5

Núcleos de sistemas expertos

Ingeniería del conocimiento

A. Rodríguez Rodríguez
J. Hernández Cabrera
A. Plácido Castro

Facultad de Informática
Universidad de Las Palmas de G.C.

Ingeniería del conocimiento

Núcleos de sistemas expertos
Volumen 5

Abraham Rodríguez Rodríguez
José Juan Hernández Cabrera
Ana María Plácido Castro

GIC S
Ingeniería del Software y del Conocimiento

Fotocopias de los documentos realizadas en el Opto. de Informática y Sistemas el 28 de Septiembre de 1.994

Dirección: Campus de Tafira, Edificio Departamental de Informática y Sistemas y Matemática Aplicada

Marca: RANK XEROX **Modelo:** 5028
Nº de serie: 2116197190

ISBN: 84-8098-027-3 (Volumen 5)
ISBN: 84-8098-023-0 (obra completa)

Contenidos

I Representación del Conocimiento

II Estrategias de resolución de problemas

III RETE

IV Sistemas de mantenimiento del razonamiento

CAPÍTULO I

Representación del Conocimiento

Índice

1 Introducción 1

2 Representaciones lógicas 3
- 2.1 Lógica por Defecto 4
- 2.2 Lógica Difusa 5

3 Redes semánticas 7
- 3.1 Redes Particionadas 9
- 3.2 Esquemas de Propagación de Marcas 9
- 3.3 Jerarquías de Tópicos 11
- 3.4 Redes Proposicionales 13

4 Representaciones procedurales 15
- 4.1 Winograd 16
- 4.2 Redes Semánticas Procedurales 16

5 Programación lógica 19

6 Marcos (Frames) 21

7 Arquitecturas de sistemas de producción 23

8 Blackboards (Pizarras) 25

9 Conclusiones 27

Bibliografía 29

1 Introducción

Lo que se ha dado en llamar *Representación del Conocimiento* en Inteligencia Artificial surge como respuesta a la necesidad de plasmar el 'conocimiento experto'. Este conocimiento específico de los distintos dominios de aplicación posee determinadas características que hace muy difícil su codificación mediante las técnicas de programación convencional. Según Levesque (1985) *'(la representación del conocimiento)...simplemente consiste en expresar, mediante algún lenguaje o sistema comunicativo, descripciones o escenas que se corresponden de alguna manera notoria con el mundo o algún estado del mismo'.*

En contraste con los sistemas tradicionales de bases de datos, los sistemas de IA requieren una base de conocimiento con diversos tipos de conocimiento. Estos incluyen, aunque no están limitados por, conocimiento acerca de objetos, de procesos, y el tan difícil de representar *sentido común*, conocimiento de metas, motivaciones, causalidades, tiempo, acciones, etc... A raíz de esta variedad de conocimiento surgen las siguientes cuestiones:

- ¿Cómo estructuramos el conocimiento explícito en una base de conocimiento?
- ¿Cómo codificamos las reglas para manipular el conocimiento explícito de una base de conocimiento para inferir el conocimiento contenido implícitamente dentro de la base de conocimiento?
- ¿Cómo controlamos dichas inferencias?
- ¿Cómo especificamos formalmente la semántica de una base de conocimiento?
- ¿Cómo se trata el conocimiento incompleto?
- ¿Cómo extraemos el conocimiento experto?
- ¿Cómo adquirimos conocimiento automáticamente a medida que pasa el tiempo de forma que la base de conocimiento mantenga su actualidad?

En los primeros sistemas de IA apenas se reconocía que la representación del conocimiento fuera una cuestión importante por propio derecho, aunque la mayoría de los sistemas incorporaban conocimiento mediante reglas y estructuras de datos. La representación del conocimiento surge lentamente a finales de los '60 como un área de estudio independiente. Comenzaron a manifestarse múltiples aproximaciones a la representación del conocimiento que desembocaron en distintos formalismos que todavía se utilizan hoy. Las principales aproximaciones son las representaciones lógicas, las redes semánticas, las representaciones procedurales, la programación lógica, las representaciones basadas en marcos, y las arquitecturas de sistemas de producción. Estas posibilidades no son mutuamente exclusivas, y algunas de ellas comparten diversos elementos en común. A continuación se introducirán estas aproximaciones.

2 Representaciones lógicas

La utilidad de las representaciones lógicas en el contexto de la representación del conocimiento se hizo evidente durante la década de los sesenta, principalmente como resultado de la investigación en la resolución de problemas mecánicamente y en sistemas de respuestas a preguntas.

Los esquemas de representación lógicos emplean las nociones de constante, variable, función, predicado, conectivo lógico, y cuantificador para representar hechos como fórmulas lógicas en alguna lógica (primer orden o mayor, multi-valuada, difusa, etc...). Según esto, una base de conocimiento es una colección de fórmulas lógicas que proporcionan una descripción parcial de un estado. Las modificaciones de la base de datos se producen con la introducción o eliminación de fórmulas lógicas. En este sentido, las fórmulas lógicas, sirven como unidades atómicas para la manipulación de la base de conocimiento en tales esquemas.

Una ventaja importante de los esquemas lógicos es la disponibilidad de las reglas de inferencia, que consisten en procedimientos automáticos de deducción de hechos de otros hechos. Tales procedimientos pueden utilizarse para la recuperación de información, la comprobación de restricciones semánticas, y la resolución de problemas.

Otra fortaleza de los esquemas lógicos es la disponibilidad de una semántica formal limpia, fácilmente comprensible, y ampliamente aceptada, al menos para aquellos esquemas lógicos 'puros' cercanos a la lógica de primer orden o el cálculo de predicados, lo que elimina las ambigüedades del lenguaje natural. En los esquemas de representación del cálculo de predicados la disponibilidad de una semántica y de una sintaxis precisa de la forma, y la existencia de un procedimiento automático de inferencia proporcionan la descripción completa que se necesita; o de otra forma, un forma interna está incompleta sin una especificación de lo que significa y de como se va a utilizar. A medida que nos desplazamos hacia los esquemas de representación que se enfrentan con la adquisición del conocimiento, con los valores por defecto, con creencias, con conocimiento cambiante, etc..., la disponibilidad de una semántica formal clara se hace más problemática.

La uniformidad de la notación empleada en los esquemas lógicos también es una ventaja, ya que facilita una descripción comprensible de la base de conocimiento, y la integración de

nueva información con la existente en la base de conocimiento. Otra ventaja más consiste en la economía conceptual que impulsan dichos esquemas, permitiendo que un hecho sólo se represente una vez, independientemente de sus diferentes usos durante el curso de su presencia en la base de conocimiento.

Entre las desventajas está la falta de principios organizacionales de los hechos que constituyen una base de conocimiento. Al igual que un programa de grandes dimensiones, una base de conocimiento grande necesita una serie de principios organizacionales que faciliten su comprensión como una unidad. Sin ellos, una base de conocimiento puede ser tan inmanejable como un programa, escrito en un lenguaje de programación que no posee facilidades de abstracción. En los casos en los que se necesiten estas facilidades, la lógica de primer orden es claramente insuficiente.

Realmente, la lógica de predicados es insuficiente para cualquier aplicación que pretenda trabajar con situaciones reales debido a las restricciones con el tiempo y el espacio. El número de predicados necesarios para describir cualquier situación del mundo real sería inmenso, y el intento de hacerlo completamente consistente sería casi imposible. Aún más, la naturaleza dinámica del mundo real dejaría rápidamente obsoleto cualquier conjunto de predicados. Desafortunadamente, el cálculo de predicados es especialmente inapropiado para representar información cambiante. Una dificultad corriente es el 'frame problem' - ¿cómo distinguir entre cosas que no cambian con el tiempo de aquellas que si cambian.

Una última desventaja es la dificultad para representar conocimiento procedural y heurístico como:

Si intentas hacer A mientras se verifica la condición B, intenta las estrategias C1, C2, ..., Cn.

2.1 Lógica por Defecto

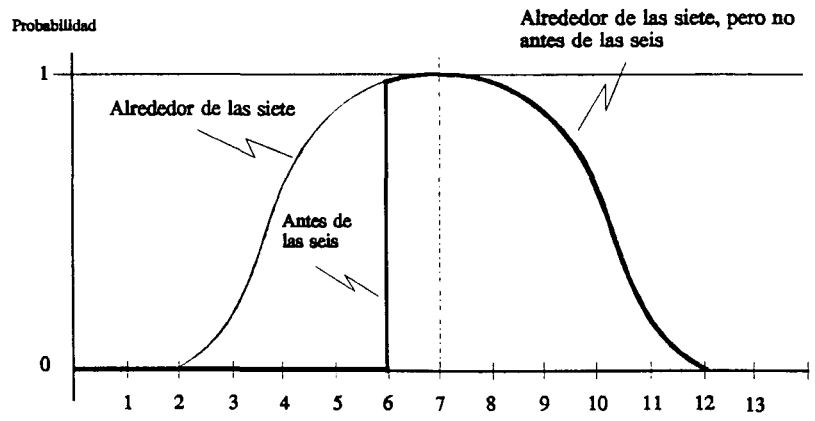
Por lo tanto, se han desarrollado gran cantidad de proyectos intentando extender la potencia del cálculo de predicados (a la vez que mantener su semántica formal) para manejar situaciones en las que el conocimiento es incompleto o cambiante. Este trabajo se ha realizado bajo la etiqueta general de *lógica no-monótona*, es decir, lógica en la que los valores de verdad de un predicado pueden cambiar en el tiempo. Un ejemplo de este tipo de lógica es la lógica por defecto. Una regla por defecto se puede formalizar como:

$$\frac{\alpha(x) \cdot M\beta_1(x), \ldots, M\beta_n(x)}{\omega(x)}$$

donde $\alpha(x)$, $\beta_1(x)$, ..., $\beta_n(x)$, y $\omega(x)$ son fórmulas de primer orden cuyas variables libres pertenecen a $x = x_1, \ldots, x_m$. Esta regla se interpretaría como: para todos los individuos $x_1, \ldots, x_m$, si se cree $\alpha(x)$ y si cada $\beta_1(x)$, ..., $\beta_n(x)$ es consistente con nuestras creencias, entonces puede creerse $\omega(x)$. Así, 'normalmente los perros ladran' se puede representar PERRO (x): M LADRA(x) / LADRA(x). La utilización de estas reglas por defecto permiten capturar en la base de conocimiento conceptos como 'normalmente', o 'en la ausencia de información que indique lo contrario', o 'típicamente', etc., liberando al diseñador del sistema de tener que especificar explícitamente cada excepción de cada una de las reglas. Las otras aproximaciones a la lógica no-monótona siguen pautas similares.

