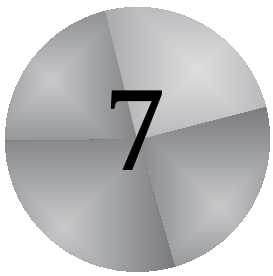


7 Agentes lógicos

Donde diseñaremos agentes que pueden construir representaciones del mundo, utilizar un proceso de inferencia para derivar nuevas representaciones del mundo, y emplear éstas para deducir qué hacer.

En este capítulo se introducen los agentes basados en conocimiento. Los conceptos que discutiremos (la *representación* del conocimiento y los procesos de *razonamiento* que permiten que éste evolucione) son centrales en todo el ámbito de la inteligencia artificial.

De algún modo, las personas conocen las cosas y realizan razonamientos. Tanto el conocimiento como el razonamiento son también importantes para los agentes artificiales, porque les permiten comportamientos con éxito que serían muy difíciles de alcanzar mediante otros mecanismos. Ya hemos visto cómo el conocimiento acerca de los efectos de las acciones permiten a los agentes que resuelven problemas actuar correctamente en entornos complejos. Un agente reflexivo sólo podría hallar un camino de Arad a Bucarest mediante la suerte del principiante. Sin embargo, el conocimiento de los agentes que resuelven problemas es muy específico e inflexible. Un programa de ajedrez puede calcular los movimientos permitidos de su rey, pero no puede saber de ninguna manera que una pieza no puede estar en dos casillas diferentes al mismo tiempo. Los agentes basados en conocimiento se pueden aprovechar del conocimiento expresado en formas muy genéricas, combinando y recombinando la información para adaptarse a diversos propósitos. A veces, este proceso puede apartarse bastante de las necesidades del momento (como cuando un matemático demuestra un teorema o un astrónomo calcula la esperanza de vida de la Tierra).

El conocimiento y el razonamiento juegan un papel importante cuando se trata con entornos parcialmente observables. Un agente basado en conocimiento puede combinar

el conocimiento general con las percepciones reales para inferir aspectos ocultos del estado del mundo, antes de seleccionar cualquier acción. Por ejemplo, un médico diagnostica a un paciente (es decir, infiere una enfermedad que no es directamente observable) antes de seleccionar un tratamiento. Parte del conocimiento que utiliza el médico está en forma de reglas que ha aprendido de los libros de texto y sus profesores, y parte en forma de patrones de asociación que el médico no es capaz de describir explícitamente. Si este conocimiento está en la cabeza del médico, es su conocimiento.

El entendimiento del lenguaje natural también necesita inferir estados ocultos, en concreto, la intención del que habla. Cuando escuchamos, «John vió el diamante a través de la ventana y lo codició», sabemos que «lo» se refiere al diamante y no a la ventana (quizá de forma inconsciente, razonamos con nuestro conocimiento acerca del papel relativo de las cosas). De forma similar, cuando escuchamos, «John lanzó el ladrillo a la ventana y se rompió», sabemos que «se» se refiere a la ventana. El razonamiento nos permite hacer frente a una variedad virtualmente infinita de manifestaciones utilizando un conjunto finito de conocimiento de sentido común. Los agentes que resuelven problemas presentan dificultades con este tipo de ambigüedad debido a que su representación de los problemas con contingencias es inherentemente exponencial.

Nuestra principal razón para estudiar los agentes basados en conocimiento es su flexibilidad. Ellos son capaces de aceptar tareas nuevas en forma de objetivos descritos explícitamente, pueden obtener rápidamente competencias informándose acerca del conocimiento del entorno o aprendiéndolo, y pueden adaptarse a los cambios del entorno actualizando el conocimiento relevante.

En la Sección 7.1 comenzamos con el diseño general del agente. En la Sección 7.2 se introduce un nuevo entorno muy sencillo, el mundo de *wumpus*, y se muestra la forma de actuar de un agente basado en conocimiento sin entrar en los detalles técnicos. Entonces, en la Sección 7.3, explicamos los principios generales de la **lógica**. La lógica será el instrumento principal para la representación del conocimiento en toda la Parte III de este libro. El conocimiento de los agentes lógicos siempre es *categórico* (cada proposición acerca del mundo es verdadera o falsa, si bien, el agente puede ser agnóstico acerca de algunas proposiciones).

La lógica presenta la ventaja pedagógica de ser un ejemplo sencillo de representación para los agentes basados en conocimiento, pero tiene serias limitaciones. En concreto, gran parte del razonamiento llevado a cabo por las personas y otros agentes en entornos parcialmente observables se basa en manejar conocimiento que es *incierto*. La lógica no puede representar bien esta incertidumbre, así que trataremos las probabilidades en la Parte V, que sí puede. Y en la Parte VI y la Parte VII trataremos otras representaciones, incluidas algunas basadas en matemáticas continuas como combinaciones de funciones Gaussianas, redes neuronales y otras representaciones.

En la Sección 7.4 de este capítulo se presenta una lógica muy sencilla denominada **lógica proposicional**. Aunque es mucho menos expresiva que la **lógica de primer orden** (Capítulo 8), la lógica proposicional nos permitirá ilustrar los conceptos fundamentales de la lógica. En las secciones 7.5 y 7.6 describiremos la tecnología, que está bastante desarrollada, para el razonamiento basado en lógica proposicional. Finalmente, en la sección 7.7 se combina el concepto de agente lógico con la tecnología de la lógica proposicional para la construcción de unos agentes muy sencillos en nuestro ejemplo

del mundo de *wumpus*. Se identifican ciertas deficiencias de la lógica proposicional, que nos permitirán el desarrollo de lógicas más potentes en los capítulos siguientes.

7.1 Agentes basados en conocimiento

BASE DE CONOCIMIENTO

SENTENCIA

LENGUAJE DE REPRESENTACIÓN DEL CONOCIMIENTO

El componente principal de un agente basado en conocimiento es su **base de conocimiento**, o BC. Informalmente, una base de conocimiento es un conjunto de **sentencias**. (Aquí «sentencia» se utiliza como un término técnico. Es parecido, pero no idéntico, a las sentencias en inglés u otros lenguajes naturales.) Cada sentencia se expresa en un lenguaje denominado **lenguaje de representación del conocimiento** y representa alguna aserción acerca del mundo.

Debe haber un mecanismo para añadir sentencias nuevas a la base de conocimiento, y uno para preguntar qué se sabe en la base de conocimiento. Los nombres estándar para estas dos tareas son DECIR y PREGUNTAR, respectivamente. Ambas tareas requieren realizar **inferencia**, es decir, derivar nuevas sentencias de las antiguas. En los **agentes lógicos**, que son el tema principal de estudio de este capítulo, la inferencia debe cumplir con el requisito esencial de que cuando se PREGUNTA a la base de conocimiento, la respuesta debe seguirse de lo que se HA DICHO a la base de conocimiento previamente. Más adelante, en el capítulo, seremos más precisos en cuanto a la palabra «seguirse». Por ahora, tómate su significado en el sentido de que la inferencia no se inventaría cosas poco a poco.

INFERENCIA

AGENTES LÓGICOS

La Figura 7.1 muestra el esquema general de un programa de un agente basado en conocimiento. Al igual que todos nuestros agentes, éste recibe una percepción como entrada y devuelve una acción. El agente mantiene una base de conocimiento, *BC*, que inicialmente contiene algún **conocimiento de antecedentes**. Cada vez que el programa del agente es invocado, realiza dos cosas. Primero, DICE a la base de conocimiento lo que ha percibido. Segundo, PREGUNTA a la base de conocimiento qué acción debe ejecutar. En este segundo proceso de responder a la pregunta, se debe realizar un razonamiento extensivo acerca del estado actual del mundo, de los efectos de las posibles acciones, etcétera. Una vez se ha escogido la acción, el agente graba su elección mediante un DECIR y ejecuta la acción. Este segundo DECIR es necesario para permitirle a la base de conocimiento saber que la acción hipotética realmente se ha ejecutado.

CONOCIMIENTO DE ANTECEDENTES

```
función AGENTE-BC(percepción) devuelve una acción
  variables estáticas: BC, una base de conocimiento
                       t, un contador, inicializado a 0, que indica el tiempo

  DECIR(BC, CONSTRUIR-SENTENCIA-DE-PERCEPCIÓN(percepción, t))
  acción ← PREGUNTAR(BC, PEDIR-ACCIÓN(t))
  DECIR(BC, CONSTRUIR-SENTENCIA-DE-ACCIÓN(acción, t))
  t ← t + 1
  devolver acción
```

Figura 7.1 Un agente basado en conocimiento genérico.

Los detalles del lenguaje de representación están ocultos en las dos funciones que implementan la interfaz entre los sensores, los accionadores, el núcleo de representación y el sistema de razonamiento. CONSTRUIR-SENTENCIA-DE-PERCEPCIÓN toma una percepción y un instante de tiempo y devuelve una sentencia afirmando lo que el agente ha percibido en ese instante de tiempo. PEDIR-ACCIÓN toma un instante de tiempo como entrada y devuelve una sentencia para preguntarle a la base de conocimiento qué acción se debe realizar en ese instante de tiempo. Los detalles de los mecanismos de inferencia están ocultos en DECIR y PREGUNTAR. En las próximas secciones del capítulo se mostrarán estos detalles.

El agente de la Figura 7.1 se parece bastante a los agentes con estado interno descritos en el Capítulo 2. Pero gracias a las definiciones de DECIR y PREGUNTAR, el agente basado en conocimiento no obtiene las acciones mediante un proceso arbitrario. Es compatible con una descripción al **nivel de conocimiento**, en el que sólo necesitamos especificar lo que el agente sabe y los objetivos que tiene para establecer su comportamiento. Por ejemplo, un taxi automatizado podría tener el objetivo de llevar un pasajero al condado de Marin, y podría saber que está en San Francisco y que el puente Golden Gate es el único enlace entre las dos localizaciones. Entonces podemos esperar que el agente cruce el puente Golden Gate *porque él sabe que hacerlo le permitirá alcanzar su objetivo*. Fíjate que este análisis es independiente de cómo el taxi trabaja al **nivel de implementación**. Al agente no le debe importar si el conocimiento geográfico está implementado mediante listas enlazadas o mapas de píxeles, o si su razonamiento se realiza mediante la manipulación de textos o símbolos almacenados en registros, o mediante la propagación de señales en una red de neuronas.

NIVEL DE CONOCIMIENTO

NIVEL DE IMPLEMENTACIÓN

Tal como comentamos en la introducción del capítulo, *uno puede construir un agente basado en conocimiento simplemente DICIÉNDOLE al agente lo que necesita saber*. El programa del agente, inicialmente, antes de que empiece a recibir percepciones, se construye mediante la adición, una a una, de las sentencias que representan el conocimiento del entorno que tiene el diseñador. El diseño del lenguaje de representación que permita, de forma más fácil, expresar este conocimiento mediante sentencias simplifica muchísimo el problema de la construcción del agente. Este enfoque en la construcción de sistemas se denomina **enfoque declarativo**. Por el contrario, el enfoque procedural codifica los comportamientos que se desean obtener directamente en código de programación; mediante la minimización del papel de la representación explícita y del razonamiento se pueden obtener sistemas mucho más eficientes. En la Sección 7.7 veremos agentes de ambos tipos. En los 70 y 80, defensores de los dos enfoques se enfrentaban en acalorados debates. Ahora sabemos que para que un agente tenga éxito su diseño debe combinar elementos declarativos y procedurales.

ENFOQUE DECLARATIVO

A parte de DECIRLE al agente lo que necesita saber, podemos proveer a un agente basado en conocimiento de los mecanismos que le permitan aprender por sí mismo. Estos mecanismos, que se verán en el Capítulo 18, crean un conocimiento general acerca del entorno con base en un conjunto de percepciones. Este conocimiento se puede incorporar a la base de conocimiento del agente y utilizar para su toma de decisiones. De esta manera, el agente puede ser totalmente autónomo.

Todas estas capacidades (representación, razonamiento y aprendizaje) se apoyan en la teoría y tecnología de la lógica, desarrolladas a lo largo de los siglos. Sin embargo,

antes de explicar dichas teoría y tecnología, crearemos un mundo sencillo que nos permitirá ilustrar estos mecanismos.

7.2 El mundo de *wumpus*

MUNDO DE *WUMPUS*

El **mundo de *wumpus*** es una cueva que está compuesta por habitaciones conectadas mediante pasillos. Escondido en algún lugar de la cueva está el *wumpus*, una bestia que se come a cualquiera que entre en su habitación. El *wumpus* puede ser derribado por la flecha de un agente, y éste sólo dispone de una. Algunas habitaciones contienen hoyos sin fondo que atrapan a aquel que deambula por dichas habitaciones (menos al *wumpus,* que es demasiado grande para caer en ellos). El único premio de vivir en este entorno es la posibilidad de encontrar una pila de oro. Aunque el mundo de *wumpus* pertenece más al ámbito de los juegos por computador, es un entorno perfecto para evaluar los agentes inteligentes. Michael Genesereth fue el primero que lo propuso.

En la Figura 7.2 se muestra un ejemplo del mundo de *wumpus*. La definición precisa del entorno de trabajo, tal como sugerimos en el Capítulo 2, mediante la descripción REAS, es:

- **Rendimiento:** $+1.000$ por recoger el oro, -1.000 por caer en un hoyo o ser comido por el *wumpus*, -1 por cada acción que se realice y -10 por lanzar la flecha.
- **Entorno:** una matriz de 4×4 habitaciones. El agente siempre comienza en la casilla etiquetada por [1, 1], y orientado a la derecha. Las posiciones del oro y del *wumpus* se escogen de forma aleatoria, mediante una distribución uniforme, a partir de todas las casillas menos la de salida del agente. Además, con probabilidad 0,2, cada casilla puede tener un hoyo.
- **Actuadores:** el agente se puede mover hacia delante, girar a la izquierda 90°, o a la derecha 90°. El agente puede fallecer de muerte miserable si entra en una casilla en la que hay un hoyo o en la que está el *wumpus* vivo. (No sucede nada malo, aunque huele bastante mal, si el agente entra en una casilla con un *wumpus* muerto.) Si hay un muro en frente y el agente intenta avanzar, no sucede nada. La acción *Agarrar* se puede utilizar para tomar un objeto de la misma casilla en donde se encuentre el agente. La acción *Disparar* se puede utilizar para lanzar una flecha en línea recta, en la misma dirección y sentido en que se encuentra situado el agente. La flecha avanza hasta que se choca contra un muro o alcanza al *wumpus* (y entonces lo mata). El agente sólo dispone de una flecha, así que, sólo tiene efecto el primer *Disparo*.
- **Sensores:** el agente dispone de cinco sensores, y cada uno le da una pequeña información acerca del entorno.
  - — El agente percibirá un mal hedor si se encuentra en la misma casilla que el *wumpus* o en las directamente adyacentes a él (no en diagonal).
  - — El agente recibirá una pequeña brisa en las casillas directamente adyacentes donde hay un hoyo.
  - — El agente verá un resplandor en las casillas donde está el oro.

— Si el agente intenta atravesar un muro sentirá un golpe.
— Cuando el *wumpus* es aniquilado emite un desconsolado grito que se puede oír en toda la cueva.

Las percepciones que recibirá el agente se representan mediante una lista de cinco símbolos: por ejemplo, si el agente percibe un mal hedor o una pequeña brisa, pero no ve un resplandor, no siente un golpe, ni oye un grito, el agente recibe la lista [*Hedor*, *Brisa*, *Nada*, *Nada*, *Nada*].

En el Ejercicio 7.1 se pide definir el entorno del *wumpus* a partir de las diferentes dimensiones tratadas en el Capítulo 2. La principal dificultad para el agente es su ignorancia inicial acerca de la configuración del entorno; para superar esta ignorancia parece que se requiere el razonamiento lógico. En muchos casos del mundo de *wumpus*, para el agente es posible obtener el oro de forma segura. En algunos casos, el agente debe escoger entre volver a casa con las manos vacías o arriesgarse para encontrar el oro. Cerca del 21 por ciento de los casos son completamente injustos, ya que el oro se encuentra en un hoyo o rodeado de ellos.

Vamos a ver un agente basado en conocimiento en el mundo de *wumpus,* explorando el entorno que se muestra en la Figura 7.2. La base de conocimiento inicial del agente contiene las reglas del entorno, tal como hemos listado anteriormente; en concreto, el agente sabe que se encuentra en la casilla [1, 1] y que ésta es una casilla segura. Veremos cómo su conocimiento evoluciona a medida que recibe nuevas percepciones y las acciones se van ejecutando.

La primera percepción es [*Nada, Nada, Nada, Nada, Nada*], de la cual, el agente puede concluir que las casillas vecinas son seguras. La Figura 7.3(a) muestra el conocimiento del estado del agente en ese momento. En esta figura mostramos (algunas de) las sentencias de la base de conocimiento utilizando letras como la *B* (de brisa) y *OK* (de casilla segura, no hay hoyo ni está el *wumpus*) situadas en las casillas adecuadas. En cambio, la Figura 7.2 muestra el mundo tal como es.

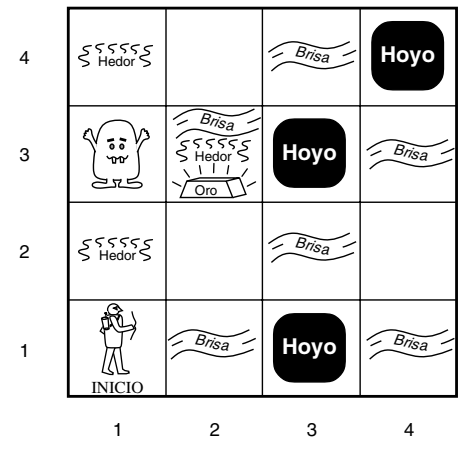


Figura 7.2 Un mundo de wumpus típico. El agente está situado en la esquina inferior izquierda.

De los hechos, que no hay mal hedor ni brisa en la casilla [1, 1], el agente infiere que las casillas [1, 2] y [2, 1] están libres de peligro. Entonces las marca con *OK* para indicar esta conclusión. Un agente que sea cauto sólo se moverá hacia una casilla en la que él sabe que está *OK*. Supongamos que el agente decide moverse hacia delante a la casilla [2,1], alcanzando la situación de la Figura 7.3(b).

El agente detecta una brisa en la casilla [2, 1], por lo tanto, debe haber un hoyo en alguna casilla vecina. El hoyo no puede estar en la casilla [1, 1], teniendo en cuenta las reglas del juego, así que debe haber uno en la casilla [2, 2] o en la [3, 1], o en ambas. La etiqueta *¿P?* de la Figura 7.3(b), nos indica que puede haber un posible hoyo en estas casillas. En este momento, sólo se conoce una casilla que está *OK* y que no ha sido visitada aún. Así que el agente prudente girará para volver a la casilla [1, 1] y entonces se moverá a la [1, 2].

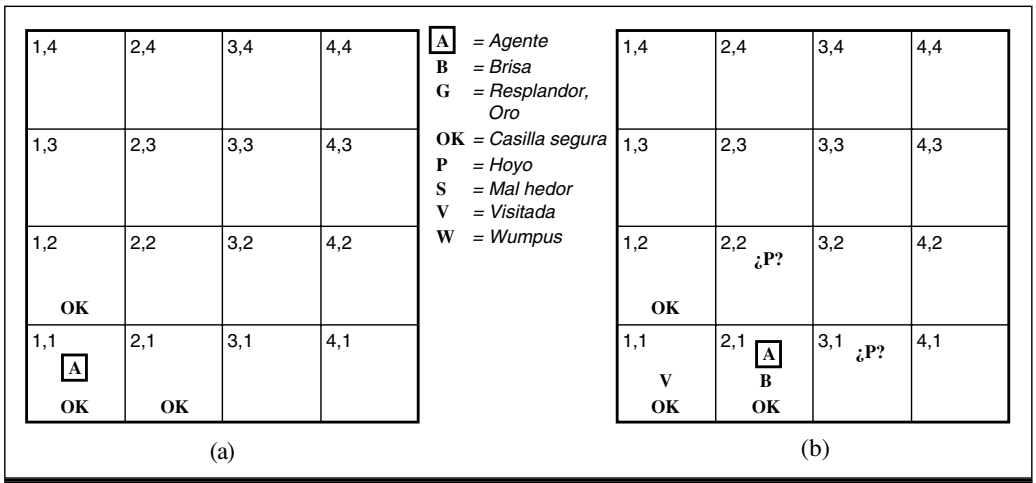


Figura 7.3 El primer paso dado por el agente en el mundo de *wumpus*. (a) La situación inicial, después de la percepción [*Nada, Nada, Nada, Nada, Nada*]. (b) Después del primer movimiento, con la percepción [*Nada, Brisa, Nada, Nada, Nada*].

