Representación del Conocimiento

Práctica 1: Encadenamiento hacia delante y Completitud

Arahí Fernández Monagas

Iker Martínez Gómez

Carlos Velázquez Fernández

Grado en Ingeniería
Informática
Mención en Computación
Curso 2023-2024

1 Fundamente teóricos

El objetivo de esta práctica es implementar el encadenamiento hacia delante en lógica proposicional y desarrollar un algoritmo que decida si el encadenamiento es completo o no. Para esto es necesario explicar una serie de términos con los que trabajaremos a lo largo de la práctica.

Cláusula de Horn: Una cláusula de Horn en lógica proposicional se define como una cláusula (disyunción de literales) la cual contiene como máximo un literal positivo.

$$\neg\, p \ \vee\ \neg\, q \ \vee\ \ldots\ \vee\ \neg\, t \ \vee\ u$$

lo que también se traduce en:

$$(p \ \wedge\ q \ \wedge\ \ldots\ \wedge\ t) \ \rightarrow\ u$$

La cláusulas de Horn que nos interesan son:

$$\top \ \rightarrow\ p \text{ es un hecho}$$
$$p_1 \ \wedge\ p_2 \ \wedge\ \ldots\ \wedge\ p_n \ \rightarrow\ q \text{ es una regla}$$
$$p_1 \ \wedge\ p_2 \ \wedge\ \ldots\ \wedge\ p_n \ \rightarrow \bot \text{ es un objetivo}$$

Modus ponens: Es una regla de inferencia de la lógica proposicional tal que si tenemos un conjunto de premisas P compuesto por una regla de la forma $\varphi \ \rightarrow\ \kappa$ y tenemos φ como hecho, es posible derivar κ.

Encadenamiento de reglas: Al disparar la regla del modus ponens nuestra base de conocimiento (BC) aumenta. Esto significa que ahora puede existir la posibilidad de activar nuevas reglas que anteriormente no fue posible. Esto se le conoce como encadenamiento.

Algorithm 1 Encadenamiento de reglas

while $\exists\, \varphi\ \exists\, \kappa\ [\{\varphi \ \rightarrow\ \kappa\ ,\ \varphi\} \ \subseteq\ BC$ **and** $\kappa \ \notin\ BC]$ **do**
  $BC \ \leftarrow\ BC \ \cup\ \{\kappa\}$
end while
return $\psi \ \in\ BC$

Encadenamiento correcto o completo: Para un encadenamiento de reglas $BC \vdash_i psi$: si $BC \ \vdash_i\ \psi$.

Si y solo si $\forall\, BC\ \forall\, \psi\ [BC \ \vdash\ \psi \implies BC \ \vDash\ \psi\ ]$ se dice que i es correcto.
Si y solo si $\forall\, BC\ \forall\, \psi\ [BC \ \vDash\ \psi \implies BC \ \vdash\ \psi\ ]$ se dice que i es completo.

En otras palabras, considerando todo aquello que se puede derivar semánticamente como $BC \vDash \psi_1$ y todo lo derivable sintácticamente como $BC \vdash \psi_2$, podemos diferenciar distintos casos:

- Si $\psi_2 \subset \psi_1$, el encadenamiento es correcto pero no completo.
- Si $\psi_1 \subset \psi_2$, el encadenamiento es completo pero no correcto.
- Si $\psi_1 = \psi_2$, el encadenamiento es correcto y completo a la vez.
- Si no se cumple ninguna de las condiciones anteriores, entonces el encadenamiento no es ni completo ni correcto

2 Codificación de los algoritmos

2.1 Encadenamiento hacia delante

En la práctica hemos programado una función *encadenamiento* la cual recibe dos argumentos, cada uno de ellos siendo un conjunto: *reglas*[1] y *hechos*. Con objetivo de facilitar el trabajo al usuario, se han desarrollado dos funciones auxiliares *stringToSet* y *setToString*, que realizan transformaciones entre strings y las estructuras de datos utilizadas por programa. De esta forma, solo es necesario introducir una BC en formato de string y con los siguientes caracteres:

- Caracteres alfanuméricos como nombres de literales
- Acento circunflejo ($\hat{}$) para conjunciones.
- Letra v para disyunciones.
- Virgulilla ($\sim$) para negaciones.
- Guión y mayor que ($->$) para implicaciones.
- Espacios y paréntesis.

