sigue lógicamente de la base de conocimiento pueda, efectivamente, ser derivado por el procedimiento de inferencia.

2.3.2 ¿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?

Primeramente, se hace la hipótesis $a =$ True y se sigue con:

$R1$:

$(b \wedge c) \rightarrow a$

Entonces debo probar que b y c son verdaderos. ¿Cómo pruebo que son verdaderos? Partiendo de R2 para b y $R4$ para c:

$R2$:

$(d \wedge e) \rightarrow b$

Como d y e son verdades de BC (axiomas), entonces \$b = True" en BC.

$R4$:

$e \rightarrow c$

Como e es una verdad de BC (axioma), entonces $c =$ True en BC.

Acabamos de probar que b y c son verdaderos en BC, por lo que a deberá ser también verdadero en BC.

a = True

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

Ahora, utilizamos la regla de resolución para encontrar una contradicción, que se manifiesta como la derivación de la cláusula vacía {}.

- Paso 1: Resolvemos C5 (b) con C2 (¬b ¬a).

Al resolver b con ¬b, eliminamos ambos.

Resultado: ¬a

Esta es nuestra nueva cláusula, C7. Esto no nos ayuda a avanzar, ya que es lo mismo que C6.

¬Paso 2: Intentemos resolver C5 (b) con C1 (¬b c).

Al resolver b con ¬b, el resultado es c.

Resultado: c

Esta es nuestra nueva cláusula, C8.

- Paso 3: Resolvemos C8 (c) con C3 (¬c a).

Al resolver c con ¬c, el resultado es a.

Resultado: a

Esta es nuestra nueva cláusula, C9.

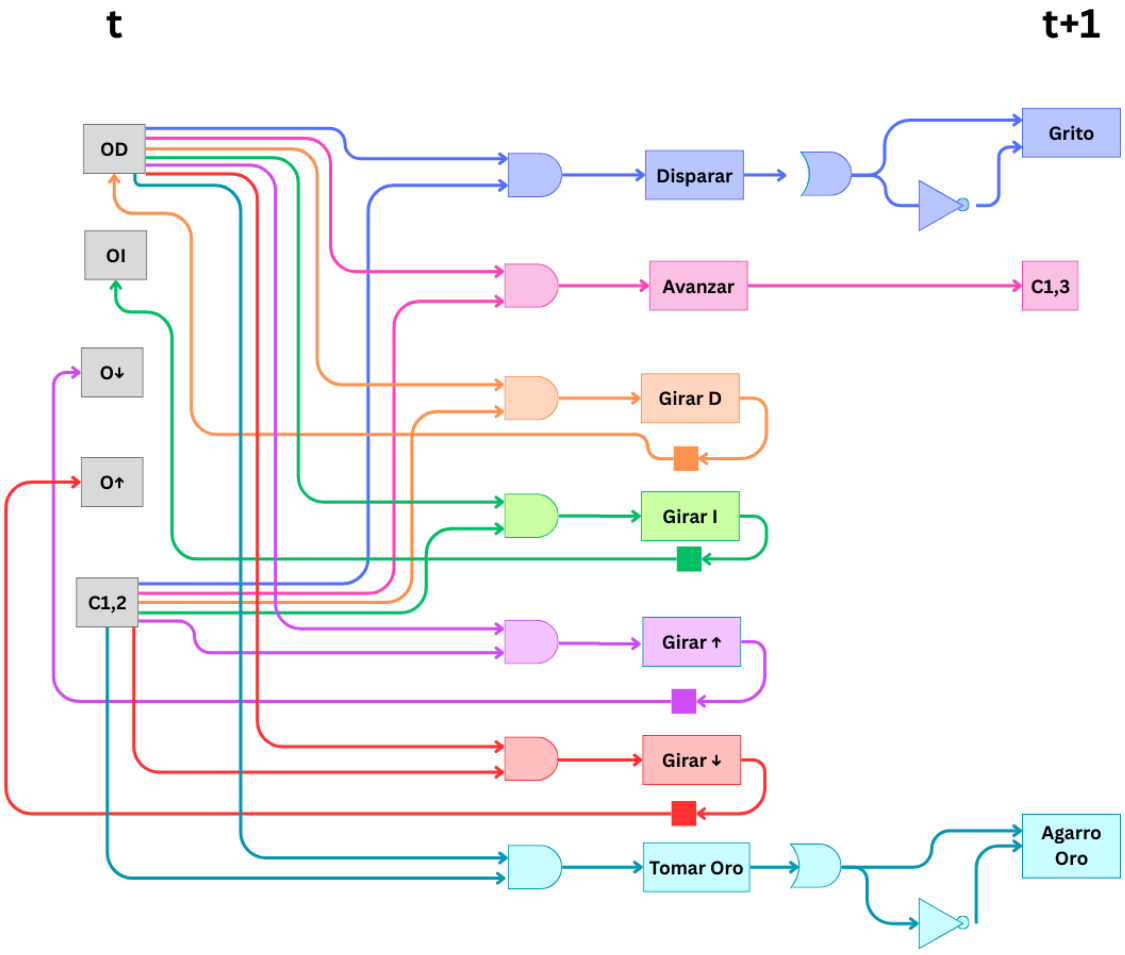
- Paso 4: Ahora, resolvemos C9 (a) con nuestra suposición inicial, C6 (¬a).

