Breves Notas sobre
Inteligencia Artificial

Jorge L. Ortega Arjona
Departamento de Matemáticas
Facultad de Ciencias, UNAM

Junio 2005

Índice general

Prefacio

Las *Breves Notas sobre Inteligencia Artificial* presentan en forma simple y sencilla algunos temas relevantes de Inteligencia Artificial. No tienen la intención de substituir a los diversos libros y publicaciones formales en el área, ni cubrir por completo los cursos relacionados, sino más bien, su objetivo es exponer brevemente y guiar al estudiante a través de los temas que, por su relevancia, se consideran esenciales para el conocimiento básico de esta área, desde una perspectiva del estudio de la Computación.

Los temas principales que se incluyen en estas notas son: Árboles de Juego, Redes Neuronales, Perceptrones, Computadoras Auto-reproductivas y Programación Lógica. Estos temas se exponen haciendo énfasis en los elementos que el estudiante (particularmente el estudiante de Computación) debe comprender en las asignaturas que se imparten como parte de la Licenciatura en Ciencias de la Computación, Facultad de Ciencias, UNAM.

Jorge L. Ortega Arjona
Junio 2005

Capítulo 1

Árboles de Juego
El Método Minimax

Un "árbol de juego" es una aproximación común para programar cualquier juego interactivo. En un árbol de juego, cada nodo representa una posible posición en el juego y cada rama representa un posible movimiento. Para ilustrar esta idea, se presenta un juego de damas simplificado, que utiliza un tablero de 4×4 (figura 1.1). En esta figura sólo se muestran tres niveles del árbol, que corresponden a posibles escenarios que se van dando conforme el juego avanza.

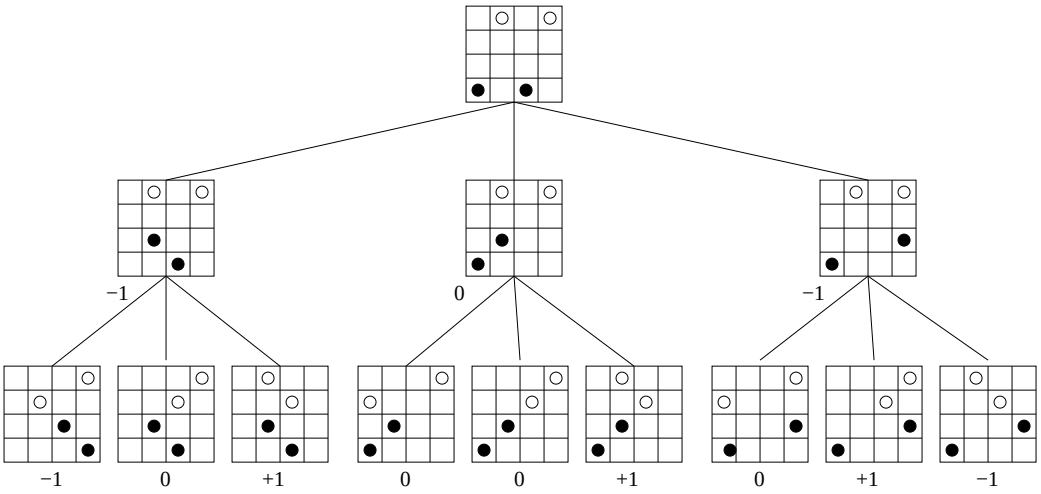

Figura 1.1: Parte de un juego de damas de 4×4

A cada nivel del árbol, los movimientos se alternan entre blancas y negras. Supóngase ahora que se pudiera analizar los tableros al tercer nivel del árbol, de modo que se pudiese "medir" y llegar a un número que refleje el estado de las fichas blancas de cada tablero a ese nivel del árbol. Por ejemplo, en el tablero de la extrema izquierda, las fichas blancas están a punto de perder una ficha, mientras que el siguiente tablero a la derecha representa un empate, y el siguiente es una victoria para las fichas blancas. Estas posibilidades se reflejan por los valores -1, 0 y $+1$ dados a cada uno de esos tableros. En el movimiento previo a estas tres posibilidades, en el segundo nivel del árbol, es el turno de las fichas negras, y quien las juega obviamente debe escoger el movimiento a la extrema izquierda, de modo que el resultado del juego se incline a su favor.

Por tanto, para cada tablero del segundo nivel, se puede seleccionar el valor mínimo asignado a sus subsequentes tableros. Esto lleva a una secuencia -1, 0, -1 de valores para las blancas en los tres tableros del segundo nivel. Sin embargo, las blancas pueden escoger el movimiento que deseen para llegar a este punto, y obviamente intentan aquel movimiento que les reditúe un mayor valor. En este caso, el movimiento central es su mejor opción, ya que aunque el valor del tablero es 0, al menos las blancas no pierden.

El proceso descrito hasta este punto se conoce con el nombre de *procedimiento Minimax*. De hecho, este procedimiento hace que una computadora programada para jugar fichas negras en un juego de damas de 4×4 intente aventajar a su oponente mediante aprovechar la estructura de árbol de juego, usando tres tipos de subprogramas:

- *Generación del árbol.* Los programas para la generación de árboles no son difíciles de construir para la mayoría de los juegos. Habiendo decidido un método de representación del tablero o situación de juego, el programador diseña un procedimiento para generar y almacenar todos los movimientos válidos a partir de una posición inicial.

- *Evaluación de posición.* Los programas para la evaluación de posición son un poco más complicados. Si fuera posible evaluar todos los posibles resultados de un juego, el programa evaluador de posición tendría la tarea relativamente simple de reconocer un empate o una victoria y, consecuentemente, una derrota entre las partes del juego.

  Normalmente, no se tiene ni el tiempo ni el espacio de generar el árbol completo de un juego, y el programa evaluador de posición se invoca en un nivel más profundo, pidiéndosele que retorne un valor para las

posiciones en que no resulta tan obvio distinguir quién va ganando. De cualquier modo, el programa evaluador utiliza varios criterios que deben ser especificados por el programador. Para el ejemplo del juego de damas: ¿qué tanta ventaja tienen las negras? ¿qué tan buena es la ventaja en todas y cada una de las posiciones de las negras de acuerdo con una medida numérica simple?

- *Procedimiento Minimax.* A partir del árbol evaluado por el programa, este procedimiento retorna los valores mínimos de algunas posiciones como consecuencia de las decisiones del oponente; de otro modo, retorna el valor máximo, lo que significa que busca inclinar el resultado del juego a su favor. Observando esto, es notorio que el procedimiento Minimax va alternando valores máximos y mínimos por cada nivel del árbol, intentando mejorar sus posibilidades de ganar el juego. Las blancas (por ejemplo, la computadora) selecciona cualquier movimiento que arroje el mayor valor en la posición actual. En seguida, el programa entra en un nuevo ciclo de operación, a fin de explorar tan profundo en el árbol como le sea posible, en busca de una probable respuesta del oponente. La búsqueda se limita atendiendo a la memoria y el tiempo que sean disponibles.

Un programa como el descrito fue realizado por Arthur Samuel en 1962. Tal programa podía jugar damas de 8×8, y alguna vez venció a un campeón estatal de los Estados Unidos.

Para juegos más complicados como el ajedrez, con un número de movimientos mayor por turno, el papel del programa evaluador se vuelve aun más crítico. Por ahora, los programas capaces de jugar ajedrez (algunos de los cuales son variantes del programa descrito aquí) se desarrollan en computadoras más poderosas, debido a la creciente necesidad de recursos para "mejorar su juego".

Es por esto que tiene un valor especial que para juegos más complejos como ajedrez y go se cuente con alguna técnica para disminuir el árbol de juego, a fin de que no se vuelva inmanejable rápidamente. Es interesante, pero tal técnica existe.

