# Notas de Clase para IL

# 3. Deducción en Lógica Proposicional

Rafael Farré, Robert Nieuwenhuis,
Pilar Nivela, Albert Oliveras, Enric Rodríguez

3 de septiembre de 2009

# 1. Formas normales y cláusulas

- **Fórmulas como conjuntos:** Sea $F$ una fórmula construida sólo mediante la conectiva $\wedge$ a partir de subfórmulas $F_1 \ldots F_n$. Por ejemplo, $F$ puede ser la fórmula $(F_1 \wedge ((F_2 \wedge (F_3 \wedge F_4)) \wedge F_5))$. Por las propiedades de asociatividad, conmutatividad e idempotencia del $\wedge$ podemos escribir $F$ equivalentemente como $F_1 \wedge \ldots \wedge F_n$, o incluso como un *conjunto* $\{F_1, \ldots, F_n\}$, porque no importan los paréntesis (asociatividad), ni el orden de los elementos (conmutatividad), ni los elementos repetidos (idempotencia). Lo mismo pasa con la conectiva $\vee$.
- **Literales:** Un *literal* es un símbolo de predicado $p$ (*literal positivo*) o un símbolo de predicado negado $\neg p$ (*literal negativo*).
- **CNF:** Una fórmula está en *forma normal conjuntiva* (CNF, del inglés) si es una conjunción de disyunciones de literales, es decir, si es de la forma

  $(l_{1,1} \vee \ldots \vee l_{1,k_1}) \ \wedge \ \ldots \ \wedge \ (l_{n,1} \vee \ldots \vee l_{n,k_n})$,

  donde cada $l_{i,j}$ es un *literal*.
- **DNF:** Una fórmula está en *forma normal disyuntiva* (DNF, del inglés) si es una disyunción de conjunciones de literales, es decir, si es de la forma

  $(l_{1,1} \wedge \ldots \wedge l_{1,k_1}) \ \vee \ \ldots \ \vee \ (l_{n,1} \wedge \ldots \wedge l_{n,k_n})$,

  donde cada $l_{i,j}$ es un *literal*.
- **Cláusulas:** Una *cláusula* es una disyunción de literales, es decir, una fórmula de la forma $l_1 \vee \ldots \vee l_n$, donde cada $l_i$ es un literal, o, equivalentemente, una fórmula $p_1 \vee \ldots \vee p_m \vee \neg q_1 \vee \ldots \vee \neg q_n$, donde las $p_i$ y $q_j$ son símbolos de predicado.
- **Conjunto de cláusulas:** Una fórmula en CNF es pues una conjunción de cláusulas que puede verse como un *conjunto* de cláusulas.
- **Cláusula vacía:** La *cláusula vacía* es una cláusula $l_1 \vee \ldots \vee l_n$ donde $n = 0$, es decir, es la disyunción de cero literales. La cláusula vacía se suele denotar por $\square$. Nótese que, según la sintaxis de la lógica proposicional, la cláusula vacía no es una fórmula. Por eso, en esta hoja asumiremos las siguientes extensiones de la sintaxis y de la semántica.

  En cuanto a la sintaxis, definimos que, si $n \geq 0$, y $F_1, \ldots, F_n$ son fórmulas, entonces también son fórmulas $\bigwedge_{i \in 1..n} F_i$ y $\bigvee_{i \in 1..n} F_i$.

  En cuanto a la extensión correspondiente de la semántica, si $I$ es una interpretación:

  $$eval_I(\ \textstyle\bigwedge_{i \in 1..n} F_i\ ) \ = \ min\{\ 1,\ eval_I(F_1),\ \ldots,\ eval_I(F_n)\ \} \text{ y}$$
  $$eval_I(\ \textstyle\bigvee_{i \in 1..n} F_i\ ) \ = \ max\{\ 0,\ eval_I(F_1),\ \ldots,\ eval_I(F_n)\ \}.$$

  Intuitivamente, esta definición refleja que la conjuncíon es cierta en $I$ si lo son las $n$ fórmulas: si $n = 0$, la conjunción es trivialmente cierta. Similarmente, la disyunción es cierta si lo es al menos una: si $n = 0$, la disyunción es trivialmente falsa.
- **Cláusula de Horn:** Una *cláusula de Horn* es una cláusula $p_1 \vee \ldots \vee p_m \vee \neg q_1 \vee \ldots \vee \neg q_n$ con $m \leq 1$, es decir, que tiene como máximo un literal positivo.

# 2. Ejercicios

1. (dificultad 2) Demuestra que, para toda fórmula $F$, hay al menos una fórmula lógicamente equivalente que está en DNF. Ídem para CNF. Ayuda: obtener las fórmulas en CNF y DNF a partir de la tabla de verdad para $F$.
2. (dificultad 3) Da una manera de calcular una fórmula $\hat{F}$ en CNF para una fórmula $F$ dada (con $\hat{F} \equiv F$) sin necesidad de construir previamente la tabla de verdad. Ayuda: aplica equivalencias lógicas como las leyes de De Morgan y la distributividad y el Lema de Sustitución.

3. (dificultad 3) Cada fórmula de lógica proposicional puede verse como un circuito electrónico que tiene una puerta lógica por cada conectiva $\wedge$, $\vee$, $\neg$ que aparezca en la fórmula (aunque las fórmulas tienen estructura de árbol, mientras los circuitos en realidad permiten compartir subárboles repetidos, es decir, son grafos dirigidos acíclicos).

   El problema del *diseño lógico* consiste en encontrar un circuito adecuado que implemente una función booleana dada. Para conseguir circuitos *rápidos*, nos va bien representar la función booleana como una fórmula en CNF (o DNF), porque la *profundidad* del circuito será como máximo tres. Pero también es importante utilizar el mínimo número de conectivas (puertas lógicas). Los métodos de obtener CNFs vistos en los ejercicios anteriores, ¿nos dan la CNF más corta en ese sentido? ¿Se te ocurre alguna mejora?