El resultado es la cláusula vacía, {}.

- Conclusión

Hemos llegado a una contradicción al derivar la cláusula vacía. Esto demuestra que nuestra suposición de que ¬a era verdadera es falsa. Por lo tanto, se ha probado que a es verdadera.

2.4 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.



2.5 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.

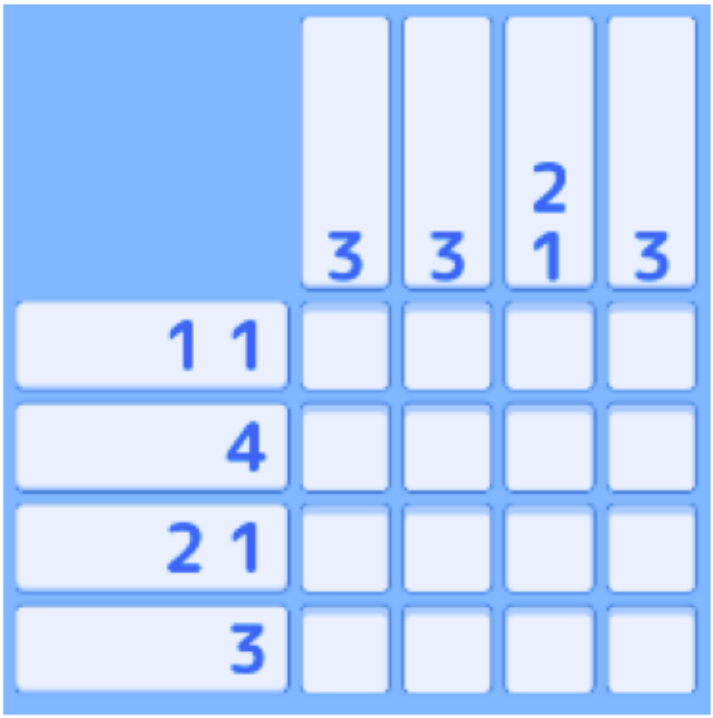
[1]:
```
import requests
from PIL import Image
from io import BytesIO
import matplotlib.pyplot as plt

# 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()
```



Vamos a identificar el tablero como una matriz de 4X4, donde cada casilla se identificara con i,j, donde i es el numero de fila (de arriba hacia abajo) y j el número de columna (de izquierda a derecha).

$C_{i,j}$ = casilla i,j pintada.

$\neg C_{i,j}$ = casilla i,j en blanco.

Definimos las reglas iniciales del tablero para cada fila y columna, mediante las combinaciones de casillas que deben pintarse:

Filas:

1. R1: $(C_{1,1} \wedge C_{1,3}) \vee (C_{1,2} \wedge C_{1,4}) \vee (C_{1,1} \wedge C_{1,4})$

2. R2: $C_{2,1} \wedge C_{2,2} \wedge C_{2,3} \wedge C_{2,4}$
3. R3: $C_{3,1} \wedge C_{3,2} \wedge C_{3,4}$
4. R4: $(C_{4,1} \wedge C_{4,2} \wedge C_{4,3}) \vee (C_{4,2} \wedge C_{4,3} \wedge C_{4,4})$

Columnas:

1. R5: $(C_{1,1} \wedge C_{2,1} \wedge C_{3,1}) \vee (C_{2,1} \wedge C_{3,1} \wedge C_{4,1})$
2. R6: $(C_{1,2} \wedge C_{2,2} \wedge C_{3,2}) \vee (C_{2,2} \wedge C_{3,2} \wedge C_{4,2})$
3. R7: $(C_{1,3} \wedge C_{2,3} \wedge C_{4,3})$
4. R8: $((C_{1,4} \wedge C_{2,4} \wedge C_{3,4}) \vee (C_{2,4} \wedge C_{3,4} \wedge C_{4,4}))$

Partimos de las filas y columnas que tienen una única posibilidad de resolución. Fila 2: indica que los 4 de la fila deben pintarse.