Examinando la porción superior del árbol de juego para damas de 4×4 una vez más (Figura 1.2), es notorio para dos de las posiciones disponibles para las blancas en el primer movimiento, hay una posición disponible para las negras que resulta en un valor de -1 para las blancas. Si los únicos valores

disponibles al tercer nivel fueran los dos -1, se podría eliminar la necesidad de explorar el árbol por las ramas que lleven a posiciones más allá del -1. De este modo, dos ramas del árbol de juego pueden ser eliminadas, bajo la suposición de que el oponente seleccionará en el segundo nivel cualquier movimiento que lleve a un -1 en ese nivel.

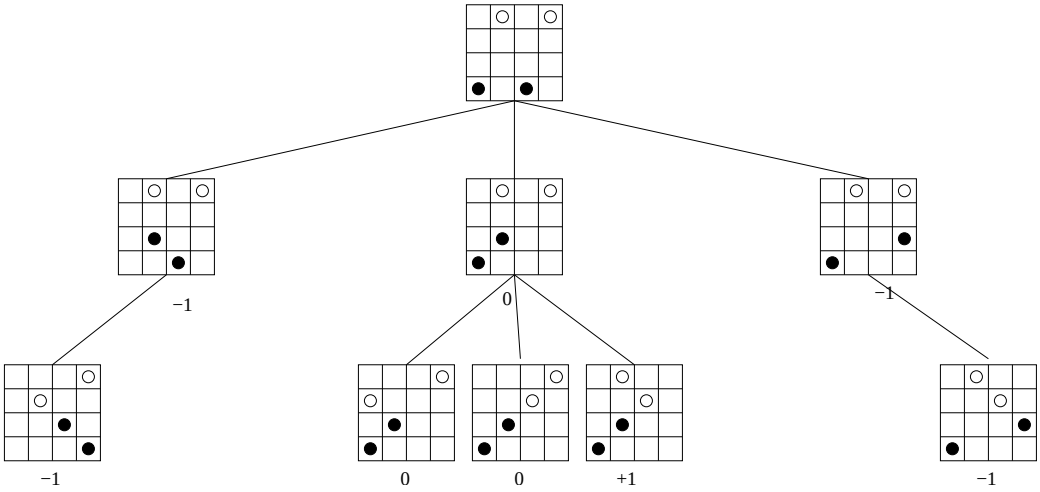

Figura 1.2: Disminuyendo el árbol de juego

Hasta este punto, hay solo tres ramas del árbol de juego que son necesarias de explorar: las tres resultantes en el centro. Supóngase que al explorar la tercera rama el programa llegara a un valor de $+1$ para uno de los dos posibles subsequentes tableros (Figura 1.3). No habría entonces necesidad de explorar la otra rama, ya que la computadora tendría un movimiento muy bueno disponible a partir del tablero previo. Se supone que las negras tratarían de evitar tal tablero, elegiendo en su turno cualquier otro movimiento.

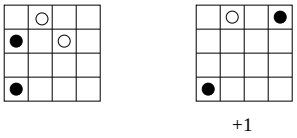

Figura 1.3: Un tablero subsecuente es una victoria para las blancas.

Este tipo de disminución del árbol de juego es lo que se conoce como *poda alfa-beta*. Como *Minimax*, este procedimiento busca una posición con valor α que represente el valor más pequeño al cual las blancas deben atenerse para cualquier movimiento que las negras hagan (Figura 1.4). Específicamente, supóngase una posición negra C cuyo valor se sabe que es α. Puede ser que la exploración de una rama E a partir de una posición B resulte en un valor $v < \alpha$. Por tanto, no hay razón de explorar las otras ramas de B, ya que es claro que las blancas preferirían un movimiento a C que a B. A partir de esto, las negras pueden reducir el valor de las blancas al valor v. Por lo tanto, las ramas de B se eliminan, y el análisis del árbol de juego sigue en la posición D.

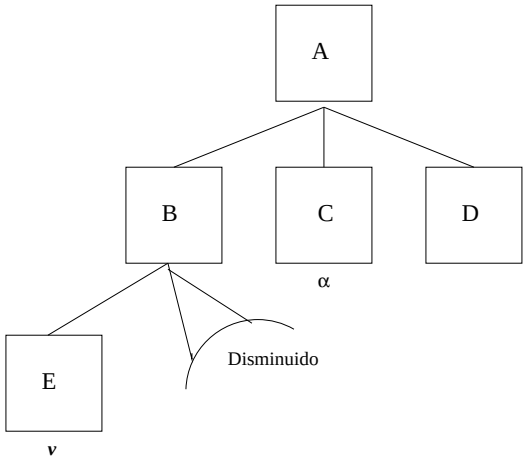

Figura 1.4: Un corte α.

En la búsqueda de un máximo, la poda alfa-beta busca el máximo valor β al cual las negras puedan mantenerse (Figura 1.5). Si se sabe que una posición de las blancas G tiene ya un valor β y si la exploración de otro tablero H arroja una posición I con un valor $v > \beta$, entonces claramente las negras prefieren un movimiento a G en lugar que a H, y no hay razón para explorar las otras ramas de H. Esto produce un "corte en beta".

Mediante esta técnica adicional, grandes trozos del árbol de juego pueden eliminarse y no ser considerados. Entre otras cosas, esto da al programador la opción de aumentar la velocidad del programa de juego o de mejorar su desempeño dentro del mismo marco de tiempo, permitiendo explorar más del árbol de juego que lo que antes era posible.

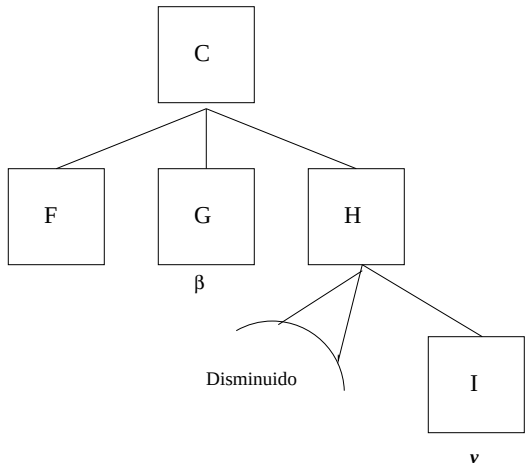

Figura 1.5: Un corte β.

No es difiícil mostrar que la poda alfa-beta de disminución de árboles resulta en exactamente los mismos movimientos de las blancas que si se exploraran todas aquellas ramas eliminadas. La mayoría de los programas de juegos actuales usan la poda alfa-beta, ya que casi siempre resulta en ahorros enormes de tiempo y espacio. Claro, existen árbol hipotéticos en los cuales la poda alfa-beta no ahorraría tiempo en lo absoluto, pero éstos no parecen ocurrir dentro de la práctica de los juegos.

Capítulo 2

Redes Neuronales
Un Intento de Cerebro

Las neuronas, tanto humanas como animales, son celdas delicadas y altamente complejas que llevan a cabo el pensamiento y toma de decisiones. Actualmente, se sabe que la membrana de una neurona es capaz de sostener una carga eléctrica. Cuando esta carga alcanza una cierta cantidad de carga eléctrica (llamado umbral), la neurona "dispara": una onda de de-polarización se extiende rápidamente sobre la superficie de la célula, viajando a lo largo de su axón en la forma de un impulso nervioso. Se trata de una onda de rápido incremento, seguida por un rápido decremento de carga. Aun cuando el axón se divide en un árbol de dendritas, el impulso nervioso viaja por cada rama del árbol, llegando finalmente a un pequeño bulbo adyacente a alguna otra neurona con la cual se comunica a través de una sinapsis. Si la sinapsis es excitatoria, el impulso que llega a la segunda célula incrementará la carga de su superficie, pudiendo causar su disparo. Pero si la sinapsis es inhibitoria, la célula no podrá disparar por algún tiempo.