La nueva percepción en la casilla [1, 2] es [*Hedor, Nada, Nada, Nada, Nada*], obteniendo el estado de conocimiento que se muestra en la Figura 7.4(a). El mal hedor en la [1, 2] significa que debe haber un *wumpus* muy cerca. Pero el *wumpus* no puede estar en la [1, 1], teniendo en cuenta las reglas del juego, y tampoco puede estar en [2, 2] (o el agente habría detectado un mal hedor cuando estaba en la [2, 1]). Entonces el agente puede inferir que el *wumpus* se encuentra en la casilla [1, 3], que se indica con la etiqueta *¡W!* Más aún, la ausencia de *Brisa* en la casilla [1, 2] implica que no hay un hoyo en la [2, 2]. Como ya habíamos inferido que debía haber un hoyo en la casilla [2, 2] o en la [3, 1], éste debe estar en la [3, 1]. Todo esto es un proceso de inferencia realmente costoso, ya que debe combinar el conocimiento adquirido en diferentes instantes de tiempo y en distintas situaciones, para resolver la falta de percepciones y poder realizar cualquier paso crucial. La inferencia pertenece a las habilidades de muchos animales, pero es típico del tipo de razonamiento que un agente lógico realiza.

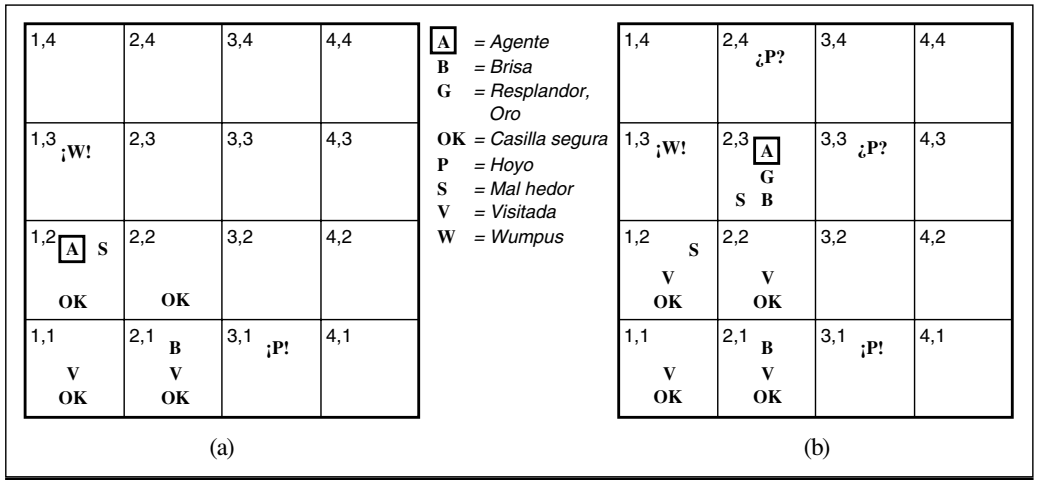


Figura 7.4 Los dos últimos estados en el desarrollo del juego. (a) Después del tercer movimiento, con la percepción [*Hedor, Nada, Nada, Nada, Nada*]. (b) Después del quinto movimiento, con la percepción [*Hedor, Brisa, Resplandor, Nada, Nada*].

El agente ha demostrado en este momento que no hay ni un hoyo ni un *wumpus* en la casilla [2, 2], así que está *OK* para desplazarse a ella. No mostraremos el estado de conocimiento del agente en [2, 2]; asumimos que el agente gira y se desplaza a [2, 3], tal como se muestra en la Figura 7.4(b). En la casilla [2, 3] el agente detecta un resplandor, entonces el agente cogería el oro y acabaría el juego.

En cada caso en que el agente saca una conclusión a partir de la información que tiene disponible, se garantiza que dicha conclusión es correcta si la información disponible también lo es. Esta es una propiedad fundamental del razonamiento lógico. En lo que queda del capítulo vamos a describir cómo construir agentes lógicos que pueden representar la información necesaria para sacar conclusiones similares a las que hemos descrito en los párrafos anteriores.

7.3 Lógica

Esta sección presenta un repaso de todos los conceptos fundamentales de la representación y el razonamiento lógicos. Dejamos los detalles técnicos de cualquier clase concreta de lógica para la siguiente sección. En lugar de ello, utilizaremos ejemplos informales del mundo de *wumpus* y del ámbito familiar de la aritmética. Adoptamos este enfoque poco común, porque los conceptos de la lógica son bastante más generales y bellos de lo que se piensa a priori.

SINTAXIS

En la sección 7.1 dijimos que las bases de conocimiento se componen de sentencias. Estas sentencias se expresan de acuerdo a la **sintaxis** del lenguaje de representación, que especifica todas las sentencias que están bien formadas. El concepto de sintaxis está suficientemente claro en la aritmética: «$x + y = 4$» es una sentencia bien formada, mien-

tras que «*x2y+ =*» no lo es. Por lo general, la sintaxis de los lenguajes lógicos (y la de los aritméticos, en cuanto al mismo tema) está diseñada para escribir libros y artículos. Hay literalmente docenas de diferentes sintaxis, algunas que utilizan muchas letras griegas y símbolos matemáticos complejos, otras basadas en diagramas con flechas y burbujas visualmente muy atractivas. Y sin embargo, en todos estos casos, las sentencias de la base de conocimiento del agente son configuraciones físicas reales (de las partes) del agente. El razonamiento implica generar y manipular estas configuraciones.

SEMÁNTICA

VALOR DE VERDAD

MUNDO POSIBLE

Una lógica también debe definir la **semántica** del lenguaje. Si lo relacionamos con el lenguaje hablado, la semántica trata el «significado» de las sentencias. En lógica, esta definición es bastante más precisa. La semántica del lenguaje define el **valor de verdad** de cada sentencia respecto a cada **mundo posible**. Por ejemplo, la semántica que se utiliza en la aritmética especifica que la sentencia «$x + y = 4$» es verdadera en un mundo en el que x sea 2 e y sea 2, pero falsa en uno en el que x sea 1 e y sea 1[1]. En las lógicas clásicas cada sentencia debe ser o bien verdadera o bien falsa en cada mundo posible, no puede ser lo uno y lo otro[2].

MODELO

Cuando necesitemos ser más precisos, utilizaremos el término **modelo** en lugar del de «mundo posible». (También utilizaremos la frase «m es un modelo de α» para indicar que la sentencia α es verdadera en el modelo m.) Siempre que podamos pensar en los mundos posibles como en (potencialmente) entornos reales en los que el agente pueda o no estar, los modelos son abstracciones matemáticas que simplemente nos permiten definir la verdad o falsedad de cada sentencia que sea relevante. Informalmente podemos pensar, por ejemplo, en que x e y son el número de hombres y mujeres que están sentados en una mesa jugando una partida de *bridge*, y que la sentencia $x + y = 4$ es verdadera cuando los que están jugando son cuatro en total; formalmente, los modelos posibles son justamente todas aquellas posibles asignaciones de números a las variables x e y. Cada una de estas asignaciones indica el valor de verdad de cualquier sentencia aritmética cuyas variables son x e y.

IMPLICACIÓN

Ahora que ya disponemos del concepto de valor de verdad, ya estamos preparados para hablar acerca del razonamiento lógico. Éste requiere de la relación de **implicación** lógica entre las sentencias (la idea de que una sentencia *se sigue lógicamente* de otra sentencia). Su notación matemática es

$$\alpha \models \beta$$

para significar que la sentencia α implica la sentencia β. La definición formal de implicación es esta: $\alpha \models \beta$ si y sólo si en cada modelo en el que α es verdadera, β también lo es. Otra forma de definirla es que si α es verdadera, β también lo debe ser. Informalmente, el valor de verdad de β «está contenido» en el valor de verdad de α. La relación de implicación nos es familiar en la aritmética; no nos disgusta la idea de que la sentencia $x + y = 4$ implica la sentencia $4 = x + y$. Es obvio que en cada modelo en

[1] El lector se habrá dado cuenta de la semejanza entre el concepto de valor de verdad de las sentencias y la satisfacción de restricciones del Capítulo 5. No es casualidad (los lenguajes de restricciones son en efecto lógicas y la resolución de restricciones un tipo de razonamiento lógico).

[2] La **lógica difusa**, que se verá en el Capítulo 14, nos permitirá tratar con grados de valores de verdad.

el que $x + y = 4$ (como lo es el modelo en el que x es 2 e y es 2) también lo es para $4 = x + y$. Pronto veremos que una base de conocimiento puede ser considerada como una afirmación, y a menudo hablaremos de que una base de conocimiento implica una sentencia.

Ahora podemos aplicar el mismo tipo de análisis que utilizamos en la sección anterior al mundo del *wumpus*. Si tomamos la situación de la Figura 7.3(b): el agente no ha detectado nada en la casilla [1, 1], y ha detectado una brisa en la [2, 1]. Estas percepciones, combinadas con el conocimiento del agente sobre las reglas que definen el funcionamiento del mundo de *wumpus* (la descripción REAS de la página 221), constituyen su BC. El agente está interesado (entre otras cosas) en si las casillas adyacentes [1, 2], [2, 2] y [3, 1] tienen hoyos sin fondo. Cada una de las tres casillas pueden o no tener un hoyo, por lo tanto (al menos en este ejemplo) hay $2^3 = 8$ modelos posibles. Tal como se muestran en la Figura 7.5[3].

La BC es falsa en los modelos que contradicen lo que el agente sabe (por ejemplo, la BC es falsa en cualquier modelo en el que la casilla [1, 2] tenga un hoyo), porque no ha detectado ninguna brisa en la casilla [1, 1]. De hecho, hay tres modelos en los que la BC es verdadera, los que se muestran como subconjunto de los modelos de la Figura 7.5. Ahora consideremos las dos conclusiones:

α_1 = «No hay un hoyo en la casilla [1, 2]».
α_2 = «No hay un hoyo en la casilla [2, 2]».

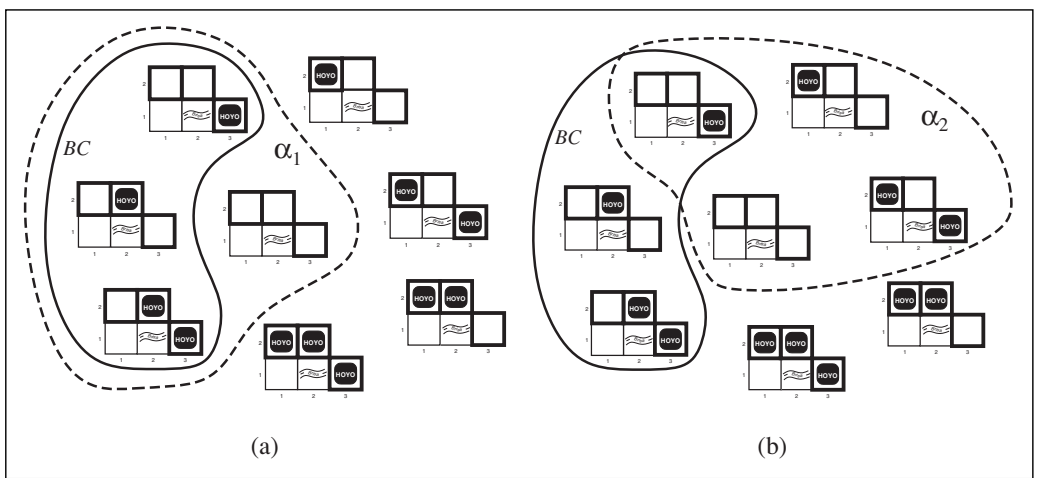


Figura 7.5 Modelos posibles para la presencia de hoyos en las casillas [1, 2], [2, 2] y [3, 1], dadas las observaciones de que no hay nada en la casilla [1, 1] y hay una brisa en la [2, 1]. (a) Modelos de la base de conocimiento y α_1 (no hay un hoyo en [1, 2]). (b) Modelos de la base de conocimiento y α_2 (no hay un hoyo en [2, 2]).

[3] En la Figura 7.5 los modelos se muestran como mundos parciales, porque en realidad tan sólo son asignaciones de *verdadero* y *falso* a sentencias como «hay un hoyo en la casilla [1, 2]», etc. Los modelos, desde el punto de vista matemático, no necesitan tener horribles wumpus etéreos ambulando en ellos.

Hemos rodeado (con línea discontinua) los modelos de α_1 y α_2 en las Figuras 7.5(a) y 7.5(b) respectivamente. Si observamos, podemos ver lo siguiente:

en cada modelo en el que la *BC* es verdadera, α_1 también lo es.

De aquí que $BC \models \alpha_1$: no hay un hoyo en la casilla [1, 2]. También podemos ver que

en algunos modelos en los que la *BC* es verdadera, α_2 es falsa.

De aquí que $BC \not\models \alpha_2$: el agente no puede concluir que no haya un hoyo en la casilla [2, 2]. (Ni tampoco puede concluir que lo haya.[4])

El ejemplo anterior no sólo nos muestra el concepto de implicación, sino, también cómo el concepto de implicación se puede aplicar para derivar conclusiones, es decir, llevar a cabo la **inferencia lógica**. El algoritmo de inferencia que se muestra en la Figura 7.5 se denomina **comprobación de modelos** porque enumera todos los modelos posibles y comprueba si α es verdadera en todos los modelos en los que la *BC* es verdadera.

INFERENCIA LÓGICA

COMPROBACIÓN DE MODELOS

Para entender la implicación y la inferencia nos puede ayudar pensar en el conjunto de todas las consecuencias de la *BC* como en un pajar, y en α como en una aguja. La implicación es como la aguja que se encuentra en el pajar, y la inferencia consiste en encontrarla. Esta distinción se expresa mediante una notación formal: si el algoritmo de inferencia i puede derivar α de la *BC*, entonces escribimos

$$BC \vdash_i \alpha,$$

que se pronuncia como «α se deriva de la *BC* mediante i» o «i deriva α de la *BC*».

Se dice que un algoritmo de inferencia que deriva sólo sentencias implicadas es **sólido** o que **mantiene la verdad**. La solidez es una propiedad muy deseable. Un procedimiento de inferencia no sólido tan sólo se inventaría cosas poco a poco (anunciaría el descubrimiento de agujas que no existirían). Se puede observar fácilmente que la comprobación de modelos, cuando es aplicable[5], es un procedimiento sólido.

SÓLIDO

MANTENIMIENTO DE LA VERDAD

También es muy deseable la propiedad de **completitud**: un algoritmo de inferencia es completo si puede derivar cualquier sentencia que está implicada. En los pajares reales, que son de tamaño finito, parece obvio que un examen sistemático siempre permite decidir si hay una aguja en el pajar. Sin embargo, en muchas bases de conocimiento, el pajar de las consecuencias es infinito, y la completitud pasa a ser una problemática importante[6]. Por suerte, hay procedimientos de inferencia completos para las lógicas que son suficientemente expresivas para manejar muchas bases de conocimiento.

COMPLETITUD

---

[4] El agente podría calcular la *probabilidad* de que haya un hoyo en la casilla [2, 2]; en el Capítulo 13 lo veremos.

[5] La comprobación de modelos trabaja bien cuando el espacio de los modelos es finito (por ejemplo, en un mundo del *wumpus* de tamaño de casillas fijo). Por el otro lado, en la aritmética, el espacio de modelos es infinito: aun limitándonos a los enteros, hay infinitos pares de valores para x e y en la sentencia $x + y = 4$.

[6] Compárelo con el caso de la búsqueda en espacios infinitos de estados del Capítulo 3, en donde la búsqueda del primero en profundidad no es completa.

Hemos descrito un proceso de razonamiento en el que se garantiza que las conclusiones sean verdaderas en cualquier mundo en el que las premisas lo sean; en concreto, *si una BC es verdadera en el mundo real, entonces cualquier sentencia α que se derive de la BC mediante un procedimiento de inferencia sólido también será verdadera en el mundo real*. Así, mientras que un proceso de inferencia opera con la «sintaxis» (las configuraciones físicas internas, tales como los bits en los registros o los patrones de impulsos eléctricos en el cerebro) el proceso se corresponde con la relación del mundo real según la cual algún aspecto del mundo real es cierto[7] en virtud de que otros aspectos del mundo real lo son. En la Figura 7.6 se ilustra esta correspondencia entre el mundo y la representación.

DENOTACIÓN

El último asunto que debe ser tratado mediante una computación basada en agentes lógicos es el de la **denotación** (la conexión, si la hay, entre los procesos de razonamiento lógico y el entorno real en el que se encuentra el agente). En concreto, *¿cómo sabemos que la BC es verdadera en el mundo real?* (Después de todo, la *BC* sólo es «sintaxis» dentro de la cabeza del agente.) Ésta es una cuestión filosófica acerca de la cual se han escrito muchos, muchísimos libros. (Ver Capítulo 26.) Una respuesta sencilla es que los sensores del agente crean la conexión. Por ejemplo, nuestro agente del mundo de *wumpus* dispone de un sensor de olores. El programa del agente crea una sentencia adecuada siempre que hay un olor. Entonces, siempre que esa sentencia esté en la base de conocimiento será verdadera en el mundo real. Así, el significado y el valor de verdad de las sentencias de las percepciones se definen mediante el proceso de los sensores y el de la construcción de las sentencias, activada por el proceso previo. ¿Qué sucede con el resto del conocimiento del agente, tal como sus creencias acerca de que el *wumpus* causa mal hedor en las casillas adyacentes? Ésta no es una representación directa de una simple percepción, pero sí es una regla general (derivada, quizá, de la experiencia de las percepciones aunque no idéntica a una afirmación de dicha experiencia). Las reglas generales como ésta se generan mediante un proceso de construcción de sentencias denominado **aprendizaje**, que es el tema que trataremos en la Parte VI. El aprendizaje es falible. Puede darse el caso en el que el *wumpus* cause mal hedor *excepto el 29 de febrero en años bisiestos*, que

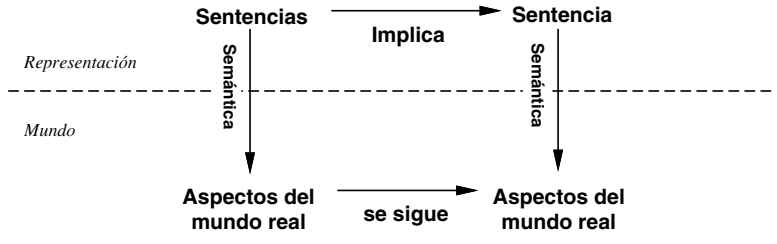


Figura 7.6 Las sentencias son configuraciones físicas del agente, y el razonamiento es el proceso de construcción de nuevas configuraciones físicas a partir de las antiguas. El razonamiento lógico debería asegurar que las nuevas configuraciones representen aspectos del mundo que realmente se siguen de los aspectos que las antiguas configuraciones representan.

[7] Tal como escribió Wittgenstein (1922) en su famoso *Tractatus*: «El mundo es cada cosa que es cierta».

es cuando toma su baño. Así, la *BC* no sería verdadera en el mundo real, sin embargo, mediante procedimientos de aprendizaje buenos no hace falta ser tan pesimistas.

7.4 Lógica proposicional: una lógica muy sencilla

LÓGICA PROPOSICIONAL

Ahora vamos a presentar una lógica muy sencilla llamada **lógica proposicional**[8]. Vamos a cubrir tanto la sintaxis como la semántica (la manera como se define el valor de verdad de las sentencias) de la lógica proposicional. Luego trataremos la **implicación** (la relación entre una sentencia y la que se sigue de ésta) y veremos cómo todo ello nos lleva a un algoritmo de inferencia lógica muy sencillo. Todo ello tratado, por supuesto, en el mundo de *wumpus*.

Sintaxis

SENTENCIAS ATÓMICAS

SÍMBOLO PROPOSICIONAL

La **sintaxis** de la lógica proposicional nos define las sentencias que se pueden construir. Las **sentencias atómicas** (es decir, los elementos sintácticos indivisibles) se componen de un único **símbolo proposicional**. Cada uno de estos símbolos representa una proposición que puede ser verdadera o falsa. Utilizaremos letras mayúsculas para estos símbolos: P, Q, R, y siguientes. Los nombres de los símbolos suelen ser arbitrarios pero a menudo se escogen de manera que tengan algún sentido mnemotécnico para el lector. Por ejemplo, podríamos utilizar $W_{1,3}$ para representar que el *wumpus* se encuentra en la casilla [1, 3]. (Recuerde que los símbolos como $W_{1,3}$ son *atómicos*, esto es, W, 1, y 3 no son partes significantes del símbolo.) Hay dos símbolos proposicionales con significado fijado: *Verdadero*, que es la proposición que siempre es verdadera; y *Falso*, que es la proposición que siempre es falsa.