R2: $(C_{2,1} \wedge C_{2,2} \wedge C_{2,3} \wedge C_{2,4})$.

De esta regla se pueden derivar 4 reglas mas que indiquen que las 4 casillas deben pintarse:

- R2 $_1$: $(C_{2,1})$
- R2 $_2$: $(C_{2,2})$
- R2 $_3$: $(C_{2,3})$
- R2 $_4$: $(C_{2,4})$

Fila 3: Indica que primero se pintan dos casillas de la fila, y luego 1. La única posibilidad es que la casilla (3,3) quede en blanco, tal y como undica la regla 3:

R3: $(C_{3,1} \wedge C_{3,2} \wedge C_{3,4})$

De estas reglas se infieren otras 3:

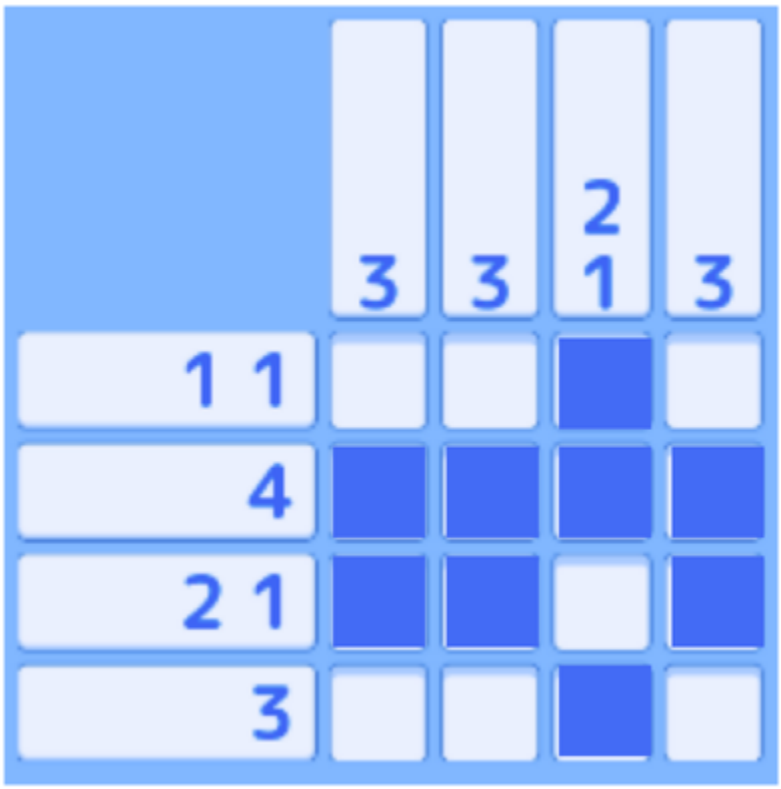
- R3 $_1$: $(C_{3,1})$
- R3 $_2$: $(C_{3,2})$
- R3 $_3$: $(C_{3,4})$

Columna 3: Indica que se deben pintar dos casillas y luego una, por lo que la unica posibilidad es que la casilla (3,3) quede sin pintar.

R7: $(C_{1,3} \wedge C_{2,3} \wedge C_{4,3})$

- R7 $_1$: $(C_{1,3})$
- R7 $_2$: $(C_{2,3})$
- R7 $_3$: $(C_{4,3})$

Hasta ahora hemos pintado todas las casillas que se infieren de las reglas iniciales:



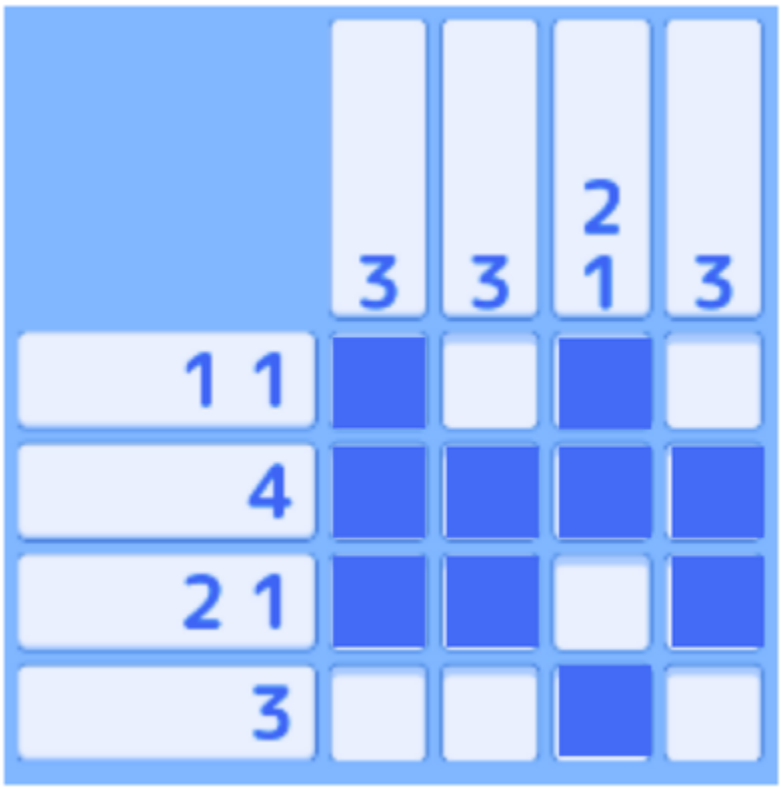
Seguimos nuestra deducción desde la regla 7.1. Si en la fila 1, la casilla (1,3) ya está pintada, sus casillas adyacentes en la fila no pueden estar pintadas, como indica la regla 1:

R1: $(C_{1,1} \wedge C_{1,3}) \vee (C_{1,2} \wedge C_{1,4}) \vee (C_{1,1} \wedge C_{1,4})$

De la regla R7 $_1$, se infieren:

- R7 $_1 \rightarrow$ R1 $_1$: $(\neg C_{1,2} \wedge \neg C_{1,4})$
- R1 $_2$: $(C_{1,1})$

Ahora hemos pintado una casilla mas, la (1,1):



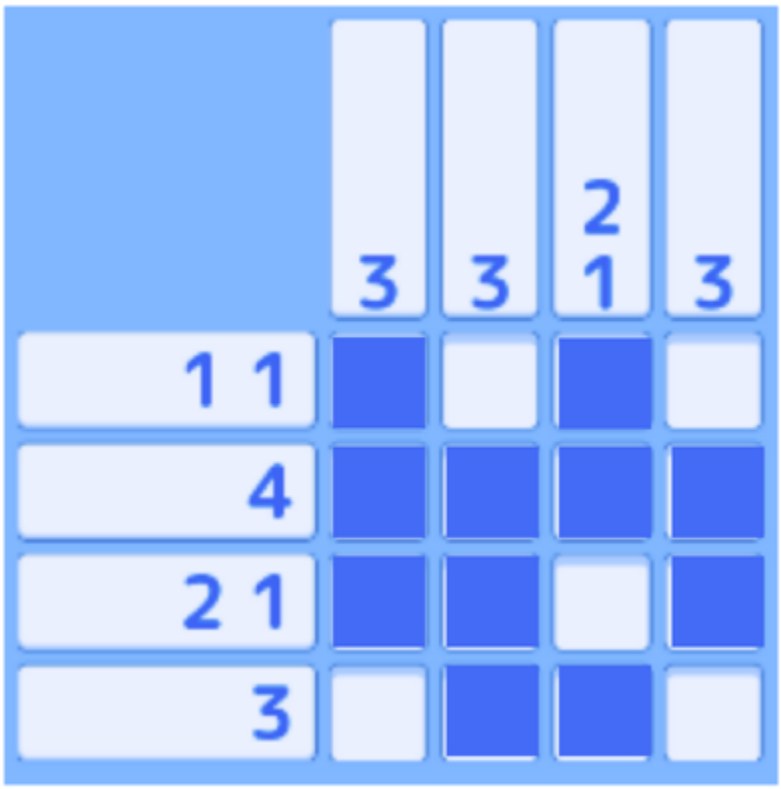
En base a la regla 1.1, si la casilla (1,2) no está pintada, significa que el resto de la columna 2 debe pintarse, ya que la condición de la columna 2 es que deben pintarse 3 casillas seguidas.

R6: $(C_{1,2} \land C_{2,2} \land C_{3,2}) \lor (C_{2,2} \land C_{3,2} \land C_{4,2})$

Luego, de la regla 1.1 se infieren:

R1 $_1 \rightarrow$ R6 $_1$: $(C_{2,2} \land C_{3,2} \land C_{4,2})$

- R6 $_1$.1: $(C_{2,2})$
- R6 $_1$.2: $(C_{3,2})$
- R6 $_1$.3: $(C_{4,2})$



Segun las reglas 1.1 y 2.1 y 3.1 las casillas (1,1), (2,1) y (3,1) se pintan. Segun la regla 5, solo tres casillas de la columna deben pintarse seguidas.