2.2 Lógica Difusa

Un aspecto que no aborda el cálculo de predicados es el de la incertidumbre, es decir, la representación de sentencias que no son siempre estrictamente verdaderas o falsas. La lógica difusa (fuzzy) fue introducida por Zadeh en 1979, afronta la imprecisión aparente mediante la asignación apriori de distintas medidas numéricas de credibilidad a proposiciones. La siguiente figura muestra en trazo grueso la frase 'alrededor de las siete, aunque no antes de las seis'. La primera parte de la sentencia se representa por una campana de Gauss, mientras que la segunda se asemeja a un escalón unitario.



3 Redes semánticas

Existe una gran variedad de redes semánticas, cada una enfatizando diferentes tipos de relaciones entre la información presentada en la red. Sin embargo, todos los esquemas de representación se basan en la idea del conocimiento representado mediante estructuras de grafo, con nodos que representan conceptos conectados con enlaces que representan relaciones semánticas binarias entre conceptos. Según este punto de vista, una base de conocimiento es una colección de objetos y relaciones, y las modificaciones se realizan mediante la inserción o eliminación de objetos y la manipulación de las relaciones. La mayoría de los esquemas favorecen el uso de enlaces binarios como medio de representación de relaciones binarias y de componentes de relaciones n-arias. Una base de conocimiento tiene una representación gráfica obvia donde cada nodo denota un objeto y cada pareja etiquetada (n_1, R, n_2) indica que $(n_1, n_2) \in R$, siendo R una de las relaciones utilizadas en la base de conocimiento.

Con una potencia representacional similar a la del cálculo de predicados (la lógica de primer orden tiene una traducción directa en redes semánticas), las redes semánticas se distinguen de dichas representaciones por su sentido de *conceptos cercanos*, es decir, lo cerca que están unos conceptos de otros en la red, y por los procedimientos que recorren los distintos enlaces de concepto a concepto para poder realizar las inferencias.

Las redes semánticas son realmente un intento de modelar las capacidades asociativas como las que emplean con éxito los humanos cuando razonan por analogía, y hablan metafóricamente. Desde un punto de vista semántico, las conexiones asociativas pueden hacerse entre conceptos directamente enlazados o al menos cercanos en la red.

Las primeras versiones de los esquemas en red tienden a premiar una proliferación de las relaciones que tienen poca o ninguna semántica cuando se representa conocimiento nuevo. Posteriormente se han diseñado otros esquemas que proponen un conjunto de relaciones primitivas.

Las bases organizacionales que ofrecen para estructurar una base de conocimiento constituyen una cuestión primordial en los esquemas en red. Estas sugieren direcciones de búsqueda de la información importante, restringiendo el tipo de búsqueda. A continuación

se comentan algunas de estas bases:

Clasificación: Un objeto (ej: Abraham Rodríguez) se asociaría con sus tipo/s genérico/s (ej. estudiante, persona, varón). Incluir esta base organizacional en un esquema en red fuerza a distinción entre elementos (ej: Abraham Rodríguez) y tipos (ej: persona). Algunos esquemas en red utilizan la clasificación recursivamente para definir (meta) tipos con tipos instancias, etc...

Agregación: Esta base relaciona un objeto (ej.: Abraham Rodríguez) con su componentes. Por ejemplo, las partes de Abraham Rodríguez, desde el punto de vista físico, son su cabeza, brazos, etc., cuando se le ve como un objeto social, son su dirección, dni, etc. Al igual que la clasificación, la agregación se puede aplicar recursivamente de forma que se puedan representar componentes de componentes de un objeto, etc. Así, esta agregación define una segunda dimensión organizacional para los esquemas en red.

Generalización: Relaciona un tipo (ej.: estudiante) con otros más genéricos (ej: persona). La relación de generalización entre tipos, denominada a menudo *is-a*, es una ordenación parcial y organiza los tipos en una jerarquía de generalización o is-a. Esta jerarquía ha servido para minimizar los requisitos de almacenamiento permitiendo que los objetos más especializados hereden propiedades de otros más generales. Además, la generalización y los otros tipos de primitivas de asociación proporcionan los medios para la organización y gestión de una gran base de conocimiento en conjunto.

Particiones: Esta implica la agrupación de objetos y elementos de relaciones en particiones que se organizan jerárquicamente, de forma que si la partición A está debajo de la B, todo lo que sea visible o esté presente en B también lo estará en A, a menos que se especifique lo contrario. Las particiones han sido útiles en la representación del tiempo, de mundos hipotéticos, y espacios de creencias.

Debido a su naturaleza, los esquemas en red abordan directamente los temas de recuperación de la información, ya que las asociaciones entre objetos definen caminos de acceso para recorrer la red. Otra característica importante de los esquemas en red es la

disponibilidad de principios organizacionales. Una tercera es la notación gráfica que puede utilizarse en estas bases de conocimiento y que mejora su legibilidad.

La principal desventaja de los esquemas en red ha sido la falta de una semántica formal y de una terminología estándar. Otra es la dificultad con la que afronta la inferencia con información dinámica o incompleta (no-monotonía).

A continuación consideraremos algunos ejemplos con mayor detalle.

3.1 Redes Particionadas

Hendrix acuñó esta versión de redes semánticas en 1979. "El principal objetivo de la partición es el de posibilitar la cohesión de grupos de nodos y arcos en unidades llamadas 'espacios', las cuales son entidades fundamentales en las redes particionadas, al mismo nivel que los nodos y los arcos.

Estos espacios pueden solaparse; cada nodo y arco de la red pueden pertenecer a uno o más espacios. Los conjuntos de espacios relacionados se agrupan en 'vistas', y la mayoría de las operaciones de la red intentan operar sobre la colección de nodos y arcos que comprenden las vistas. Aunque pueden crearse vistas libre y arbitrariamente, esta posibilidad se ejecuta rara vez, y la mayoría de los esquemas organizacionales tienden a ser jerárquicos.

Después de esbozar estructuras para la deducción lógica (conectivos lógicos, cuantificadores, y algoritmos de deducción), se incluyeron varias características como la herencia de información, razonamiento por valoraciones, razonamiento acerca de procesos, y comprensión del lenguaje natural.

3.2 Esquemas de Propagación de Marcas

La aproximación de Fahlman (1979) intenta solucionar el problema del mapeo de símbolos (symbol-mapping). La definición de este problema es la siguiente: 'supongamos que te digo que un determinado animal -llamémosle Clyde- es un elefante. Tú aceptas esta afirmación y la registras sin ningún esfuerzo mental aparente. Y a pesar de ello, parece como si de repente conocieras una gran cantidad de datos acerca de Clyde. Si digo que Clyde se sube a los árboles, o toca el piano, inmediatamente empezarás a dudar de mi credibilidad. De

alguna manera, 'elefante' ha servido de algo más que una mera etiqueta: en cierta medida es un 'paquete' completo de información con relaciones y propiedades que puede representarse con relaciones is-a. El medio por el que el símbolo elefante se extiende a las distintas implicaciones de la 'elefantería' se denomina problema del mapeo de símbolos.

Fahlman concibió una máquina de procesamiento paralelo en la que cada entidad conceptual (Clyde, elefante, gris) se representa por un dispositivo hardware llamado *nodo*. También tiene un control centralizado de dichos nodos a través de un bus compartido. Las relaciones se representan con otros dispositivos hardware denominados *enlaces*, los cuales pueden responder a órdenes en paralelo. El problema de esta máquina no radica tanto en la posibilidad de poner miles de nodos y enlaces en un chip, sino en la posibilidad de generar nuevas conexiones entre nodos y enlaces a medida que se añade más conocimiento. Esta conexiones deben ser líneas privadas o de lo contrario se perdería todo el paralelismo.

Además de desarrollar una notación de red paralela, Fahlman desarrolló un lenguaje de representación del conocimiento denominado NETL con las siguientes facilidades:

- La creación de un nuevo individuo de una clase determinado (ej. Clyde).
- La creación de una nueva descripción de un prototipo (ej. elefante).
- Lo división de una clase existente en clases no solapadas, y la detección de cualquier intento de violar la separación.
- La creación de papeles de herencia dentro de una descripción. Estos pueden tener valores por defecto, etc..
- La creación de tipos-de-papeles que se cargarán con un conjunto de objetos dentro de la descripción de un individuo particular.
- La creación y el procesamiento de excepciones a las afirmaciones globales.
- La creación de nuevos tipos de relaciones y de los enlaces que los representen.
- La creación de una jerarquía de áreas de contexto, sus partes y subpartes, y ámbito de las sentencias dentro de dichas áreas.
- La representación de acciones o eventos individuales en una jerarquía de tipos de eventos y el uso del tiempo como contexto para representar el efecto de una acción sobre el universo.
- La creación de universos hipotéticos que difieren del real en ciertos aspectos especificados.
- La separación de las propiedades que definen un conjunto de las incidentales.
- El uso de estrategias simples de reorganización de la red para realizar abstracciones y aprendizaje.