SENTENCIAS COMPLEJAS

CONECTIVAS LÓGICAS

NEGACIÓN

LITERAL

CONJUNCIÓN

DISYUNCIÓN

IMPLICACIÓN

PREMISA

CONCLUSIÓN

Las **sentencias complejas** se construyen a partir de sentencias más simples mediante el uso de las **conectivas lógicas**, que son las siguientes cinco:

- $\neg$ (no). Una sentencia como $\neg W_{1,3}$ se denomina **negación** de $W_{1,3}$. Un **literal** puede ser una sentencia atómica (un **literal positivo**) o una sentencia atómica negada (un **literal negativo**).
- $\wedge$ (y). Una sentencia que tenga como conectiva principal $\wedge$, como es $W_{1,3} \wedge H_{3,1}$, se denomina **conjunción**; sus componentes son los **conjuntores**.
- $\vee$ (o). Una sentencia que utiliza la conectiva $\vee$, como es $(W_{1,3} \wedge H_{3,1}) \vee W_{2,2}$, es una **disyunción** de los **disyuntores** $(W_{1,3} \wedge H_{3,1})$ y $W_{2,2}$. (Históricamente, la conectiva $\vee$ proviene de «vel» en Latín, que significa «o». Para mucha gente, es más fácil recordarla como la conjunción al revés.)
- $\Rightarrow$ (implica). Una sentencia como $(W_{1,3} \wedge H_{3,1}) \Rightarrow \neg W_{2,2}$ se denomina **implicación** (o condicional). Su **premisa** o **antecedente** es $(W_{1,3} \wedge H_{3,1})$, y su **conclusión** o **consecuente** es $\neg W_{2,2}$. Las implicaciones también se conocen como **reglas** o afir-

[8] A la lógica proposicional también se le denomina **Lógica Booleana**, por el matemático George Boole (1815-1864).

maciones **si-entonces**. Algunas veces, en otros libros, el símbolo de la implicación se representa mediante $\supset$ o $\rightarrow$.

BICONDICIONAL

$\Leftrightarrow$ (si y sólo si). La sentencia $W_{1,3} \Leftrightarrow \neg W_{2,2}$ es una **bicondicional**.

En la Figura 7.7 se muestra una gramática formal de la lógica proposicional; mira la página 984 si no estás familiarizado con la notación BNF.

Sentencia → *Sentencia Atómica* | *Sentencia Compleja*
Sentencia Atómica → **Verdadero** | **Falso** | *Símbolo Proposicional*
Símbolo Proposicional → **P** | **Q** | **R** | **…**
Sentencia Compleja → ¬ *Sentencia*
| (*Sentencia* ∧ *Sentencia*)
| (*Sentencia* ∨ *Sentencia*)
| (*Sentencia* ⇒ *Sentencia*)
| (*Sentencia* ⇔ *Sentencia*)

Figura 7.7 Una gramática BNF (Backus-Naur Form) de sentencias en lógica proposicional.

Fíjese en que la gramática es muy estricta respecto al uso de los paréntesis: cada sentencia construida a partir de conectivas binarias debe estar encerrada en paréntesis. Esto asegura que la gramática no sea ambigua. También significa que tenemos que escribir, por ejemplo, $((A \wedge B) \Rightarrow C)$ en vez de $A \wedge B \Rightarrow C$. Para mejorar la legibilidad, a menudo omitiremos paréntesis, apoyándonos en su lugar en un orden de precedencia de las conectivas. Es una precedencia similar a la utilizada en la aritmética (por ejemplo, $ab + c$ se lee $((ab) + c)$ porque la multiplicación tiene mayor precedencia que la suma). El orden de precedencia en la lógica proposicional (de mayor a menor) es: $\neg$, $\wedge$, $\vee$, $\Rightarrow$ y $\Leftrightarrow$. Así, la sentencia

$$\neg P \vee Q \wedge R \Rightarrow S$$

es equivalente a la sentencia

$$((\neg P) \vee (Q \wedge R)) \Rightarrow S$$

La precedencia entre las conectivas no resuelve la ambigüedad en sentencias como $A \wedge B \wedge C$, que se podría leer como $((A \wedge B) \wedge C)$ o como $(A \wedge (B \wedge C))$. Como estas dos lecturas significan lo mismo según la semántica que mostraremos en la siguiente sección, se permiten este tipo de sentencias. También se permiten sentencias como $A \vee B \vee C$ o $A \Leftrightarrow B \Leftrightarrow C$. Sin embargo, las sentencias como $A \Rightarrow B \Rightarrow C$ no se permiten, ya que su lectura en una dirección y su opuesta tienen significados muy diferentes; en este caso insistimos en la utilización de los paréntesis. Por último, a veces utilizaremos corchetes, en vez de paréntesis, para conseguir una lectura de la sentencia más clara.

Semántica

Una vez especificada la sintaxis de la lógica proposicional, vamos a definir su semántica. La semántica define las reglas para determinar el valor de verdad de una sentencia respecto a un modelo en concreto. En la lógica proposicional un modelo define el va-

lor de verdad (*verdadero* o *falso)*. Por ejemplo, si las sentencias de la base de conocimiento utilizan los símbolos proposicionales $H_{1,2}$, $H_{2,2}$, y $H_{3,1}$, entonces un modelo posible sería

$$m_1 = \{H_{1,2} = falso, H_{2,2} = \text{falso}, H_{3,1} = verdadero\}$$

Con tres símbolos proposicionales hay $2^3 = 8$ modelos posibles, exactamente los que aparecen en la Figura 7.5. Sin embargo, fíjese en que gracias a que hemos concretado la sintaxis, los modelos se convierten en objetos puramente matemáticos sin tener necesariamente una conexión al mundo de *wumpus*. $H_{1,2}$ es sólo un símbolo, podría denotar tanto «hay un hoyo en la casilla [1, 2]», como «estaré en París hoy y mañana».

La semántica en lógica proposicional debe especificar cómo obtener el valor de verdad de *cualquier* sentencia, dado un modelo. Este proceso se realiza de forma recursiva. Todas las sentencias se construyen a partir de las sentencias atómicas y las cinco conectivas lógicas; entonces necesitamos establecer cómo definir el valor de verdad de las sentencias atómicas y cómo calcular el valor de verdad de las sentencias construidas con las cinco conectivas lógicas. Para las sentencias atómicas es sencillo:

- *Verdadero* es verdadero en todos los modelos y *Falso* es falso en todos los modelos.
- El valor de verdad de cada símbolo proposicional se debe especificar directamente para cada modelo. Por ejemplo, en el modelo anterior m_1, $H_{1,2}$ es falso.

Para las sentencias complejas, tenemos reglas como la siguiente

- Para toda sentencia s y todo modelo m, la sentencia $\neg s$ es verdadera en m si y sólo si s es falsa en m.

Este tipo de reglas reducen el cálculo del valor de verdad de una sentencia compleja al valor de verdad de las sentencias más simples. Las reglas para las conectivas se pueden resumir en una **tabla de verdad** que especifica el valor de verdad de cada sentencia compleja según la posible asignación de valores de verdad realizada a sus componentes. En la Figura 7.8 se muestra la tabla de verdad de las cinco conectivas lógicas. Utilizando estas tablas de verdad, se puede obtener el valor de verdad de cualquier sentencia s según un modelo m mediante un proceso de evaluación recursiva muy sencillo. Por ejemplo, la sentencia $\neg H_{1,2} \wedge (H_{2,2} \vee H_{3,1})$ evaluada según m_1, da *verda-*

TABLA DE VERDAD

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
falso	*falso*	*verdadero*	*falso*	*falso*	*verdadero*	*verdadero*
falso	*verdadero*	*verdadero*	*falso*	*verdadero*	*verdadero*	*falso*
verdadero	*falso*	*falso*	*falso*	*verdadero*	*falso*	*falso*
verdadero	*verdadero*	*falso*	*verdadero*	*verdadero*	*verdadero*	*verdadero*

Figura 7.8 Tablas de verdad para las cinco conectivas lógicas. Para utilizar la tabla, por ejemplo, para calcular el valor de $P \vee Q$, cuando P es verdadero y Q falso, primero mire a la izquierda en donde P es *verdadera* y Q es *falsa* (la tercera fila). Entonces mire en esa fila justo en la columna de $P \vee Q$ para ver el resultado: *verdadero*. Otra forma de verlo es pensar en cada fila como en un modelo, y que sus entradas en cada fila dicen para cada columna si la sentencia es verdadera en ese modelo.

dero $\wedge$ (*falso* $\vee$ *verdadero*) = *verdadero* $\wedge$ *verdadero* = *verdadero*. El Ejercicio 7.3 pide que escriba el algoritmo ¿V-VERDAD-LP?(*s, m*) que debe obtener el valor de verdad de una sentencia *s* en lógica proposicional según el modelo *m*.

Ya hemos comentado que una base de conocimiento está compuesta por sentencias. Ahora podemos observar que esa base de conocimiento lógica es una conjunción de dichas sentencias. Es decir, si comenzamos con una *BC* vacía y ejecutamos DECIR(*BC,* S_1)… DECIR(*BC,* S_n) entonces tenemos $BC = S_1 \wedge \ldots \wedge S_n$. Esto significa que podemos manejar bases de conocimiento y sentencias de manera intercambiable.

Los valores de verdad de «y», «o» y «no» concuerdan con nuestra intuición, cuando los utilizamos en lenguaje natural. El principal punto de confusión puede presentarse cuando $P \vee Q$ es verdadero porque *P* lo es, *Q* lo es, o *ambos* lo son. Hay una conectiva diferente denominada «o exclusiva» («xor» para abreviar) que es falsa cuando los dos disyuntores son verdaderos[9]. No hay consenso respecto al símbolo que representa la o exclusiva, siendo las dos alternativas $\dot{\vee}$ y $\oplus$.

El valor de verdad de la conectiva $\Rightarrow$ puede parecer incomprensible al principio, ya que no encaja en nuestra comprensión intuitiva acerca de «*P* implica *Q*» o de «si *P* entonces *Q*». Para una cosa, la lógica proposicional no requiere de una relación de causalidad o relevancia entre *P* y *Q*. La sentencia «que 5 sea impar implica que Tokio es la capital de Japón» es una sentencia verdadera en lógica proposicional (bajo una interpretación normal), aunque pensándolo es, decididamente, una frase muy rara. Otro punto de confusión es que cualquier implicación es verdadera siempre que su antecedente sea falso. Por ejemplo, «que 5 sea par implica que Sam es astuto» es verdadera, independientemente de que Sam sea o no astuto. Parece algo estrafalario, pero tiene sentido si piensa acerca de «$P \Rightarrow Q$» como si dijera, «si *P* es verdadero, entonces estoy afirmando que *Q* es verdadero. De otro modo, no estoy haciendo ninguna afirmación.» La única manera de hacer esta sentencia *falsa* es haciendo que *P* sea cierta y *Q* falsa.

La tabla de verdad de la bicondicional $P \Leftrightarrow Q$ muestra que la sentencia es verdadera siempre que $P \Rightarrow Q$ *y* $Q \Rightarrow P$ lo son. En lenguaje natural a menudo se escribe como «*P* si y sólo si *Q*» o «*P* si *Q*». Las reglas del mundo de *wumpus* se describen mejor utilizando la conectiva $\Leftrightarrow$. Por ejemplo, una casilla tiene corriente de aire *si* alguna casilla vecina tiene un hoyo, y una casilla tiene corriente de aire *sólo si* una casilla vecina tiene un hoyo. De esta manera necesitamos bicondicionales como

$$B_{1,1} \Leftrightarrow (H_{1,2} \vee H_{2,1}),$$

en donde $B_{1,1}$ significa que hay una brisa en la casilla [1,1]. Fíjese en que la implicación

$$B_{1,1} \Rightarrow (H_{1,2} \vee H_{2,1})$$

es verdadera, aunque incompleta, en el mundo de *wumpus*. Esta implicación no descarta modelos en los que $B_{1,1}$ sea falso y $H_{1,2}$ sea verdadero, hecho que violaría las reglas del mundo de *wumpus*. Otra forma de observar esta incompletitud es que la implicación necesita la presencia de hoyos si hay una corriente de aire, mientras que la bicondicional además necesita la ausencia de hoyos si no hay ninguna corriente de aire.

[9] En latín está la palabra específica *aut* para la o exclusiva.

Una base de conocimiento sencilla

Ahora que ya hemos definido la semántica de la lógica proposicional, podemos construir una base de conocimiento para el mundo de *wumpus*. Para simplificar, sólo trataremos con hechos y reglas acerca de hoyos; dejamos el tratamiento del *wumpus* como ejercicio. Vamos a proporcionar el conocimiento suficiente para llevar a cabo la inferencia que se trató en la Sección 7.3.

Primero de todo, necesitamos escoger nuestro vocabulario de símbolos proposicionales. Para cada *i, j*:

- Hacemos que $H_{i,j}$ sea verdadero si hay un hoyo en la casilla $[i, j]$.
- Hacemos que $B_{i,j}$ sea verdadero si hay una corriente de aire (una brisa) en la casilla [i, j].

La base de conocimiento contiene, cada una etiquetada con un identificador, las siguientes sentencias:

- No hay ningún hoyo en la casilla [1, 1].

$$R_1: \quad \neg H_{1,1}$$

- En una casilla se siente una brisa si y sólo si hay un hoyo en una casilla vecina. Esta regla se ha de especificar para cada casilla; por ahora, tan sólo incluimos las casillas que son relevantes:

$$R_2: \quad B_{1,1} \quad \Leftrightarrow \quad (H_{1,2} \vee H_{2,1})$$
$$R_3: \quad B_{2,1} \quad \Leftrightarrow \quad (H_{1,1} \vee H_{2,2} \vee H_{3,1})$$

- Las sentencias anteriores son verdaderas en todos los mundos de *wumpus*. Ahora incluimos las percepciones de brisa para las dos primeras casillas visitadas en el mundo concreto en donde se encuentra el agente, llegando a la situación que se muestra en la Figura 7.3(b).

$$R_4: \quad \neg B_{1,1}$$
$$R_5: \quad B_{2,1}$$

Entonces, la base de conocimiento está compuesta por las sentencias R_1 hasta R_5. La *BC* también se puede representar mediante una única sentencia (la conjunción $R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5$) porque dicha sentencia aserta que todas las sentencias son verdaderas.

Inferencia

Recordemos que el objetivo de la inferencia lógica es decidir si $BC \models \alpha$ para alguna sentencia α. Por ejemplo, si se deduce $H_{2,2}$. Nuestro primer algoritmo para la inferencia será una implementación directa del concepto de implicación: enumerar los modelos, y averiguar si α es verdadera en cada modelo en el que la *BC* es verdadera. En la lógica proposicional los modelos son asignaciones de los valores *verdadero* y *falso* sobre cada símbolo proposicional. Volviendo a nuestro ejemplo del mundo de *wumpus*, los símbolos proposicionales relevantes son $B_{1,1}, B_{2,1}, H_{1,1}, H_{1,2}, H_{2,1}, H_{2,2}$ y $H_{3,1}$. Con es-

tos siete símbolos, tenemos $2^7 = 128$ modelos posibles; y en tres de estos modelos, la *BC* es verdadera (Figura 7.9). En esos tres modelos $\neg H_{1,2}$ es verdadera, por lo tanto, no hay un hoyo en la casilla [1, 2]. Por el otro lado, $H_{2,2}$ es verdadera en dos de esos tres modelos y falsa en el tercero, entonces todavía no podemos decir si hay un hoyo en la casilla [2, 2].

La Figura 7.9 reproduce más detalladamente el razonamiento que se mostraba en la Figura 7.5. En la Figura 7.10 se muestra un algoritmo general para averiguar la implicación en lógica proposicional. De forma similar al algoritmo de BÚSQUEDA-CON-BACKTRACKING de la página 86, ¿IMPLICACIÓN-EN-TV? Realiza una enumeración recursiva de un espacio finito de asignaciones a variables. El algoritmo es **sólido** porque implemen-

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	BC
falso	*falso*	*falso*	*falso*	*falso*	*falso*	*falso*	*verdadero*	*verdadero*	*verdadero*	*verdadero*	*falso*	*falso*
falso	*falso*	*falso*	*falso*	*falso*	*falso*	*verdadero*	*verdadero*	*verdadero*	*falso*	*verdadero*	*falso*	*falso*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
falso	*verdadero*	*falso*	*falso*	*falso*	*falso*	*falso*	*verdadero*	*verdadero*	*falso*	*verdadero*	*verdadero*	*falso*
falso	*verdadero*	*falso*	*falso*	*falso*	*falso*	*verdadero*	*verdadero*	*verdadero*	*verdadero*	*verdadero*	*verdadero*	*verdadero*
falso	*verdadero*	*falso*	*falso*	*falso*	*verdadero*	*falso*	*verdadero*	*verdadero*	*verdadero*	*verdadero*	*verdadero*	*verdadero*
falso	*verdadero*	*falso*	*falso*	*falso*	*verdadero*	*verdadero*	*verdadero*	*verdadero*	*verdadero*	*verdadero*	*verdadero*	*verdadero*
falso	*verdadero*	*falso*	*falso*	*verdadero*	*falso*	*falso*	*verdadero*	*falso*	*falso*	*verdadero*	*verdadero*	*falso*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
verdadero	*verdadero*	*verdadero*	*verdadero*	*verdadero*	*verdadero*	*verdadero*	*falso*	*verdadero*	*verdadero*	*falso*	*verdadero*	*falso*

Figura 7.9 Una tabla de verdad construida para la base de conocimiento del ejemplo. La *BC* es verdadera si R_1 hasta R_5 son verdaderas, cosa que sucede en tres de las 128 filas. En estas tres filas, $H_{1,2}$ es falsa, así que no hay ningún hoyo en la casilla [1, 2]. Por otro lado, puede haber (o no) un hoyo en la casilla [2, 2].

```
función ¿IMPLICACIÓN-EN-TV?(BC, α) devuelve verdadero o falso
  entradas: BC, la base de conocimiento, una sentencia en lógica proposicional
            α, la sentencia implicada, una sentencia en lógica proposicional

  símbolos ← una lista de símbolos proposicionales de la BC y α
  devuelve COMPROBAR-TV(BC, α, símbolos, [ ])

función COMPROBAR-TV(BC, α, símbolos, modelo) devuelve verdadero o falso
  si ¿VACÍA?(símbolos) entonces
    si ¿VERDADERO-LP?(BC, modelo) entonces devuelve ¿VERDADERO-LP?(α, modelo)
    sino devuelve verdadero
  sino hacer
    P ← PRIMERO(símbolos); resto ← RESTO(símbolos)
    devuelve CHEQUEAR-TV(BC, α, resto, EXTENDER(P, verdadero, modelo)) y
             COMPROBAR-TV(BC, α, resto, EXTENDER(P, falso, modelo))
```

Figura 7.10 Un algoritmo de enumeración de una tabla de verdad para averiguar la implicación proposicional. TV viene de tabla de verdad. ¿VERDADERO-LP? Devuelve verdadero si una sentencia es verdadera en un modelo. La variable *modelo* representa un modelo parcial (una asignación realizada a un subconjunto de las variables). La llamada a la función EXTENDER(*P*, *verdadero*, *modelo*) devuelve un modelo parcial nuevo en el que *P* tiene el valor de verdad *verdadero*.

ta de forma directa la definición de implicación, y es **completo** porque trabaja para cualquier *BC* y sentencia α, y siempre finaliza (sólo hay un conjunto finito de modelos a ser examinados).

Por supuesto, que «conjunto finito» no siempre es lo mismo que «pequeño». Si la *BC* y α contienen en total n símbolos, entonces tenemos 2^n modelos posibles. Así, la complejidad temporal del algoritmo es $O(2^n)$. (La complejidad espacial sólo es $O(n)$ porque la enumeración es en primero en profundidad.) Más adelante, en este capítulo, veremos algoritmos que en la práctica son mucho más eficientes. Desafortunadamente, *cada algoritmo de inferencia que se conoce en lógica proposicional tiene un caso peor, cuya complejidad es exponencial respecto al tamaño de la entrada*. No esperamos mejorarlo, ya que demostrar la implicación en lógica proposicional es un problema co-NP-completo. (*Véase* Apéndice A.)

Equivalencia, validez y *satisfacibilidad*

Antes de que nos sumerjamos en los detalles de los algoritmos de inferencia lógica necesitaremos algunos conceptos adicionales relacionados con la implicación. Al igual que la implicación, estos conceptos se aplican a todos los tipos de lógica, sin embargo, se entienden más fácilmente para una en concreto, como es el caso de la lógica proposicional.