4. (dificultad 1) La cláusula vacía $\square$ es el caso más sencillo de fórmula insatisfactible. Una CNF que es un conjunto de cero cláusulas, ¿es satisfactible o insatisfactible?

5. (dificultad 2) Demuestra que una cláusula es una tautología si, y sólo si, contiene a la vez $p$ y $\neg p$ para un cierto símbolo proposicional $p$.

6. (dificultad 2) Sea $S$ un conjunto de cláusulas con $\square \notin S$. Demuestra que $S$ es satisfactible (dando un modelo para $S$) en cada una de las siguientes situaciones:

   *a*) Toda cláusula de $S$ tiene algún literal positivo.

   *b*) Toda cláusula de $S$ tiene algún literal negativo.

   *c*) Para todo símbolo de predicado $p$ se cumple que: o bien $p$ aparece sólo en literales positivos en $S$, o bien $p$ aparece sólo en literales negativos en $S$.

7. (dificultad 2) Dados $n$ símbolos proposicionales:

   *a*) ¿Cuántas cláusulas distintas (como conjuntos de literales) hay?

   *b*) ¿Cuántas de estas cláusulas son insatisfactibles?

   *c*) ¿Cuántas cláusulas distintas y que no son tautologías hay?

   *d*) ¿Cuántas cláusulas distintas que contienen exactamente un literal por cada símbolo proposicional hay?

8. (dificultad 4) Propón un algoritmo eficiente que, dado un conjunto de cláusulas $S$, retorna un conjunto de cláusulas $S'$ (no necesariamente definido sobre los mismos símbolos de predicado que $S$) con como mucho 3 literales por cláusula, que es *equisatisfactible* a $S$ (es decir, que es satisfactible si y sólo si $S$ lo es). Ayuda: es posible introducir algún símbolo de predicado $p$ nuevo, que signifique: "$l \vee l'$ es cierto" para algún par de literales $l$ y $l'$.

9. (dificultad 2) Sea $S$ un conjunto de cláusulas de Horn con $\square \notin S$. Demuestra que $S$ es satisfactible (dando un modelo para $S$) si no hay ninguna cláusula que sólo conste de un único literal positivo.

10. (dificultad 2) Demuestra que el enunciado del ejercicio previo es falso cuando $S$ no es de Horn.

11. (dificultad 3) Definimos: un literal en una fórmula en CNF es *puro* si aparece o bien siempre negado o bien siempre sin negar. Además, se dice que una cláusula $C$ es *redundante* si contiene al menos un literal puro. Demuestra que, si $C'$ es una cláusula redundante, entonces $\{C_1, \ldots, C_n, C'\}$ es satisfactible si y sólo si $\{C_1, \ldots, C_n\}$ es satisfactible.

# 3. Nociones informales de decidibilidad y complejidad

Para una interpretación $I$ y una fórmula $F$ dadas, podemos determinar *eficientemente* mediante un programa de ordenador si $I$ satisface $F$ o no. En cambio, son muy *costosos* otros problemas, como el de decidir si una fórmula $F$ dada es satisfactible. Para comprender mejor qué queremos decir con palabras

como "eficiente" o "costoso", aquí a continuación damos *a nivel intuitivo e informal* algunas nociones sobre el *coste* de resolver ciertos problemas. Todos estos conceptos serán formalizados en detalle en otras asignaturas posteriores.

Consideraremos *algoritmos*, o programas, expresados en un lenguaje de programación como `C`, `Java`, o `C++`, que, para resolver un *problema computacional*, toman una *entrada* y calculan una *salida* mediante una secuencia de *pasos* de cómputo. Cada paso se supone que es muy sencillo, por ejemplo, una instrucción de lenguaje máquina o un número pequeño acotado de ellas (como veremos, esta distinción no es muy relevante).

En los problemas llamados *decisionales*, la salida es una respuesta de tipo sí/no. Por ejemplo, es decisional el problema cuya entrada es una lista de números, y que consiste en determinar si su suma es par o no. En cambio, no es decisional el problema de escribir la suma de los números, ni el de escribir la lista de números ordenados de menor a mayor.

Como sabemos, la lógica proposicional es especialmente interesante porque todos los problemas decisionales típicos (por ejemplo, si una fórmula es satisfactible, o si es tautología, etc.) son *decidibles*: para cada uno de ellos hay algún programa de ordenador que siempre termina y nos da una respuesta correcta sí/no.

Por ejemplo, consideremos el problema decisional cuya entrada son dos fórmulas $F$ y $G$, y que consiste en determinar si son lógicamente equivalentes o no. Un algoritmo puede *decidir* este problema simplemente construyendo su *tabla de verdad*, la lista de todas las posibles interpretaciones $I$ para el conjunto $\mathcal{P}$ de símbolos de predicado que aparecen en $F$ y $G$, y averiguar si para todas ellas tenemos que $eval_I(F) = eval_I(G)$.

Esto es posible en lógica proposicional porque se cumplen dos propiedades básicas: por un lado, para un conjunto de símbolos $\mathcal{P}$ finito, el número de interpretaciones es finito (aunque puede ser muy grande!) y, por otro, dada una interpretación $I$ y una fórmula $F$, es también decidible si $I$ satisface $F$. Como veremos, estas dos propiedades no se suelen tener en lógicas con mayor *poder expresivo*, como, por ejemplo, la lógica de primer orden, que permiten describir/modelar más cosas de la vida real.

## 3.1. Lo importante es el coste como función del tamaño de la entrada

Volvamos ahora a las cuestiones de eficiencia. Aquí consideraremos sólo el coste computacional en cuanto a *tiempo*, es decir, el número de pasos de cómputo, sin entrar en otros recursos como la cantidad de memoria de ordenador necesaria.

Evidentemente, para todo problema, resulta más costoso (son necesarios más pasos) resolverlo cuando la entrada es grande que cuando es pequeña. Por ejemplo, cuesta más sumar un millón de números que sumar cien.