Estas características básicas de la neurofisiología eran ya bien conocidas en 1941, cuando el matemático y médico Warren McCullough y el neurofisiólogo Walter Pitts decidieron construir un modelo de neuronas que reflejara su interconexión. La única propiedad de una neurona real que se mantiene con razonable exactitud en su modelo era la característica de todo-o-nada en la forma como las neuronas disparan. El modelo se inspiró en la visión de McCullough en cuanto a las cualidades lógicas de la actividad eléctrica de las neuronas: "si las neuronas A, B y C disparan, entonces también lo hará E, exceptuando que D dispare", se traduce a "si las proposiciones A, B y C son verdaderas, y si D es falsa, entonces E es verdadera". McCullough y Pitts

probaron que cualquier proposición lógica podía realizarse en la forma de una red de neuronas. Este resultado fue considerado por algunas personas a mediados del siglo XX como la explicación primitiva de cómo realmente los seres humanos piensan. Formó una base para lo que podría llamarse la "era cibernética", un período de varias décadas en las que los científicos creían que los cerebros artificiales estaban ya pronto a ser desarrollados, y en que muchas veces los supuestos de la imaginación rebasaban los resultados reales.

En este breve resumen, se presenta un ejemplo sencillo de una red neuronal que realiza un reconocimiento de patrones. Más adelante, se muestra que existe una equivalencia computacional entre redes neuronales y autómatas finitos. La intención es ilustrar cómo modelos computacionales que surgen en contextos completamente diferentes y que presentan muy pocas similitudes externas entre sí, resultan ser equivalentes. Por ejemplo, las máquinas de Turing y las funciones recursivas (ambas, mucho más poderosas que un autómata finito) parecen muy diferentes formalmente, pero al final resultan ser formulaciones equivalentes.

Una red neuronal (como aquéllas descritas por McCullough y Pitts) es una colección de neuronas y fibras. Cada neurona tiene un umbral, y cada fibra es excitatoria o inhibitoria. Las fibras se subdividen más aún, clasificándose en fibras de entrada, de interconexión y de salida. Un reloj maestro se usa para sincronizar los eventos en la red, generando un conteo entero $1, 2, 3, \ldots$ Una neurona puede disparar en la transición de t a $t+1$ si y sólo si el número de fibras excitatorias de entrada que llevan un pulso exceden su umbral, y claro, siempre y cuando ninguna fibra inhibitoria de entrada presente un pulso. Si una neurona dispara, podría pensarse que los pulsos que envía por sus fibras de salida toman una unidad de tiempo para alcanzar sus varios destinos.

Además de representar proposiciones lógicas, las redes neuronales son capaces de muchas otras cosas. La Figura 2.1 muestra una porción de una red neuronal que puede utilizarse para reconocer una forma sólida rectangular en el cuadriculado de una imagen. La cuadrícula de 6×6 puede considerarse como una "retina" formalizada, en la cual cada cuadro está claro u obscuro dependiendo de un patrón particular proyectado sobre la cuadrícula. Para cada conjunto de cuatro cuadros adyacentes, se crea un conjunto de cinco neuronas como se muestra a un lado de la cuadrícula. La fibra que va desde un cuadrado a una neurona llevará un pulso en el tiempo t si tal cuadro está iluminado; de otra manera, no habrá pulso en la fibra. La salida de la

neurona a la extrema derecha conducirá un pulso al tiempo $t+2$ si al tiempo t exactamente uno de los cuadros del conjunto es iluminado. Esto es fácil de comprender, ya que el único momento en que la neurona a la derecha puede recibir un pulso en el tiempo $t+1$ es cuando al menos una de las cuatro neuronas precedentes dispara. Sucede que sólo una de ellas puede disparar en un momento dado, que es cuando el cuadro que la excita (y solo ese cuadro) se ilumina: nótese que el pulso correspondiente a tal cuadro inhibe las otras tres neuronas.

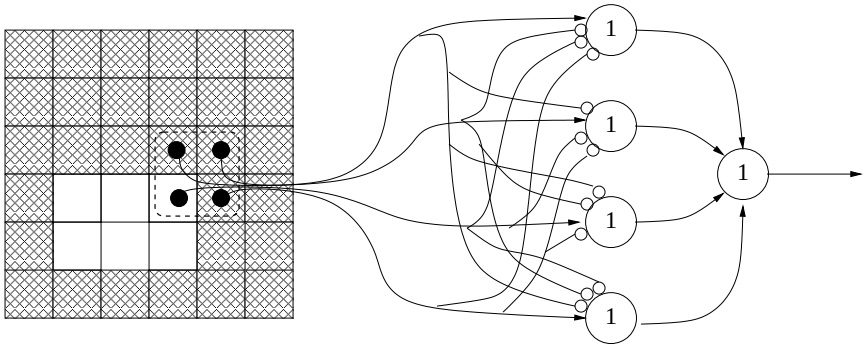

Figura 2.1: Parte de una red que reconoce rectángulos.

Para la cuadrícula de 6×6 que se muestra en la Figura 2.1, se requiere de un total de 25 conjuntos de neuronas que reportan las condiciones de iluminación sobre la cuadrícula. La neurona más a la derecha de estos 25 conjuntos envía una fibra excitatoria a dos neuronas extras: una de umbral 4 y otra de umbral 5 (Figura 2.2).

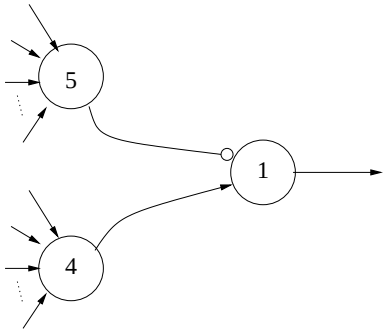

Figura 2.2: La decisión final se hace por tres neuronas.

Por lo tanto, si la figura iluminando la retina en el tiempo t tiene al menos 4 esquinas, entonces la neurona con umbral 4 disparará en el tiempo $t+3$. Si tiene más de 4 esquinas, entonces la neurona con umbral 5 disparará en el tiempo $t+3$. Evidentemente, la neurona final de la cadena debe disparar en el tiempo $t+4$, y solo si la figura tiene exactamente 4 esquinas. Suponiendo que la figura fuera conexa y no tuviera hoyos, entonces debe tratarse de un rectángulo.

Aun cuando poco impresiona la tarea de reconocimiento que esta red neuronal es capaz de hacer sobre la cuadrícula, debe recordarse que las redes neuronales se encuentran sujetas a un gran conjunto de limitaciones. Más aun, como se muestra a continuación, las redes neuronales no resultan más poderosas que un autómata finito, la más humilde de las computadoras en la jerarquía de Chomsky.

Sea N una red neuronal constituida de n neuronas. Constrúyase un autómata finito A con 2^n estados, y establézcase una correspondencia uno a uno entre los 2^n subconjuntos de las neuronas y los estados del autómata finito. Para esto, con un subconjunto particular X de las n neuronas, se asocian un estado x del autómata A. Es ciertamente posible (pero cuando n es grande, algo cansado) analizar la red N y determinar que por cada subconjunto X y la combinación I de fibras de entrada, qué subconjunto de neuronas X' se dispara en el tiempo $t+1$, dado que:

1. Las neuronas en X (y sólo esas neuronas) disparan en el tiempo t.
2. Las fibras en I (y sólo esas fibras de entrada) llevan pulsos durante el tiempo t.

De esta forma, se definen las transiciones entre los estados de A. El alfabeto de A es tan solo un conjunto de símbolos correspondientes al número total de combinaciones posibles de las fibras de entrada I, las cuales pueden portar un pulso durante cualquier intervalo de tiempo en particular. Si hay m fibras de entrada, entonces el alfabeto de A contiene 2^m símbolos.

