bolo impuro. Es fácil observar que si una sentencia tiene un modelo, entonces tiene un modelo con símbolos puros asignados para hacer que sus literales sean *verdaderos*, porque al hacerse así una cláusula nunca puede ser falsa. Fíjese en que, al determinar la pureza de un símbolo, el algoritmo puede ignorar las cláusulas que ya se sabe que son verdaderas en el modelo construido hasta ahora. Por ejemplo, si el modelo contiene $B = falso$, entonces la cláusula $(\neg B \vee \neg C)$ ya es verdadera, y C pasa a ser un símbolo puro, ya que sólo aparecerá en la cláusula $(C \vee A)$.

- *Heurística de cláusula unitaria*: anteriormente había sido definida una **cláusula unitaria** como aquella que tiene sólo un literal. En el contexto del DPLL, este concepto también determina a aquellas cláusulas en las que todos los literales, menos uno, tienen asignado el valor *falso* en el modelo. Por ejemplo, si el modelo contiene $B = falso$, entonces $(B \vee \neg C)$ pasa a ser una cláusula unitaria, porque es equivalente a $(Falso \vee \neg C)$, o justamente $\neg C$. Obviamente, para que esta cláusula sea verdadera, a C se le debe asignar *falso*. La heurística de cláusula unitaria asigna dichos símbolos antes de realizar la ramificación restante. Una consecuencia importante de la heurística es que cualquier intento de demostrar (mediante refutación) un literal que ya esté en la base de conocimiento, tendrá éxito inmediatamente (Ejercicio 7.16). Fíjese también en que la asignación a una cláusula unitaria puede crear otra cláusula unitaria (por ejemplo, cuando a C se le asigna *falso*, $(C \vee A)$ pasa a ser una cláusula unitaria, causando que le sea asignado a A el valor verdadero). A esta «cascada» de asignaciones forzadas se la denomina **propagación unitaria**. Este proceso se asemeja al encadenamiento hacia delante con las cláusulas de Horn, y de hecho, si una expresión en FNC sólo contiene cláusulas de Horn, entonces el DPLL esencialmente reproduce el encadenamiento hacia delante. (*Véase* el Ejercicio 7.17.)

PROPAGACIÓN UNITARIA

En la Figura 7.16 se muestra el algoritmo DPLL. Sólo hemos puesto la estructura básica del algoritmo, que describe, en sí mismo, el proceso de búsqueda. No hemos descrito las estructuras de datos que se deben utilizar para hacer que cada paso de la búsqueda sea eficiente, ni los trucos que se pueden añadir para mejorar su comportamiento: aprendizaje de cláusulas, heurísticas para la selección de variables y reinicialización aleatoria. Cuando estas mejoras se añaden, el DPLL es uno de los algoritmos de *satisfacibilidad* más rápidos, a pesar de su antigüedad. La implementación CHAFF se utiliza para resolver problemas de verificación de *hardware* con un millón de variables.

Algoritmos de búsqueda local

Hasta ahora, en este libro hemos visto varios algoritmos de búsqueda local, incluyendo la ASCENSIÓN-DE-COLINA (página 126) y el TEMPLADO-SIMULADO (página 130). Estos algoritmos se pueden aplicar directamente a los problemas de *satisfacibilidad*, a condición de que elijamos la correcta función de evaluación. Como el objetivo es encontrar una asignación que satisfaga todas las cláusulas, una función de evaluación que cuente el número de clásulas *insatisfacibles* hará bien el trabajo. De hecho, ésta es exactamente la medida utilizada en el algoritmo MIN-CONFLICTOS para los PSR (página 170). Todos estos algoritmos realizan los pasos en el espacio de asignaciones completas,

```
función ¿SATISFACIBLE-DPLL?(s) devuelve verdadero o falso
  entradas: s, una sentencia en lógica proposicional

  cláusulas ← el conjunto de cláusulas de s en representación FNC
  símbolos ← una lista de los símbolos proposicionales de s
  devolver DPLL(cláusulas, símbolos, [ ])

función DPLL(cláusulas, símbolos, modelo) devuelve verdadero o falso

  si cada cláusula en cláusulas es verdadera en el modelo entonces devolver verdadero
  si alguna cláusula en cláusulas es falsa en el modelo entonces devolver falso
  P, valor ← ENCONTRAR-SÍMBOLO-PURO(símbolos, cláusulas, modelo)
  si P no está vacío entonces devolver
  DPLL(cláusulas, símbolos − P, EXTENDER(P, valor, modelo)
  P, valor ← ENCONTRAR-CLÁUSULA-UNITARIA(cláusulas, modelo)
  si P no está vacío entonces devolver
  DPLL(cláusulas, símbolos − P, EXTENDER(P, valor, modelo)
  P ← PRIMERO(símbolos); resto ← RESTO(símbolos)
  devolver DPLL(cláusulas, resto, EXTENDER(P, verdadero, modelo)) o
           DPLL(cláusulas, resto, EXTENDER(P, falso, modelo))
```

Figura 7.16 El algoritmo DPLL para la comprobación de la *satisfacibilidad* de una sentencia en lógica proposicional. En el texto se describen ENCONTRAR-SÍMBOLO-PURO y ENCONTRAR-CLÁUSULA-UNITARIA; cada una devuelve un símbolo (o ninguno) y un valor de verdad para asignar a dicho símbolo. Al igual que ¿IMPLICACIÓN-TV?, este algoritmo trabaja sobre modelos parciales.

intercambiando el valor de verdad de un símbolo a la vez. El espacio generalmente contiene muchos mínimos locales, requiriendo diversos métodos de aleatoriedad para escapar de ellos. En los últimos años se han realizado una gran cantidad de experimentos para encontrar un buen equilibrio entre la voracidad y la aleatoriedad.