A modo de ejemplo, supongamos que tenemos en memoria una tabla `int A[N]` con $N$ números enteros distintos, ordenados de menor a mayor, y necesitamos un algoritmo que encuentre en qué posición de la tabla se encuentra el número cero. Veamos dos posibles soluciones.

**Algoritmo 1: Búsqueda lineal:** podemos recorrer la tabla con un bucle como:

```
while (A[i]!=0) i++;
```

En el caso peor, que se da cuando el cero es el último elemento, el número de pasos será proporcional al número de veces que se ejecuta el bucle, que es proporcional a $N$, por lo que se dice que *el algoritmo 1 tiene coste lineal en $N$*, o que funciona en *tiempo lineal*, o simplemente, que *es lineal*.

**Algoritmo 2: Búsqueda dicotómica:** puesto que `A` está ordenado, también podemos primero inspeccionar sólo el elemento central `A[N/2]`; así ya sabemos en qué mitad de la tabla está el cero (a la izquierda o a la derecha de `A[N/2]`); luego inspeccionamos el centro de esa mitad, y así sucesivamente, reduciendo en cada iteración a la mitad el trozo de tabla donde tenemos que buscar. No es difícil de ver que el número máximo de iteraciones será proporcional a $\log N$, el logaritmo en base 2 de $N$. Por ejemplo, si $N \leq 2^5 = 32$, en 5 iteraciones encontraremos dónde está el cero. Por eso se dice que el algoritmo `2` tiene *coste logarítmico* en $N$ (o que funciona en tiempo logarítmico, o que es logarítmico).

Si la $N$ es pequeña, no está claro cuál de estos dos algoritmos es mejor. Quizás en ese caso la opción `1` es más rápida porque probablemente necesita menos instrucciones por cada iteración. Pero, aunque el

algoritmo 2 necesite muchas más instrucciones por iteración que el algoritmo 1, *a partir de cierto tamaño de la entrada N suficientemente grande, siempre será mejor un algoritmo logarítmico que uno lineal*. Por ejemplo, si $N \geq 2^{10} = 1024$, el algoritmo 2 será más rápido incluso si cada iteración suya necesita 100 veces más pasos que cada iteración del algoritmo 1!

Por eso, *lo que verdaderamente importa es cuán rápido crece el coste del algoritmo en función del tamaño de la entrada*, más que el número exacto de pasos de cómputo que necesita, o el número exacto de instrucciones de lenguaje máquina o ciclos de reloj de procesador que necesite cada paso.

Por orden creciente de coste, algunas funciones típicas son: $\log N$ (coste *logarítmico*); $N$ (*lineal*); $N^2$ (*cuadrático*: el coste es proporcional a cierto polinomio de grado 2, por ejemplo, $7N^2 + 3N$); $N^3$ (*cúbico*, un polinomio de grado 3); o incluso $2^N$ (*exponencial*). Si el coste es proporcional a un polinomio en $N$, o inferior, (logarítmico, lineal, cuadrático, cúbico, etc.) se dice que es *polinómico*.

Es fácil de ver que, conforme crece la $N$, el coste de los algoritmos exponenciales crece muchísimo más deprisa que los polinómicos. Por ejemplo, $1000^2$ es *sólo* un millón, ¡mientras que $2^{1000}$ es muy superior al número de átomos que hay en el universo! Por eso, para un algoritmo exponencial siempre habrá entradas relativamente pequeñas que resulten inabordables; tener un superordenador con un millón de procesadores sólo podrá producir mejoras en un factor constante de un millón, lo cual es irrelevante si un simple incremento en 1 del tamaño de la entrada duplica el número de pasos necesarios.

# 4. Ejercicios

12. (dificultad 3) Para una fórmula en DNF, ¿cuál es el mejor algoritmo posible para decidir si es satisfactible? ¿Qué coste tiene?

# 5. Resolución. Corrección y completitud

- Aquí denotaremos las cláusulas por mayúsculas $C$ o $D$ y escribiremos $l \vee C$ para denotar una cláusula que tiene un literal $l$ y en la que $C$ es la disyunción (la cláusula) de los demás literales.

- **Resolución:** Una *regla deductiva* nos dice cómo obtener (o *deducir*) ciertas fórmulas nuevas a partir de fórmulas dadas. Una regla deductiva concreta es la *resolución*: dadas dos cláusulas de la forma $p \vee C$ y $\neg p \vee D$ (las *premisas*), por resolución se deduce la nueva cláusula $C \vee D$ (la *conclusión*). Esto se suele escribir como:

$$\frac{p \vee C \qquad \neg p \vee D}{C \vee D} \qquad \textit{Resolución} \text{ (para lógica proposicional)}$$

- **Clausura bajo resolución**: Sea $S$ un conjunto de cláusulas. La *clausura de $S$ bajo resolución*, denotada por $Res(S)$, es el conjunto de todas las cláusulas que se pueden obtener con cero o más pasos de resolución a partir de cláusulas de $S$.

  Formalmente se define de la siguiente manera. Sea $Res_1(S)$ el conjunto de las cláusulas que se pueden obtener en exactamente un paso de resolución a partir de $S$:

$$Res_1(S) = \{C \vee D \mid p \vee C \in S,\ \neg p \vee D \in S\}$$

  es decir, el conjunto de todas las cláusulas $C \vee D$ tales que, para algún símbolo de predicado $p$, hay cláusulas en $S$ de la forma $p \vee C$ y $\neg p \vee D$.

  Ahora definimos para toda $i \geq 0$ (por inducción):

$$\begin{array}{ll} S_0 & = S \\ S_{i+1} & = S_i \cup Res_1(S_i) \end{array} \qquad \text{y definimos:} \qquad Res(S) = \bigcup_{i=0}^{\infty} S_i$$

  Nótese que esta definición nos da una manera *efectiva* (práctica) de construir $Res(S)$.

- **Clausura bajo una regla deductiva cualquiera**: Sea $R$ una regla deductiva (como, por ejemplo, la resolución) y sea $S$ un conjunto de fórmulas. Denotamos por $R(S)$ la clausura de $S$ bajo $R$: el conjunto de todas las fórmulas que se pueden obtener con cero o más pasos de deducción de $R$ a partir de fórmulas de $S$.