Los autómatas finitos normalmente se consideran como "aceptadores" de lenguajes. La red neuronal que se muestra en la Figura 2.1 puede considerarse de manera similar al considerar el "lenguaje" como el conjunto de todas las formas rectangulares. La entrada a un estado de aceptación se simboliza por la neurona final, y solo esa neurona, cuando dispara.

Nótese que el autómata al que son equivalentes las redes neuronales es ligeramente diferente a un autómata finito propiamente hablando. Tal

autómata es conocido como *máquina de Mealy*, y es esencialmente un autómata finito en el que cada una de sus transiciones tiene asociado un símbolo de salida, tomado de un alfabeto de salida. Las P fibras de salida de la red neuronal N, por tanto, permiten completar la construcción de una máquina Mealy equivalente, creando un alfabeto de salida con 2^P símbolos, uno por cada posible combinación de fibras de salida que pueden llevar pulsos en un tiempo dado. Por tanto, una combinación X de neuronas dispara en el tiempo t, conjuntamente con la combinación de entrada I, determina no sólo la siguiente combinación X' de neuronas a disparar, sino también la combinación P de fibras de salida que llevan pulsos. En términos de su potencia computacional esencial (es decir, aceptación de lenguajes), las máquinas de Mealy y los autómatas finitos son idénticos.

Las redes neuronales que se describen aquí han sido desplazadas por modelos más sofisticados desarrollados principalmente por el matemático Stephen Grossberg y el físico John Hopfield, entre otros, durante los principios de los años 1970. En el nuevo estilo de redes neuronales, las señales viajan en forma asíncrona (no hay un reloj global), y manejan valores reales en lugar de códigos binarios. Las neuronas todavía disparan si la suma de las señales recibidas dentro de un tiempo dado excede su umbral. Tales redes no se consideran como entidades estáticas, sino que tienen la capacidad de ser "entrenadas": si la intensidad de la señal de varias fibras conectadas a una neurona aumenta mientras otras disminuye, el comportamiento de la red tiende a modificarse.

La descripción más moderna provee de redes neuronales capaces de simular aspectos simples de memoria asociativa humana y de encontrar soluciones a algunos problemas matemáticos. En ambos casos, la "experiencia" de entrada de la red, junto con un proceso contínuo de auto-ajuste, le permiten realizar sus funciones.

Capítulo 3

Perceptrones
Una Falta de Visión

Un *perceptrón* es una clase especial de computadora que examina y clasifica diferentes patrones que se presentan en una retina en forma de una cuadrícula, ponderando la evidencia que se le aplica por un número de dispositivos, cada uno "viendo" solo a una porción específica de la retina.

Por ejemplo, el perceptrón Π clasifica patrones en dos conjuntos (Figura 3.1):

- R: conjuntos de rectángulos disjuntos
- R': patrones no existentes en R

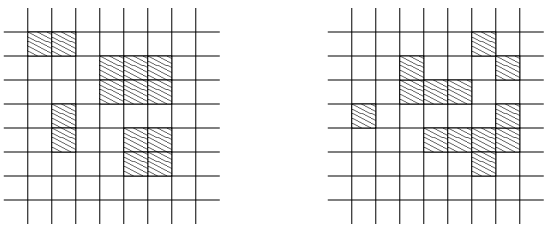

Figura 3.1: Rectángulos y no-rectángulos.

El perceptrón Π puede ser fácilmente construido, visualizándolo como una unidad ponderadora de evidencias que recibe entradas de sus dispositivos desplegados sobre la retina como se muestra en la Figura 3.2.

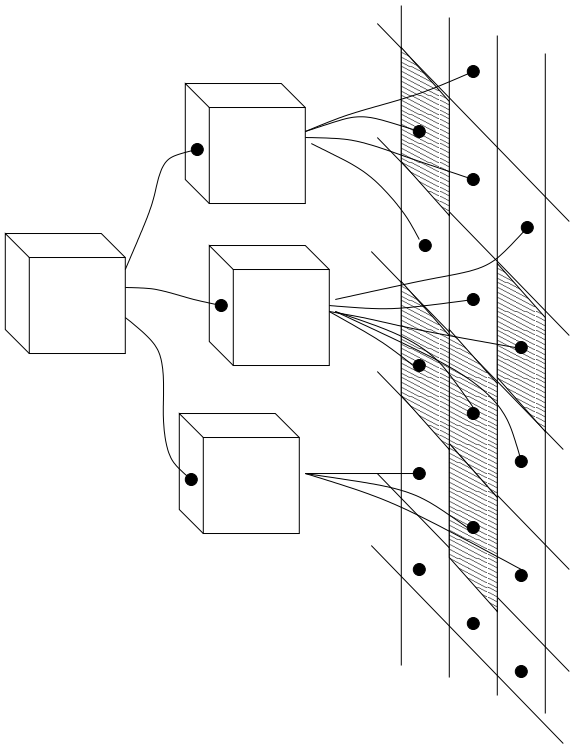

Figura 3.2: Un perceptrón.

Los dispositivos que reunen la evidencia acerca del estado actual del patrón sobre la retina x son llamados *predicados*, ya que:

- Cada uno examina un número finito de celdas en la retina, conocidas como sus *soportes*.
- Cada uno puede representarse como una función lógica de sus soportes. Cada celda puede considerarse como una variable booleana que es 1 si la celda está obscura, y es 0 de otra manera.

Cada predicado, por lo tanto, transmite un 1 o un 0 al componente principal del perceptrón. En cada predicado se realiza una suma de las ponderaciones de los valores de cada soporte, $\phi(x)$, que se compara con un umbral dado θ:

$$\sum \alpha_i \phi_i(x) : \theta$$

Si la suma no cae por debajo del umbral, entonces el patrón de entrada se declara como un miembro de la clase decidida por el perceptrón; de otra manera, se rechaza.

En el caso del perceptrón Π, cuyo trabajo es decidir si un patrón de entrada consiste de rectángulos disjuntos, la suma y el umbral son:

$$\sum (-1)\phi_i(x) : 0$$

Esta última expresión es fácil de computar. Para el i-ésimo punto de intersección en la retina, hay cuatro celdas tangentes y un predicado ϕ_i el cual las examina. Genera un 1 si cualquiera de los siguientes seis patrones se hallan en estas celdas (Figura 3.3).

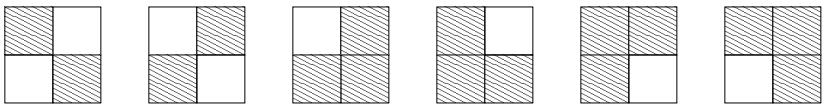

Figura 3.3: Patrones locales que producen no-rectángulos.

No debe ser difícil notar que un patrón presentado a Π es una colección disjunta de rectángulos si (y sólo si) ninguno de los sub-patrones anteriores aparecen en él. Pero ninguno de éstos aparece si (y sólo si) cada $\phi_i(x) = 0$.

Esta última condición es equivalente a escribir:

$$\sum \phi_i(x) \leq 0$$

ó

$$\sum (-1)\phi_i(x) \geq 0$$

La idea de perceptrones se debe originalmente a Frank Rosenblatt, quien con ello inició una gran actividad de investigación durante las décadas de 1950 y 1960, realizada por quienes estaban convencidos de que los perceptrones en ciertos aspectos eran capaces de actividades perceptuales inteligentes.

Sin embargo, ¿qué tan "inteligentes" son los perceptrones? Considerando el ejemplo previo, es posible descubrir algunas capacidades de los perceptrones en cuanto a la clasificación de patrones. Pero por otro lado, científicos como Marvin Minsky y Seymour Papert, convencidos de que un esquema tan simple de evidencia-ponderación no podría ser capaz de una genuina actividad inteligente, investigaron las limitaciones de los perceptrones durante la década de 1960. Descubrieron muchos problemas interesantes de discriminación en los que los perceptrones fallaron estrepitosamente. Uno de tales problemas fue el *problema de la conectividad*.