R5: $(C_{1,1} \wedge C_{2,1} \wedge C_{3,1}) \vee (C_{2,1} \wedge C_{3,1} \wedge C_{4,1})$

Es decir que la casilla 4,1 no debe pintarse, por lo que se deduce la regla:

- R5 $_1$: $\neg\ (C_{4,1})$

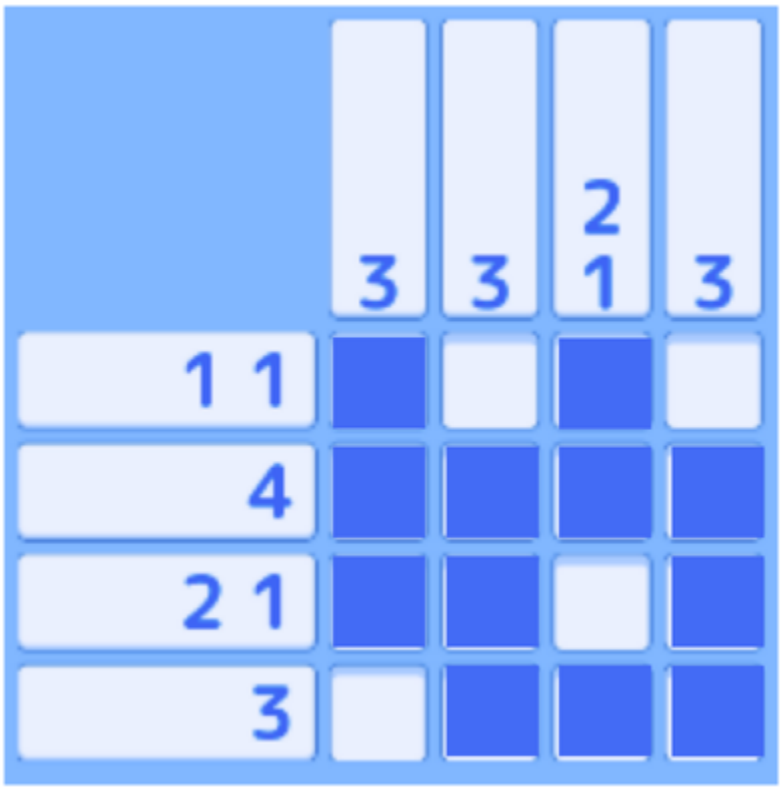
Segun la regla 7.3, la casilla (4,3) se pinta, y segun la regla 6.1.2 la casilla (4,2) tambien se pinta. Pero la regla 5.1 indica que la casilla (4,1) no se debe pintar. Segun la regla 4, en la fila 4 solo deben pintarse tres casillas seguidas.

R4: $(C_{4,1} \wedge C_{4,2} \wedge C_{4,3}) \vee (C_{4,2} \wedge C_{4,3} \wedge C_{4,4})$

Por lo tanto se infiere:

- R $_1$: $(C_{4,4})$

De esta forma, ya se ha analizado el estado de todas las casillas, por lo que queda resuelto en nanograma:



3 Ejercicios de Implementación

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

[ ]:
```python
import os

def forward_chaining(SetReglas, SetVerdades, Objetive, verbose=True):
    """
    Implementa un motor de inferencia con encadenamiento hacia adelante.

    Args:
        reglas (list): Una lista de reglas en formato (premisa, conclusion).
                       Las premisas son un conjunto, y la conclusion es un␣
 ↪string.
        hechos_iniciales (set): Un conjunto de hechos conocidos.
        objetivo (str): La proposición que se quiere demostrar.

    Returns:
        bool: True si el objetivo se puede derivar, False en caso contrario.
    """

    if SetReglas is None or SetVerdades is None or Objetive is None:
```

```
        raise ValueError("SetReglas, SetVerdades y Objetive no pueden ser None.
↪")
    elif not isinstance(SetReglas, list) or not all(isinstance(r, tuple) and␣
↪len(r) == 2 and isinstance(r[0], frozenset) and isinstance(r[1], str) for r␣
↪in SetReglas):
        raise ValueError("SetReglas debe ser una lista de tuplas (premisa,␣
↪conclusion) donde la premisa es un frozenset y la conclusion es un string.")
    elif not isinstance(SetVerdades, set) or not all(isinstance(h, str) for h␣
↪in SetVerdades):
        raise ValueError("SetVerdades debe ser un conjunto de strings.")
    elif not isinstance(Objetive, str):
        raise ValueError("Objetive debe ser un string.")


    # Por si el objetivo ya es un hecho
    if Objetive in SetVerdades:
        if verbose: print(f"Objetivo '{Objetive}' ya estaba en los hechos.")
        return True

    nombre_por_premisa = {
        frozenset({'b','c'}): 'R1',
        frozenset({'d','e'}): 'R2',
        frozenset({'g','e'}): 'R3',
        frozenset({'e'}):     'R4',
        frozenset({'a','g'}): 'R7',
    }

    nuevos_hechos_derivados = True

    # Bucle que se ejecuta mientras se puedan derivar nuevos hechos.
    while nuevos_hechos_derivados:

        # Cuando se completa el ciclo sin derivar nuevos hechos, se detiene.
        nuevos_hechos_derivados = False

        # Iterar a través de cada regla en la base de conocimiento.
        for premisa, conclusion in SetReglas:

            # Comprobar si todas las proposiciones en la premisa son hechos␣
↪conocidos.
            # Si la conclusión no es un hecho conocido, la añadimos.

            if premisa.issubset(SetVerdades) and conclusion not in SetVerdades:

                    SetVerdades.add(conclusion)
                    nuevos_hechos_derivados = True
```

```
                    regla = nombre_por_premisa.get(premisa, '?')

                    prem_str = '   '.join(sorted(premisa))

                    print(f" >>Se ha inferido '{conclusion}' a partir de␣
 ↪'{regla}'")
                    print(f"{regla}: {prem_str}   {conclusion}\n")

                    if conclusion == Objetive:
                        if verbose: print(f"Objetivo '{Objetive}' alcanzado.")
                        nuevos_hechos_derivados = False
                        return True

    # Comprobar si el objetivo está en el conjunto de hechos conocidos.
    return Objetive in SetVerdades

# --- Prueba con las proposiciones del ejercicio 3 ---

# 1. Definir la base de conocimiento como reglas
# R1: b AND c -> a
# R2: d AND e -> b
# R3: g AND e -> b
# R4: e -> c
# R7: a AND g -> f
# Las premisas son conjuntos, para manejar AND.

reglas_ejercicio3 = [
    (frozenset(['b', 'c']), 'a'),
    (frozenset(['d', 'e']), 'b'),
    (frozenset(['g', 'e']), 'b'),
    (frozenset(['e']), 'c'),
    (frozenset(['a', 'g']), 'f')
]

# 2. Definir los hechos iniciales (del ejercicio 3)
# R5: d
# R6: e
# g no se menciona, por lo que no es un hecho.
hechos_ejercicio3 = {'d', 'e'}

# 3. Definir el objetivo
objetivo_a = 'a'



if __name__ == "__main__":
```

```
    clear = lambda: os.system('cls' if os.name == 'nt' else 'clear')
    clear()

    # 4. Ejecutar el motor de inferencia
    print("--- Base de Conocimiento ---")

    for s in ["R1: b   c → a",
              "R2: d   e → b",
              "R3: g   e → b",
              "R4: e → c",
              "R5: d",
              "R6: e",
              "R7: a   g → f"]:
        print(s)

    print("\nIniciando motor de inferencia con encadenamiento hacia adelante...
↪\n")
    resultado = forward_chaining(reglas_ejercicio3, hechos_ejercicio3,␣
↪objetivo_a)

    # 5. Imprimir el resultado
    print("\n\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
    print("| \t\t\t Resultado \t\t\t      |")
    print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
    if resultado:
        print(f"\n >>Éxito: Se ha demostrado que '{objetivo_a}' es verdadero.")
    else:
        print(f"\n >>Fallo: No se pudo demostrar que '{objetivo_a}' es␣
↪verdadero.")
```

```
--- Base de Conocimiento ---
R1: b   c → a
R2: d   e → b
R3: g   e → b
R4: e → c
R5: d
R6: e
R7: a   g → f

Iniciando motor de inferencia con encadenamiento hacia adelante…

Se ha inferido 'b' a partir de 'R2'
R2: d   e   b

Se ha inferido 'c' a partir de 'R4'
R4: e   c
```

```
Se ha inferido 'a' a partir de 'R1'
R1: b   c   a

Objetivo 'a' alcanzado.

--- Resultado ---
Éxito: Se ha demostrado que 'a' es verdadero.
```