- **Corrección y completitud de una regla deductiva:** Si $S$ es un conjunto de fórmulas $\{F_1 \dots F_n\}$, a menudo consideraremos $S$ como la conjunción $F_1 \wedge \dots \wedge F_n$; por ejemplo, escribiremos $S \models F$ en vez de $F_1 \wedge \dots \wedge F_n \models F$.

  Definimos: la regla deductiva $R$ es *correcta* si mediante $R$ sólo podemos deducir fórmulas nuevas que son consecuencias lógicas de las que ya tenemos: si para toda fórmula $F$ y todo conjunto de fórmulas $S$, se cumple que $F \in R(S)$ implica $S \models F$.

  Definimos: la regla deductiva $R$ es *completa* si mediante $R$ podemos deducir todas las consecuencias lógicas: si para toda fórmula $F$ y todo conjunto de fórmulas $S$, se cumple que $S \models F$ implica $F \in R(S)$.

  Nótese que es fácil definir una regla deductiva correcta: basta con decir que *ninguna* fórmula se deduce. Igualmente, es fácil definir una regla deductiva completa: basta con decir que *toda* fórmula se deduce. Lo difícil es definir una regla deductiva $R$ que es tanto correcta como completa; en ese caso tenemos $S \models F$ si y sólo si $F \in R(S)$, es decir, que $R$ nos permite deducir todas las consecuencias lógicas, y nada más.

- **Completitud refutacional de la resolución**: La resolución es *refutacionalmente completa*, es decir, si $S$ es insatisfactible, entonces $\square \in Res(S)$.

# 6. Ejercicios

13. (dificultad 2) Utiliza resolución para demostrar que $p \to q$ es una consecuencia lógica de

$$\begin{array}{rcl} t & \to & q \\ \neg r & \to & \neg s \\ p & \to & u \\ \neg t & \to & \neg r \\ u & \to & s \end{array}$$

14. (dificultad 2) Demuestra por resolución que son tautologías:

    *a*) $p \to (q \to p)$

    *b*) $(p \wedge (p \to q)) \to q$

    *c*) $((p \to q) \wedge \neg q) \to \neg p$

    *d*) $((p \to q) \wedge \neg q) \to \neg q$

15. (dificultad 2) Demuestra que la resolución es correcta.

16. (dificultad 2) Demuestra que, para todo conjunto finito de cláusulas $S$, $Res(S)$ es un conjunto finito de cláusulas, si se consideran las cláusulas como conjuntos de literales (por ejemplo, $C \vee p$ es la misma cláusula que $C \vee p \vee p$).

17. (dificultad 3) Sea $S$ un conjunto de cláusulas. Demuestra que $Res(S)$ es lógicamente equivalente a $S$.

18. (dificultad 2) ¿La resolución es completa? Demuéstralo.

19. (dificultad 2) Sea $S$ un conjunto de cláusulas insatisfactible. Por la completitud refutacional de la resolución, sabemos que existe una demostración por resolución de que $\square \in Res(S)$. ¿Es esta demostración única?

20. (dificultad 4) Demuestra la completitud refutacional de la resolución, esto es, si $S$ es un conjunto de cláusulas insatisfactible entonces $\square \in Res(S)$.

    Ayuda: demuestra el contrarrecíproco por inducción sobre el número $N$ de símbolos de predicado de $S$.

21. (dificultad 2) Demuestra que el lenguaje de las cláusulas de Horn es cerrado bajo resolución, es decir, a partir de cláusulas de Horn por resolución sólo se obtienen cláusulas de Horn.

22. (dificultad 2) Considera el siguiente caso particular de la resolución:

$$\frac{p \qquad \neg p \vee C}{C} \qquad \textit{Resolución Unitaria}$$

    Demuestra que la resolución unitaria es correcta.

23. (dificultad 2) Demuestra que la resolución unitaria no es refutacionalmente completa para cláusulas que no son de Horn.

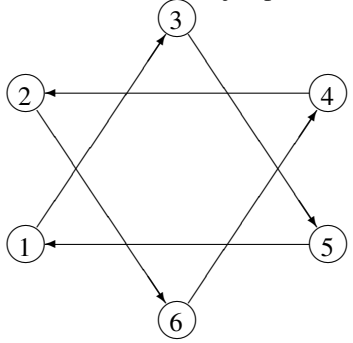
24. (dificultad 3) Demuestra que la resolución unitaria es refutacionalmente completa para cláusulas de Horn. Ayuda: Basta con ver que, si $S$ es un conjunto de cláusulas de Horn y $\square \notin ResUnit(S)$, entonces $ResUnit(S)$ (y por lo tanto $S$) tiene un modelo $I$. Define $I$ como $I(p) = 1$ si y sólo si $p$ es una cláusula (de un solo literal) en $ResUnit(S)$ y demuestra $I \models ResUnit(S)$ por inducción sobre el número de literales de las cláusulas.

25. (dificultad 2) ¿Cuál es la complejidad del problema de determinar si un conjunto de cláusulas de Horn $S$ es satisfactible? Ayuda: analiza (informalmente) la corrección y la complejidad del siguiente algoritmo (que intenta construir sistemáticamente el modelo *minimal* $I$ de $S$):

    (0) inicialmente, $I(p) = 0$ para todo $p$
    (1) hacer ciertos en $I$ los $p$ que son cláusulas unitarias positivas;
    (2) eliminar de todas las cláusulas los literales $\neg p$ con $I(p) = 1$;
        si esto da lugar a la cláusula vacía: insatisfactible
        si no, si esto da lugar a alguna cláusula unitaria nueva, volver a (1)
            si no, la interpretación $I$ construida es un modelo.