La Figura 3.4 muestra dos patrones. El patrón izquierdo se considera como conexo, mientras que el derecho no lo es. A partir de una consideración como ésta, ¿será posible construir un perceptrón que responda afirmativamente cada vez que se le presente un patrón conexo? La respuesta a esta pregunta depende de cómo se proponga.

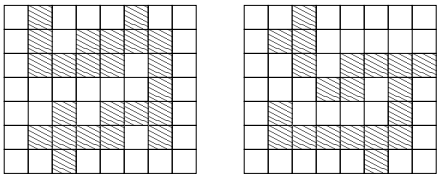

Figura 3.4: Patrones conexo y no conexo.

Si por ejemplo, el perceptrón tiene una retina fija de $n \times n$, entonces es posible construir un solo predicado capaz de distinguir patrones conexos de patrones no conexos. En términos lógicos, es posible construir una

enorme tabla de verdad con 2^{n^2} renglones, uno por capa patrón posible. El valor de verdadero del predicado puede ser 1 para cada renglón de la tabla correspondiente a un patrón conexo.

Tal construcción, sin embargo, se aleja de la idea original del perceptrón, en la cual la suma de ponderaciones juega un papel central. La única forma de prevenir la degeneración a esquemas de un solo predicado como el anterior es proporcionar algunas restricciones a los propios predicados. Dos limitaciones estudiadas por Minsky y Papert son bastante razonables desde un punto de vista práctico:

- *Perceptrones limitados en diámetro.* Cada predicado recibe las entradas dentro de un cuadro de k unidades de lado.
- *Perceptrones limitados en orden.* Cada predicado recibe entradas de a lo más k celdas de la retina.

Como una muestra de algunas interesantes y elegantes construcciones de Minsky y Papert, se muestra a continuación como el perceptrón limitado en diámetro no es capaz siempre de discriminar un patrón conexo de uno no conexo.

Considérense los cuatro patrones de la Figura 3.5, cada uno de los cuales con más de k celdas de longitud. Sea Γ un perceptrón limitado en diámetro cuyo diseñador sostiene que siempre puede determinar un patrón conexo de uno no conexo. Se le presenta a Γ un patrón C, y contesta un "no".

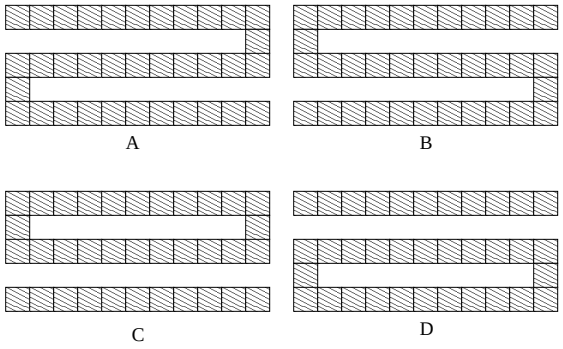

Figura 3.5: Engañando a un parceptrón limitado en diámetro.

Nótese que cada predicado de Γ debe ser de alguna de las siguientes clases:

- L: aquéllos que examinan al menos una celda del extremo izquierdo del patrón.
- R: aquéllos que examinan al menos una celda del extremo derecho del patrón.
- S: aquéllos que no están en L o R.

Por tanto, la desigualdad del umbral que no se satisface puede escribirse como:

$$\sum_{l \in L} \alpha_l \phi_l(C) + \sum_{s \in S} \alpha_s \phi_s(C) + \sum_{r \in R} \alpha_r \phi_r(C) < \theta$$

Si ahora reemplazamos el patrón C por el patrón A, sólo el extremo izquierdo del patrón cambia, de tal modo que solo la suma sobre L puede cambiar (las otras dos sumas permanecen sin cambio alguno). Ya que Γ debe responder un "sí" para A, se satisface la desigualdad, y debe ser cierto que:

$$\sum_{l \in L} \alpha_l \phi_l(A) > \sum_{l \in L} \alpha_l \phi_l(C)$$

En forma similar, al reemplazar C por B, sólo el extremo derecho del patrón cambia, y se tiene que:

$$\sum_{r \in R} \alpha_r \phi_r(B) > \sum_{r \in R} \alpha_r \phi_r(C)$$

La desigualdad se satisface de nuevo, y B es conexo. Ahora bien, cualquier incremento debe ser suficiente como para llevar a la suma sobre el umbral θ. Pero, ¿qué pasa cuando se le presenta a Γ el patrón D? Aquí, ambas sumas se incrementan, con la suma sobre S sin cambio, y la desigualdad es ciertamente satisfecha, así que el perceptrón Γ responde con "sí" para D, siendo que desafortunadamente, no es un patrón conexo.

Minsky y Papert continuaron demostrando que ningún perceptrón limitado en orden podría reconocer las figuras conexas. Además de problemas de conectividad, ambos autores encontraron muchas otras tareas en las que el perceptrón falla.

Capítulo 4

Computadoras Auto-reproductivas
La Máquina de Codd

Justo antes de su muerte en 1957, John von Neumann desarrolló la idea de una computadora capaz no sólo de computar cualquier función computable, sino también de reproducirse a sí misma. No es de sorprenderse que el método de la máquina para reproducirse tuviera poca semejanza con los métodos naturales. Más aún, tal computadora existió sólo como una especificación incompleta de cómo implementar una enorme colección de autómatas de 29 estados dentro de un plano. Aun si hubiera sido construida, tal vez no hubiera sido tan impresionante, ya que solo podría haber reproducido los patrones de sus estados, y no los dispositivos físicos que la componen.

Para ser más específicos, la idea que von Neumann tenía en mente se describe como sigue: un plano se divide en una malla cuadrada infinita, y cada cuadro se encuentra ocupado por el mismo autómata, llamado J. Inicialmente, todos excepto un número finito de estos autómatas se encuentran en un estado especial de reposo, mientras que el resto muestra un patrón de estados representando la máquina de von Neumann. El estado de cada autómata en el tiempo $t + 1$ depende estrictamente de su propio estado y del estado de sus cuatro vecinos durante el tiempo t (Figura 4.1).

No solamente podía la máquina de von Neumann computar cualquier función Turing-computable, sino también podía producir un duplicado de su patrón de estados ó, de hecho, un patrón de estados representando cualquier máquina de Turing especificada para aparecer en algún lugar de la malla.

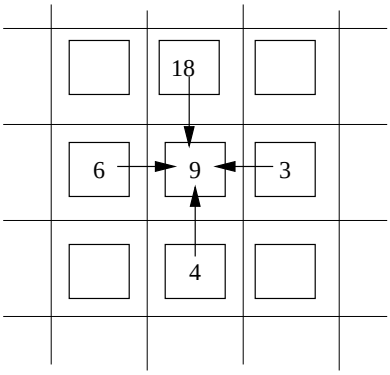

Figura 4.1: Una vecindad en el espacio celular de von Neumann.

Una máquina así creada podría entonces dejarse libre para realizar cualquier cómputo para el que estuviese diseñada.

Resulta entretenido tratar de imaginar el "constructor universal de computadoras" (*Universal Computer-Constructor* ó UCC), como von Neumann nombró a su computadora, propagándose a sí misma en forma interminable sobre una malla conceptual e infinita.

A mediados de los años 1960, E.F. Codd introdujo mejoras al diseño de von Neumann en muchos aspectos. La mejora más notoria fue la reducción de los 29 estados de autómata de von Neumann a tan solo una máquina de 8 estados. Tomaría mucho más que unas simples notas explicar la máquina de Codd en detalle, de tal modo que aquí únicamente puede darse una descripción general de ella. Por tanto, para dar una idea de su operación, se explica un aspecto específico de tal máquina de 8 estados: su "constructor de rutas" (*constructor paths*).