EQUIVALENCIA LÓGICA

El primer concepto es la **equivalencia lógica**: dos sentencias α y β son equivalentes lógicamente si tienen los mismos valores de verdad en el mismo conjunto de modelos. Este concepto lo representamos con $\alpha \Leftrightarrow \beta$. Por ejemplo, podemos observar fácilmente (mediante una tabla de verdad) que $P \wedge Q$ y $Q \wedge P$ son equivalentes lógicamente. En la Figura 7.11 se muestran otras equivalencias. Éstas juegan el mismo papel en la lógica que las igualdades en las matemáticas. Una definición alternativa de equivalencia es la siguiente: para dos sentencias α y β cualesquiera,

$$\alpha \equiv \beta \quad \text{si y sólo si} \quad \alpha \models \beta \text{ y } \beta \models \alpha$$

(Recuerde que $\models$ significa implicación.)

$$
\begin{aligned}
(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) && \text{Conmutatividad de } \wedge \\
(\alpha \vee \beta) &\equiv (\beta \vee \alpha) && \text{Conmutatividad de } \vee \\
((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) && \text{Asociatividad de } \wedge \\
((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) && \text{Asociatividad de } \vee \\
\neg(\neg\alpha) &\equiv \alpha && \text{Eliminación de la doble negación} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\beta \Rightarrow \neg\alpha) && \text{Contraposición} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\alpha \vee \beta) && \text{Eliminación de la implicación} \\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) && \text{Eliminación de la bicondicional} \\
\neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) && \text{Ley de Morgan} \\
\neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) && \text{Ley de Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) && \text{Distribución de } \wedge \text{ respecto a } \vee \\
(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) && \text{Distribución de } \vee \text{ respecto a } \wedge
\end{aligned}
$$

Figura 7.11 Equivalencias lógicas. Los símbolos α, β y γ se pueden sustituir por cualquier sentencia en lógica proposicional.

VALIDEZ

TAUTOLOGÍA

El segundo concepto que necesitaremos es el de **validez**. Una sentencia es válida si es verdadera en *todos* los modelos. Por ejemplo, la sentencia $P \vee \neg P$ es una sentencia válida. Las sentencias válidas también se conocen como **tautologías**, son *necesariamente* verdaderas y por lo tanto vacías de significado. Como la sentencia *Verdadero* es verdadera en todos los modelos, toda sentencia válida es lógicamente equivalente a *Verdadero*.

TEOREMA DE LA DEDUCCIÓN

¿Qué utilidad tienen las sentencias válidas? De nuestra definición de implicación podemos derivar el **teorema de la deducción**, que ya se conocía por los Griegos antiguos:

Para cualquier sentencia α y β, $\alpha \models \beta$ si y sólo si la sentencia $(\alpha \Rightarrow \beta)$ es válida.

(En el Ejercicio 7.4 se pide demostrar una serie de aserciones.) Podemos pensar en el algoritmo de inferencia de la Figura 7.10 como en un proceso para averiguar la validez de $(BC \Rightarrow \alpha)$. A la inversa, cada sentencia que es una implicación válida representa una inferencia correcta.

SATISFACIBILIDAD

El último concepto que necesitaremos es el de ***satisfacibilidad***. Una sentencia es satisfactoria si es verdadera para *algún* modelo. Por ejemplo, en la base de conocimiento ya mostrada, $(R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5)$ es *satisfacible* porque hay tres modelos en los que es verdadera, tal como se muestra en la Figura 7.9. Si una sentencia α es verdadera en un modelo m, entonces decimos que m **satisface** α, o que m **es un modelo de** α. La *satisfacibilidad* se puede averiguar enumerando los modelos posibles hasta que uno satisface la sentencia. La determinación de la *satisfacibilidad* de sentencias en lógica proposicional fue el primer problema que se demostró que era NP-completo.

SATISFACE

Muchos problemas en las ciencias de la computación son en realidad problemas de *satisfacibilidad*. Por ejemplo, todos los problemas de satisfacción de restricciones del Capítulo 5 se preguntan esencialmente si un conjunto de restricciones se satisfacen dada una asignación. Con algunas transformaciones adecuadas, los problemas de búsqueda también se pueden resolver mediante *satisfacibilidad*. La validez y la *satisfacible* están íntimamente relacionadas: α es válida si y sólo si $\neg\alpha$ es *insatisfacible*; en contraposición, α es *satisfacible* si y sólo si $\neg\alpha$ no es válida.

$\alpha \models \beta$ si y sólo si la sentencia $(\alpha \wedge \neg\beta)$ es insatisfactoria.

REDUCTIO AD ABSURDUM

REFUTACIÓN

La demostración de β a partir de α averiguando la insatisfacibilidad de $(\alpha \wedge \neg\beta)$ se corresponde exactamente con la técnica de demostración en matemáticas de la *reductio ad absurdum* (que literalmente se traduce como «reducción al absurdo»). Esta técnica también se denomina demostración mediante **refutación** o demostración por **contradicción**. Asumimos que la sentencia β es falsa y observamos si se llega a una contradicción con las premisas en α. Dicha contradicción es justamente lo que queremos expresar cuando decimos que la sentencia $(\alpha \wedge \neg\beta)$ es *insatisfacible*.

7.5 Patrones de razonamiento en lógica proposicional

Esta sección cubre los patrones estándar de inferencia que se pueden aplicar para derivar cadenas de conclusiones que nos llevan al objetivo deseado. Estos patrones de infe-

REGLAS DE INFERENCIA

MODUS PONENS

rencia se denominan **reglas de inferencia**. La regla más conocida es la llamada **Modus Ponens** que se escribe como sigue:

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$$

La notación nos dice que, cada vez que encontramos dos sentencias en la forma $\alpha \Rightarrow \beta$ y α, entonces la sentencia β puede ser inferida. Por ejemplo, si tenemos (*WumpusEnFrente* $\wedge$ *WumpusVivo*) $\Rightarrow$ *Disparar* y (*WumpusEnFrente* $\wedge$ *WumpusVivo*), entonces se puede inferir *Disparar*.

ELIMINACIÓN-$\wedge$

Otra regla de inferencia útil es la **Eliminación-$\wedge$**, que expresa que, de una conjunción se puede inferir cualquiera de sus conjuntores:

$$\frac{\alpha \wedge \beta}{\alpha}$$

Por ejemplo, de (*WumpusEnFrente* $\wedge$ *WumpusVivo*), se puede inferir *WumpusVivo*.

Teniendo en cuenta los posibles valores de verdad de α y β se puede observar fácilmente, de una sola vez, que el Modus Ponens y la Eliminación-$\wedge$ son reglas sólidas. Estas reglas se pueden utilizar sobre cualquier instancia en la que es aplicable, generando inferencias sólidas, sin la necesidad de enumerar todos los modelos.

Todas las equivalencias lógicas de la Figura 7.11 se pueden utilizar como reglas de inferencia. Por ejemplo, la equivalencia de la eliminación de la bicondicional nos lleva a las dos reglas de inferencia

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad \text{y} \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}$$

Pero no todas las reglas de inferencia se pueden usar, como ésta, en ambas direcciones. Por ejemplo, no podemos utilizar el Modus Ponens en la dirección opuesta para obtener $\alpha \Rightarrow \beta$ y α a partir de β.

Veamos cómo se pueden usar estas reglas de inferencia y equivalencias en el mundo de *wumpus*. Comenzamos con la base de conocimiento conteniendo R_1 a R_5, y mostramos cómo demostrar $\neg H_{1,2}$, es decir, que no hay un hoyo en la casilla [1, 2]. Primero aplicamos la eliminación de la bicondicional a R_2 para obtener

$$R_6: \quad (B_{1,1} \Rightarrow (H_{1,2} \vee H_{2,1})) \wedge ((H_{1,2} \vee H_{2,1}) \Rightarrow \mathrm{B}_{1,1})$$

Entonces aplicamos la Eliminación-$\wedge$ a R_6 para obtener

$$R_7: \quad ((H_{1,2} \vee H_{2,1}) \Rightarrow B_{1,1})$$

Y por la equivalencia lógica de contraposición obtenemos

$$R_8: \quad (\neg B_{1,1} \Rightarrow \neg(H_{1,2} \vee H_{2,1}))$$

Ahora aplicamos el Modus Ponens con R_8 y la percepción R_4 (por ejemplo, $\neg \mathrm{B}_{1,1}$), para obtener

$$R_9: \quad \neg(H_{1,2} \vee H_{2,1})$$

Finalmente, aplicamos la ley de Morgan, obteniendo la conclusión

$$R_{10}: \neg H_{1,2} \wedge \neg H_{2,1}$$

Es decir, ni la casilla [1, 2] ni la [2, 1] contienen un hoyo.

PRUEBA

A la derivación que hemos realizado (una secuencia de aplicaciones de reglas de inferencia) se le denomina una **prueba** (o **demostración**). Obtener una prueba es muy semejante a encontrar una solución en un problema de búsqueda. De hecho, si la función sucesor se define para generar todas las aplicaciones posibles de las reglas de inferencia, entonces todos los algoritmos de búsqueda del Capítulo 3 se pueden utilizar para obtener una prueba. De esta manera, la búsqueda de pruebas es una alternativa a tener que enumerar los modelos. La búsqueda se puede realizar hacia delante a partir de la base de conocimiento inicial, aplicando las reglas de inferencia para derivar la sentencia objetivo, o hacia atrás, desde la sentencia objetivo, intentando encontrar una cadena de reglas de inferencia que nos lleven a la base de conocimiento inicial. Más adelante, en esta sección, veremos dos familias de algoritmos que utilizan estas técnicas.

El hecho de que la inferencia en lógica proposicional sea un problema NP-completo nos hace pensar que, en el peor de los casos, la búsqueda de pruebas va a ser no mucho más eficiente que la enumeración de modelos. Sin embargo, en muchos casos prácticos, *encontrar una prueba puede ser altamente eficiente simplemente porque el proceso puede ignorar las proposiciones irrelevantes, sin importar cuántas de éstas haya*. Por ejemplo, la prueba que hemos visto que nos llevaba a $\neg H_{1,2} \wedge \neg H_{2,1}$ no utiliza las proposiciones $B_{2,1}$, $H_{1,1}$, $H_{2,2}$ o $H_{3,1}$. Estas proposiciones se pueden ignorar porque la proposición objetivo $H_{1,2}$ sólo aparece en la sentencia R_4, y la otra proposición de R_4 sólo aparece también en R_2; por lo tanto, R_1, R_3 y R_5 no juegan ningún papel en la prueba. Sucedería lo mismo aunque añadiésemos un millón de sentencias a la base de conocimiento; por el otro lado, el algoritmo de la tabla de verdad, aunque sencillo, quedaría saturado por la explosión exponencial de los modelos.

MONÓTONO

Esta propiedad de los sistemas lógicos en realidad proviene de una característica mucho más fundamental, denominada **monótono**. La característica de monotonismo nos dice que el conjunto de sentencias implicadas sólo puede *aumentar* (pero no cambiar) al añadirse información a la base de conocimiento[10]. Para cualquier sentencia α y β,

$$\text{si} \quad BC \models \alpha \quad \text{entonces} \quad BC \wedge \beta \models \alpha$$

Por ejemplo, supongamos que la base de conocimiento contiene una aserción adicional β, que nos dice que hay exactamente ocho hoyos en el escenario. Este conocimiento podría ayudar al agente a obtener conclusiones *adicionales*, pero no puede invalidar ninguna conclusión α ya inferida (como la conclusión de que no hay un hoyo en la casilla [1, 2]). El monotonismo permite que las reglas de inferencia se puedan aplicar siempre que se hallen premisas aplicables en la base de conocimiento; la conclusión de la regla debe permanecer *sin hacer caso de qué más hay en la base de conocimiento*.

[10] Las lógicas **No Monótonas**, que violan la propiedad de monotonismo, modelan una característica propia del razonamiento humano: cambiar de opinión. Estas lógicas se verán en la Sección 10.7.

Resolución

Hemos argumentado que las reglas de inferencia vistas hasta aquí son *sólidas*, pero no hemos visto la cuestión acerca de lo *completo* de los algoritmos de inferencia que las utilizan. Los algoritmos de búsqueda como el de búsqueda en profundidad iterativa (página 87) son completos en el sentido de que éstos encontrarán cualquier objetivo alcanzable, pero si las reglas de inferencia no son adecuadas, entonces el objetivo no es alcanzable; no existe una prueba que utilice sólo esas reglas de inferencia. Por ejemplo, si suprimimos la regla de eliminación de la bicondicional la prueba de la sección anterior no avanzaría. En esta sección se introduce una regla de inferencia sencilla, la **resolución**, que nos lleva a un algoritmo de inferencia completo cuando se empareja a un algoritmo de búsqueda completo.

Comenzaremos utilizando una versión sencilla de la resolución aplicada al mundo de *wumpus*. Consideremos los pasos que nos llevaban a la Figura 7.4(a): el agente vuelve de la casilla [2, 1] a la [1, 1] y entonces va a la casilla [1, 2], donde percibe un hedor, pero no percibe una corriente de aire. Ahora añadimos los siguientes hechos a la base de conocimiento:

$$R_{11}: \quad \neg B_{1,2}$$
$$R_{12}: \quad B_{1,2} \Leftrightarrow (H_{1,1} \vee H_{2,2} \vee H_{1,3})$$

Mediante el mismo proceso que nos llevó antes a R_{10}, podemos derivar que no hay ningún hoyo en la casilla [2, 2] o en la [1, 3] (recuerde que se sabe que en la casilla no había ninguna percepción de hoyos):

$$R_{13}: \quad \neg \mathrm{H}_{2,2}$$
$$R_{14}: \quad \neg \mathrm{H}_{1,3}$$

También podemos aplicar la eliminación de la bicondicional a la R_3, seguido del Modus Ponens con la R_5, para obtener el hecho de que puede haber un hoyo en la casilla [1, 1], la [2, 2] o la [3, 1]:

$$R_{15}: \quad \mathrm{H}_{1,1} \vee \mathrm{H}_{2,2} \vee \mathrm{H}_{3,1}$$

Ahora viene la primera aplicación de la regla de resolución: el literal $\neg \mathrm{H}_{2,2}$ de la R_{13} *se resuelve con* el literal $\mathrm{H}_{2,2}$ de la R_{15} , dando el *resolvente*

$$R_{16}: \quad \mathrm{H}_{1,1} \vee \mathrm{H}_{3,1}$$

En lenguaje natural: si hay un hoyo en la casilla [1, 1], o en la [2, 2], o en la [3, 1], y no hay ninguno en la [2, 2], entonces hay uno en la [1, 1] o en la [3, 1]. De forma parecida, el literal $\neg \mathrm{H}_{1,1}$ de la R_1 se resuelve con el literal $\mathrm{H}_{1,1}$ de la R_{16}, dando

$$R_{17}: \quad \mathrm{H}_{3,1}$$

En lenguaje natural: si hay un hoyo en la casilla [1, 1] o en la [3, 1], y no hay ninguno en la [1, 1], entonces hay uno en la [3, 1]. Los últimos dos pasos de inferencia son ejemplo de la regla de inferencia de **resolución unitaria**,

RESOLUCIÓN UNITARIA

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k}$$

en donde cada ℓ es un literal y ℓ_i y m son **literales complementarios** (por ejemplo, uno es la negación del otro). Así, la resolución unitaria toma una **cláusula** (una disyunción de literales) y un literal para producir una nueva cláusula. Fíjese en que un literal se puede ver como una disyunción con un solo literal, conocido como **cláusula unitaria**.

LITERALES COMPLEMENTARIOS

CLÁUSULA

CLÁUSULA UNITARIA

La regla de resolución unitaria se puede generalizar a la regla general de **resolución**,

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n}$$

donde ℓ_i y m_j son literales complementarios. Si sólo tratáramos con cláusulas de longitud dos, podríamos escribir la regla así

$$\frac{\ell_1 \vee \ell_2, \qquad \neg\ell_2 \vee \ell_3}{\ell_1 \vee \ell_3}$$

Es decir, la resolución toma dos cláusulas y genera una cláusula nueva con los literales de las dos cláusulas originales *menos* los literales complementarios. Por ejemplo, tendríamos

$$\frac{P_{1,1} \vee P_{3,1}, \qquad \neg P_{1,1} \vee \neg P_{2,2}}{P_{3,1} \vee \neg P_{2,2}}$$

Hay otro aspecto técnico relacionado con la regla de resolución: la cláusula resultante debería contener sólo una copia de cada literal[11]. Se le llama **factorización** al proceso de eliminar las copias múltiples de los literales. Por ejemplo, si resolvemos $(A \vee B)$ con $(A \vee \neg B)$ obtenemos $(A \vee A)$, que se reduce a A.

FACTORIZACIÓN

La *solidez* de la regla de resolución se puede ver fácilmente si consideramos el literal ℓ_i. Si ℓ_i es verdadero, entonces m_j es falso, y de aquí $m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n$ debe ser verdadero, porque se da $m_1 \vee \cdots \vee m_n$. Si ℓ_i es falso, entonces $\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k$ debe ser verdadero, porque se da $\ell_1 \vee \cdots \vee \ell_k$. Entonces l_i es o bien verdadero o bien falso, y así, se obtiene una de las dos conclusiones, exactamente tal cómo establece la regla de resolución.

Lo que es más sorprendente de la regla de resolución es que crea la base para una familia de procedimientos de inferencia *completos*. *Cualquier algoritmo de búsqueda completo, aplicando sólo la regla de resolución, puede derivar cualquier conclusión implicada por cualquier base de conocimiento en lógica proposicional*. Pero hay una advertencia: la resolución es completa en un sentido muy especializado. Dado que A sea

[11] Si una cláusula se ve como un conjunto de literales, entonces esta restricción se respeta de forma automática. Utilizar la notación de conjuntos para representar cláusulas hace que la regla de resolución sea más clara, con el coste de introducir una notación adicional.

verdadero, no podemos utilizar la resolución para generar de forma automática la consecuencia $A \vee B$. Sin embargo, podemos utilizar la resolución para responder a la pregunta de si $A \vee B$ es verdadero. Este hecho se denomina **completitud de la resolución**, que indica que la resolución se puede utilizar siempre para confirmar o refutar una sentencia, pero no se puede usar para enumerar sentencias verdaderas. En las dos siguientes secciones explicamos cómo la resolución lleva a cabo este proceso.

COMPLETITUD DE LA RESOLUCIÓN

Forma normal conjuntiva

La regla de resolución sólo se puede aplicar a disyunciones de literales, por lo tanto, sería muy importante que la base de conocimiento y las preguntas a ésta estén formadas por disyunciones. Entonces, ¿cómo nos lleva esto a un procedimiento de inferencia completa para toda la lógica proposicional? La respuesta es que *toda sentencia en lógica proposicional es equivalente lógicamente a una conjunción de disyunciones de literales*. Una sentencia representada mediante una conjunción de disyunciones de literales se dice que está en **forma normal conjuntiva** o **FNC**. Que lo consideraremos bastante útil más tarde, al tratar la reducida familia de sentencias ***k*-FNC**. Una sentencia k-FNC tiene exactamente k literales por cláusula:

FORMA NORMAL CONJUNTIVA

K-FNC

$$(\ell_{1,1} \vee \ldots \vee \ell_{1,k}) \wedge \ldots \wedge (\ell_{n,1} \vee \ldots \vee \ell_{n,k})$$

De manera que se puede transformar cada sentencia en una sentencia de tipo 3-FNC, la cual tiene un conjunto de modelos equivalente.

Mejor que demostrar estas afirmaciones (*véase* el Ejercicio 7.10), vamos a describir un procedimiento de conversión muy sencillo. Vamos a ilustrar el procedimiento con la conversión de R_2, la sentencia $B_{1,1} \Leftrightarrow (H_{1,2} \vee H_{2,1})$, a FNC. Los pasos a seguir son los siguientes:

1. Eliminar $\Leftrightarrow$, sustituyendo $\alpha \Leftrightarrow \beta$ por $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

$$(B_{1,1} \Rightarrow (H_{1,2} \vee H_{2,1})) \wedge ((H_{1,2} \vee H_{2,1}) \Rightarrow B_{1,1})$$

2. Eliminar $\Rightarrow$, sustituyendo $\alpha \Rightarrow \beta$ por $\neg\alpha \vee \beta$.

$$(\neg B_{1,1} \vee H_{1,2} \vee H_{2,1}) \wedge (\neg(H_{1,2} \vee H_{2,1}) \vee B_{1,1})$$

3. Una FNC requiere que la $\neg$ se aplique sólo a los literales, por lo tanto, debemos «anidar las $\neg$» mediante la aplicación reiterada de las siguientes equivalencias (sacadas de la Figura 7.11).

$$\neg(\neg\alpha) \equiv \alpha \text{ (eliminación de la doble negación)}$$
$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \text{ (de Morgan)}$$
$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) \text{ (de Morgan)}$$

En el ejemplo, sólo necesitamos una aplicación de la última regla:

$$(\neg B_{1,1} \vee H_{1,2} \vee H_{2,1}) \wedge ((\neg H_{1,2} \wedge \neg H_{2,1}) \vee B_{1,1})$$

4. Ahora tenemos una sentencia que tiene una $\wedge$ con operadores de $\vee$ anidados, aplicados a literales y a una $\wedge$ anidada. Aplicamos la ley de distributividad de la Figura 7.11, distribuyendo la $\vee$ sobre la $\wedge$ cuando nos es posible.

$$(\neg B_{1,1} \vee H_{1,2} \vee H_{2,1}) \wedge (\neg H_{1,2} \vee B_{1,1}) \wedge (\neg H_{2,1} \vee \mathrm{B}_{1,1})$$

La sentencia inicial ahora está en FNC, una conjunción con tres cláusulas. Es más difícil de leer pero se puede utilizar como entrada en el procedimiento de resolución.