26. (dificultad 2) Las *cláusulas de Krom* son aquellas que tienen a lo sumo dos literales. ¿Cuántas cláusulas de Krom se pueden construir con $n$ símbolos de predicado? Demuestra que basta un número cuadrático de pasos de resolución para decidir si un conjunto de cláusulas de Krom es satisfactible o no.

27. (dificultad 3) Un *grafo* $G = (V, E)$ es un conjunto $V$ de objetos llamados *vértices* conectados por enlaces llamados *aristas*; la arista que une los vértices $u$ y $v$ se representa como el par de vértices $(u, v)$, y el conjunto de aristas se denota por $E$. Además, se dice que un grafo está *dirigido* si se distingue entre las dos posibles orientaciones de las aristas. Por ejemplo, en el grafo dirigido siguiente:



el conjunto de vértices $V$ es $\{1, 2, 3, 4, 5, 6\}$, y el de aristas $E$ es $\{(1, 3), (3, 5), (5, 1), (2, 6), (6, 4), (4, 2)\}$. Finalmente, se dice que una secuencia de vértices $v_0$, ..., $v_k$ es un *camino* si se tiene que los vértices $v_0$, ..., $v_k$ están sucesivamente conectados, es decir, si $(v_{i-1}, v_i) \in E$ para todo $1 \leq i \leq k$. Por ejemplo, 1, 3 y 5 forman un camino del grafo dibujado más arriba, ya que $(1, 3) \in E$ y $(3, 5) \in E$. Los grafos son objetos muy importantes en matemáticas e informática y se estudian en detalle en varias asignaturas posteriores.

Dado $S$ un conjunto de cláusulas con 1 ó 2 literales por cláusula, definido sobre los símbolos proposicionales $p_1$, ..., $p_n$, se define el *grafo asociado a* $S$, denotado $G_S$, como el grafo dirigido $G_S = (V, E)$, donde $V = \{p_1, \ldots, p_n, \neg p_1, \ldots, \neg p_n\}$ y $E = \{(l, l') \mid \neg l \vee l' \in S\}$ (en este ejercicio, abusando de la notación, dado un literal $l$ de la forma $\neg p$, vamos a considerar que $\neg l$ representa $p$; además, en la construcción del grafo las cláusulas de 1 literal, o sea de la forma $p$, se consideran como $p \vee p$).

*a*) Demuestra que si hay un camino de $l$ a $l'$ en $G_S$, entonces $\neg l \vee l' \in Res(S)$. Recíprocamente, demuestra que si $l \vee l' \in Res(S)$, entonces hay un camino de $\neg l$ a $l'$ en $G_S$.

*b*) Demuestra que $S$ es insatisfactible si y sólo si existe un símbolo proposicional $p$ tal que hay un camino en $G_S$ de $p$ a $\neg p$, y otro camino de $\neg p$ a $p$.

*c*) Basándote en el apartado previo, propón un algoritmo para determinar la satisfactibilidad de un conjunto de cláusulas con 1 ó 2 literales por cláusula. ¿Qué complejidad tiene, en términos del número de cláusulas y de símbolos proposicionales de $S$?

# 7. Resolver problemas prácticos con la lógica proposicional

En la Sección 3 vimos informalmente qué significa que un algoritmo tenga coste polinómico o exponencial. Hay una importante clase de problemas para los que no se han descubierto algoritmos polinómicos. Para los problemas de esta clase (los llamados problemas *NP-completos*, ver abajo) sólo se conocen algoritmos que, en el caso peor, necesitan un número exponencial de pasos. Esta clase incluye miles de problemas prácticos importantes que surgen, por ejemplo, al trazar rutas de transporte o redes de comunicación, asignar máquinas u otros recursos en procesos industriales, confeccionar horarios de hospitales, escuelas, líneas aéreas, cargar un camión o un barco, etc.

Uno de los problemas NP-completos más famosos es *SAT*: el problema de decidir si una fórmula de lógica proposicional dada es satisfactible o no. Los algoritmos para SAT, los llamados *SAT solvers*, están muy estudiados y a menudo son capaces de tratar fórmulas grandes. Por eso es muy útil saber que los SAT solvers pueden usarse también para intentar resolver casos concretos de los demás problemas NP-completos.

Por ejemplo, podemos expresar fácilmente el problema de resolver un Sudoku (otro problema NP-completo) como un problema de SAT. Vamos a hacerlo aquí con $9^3 = 729$ símbolos de predicado $p_{ijk}$, que significan: "en la fila $i$ columna $j$ del Sudoku hay el valor $k$", con $1 \leq i, j, k \leq 9$. Aquí tenemos un Sudoku:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 7 | | 6 | | | 3 | | 9 |
| | 2 | | 3 | | 9 | | 7 | 1 |
| 1 | | | | 8 | | | | |
| | 5 | | 7 | | 3 | | 8 | 6 |
| | | 6 | | | | 4 | | |
| 4 | 1 | | 8 | | 6 | | 5 | |
| | | | | 6 | | | | 2 |
| 8 | 9 | | 5 | | 2 | | 6 | |
| 2 | | 3 | | | 4 | | 1 | 8 |

Tenemos que expresar:

1. **En cada casilla [i,j] hay al menos un valor.** Para expresar esto, en nuestro problema de SAT incluimos cláusulas de la forma $p_{ij1} \vee p_{ij2} \vee \ldots \vee p_{ij9}$.
   Por ejemplo, la cláusula $p_{111} \vee p_{112} \vee \ldots \vee p_{119}$ expresa que: "en la casilla [1,1] hay un 1, o en la

casilla [1,1] hay un 2, o hay un 3, . . ., o hay un 9".
Para definir más formalmente qué cláusulas incluímos, escribimos:

Para todos los $i, j$ con $1 \leq i, j \leq 9$ tenemos la cláusula $p_{ij1} \vee p_{ij2} \vee \ldots \vee p_{ij9}$

total: 81 claúsulas de 9 literales cada una.