Una imagen general de la máquina de Codd (o simplemente UCC) involucra una "caja negra" y un número de cintas integradas en medio de una ranura en reposo, unidimensional e infinita en la malla (Figura 4.2).

La caja marcada *control de ejecución de la UCC* consiste de un patrón inmenso de estados, continuamente cambiando cuando la máquina se encuentra en operación; esto representa algo similar a los circuitos lógicos de una computadora moderna. La operación de esta enorme y plana computadora se describe a continuación.

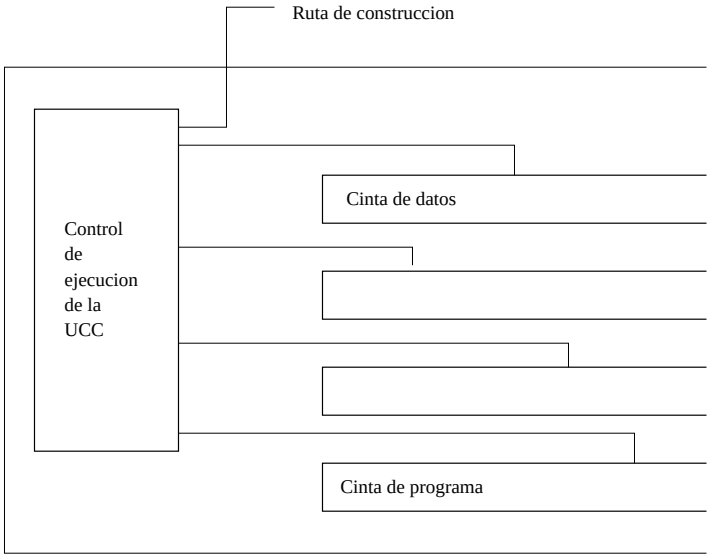

Figura 4.2: Un esquema de la UCC.

La descripción de una máquina de Turing arbitraria (o programa) se coloca en la *cinta de programa*, y los datos sobre los cuales tal máquina de Turing debe operar, se colocan en la *cinta de datos*. El control de ejecución entonces lee de la cinta de programa, e imita la acción de la máquina de Turing que se especifica ahí, leyendo y escribiendo la cinta de datos. Al hacer esto, la UCC va creando una serie de "rutas de datos" a partir de la acción del control de ejecución, es decir, del programa sobre los datos. Las rutas se extienden o retraen conforme procede la simulación de la máquina de Turing. Varios patrones de estados, representando símbolos a ser leídos o escritos, atraviesan estas rutas.

Se considera a la UCC como un "constructor universal". Dada la descripción de una máquina de Turing particular en la cinta del programa, la UCC es capaz de construir la máquina al ir extendiendo una ruta en un área en reposo adyacente a la UCC, y creando un patrón de estados que, cuando se activan, computan la función de esa máquina de Turing. De hecho, de esta forma, la UCC puede construir una copia de sí misma.

La máquina a ser construida por la UCC se describe en un lenguaje especial que, en efecto, dice mucho sobre cómo llevar a cabo la construcción en términos del desarrollo de la ruta de construcción. Específicamente, para

asegurar que cada celda se coloque en el estado inicial correcto, se extiende la ruta para esa celda, y una señal especial se propaga a lo largo de la ruta para efectuar la transición apropiada.

Respecto a las transiciones, aun cuando cada celda de este espacio contiene un autómata de 8 estados, el alfabeto es algo largo. De hecho, con cada una de sus cuatro celdas vecinas (y consigo misma) estando en cualquiera de los 8 estados, hay $8^5 = 32768$ posibles combinaciones de señales para las cuales el autómata debe responder. Sin embargo, Codd no consideró necesario especificar cada una de estas transiciones, ya que realmente solo se necesitan un "pequeño" subconjunto de esas transiciones.

A continuación, se explica finalmente cómo se crean y usan las rutas. Los varios estados en los cuales cada autómata puede encontrarse se etiquetan numéricamente: 0, 1, 2, 3, 4, 5, 6 y 7, siendo 0 el estado en reposo. Si una celda se encuentra en estado 0 y todas sus vecinas se encuentran en ese estado, entonces permanece en estado 0.

Esencialmente, una ruta en un estado que no se propaga consiste de una configuración de celdas en estado 1 rodeadas por un "aislante" de celdas en estado 2 (Figura 4.3). Para extender la ruta en la dirección a la que se dirige, se propaga el patrón 0607 a lo largo de la ruta. Para cuando llega al final de la ruta, se han añadido una celda en estado 1, y tres celdas en estado 2 al aislante (Figura 4.4).

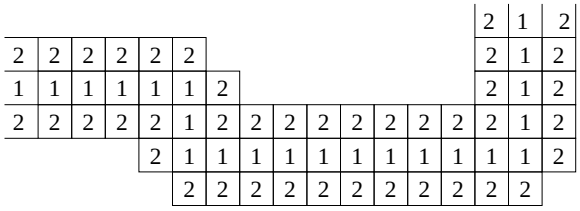

Figura 4.3: Una ruta no-propagable.

Nótese que cada uno de los pasos involucra una transición en un pequeño número de autómatas. Por ejemplo, la celda a la extrema derecha del segundo paso de la ruta que se encuentra en estado 1, está rodeada de celdas en estados 7, 2, 2, 2, tomadas en un orden convencional. Bajo las reglas de transición que gobiernan al autómata de Codd, se tiene la transición hacia el siguiente estado de la celda un instante de tiempo después. Esta transición es verdadera para cualquier autómata en el arreglo infinito que comprende

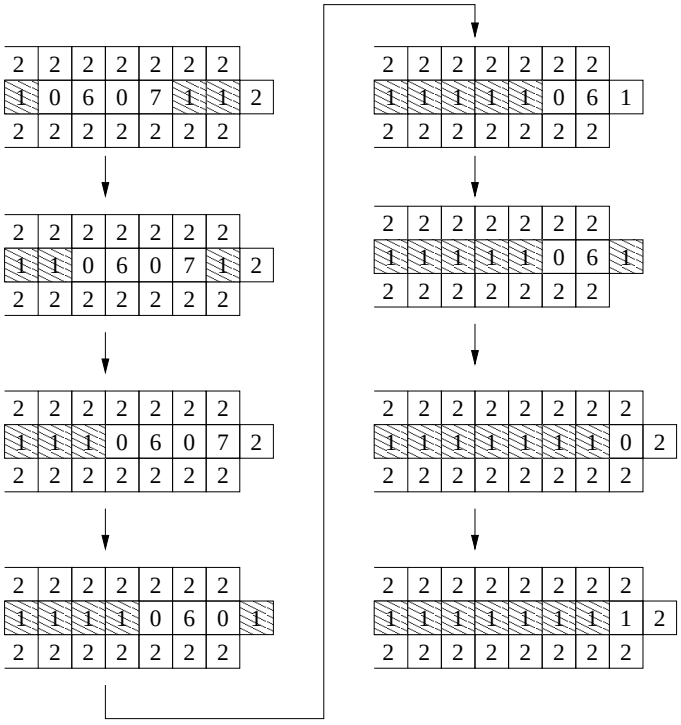

Figura 4.4: Extendiendo la ruta.

el espacio de las celdas: cuando una celda se encuentra en estado 1 y sus vecinos se encuentran en los estados 7, 2, 2 y 2, respectivamente, tal celda debe entrar al estado 7. El lector puede examinar y comprobar tal cambio en la Figura 4.4, y puede observar y desarrollar otras transiciones que el autómata debe obedecer.