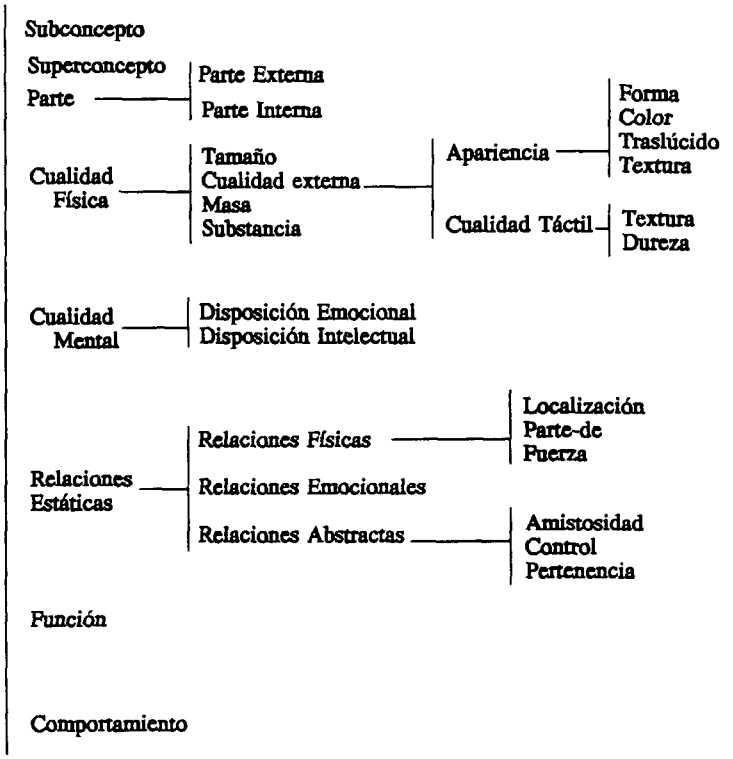
3.3 Jerarquías de Tópicos

Está claro que cualquier sistema diseñado para el razonamiento acerca de su mundo puede explotar eficientemente la herencia de propiedades mediante la generalización (is-a) o la agregación (part-of). Lo que complica este problema es que las entidades conceptuales consisten de múltiples componentes como las partes de un objeto, los participantes de una acción, etc...

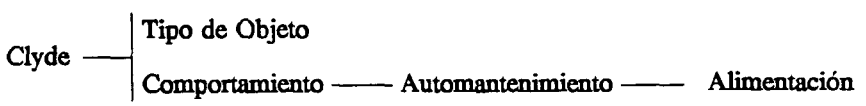
Aún más, el acceso al conocimiento del elefante a través de la herencia no garantiza una respuesta rápida o un chequeo de consistencia, debido al gran volumen de información que conocemos acerca de los elefantes.

Para solventar problemas como este, se dispone de dos características adicionales. Una es la necesidad de clasificar la proposiciones tópicamente como colores, localizaciones, tamaños, etc.. Este esquema de clasificación ayudaría a evitar la búsqueda exhaustiva de combinaciones de proposiciones que apoyen una determinada solución. La otra es la necesidad de un acceso a sólo aquellas proposiciones acerca de un concepto que pertenezcan a uno de los tópicos superiores. Se define un tópico como un predicado sobre parejas de proposiciones-conceptos. (ej. concepto: elefante, proposición: los elefantes son grises, tópico: color). Estos tópicos se relacionan por enlaces de subtópicos y supertópicos, formando todos ellos una jerarquía. Estas son un intento de clasificar el conocimiento acerca de objetos físicos con el mínimo de solape entre ellos, tal como muestra la figura de la página siguiente.

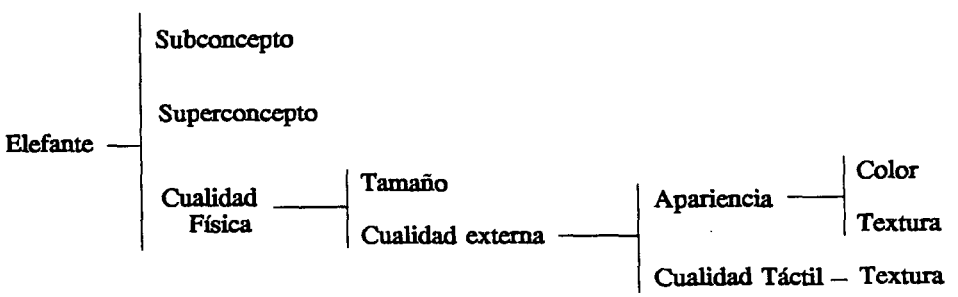
Una vez que se ha definido una jerarquía de tópicos para un tipo particular de nodos, se organizan las proposiciones asociadas con cualquier nodo de dicho tipo de acuerdo con la jerarquía. Esto se consigue superponiendo una estructura de acceso denominada *estructura de acceso a tópicos* sobre la proposición asociada. La estructura de acceso imita una parte de la jerarquía de tópicos, es decir, a aquella parte necesaria para suministrar caminos de acceso a todas las proposiciones disponibles acerca del nodo, cuando estas están asociadas con los correspondientes tópicos terminales.



Por ejemplo, si los únicos hechos conocidos acerca de Clyde son que es un elefante, que le gustan manises, estos se insertarían en la estructura de acceso



Si se sabe que los elefantes son muy grandes, grises, rugosos, y que Clyde es un elefante, estos hechos se asociarían con la estructura de acceso.



Las redes organizadas tópicamente facilitan distintos tipos de inferencias acerca de objetos y sus tipos. Existen algoritmos de clasificación automática de tópicos, y de optimización de tiempo y espacio.

3.4 Redes Proposicionales

Este formalismo se debe a Shapiro (1979) y es la culminación del trabajo que realizó sobre redes semánticas durante casi una década. El lenguaje SNePS proporciona facilidades para definir etiquetas de arcos, añadir información, examinar estructuras de red, eliminar nodos, encontrar nodos, y realizar una inferencia deductiva sobre los nodos. SNePS no es una red semántica particular, y por lo tanto pueden implementarse fácilmente diferentes notaciones de redes semánticas, ya que SNePS proporciona los mecanismos básicos para la construcción de dichas redes. Ha existido una gran polémica en relación a si estas redes semánticas son sólo variantes del cálculo de predicados, o si ofrecen algo más allá que un esquema de primer orden.

4 Representaciones procedurales

El conocimiento también se puede almacenar en forma de procedimientos en lugar que en forma de proposiciones. El conocimiento incluido en procedimientos puede accederse o directamente o utilizando alguna técnica de reconocimiento de patrones. Los partidarios de la representación del conocimiento con medios procedurales resaltan la similaridad que existe entre la forma en la que la gente utiliza el conocimiento que posee, y la variedad de actividades que realiza, y su comportamiento intelectual es debido en gran medida a los *procedimientos* que poseen para llevar a cabo sus actividades.

La representación procedural considera una base de conocimiento como un conjunto de agentes o procesos activos. Uno de los sistemas más importantes es el PLANNER (Hewitt 1972), que introdujo la noción de *invocación directa de procedimientos*. Una base de conocimiento se ve como una base de datos de afirmaciones y un conjunto de *teoremas* (o demonios) que vigilan y se activan cuando se modifica o se busca en la base de datos. Cada teorema tiene un patrón asociado que, cuando se activa el teorema, se compara con los datos que se van a insertar, o eliminar, o recuperar de la base de datos. Si la comparación es exitosa, se ejecuta el teorema. De esta forma el mecanismo usual de llamada se sustituye por uno en el que los procedimientos se llaman cuando se satisface una condición.

Los sistemas de producción son en cierta medida bastante similares al PLANNER. Una base de conocimiento es un conjunto de reglas de producción y una base de datos global. Las reglas, al igual que los teoremas, se componen de un patrón y de un cuerpo que implica una o más acciones. La base de datos comienza en algún estado inicial, y se intentan las reglas en algún orden pre-especificado hasta que se encuentra alguna en la que se verifiquen sus condiciones. Entonces se ejecuta el cuerpo de dicha regla, y continúa la búsqueda de otras.

De igual manera existen una serie de diferencias entre el mecanismo de activación de un teorema de PLANNER y una regla de un sistema de producción. El orden en el que se comparan los patrones de un teorema está indeterminado en PLANNER. Los sistemas de producción normales poseen una ordenación de las reglas fija que determina cuando se compara cada regla con la base de datos. Otra diferencia importante es que los teoremas pueden llamar directamente otros teoremas, mientras que los sistemas de producción sólo lo pueden hacer indirectamente por medio de la afirmación de la información apropiada en la

base de datos.

En relación con las estructuras de control, existen una serie de propuestas que extienden o modifican la estructura de control jerárquica de LISP o ALGOL. Una base de conocimiento en un sistema de producción consiste de una colección de reglas muy poco dependientes, lo que facilita la comprensión y modificación de dichas bases de conocimiento. La estructura de control de los teoremas de PLANNER utiliza *backtracking*, y cuando se ejecuta el cuerpo de un teorema y falla al conseguir una meta predeterminada, se eliminan los efectos colaterales del teorema fallido, y se intentan otros hasta que se encuentre uno que funcione. Se ha demostrado el excesivo costo de esta estrategia, y debe evitarse a cualquier precio.

La ventaja de los esquemas procedurales sobre otros tipos de esquemas consiste en que permiten la especificación de interacciones directas entre hechos, eliminando la necesidad de búsquedas infructuosas. Por contra, una base de conocimiento procedural, al igual que un programa, es difícil de entender y de modificar.

4.1 Winograd

Su tesis doctoral describió un sistema de procedimientos que interpretaban y generaban una gran variedad de sentencias del lenguaje natural que servían para apilar y desapilar bloques. El sistema, basado en el esquema funcional del PLANNER, reconocía y operaba correctamente con frases como

Pick up the big red block and put it in the blue box

El trabajo de Winograd inició la controversia entre proceduralistas y declarativistas (entre saber *como* y saber *que*). Los primeros afirman que nuestro conocimiento es básicamente saber como, es decir, que es posible utilizar procedimientos como lenguaje de representación, mientras que los declarativistas creen que existe un conjunto bastante general de procedimientos para manipular hechos concretos que describen determinado conocimiento del dominio.

4.2 Redes Semánticas Procedurales

Dos de los mayores problemas de las redes semánticas, que en el mejor de los casos se interpretan en términos de la lógica de primer orden, son: primero, la especificación de

conceptos cuyas propiedades cambien con el tiempo. El formalismo desarrollado por Shubert en 1976 soluciona este problema aunque de una manera un tanto ineficiente. El otro problema, más grave, está relacionado con el mecanismo de control a utilizar con dicha representación: por ejemplo, no se distingue entre una regla de inferencia que *pueda* utilizarse de la que *debería* usarse. Para solucionar estos temas, se han desarrollado formalismos que integran las redes semánticas con programas.