2. **En cada casilla no hay más de un valor.** Para expresar esto, en nuestro problema de SAT incluimos muchas cláusulas de dos literales. Por ejemplo, la cláusula $\neg p_{111} \vee \neg p_{112}$ expresa que "en la casilla $[1, 1]$ no hay un 1 o en la casilla $[1, 1]$ no hay un 2" (es decir, si hay un 1 no hay un 2, y si hay un 2 no hay un 1). Tenemos que expresar esto para todas las casillas $[i, j]$, y todos los pares de valores distintos $k$ y $k'$. Formalmente:

   Para todos los $i, j$ con $1 \leq i, j \leq 9$, y

   para todos los $k, k'$ con $1 \leq k < k' \leq 9$ tenemos la cláusula $\neg p_{ijk} \vee \neg p_{ijk'}$.

   total: $81 \cdot 36 = 2916$ cláusulas de dos literales.

   (por cada una de las 81 casillas, hay 36 pares $k, k'$ posibles ya que el conjunto $\{1, \ldots, 9\}$ tiene $\binom{9}{2} = 36$ subconjuntos de 2 elementos).

3. **En cada fila (o columna, o cuadrado de 3x3) ningún valor se repite.** Para esto nuevamente incluimos cláusulas de dos literales. Por ejemplo, para las dos primeras casillas de la fila 1, la cláusula $\neg p_{111} \vee \neg p_{121}$ expresa que "en la casilla $[1, 1]$ no hay un 1 o en la casilla $[1, 2]$ no hay un 1". Formalmente, para las filas:

   Para todos los $i, k$ con $1 \leq i, k \leq 9$, y

   para todos los $j, j'$ con $1 \leq j < j' \leq 9$ tenemos la cláusula $\neg p_{ijk} \vee \neg p_{ij'k}$.

   total: $81 \cdot 36 = 2916$ cláusulas de dos literales para las filas,

   y dos veces 2916 más para las columnas y cuadrados de 3x3.

4. **Cada número ya puesto en el sudoku.** Tendremos cláusulas de un solo literal, como $p_{115}$ (el 5 en la casilla $[1, 1]$, la superior izquierda). Estas son las únicas que cambian en cada Sudoku; las cláusulas de los puntos 1,2 y 3 son siempre las mismas.

En `www.lsi.upc.edu/~roberto/il.html` está disponible un sencillo programa (en el lenguaje de programación lógica Prolog, que veremos más adelante en esta asignatura) que expresa (o *traduce*) así Sudokus como problemas de SAT. También hay un SAT solver llamado *Siege*, que resuelve en 0.01 segundos este problema concreto así generado, de 11781 cláusulas, y otro programa Prolog que toma como entrada el modelo que hemos encontrado para el problema de SAT y *lo traduce a una solución del problema de Sudoku que teníamos*.

En la lista de ejercicios veremos cómo resolver mediante SAT otros problemas NP-completos, siempre traduciéndolos directamente a una CNF, es decir, a un conjunto de cláusulas. Si para esta CNF se encuentra una solución (un modelo), podremos hacer la traducción al revés para reconvertirlo en una solución para nuestro problema original!

**Un poco de cultura informal sobre los problemas NP-completos.** Se dice que un problema *está en NP* si hay algún algoritmo *No-determinista Polinómico* que lo resuelve. Informalmente, esto significa que en tiempo polinómico podemos "adivinar" una posible solución y comprobar si efectivamente lo es. Por ejemplo, en SAT las posibles soluciones son las interpretaciones $I$; podemos generar una $I$ mediante $|\mathcal{P}|$ "adivinanzas" binarias 1/0, y verificar si esta $I$ concreta es solución para la fórmula dada $F$ (verificar si $I \models F$) es también polinómico. Un algoritmo para SAT que simplemente pruebe todas las interpretaciones posibles será exponencial, porque hay $2^{|\mathcal{P}|}$ de ellas!

Un problema que está en NP se dice que es *NP-completo* si además es posible utilizarlo para *expresar* en tiempo polinómico cualquier otro problema de NP (por ejemplo, hemos usado SAT para expresar el problema de los Sudokus). Como ya hemos dicho, hay miles de problemas prácticos importantes que son

NP-completos. Hoy día no se sabe si es posible resolver los problemas NP-completos en tiempo polinómico, pero se piensa que no[1]. Si tuviéramos un algoritmo polinómico para sólo uno de los miles de problemas NP-completos, ya lo tendríamos para todos, porque podríamos expresarlos todos en términos de ése!

La instancia (o entrada) concreta de SAT obtenida a partir del Sudoku de nuestro ejemplo es relativamente fácil de resolver. Pero, dada la NP-completitud del problema de SAT, no es ninguna sorpresa que existan instancias de SAT no muy grandes que ni los mejores SAT solvers son capaces de tratar en, digamos, una semana. Sin embargo, los SAT solvers actuales a menudo pueden resolver instancias de SAT relativamente grandes. En la siguiente sección veremos cómo funcionan estos algoritmos.

# 8. Ejercicios

28. (dificultad 3) Dado un mapa de un continente, es posible colorearlo con cuatro colores sin que dos países con frontera común tengan el mismo color (es el famoso *four color problem*). Para grafos, el problema se generaliza al problema NP-completo de *K-coloreado:* dado un grafo $G$ y un número natural $K$, decidir si podemos asignar a cada vértice un natural entre 1 y $K$ (un *color*), tal que todo par de vértices adyacentes tengan colores distintos.

    Expresa este problema mediante una CNF, de modo que se pueda resolver con un SAT solver. ¿Cuántos símbolos de predicado se necesitan? ¿Cuántas cláusulas se obtienen?

29. (dificultad 3) Dados un edificio de una sola planta con muchos pasillos rectos que se cruzan, y un número natural $K$, ¿se pueden colocar cámaras giratorias en los cruces de los pasillos de modo que sea posible vigilar todos los pasillos con como mucho $K$ cámaras? Este problema se puede formalizar como el problema de grafos llamado *vertex cover* o, traducido, *recubrimiento de vértices*: ¿existe un subconjunto de tamaño como mucho $K$ de los $N$ vértices, el *recubrimiento*, tal que toda arista tenga al menos un extremo en el recubrimiento (es decir, quede *cubierta*)?