Al desarrollar una ruta, ya sea leyendo una cinta o extendiéndola en una región en reposo, es útil hacer que la ruta gire a la izquierda o a la derecha. Por ejemplo, una vuelta a la izquierda puede señalarse con el patrón 04 propagándose por la ruta (Figura 4.5). Cuando se logra esta configuración, se envía otro 04 a lo largo de la ruta (Figura 4.6). En seguida, un 05 se transmite, seguido de un 06, resultando finalmente en la configuración que se muestra en la Figura 4.7, que puede extenderse hacia la izquierda de su dirección original mendiante un comando 0607.

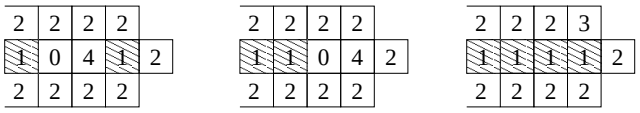

Figura 4.5: La ruta está a punto de dar vuelta.

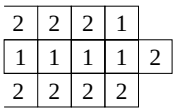

Figura 4.6: A la izquierda.

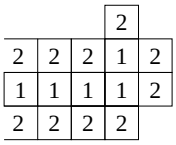

Figura 4.7: Finalmente, una vuelta a la izquierda.

Hay señales que provocan a la ruta entrar en estados específicos, particularmente modificando el estado de la celda en el extremo de construcción, o por ejemplo, resultar en la retracción de la ruta paso a paso.

Aun cuando sólo algunas partes de esta máquina han sido simulados por computadora para verificar la corrección de sus componentes, se ha especificado en suficiente detalle como para construir una, siempre y cuando se cuente con algunos cientos de años para crear un arreglo con el tamaño adecuado. En cualquier caso, es probable que sea, hasta ahora, el dispositivo de cómputo más grande y complejo que se haya concebido.

Capítulo 5

Programación Lógica
Prólogo a un Sistema Experto

Los lenguajes de programación tradicionales permiten al programador resolver problemas especificando soluciones programadas en forma de secuencias de enunciados o instrucciones. Sin embargo, otro tipo de lenguajes de programación permite representar en una forma totalmente diferente las soluciones computacionales, tan solo especificando ciertas propiedades lógicas que la solución debe tener. Los lenguajes de programación de esta segunda aproximación son aquéllos que desarrollan la forma de programación conicida como *programación lógica*.

La programación lógica se basa en el cálculo de primer orden para su operación. El lenguaje de programación lógica mejor conocido es tal vez *Prolog*, cuyo nombre se obtiene de la contracción de *pro*gramación *lóg*ica. Este lenguaje fue desarrollado por un grupo de científicos en Marsella, Francia, en 1976 (aun cuando una versión anterior se investigó en 1969 en el Reino Unido). Prolog ha sido propuesto en varios foros como el lenguaje fundamental para el desarrollo de aplicaciones de cómputo de la *quinta generación* de computadoras.

El poder lógico de Prolog puede ilustrarse en forma sencilla utilizando un ejemplo basado en las relaciones dentro de un árbol familiar (Figura 5.1). Un programa en Prolog consiste de un conjunto de *cláusulas*, cada una similar a las expresiones en cálculo de predicados.

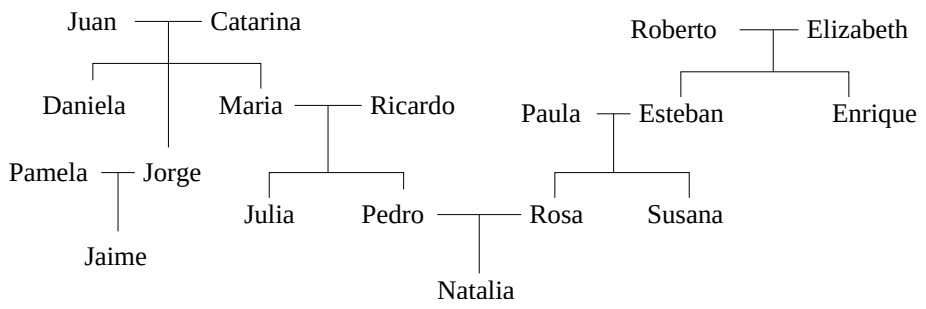

Figura 5.1: Un árbol genealógico.

Algunas cláusulas son *instancias* (contienen constantes pero no variables) mientras que otras contienen *variables* que pueden ser cuantificadas. Considérese por ejemplo el siguiente enunciado:

$$femenino(\text{María})$$

Esta es una instancia o cláusula unitaria. Sin embargo, el enunciado:

$$hermano(\text{X,Y}) \text{:-}\ masculino(\text{X}),\ progenitor(\text{Z,X}),\ progenitor(\text{Z,Y})$$

representa una cláusula cuantificable. Puede leerse como sigue: para todo X, Y, Z, si X es masculino, Z es progenitor de X, y Z es progenitor de Y, entonces X es hermano de Y. En notación lógica, este enunciado tiene la forma:

$$\forall X, Y, Z(masculino(X) \wedge progenitor(Z, X) \wedge progenitor(Z, Y) \Rightarrow hermano(X, Y))$$

Con este ejemplo, se puede comenzar a escribir un programa simple en Prolog. Primero, se listan un número de instancias, cada una consistente en un solo enunciado Prolog:

$progenitor(\text{Juan,Daniela})$
$progenitor(\text{Juan,Jorge})$
$progenitor(\text{Juan,Maria})$
$femenino(\text{Daniela})$
$masculino(\text{Jorge})$
$femenino(\text{Maria})$
$hermano(\text{X,Y})$:- $masculino(\text{X}),\ progenitor(\text{Z,X}),\ progenitor(\text{Z,Y})$

Hay al menos dos formas de interpretar la última línea de este programa. No se trata únicamente de la definición del significado de "hermano", sino que también provee de un procedimiento implícito para decidir si X es hermano de Y. Primero, se establece que X es masculino. Después, se ve si se tiene a un progenitor de X, y se comprueba que tal progenitor tiene dentro de su descendencia a Y.

El programa comienza su ejecución cuando el usuario hace una pregunta: ¿es Jorge hermano de Daniela? La pregunta se escribe más bien como un enunciado verdadero o falso, que se prueba mediante el sistema Prolog:

hermano(Jorge,Daniela)

Vistos como una serie de metas a alcanzar, las tres condiciones de la última línea del programa se examinan en orden, de izquierda a derecha. Revisando las cláusulas unitarias, el sistema Prolog establece rápidamente que *masculino*(X) es verdadero cuando X = Jorge. Esta es una forma sencilla de verificación. En seguida, Prolog descubre que *progenitor*(Juan,Jorge) es también verdadero. Por lo tanto, comienza a buscar un valor Z tal que *progenitor*(Z,Daniela) sea verdadero. De nuevo, Z = Juan cumple esta cláusula, y entonces, el programa termina con la respuesta "SI", ya que las tres condiciones se han cumplido.

El poder de deducción de Prolog va más allá de lo que este ejemplo muestra. Se pueden desarrollar programas más complejos que tengan otras respuestas además de "SI" o "NO". Supóngase, por ejemplo, que se desea descubrir los nombres de los primos de Jaime. Supóngase además que la primera parte del programa Prolog ya contiene todas las cláusulas unitarias relevantes, como por ejemplo, *progenitor*(María,Julia). Por supuesto, habrá algunas cláusulas unitarias que no provean de ninguna ayuda para obtener una respuesta. En los programas Prolog, no se puede estar seguro qué cláusulas unitarias serán útiles para un cómputo dado.