Los componentes de una red semántica procedural incluyen clases, relaciones, y programas que son todos (pero no los únicos) objetos. Una clase, es una colección de objetos que comparten algunas propiedades; estos objetos se dice que son instancias de la clase, y ellos mismos pueden ser una clase. Una relación es un enlace entre dos clases. Las relaciones pueden considerarse clases con afirmaciones como instancias. Un programa es una clase cuyas instancias son los procesos y se corresponden con activaciones de programas. Existen cuatro operaciones básicas mediante la asignación de cuatro programas a cada clase o relación: *añadir* una instancia a una clase, *eliminar* una instancia de una clase, *buscar* una instancia de una clase, y *comprobar* la pertenencia de una instancia en una clase.

La red semántica se organiza en una jerarquía de abstracción mediante los mecanismos de generalización / especialización (IS-A), y de agregación / descomposición (PART-OF).

Los programas (procedimientos y funciones) especifican el comportamiento de las clases y de las relaciones. Estos se caracterizan por:

- *Prerequisito:* una sentencia lógica que debe ser verdadera para que se pueda ejecutar el cuerpo.
- *Cuerpo:* una acción para los procedimientos, o una expresión para las funciones.
- *Efecto:* Una acción a realizar después de que se ha ejecutado el cuerpo con éxito.
- *Disculpa:* una acción o una expresión a utilizar después de la finalización de un cuerpo sin éxito.

Como los programas son clases, pueden organizarse en una jerarquía de IS-A y beneficiase de las reglas de herencia.

Levesque ha identificado las ventajas de su formalización de redes semánticas, en particular de la utilidad de incluir procedimientos directamente en la red. Entre ellas se encuentran:

- La utilización de programas proporciona una fundamentación semántica de las redes semánticas.
- Las jerarquías IS-A, y las jerarquías de instancias son útiles y proporcionan reglas explícitas para la herencia.
- Se distingue entre propiedades estructurales y afirmacionales, y se aprovechan sus características.
- Se introduce el concepto de metaclase como método para representar determinados aspectos de la representación dentro de la misma representación.
- Se integran programas activamente dentro de la representación en forma de clases con las mismas características (ej. herencia).

5 Programación lógica

Los lenguajes de programación como PROLOG son el resultado de la combinación de la lógica y de los procedimientos. En esta aproximación, los esquemas de representación procedurales y lógicos se combinan en una forma: *programas lógicos*

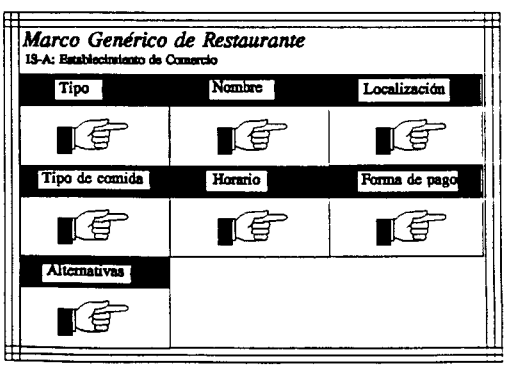
La potencia de la programación lógica deriva del hecho de que las sentencias declarativas del cálculo de predicados también son programas, una mezcla especialmente útil de estilos declarativo y procedimental en el que las capacidades de inferencia pueden alcanzarse sin una pérdida excesiva en su facilidad de comprensión. También existe una uniformidad en la representación entre el proceso que interpreta el lenguaje natural y la notación en la que se traduce el lenguaje, y es realmente fácil incorporar el contenido de información de la entrada en lenguaje natural a la base de conocimiento.

6 Marcos (Frames)

La noción de *marco o frame* ha jugado un papel crucial en la investigación sobre Representación del Conocimiento desde que Minsky publicó su propuesta de frames en 1975. Un frame es una estructura compleja de datos para representar una situación estereotípica, como la de estar en cierto tipo de habitación, o ir al cumpleaños de un niño. El marco tiene slots para los objetos que intervienen en la situación estereotípica así como las relaciones entre dichos slots. A cada frame se le asocia diferentes tipos de información, del tipo de como utilizarlo, qué hacer si pasa algo inesperado, valores por defecto para los slots, etc. Una base de conocimiento es un conjunto de marcos organizados en función de unos determinados ejes, u otros principios más débiles como la noción de *similaridad* entre dos frames.

La propuesta original de Minsky proporcionó básicamente una estructura para desarrollar un esquema de representación que combinaba las ideas de redes semánticas, de los esquemas procedurales, de lingüística, etc.

La siguiente figura representa un marco para un restaurante estereotípico. Los punteros representan slots a los que deben asignarse instancias específicas del tipo de restaurante, alternativas, localizaciones, y estilo de comida. La mayoría de los formalismos de frames proporcionan una variedad de mecanismos para ayudar a completar los slots, que varían desde valores preespecificados por defecto hasta otras capacidades de inferencia más complejas. Una vez que se ha activado un frame, podrían realizarse múltiples inferencias acerca del estado del mundo. Algunas de estas inferencias están contenidas en el propio frame; otras pueden hacerla a partir del hecho de que un frame está relacionado estáticamente con otros frames, casi siempre por la relación IS-A.



Parece que la principal diferencia entre redes y frames radica en el hecho de que los conceptos de un frame son principalmente *funcionales* más que *estructurales*. Es decir, consideramos que una estructura de memoria es un frame por los tipos de conocimiento y las capacidades que se le atribuyen, más que por alguna propiedad estructural.

Los marcos organizan el conocimiento que representan según la función de dicho conocimiento. El siguiente ejemplo ilustra la organización funcional de los frames, especialmente el acceso asociativo de los slots de los frames. El conocimiento accedido vía slot puede ser codificado proceduralmente. La ejecución de dichos procedimientos proporciona una comprensión procedural del concepto y del contexto. En las frases:

Juan ganó la partida,
utilizó el oro.

Juan se compró un avión,
utilizó el oro.

el significado de *oro* es diferente en las dos frases, dependiendo de la activación del frame ganar-partida o comprar-aviones. Los frames contendrían slots para oro-partida y oro-avión, o slots a otros frames que expliquen oro-partida y oro-avión.

7 Arquitecturas de sistemas de producción

Fueron propuestas por Newell en 1973 como un modelo del razonamiento humano. Un conjunto de reglas de producción (cada uno básicamente una pareja patrón -> acción) opera sobre una memoria de corto plazo de conceptos. Un bucle de control intenta cada regla en sucesión, ejecutando la acción de aquella regla que verifique su antecedente. La base de datos de estos sistemas almacena información de estado del dominio de la aplicación, mientras que el sistema de producción especifica como efectuara cambios en el estado

8 Blackboards (Pizarras)

Las pizarras son sistemas que explotan la idea de 'agentes cooperantes'. Es decir, diversas fuentes de conocimiento que contienen el conocimiento necesario para resolver el problema. Un modulo de control decide cual de las fuentes de conocimiento activar en cada momento. Las fuentes de conocimiento suelen poseer distintas representaciones e inferencias, de forma que los módulos son verdaderos especialistas en determinados aspectos de la tarea. (ej. un módulo puede implementar un sistema de producción, otro una red semántica, ...). Los resultados que se vayan obteniendo se reflejan en una estructura de datos global que se denomina 'pizarra'. Los distintos agentes observan continuamente el contenido de la pizarra y cuando observan que se reúnen las condiciones necesarias para su activación, se ejecutan. Las conclusiones resultantes de su activación las volcarán a su vez en la pizarra.

9 Conclusiones

Se han tratado de comentar algunos de los paradigmas de representación de Inteligencia Artificial. Las características que permiten diferenciar entre distintos modelos son cada vez más ambiguas, debido a que las actuales tendencias integran distintas aproximaciones explotando las mejores características de ellas. Por último se expondrán algunos principios generales de representación del conocimiento:

- Debe poderse representar información de distintos tipo, incluyendo conocimiento del mundo, de las metas e intenciones, conocimiento del contexto, etc...

- La representación del conocimiento es relativística, el mejor esquema de representación del conocimiento depende a menudo de los requisitos particulares de la aplicación.

- El conocimiento debe ser representable a todos los niveles. No existe un nivel absoluto de primitivas que no pueda desdoblarse.

- Los procesos que manipulan un esquema de representación del conocimiento deben ejecutarse con recursos (espacio y tiempo) limitados.

- Un esquema de representación debe representarse explícitamente. Las aproximaciones ad-hoc dejaron de ser satisfactorias.

Bibliografía

AAAI-90. Proceedings Eighth National Conference On Artificial Intelligence

Brachman. Levesque. Readings in Knowlege Representation. ed Morgan Kaufman 1985

Brodie, Michael. On Conceptual Modelling. ed. Springer Verlag

Cercone, Nick. The Knowledge Frontier. ed. Springer Verlag 1987

Hayes-Roth, Frederick. Waterman, Donald. Lenat. Building Expert Systems. ed. Addison Wesley 1983