Un algoritmo de resolución

Los procedimientos de inferencia basados en la resolución trabajan utilizando el principio de prueba mediante contradicción que vimos al final de la Sección 7.4. Es decir, para demostrar que $BC \models \alpha$, demostramos que $(BC \wedge \neg\alpha)$ es *insatisfacible*. Lo hacemos demostrando una contradicción.

En la Figura 7.12 se muestra un algoritmo de resolución. Primero se convierte $(BC \wedge \neg\alpha)$ a FNC. Entonces, se aplica la regla de resolución a las cláusulas obtenidas. Cada par que contiene literales complementarios se resuelve para generar una nueva cláusula, que se añade al conjunto de cláusulas si no estaba ya presente. El proceso continúa hasta que sucede una de estas dos cosas:

- No hay nuevas cláusulas que se puedan añadir, en cuyo caso α no implica β , o
- Se deriva la cláusula vacía de una aplicación de la regla de resolución, en cuyo caso α implica β.

La cláusula vacía (una disyunción sin disyuntores) es equivalente a *Falso* porque una disyunción es verdadera sólo si al menos uno de sus disyuntores es verdadero. Otra forma de ver que la cláusula vacía representa una contradicción es observar que se presenta sólo si se resuelven dos cláusulas unitarias complementarias, tales como P y ¬P.

```
función RESOLUCIÓN-LP(BC, α) devuelve verdadero o falso
  entradas: BC, la base de conocimiento, una sentencia en lógica proposicional
            α, la petición, una sentencia en lógica proposicional

  cláusulas ← el conjunto de cláusulas de BC ∧ ¬α en representación FNC
  nueva ← { }
  bucle hacer
      para cada C_i, C_j en cláusulas hacer
          resolventes ← RESUELVE-LP(C_i, C_j)
          si resolventes contiene la cláusula vacía entonces devolver verdadero
          nueva ← nueva ∪ resolventes
      si nueva ⊆ cláusulas entonces devolver falso
      cláusulas ← cláusulas ∪ nueva
```

Figura 7.12 Un algoritmo sencillo de resolución para la lógica proposicional. La función RESUELVE-LP devuelve el conjunto de todas las cláusulas posibles que se obtienen de resolver las dos entradas.

Ahora podemos aplicar el procedimiento de resolución a una inferencia sencilla del mundo de *wumpus*. Cuando el agente está en la casilla [1, 1] no percibe ninguna brisa, por lo tanto no puede haber hoyos en las casillas vecinas. Las sentencias relevantes en la base de conocimiento son

$$BC = R_2 \wedge R_4 = (B_{1,1} \Leftrightarrow (H_{1,2} \vee H_{2,1})) \wedge \neg B_{1,1}$$

y deseamos demostrar α, es decir $\neg H_{1,2}$. Cuando convertimos $(BC \wedge \neg\alpha)$ a FNC obtenemos las cláusulas que se muestran en la fila superior de la Figura 7.13. La segunda fila en la figura muestra todas las cláusulas obtenidas resolviendo parejas de la primera fila. Entonces, cuando $H_{1,2}$ se resuelve con $\neg H_{1,2}$ obtenemos la cláusula vacía, representada mediante un cuadrado pequeño. Una revisión de la Figura 7.13 nos revela que muchos pasos de resolución no nos sirven de nada. Por ejemplo, la cláusula $B_{1,1} \vee \neg B_{1,1} \vee H_{1,2}$ es equivalente a *Verdadero* $\vee H_{1,2}$, que es también equivalente a *Verdadero*. Deducir que *Verdadero* es verdadero no nos es muy útil. Por lo tanto, se puede descartar cualquier cláusula que contenga dos literales complementarios.

Completitud de la resolución

Para concluir con nuestro debate acerca de la resolución, ahora vamos a demostrar por qué es completo el procedimiento RESOLUCIÓN-LP. Para hacerlo nos vendrá bien introducir el concepto de **cierre de la resolución** $CR(S)$ del conjunto de cláusulas S, que es el conjunto de todas las cláusulas derivables, obtenidas mediante la aplicación repetida de la regla de resolución a las cláusulas de S o a las derivadas de éstas. El cierre de la resolución es lo que calcula el procedimiento RESOLUCIÓN-LP y asigna como valor final a la variable *cláusulas*. Es fácil ver que $CR(S)$ debe ser finito, porque sólo hay un conjunto finito de las diferentes cláusulas que se pueden generar a partir del conjunto de símbolos $P_1 \ldots, P_k$ que aparecen en S, (fíjese que esto no sería cierto si no aplicáramos el procedimiento de factorización, que elimina las copias múltiples de un literal). Por eso, el procedimiento RESOLUCIÓN-LP siempre termina.

CIERRE DE LA RESOLUCIÓN

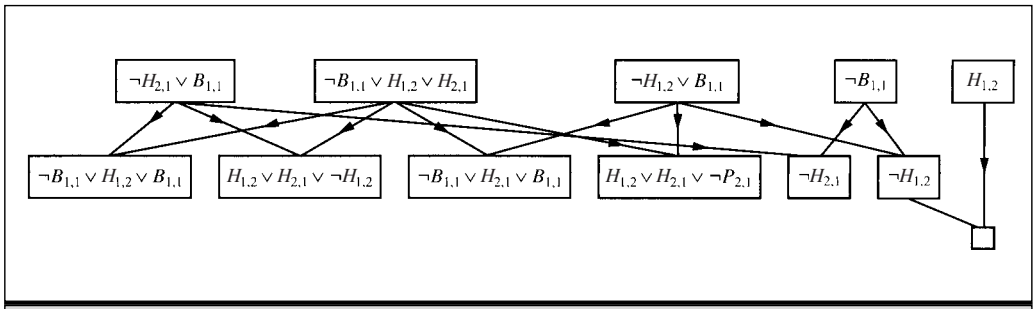


Figura 7.13 Aplicación parcial de RESOLUCIÓN-LP a una inferencia sencilla en el mundo de wumpus. Se observa que $\neg H_{1,2}$ se sigue de las cláusulas 3.ª y 4.ª de la fila superior.

TEOREMA FUNDAMENTAL DE LA RESOLUCIÓN

El teorema de la completitud para la resolución en lógica proposicional se denomina **teorema fundamental de la resolución**:

> Si un conjunto de cláusulas es *insatisfacible*, entonces el cierre de la resolución de esas cláusulas contiene la cláusula vacía.

Vamos a probar este teorema demostrando su contraposición: si el cierre $CR(S)$ *no* contiene la cláusula vacía, entonces S es *satisfacible*. De hecho, podemos construir un modelo de S con los valores de verdad adecuados para $P_1, \ldots, P_k$. El procedimiento de construcción es como sigue:

Para i de 1 a k,

- Si hay una cláusula en $CR(S)$ que contenga el literal $\neg P_i$, tal que todos los demás literales de la cláusula sean falsos bajo la asignación escogida para $P_1, \ldots, P_{i-1}$, entonces asignar a P_i el valor de *falso*.
- En otro caso, asignar a P_i el valor de *verdadero*.

Queda por demostrar que esta asignación a $P_1, \ldots, P_k$ es un modelo de S, a condición de que $CR(S)$ se cierre bajo la resolución y no contenga la cláusula vacía. Esta demostración se deja como ejercicio.

Encadenamiento hacia delante y hacia atrás

La completitud de la resolución hace que ésta sea un método de inferencia muy importante. Sin embargo, en muchos casos prácticos no se necesita todo el poder de la resolución. Las bases de conocimiento en el mundo real a menudo contienen sólo cláusulas, de un tipo restringido, denominadas **cláusulas de Horn**. Una cláusula de Horn es una disyunción de literales de los cuales, *como mucho uno es positivo*. Por ejemplo, la cláusula $(\neg L_{1,1} \vee \neg Brisa \vee B_{1,1})$, en donde $L_{1,1}$ representa que el agente está en la casilla [1, 1], es una cláusula de Horn, mientras que la cláusula $(\neg B_{1,1} \vee H_{1,2} \vee H_{2,1})$ no lo es.

CLÁUSULAS DE HORN

La restricción de que haya sólo un literal positivo puede parecer algo arbitraria y sin interés, pero realmente es muy importante, debido a tres razones:

1. Cada cláusula de Horn se puede escribir como una implicación cuya premisa sea una conjunción de literales positivos y cuya conclusión sea un único literal positivo. (*Véase* el Ejercicio 7.12.) Por ejemplo, la cláusula de Horn $(\neg L_{1,1} \vee \neg Brisa \vee B_{1,1})$ se puede rescribir como la implicación $(L_{1,1} \wedge Brisa) \Rightarrow B_{1,1}$. La sentencia es más fácil de leer en la última representación: ésta dice que si el agente está en la casilla [1, 1] y percibe una brisa, entonces la casilla [1, 1] tiene una corriente de aire. La gente encuentra más fácil esta forma de leer y escribir sentencias para muchos dominios del conocimiento.

   CLÁUSULAS POSITIVAS

   CABEZA

   CUERPO

   HECHO

   Las cláusulas de Horn como ésta, con *exactamente* un literal positivo, se denominan **cláusulas positivas**. El literal positivo se denomina **cabeza**, y la disyunción de literales negativos **cuerpo** de la cláusula. Una cláusula positiva que no tiene literales negativos simplemente aserta una proposición dada, que algunas veces se le denomina **hecho**. Las cláusulas positivas forman la base de la

programación lógica, que se verá en el Capítulo 9. Una cláusula de Horn *sin* literales positivos se puede escribir como una implicación cuya conclusión es el literal *Falso*. Por ejemplo, la cláusula $(\neg W_{1,1} \vee \neg W_{1,2})$ (el *wumpus* no puede estar en la casilla [1, 1] y la [1, 2] a la vez) es equivalente a $W_{1,1} \wedge W_{1,2} \Rightarrow$ *Falso*. A este tipo de sentencias se las llama **restricciones de integridad** en el mundo de las bases de datos, donde se utilizan para indicar errores entre los datos. En los algoritmos siguientes asumimos, para simplificar, que la base de conocimiento sólo contiene cláusulas positivas y que no dispone de restricciones de integridad. Entonces decimos que estas bases de conocimiento están en forma de Horn.

RESTRICCIONES DE INTEGRIDAD

2. La inferencia con cláusulas de Horn se puede realizar mediante los algoritmos de **encadenamiento hacia delante** y de **encadenamiento hacia atrás**, que en breve explicaremos. Ambos algoritmos son muy naturales, en el sentido de que los pasos de inferencia son obvios y fáciles de seguir por las personas.

ENCADENAMIENTO HACIA DELANTE

ENCADENAMIENTO HACIA ATRÁS

3. Averiguar si hay o no implicación con las cláusulas de Horn se puede realizar en un tiempo que es *lineal* respecto al tamaño de la base de conocimiento.

Este último hecho es una grata sorpresa. Esto significa que la inferencia lógica es un proceso barato para muchas bases de conocimiento en lógica proposicional que se encuentran en el mundo real.

El algoritmo de encadenamiento hacia delante ¿IMPLICACIÓN-EHD-LP?(*BC*, *q*) determina si un símbolo proposicional *q* (la petición) se deduce de una base de conocimiento compuesta por cláusulas de Horn. El algoritmo comienza a partir de los hechos conocidos (literales positivos) de la base de conocimiento. Si todas las premisas de una implicación se conocen, entonces la conclusión se añade al conjunto de hechos conocidos. Por ejemplo, si $L_{1,1}$ y *Brisa* se conocen y $(L_{1,1} \wedge \mathit{Brisa}) \Rightarrow B_{1,1}$ está en la base de conocimiento, entonces se puede añadir $B_{1,1}$ a ésta. Este proceso continúa hasta que la petición *q* es añadida o hasta que no se pueden realizar más inferencias. En la Figura 7.14 se muestra el algoritmo detallado. El principal punto a recordar es que el algoritmo se ejecuta en tiempo lineal.

La mejor manera de entender el algoritmo es mediante un ejemplo y un diagrama. La Figura 7.15(a) muestra una base de conocimiento sencilla con cláusulas de Horn, en donde *A* y *B* se conocen como hechos. La Figura 7.15(b) muestra la misma base de conocimiento representada mediante un **grafo Y-O**. En los grafos Y-O múltiples enlaces se juntan mediante un arco para indicar una disyunción (cualquier enlace se puede probar). Es fácil ver cómo el encadenamiento hacia delante trabaja sobre el grafo. Se seleccionan los hechos conocidos (aquí *A* y *B*) y la inferencia se propaga hacia arriba tanto como se pueda. Siempre que aparece una conjunción, la propagación se para hasta que todos los conjuntores sean conocidos para seguir a continuación. Se anima al lector a que desarrolle el proceso en detalle a partir del ejemplo.

GRAFO Y-O

Es fácil descubrir que el encadenamiento hacia delante es un proceso **sólido**: cada inferencia es esencialmente una aplicación del Modus Ponens. El encadenamiento hacia delante también es **completo**: cada sentencia atómica implicada será derivada. La forma más fácil de verlo es considerando el estado final de la tabla *inferido* (después de que el algoritmo encuentra un **punto fijo** a partir del cual no es posible realizar nuevas inferencias).

PUNTO FIJO

```
función ¿IMPLICACIÓN-EHD-LP?(BC, q) devuelve verdadero o falso
  entradas: BC, la base de conocimiento, un conjunto de cláusulas de Horn en Lógica Proposicional
            q, la petición, un símbolo proposicional
  variables locales: cuenta, una tabla ordenada por cláusula, inicializada al número de cláusulas
                     inferido, una tabla, ordenada por símbolo, cada entrada inicializada a falso
                     agenda, una lista de símbolos, inicializada con los símbolos de la BC que se
                     sabe que son verdaderos

  mientras agenda no esté vacía hacer
    p ← POP(agenda)
    a menos que inferido[p] hacer
        inferido[p] ← verdadero
        para cada cláusula de Horn c en la que aparezca la premisa p hacer
            reducir cuenta[c]
            si cuenta[c] = 0 entoncer hacer
                si CABEZA[c] = q entonces devolver verdadero
                PUSH(CABEZA[c], agenda)
  devolver falso
```

Figura 7.14 El algoritmo de encadenamiento hacia delante para la lógica proposicional. La variable *agenda* almacena la pista de los símbolos que se saben son verdaderos pero no han sido «procesados» todavía. La tabla *cuenta* guarda la pista de las premisas de cada implicación que aún son desconocidas. Siempre que se procesa un símbolo p de la agenda la cuenta se reduce en uno para cada implicación en la que aparece la premisa p. (Este proceso se puede realizar en un tiempo constante si la *BC* se ordena de forma adecuada.) Si la cuenta llega a cero, todas las premisas de la implicación son conocidas, y por tanto, la conclusión se puede añadir a la agenda. Por último, necesitamos guardar la pista de qué símbolos han sido procesados; no se necesita añadir un símbolo inferido si ha sido procesado previamente. Este proceso nos evita un trabajo redundante, y también nos prevé de los bucles infinitos que podrían causarse por implicaciones tales como $P \Rightarrow Q$ y $Q \Rightarrow P$.

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

$B \wedge L \Rightarrow M$

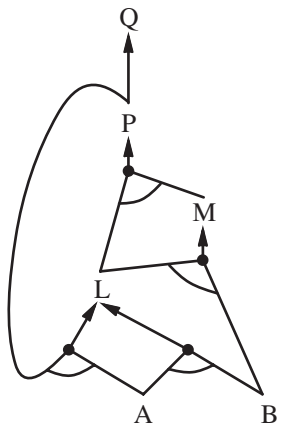
$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$

A

B

(a)



(b)

Figura 7.15 (a) Una base de conocimiento sencilla con cláusulas de Horn. (b) Su correspondiente grafo Y-O.

La tabla contiene el valor *verdadero* para cada símbolo inferido en el proceso, y el valor *falso* para los demás símbolos. Podemos interpretar la tabla como un modelo lógico, más aún, *cada cláusula positiva de la BC original es verdadera en este modelo*. Para ver esto asumamos lo opuesto, en concreto, que alguna cláusula $a_1 \wedge \ldots \wedge a_k \Rightarrow b$ sea falsa en el modelo. Entonces $a_1 \wedge \ldots \wedge a_k$ debe ser verdadero en el modelo y b debe ser falso. ¡Pero esto contradice nuestra asunción de que el algoritmo ha encontrado un punto fijo! Por lo tanto, podemos concluir que el conjunto de sentencias atómicas inferidas hasta el punto fijo define un modelo de la *BC* original. Además, cualquier sentencia atómica q que se implica de la *BC* debe ser cierta en todos los modelos y en este modelo en particular. Por lo tanto, cada sentencia implicada q debe ser inferida por el algoritmo.

DIRIGIDO POR LOS DATOS

El encadenamiento hacia delante es un ejemplo del concepto general de razonamiento **dirigido por los datos**, es decir, un razonamiento en el que el foco de atención parte de los datos conocidos. Este razonamiento se puede utilizar en un agente para derivar conclusiones a partir de percepciones recibidas, a menudo, sin la necesidad de una petición concreta. Por ejemplo, el agente de *wumpus* podría DECIR sus percepciones a la base de conocimiento utilizando un algoritmo de encadenamiento hacia delante de tipo incremental, en el que los hechos se pueden añadir a la agenda para iniciar nuevas inferencias. A las personas, a medida que les llega nueva información, se les activa una gran cantidad de razonamiento dirigido por los datos. Por ejemplo, si estoy en casa y oigo que comienza a llover, podría sucederme que la merienda quede cancelada. Con todo esto, no será muy probable que el pétalo diecisieteavo de la rosa más alta del jardín de mi vecino se haya mojado. Las personas llevan a cabo un encadenamiento hacia delante con un control cuidadoso, a fin de no hundirse en consecuencias irrelevantes.

El algoritmo de encadenamiento hacia atrás, tal como sugiere su nombre, trabaja hacia atrás a partir de la petición. Si se sabe que la petición q es verdadera, entonces no se requiere realizar ningún trabajo. En el otro caso, el algoritmo encuentra aquellas implicaciones de la base de conocimiento de las que se concluye q. Si se puede probar que todas las premisas de una de esas implicaciones son verdaderas (mediante en encadenamiento hacia atrás), entonces q es verdadera. Cuando se aplica a la petición Q de la Figura 7.15, el algoritmo retrocede hacia abajo por el grafo hasta que encuentra un conjunto de hechos conocidos que forma la base de la demostración. El algoritmo detallado se deja como ejercicio. Al igual que en el encadenamiento hacia delante, una implementación eficiente se ejecuta en tiempo lineal.

RAZONAMIENTO DIRIGIDO POR EL OBJETIVO

El encadenamiento hacia atrás es un tipo de **razonamiento dirigido por el objetivo**. Este tipo de razonamiento es útil para responder a peticiones tales como «¿Qué debo hacer ahora?» y «¿Dónde están mis llaves?» A menudo, el coste del encadenamiento hacia atrás es *mucho menor* que el orden lineal respecto al tamaño de la base de conocimiento, porque el proceso sólo trabaja con los hechos relevantes. Por lo general, un agente debería repartir su trabajo entre el razonamiento hacia delante y el razonamiento hacia atrás, limitando el razonamiento hacia delante a la generación de los hechos que sea probable que sean relevantes para las peticiones, y éstas se resolverán mediante el encadenamiento hacia atrás.

7.6 Inferencia proposicional efectiva

En esta sección vamos a describir dos familias de algoritmos eficientes para la inferencia en lógica proposicional, basadas en la comprobación de modelos: un enfoque basado en la búsqueda con *backtracking*, y el otro en la búsqueda basada en la escalada de colina. Estos algoritmos forman parte de la «tecnología» de la lógica proposicional. Esta sección se puede tan sólo ojear en una primera lectura del capítulo.