La parte principal del programa consiste de las cláusulas cuantificables:

hermano-o-hermana(X,Y):-*ne*(X,Y), *progenitor*(Z,X), *progenitor*(Z,Y)
primo(X,Y):-*progenitor*(W,X), *progenitor*(Z,Y), *hermano-o-hermana*(W,Z)

Finalmente, es necesario hacer la pregunta:

primo(Jaime,C)

Prolog enfrenta el problema buscando cláusulas a fin de encontrar una que contenga el predicado *primo*. Substituyendo X = Jaime, Prolog produce una cláusula nueva e intermedia que almacena en memoria:

primo(Jaime,C):- *progenitor*(W,Jaime), *progenitor*(Z,C),
hermano-o-hermana(W,Z)

Como en el ejemplo anterior, se establecen una serie de condiciones:

1. Encontrar un progenitor W de Jaime.
2. Encontrar un progenitor Z de C.
3. Encontrar un hermano o hermana Z de W.

En el momento en que las tres condiciones se satisfagan para un conjunto de valores dado W, C y Z, Prolog dará en su salida el valor encontrado para C, ya que esta información era implícitamente la pregunta a ser respondida.

Prolog utiliza la busqueda hacia atrás (*backtracking*) donde sea necesario para responder preguntas. Primero, revisa la lista de cláusulas unitarias, buscando un progenitor de Jaime. Encuentra que W = Pamela, para en seguida operar un par de condiciones nuevas:

1. Encontrar un progenitor Z de C.
2. Encontrar un hermano o hermana Z de Pamela.

Revisa ahora todas las cláusulas que involucran a progenitor, encontrando muchas combinaciones (Z,C) que satisfacen la primera de estas condiciones. Pero para cada valor de Z que se encuentra, Pamela no resulta ser hermana de ningún Z; simplemente, no hay indicación en las cláusulas unitarias de quiénes son los hermanos o hermanas de Pamela. Prolog hace una búsqueda hacia atrás al siguiente valor W, encontrando que se satisface que:

progenitor(W,Jaime)

Resulta que quien cumple con esta condición es Jorge. Por lo tanto, substituyendo W = Jorge en la segunda condición, se tiene ahora que encontrar un hermano o hermana Z de Jorge.

Prolog de nuevo busca las cláusulas de progenitor, y pronto descubre un progenitor Z de C que satisface la cláusula *progenitor*(Z,Jorge):

$$\text{Z} = \text{Juan}$$

En este caso, desafortunadamente, C resulta ser Daniela, y como Daniela no es progenitor de nadie, la búsqueda falla. Sin embargo, aún queda probar a María como valor para C, y cuando Prolog encuentra la cláusula unitaria *progenitor*(María,Julia), ha encontrado entonces valores para W, C y Z que satisfacen las condiciones que implican el predicado. Imprime entonces el valor de C como "Julia". Si se desean los nombres de todos los primos, se reinicia el programa, y encuentra a otro primo de Jaime: "Pedro".

Los tres predicados que comprenden la última cláusula no eran los únicos que podían iniciar el proceso de formación de condiciones. A través del cómputo anterior, Prolog contínuamente comprobaba:

$$ne(\text{X,Y}),\ progenitor(\text{Z,X}),\ progenitor(\text{Z,Y})$$

de izquierda a derecha. El primero de estos predicados pertenece al sistema de Prolog: es verdadero solamente cuando X y Y no tienen el mismo valor. Así que, para cuando Prolog prueba la paternidad de Z respecto a X y Y, éstas dos variables nunca tienen el mismo valor.

Nótese que el orden en que se escriben las cláusulas en Prolog pueden afectar la velocidad del cómputo. Sería mejor, de hecho, colocar la cláusula que define *primo* delante de la que define *hermano-o-hermana*. Esta cláusula tiene un número mayor de instancias que la satisfacen, y la mayoría no representan una respuesta. Es mejor entonces encontrar antes las instancias que fallan durante la búsqueda que hace Prolog de una solución.

Con esta breve y modesta introducción al espíritu de Prolog, es posible ilustrar su papel en sistemas expertos. Muy genéricamente, un *sistema experto* aplica un cierto grado de habilidad para razonar (típicamente, mediante el uso de programación lógica) a un conjunto de reglas y a una base de datos que comprenden lo que puede llamarse *conocimiento experto*. El usuario de tal sistema hacer preguntas dentro de un dominio de experiencia del programa y esperar respuestas útiles.

Los sistemas expertos se han construido para una gran variedad de dominios de conocimiento humano. Hay sistemas expertos que hacen una labor razonable en el diagnóstico de infecciones, en la evaluación de sitios para la

explotación minera, en la reparación de maquinaria, y en decidir algunas cuestiones legales basándose en estatutos. En cada caso, el conocimiento de hechos particulares útiles para el sistema se encuentran almacenados en un gran número de cláusulas unitarias. La clase de deducciones que un experto puede hacer basado en la evidencia disponible se guardan en predicados parecidos a aquéllos que definen *hermano-o-hermana* y *primo* en el ejemplo anterior. Además, las preguntas pueden enmarcarse dentro de la terminología del dominio en el que se utilizan tales predicados.

Aun en el humilde dominio de las relaciones humanas es posible imaginar un sistema experto en operación. Supóngase, por ejemplo, que se propone un matrimonio bajo la ley canónica de la Iglesia Católica Romana. Tal matrimonio se prohibe, o se considera nulo de realizarse, si ambas partes tienen un grado de consanguinidad menor al cuarto grado. El cómputo de ésto involucra la identificación de un antecesor común que se encuentra a tres generaciones (o menos) de los contrayentes. Es decir, se prohibe el matrimonio entre personas que tienen padre, madre, abuelo o abuela en común. Ya que los dos primeros casos implican los segundos, es suficiente proveer las siguientes cláusulas para un sistema experto en ley canónica del matrimonio:

abuelo-o-abuela(X,Y):- *padre-madre*(X,Z), *padre-madre*(Z,Y)
nulo(X,Y):- *abuelo-o-abuela*(Z,X), *abuelo-o-abuela*(Z,Y)

A estas cláusulas pueden añadirse otras que incluyan casos de matrimonios previos todavía en efecto, la edad de la contrayente menor que 14 años, la edad del contrayente menor que 16 años, o tal vez si alguno de los contrayentes ha hecho anteriomente un voto de castidad. Se puede prever que la lista es mucho más larga que lo que se muestra en este ejemplo.

Si tal sistema experto se desarrollara completamente, no requeriría grandes poderes de razonamiento. La gran mayoría de los predicados tendrían una forma no más complicada que, por ejemplo:

nulo(X,Y):- *castidad*(X)

o en el peor de los casos:

nulo(X,Y):- *ne*(Y,Z), *casado*(X,Z)

Ciertamente, tal simplicidad es común en sistemas expertos actuales. Pero el propio término *sistema experto* se utiliza ocasionalmente en forma lo suficientemente relajada como para incluir, por ejemplo, programas para jugar ajedrez. La experiencia de tales programas rara vez se desarrolla en un sistema de programación lógica principalmente debido a que en las máquinas secuenciales (que hoy por hoy, son la mayoría) la programación lógica se ejecuta en forma bastante lenta. Su operación natural sería en una computadora paralela, en la cual las muchas posibilidades del árbol de juego pueden simultáneamente probarse.

Bibliografía

[1] A. Barr and E.A. Feigenbaum, eds. *The Handbook of Artificial Intelligence, Vol. II.* Heuristic Tech Press, 1982.

[2] E.F. Codd *Cellular Automata.* Academic, 1968.

[3] D.Levy *Computer Gamemanship.* Century Publishing, 1983.

[4] M.L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, 1967.

[5] M.L. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, 1969.

[6] F. Rosenblatt *Principles of Neurodynamics.* Spartan Books, 1962.

[7] D.E. Rumelhart, J.L. McClelland, and the PDP Research Group *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations* MIT Press, 1987.

[8] A.I. Samuel *Some studies in machine learning using the game of checkers.* Computer and Thought, McGraw-Hill, 1963.

[9] J. von Neumann *Theory Automata: Construction, Reproduction, Homogeneity.* The Theory of Self-Reproducing Automata, University of Illinois Press, 1966.

[10] L. Woss *Automated Reasoning: Introduction and Applications.* Prentice-Hall, 1984.