CAPÍTULO II

Estrategias de Resolución de Problemas

Índice

1 Introducción 1

2 Pequeño espacio de búsqueda con conocimiento y datos fiables 3

3 Datos o conocimiento no fiable 5

4 Datos Variables en el Tiempo 7

5 Espacio de Soluciones Grande y Factorizable 9

6 No existe evaluador de Soluciones Parciales 13

7 No existe una ordenación fija de los Subproblemas 15

8 Subproblemas Interactuantes 17

9 Se precisa la Suposición 19

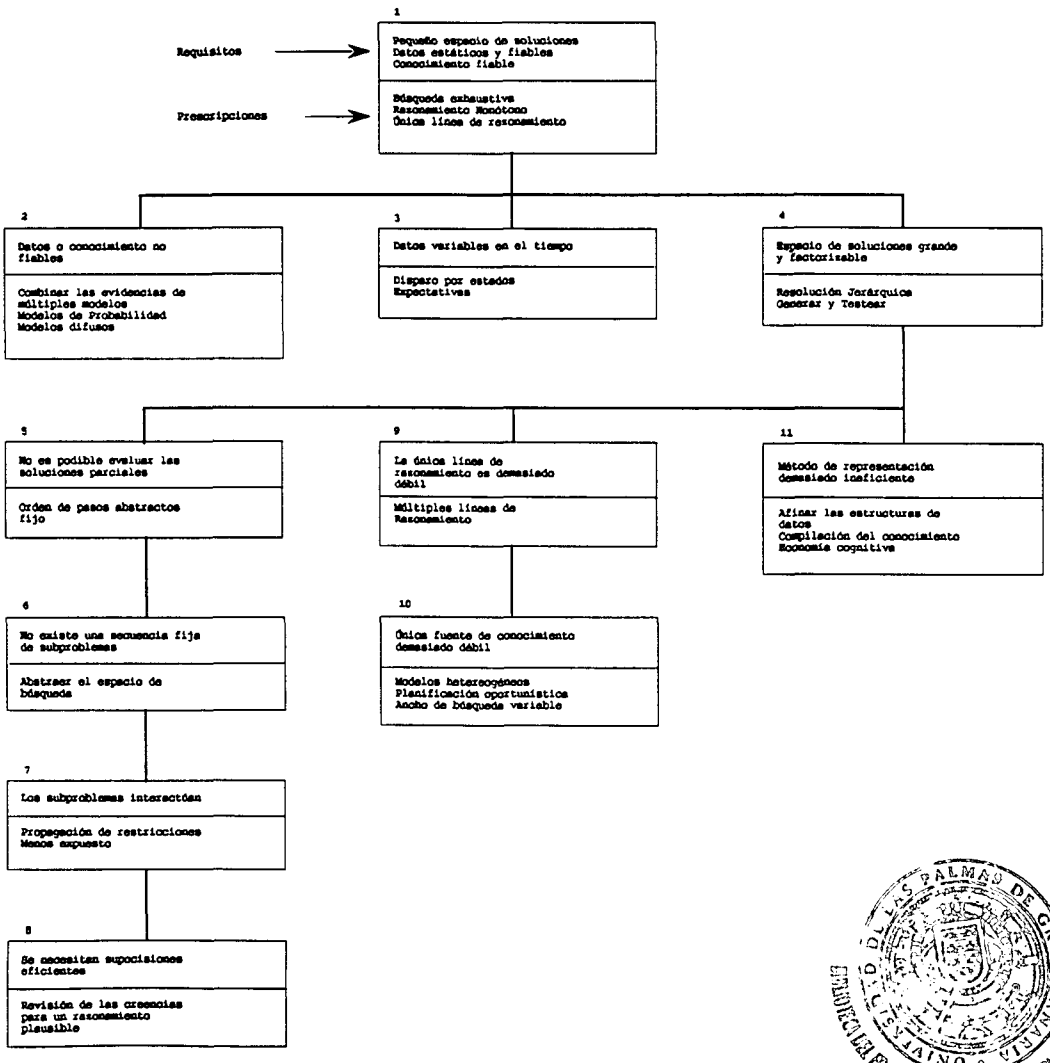
10 Una línea de razonamiento es insuficiente 21

11 Una única fuente de conocimiento es insuficiente 23

Bibliografía 25

1 Introducción

Una de las características más valiosas de los sistemas expertos es la forma en la que buscan soluciones. Diferentes características del dominio de la aplicación influyen sobre la elección del método de búsqueda, como son, entre otras, el tamaño del espacio de soluciones, los posibles errores en los datos, o la disponibilidad de abstracciones. La inferencia es el corazón del sistema de razonamiento, y cualquier fallo en su organización puede desembocar en sistemas de resolución que, en el mejor de los casos serán ineficientes. Como consecuencia, la búsqueda es uno de los tópicos más estudiados en IA. La siguiente figura contiene un diagrama con los distintos casos que se comentarán a continuación.



2 Pequeño espacio de búsqueda con conocimiento y datos fiables

Una arquitectura muy simple que ha sido utilizada en sistemas relativamente simples es la que reúne los siguientes requisitos:

- Los datos y el conocimiento son fiables.
- Los datos y el conocimiento es estático.
- El espacio de soluciones posibles es pequeño.

Aparentemente, estas restricciones son ingenuas. De hecho, mucha gente cree que la mayoría de los problemas satisfacen dichos requisitos, aunque casi la totalidad de las situaciones reales no los verifiquen.

El primer requisito es que el dato y el conocimiento sean fiables, es decir, sin errores ni ruidos. No pueden existir señales extrañas ni se pueden perder. Pocas fuentes de datos de aplicaciones reales verifican estos requisitos. El conocimiento fiable es el que es aplicable sin tener en cuenta la consistencia o correctitud. La aplicación sistemática de dicho conocimiento no debe llevar a conclusiones falsas, aproximadas, o tentativas. La principal ventaja de la fiabilidad de los datos y del conocimiento es la monotonicidad del sistema; es suficiente con desarrollar una única línea de razonamiento. Es decir, no es necesario desarrollar múltiples argumentos en favor de las conclusiones potenciales. Si fuera aplicable más de una regla de inferencia en un momento determinado, el orden en el se aplicarían no sería importante.

El segundo requisito está destinado a evitar el problema con datos que son dependientes del tiempo. Esto significa que el sistema no tiene que preocuparse de invalidar hechos a medida que pasa el tiempo. Un espacio de búsqueda pequeño implica que no van existir problemas con los recursos computacionales. Un número que podría ser máximo podría ser de 10!. Si se necesitaran 25 milisegundos para considerar una solución, entonces se tardaría 24 horas en considerar 10! soluciones. Sorprendentemente este límite es realmente bajo.

Una organización para resolver estos problemas tiene dos partes principales: la memoria y el método de inferencia. La organización de memoria más simple consistiría en una lista de creencias o hechos. Muchas veces esta organización se limita al cálculo de predicados.

3 Datos o conocimiento no fiable

Cuando el experto tiene que hacer valoraciones bajo algún tipo de presión puede que el conocimiento que utiliza no sea totalmente fiable. El problema de hacer derivaciones a partir de datos inciertos o incompletos ha generado una serie de aproximaciones técnicas.

Una de las primeras y más simples la constituye el sistema MYCIN. Este utilizaba un modelo de implicaciones por aproximaciones utilizando números llamados *factores de certeza* para indicar la fuerza de una regla heurística, o de alguna conclusión del sistema. Estos factores se combinan en los antecedentes de las reglas de forma que se va realizando una propagación ponderada por la cadena de inferencia. El sistema realizará una minimización cuando los antecedentes de una regla estén relacionados con la función AND, y una maximización cuando los antecedentes estén relacionados por la función OR. Estas reglas de combinación han demostrado poseer ciertas propiedades, como son la insensibilidad frente al orden en el que se aplican las reglas.

Una cuestión en relación a estas aproximaciones consiste en si son necesariamente *ad hoc*. MYCIN ha recibido críticas en el sentido de que introduce su propio mecanismo de razonamiento con incertidumbre cuando están disponibles las aproximaciones probabilísticas. Por ejemplo, se podría utilizar la fórmula de Bayes para calcular la probabilidad de alguna enfermedad a partir de las evidencias especificadas a partir de las probabilidades a priori de las enfermedades y de las probabilidades condicionadas de las observaciones con las enfermedades. La principal dificultad con el método Bayesiano radica en el gran número de datos necesarios para determinar las probabilidades condicionadas en la fórmula. La cantidad de datos es tan ilimitada que normalmente se asume la independencia condicional de las observaciones. Se podría argumentar que tal suposición de independencia deslegitima el modelo estadístico.

Otra aproximación al razonamiento inexacto que diverge de la lógica clásica es la lógica difusa creada por Zadeh (1979). En lógica difusa una sentencia como *"X es un número grande"* se interpreta como una denotación imprecisa caracterizada por un conjunto difuso. Un conjunto difuso es un conjunto de valores con los valores de posibilidad que se muestran a continuación:

Proposición Difusa: X es un número grande
Conjunto difuso asociado: (X ϵ (0,10), .1)
(X ϵ (10,1000), .2)
({X > 1000}, .7)

Otras aproximaciones intentan trabajar con información parcial a la vez que mantienen la exactitud del cálculo de predicados. Los sistemas basados en la revisión admiten la especificación explícita de información incompleta, a partir de reglas que definen unas preferencias por un tipo de información sobre otro. Estas reglas de corrección de datos pueden razonar con información parcial sin comprometer la exactitud del cálculo de predicados. Estos métodos dependen de la formalización del metaconocimiento que permite corregir los datos, retraer suposiciones pasadas, o combinar evidencias. La disponibilidad de ese metaconocimiento es un factor crítico en la viabilidad de estas aproximaciones.

4 Datos Variables en el Tiempo

Algunas tareas de sistemas expertos necesitan con de situaciones que cambian con el tiempo. Una de las primeras aproximaciones que tuvo esto en cuenta es el cálculo situacional de McCarthy y Hayes para representar secuencias de acciones y sus efectos. La idea principal es la de incluir *situaciones* junto con los otros objetos modelados en el dominio. Por ejemplo, la fórmula

(SOBRE bloque-1 mesa-2 situación-2)

podría representar el hecho de que en la situación 2 el bloque 1 está sobre la mesa 2. Una característica clave de esta información es que las situaciones son discretas. Esta discretitud manifiesta el uso del cálculo para la planificación de robots. Después de cada acción el estado del mundo del robot se modela por otra situación. En esta representación una situación variable puede tener situaciones como valores. En algunas implementaciones, la situación actual variable está implícita en la indexación de fórmulas asociadas con las situaciones.

Las acciones se representan por funciones cuyos dominios y rangos son situaciones. Para cada acción, un conjunto de axiomas de los frames caracteriza el conjunto de afirmaciones (el frame) que permanecen fijas mientras tiene lugar una acción dentro de ella. Cuando se planifica la acción de un robot, un ejemplo de acción sería mover un objeto a otro lugar. Un axioma de frame para Mover sería que todos los objetos que no se han movido explícitamente se dejan en su posición original.

En algunas aplicaciones, la representación de las situaciones que cambian vienen determinadas por datos que evolucionan en el tiempo. Ejemplos son los sistemas que controlan la evolución del estado de un paciente en una Unidad de Vigilancia Intensiva. El sistema VM controla el progreso post operatorio de un paciente regulando la ventilación automática que este recibe.