Las reglas son recorridas en un bucle *while* cuya condición de salida será que no se haya disparado el modus ponens sobre ninguna regla en la iteración anterior. En cada iteración se observan las reglas una a una y se comprueba si tenemos sus antecedentes en la BC. En caso de tenerlos, y no estar ya el consecuente en la BC, éste se añade a ella. Cada vez que una regla es disparada, sabemos que no se va a volver a utilizar (ya tenemos el consecuente en BC), por lo que podemos descartarla[2]. Al salir del bucle, el algoritmo retorna la base de conocimiento resultante.

Por último, cabe mencionar que contamos con otra función auxiliar *compruebaFNC*, la cual recibe como argumento una fórmula ψ y comprueba si está en forma normal conjuntiva (FNC). Dependiendo de la respuesta, se devolverá la misma fórmula en un formato u otro. Esto nos permitirá añadir correctamente los hechos a la BC independientemente de si ψ se compone de cláusulas de Horn o no.

2.2 Completitud del encadenamiento

Para determinar si un encadenamiento es completo o no, necesitamos comprobar qué hechos podemos derivar mediante los modelos de la fórmula (derivaciones semánticas) y cuáles podemos derivar mediante el mismo (derivaciones sintácticas). Si empleando el encadenamiento logramos que el primer conjunto esté contenido o sea igual al segundo, entonces el encadenamiento es completo.

En el fichero de la práctica se ha programado una función *esCompleto* que recibe como único argumento un encadenamiento en forma de string (ej.: "$(A \vee B \rightarrow C) \wedge A$")) y devuelve Verdadero o Falso en función de si éste es completo o no. El funcionamiento general es el siguiente:

Primero se analiza el string en busca de todas las variables distintas y una vez obtenidas se calcula el tamaño de la tabla de verdad. Al poder tomar cada variable dos valores diferentes (0, 1), esta tabla será una matriz de 2^n filas por $n+1$ columnas, siendo n el número total de variables[3]. Siguiendo el ejemplo anterior, la matriz resultante sería:

[1] En el código, cada elemento del conjunto se representa de la forma (X, Y), donde X es el antecedente e Y el consecuente.

[2] En el código, se almacenan en un nuevo conjunto *reglasDisparadas*. Al iterar sobre el conjunto de reglas, se comprueba si estas pertenecen o no a *reglasDisparadas* antes de trabajar con ellas.

[3] En la implementación se ha optado por el uso de la librería *numpy*.

i	$\mu_i(A)$	$\mu_i(B)$	$\mu_i(C)$	$\mu_i(BC)$
1	0	0	0	-
2	0	0	1	-
3	0	1	0	-
4	0	1	1	-
5	1	0	0	-
6	1	0	1	-
7	1	1	0	-
8	1	1	1	-

representando en cada fila el valor de verdad asignado a cada literal, es decir, $\mu(\varphi)$. El último elemento de la fila será $\mu(BC)$ (el valor de la base de conocimiento o encadenamiento para ese modelo), lo cual deberemos calcular.

Para obtener el $\mu(BC)$ correspondiente a cada fila, únicamente es necesario sustituir las variables por su propio valor en ese modelo y evaluar la expresión resultante. Sin embargo, en python deberemos dar pasos extra para traducir la fórmula a algo comprensible por el intérprete. Los símbolos usados en lógica proposicional deberán ser reemplazados por otros:

- $P \wedge Q$ (representado como P ^ Q): se ha sustituido por P & Q.
- $P \vee Q$ (representado como P v Q): se ha sustituido por P | Q.
- $\neg P$ (representado como $\sim P$): se ha sustituido por *(not P)*[4].
- $P \rightarrow Q$ (representado como P -> Q): se ha sustituido por *((P & Q) | (not P))*[5].