    Expresa este problema mediante una CNF, de modo que se pueda resolver con un SAT solver. Usa los $K \cdot N$ símbolos de predicado $p_{i,j}$ que signifiquen: "el $i$-ésimo miembro del recubrimiento (con $i$ entre 1 y $K$) es el vértice $j$ (con $j$ entre 1 y $N$)". ¿Cuántas cláusulas se obtienen?

30. (dificultad 5) Siguiendo el problema anterior, si hubiésemos usado la codificación con símbolos de predicado $p_i$ que significasen: "hay una cámara en el vértice $i$", ¿cómo expresarías de forma compacta que no hay más de $K$ cámaras? (Ayuda: piensa en circuitos sumadores y usa símbolos adicionales). Usando estos símbolos de predicado expresa el problema de *vertex cover* mediante una CNF, de modo que se pueda resolver con un SAT solver.

31. (dificultad 3) Dado un grupo de estudiantes con las listas de asignaturas que estudia cada uno, y un natural $K$, ¿hay algún subconjunto de exactamente $K$ estudiantes tal que toda asignatura tenga algún estudiante que la curse?

    Expresa este problema mediante una CNF, de modo que se pueda resolver con un SAT solver. ¿Cuántos símbolos de predicado se necesitan? ¿Cuántas cláusulas se obtienen?

32. (dificultad 3) ¿Cuál crees que es el coste mínimo de un algoritmo que calcule una DNF lógicamente equivalente para una fórmula $F$?

# 9. El procedimiento de Davis-Putnam-Logemann-Loveland (DPLL)

Casi todos los SAT solvers actuales (como Siege) utilizan variantes modernas del algoritmo de Davis-Putnam-Logemann-Loveland (DPLL). Este algoritmo sirve para CNFs, es decir, para conjuntos de cláusulas. Gracias a los avances en algoritmos DPLL, los SAT solvers están siendo usados cada vez más para resolver todo tipo de problemas NP-completos prácticos.

[1] Éste es el famoso problema de "P vs. NP" (donde P significa polinómico), uno de los siete problemas matemáticos abiertos más importantes según el Clay Mathematics Institute, que ofrece un premio de un millón de dólares a quien lo resuelva, tanto si demuestra que sí es posible como si no; ver `www.claymath.org/millennium`.

Aquí presentaremos una versión sencilla del DPLL, basada en reglas, en la que el conjunto de cláusulas $F$ dado no cambia a lo largo de la ejecución. El algoritmo explora de una manera compacta todas las posibles interpretaciones. En cada momento se tiene una interpretación parcial, representada como una secuencia de literales $M$, los que son ciertos en ese momento. $M$ nunca contiene a la vez un literal $l$ y su negado $\neg l$, ni tampoco contiene literales repetidos. Decimos que *una cláusula $C$ es falsa en $M$* si $\neg l \in M$ para todo literal $l$ de $C$. La secuencia $M$ se va extendiendo *decidiendo* (o adivinando) nuevos literales, y cada vez que una cláusula se vuelve falsa en $M$, se *invierte* la última decisión tomada (esto se llama *backtracking*). A veces un literal $l$ figura *marcado* como $l^{\mathsf{d}}$. Esta marca significa que se trata de un literal de *decisión* (adivinado), lo cual indica que aún debe ser probado también su negado. Inicialmente, $M$ es la secuencia vacía $\emptyset$, y el algoritmo simplemente va aplicando cualquier regla de las cuatro siguientes:

Propaga :

$M \quad \Longrightarrow \quad M\ l \qquad$ **SI** $\left\{ \begin{array}{l} \text{En } F \text{ hay alguna cláusula } l \vee C \text{ cuya parte } C \\ \text{es falsa en } M, \text{ y ni } l \text{ ni su negado están en } M. \end{array} \right.$

Decide :

$M \quad \Longrightarrow \quad M\ l^{\mathsf{d}} \qquad$ **SI** $\left\{ \begin{array}{l} \text{El literal } l \text{ o su negado aparece en } F, \text{ y ni } l \text{ ni} \\ \text{su negado están en } M. \end{array} \right.$

Falla :

$M \quad \Longrightarrow \quad$ "*Insat*" **SI** $\left\{ \begin{array}{l} \text{En } F \text{ hay alguna cláusula que es falsa en } M, \text{ y} \\ M \text{ no contiene literales de decisión.} \end{array} \right.$

Backtrack :

$M\ l^{\mathsf{d}}\ N \quad \Longrightarrow \quad M\ \neg l \qquad$ **SI** $\left\{ \begin{array}{l} \text{En } F \text{ hay alguna cláusula que es falsa en} \\ M\ l^{\mathsf{d}}\ N, \quad \text{y } N \text{ no contiene literales de decisión.} \end{array} \right.$

Nótese que, en la última regla, el literal $\neg l$ ya no está marcado como decisión, porque su negado ya ha sido probado. La regla Propaga aprovecha que a menudo no hace falta adivinar: para que la cláusula $l \vee C$ se haga cierta, estamos *forzados* a poner $l$ a cierto (se dice que *propagamos* la información de la que disponemos en $M$). Por motivos de eficiencia, es bueno aplicar Falla y Backtrack con mayor preferencia, y después Propaga. Si $F$ es un conjunto finito de cláusulas, tenemos los siguientes resultados:

1. Cualquier aplicación de las reglas *termina*, es decir, no existe ninguna secuencia infinita de aplicaciones: $\emptyset \Longrightarrow M_1 \Longrightarrow M_2 \Longrightarrow \ldots$
2. Si $\emptyset \Longrightarrow \ldots \Longrightarrow$ "*Insat*", entonces $F$ es insatisfactible.
3. Si $\emptyset \Longrightarrow \ldots \Longrightarrow M$ y a $M$ no se le puede aplicar ninguna regla, entonces $M$ es un modelo de $F$.