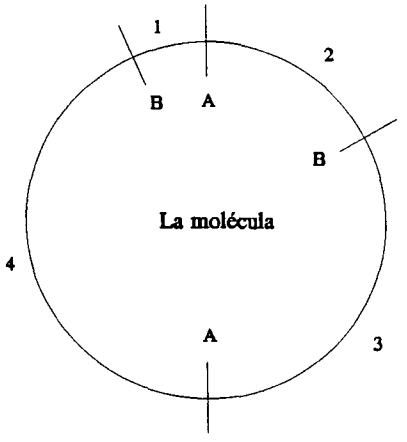
La limitación del sistema VM radica en que el razonamiento temporal está limitado a intervalos adyacentes de tiempo, considerando sólo el estado previo y el siguiente. Los programas que necesitan razonar con eventos más distantes requieren representaciones más elaboradas de eventos y de tiempo. Por ejemplo, la planificación y la predicción requiere que el cálculo situacional considere múltiples futuros posibles con operaciones indeterminadas, conjuntos de eventos futuros posibles no ordenados, y las acciones posibles de múltiples actores indeterminados.

5 Espacio de Soluciones Grande y Factorizable

En muchas tareas de análisis de datos es deseable encontrar todas las interpretaciones que sean consistentes con los datos. Esta actitud es la usual en aplicaciones de alto riesgo como el análisis de sustancias venenosas en la diagnosis médica. Una aproximación sistemática sería considerar todos los casos posibles y eliminar aquellos que sean inconsistentes con los datos. La dificultad radica en que no existe una manera sencilla de considerar todas las soluciones posibles.

El programa DENDRAL es el más conocido de los que razonan por eliminación (utilizando generar y testar -- generate-and-test). Funciona porque incorpora la poda en las primeras fases del ciclo de generar y testar. Para explicar algunas cuestiones en relación con este método utilizaremos el GA1, similar al DENDRAL aunque más simple.

El GA1 es un programa de interpretación de datos que infiere una estructura molecular completa a partir de mediciones de piezas de moléculas. La siguiente figura muestra un ejemplo del tipo de datos que posee el GA1 acerca de una molécula. La parte superior de la figura muestra que la molécula está formada de segmentos de longitud medible. Las líneas A y B que cortan el círculo indican los sitios donde se enclavan los encimas (llamados A y B respectivamente). Todas las moléculas en GA1 son circulares o lineales, lo que significa que todos los segmentos moleculares pueden organizarse linealmente. Cuando un encima digiere una muestra de moléculas, las piezas se liberan y sus tamaños pueden medirse. La meta consiste en inferir la estructura de la molécula original a partir de los datos de la digestión, y usualmente más de una estructura molecular es consistente con los datos disponibles.



Una tarea primordial en problemas como este es el de crear un generador viable de todas las soluciones posibles (todas las moléculas posibles) En GA1 el primer paso consiste en aplicar reglas de corrección de reglas y determinar un conjunto inicial de restricciones para el generador, un conjunto inicial de anclajes de encimas para poder construir moléculas candidatas. El proceso de generación combina dichos segmentos y anclajes, y comprueba si dichas combinaciones se ajustan a las evidencias. Por ejemplo, las siguientes listas (entre otras) se corresponden con la molécula de la figura anterior.

(1 A 2 B 3 A 4 B)
(2 B 3 A 4 B 1 A)
(1 B 4 A 3 B 2 A)

Estas representaciones equivalentes pueden generarse comenzando en cualquiera de los cuatro segmentos de la figura, lo que proporciona ocho representaciones equivalentes. Se dice que un generador no es redundante si produce exactamente una de las representaciones equivalentes de una solución durante el proceso de generación. GA1 lo hace incorporando reglas que podan las estructuras redundantes durante el proceso de generación como la siguiente:

Si se están generando estructuras circulares,
entonces sólo se utilizará como segmento inicial el segmento más pequeño de los existentes en la lista de segmentos.

La clave de la utilización efectiva de generar-y-testar es podar clases de candidatos inconsistentes lo más pronto posible. Por ejemplo, si se generara la siguiente estructura

(1 B 2 B)

GA1 tomaría esto como una descripción de todas las moléculas que reconocen el patrón

(1 B 2 B <segmento> <lugar> <segmento> <lugar>)

donde los segmentos y lugares restantes pueden completarse en el patrón. Es fácil de ver que ninguna molécula es consistente con los datos de la figura anterior.

GA1 funciona con problemas donde el número de soluciones candidatas es de miles de millones. Las reglas de poda permiten que la solución se alcance en unos pocos segundos. Después de que termina el generador, el número de soluciones candidatas se restringe a veinte o treinta. Sólo una o dos serán consistentes con los datos. Sería posible añadir más reglas de poda para que sólo quedara una solución consistente, pero las reglas tendrían una

complejidad y una especialización cada vez mayor, lo que dificultaría la comprobación de la validez de las mismas.

En resumen, generar-y-testar es un método a considerar cuando es importante encontrar todas las soluciones a un problema. Sin embargo, para que el método sea abordable el generador debe particionar el espacio de soluciones de forma que permita la poda en los momentos iniciales. Estos criterios se satisfacen normalmente en problemas de interpretación de datos y de diagnóstico.

6 No existe evaluador de Soluciones Parciales

El método generar-y-testar es la última alternativa para muchos sistemas que poseen grandes espacios de búsqueda. La dificultad más usual radica en que no es posible realizar la poda inicial del espacio, como en las tareas de diseño o planificación. Normalmente no es posible decir a partir de un fragmento de un plan o de un diseño si dicho fragmento forma parte de una solución satisfactoria.

Para abordar estos tipos de problemas, la situación más simple consiste en particionar el espacio abstracto (abstracción) común a todos los problemas de la aplicación. El caso se ilustra con el R1 (o XCON) que configura sistemas VAX. Su entrada son las peticiones de un cliente, y la salida consiste en un conjunto de diagramas que muestran las relaciones espaciales entre los elementos del diseño. R1 debe determinar una configuración espacial para los componentes y añadir cualquier componente que sea necesario.

La tarea de configuración puede verse como una jerarquía de subtareas con fuertes interdependencias temporales. R1 divide la tarea de configuración en las seis subtareas siguientes:

1• Determinar si existe algo incongruente en la orden de pedido del cliente (elementos no relacionables, falta alguno de principales prerequisitos, etc.).
2• Situar los componentes apropiados en la unidad central de proceso y en sus ranuras de expansión.
3• Situar las cajas en la ranura de expansión del UNIBUS y poner los componentes apropiados en dichas cajas.
4• Situar los paneles en la ranura de expansión del UNIBUS.
5• Diseñar la planta del sistema.
6• Realizar el cableado.

R1 tiene un conjunto de reglas específicas asociadas a cada subtarea, hasta un total de unas 800 reglas. Definen las situaciones en las que determinada configuración parcial debe expandirse para formar una configuración aceptable. Otras reglas describen las relaciones temporales entre las subtareas mediante la determinación de su ordenación.

R1 explora el espacio de configuraciones posibles con las operaciones básicas de crear y extender configuraciones parciales. El algoritmo explora este espacio comenzando en un estado inicial, siguiendo por estados intermedios, y terminando en un estado final, *sin realizar ninguna vuelta atrás (backtracking)*. R1 procede por sus seis tareas principales siempre en el mismo orden para todos los problemas.

El requerimiento clave de que el algoritmo sea viable es que no exista backtracking. Esto significa que cualquier estado intermedio de R1 debe ser capaz de determinar si el estado pertenece al camino de la solución. Esto requiere de la existencia de una ordenación parcial de las decisiones para la tarea de forma que las consecuencias de aplicar un operador recaigan solamente en las últimas partes de la solución.

7 No existe una ordenación fija de los Subproblemas

En aplicaciones, como el errand-running (de Hayes-Roth 1978), con una mayor gama de problemas no es posible aplicar el esquema anterior. El refinamiento descendente (top-down) considera una abstracción para ajustarla a cada problema. Los aspectos importantes de esta aproximación son los siguientes:

- Las abstracciones para cada problema se componen de términos (seleccionados de un espacio de términos) para ajustarlos a la estructura del problema.
- Estos conceptos representan soluciones parciales que se combinan y se evalúan durante la resolución del problema.
- Se asignan a los conceptos niveles de abstracción fijos y predeterminados.
- La solución del problema procede desde la cima hacia abajo, es decir, desde lo más abstracto hacia lo más específico.
- Las soluciones al problema se completan en un nivel antes de pasar al siguiente nivel.
- Dentro de cada subnivel los subproblemas se resuelven en un orden independiente del problema. (esto proporciona una ordenación parcial de los estados abstractos intermedios.)

El mejor ejemplo de esta aproximación es el ABSTRIPS. Este realiza una planificación de los movimientos que debería realizar un robot que moviera objetos entre habitaciones. Las abstracciones son planes que difieren en el nivel de detalle que utilizan para especificar las precondiciones de los operadores. Este nivel de detalle se indica mediante la asociación de un número con todos los literales utilizados en las precondiciones. Por ejemplo, Sacerdoti sugirió las siguientes asignaciones para el dominio de planificación de robots:

Tipo y Color	*4*
EnHabitación	*3*
Enchufado Desenchufado	*2*
JuntoA	*1*

En este ejemplo, los predicados para el Tipo y el Color de los objetos tienen prioridades altas (críticas), ya que el robot no posee ningún operador para modificarlos. Estos predicados, junto con el conjunto de acciones del robot se combinan para formar planes para

resolver problemas particulares; el espacio de todos los planes posibles es el conjunto de todos los planes que pueden construirse a partir de dichas piezas. Los planes más abstractos son los que sólo incluyen conceptos con prioridades altas o críticas.

La planificación comienza fijando la criticalidad a un máximo. Dentro de cada nivel se planifica hacia atrás a partir de metas. Las precondiciones cuya criticalidad sea menor que el nivel actual son invisibles al planificador, ya que se supone que se tendrán en cuenta en una fase posterior. Después de que se completa un plan en un deteminado nivel, se decrementa el nivel de criticalidad y se procede al siguiente nivel. La versión abstracta previa del plan se utiliza para guiar la elaboración del plan. Por ejemplo, una versión previa del plan podría haber determinado la ruta que seguirá el robot entre habitaciones. En una versión más detallada del plan se incluirán pasos para abrir y cerrar puertas.

8 Subproblemas Interactuantes

Una dificultad básica con el refinamiento descendente es su falta de realimentación durante el proceso de resolución. Se presupone que los mismos tipos de decisiones se harán en el mismo punto para cada problema en el dominio. Una aproximación denominada *principio del menos-expuesto (least commitment)* se basa en un principio diferente para guiar el proceso de razonamiento. La idea básica es la de que las decisiones no deben tomarse arbitraria o prematuramente sino que deben posponerse hasta que se disponga de información suficiente.

