Representación del conocimiento
Práctica 4

Víctor Castañeda Balmori, Mario Cuesta Rivavelarde,
Carlos García Arenal, Laro Ayesa Sánchez y Alexandru Solovei Popa

03/12/2025

En esta memoria explicaremos cómo hemos logrado alcanzar el objetivo de la práctica implementando un algoritmo que nos permita aplicar modus ponens en lógica proposicional y, posteriormente, decidir si el encadenamiento es completo. En este documento explicaremos con detalle nuestra implementación que se puede encontrar en la carpeta comprimida *practica4.zip*.

1. Algoritmo de encadenamiento

Para la codificación hemos creado las clases Proposition, And, Or, Not e Impl que heredan de Formula, por lo que comparten su estructura básica. Cada una implementa su propia forma de mostrar y evaluar la fórmula lógica.

```
def chaining(BC):
    proposiciones = set()
    implicaciones = list()
    BCnew = list(BC)

    for formula in BC:
        if isinstance(formula,Impl):
            implicaciones.append(formula)
        else:
            proposiciones.add(formula.to_string())

    hay_nueva_proposicion = True

    while(hay_nueva_proposicion):
        hay_nueva_proposicion = False
        for i in range(0,len(implicaciones)):
            necesarias = evaluar_proposiciones(implicaciones[i].
                head())

            if(necesarias is not None and necesarias.issubset(
                proposiciones)):
```

```
                proposiciones.add(implicaciones[i].tail().
                    to_string())
                formulas_tail = set()
                separar_and(implicaciones[i].tail(),
                    formulas_tail)
                for prop in formulas_tail:
                    if prop.to_string() not in proposiciones:
                        BCnew.append(prop)
                        proposiciones.add(prop.to_string())
                implicaciones.pop(i)
                hay_nueva_proposicion = True
                break
    return BCnew
```

Aquí mostramos el código de nuestra función principal que aplica modus ponens. Como bien sabemos, la regla del modus ponens nos permite derivar κ si encontramos φ en nuestra BC.

$$\left\{ \begin{array}{c} \varphi \to \kappa \\ \varphi \\ \hline \kappa \end{array} \right\} P$$

Para la implementación, hemos tenido en cuenta lo siguiente:

1. Verificación de la premisa (φ): Para que la regla se dispare, buscamos en la BC una fórmula que coincida sintácticamente con φ.

2. Tratamiento del consecuente (κ): Al derivar κ, distinguimos dos situaciones:

   a) Si κ es una conjunción (AND): Descomponemos la fórmula y añadimos cada proposición por separado a la BC. De esta manera, si tenemos una premisa con una o varias AND, la separamos κ sintácticamente. Esto los podemos hacer pues $p \to (s \wedge r) \equiv (p \to (s)) \wedge (p \to (r))$

   b) Caso general: Añadimos κ tal cual a la BC.

Una vez aclarado esto, explicaremos brevemente nuestra función chaining. Primero, recorremos la BC y separamos implicaciones (reglas que se dispararán) del resto de fórmulas. Después, iniciamos un bucle en el que seguiremos iterando mientras sigamos deduciendo nuevos hechos. A continuación, usamos el método auxiliar `evaluar_proposiciones()` para sacar los antecedentes de cada regla,es decir, separamos las AND, y comprobamos si son un subconjunto de lo que conocemos de la BC. Si la regla se dispara, llamamos a `separar_and()` que nos permite ser fieles a la idea planteada previamente en el punto 2.a , añadiremos cada proposición que forme la AND individualmente y en el resto de casos añadiremos la fórmula tal cual a la BC. Finalmente, eliminamos la regla disparada para evitar bucles infinitos.
Nota: las implementaciones de los métodos auxiliares pueden verse en el archivo *practica4* de la carpeta zip que hemos entregado.