Uno de los algoritmos más sencillos y eficientes que han surgido de todo este trabajo es el denominado SAT-CAMINAR (Figura 7.17). En cada iteración, el algoritmo selecciona una cláusula insatisfecha y un símbolo de la cláusula para intercambiarlo. El algoritmo escoge aleatoriamente entre dos métodos para seleccionar el símbolo a intercambiar: (1) un paso de «min-conflictos» que minimiza el número de cláusulas insatisfechas en el nuevo estado, y (2) un paso «pasada-aleatoria» que selecciona de forma aleatoria el símbolo.

¿El SAT-CAMINAR realmente trabaja bien? De forma clara, si el algoritmo devuelve un modelo, entonces la sentencia de entrada de hecho es *satisfacible*. ¿Qué sucede si el algoritmo devuelve *fallo*? En ese caso, no podemos decir si la sentencia es *insatisfacible* o si necesitamos darle más tiempo al algoritmo. Podríamos intentar asignarle a *max_intercambios* el valor infinito. En ese caso, es fácil ver que con el tiempo SAT-CAMINAR nos devolverá un modelo (si existe alguno) a condición de que la probabilidad $p > 0$. Esto es porque siempre hay una secuencia de intercambios que nos lleva a una asignación satisfactoria, y al final, los sucesivos pasos de movimientos aleatorios generarán dicha secuencia. Ahora bien, si *max_intercambios* es infinito y la sentencia es *insatisfacible*, entonces la ejecución del algoritmo nunca finalizará.

```
función SAT-CAMINAR(cláusulas, p, max_intercambios) devuelve un modelo satisfactorio o fallo
  entradas: cláusulas, un conjunto de cláusulas en lógica proposicional
            p, la probabilidad de escoger un movimiento «aleatorio», generalmente alrededor de 0,5
            max_intercambios el número de intercambios permitidos antes de abandonar
  modelo ← una asignación aleatoria de verdadero/falso a los símbolos de las cláusulas
  para i = 1 hasta max_intercambios hacer
    si modelo satisface las cláusulas entonces devolver modelo
    cláusula ← una cláusula seleccionada aleatoriamente de cláusulas que es falsa en el modelo
    con probabilidad p intercambia el valor en el modelo de un símbolo seleccionado
      aleatoriamente de la cláusula
    sino intercambia el valor de cualquier símbolo de la cláusula para que maximice el
      número de cláusulas satisfacibles
  devolver fallo
```

Figura 7.17 El algoritmo SAT-CAMINAR para la comprobación de la *satisfacibilidad* mediante intercambio aleatorio de los valores de las variables. Existen muchas versiones de este algoritmo.

Lo que sugiere este problema es que los algoritmos de búsqueda local, como el SAT-CAMINAR, son más útiles cuando esperamos que haya una solución (por ejemplo, los problemas de los que hablamos en los Capítulos 3 y 5, por lo general, tienen solución). Por otro lado, los algoritmos de búsqueda local no detectan siempre la *insatisfacibilidad*, algo que se requiere para decidir si hay relación de implicación. Por ejemplo, un agente no puede utilizar la búsqueda local para demostrar, de forma fiable, si una casilla es segura en el mundo de *wumpus*. En lugar de ello, el agente puede decir «he pensado acerca de ello durante una hora y no he podido hallar ningún mundo posible en el que la casilla *no sea segura*». Si el algoritmo de búsqueda local es por lo general realmente más rápido para encontrar un modelo cuando éste existe, el agente se podría justificar asumiendo que el impedimento para encontrar un modelo indica *insatisfacibilidad*. Desde luego que esto no es lo mismo que una demostración, y el agente se lo debería pensar dos veces antes de apostar su vida en ello.

Problemas duros de *satisfacibilidad*

Vamos a ver ahora cómo trabaja en la práctica el DPLL. En concreto, estamos interesados en los problemas *duros* (o *complejos*), porque los problemas *fáciles* se pueden resolver con cualquier algoritmo antiguo. En el Capítulo 5 hicimos algunos descubrimientos sorprendentes acerca de cierto tipo de problemas. Por ejemplo, el problema de las n-reinas (pensado como un problema absolutamente difícil para los algoritmos de búsqueda con *backtracking*) resultó ser trivialmente sencillo para los métodos de búsqueda local, como el min-conflictos. Esto es a causa de que las soluciones están distribuidas muy densamente en el espacio de asignaciones, y está garantizado que cualquier asignación inicial tenga cerca una solución. Así, el problema de las n-reinas es sencillo porque está **bajo restricciones**.

BAJO RESTRICCIONES

Cuando observamos los problemas de *satisfacibilidad* en forma normal conjuntiva, un problema bajo restricciones es aquel que tiene relativamente *pocas* cláusulas res-