El razonamiento basado en este principio requiere de las siguientes habilidades:

- La habilidad de saber cuando existe información suficiente para poder tomar una decisión.
- La habilidad de suspender la actividad de resolución en un subproblema cuando no se dispone de información suficiente.
- La habilidad de moverse entre subproblemas, reemprendiendo el trabajo a medida que se dispone de información.
- La habilidad de combinar información de subproblemas diferentes.

Un ejemplo de este estilo de razonamiento es el MOLGEN. Es un sistema experto para diseñar experimentos genéticos. Su arquitectura implica la representación de interacciones entre subproblemas como restricciones y el descubrimiento de interacciones entre subproblemas mediante la propagación de restricciones. Utiliza operadores explícitos de metanivel (en oposición a las operaciones específicas del dominio) para razonar con restricciones; alterna las estrategias de menos-expuesto y heurísticas durante la resolución.

En la estrategia del menos expuesto, MOLGEN selecciona una opción sólo cuando sus restricciones disponibles han definido suficientemente las alternativas. Sus operadores de resolución de problemas pueden suspenderse de forma que una decisión puede posponerse. La propagación de restricciones es el mecanismo para mover información entre subproblemas, de forma que MOLGEN expande los planes oportunísticamente en respuesta a dicha propagación.

La alternacia entre las estrategias heurísticas y menos expuesta ilustra una interesante limitación de esta última. Con el principio del menos-expuesto, el proceso de resolución debe parar cuando debe elegirse una opción y no existe ningún factor de peso que permita elegir entre las alternativas. Esta situación se denomina *deadlock* porque las operaciones en todos los subproblemas están en el estado de "esperar por más restricciones". Cuando MOLGEN reconoce esta situación, cambia a la aproximación heurística y realiza una suposición. En muchas ocasiones esta funcionará, y el proceso de solución podrá continuar. Cuando se realice una suposición errónea el sistema puede caer en conflictos.

9 Se precisa la Suposición

La suposición o el razonamiento plausible es parte inherente de la búsqueda heurística. Algunos ejemplos de situaciones genéricas donde la suposición es importante son:

- Muchos sistemas de resolución de problemas necesitan trabajar con conocimiento incompleto y pueden ser incapaces de determinar la mejor elección en algún momento durante la resolución. En tales casos el sistema es incapaz de terminar sin realizar una suposición. Ejemplo es el deadlock del menos expuesto.
- Un espacio de búsqueda puede ser muy denso en soluciones. Si las soluciones son todas igualmente buenas y no es necesario tenerlas todas, las suposición puede ser conveniente.
- A veces se puede converger a soluciones con la mejora sistemática de aproximaciones (ej. estrategia descendente). Cuando se necesite una convergencia rápida, puede ser apropiado suponer incluso si las soluciones son escasas.

La dificultad con las suposiciones son identificar las suposiciones erróneas y recuperarse de ellas eficientemente. Cada vez que se realiza una suposición es necesario comprobar que esta sea consistente con los datos actuales del problema. Esto se hace normalmente manteniendo una serie de registros de dependencias de todas las deducciones.

Cuando se realiza una suposición errónea, es necesario recuperar al sistema de los efectos provocados por dicha suposición. La estrategia del sistema EL (computa los parámetros de un circuito eléctrico en un nodo a partir de la información de dichos parámetros en otro nodo) en estas situaciones es la siguiente:

1. Cuando se produce una contradicción, El necesita saber que rehacer. Sería ineficiente rehacer todas las suposiciones que son antecedentes de la afirmación contradictoria. Por lo tanto EL debe decidir cuales de las suposiciones es la mejor candidata y esto requiere de conocimiento específico del dominio.

2. EL debe rehacer algunos de los análisis de propagación. En ocasiones puede ser posible mantener algunas de las manipulaciones simbólicas que se habían hecho. (qué mantener y qué olvidar)

3. Se recuerdan las contradicciones de forma que no se vuelvan a intentar combinaciones de elecciones inconsistentes.

10 Una línea de razonamiento es insuficiente

Algunos sistemas utilizan múltiples líneas de razonamiento para ampliar el entorno de una búsqueda incompleta o combinar la potencia de distintos modelos

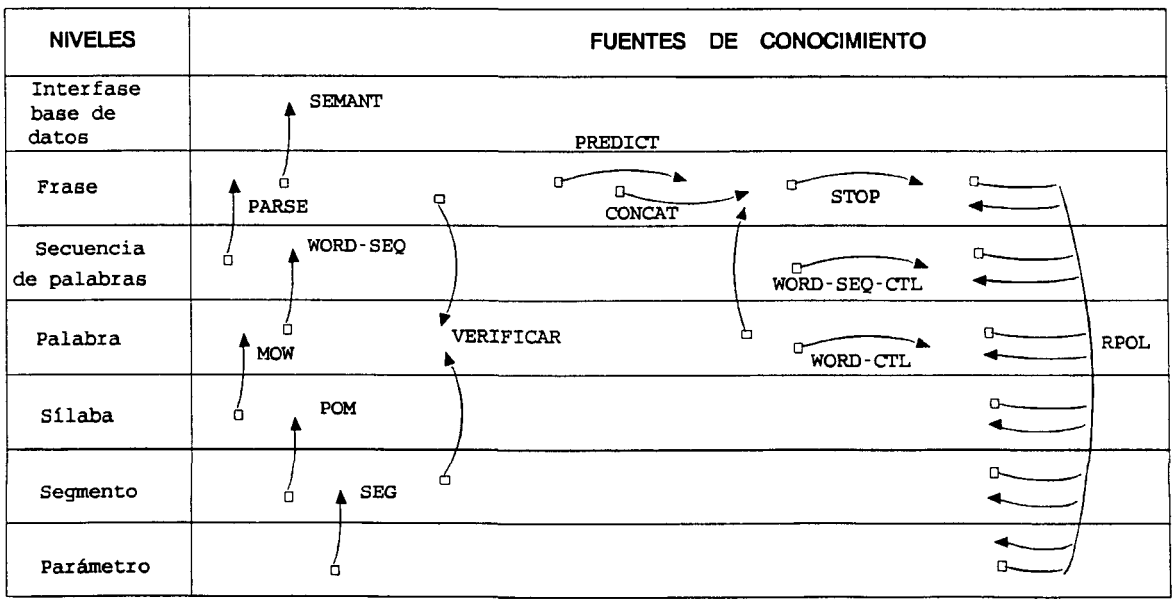
El sistema HEARSAY-II es un ejemplo del primer propósito. Es un sistema de comprensión del habla que es capaz de reconocer e interpretar peticiones de información de una base de datos utilizando un vocabulario aproximado de mil palabras. Cuando se enfrenta con las demandas conflictivas de buscar en un gran espacio de búsqueda con recursos computacionales limitados, HEARSAY-II realiza una búsqueda heurística e incompleta. En general, los sistemas que poseen evaluadores que fallan pueden disminuir las posibilidades de descartar una solución aceptable a partir de evidencias débiles llevando una serie de soluciones en paralelo.

11 Una única fuente de conocimiento es insuficiente

En relación con la utilización de múltiples líneas de razonamiento en la resolución de problema está el uso de múltiples fuentes de conocimiento. El HEARSAY-II coordina diversas fuentes de conocimiento utilizando un planificador oportunístico.

En HEARSAY-II el conocimiento se organiza como un conjunto de módulos interactivos denominados *fuentes de conocimiento* (ks). Las ks cooperan en la búsqueda de un espacio multinivel de soluciones parciales, extrayendo parámetros acústicos, clasificando segmentos acústicos en clases fonéticas, reconociendo palabras, frases, y generando y evaluando hipótesis acerca de palabras y sílabas indetectadas.

Como se muestra en la figura, las ks se comunican a través de una base de datos global denominada *pizarra*, con siete niveles de información. Las relaciones primarias entre los niveles son composicionales: las secuencias de palabras se componen de palabras, las palabras se componen de sílabas,... Las entidades en la pizarra son hipótesis. Cuando se activan las kss, crean y modifican estas hipótesis sobre la pizarra, registran los argumentos en favor entre niveles, y asignan puntuaciones de credibilidad.



Bibliografía

AAAI-90. Proceedings Eighth National Conference On Artificial Intelligence

Brachman. Levesque. Readings in Knowlege Representation. ed Morgan Kaufman 1985

Brodie, Michael. On Conceptual Modelling. ed. Springer Verlag

Cercone, Nick. The Knowledge Frontier. ed. Springer Verlag 1987

Hayes-Roth, Frederick. Waterman, Donald. Lenat. Building Expert Systems. ed. Addison Wesley 1983

CAPÍTULO III

RETE

Índice

1 Un Algoritmo de Reconocimiento Rápido 1

2 Resolución de Conflictos 3

3 Reconocimientos 5

4 El grafo del RETE 7

5 Nodos de Dos Entradas 9

6 Eliminando Test Comunes 11

Bibliografía 13

1 Un Algoritmo de Reconocimiento Rápido

El algoritmo de reconocimiento RETE es un método muy rápido para comparar un conjunto de patrones con un conjunto de objetos, determinando todos los reconocimientos (matching) posibles. Fue desarrollado por L. Forgy en 1974 en el Carnegie-Mellon, para gestionar operaciones inusualmente grandes en intérpretes de sistemas de producción. Ha sido implementado tanto en sistemas de investigación como comerciales.

A medida que los sistemas de producción han evolucionado, el RETE se ha ido revisando. La evidencia demuestra que el RETE es el algoritmo más eficiente desarrollado para sistemas monoprocesadores.