Los algoritmos que vamos a describir se utilizan para la comprobación de la *satisfacibilidad*. Ya hemos mencionado la conexión entre encontrar un modelo satisfacible para una sentencia lógica y encontrar una solución para un problema de satisfacción de restricciones, entonces, quizá no sorprenda que las dos familias de algoritmos se asemejen bastante a los algoritmos con *backtracking* de la Sección 5.2 y a los algoritmos de búsqueda local de la Sección 5.3. Sin embargo, éstos son extremadamente importantes por su propio derecho, porque muchos problemas combinatorios en las ciencias de la computación se pueden reducir a la comprobación de la *satisfacibilidad* de una sentencia proposicional. Cualquier mejora en los algoritmos de *satisfacibilidad* presenta enormes consecuencias para nuestra habilidad de manejar la complejidad en general.

Un algoritmo completo con *backtracking* («vuelta atrás»)

ALGORITMO DE DAVIS Y PUTNAM

El primer algoritmo que vamos a tratar se le llama a menudo **algoritmo de Davis y Putnam**, después del artículo de gran influencia que escribieron Martin Davis y Hilary Putnam (1960). De hecho, el algoritmo es la versión descrita por Davis, Logemann y Loveland (1962), así que lo llamaremos DPLL, las iniciales de los cuatro autores. DPLL toma como entrada una sentencia en forma normal conjuntiva (un conjunto de cláusulas). Del mismo modo que la BÚSQUEDA-CON-BACKTRACKING e ¿IMPLICACIÓN-TV?, DPLL realiza una enumeración esencialmente recursiva, mediante el primero en profundidad, de los posibles modelos. El algoritmo incorpora tres mejoras al sencillo esquema del ¿IMPLICACIÓN-TV?

- *Terminación anticipada*: el algoritmo detecta si la sentencia debe ser verdadera o falsa, aun con un modelo completado parcialmente. Una cláusula es verdadera si *cualquier* literal es verdadero, aun si los otros literales todavía no tienen valores de verdad; así la sentencia, entendida como un todo, puede evaluarse como verdadera aun antes de que el modelo sea completado. Por ejemplo, la sentencia $(A \vee B) \wedge (A \vee C)$ es verdadera si A lo es, sin tener en cuenta los valores de B y C. De forma similar, una sentencia es falsa si *cualquier* cláusula lo es, caso que ocurre cuando cada uno de sus literales lo son. Del mismo modo, esto puede ocurrir mucho antes de que el modelo sea completado. La terminación anticipada evita la evaluación íntegra de los subárboles en el espacio de búsqueda.

SÍMBOLO PURO

- *Heurística de símbolo puro*: un **símbolo puro** es un símbolo que aparece siempre con el mismo «signo» en todas las cláusulas. Por ejemplo, en las tres cláusulas $(A \vee \neg B)$, $(\neg B \vee \neg C)$, y $(C \vee A)$, el símbolo A es puro, porque sólo aparece el literal positivo, B también es puro porque sólo aparece el literal negativo, y C es un sím-

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-

tringiendo las variables. Por ejemplo, aquí tenemos una sentencia en FNC-3 con cinco símbolos, y cinco cláusulas generadas aleatoriamente[12]:

$$(\neg D \vee \neg B \vee C) \wedge (D \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \vee E) \wedge (E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C)$$

16 de las 32 posibles asignaciones son modelos de esta sentencia, por lo tanto, de media, sólo se requerirían dos pasos para encontrar un modelo.

Entonces, ¿dónde se encuentran los problemas duros? Presumiblemente, si *incrementamos* el número de cláusulas, manteniendo fijo el número de símbolos, hacemos que el problema esté más restringido, y que sea más difícil encontrar las soluciones. Permitamos que m sea el número de cláusulas y n el número de símbolos. La Figura 7.18(a) muestra la probabilidad de que una sentencia en FNC-3 sea *satisfacible*, como una función de la relación cláusula/símbolo, m/n, con n fijado a 50. Tal como esperábamos, para una m/n pequeña, la probabilidad se acerca a 1, y para una m/n grande, la probabilidad se acerca a 0. La probabilidad cae de forma bastante brusca alrededor del valor de $m/n = 4{,}3$. Las sentencias en FNC que están cerca de este **punto crítico** se podrían definir como «casi *satisfacibles*» o «casi *insatisfacibles*». ¿Es en este punto donde encontramos los problemas duros?

PUNTO CRÍTICO

La Figura 7.18(b) muestra los tiempos de ejecución de los algoritmos DPLL y SAT-CAMINAR alrededor de este punto crítico, en donde hemos restringido nuestra atención sólo a los problemas *satisfacibles*. Tres cosas están claras: primero, los problemas que están cerca del punto crítico son *mucho* más difíciles que los otros problemas aleatorios. Segundo, aun con los problemas más duros, el DPLL es bastante efectivo (una media de unos pocos miles de pasos comparados con $2^{50} \approx 10^{15}$ en la enumeración de tablas de verdad). Tercero, SAT-CAMINAR es mucho más rápido que el DPLL en todo el rango.

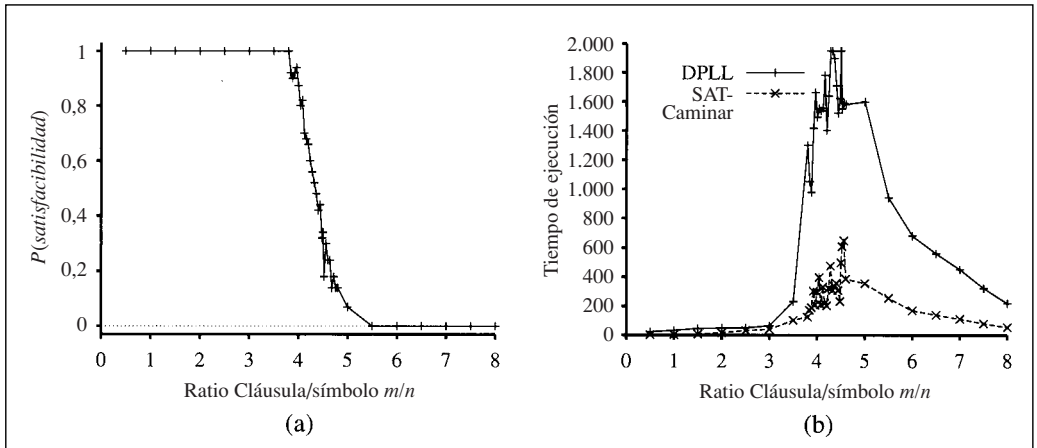


Figura 7.18 (a) Gráfico que muestra la probabilidad de que una sentencia en FNC-3 con $n = 50$ símbolos sea *satisfacible*, en función del ratio cláusula/símbolo m/n. (b) Gráfico del tiempo de ejecución promedio del DPLL y del SAT-CAMINAR sobre 100 sentencias en FNC-3 aleatorias con $n = 50$, para un rango reducido de valores de m/n alrededor del punto crítico.

[12] Cada cláusula contiene tres símbolos *diferentes* seleccionados aleatoriamente, cada uno de ellos negado con una probabilidad del 50%.

Por supuesto, estos resultados sólo son para los problemas generados aleatoriamente. Los problemas reales no tienen necesariamente la misma estructura (en términos de proporciones entre literales positivos y negativos, densidades de conexiones entre cláusulas, etcétera) que los problemas aleatorios. Pero todavía en la práctica, el SAT-CAMINAR y otros algoritmos de la misma familia son muy buenos para resolver problemas reales (a menudo, tan buenos como el mejor algoritmo de propósito específico para esas tareas). Problemas con miles de símbolos y millones de cláusulas se tratan de forma rutinaria con resolutores como el CHAFF. Estas observaciones nos sugieren que alguna combinación de los comportamientos de la heurística de min-conflictos y de pasada-aleatoria nos proporciona una gran capacidad de *propósito-general* para resolver muchas situaciones en las que se requiere el razonamiento combinatorio.

7.7 Agentes basados en lógica proposicional

En esta sección, vamos a juntar lo que hemos aprendido hasta ahora para construir agentes que funcionan utilizando la lógica proposicional. Veremos dos tipos de agentes: aquellos que utilizan algoritmos de inferencia y una base de conocimiento, como el agente basado en conocimiento genérico de la Figura 7.1, y aquellos que evalúan directamente expresiones lógicas en forma de circuitos. Aplicaremos ambos tipos de agentes en el mundo de *wumpus*, y encontraremos que ambos sufren de serias desventajas.

Encontrar hoyos y *wumpus* utilizando la inferencia lógica

Permítanos empezar con un agente que razona mediante la lógica acerca de las localizaciones de los hoyos, de los *wumpus* y de la seguridad de las casillas. El agente comienza con una base de conocimiento que describe la «física» del mundo de *wumpus*. El agente sabe que la casilla [1, 1] no tiene ningún hoyo ni ningún *wumpus*; es decir, $\neg H_{1,1}$ y $\neg W_{1,1}$. El agente también conoce una sentencia que indica cómo se percibe una brisa en una casilla $[x, y]$:

$$B_{x,y} \Leftrightarrow (H_{x,y+1} \vee H_{x,y-1} \vee H_{x+1,y} \vee H_{x-1,y}) \tag{7.1}$$

El agente conoce una sentencia que indica cómo se percibe el hedor en una casilla $[x, y]$:

$$M_{x,y} \Leftrightarrow (W_{x,y+1} \vee W_{x,y-1} \vee W_{x+1,y} \vee W_{x-1,y}) \tag{7.2}$$

Finalmente, el agente sabe que sólo hay un *wumpus*. Esto se expresa de dos maneras. En la primera, debemos definir que hay *por lo menos* un *wumpus*:

$$W_{1,1} \vee W_{1,2} \vee \ldots \vee W_{4,3} \vee W_{4,4}$$

Entonces, debemos definir que *como mucho* hay un *wumpus*. Una manera de definirlo es diciendo que para dos casillas cualesquiera, una de ellas debe estar libre de un *wum-*

pus. Con n casillas, tenemos $n(n-1)/2$ sentencias del tipo $\neg W_{1,1} \vee \neg W_{1,2}$. Entonces, para un mundo de 4×4, comenzamos con un total de 155 sentencias conteniendo 64 símbolos diferentes.

El programa del agente, que se muestra en la Figura 7.19 DICE a su base de conocimiento cualquier nueva percepción acerca de una brisa o un mal hedor. (También actualiza algunas variables del programa para guardar la pista de dónde se encuentra y que casillas ha visitado. Estos últimos datos los necesitará más adelante.) Entonces el programa escoge dónde observar antes de entre las casillas que rodean al agente, es decir, las casillas adyacentes a aquellas ya visitadas. Una casilla $[i, j]$ que rodea al agente es *probable que sea segura* si la sentencia $(\neg H_{i,j} \wedge \neg W_{i,j})$ se deduce de la base de conocimiento. La siguiente alternativa mejor es una casilla *posiblemente segura*, o sea, aquella casilla para la cual el agente no puede demostrar si hay un hoyo o un *wumpus*, es decir, aquella para la que *no* se deduce $(H_{i,j} \vee W_{i,j})$.

El cálculo para averiguar la implicación mediante PREGUNTAR se puede implementar utilizando cualquiera de los métodos descritos anteriormente en este capítulo. ¿IMPLICACIÓN-TV? (Figura 7.10) es obviamente impracticable, ya que debería enumerar 2^{64} filas. El DPLL (Figura 7.16) realiza las inferencias necesarias en pocos milisegundos, principalmente gracias a la heurística de propagación unitaria. El SAT-CAMINAR también se puede utilizar, teniendo en cuenta la advertencia acerca de la incompletitud. En el mundo de *wumpus*, los obstáculos para encontrar un modelo, realizados 10.000 intercambios,

función AGENTE-WUMPUS-LP(percepción) **devuelve** una *acción*
  entradas: *percepción*, una lista, [*hedor, brisa, resplandor*]
  variables estáticas: *BC*, una base de conocimiento, inicialmente conteniendo la «física» del mundo de wumpus
    x, y, orientación, la posición del agente (inicialmente en 1, 1) y su orientación (inicialmente *derecha*)
    visitada, una matriz indicando qué casillas han sido visitadas, inicialmente *falso*
    acción, la acción más reciente del agente, inicialmente null

  plan, una secuencia de acciones, inicialmente vacía
  actualiza *x, y, orientación, visitada* basada en *acción*
  si *hedor* **entonces** DECIR(BC, $M_{x,y}$) **sino** DECIR(BC, $\neg M_{x,y}$)
  si *brisa* **entonces** DECIR(BC, $B_{x,y}$) **sino** DECIR(BC, $\neg B_{x,y}$)
  si *resplandor* **entonces** *acción* ← *agarrar*
  sino si *plan* no está vacío **entonces** *acción* ← POP(*plan*)
  sino si para alguna casilla $[i, j]$ que nos rodea es *verdadero* PREGUNTAR(BC, $(\neg H_{i,j} \wedge \neg W_{i,j})$)
      o es *falso* PREGUNTAR(BC, $(H_{i,j} \vee W_{i,j})$) **entonces hacer**
    plan ← BUSQUEDA-GRAFO-A*(PROBLEMA-RUTA([*x, y*], *orientación*, [*i, j*], *visitada*))
    acción ← POP(*plan*)
  sino *acción* ← un movimiento escogido al azar
  devolver *acción*

Figura 7.19 Un agente en el mundo de wumpus que utiliza la lógica proposicional para identificar hoyos, wumpus y casillas seguras. La subrutina PROBLEMA-RUTA construye un problema de búsqueda cuya solución es una secuencia de acciones que nos llevan de $[x, y]$ a $[i, j]$ pasando sólo a través de las casillas previamente visitadas.

se corresponden invariablemente con la *insatisfacibilidad*, así que los errores no se deben probablemente a la incompletitud.

El programa AGENTE-WUMPUS-LP se comporta bastante bien en un mundo de *wumpus* pequeño. Sin embargo, sucede algo profundamente insatisfactorio respecto a la base de conocimiento del agente. La *BC* contiene las sentencias que definen la «física» en la forma dada en las Ecuaciones (7.1) y (7.2) *para cada casilla individual*. Cuanto más grande sea el entorno, más grande necesita ser la base de conocimiento inicial. Preferiríamos en mayor medida disponer sólo de las dos sentencias para definir cómo se presenta una brisa o un hedor en *cualquier* casilla. Esto está más allá del poder de expresión de la lógica proposicional. En el próximo capítulo veremos un lenguaje lógico más expresivo mediante el cual es más fácil expresar este tipo de sentencias.

Guardar la pista acerca de la localización y la orientación del agente

El programa del agente de la Figura 7.19 «hace trampa» porque guarda la pista de su localización *fuera* de la base de conocimiento, en vez de utilizar el razonamiento lógico[13]. Para hacerlo «correctamente» necesitaremos proposiciones acerca de la localización. Una primera aproximación podría consistir en utilizar un símbolo como $L_{1,1}$ para indicar que el agente se encuentra en la casilla [1, 1]. Entonces la base de conocimiento inicial podría incluir sentencias como

$$L_{1,1} \wedge \mathit{OrientadoDerecha} \wedge \mathit{Avanzar} \Rightarrow L_{2,1}$$

Vemos inmediatamente que esto no funciona correctamente. Si el agente comienza en la casilla [1, 1] orientado a la derecha y avanza, de la base de conocimiento se deduciría $L_{1,1}$ (la localización original del agente) y $L_{2,1}$ (su nueva localización). ¡Aunque estas dos proposiciones no pueden ser verdaderas a la vez! El problema es que las proposiciones de localización deberían referirse a dos instantes de tiempo diferentes. Necesitamos $L^1_{1,1}$ para indicar que el agente se encuentra en la casilla [1, 1] en el instante de tiempo 1, $\mathrm{L}^2_{2,1}$ para indicar que el agente se encuentra en la casilla [2, 1] en el instante de tiempo 2, etcétera. Las proposiciones acerca de la orientación y la acción también necesitan depender del tiempo. Por lo tanto, la sentencia correcta es

$$L^1_{1,1} \wedge \mathit{OrientadoDerecha}^1 \wedge \mathit{Avanzar}^1 \Rightarrow \mathrm{L}^2_{2,1}$$
$$\mathit{OrientadoDerecha} \wedge \mathit{GirarIzquierda}^1 \Rightarrow \mathit{OrientadoArriba}^2$$

De esta manera resulta bastante difícil construir una base de conocimiento completa y correcta para guardar la pista de cada cosa que sucede en el mundo de *wumpus*; pospondremos la discusión en su detalle para el Capítulo 10. Lo que nos proponemos hacer aquí es que la base de conocimiento inicial contenga sentencias como los dos ejemplos anteriores para cada instante de tiempo *t*, así como para cada localización. Es

[13] El lector que sea observador se habrá dado cuenta de que esto nos posibilitó afinar la conexión que hay entre las percepciones, como *Brisa* y la proposiciones acerca de la localización específica, como $B_{1,1}$.

decir, que para cada instante de tiempo t y localización $[x, y]$, la base de conocimiento contenga una sentencia del tipo

$$L^t_{x,y} \wedge OrientadoDerecha^t \wedge Avanzar^t \Rightarrow L^{t+1}_{x+1,y} \tag{7.3}$$

Aunque pongamos un límite superior de pasos permitidos en el tiempo (por ejemplo 100) acabamos con decenas de miles de sentencias. Se presenta el mismo problema si añadimos las sentencias «a medida que las necesitemos» en cada paso en el tiempo. Esta proliferación de cláusulas hace que la base de conocimiento sea ilegible para las personas, sin embargo, los resolutores rápidos en lógica proposicional aún pueden manejar un mundo de *wumpus* de 4×4 con cierta facilidad (éstos encuentran su límite en los mundos de 100×100 casillas). Los agentes basados en circuitos de la siguiente sección ofrecen una solución parcial a este problema de proliferación de las cláusulas, pero la solución íntegra deberá esperar hasta que hayamos desarrollado la lógica de primer orden en el Capítulo 8.

Agentes basados en circuitos

AGENTE BASADO EN CIRCUITOS

CIRCUITO SECUENCIAL

PUERTAS

REGISTROS

Un **agente basado en circuitos** es un tipo particular de agente reflexivo con estado interno, tal como se definió en el Capítulo 2. Las percepciones son las entradas de un **circuito secuencial** (una red de **puertas**, cada una de ellas implementa una conectiva lógica, y de **registros**, cada uno de ellos almacena el valor lógico de una proposición atómica). Las salidas del circuito son los registros que se corresponden con las acciones, por ejemplo, la salida *Agarrar* se asigna a *verdadero* si el agente quiere coger algo. Si la entrada *Resplandor* se conecta directamente a la salida *Agarrar*, el agente cogerá el objetivo (el objeto *oro*) siempre que vea el resplandor. (*Véase* Figura 7.20.)

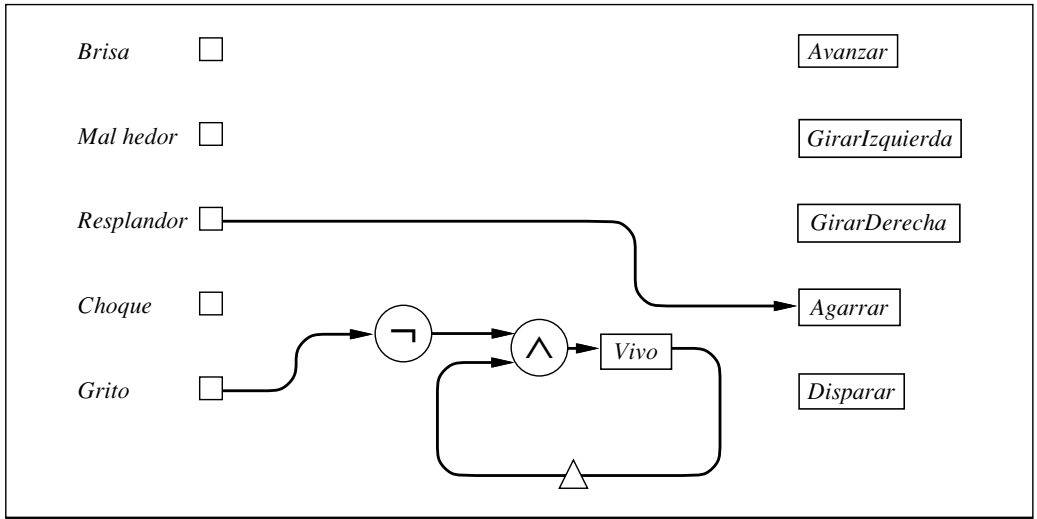


Figura 7.20 Parte de un agente basado en circuitos para el mundo de *wumpus*, mostrando las entradas, las salidas, el circuito para coger el oro, y el circuito que determina si el *wumpus* está vivo. Los registros se muestran como rectángulos, y los retardos de un paso se muestran como pequeños triángulos.

FLUJO DE DATOS

Los circuitos se evalúan de igual modo que los **flujos de datos**: en cada instante de tiempo, se asignan las entradas y se propagan las señales a través del circuito. Siempre que una puerta disponga de todas sus entradas, ésta produce una salida. Este proceso está íntimamente relacionado con el de encadenamiento hacia delante en un grafo Y-O, como el de la Figura 7.15(b).