**Ejemplo:** Sea $F$ el siguiente conjunto de cláusulas (donde los símbolos de predicado se representan como naturales, y la negación con una rayita):

$$\begin{array}{lllllllllll} 1. & 1 & \vee & \overline{2} & \vee & 3 & \vee & \overline{4} & \vee & \overline{5} \\ 2. & 1 & \vee & & & 3 & \vee & 4 & \vee & 5 \\ 3. & 1 & \vee & & & \overline{3} & \vee & 4 & & \\ 4. & 1 & \vee & & & 3 & \vee & \overline{4} & \vee & 5 \\ 5. & 1 & \vee & & & \overline{3} & \vee & \overline{4} & & \\ 6. & \overline{1} & \vee & \overline{2} & & & & & & \\ 7. & & & 2 & & & & & & \\ 8. & & & \overline{2} & \vee & 3 & \vee & 4 & \vee & \overline{5} \end{array}$$

Anotando cada $\Longrightarrow$ con la primera letra de la regla aplicada (Propaga, Decide, Falla, o Backtrack), y el número de la cláusula usada, tenemos:

$$\emptyset \ \Longrightarrow_{p7}\ 2 \ \Longrightarrow_{p6}\ 2\,\overline{1} \ \Longrightarrow_{d}\ 2\,\overline{1}\,3^{\mathsf{d}} \ \Longrightarrow_{p5}\ 2\,\overline{1}\,3^{\mathsf{d}}\,\overline{4} \ \Longrightarrow_{b3}\ 2\,\overline{1}\,\overline{3} \ \Longrightarrow_{d}$$

$$2\,\overline{1}\,\overline{3}\,4^{\mathsf{d}} \;\Longrightarrow_{p1}\; 2\,\overline{1}\,\overline{3}\,4^{\mathsf{d}}\,\overline{5} \;\Longrightarrow_{b4}\; 2\,\overline{1}\,\overline{3}\,\overline{4} \;\Longrightarrow_{p2}\; 2\,\overline{1}\,\overline{3}\,\overline{4}\,5 \;\Longrightarrow_{f8}\; \text{“}\mathit{Insat}\text{”}$$

con lo cual hemos demostrado su insatisfactibilidad. Se invita al lector a comprobar el trabajo necesario para hacer lo mismo mediante resolución u otros métodos deductivos. Sin la cláusula 8., y con la misma secuencia de pasos, el algoritmo habría acabado después del penúltimo paso, encontrando el modelo $2\,\overline{1}\,\overline{3}\,\overline{4}\,5$.

# 10. Ejercicios

33. (dificultad 2) Di cuáles de las siguientes fórmulas son satisfactibles utilizando el procedimiento DPLL:

    *a*) $(p \vee \neg q \vee r \vee \neg s) \wedge (\neg r \vee s) \wedge q \wedge \neg p$

    *b*) $(p \vee q) \wedge (\neg p \vee \neg q) \wedge (p \vee r) \wedge (\neg p \vee \neg r)$

    *c*) $(p \vee q) \wedge (\neg p \vee \neg q) \wedge (p \vee r) \wedge (\neg p \vee \neg r) \wedge (q \vee r) \wedge (\neg q \vee \neg r)$

34. (dificultad 2) Utiliza el procedimiento DPLL para demostrar que $p \to q$ es una consecuencia lógica de

$$\begin{array}{rcl} t & \to & q \\ \neg r & \to & \neg s \\ p & \to & u \\ \neg t & \to & \neg r \\ u & \to & s \end{array}$$

35. (dificultad 2) Demuestra que son tautologías utilizando el procedimiento DPLL:

    *a*) $p \to (q \to p)$

    *b*) $(p \wedge (p \to q)) \to q$

    *c*) $((p \to q) \wedge \neg q) \to \neg p$

    *d*) $((p \to q) \wedge \neg q) \to \neg q$

36. (dificultad 3) (*Invariantes de DPLL*) Supongamos que se aplica el procedimiento DPLL a un conjunto de cláusulas $F$ y que se obtiene la traza $\emptyset \Longrightarrow \cdots \Longrightarrow M$, con $M \neq$ “*Insat*”. Demuestra que entonces se cumplen las propiedades siguientes:

    *a*) Todos los símbolos proposicionales en $M$ son símbolos proposicionales de $F$.

    *b*) $M$ no contiene ningún literal más de una vez y no contiene $p$ y $\neg p$ para ningún símbolo proposicional $p$.

    *c*) Si $M$ es de la forma $N_0\; l_1^{\mathsf{d}}\; N_1\; l_2^{\mathsf{d}}\; \cdots\; l_n^{\mathsf{d}}\; N_n$, donde $l_1$, ..., $l_n$ son los literales de decisión de $M$, entonces $F \cup \{l_1, \ldots, l_i\} \models N_i$ para cada $i = 0 \ldots n$, interpretando $N_i$ como la conjunción de todos sus literales.

37. (dificultad 3) (*Corrección y completitud del procedimiento DPLL*) Supongamos que se aplica el procedimiento DPLL a un conjunto de cláusulas $F$, y que se obtiene la traza $\emptyset \Longrightarrow \cdots \Longrightarrow M$, a partir de donde ya no se puede aplicar más ninguna de las reglas Propaga, Decide, Falla o Backtrack. Utilizando los invariantes de DPLL, demuestra que:

    *a*) Si $M =$ “*Insat*” entonces $F$ es insatisfactible.

    *b*) Si $M \neq$ “*Insat*” entonces $M \models F$ (y en particular $F$ es satisfactible).

38. (dificultad 4) (*Terminación del procedimiento DPLL*) Supongamos que se aplica el procedimiento DPLL a un conjunto de cláusulas. Demuestra que no existen secuencias infinitas de la forma $\emptyset \Longrightarrow \cdots$

39. (dificultad 3) El problema de *AllSAT* consiste en obtener todos los modelos de una fórmula proposicional $F$. Desarrolla un algoritmo para *AllSAT* utilizando llamadas independientes al procedimiento DPLL.