Con la nueva fórmula expresada de esta forma (((*A* & *B*) | (*not A*)) & *A*), sustituimos por los valores de la tabla de verdad y comprobamos el resultado. Ahora la última columna de la matriz tomará los valores:

i	$\mu_i(A)$	$\mu_i(B)$	$\mu_i(C)$	$\mu_i(BC)$
1	0	0	0	0
2	0	0	1	0
3	0	1	0	0
4	0	1	1	0
5	1	0	0	0
6	1	0	1	1
7	1	1	0	0
8	1	1	1	1

Aquí podemos observar que los modelos de la fórmula (aquellos modelos para los que la fórmula evalúa a verdadero) son el sexto y el octavo, es decir, $M(BC) = \{\mu_6, \mu_8\}$. Podemos derivar como hechos aquellas variables que se cumplen para todo M(BC), i.e., las variables cuyo valor sea igual a 1 en μ_6 y μ_8 simultáneamente[6]. En este caso son A y C, lo cual analizando la fórmula original $(A \vee B \rightarrow C) \wedge A)$, tiene mucho sentido. Si sabemos que cuando se cumplen o A, o B, o las dos, tenemos C, y también sabemos que se cumple A, sí o sí sabemos que vamos a tener C, independientemente del valor de B. Dado que A ya lo teníamos como hecho, podemos decir que $BC \vDash C$.

Al usar modus ponens sobre esta misma fórmula, nos surge un problema. Si consideremos $(A \vee B)$ como φ y C como κ, nos encontramos con que tenemos $\varphi \rightarrow \kappa$. Sin embargo, al no tener como hecho

[4]Con ayuda de la función auxiliar *replaceNot*.
[5]Con ayuda de la función auxiliar *replaceImplication*.
[6]En el código se hace un AND lógico de todas las filas cuyo $\mu_i(BC) == 1$.

φ completo, no podemos derivar de ninguna manera κ, es decir, $BC \nvdash C$.

Nuestro algoritmo de completitud realiza los pasos anteriores y, llegado a este punto, compara si el encadenamiento ha sido capaz de derivar todo aquello que también es posible derivar mediante la tabla de modelos. Si esto se cumple, podemos decir que el encadenamiento es completo.

3 Demostración del funcionamiento

Para demostrar de manera efectiva el funcionamiento de los algoritmos de encadenamiento y de completitud, se ha elaborado un test que prueba los diferentes casos los que nos podemos encontrar. Estos casos, resumidos de una forma general, son:

- **Las reglas y los hechos son cláusulas Horn**

  Reglas y hechos son expresiones únicamente con implicaciones y conjunciones. El encadenamiento debe ser capaz de derivar todo lo que se puede derivar semánticamente, es decir, será siempre completo: $\{q \mid BC \vdash_i q\} = \{q \mid BC \models_i q\}$.

- **Las reglas y los hechos no tienen por qué ser cláusulas de Horn**

  Si las reglas son cláusulas Horn, pero los hechos no lo son, el algoritmo pierde eficacia. La introducción de disyunciones y negaciones provoca que puedan existir diferencias entre las derivaciones, por lo que podría ocurrir que $\{q \mid BC \vdash_i q\} \neq \{q \mid BC \models_i q\}$.

Para comprobar que el programa funciona correctamente dado cualquier input, se han testado todas las combinaciones de reglas y hechos siendo o no cláusulas de Horn. Para cada una se han preparado ejemplos en los que el encadenamiento sea completo y otros en los que no lo sea. En total se han reunido 16 ejemplos a probar, todos siguiendo la estructura que se presenta a continuación:

Se define una base de conocimiento x y se utiliza la función *stringToSet* para convertirla en un conjunto de reglas y hechos iniciales. A continuación, se aplica el encadenamiento hacia adelante con las reglas y hechos proporcionados, y el resultado se almacena en una variable *enc*. Se verifica si el encadenamiento es completo utilizando la función *esCompleto(enc)*, y finalmente, se imprime en la consola el resultado del encadenamiento, utilizando una aserción para garantizar que el resultado ha sido el esperado.

Es fundamental probar todos los casos posibles, tanto aquellos en los que las cláusulas son de Horn como aquellos en los que no lo son, al evaluar y validar un programa de encadenamiento hacia adelante en lógica proposicional. En primer lugar, garantiza que el programa sea general y capaz de abordar una amplia variedad de fórmulas. La prueba exhaustiva también revela posibles errores o excepciones en el programa, lo que es crucial para detectar y corregir problemas. Asimismo, la evaluación de diferentes casos proporciona información sobre el rendimiento del programa, permitiendo identificar posibles cuellos de botella o ineficiencias en términos de complejidad temporal y espacial. Por último, la validación de resultados a través de aserciones en casos de prueba asegura que las conclusiones obtenidas sean correctas en todos los contextos, independientemente de la base de conocimiento inicial.