En la sección precedente dijimos que los agentes basados en circuitos manejan el tiempo de forma más satisfactoria que los agentes basados en la inferencia proposicional. Esto se debe a que cada registro nos da el valor de verdad de su correspondiente símbolo proposicional *en el instante de tiempo actual t*, en vez de disponer de una copia diferente para cada instante de tiempo. Por ejemplo, podríamos tener un registro para *Vivo* que debería contener el valor *verdadero* cuando el *wumpus* esté vivo, y *falso* cuando esté muerto. Este registro se corresponde con el símbolo proposicional $Vivo^t$, de esta manera, en cada instante de tiempo el registro se refiere a una proposición diferente. El estado interno del agente, es decir, su memoria, mantiene conectando hacia atrás la salida de un registro con el circuito mediante una **línea de retardo**. Este mecanismo nos da el valor del registro en el instante de tiempo previo. En la Figura 7.20 se muestra un ejemplo. El valor del registro *Vivo* se obtiene de la conjunción de la negación del registro *Grito* y de su propio valor anterior. En términos de proposiciones, el circuito del registro *Vivo* implementa la siguiente conectiva bicondicional

LÍNEA DE RETARDO

$$Vivo^t \Leftrightarrow \neg Grito^t \wedge Vivo^{t-1} \tag{7.4}$$

que nos dice que el *wumpus* está vivo en el instante *t si y sólo si* no se percibe ningún grito en el instante *t* y estaba vivo en el instante $t - 1$. Asumimos que el circuito se inicializa con el registro *Vivo* asignado a *verdadero*. Por lo tanto, *Vivo* permanecerá siendo verdadero hasta que haya un grito, con lo que se convertirá y se mantendrá en el valor falso. Esto es exactamente lo que deseamos.

La localización del agente se puede tratar, en mucho, de la misma forma que la salud del *wumpus*. Necesitamos un registro $L_{x,y}$ para cada *x* e *y*; su valor debería ser *verdadero* si el agente se encuentra en la casilla $[x, y]$. Sin embargo, el circuito que asigna el valor de $L_{x,y}$ es mucho más complicado que el circuito para el registro *Vivo*. Por ejemplo, el agente está en la casilla [1, 1] en el instante *t* si: (a) estaba allí en el instante $t - 1$ y no se movió hacia delante o lo intentó pero tropezó con un muro; o (b) estaba en la casilla [1, 2] orientado hacia abajo y avanzó; o (c) estaba en la casilla [2, 1] orientado a la izquierda y avanzó:

$$\begin{aligned} L^t_{1,1} \quad \Leftrightarrow \quad & (L^{t-1}_{1,1} \wedge (\neg Avanzar^{t-1} \vee Tropezar^t)) \\ & \vee (L^{t-1}_{1,2} \wedge (OrientadoAbajo^{t-1} \wedge Avanzar^{t-1})) \\ & \vee (L^{t-1}_{2,1} \wedge (OrientadoIzquierda^{t-1} \wedge Avanzar^{t-1})) \end{aligned} \tag{7.5}$$

En la Figura 7.21 se muestra el circuito para $L_{1,1}$. Cada registro de localización tiene enlazado un circuito similar a éste. En el Ejercicio 7.13(b) se pide diseñar un circuito para las proposiciones de orientación.

Los circuitos de las Figuras 7.20 y 7.21 mantienen los valores correctos de los registros *Vivo* y $L_{x,y}$ en todo momento. Sin embargo, estas proposiciones son extrañas, en el sentido de que *sus valores de verdad correctos siempre se pueden averiguar*. En lu-

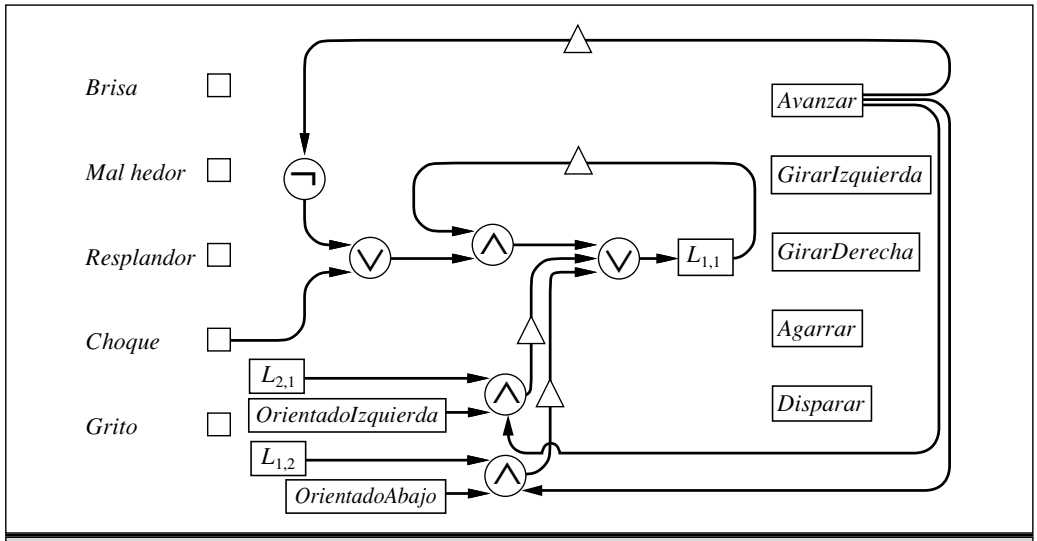


Figura 7.21 El circuito para determinar si el agente está en la casilla [1, 1]. Cada registro de localización y de orientación tiene enlazado un circuito similar.

gar de ello, consideremos la proposición $B_{4,4}$: en la casilla [4, 4] se percibe una brisa. Aunque el valor de verdad de esta proposición se mantiene fijo, el agente no puede aprender su valor hasta que haya visitado la casilla [4, 4] (o haya deducido que hay un hoyo cercano). Las lógicas proposicional y de primer orden están diseñadas para representar proposiciones verdaderas, falsas, o inciertas, de forma automática. Los circuitos no pueden hacerlo: el registro $B_{4,4}$ debe contener *algún* valor, bien *verdadero* o *falso*, aun antes de que se descubra su valor de verdad. El valor del registro bien podría ser el erróneo, y esto nos llevaría a que el agente se extraviara. En otras palabras, necesitamos representar tres posibles estados (se sabe que $B_{4,4}$ es verdadero, falso, o desconocido) y sólo tenemos un *bit* para representar estos estados.

La solución a este problema es utilizar dos bits en vez de uno. $B_{4,4}$ se puede representar mediante dos registros, a los que llamaremos $K(B_{4,4})$ y $K(\neg B_{4,4})$, en donde K significa «conocido». (¡Tenga en cuenta que esta representación consiste tan sólo en unos símbolos con nombres complicados, aunque parezcan expresiones estructuradas!) Cuando ambas, $K(B_{4,4})$ y $K(\neg B_{4,4})$, son falsas, significa que el valor de verdad de $B_{4,4}$ es desconocido. (¡Si ambas son ciertas, entonces la base de conocimiento tiene un fallo!) A partir de ahora, utilizaríamos $K(B_{4,4})$ en vez de $B_{4,4}$, siempre que ésta aparezca en alguna parte del circuito. Por lo general, representamos cada proposición que es potencialmente indeterminada mediante dos **proposiciones acerca del conocimiento** para especificar si la proposición subyacente se sabe que es verdadera y se sabe que es falsa.

PROPOSICIONES ACERCA DEL CONOCIMIENTO

En breve veremos un ejemplo de cómo utilizar las proposiciones acerca del conocimiento. Primero necesitamos resolver cómo determinar los valores de verdad de las propias proposiciones acerca del conocimiento. Fíjese en que, mientras $B_{4,4}$ tiene un valor de verdad fijo, $K(B_{4,4})$ y $K(\neg B_{4,4})$ *cambian* a medida que el agente descubre más cosas acerca del mundo. Por ejemplo, $K(B_{4,4})$ comienza siendo falsa y entonces se transforma en verdadera tan pronto como se puede determinar $B_{4,4}$ que es verdadera (es decir, cuan-

do el agente está en la casilla [4, 4] y detecta una brisa). A partir de entonces permanece siendo verdadera. Así tenemos

$$K(B_{4,4})^t \Leftrightarrow K(B_{4,4})^{t-1} \vee (L^t_{4,4} \wedge Brisa^t) \tag{7.6}$$

Se puede escribir una ecuación similar para $K(\neg B_{4,4})^t$.

Ahora que el agente sabe acerca de las casillas con brisa, puede ocuparse de los hoyos. La ausencia de un hoyo en una casilla se puede averiguar si y sólo si se sabe que en una de sus casillas vecinas no hay ninguna brisa. Por ejemplo, tenemos

$$K(\neg H_{4,4})^t \Leftrightarrow K(\neg B_{3,4})^t \vee K(\neg B_{4,3})^t \tag{7.7}$$

Determinar si *hay* un hoyo en una casilla es más difícil, debe haber una brisa en una casilla adyacente que no sea causada por otro hoyo:

$$\begin{array}{rcl} K(H_{4,4})^t & \Leftrightarrow & (K(B_{3,4})^t \wedge K(\neg H_{2,4})^t \wedge K(\neg H_{3,3})^t) \\ & \vee & (K(B_{4,3})^t \wedge K(\neg H_{4,2})^t \wedge K(\neg H_{3,3})^t) \end{array} \tag{7.8}$$

Mientras que utilizar los circuitos para determinar la presencia o ausencia de hoyos puede resultar algo peliagudo, *sólo se necesitan un número constante de puertas para cada casilla*. Esta propiedad es esencial si debemos construir agentes basados en circuitos que se pueden ampliar de forma razonable. En realidad ésta es una propiedad del propio mundo de *wumpus*; decimos que un entorno manifiesta **localidad** si el valor de verdad de una proposición relevante se puede obtener observando sólo un número constante de otras proposiciones. La localidad es muy sensible a la «física» precisa del entorno. Por ejemplo, el dominio de los dragaminas (Ejercicio 7.11) no es un dominio localista, porque determinar si una mina se encuentra en una casilla dada puede requerir observar las casillas que están arbitrariamente lejos. Los agentes basados en circuitos no son siempre practicables para los dominios no localistas.

ACÍCLICO

Hay un asunto por el que hemos caminado de puntillas y con mucho cuidado: el tema es la propiedad de ser **acíclico**. Un circuito es acíclico si cada uno de sus caminos en el que se conecta hacia atrás la salida de un registro con su entrada se realiza mediante un elemento de retardo. ¡Necesitamos que todos los circuitos sean acíclicos porque los que no lo son, al igual que los artefactos físicos, no funcionan! Éstos pueden entrar en oscilaciones inestables produciendo valores indefinidos. Como ejemplo de un circuito cíclico, tenga en cuenta la siguiente ampliación de la Ecuación (7.6):

$$K(B_{4,4})^t \Leftrightarrow K(B_{4,4})^{t-1} \vee (L^t_{4,4} \wedge Brisa^t) \vee K(H_{3,4})^t \vee K(H_{4,3})^t \tag{7.9}$$

Los disyuntores extras, $K(H_{3,4})^t$ y $K(H_{4,3})^t$, permiten al agente determinar si hay brisa a partir del conocimiento de la presencia de hoyos en las casillas adyacentes, algo que parece totalmente razonable. Pero ahora, desafortunadamente, tenemos que la detección de la brisa depende de los hoyos adyacentes, y la detección de los hoyos depende de las brisas adyacentes, si tenemos en cuenta la Ecuación (7.8). Por lo tanto el circuito, en su conjunto, tendría ciclos.

La dificultad no es que la Ecuación (7.9) ampliada sea *incorrecta*. Más bien, el problema consiste en que las dependencias entrelazadas que se presentan en estas ecuaciones no se pueden resolver por el simple mecanismo de la propagación de los valores de

verdad en el correspondiente circuito Booleano. La versión acíclica utilizando la Ecuación (7.6), que determina si hay brisa sólo a partir de las observaciones directas en la casilla es *incompleta*, en el sentido de que en algunos puntos, el agentes basado en circuitos podría saber menos que un agente basado en inferencia utilizando un procedimiento de inferencia completo. Por ejemplo, si hay una brisa en la casilla [1, 1], el agente basado en inferencia puede concluir que también hay una brisa en la casilla [2, 2], mientras que el agente basado en circuitos no puede hacerlo, utilizando la Ecuación (7.6). *Se puede* construir un circuito completo (después de todo, los circuitos secuenciales pueden emular cualquier computador digital) pero sería algo significativamente más complejo.

Una comparación

Los agentes basados en inferencia y los basados en circuitos representan los extremos declarativo y procesal en el diseño de agentes. Se pueden comparar según diversas dimensiones:

- *Precisión*: el agente basado en circuitos, a diferencia del agente basado en inferencia, no necesita disponer de copias diferentes de su «conocimiento» para cada instante de tiempo. En vez de ello, éste sólo se refiere al instante actual y al previo. Ambos agentes necesitan copias de la «física» de cada casilla (expresada mediante sentencias o circuitos), y por lo tanto, no se amplían adecuadamente a entornos mayores. En entornos con muchos objetos relacionados de forma compleja el número de proposiciones abrumará a cualquier agente proposicional. Este tipo de entornos requiere del poder expresivo de la lógica de primer orden. (*Véase* Capítulo 8.) Además, ambos tipos de agente proposicional están poco preparados para expresar o resolver el problema de encontrar un camino a una casilla segura que esté cercana. (Por esta razón, el AGENTE-WUMPUS-LP recurre a un algoritmo de búsqueda.)
- *Eficiencia computacional*: en el *peor de los casos*, la inferencia puede tomar un tiempo exponencial respecto al número de símbolos, mientras que evaluar un circuito toma un tiempo lineal respecto al tamaño del circuito (o lineal respecto a la *intensidad de integración* del circuito, si éste se construye como un dispositivo físico). Sin embargo, vemos que en la *práctica*, el DPLL realiza las inferencias requeridas bastante rápidamente[14].
- *Completitud*: anteriormente insinuamos que el agente basado en circuitos podría ser incompleto, debido a la restricción de que sea acíclico. Las causas de la incompletitud son en realidad más básicas. Primero, recordemos que un circuito se ejecuta en tiempo lineal respecto a su tamaño. Esto significa que, para algunos entornos, un circuito que sea completo (a saber, uno que calcula el valor de verdad de cada proposición determinable) debe ser exponencialmente más grande que la *BC* de un agente basado en inferencia. Dicho de otro modo, deberíamos poder resolver el problema de la implicación proposicional en menor tiempo que el tiempo exponencial, lo que parece improbable. Una segunda causa es la naturaleza del

[14] De hecho, ¡todas las inferencias hechas por un circuito se pueden hacer por el DPLL en tiempo lineal! Esto es debido a que la evaluación de un circuito, al igual que el encadenamiento hacia delante, se puede emular mediante el DPLL, utilizando la regla de propagación unitaria.

estado interno del agente. El agente basado en inferencia recuerda cada percepción que ha tenido, y conoce, bien implícita o explícitamente, cada sentencia que se sigue de las percepciones y la *BC* inicial. Por ejemplo, dado $B_{1,1}$, el agente conoce la disyunción $H_{1,2} \vee H_{2,1}$, de lo cual se sigue $B_{2,2}$. Por el otro lado, el agente basado en circuitos olvida todas sus percepciones anteriores y recuerda tan sólo las proposiciones individuales almacenadas en los registros. De este modo, $H_{1,2}$ y $H_{2,1}$ permanecen desconocidas *cada una de ellas* aún después de la primera percepción, así que no se sacará ninguna conclusión acerca de $B_{2,2}$.

- *Facilidad de construcción*: este es un aspecto muy importante acerca del cual es difícil ser precisos. Desde luego, los autores encontramos mucho más fácil especificar la «física» de forma declarativa, mientras que idear pequeños circuitos acíclicos, y no demasiado incompletos, para la detección directa de hoyos, nos ha parecido bastante dificultoso.

En suma, parece que hay *compensaciones* entre la eficiencia computacional, la concisión, la completitud y la facilidad de construcción. Cuando la conexión entre las percepciones y las acciones es simple (como en la conexión entre *Resplandor* y *Agarrar*) un circuito parece que es óptimo. En un dominio como el ajedrez, por ejemplo, las reglas declarativas son concisas y fácilmente codificables (como mínimo en la lógica de primer orden), y en cambio, utilizar un circuito para calcular los movimientos directamente a partir de los estados del tablero, sería un esfuerzo inimaginablemente enorme.

Podemos observar estos diferentes tipos de compensaciones en el reino animal. Los animales inferiores, con sus sistemas nerviosos muy sencillos, quizá estén basados en circuitos, mientras que los animales superiores, incluyendo a los humanos, parecen realizar inferencia con base en representaciones explícitas. Esta característica les permite ejecutar funciones mucho más complejas de un agente. Los humanos también disponen de circuitos para implementar sus reflejos, y quizá, también para **compilar** sus representaciones declarativas en circuitos, cuando ciertas inferencias pasan a ser una rutina. En este sentido, el diseño de un **agente híbrido** (*véase* Capítulo 2) podría poseer lo mejor de ambos mundos.

COMPILACIÓN

7.8 Resumen

Hemos introducido los agentes basados en conocimiento y hemos mostrado cómo definir una lógica con la que los agentes pueden razonar acerca del mundo. Los principales puntos son los siguientes:

- Los agentes inteligentes necesitan el conocimiento acerca del mundo para tomar las decisiones acertadas.
- Los agentes contienen el conocimiento en forma de **sentencias** mediante un **lenguaje de representación del conocimiento**, las cuales quedan almacenadas en una **base de conocimiento**.
- Un agente basado en conocimiento se compone de una base de conocimiento y un mecanismo de inferencia. El agente opera almacenando las sentencias acerca del mundo en su base de conocimiento, utilizando el mecanismo de inferencia para

inferir sentencias nuevas, y utilizando estas sentencias nuevas para decidir qué acción debe tomar.

- Un lenguaje de representación del conocimiento se define por su **sintaxis**, que especifica la estructura de las sentencias, y su **semántica**, que define el **valor de verdad** de cada sentencia en cada **mundo posible**, o **modelo**.
- La relación de **implicación** entre las sentencias es crucial para nuestro entendimiento acerca del razonamiento. Una sentencia α implica otra sentencia β si β es verdadera en todos los mundos donde α lo es. Las definiciones familiares a este concepto son: la **validez** de la sentencia $\alpha \Rightarrow \beta$, y la ***insatisfacibilidad*** de la sentencia $\alpha \wedge \neg\beta$.
- La inferencia es el proceso que consiste en derivar nuevas sentencias a partir de las ya existentes. Los algoritmos de inferencia **sólidos** *sólo* derivan aquellas sentencias que son implicadas; los algoritmos **completos** derivan todas las sentencias implicadas.
- La **lógica proposicional** es un lenguaje muy sencillo compuesto por los **símbolos proposicionales** y las **conectivas lógicas**. De esta manera se pueden manejar proposiciones que se sabe son ciertas, falsas, o completamente desconocidas.
- El conjunto de modelos posibles, dado un vocabulario proposicional fijado, es finito, y así se puede comprobar la implicación tan sólo enumerando los modelos. Los algoritmos de inferencia basados en la **comprobación de modelos** más eficientes para la lógica proposicional, entre los que se encuentran los métodos de búsqueda local y *backtracking*, a menudo pueden resolver problemas complejos muy rápidamente.
- Las **reglas de inferencia** son patrones de inferencia sólidos que se pueden utilizar para encontrar demostraciones. De la regla de **resolución** obtenemos un algoritmo de inferencia completo para bases de conocimiento que están expresadas en **forma normal conjuntiva**. El **encadenamiento hacia delante** y el **encadenamiento hacia atrás** son algoritmos de razonamiento muy adecuados para bases de conocimiento expresadas en **cláusulas de Horn**.
- Se pueden diseñar dos tipos de agentes que utilizan la lógica proposicional: los **agentes basados en inferencia** utilizan algoritmos de inferencia para guardar la pista del mundo y deducir propiedades ocultas, mientras que los **agentes basados en circuitos** representan proposiciones mediante bits en registros, y los actualizan utilizando la propagación de señal de los circuitos lógicos.
- La lógica proposicional es razonablemente efectiva para ciertas tareas de un agente, pero no se puede escalar para entornos de tamaño ilimitado, a causa de su falta de poder expresivo para manejar el tiempo de forma precisa, el espacio, o patrones genéricos de relaciones entre objetos.