2. Comprobación de Completitud

```
ef es_completo(bc, variables):

    derivados_objetos = chainingv2(bc)

    proposiciones_derivadas = set()
    for f in derivados_objetos:
        if isinstance(f, Proposition):
            proposiciones_derivadas.add(f.to_string())

    print(f"Es cierto segun el encadenamiento: {
       proposiciones_derivadas}")

    # Obtenemos lo que es cierto, por defecto decimos que todo
       es cierto
    consecuencias_logicas = set(v.to_string() for v in variables
       )

    # Combinaciones de valores para las variables
    rangos = [range(2) for _ in variables]
    hay_modelos_validos = False

    for combinacion in product(*rangos):
        # Asignamos los valores a las variables
        for i, val in enumerate(combinacion):
            variables[i].set_value(bool(val))

        # Verificamos si la BC es verdadera bajo esta asignación
            (flitramos filas de la tabla)
        es_modelo_valido = True
        for formula in bc:
            if not formula.get_value():
                es_modelo_valido = False
                break

        # Si la BC es verdadera (es un modelo válido), miramos
           qué variables son verdaderas
        if es_modelo_valido:
            hay_modelos_validos = True
            vars_verdaderas_en_modelo = set()
            for v in variables:
                if v.get_value():
                    vars_verdaderas_en_modelo.add(v.to_string())

            # La intersección mantiene solo las variables que
               son TRUE en TODOS los modelos válidos encontrados
                hasta ahora
            consecuencias_logicas = consecuencias_logicas.
               intersection(vars_verdaderas_en_modelo)

    # aqui miramos si hay modelos validos, si no los hay la BC
       es mala, consideramos que es completo porque no deriva
```

```
            nada
43      if not hay_modelos_validos:
44          print("La base de conocimiento es contradictoria (
                siempre falsa).")
45          return True
46
47      print(f"La lógica dicta que es verdad: {
            consecuencias_logicas}")
48
49      # Comparamos lo que es logicamente cierto con lo que
            derivamos
50      faltantes = consecuencias_logicas - proposiciones_derivadas
51
52      if len(faltantes) > 0:
53          print(f"INCOMPLETO. El algoritmo no pudo derivar: {
                faltantes}")
54          return False
55      else:
56          print("COMPLETO. El algoritmo derivó todas las
                consecuencias lógicas.")
57          return True
```

Para comprobar si la solución es completa hemos evaluado todas las posibles combinaciones de las proposiciones. De esta podemos saber que modelos son validos, es decir, las combinaciones que al evaluarse devuelven verdadero. En caso de no tener modelos válidos, la base del conocimiento es contradictoria.

De los modelos válidos hemos hecho la interseccion para saber que variables siempre son ciertas, el resultado de esta intersección seran consecuencias lógicas de nuestra base del conocimiento. Por ultimo, se comparan los resultados para saber si la solución del encadenamiento ha sido completa.

2.1. Ejemplos

En cuánto a los ejemplos, hemos seleccionado los vistos en clase.
Por un lado:
(Llueve $\vee$ Nieva $\rightarrow$ Mojado) $\wedge$ Llueve
Como es de esperar, nuestro algoritmo devuelve que es INCOMPLETO porque no hemos sido capaces de derivar Mojado. Para derivar mojado tendríamos que tener (Llueve $\vee$ Nieva) pero solo tenemos Llueve. Por lo tanto, la regla no se dispara y no podemos derivar mojado.

También hemos probado el siguiente ejemplo:

$$(A \wedge C \rightarrow E) \ \wedge \ (B \rightarrow C) \ \wedge \ (B \wedge D \rightarrow F) \ \wedge \ (C \rightarrow D) \ \wedge \ B$$

En este caso, nuestro algoritmo devuelvo que es COMPLETO, que coincide con el resultado esperado, ya que, como vimos en clase tras disparar las reglas $(B \rightarrow C)$, $(C \rightarrow D)$ y $(B \wedge D \rightarrow F)$ conseguimos disparar F, y por tanto, concluimos en que es completo