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

[5]:
```python
import os

def backward_chaining(SetReglas, SetVerdades, Objetive):
    """
    Implementa un motor de inferencia con encadenamiento hacia atrás.

    Args:
        SetReglas (list): Una lista de reglas en formato (premisa, conclusion).
                        Las premisas son un conjunto, y la conclusion es un string.
        SetVerdades (set): Un conjunto de hechos conocidos.
        Objetive (str): La proposición que se quiere demostrar.

    Returns:
        bool: True si el objetivo se puede derivar, False en caso contrario.
    """

    if SetReglas is None or SetVerdades is None or Objetive is None:
        raise ValueError("SetReglas, SetVerdades y Objetive no pueden ser None.")
    elif not isinstance(SetReglas, list) or not all(isinstance(r, tuple) and len(r) == 2 and isinstance(r[0], frozenset) and isinstance(r[1], str) for r in SetReglas):
        raise ValueError("SetReglas debe ser una lista de tuplas (premisa, conclusion) donde la premisa es un frozenset y la conclusion es un string.")
    elif not isinstance(SetVerdades, set) or not all(isinstance(h, str) for h in SetVerdades):
        raise ValueError("SetVerdades debe ser un conjunto de strings.")
    elif not isinstance(Objetive, str):
        raise ValueError("Objetive debe ser un string.")


    # 1. Caso base: El objetivo ya es un hecho conocido.
    if Objetive in SetVerdades:
        print(f"\n >>Éxito: El objetivo '{Objetive}' ya es un hecho conocido.")
        return True
```

```
    nombre_por_premisa = {
        frozenset({'b','c'}): 'R1',
        frozenset({'d','e'}): 'R2',
        frozenset({'g','e'}): 'R3',
        frozenset({'e'}):     'R4',
        frozenset({'a','g'}): 'R7',
    }

    # 2. Búsqueda de reglas que pueden probar el objetivo.
    RulesAvaible = [regla for regla in SetReglas if regla[1] == Objetive]

    # 3. Si no hay reglas para probar el objetivo, la prueba falla.
    if not RulesAvaible:
        print("\n------------------------------------------------------------")
        print(f"Fallo: No hay reglas para probar el objetivo '{Objetive}'.")
        print("------------------------------------------------------------\n")
        return False

    # 4. Intentar probar el objetivo a través de cada regla posible.
    for premisa, _ in RulesAvaible:

        RuleName = nombre_por_premisa.get(premisa, '?')

        print(f"\nIntentando probar '{Objetive}' usando la regla '{RuleName}'")
        print(f"{RuleName}: {'   '.join(sorted(premisa))} → {Objetive}")

        flag_subobjetivos = True

        # 5. Intentar probar cada subobjetivo en la premisa.
        for sub_objetivo in premisa:
            # Llamada recursiva para probar el subobjetivo.
            if not backward_chaining(SetReglas, SetVerdades, sub_objetivo):
                flag_subobjetivos = False
                break

        # 6. Si todos los subobjetivos se probaron, el objetivo es verdadero.
        if flag_subobjetivos:
            print(f"\n >>Éxito: Se ha demostrado que '{Objetive}' es verdadero␣
 ↪usando la regla '{RuleName}'.")
            return True

    # 7. Si ninguna regla pudo probar el objetivo, la prueba falla.
    return False

# --- Prueba con las proposiciones del ejercicio 3 ---
```

```python
# 1. Definir la base de conocimiento como reglas
# R1: b AND c -> a
# R2: d AND e -> b
# R3: g AND e -> b
# R4: e -> c
# R7: a AND g -> f
# Las premisas son conjuntos, para manejar AND.

reglas_ejercicio3 = [
    (frozenset(['b', 'c']), 'a'),
    (frozenset(['d', 'e']), 'b'),
    (frozenset(['g', 'e']), 'b'),
    (frozenset(['e']), 'c'),
    (frozenset(['a', 'g']), 'f')
]

# 2. Definir los hechos iniciales (del ejercicio 3)
# R5: d
# R6: e
hechos_ejercicio3 = {'d', 'e'}

# 3. Definir el objetivo
objetivo_a = 'a'

if __name__ == "__main__":

    clear = lambda: os.system('cls' if os.name == 'nt' else 'clear')
    clear()


    # 4. Imprimir la base de conocimiento
    print("--- Base de Conocimiento ---")
    for s in ["R1: b   c → a",
          "R2: d   e → b",
          "R3: g   e → b",
          "R4: e → c",
          "R5: d",
          "R6: e",
          "R7: a   g → f"]:
        print(s)

    # 5. Ejecutar el motor de inferencia
    print("\nIniciando motor de inferencia con encadenamiento hacia atrás...\n")
    resultado = backward_chaining(reglas_ejercicio3, hechos_ejercicio3,␣
 ↪objetivo_a)
```

```
    # 5. Imprimir el resultado
    print("\n\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
    print("| \t\t\t Resultado \t\t\t      |")
    print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
    if resultado:
        print(f"\n >>Éxito: Se ha demostrado que '{objetivo_a}' es verdadero.")
    else:
        print(f"\n >>Fallo: No se pudo demostrar que '{objetivo_a}' es␣
↪verdadero.")
```

```
--- Base de Conocimiento ---
R1: b   c → a
R2: d   e → b
R3: g   e → b
R4: e → c
R5: d
R6: e
R7: a   g → f

Iniciando motor de inferencia con encadenamiento hacia atrás…


Intentando probar 'a' usando la regla 'R1'
R1: b   c → a

Intentando probar 'c' usando la regla 'R4'
R4: e → c

 >>Éxito: El objetivo 'e' ya es un hecho conocido.

 >>Éxito: Se ha demostrado que 'c' es verdadero usando la regla 'R4'.

Intentando probar 'b' usando la regla 'R2'
R2: d   e → b

 >>Éxito: El objetivo 'd' ya es un hecho conocido.

 >>Éxito: El objetivo 'e' ya es un hecho conocido.

 >>Éxito: Se ha demostrado que 'b' es verdadero usando la regla 'R2'.

 >>Éxito: Se ha demostrado que 'a' es verdadero usando la regla 'R1'.


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
|                         Resultado                            |
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
>>Éxito: Se ha demostrado que 'a' es verdadero.
```