NOTAS BIBLIOGRÁFICAS E HISTÓRICAS

La ponencia «Programas con Sentido Común» de John McCarthy (McCarthy, 1958, 1968) promulgaba la noción de agentes que utilizan el razonamiento lógico para mediar entre sus percepciones y sus acciones. En él también erigió la bandera del declarativis-

mo, señalando que decirle a un agente lo que necesita saber es una forma muy elegante de construir *software*. El artículo «El Nivel de Conocimiento» de Allen Newell (1982) propone que los agentes racionales se pueden describir y analizar a un nivel abstracto definido a partir del conocimiento que el agente posee más que a partir de los programas que ejecuta. En Boden (1977) se comparan los enfoques declarativo y procedural en la IA. Este debate fue reanimado, entre otros, por Brooks (1991) y Nilsson (1991).

La Lógica tiene sus orígenes en la Filosofía y las Matemáticas de los Griegos antiguos. En los trabajos de Platón se encuentran esparcidos diversos principios lógicos (principios que conectan la estructura sintáctica de las sentencias con su verdad o falsedad, con su significado, o con la validez de los argumentos en los que aparecen). El primer estudio sistemático acerca de la lógica que se conoce lo llevó a cabo Aristóteles, cuyo trabajo fue recopilado por sus estudiantes después de su muerte, en el 322 a.C., en un tratado denominado *Organon*. Los **silogismos** de Aristóteles fueron lo que ahora podríamos llamar reglas de inferencia. Aunque los silogismos tenían elementos, tanto de la lógica proposicional como de la de primer orden, el sistema como un todo era algo endeble según los estándares actuales. No permitió aplicar los patrones de inferencia a sentencias de complejidad diversa, cosa que sí lo permite la lógica proposicional moderna.

SILOGISMOS

Las escuelas íntimamente ligadas de los estoicos y los megarianos (que se originaron en el siglo v d.C. y continuaron durante varios siglos después) introdujeron el estudio sistemático de la implicación y otras estructuras básicas que todavía se utilizan en la lógica proposicional moderna. La utilización de las tablas de verdad para definir las conectivas lógicas se debe a Filo de Megara. Los estoicos tomaron como válidas cinco reglas de inferencia básicas sin demostrarlas, incluyendo la regla que ahora denominamos Modus Ponens. Derivaron un buen número de reglas a partir de estas cinco, utilizando entre otros principios, el teorema de la deducción (página 236) y fueron mucho más claros que Aristóteles acerca del concepto de demostración. Los estoicos afirmaban que su lógica era completa, en el sentido de ser capaz de reproducir todas las inferencias válidas; sin embargo, lo que de ellos queda es un conjunto de explicaciones demasiado fragmentadas. Que se sepa, un buen relato acerca de las lógicas Megariana y Estoica es el texto de Benson Mates (1953).

La idea de reducir la lógica a un proceso puramente mecánico, aplicado a un lenguaje formal es de Leibniz (1646-1716). Sin embargo, su propia lógica matemática era gravemente deficiente, y él es mucho más recordado simplemente por introducir estas ideas como objetivos a alcanzar que por sus intentos de lograrlo.

George Boole (1847) introdujo el primer sistema detallado y factible sobre lógica formal en su libro *El análisis Matemático de la Lógica*. La lógica de Boole estaba totalmente modelada a partir del álgebra de los números reales y utilizó la sustitución de expresiones lógicamente equivalentes como su principal método de inferencia. Aunque el sistema de Boole no abarcaba toda la lógica proposicional, estaba lo bastante cerca como para que otros matemáticos completaran las lagunas. Schröder (1877) describió la forma normal conjuntiva, mientras que las cláusulas de Horn fueron introducidas mucho más tarde por Alfred Horn (1951). La primera explicación completa acerca de la lógica proposicional moderna (y de la lógica de primer orden) se encuentra en el *Begriffschrift* («Escritura de Conceptos» o «Notación conceptual») de Gottlob Frege (1879).

El primer artefacto mecánico para llevar a cabo inferencias lógicas fue construido por el tercer Conde de Stanhope (1753-1816). El demostrador de Stanhope podía manejar silogismos y ciertas inferencias con la teoría de las probabilidades. William Stanley Jevons, uno de esos que hizo mejoras y amplió el trabajo de Boole, construyó su «piano lógico» en 1869, artefacto para realizar inferencias en lógica Booleana. En el texto de Martín Gardner (1968) se encuentra una historia entretenida e instructiva acerca de estos y otros artefactos mecánicos modernos utilizados para el razonamiento. El primer programa de computador para la inferencia lógica publicado fue el Teórico Lógico de Newell, Shaw y Simon (1957). Este programa tenía la intención de modelar los procesos del pensamiento humano. De hecho, Martin Davis (1957) había diseñado un programa similar que había presentado en una demostración en 1954, pero los resultados del Teórico Lógico fueron publicados un poco antes. Tanto el programa de 1954 de Davis como el Teórico Lógico estaban basados en algunos métodos *ad hoc* que no influyeron fuertemente más tarde en la deducción automática.

Las tablas de verdad, como un método para probar la validez o *insatisfacibilidad* de las sentencias en el lenguaje de la lógica proposicional, fueron introducidas por separado, por Ludwig Wittgenstein (1922) y Emil Post (1921). En los años 30 se realizaron una gran cantidad de avances en los métodos de inferencia para la lógica de primer orden. En concreto, Gödel (1930) demostró que se podía obtener un procedimiento completo para la inferencia en lógica de primer orden, mediante su reducción a la lógica proposicional utilizando el teorema de Herbrand (Herbrand, 1930). Retomaremos otra vez esta historia en el Capítulo 9; aquí el punto importante es que el desarrollo de los algoritmos proposicionales eficientes de los años 60 fue causado en gran parte, por el interés de los matemáticos en un demostrador de teoremas efectivo para la lógica de primer orden. El algoritmo de Davis y Putnam (Davis y Putnam, 1960) fue el primer algoritmo efectivo para la resolución proposicional, pero en muchos casos era menos eficiente que el algoritmo DPLL con *backtracking* introducido dos años después (1962). En un trabajo de gran influencia de J. A. Robinson (1965) apareció la regla de resolución general y una demostración de su completitud, en el que también se mostró cómo razonar con lógica de primer orden, sin tener que recurrir a las técnicas proposicionales.

Stephen Cook (1971) demostró que averiguar la *satisfacibilidad* de una sentencia en lógica proposicional es un problema NP-completo. Ya que averiguar la implicación es equivalente a averiguar la *insatisfacibilidad*, éste es un problema co-NP-completo. Se sabe que muchos subconjuntos de la lógica proposcional se pueden resolver en un tiempo polinómico; las cláusulas de Horn son uno de estos subconjuntos. El algoritmo de encadenamiento hacia delante en tiempo lineal para las cláusulas de Horn se debe a Dowling y Gallier (1984), quienes describen su algoritmo como un proceso de flujo de datos parecido a la propagación de señales en un circuito. La *satisfacibilidad* ha sido uno de los ejemplos canónicos para las reducciones NP; por ejemplo, Kaye (2000) demostró que el juego del dragaminas (*véase* Ejercicio 7.11) es NP-completo.

Los algoritmos de búsqueda local para la *satisfacibilidad* se intentaron por diversos autores en los 80; todos los algoritmos estaban basados en la idea de minimizar las cláusulas insatisfechas (Hansen y Jaumard, 1990). Un algoritmo particularmente efectivo fue desarrollado por Gu (1989) e independientemente por Selman *et al*. (1992), quien lo lla-

mó GSAT y demostró que era capaz de resolver muy rápidamente un amplio rango de problemas muy duros. El algoritmo SAT-CAMINAR descrito en el capítulo es de Selman *et al*. (1996).

La «transición de fase» en los problemas aleatorios *k*-SAT de *satisfacibilidad* fue identificada por primera vez por Simon y Dubois (1989). Los resultados empíricos de Crawford y Auton (1993) sugieren que se encuentra en el valor del ratio cláusula/variable alrededor de 4,24 para problemas grandes de 3-SAT; este trabajo también describe una implementación muy eficiente del DPLL. (Bayardo y Schrag, 1997) describen otra implementación eficiente del DPLL utilizando las técnicas de satisfacción de restricciones, y (Moskewicz *et al*. 2001) describen el CHAFF, que resuelve problemas de verificación de *hardware* con millones de variables y que fue el ganador de la Competición SAT 2002. Li y Anbulagan (1997) analizan las heurísticas basadas en la propagación unitaria que permiten que los resolutores sean más rápidos. Cheeseman *et al*. (1991) proporcionan datos acerca de un número de problemas de la misma familia y conjeturan que todos los problemas NP duros tienen una transición de fase. Kirkpatrick y Selman (1994) describen mecanismos mediante los cuales la física estadística podría aclarar de forma precisa las «condiciones» que definen la transición de fase. Los análisis teóricos acerca de su *localización* son todavía muy flojos: todo lo que se puede demostrar es que se encuentra en el rango [3.003, 4.598] para 3-SAT aleatorio. Cook y Mitchell (1997) hacen un excelente repaso de los resultados en este y otros temas relacionados con la *satisfacibilidad*.

Las investigaciones teóricas iniciales demostraron que DPLL tiene una complejidad media polinómica para ciertas ditribuciones normales de problemas. Este hecho, potencialmente apasionante, pasó a ser menos apasionante cuando Franco y Paull (1983) demostraron que los mismos problemas se podían resolver en tiempo constante simplemente utilizando asignaciones aleatorias. El método de generación aleatoria descrito en el capítulo tiene problemas más duros. Motivados por el éxito empírico de la búsqueda local en estos problemas, Koutsoupias y Papadimitriou (1992) demostraron que un algoritmo sencillo de ascensión de colina puede resolver muy rápido *casi todas* las instancias del problema de la *satisfacibilidad*; sugiriendo que los problemas duros son poco frecuentes. Más aún, Schöning (1999) presentó una variante aleatoria de GSAT cuyo tiempo de ejecución esperado en el *peor de los casos* era de 1.333^n para los problemas 3-SAT (todavía exponencial, pero sustancialmente más rápido que los límites previos para los peores casos). Los algoritmos de *satisfacibilidad* son todavía una área de investigación muy activa; la colección de artículos de Du *et al*. (1999) proporciona un buen punto de arranque.

Los agentes basados en circuitos se remontan al trabajo de gran influencia de McCulloch y Pitts (1943), quienes iniciaron el campo de las redes neuronales. Al contrario del supuesto popular, el trabajo trata de la implementación del diseño de un agente basado en circuitos Booleanos en su cerebro. Sin embargo, los agentes basados en circuitos han recibido muy poca atención en la IA. La excepción más notable es el trabajo de Stan Rosenschein (Rosenschein, 1985; Kaelbling y Rosenschein, 1990), quienes desarrollaron mecanismos para compilar agentes basados en circuitos a partir de las descripciones declarativas del entorno y la tarea. Los circuitos para actualizar las proposiciones almacenadas en registros están íntimamente relacionados con el **axioma del estado sucesor** desarrollado para la lógica de primer orden por Reiter (1991). El trabajo de Rod Brooks

(1986, 1989) demuestra la efectividad de los diseños basados en circuitos para controlar robots (un tema del que nos ocuparemos en el Capítulo 25). Brooks (1991) sostiene que los diseños basados en circuitos son *todo* lo que se necesita en la IA (dicha representación y razonamiento es algo incómodo, costoso, e innecesario). Desde nuestro punto de vista, ningún enfoque es suficiente por sí mismo.

El mundo de *wumpus* fue inventado por Gregory Yob (1975). Irónicamente, Yob lo desarrolló porque estaba aburrido de los juegos basados en una matriz: la topología de su mundo de *wumpus* original era un dodecahedro; nosotros lo hemos retornado a la aburrida matriz. Michael Genesereth fue el primero en sugerir que se utilizara el mundo de *wumpus* para evaluar un agente.

EJERCICIOS

7.1 Describa el mundo de *wumpus* según las características de entorno tarea listadas en el Capítulo 2.

7.2 Suponga que el agente ha avanzado hasta el instante que se muestra en la Figura 7.4(a), sin haber percibido nada en la casilla [1, 1], una brisa en la [2, 1], y un hedor en la [1, 2], y en estos momentos está interesado sobre las casillas [1, 3], [2, 2] y [3, 1]. Cada una de ellas puede tener un hoyo y como mucho en una se encuentra el *wumpus*. Siguiendo el ejemplo de la Figura 7.5, construya el conjunto de los mundos posibles. (Debería encontrar unos 32 mundos.) Marque los mundos en los que la BC es verdadera y aquellos en los que cada una de las siguientes sentencias es verdadera:

$$\alpha_2 = \text{«No hay ningún hoyo en la casilla [2, 2]»}$$
$$\alpha_3 = \text{«Hay un } wumpus \text{ en la casilla [1, 3]»}$$

Por lo tanto, demuestre que $BC \models \alpha_2$ y $BC \models \alpha_3$.

7.3 Considere el problema de decidir si una sentencia en lógica proposicional es verdadera dado un modelo.

- ***a***) Escriba un algoritmo recursivo ¿VERDADERA-LP?(*s*, *m*) que devuelva *verdadero* si y sólo si la sentencia *s* es verdadera en el modelo *m* (donde el modelo *m* asigna un valor de verdad a cada símbolo de la sentencia *s*). El algoritmo debería ejecutarse en tiempo lineal respecto al tamaño de la sentencia. (De forma alternativa, utiliza una versión de esta función del repositorio de código en línea, *online*.)
- ***b***) Dé tres ejemplos de sentencias de las que se pueda determinar si son verdaderas o falsas dado un modelo parcial, que no especifique el valor de verdad de algunos de los símbolos.
- ***c***) Demuestre que el valor de verdad (si lo tiene) de una sentencia en un modelo parcial no se puede determinar, por lo general, eficientemente.
- ***d***) Modifique el algoritmo ¿VERDADERA-LP? para que algunas veces pueda juzgar la verdad a partir de modelos parciales, manteniendo su estructura recursiva y tiempo de ejecución lineal. Dé tres ejemplos de sentencias de las cuales *no* se detecte su verdad en un modelo parcial mediante su algoritmo.

e) Investigue si el algoritmo modificado puede hacer que ¿IMPLICACIÓN-TV? sea más eficiente.

7.4 Demuestre cada una de las siguientes aserciones:

a) α es válido si y sólo si *Verdadero* $\models \alpha$.
b) Para cualquier α, *Falso* $\models \alpha$.
c) $\alpha \models \beta$ si y sólo si la sentencia $(\alpha \Rightarrow \beta)$ es válida.
d) $\alpha \equiv \beta$ si y sólo si la sentencia $(\alpha \Leftrightarrow \beta)$ es válida.
e) $\alpha \models \beta$ si y sólo si la sentencia $(\alpha \wedge \neg\beta)$ es *insatisfacible*.

7.5 Considere un vocabulario con sólo cuatro proposiciones, A, B, C, y D. ¿Cuántos modelos hay para las siguientes sentencias?

a) $(\text{A} \wedge \text{B}) \vee (\text{B} \wedge \text{C})$
b) $\text{A} \vee \text{B}$
c) $\text{A} \Leftrightarrow \text{B} \Leftrightarrow \text{C}$

7.6 Hemos definido cuatro conectivas lógicas binarias.

a) ¿Hay otras conectivas que podrían ser útiles?
b) ¿Cuántas conectivas binarias puede haber?
c) ¿Por qué algunas de ellas no son muy útiles?

7.7 Utilizando un método a tu elección, verifique cada una de las equivalencias de la Figura 7.11.

7.8 Demuestre para cada una de las siguientes sentencias, si es válida, *insatisfacible*, o ninguna de las dos cosas. Verifique su decisión utilizando las tablas de verdad o las equivalencias de la Figura 7.11.

a) $Humo \Rightarrow Humo$
b) $Humo \Rightarrow Fuego$
c) $(Humo \Rightarrow Fuego) \Rightarrow (\neg Humo \Rightarrow \neg Fuego)$
d) $Humo \vee Fuego \vee \neg Fuego$
e) $((Humo \wedge Calor) \Rightarrow Fuego) \Leftrightarrow ((Humo \Rightarrow Fuego) \vee (Calor \Rightarrow Fuego))$
f) $(Humo \Rightarrow Fuego) \Rightarrow ((Humo \wedge Calor) \Rightarrow Fuego)$
g) $Grande \vee Mudo \vee (Grande \Rightarrow Mudo)$
h) $(Grande \wedge Mudo) \vee \neg Mudo$

7.9 (Adaptado de Barwise y Etchemendy (1993).) Dado el siguiente párrafo, ¿puede demostrar que el unicornio es un animal mitológico? ¿que es mágico?, ¿que tiene cuernos?

> Si el unicornio es un animal mitológico, entonces es inmortal, pero si no es mitológico, entonces es un mamífero mortal. Si el unicornio es inmortal o mamífero, entonces tiene cuernos. El unicornio es mágico si tiene cuernos.

7.10 Cualquier sentencia en lógica proposicional es lógicamente equivalente a la aserción de que no se presenta el caso en que cada mundo posible la haga falsa. Demuestre a partir de esta observación que cualquier sentencia se puede escribir en FNC.

7.11 El muy conocido juego del dragaminas está íntimamente relacionado con el mundo de *wumpus*. Un mundo del dragaminas es una matriz rectangular de N casillas con M minas invisibles esparcidas por la matriz. Cualquier casilla puede ser visitada por el agente; si se encuentra que hay una mina obtiene una muerte instantánea. El dragaminas indica la presencia de minas mostrando en cada casilla visitada el *número* de minas que se encuentran alrededor directa o diagonalmente. El objetivo es conseguir visitar todas las casillas libres de minas.

a) Dejemos que $X_{i,j}$ sea verdarero si y sólo si la casilla $[i, j]$ contiene una mina. Anote en forma de sentencia la aserción de que hay exactamente dos minas adyacentes a la casilla [1, 1] apoyándose en alguna combinación lógica de proposiciones con $X_{i,j}$.

b) Generalice su aserción de la pregunta (a) explicando cómo construir una sentencia en FNC que aserte que k de n casillas vecinas contienen minas.

c) Explique de forma detallada, cómo un agente puede utilizar DPLL para demostrar que una casilla contiene (o no) una mina, ignorando la restricción global de que hay exactamente M minas en total.

d) Suponga que la restricción global se construye mediante su método de la pregunta (b). ¿Cómo depende el número de cláusulas de M y N? Proponga una variación del DPLL para que la restricción global no se tenga que representar explícitamente.

e) ¿Algunas conclusiones derivadas con el método de la pregunta (c) son invalidadas cuando se tiene en cuenta la restricción global?

f) Dé ejempos de configuraciones de explorar valores que induzcan *dependencias a largo plazo* como la que genera que el contenido de una casilla no explorada daría información acerca del contenido de una casilla bastante distante. [*Pista*: pruebe primero con un tablero de $N \times 1$.]

7.12 Este ejercicio trata de la relación entre las cláusulas y las sentencias de implicación.

a) Demuestre que la cláusula $(\neg P_1 \vee \cdots \vee \neg P_m \vee Q)$ es lógicamente equivalente a la implicación $(P_1 \wedge \cdots \wedge P_m) \Rightarrow Q$.

b) Demuestre que cada cláusula (sin tener en cuenta el número de literales positivos) se puede escribir en la forma $(P_1 \wedge \cdots \wedge P_m) \Rightarrow (Q_1 \vee \cdots \vee Q_n)$, donde las Ps y las Qs son símbolos proposicionales. Una base de conocimiento compuesta por este tipo de sentencia está en **forma normal implicativa** o **forma de Kowalski**.

c) Interprete la regla de resolución general para las sentencias en forma normal implicativa.

7.13 En este ejercicio diseñaremos más cosas del agente *wumpus* basado en circuitos.

a) Escriba una ecuación, similar a la Ecuación (7.4), para la proposición *Flecha*, que debería ser verdadera cuando el agente aún tiene una flecha. Dibuje su circuito correspondiente.

b) Repita la pregunta (a) para *OrientadoDerecha*, utilizando la Ecuación (7.5) como modelo.

***c*)** Cree versiones de las Ecuaciones 7.7 y 7.8 para encontrar el *wumpus*, y dibuje el circuito.

7.14 Discuta lo que quiere significar un comportamiento *óptimo* en el mundo de *wumpus*. Demuestre que nuestra definición de AGENTE-WUMPUS-LP no es óptima, y sugiera mecanismos para mejorarla.

7.15 Amplíe el AGENTE-WUMPUS-LP para que pueda guardar la pista de todos los hechos relevantes que están *dentro* de su base de conocimiento.

7.16 ¿Cuánto se tarda en demostrar que $BC \models \alpha$ utilizando el DPLL cuando α es un literal que ya está en la BC? Explíquelo.

7.17 Traza el comportamiento del DPLL con la base de conocimiento de la Figura 7.15 cuando intenta demostrar Q, y compare este comportamiento con el del algoritmo del encadenamiento hacia delante.