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

[ ]:
```python
import itertools

def to_cnf(regla):
    """
    Convierte una regla simple (implicación) a FNC.
    Asume que la premisa es una conjunción de literales.
    """
    premisa, conclusion = regla
    literales_premisa = [f"¬{p}" for p in premisa]

    # La implicación (A -> B) se convierte en ¬A $\lor$ B
    # donde A es la conjunción de la premisa.
    return frozenset(literales_premisa + [conclusion])

def resolver_clausulas(c1, c2):
    """
    Aplica la regla de resolución a dos cláusulas.
    Retorna la cláusula resuelta o None si no se pueden resolver.
    """
    for literal in c1:
        negado = f"¬{literal}" if not literal.startswith("¬") else literal[1:]
        if negado in c2:
            nueva_clausula = (c1 - {literal}) | (c2 - {negado})
            return nueva_clausula
    return None

def formato_clausula(clausula):
    """
    Función auxiliar para formatear una cláusula para una salida legible.
    Convierte un frozenset a una cadena de texto.
    """
    if not clausula:
        return " (Cláusula Vacía)"
    return " $\lor$ ".join(sorted(list(clausula)))

def motor_inconsistencia_resolucion(clausulas):
    """
    Motor de inferencia que detecta inconsistencias usando resolución.

    Args:
        clausulas (list): Una lista de conjuntos inmutables (frozenset) de␣
 ↪literales.
```

```
    Returns:
        bool: True si el conjunto es inconsistente, False en caso contrario.
    """
    clausulas_conocidas = set(clausulas)

    print("Analizando la base de conocimiento para detectar inconsistencia...")

    # --- Nuevo: Imprimir la base de conocimiento inicial ---
    print("\nBase de conocimiento inicial:")
    for i, c in enumerate(clausulas):
        print(f"  Cláusula {i+1}: ({formato_clausula(c)})")
    print("-" * 30)

    while True:
        nuevas_clausulas = set()
        pares_clausulas = itertools.combinations(clausulas_conocidas, 2)

        for c1, c2 in pares_clausulas:
            resultado_resolucion = resolver_clausulas(c1, c2)

            if resultado_resolucion is not None:
                print(f"Resolviendo ({formato_clausula(c1)}) y␣
↪({formato_clausula(c2)}) -> ({formato_clausula(resultado_resolucion)})")

                if not resultado_resolucion:
                    print("\n¡Inconsistencia detectada! Se ha derivado la␣
↪cláusula vacía.")
                    return True

                if resultado_resolucion not in clausulas_conocidas:
                    nuevas_clausulas.add(resultado_resolucion)

        if not nuevas_clausulas:
            print("\nNo se encontraron nuevas cláusulas. El conjunto de␣
↪proposiciones es consistente.")
            return False

        clausulas_conocidas.update(nuevas_clausulas)

# --- Prueba con un conjunto inconsistente ---
clausulas_base = [
    frozenset(['¬b', '¬c', 'a']),
    frozenset(['¬d', '¬e', 'b']),
    frozenset(['¬g', '¬e', 'b']),
    frozenset(['¬e', 'c']),
    frozenset(['d']),
```

```
    frozenset(['e'])
]

# Añadimos la proposición que hace que el conjunto sea inconsistente.
clausulas_inconsistentes = clausulas_base + [frozenset(['¬c'])]

if __name__ == "__main__":
    print("--- Resultado ---")
    es_inconsistente = motor_inconsistencia_resolucion(clausulas_inconsistentes)
    if es_inconsistente:
        print("\nEl conjunto de proposiciones es inconsistente. ¡Prueba exitosa!
 ↪ ")
    else:
        print("\nEl conjunto de proposiciones es consistente.")
```

4 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