Un sistema de producción se compone de un conjunto no-ordenado de sentencias condicionales, o reglas, denominadas producciones. Las producciones operan sobre expresiones almacenadas en una base de datos global denominada *memoria de trabajo (working memory)*. Por ejemplo, en OPS5 (un lenguaje de sistema de producción que implementa el RETE), la memoria de trabajo se compone de cientos de objetos; cada objeto tiene asociado entre 10 y 100 parejas de atributo-valor. Un objeto, junto con sus parejas atributo-valor, es un elemento de la memoria de trabajo. Las producciones se almacenan en una memoria diferente -*la memoria de producciones.*

En general, la potencia de un sistema de producción aumenta a medida que se añaden más reglas de inferencia y aumenta la complejidad de las interacciones entre las reglas. Mientras que mejores reglas significa un sistema más efectivo, es más lento porque los sistemas de producción hacen un gran uso del reconocimiento de patrones entre los componentes de las reglas (las condiciones y sus acciones), y una memoria de trabajo dinámica. El sistema debe identificar las reglas que son las más relevantes o que dejan de serlo en cada instante, a medida que la memoria de trabajo cambia. Por lo tanto, debe poderse acceder a las reglas mediante valores de patrones reconocidos, y el reconocimiento de patrones es la operación más lenta que debe realizar un intérprete. Algunos sistemas dedican más del noventa por ciento del tiempo total de ejecución a realizar el reconocimiento.

Un sistema experto que sea algo más que trivial debe tener miles de reglas, por lo que las disponibilidad de intérpretes eficientes es crucial. Uno de esos sistemas es el R1 (XCON), que utiliza más de 5000 reglas como editor técnico para la configuración de sistemas VAX y PDP.

Pero el rendimiento de un sistema de producción no se mide en miles de reglas. Se mide por la compleja red que el sistema genera cuando se mueve por la tarea. Una única regla del XCON dará una idea del problema:

IF the most current active context is assigning a power supply,
and a UNIBUS adaptor has been put in a cabinet,
and the position it ocupies in the cabinet is known,
and there is space available in the cabinet for a power supply
.....
and there is an available power supply,
and there is no H7101 regulator available,

THEN add an H7101 regulator to the order.

Un intérprete de un sistema de producción ejecuta un sistema de producción realizando un ciclo de *reconocer y actuar*. Este ciclo se realiza en el OPS5 de la siguiente forma:

Resolución de Conflictos. Tiene como entrada el conjunto de conflictos (salida del RETE). Seleccionar una producción con una LHS (parte izquierda o condiciones) satisfecha. Si ninguna producción verifica su LHS, devolver el control al usuario.

Actuar. Realizar las acciones especificadas en la RHS (parte derecha o acciones) de la producción seleccionada.

Reconocer. Evaluar las LHSs de las producciones para determinar cuales se verifican dado el contenido actual de la memoria de trabajo. La salida es el conjunto de conflictos.

Si Parar, retornar el control al usuario o ir al paso 1.

2 Resolución de Conflictos

El RETE es un algoritmo que sirve para computar el conjunto de conflictos. Es decir, es un algoritmo para comparar un conjunto de sentencias LHS con un conjunto de elementos para descubrir todas las activaciones.

Las activaciones (instantations) son los objetos en el conjunto de conflictos. Una activación es una pareja ordenada compuesta por un nombre de producción y una lista de elementos de la memoria de trabajo que satisfacen la parte izquierda de la producción. Un conjunto de conflictos es una colección de parejas ordenadas de la forma:

<Nombre de la producción. Lista de elementos reconocidos por la parte izquierda de la regla>

Durante el paso de resolución de conflictos, el intérprete OPS5 examina el conjunto de conflictos para encontrar una activación que domine sobre todas las otras bajo un conjunto de reglas ordenadas que forman *la estrategia de resolución de conflictos:* OPS5 ofrece dos de dichas estrategias, LEX y MEA.

Lo que hacen esta dos estrategias es simple: primero evitan que una activación se ejecute más de una vez. También facilitan que el sistema de producción atienda a los datos más recientes de memoria. Una vez que el sistema comienza una subtarea, es poco probable que sea distraído por otros datos remanentes de tareas anteriores.

Las estrategias de resolución de conflictos del OPS5 dan preferencia a las reglas con partes izquierdas más específicas. Ya que las reglas con condiciones más específicas se satisfacen en muchas menos ocasiones, es más probable que sean las apropiadas para disparar cuando se verifiquen sus condiciones.

Estas tres cosas son importantes porque facilitan la adición de producciones a un conjunto existente de reglas, y que las nuevas se disparen en el momento oportuno. También facilitan la simulación de las construcciones usuales de control como los bucles, y las llamadas a subrutinas.

Durante la fase de acción del ciclo de reconocer-actuar, se ejecutan las acciones de la producción seleccionada una cada vez, en el orden en el que están escritas. Las acciones

tienen efecto inmediatamente.

Durante la fase de reconocimiento, el intérprete determina cada activación de cada producción. Es decir, encuentra cada producción activada, y si alguna producción puede activarse con más de una lista de elementos de la memoria de trabajo, encuentra todas las listas de elementos. Todas las activaciones se sitúan en el conjunto de conflictos.

El conjunto de conflictos se compone de parejas ordenadas cuyo primer elemento es el nombre de una regla, y cuyo segundo elemento es una lista de elementos de la memoria de trabajo. La lista de elementos de la memoria de trabajo en el orden mostrado satisfacen los patrones *no-negados* de la regla. Dadas las restricciones impuestas por la variable en la parte izquierda de la regla, ningún elemento de la memoria de trabajo reconoce alguno de los patrones negados.

La operación de reconocimiento es difícil por la cantidad de datos que deben procesarse. Un sistema típico contendrá desde cientos hasta unos pocos miles de reglas, un número similar de elementos de la memoria de trabajo, y, normalmente, una media de tres hasta diez patrones (negados y no negados) por regla.

3 Reconocimientos

El RETE explota dos propiedades comunes a todos los sistemas de producción para reducir el esfuerzo de realización del reconocimiento. En estos sistemas el contenido de la memoria de trabajo es muy grande y cambia muy poco después de cada ejecución. Las partes izquierdas de las reglas contienen patrones muy similares o incluso iguales.

Ya que muy poca información de la memoria de trabajo cambia después de cada ciclo (la ejecución de una regla), gran parte de la información de la que utiliza el proceso de reconocimiento para un ciclo puede utilizarse para el siguiente. RETE se aprovecha de esto registrando la información de las partes IF y de reconocimientos parciales de ciclo a ciclo, actualizando la información cuando sea necesario para que se reflejen los cambios sobre la memoria de trabajo en cada ciclo. Así, la cantidad de esfuerzo utilizado por el reconocedor depende principalmente de la tasa de cambios sobre la memoria de trabajo en lugar del tamaño absoluto de la memoria de trabajo.

Para aprovechar el que muchas reglas comparten muchas condiciones, RETE procesa los patrones antes de que el sistema se interprete. El proceso es similar a la eliminación de subexpresiones que usualmente se utiliza en los compiladores de los lenguajes tradicionales. Antes de que se ejecute un sistema de producción, un compilador de reglas lo procesa, localizando los términos comunes y eliminando tantos de ellos como sea posible. Esto permite que muchas operaciones sobre el conjunto total de reglas se realicen sólo una vez.

Dichos patrones se compilan sobre un grafo especial de flujo de datos denominada *red rete*, que consiste en una red ordenada en forma de árbol o índice para las producciones. Los nodos en el grafo especifican las computaciones a realizar por el reconocedor. Los enlaces indican como fluyen los datos de nodo a nodo por el grafo. Las computaciones redundantes se eliminan construyendo un único nodo para una determinada computación, enlazando el nodo con todos los lugares donde se necesite.

El grafo sirve como una función para reflejar los cambios en la memoria de trabajo en cambios en el conjunto de producciones satisfechas:

(delta-wm) -> GRAFO -> (delta-reglas-satisfechas)

Un nodo en el grafo se denomina como el nodo raíz. Cada vez que se añade o se elimina un elemento de la memoria de trabajo, se crea una estructura de datos llamada *testigo* para describir el cambio. Un testigo es una pareja ordenada compuesta de una marca y de una lista de elementos de datos. En la implementación más sencilla del algoritmo sólo se necesitan dos marcas, + y -, para indicar que se ha añadido o eliminado algo de la memoria de trabajo.

El testigo se envía al nodo raíz, el cual lo procesa. Si es necesario, el nodo crea más testigos y los envía a los nodos conectados a él. Cada uno de estos nodos lo procesa, y si fuera necesario crea nuevas copias del testigo y lo envían a los nodos enlazados. Estos nodos pueden realizar otros test u operaciones. Los testigos continúan su flujo a través de la red, activando distintos nodos para realizar distintas operaciones relacionadas con el reconocimiento.

Unos pocos nodos de la red son los responsables de proporcionar información acerca de la satisfacción parcial o total de los antecedentes de las reglas, y de hacer cambios en el conjunto de reglas satisfechas. Cuando uno de estos se activa al recibir un testigo, añade un elemento o lo elimina del conjunto.

4 El grafo del RETE

Para entender el grafo del RETE es necesario entender tres cosas:

1. Las computaciones que realizan los nodos.
2. Los tipos de testigos que pueden pasarse entre nodos.
3. Como organiza el preprocesador de los patrones los nodos para un conjunto de reglas.

Las reglas más fáciles de procesar son aquellas con sólo un elemento en la parte izquierda. Una de tales reglas podría ser:

```
(p activate-goal
   (goal   ^status  pending)
->
   (mofify  ^status  active))
```

Para procesar esta regla sólo se usaría un tipo de testigo- el testigo alfa. Este testigo contiene dos piezas de información: un puntero al elemento de la memoria de trabajo y una indicación de si el elemento se está añadiendo o eliminando de dicha memoria.

Esta red utiliza tres tipos de nodos: un nodo raíz, nodos de una entrada, y un nodo terminal.

El nodo raíz es esencialmente no operacional; se utiliza principalmente para proporcionar al reconocedor un punto de partida. Un nodo de una entrada comprueba un valor de la memoria de trabajo en un testigo alfa; si el valor es el que está buscando, crea un nodo idéntico y lo pasa a sus sucesores.

Los nodos terminales almacenan información acerca de satisfacciones completas de la parte izquierda de las reglas. La activación de un nodo terminal indica que el elemento o elementos de la memoria de trabajo en el testigo que le acaba de llegar reconocen los patrones de una regla determinada. El nodo terminal añade la regla al conjunto de conflicto.

Los nodos para esta regla se disponen en una secuencia simple: un nodo raíz, un nodo de una entrada para cada test de comparación de un elemento de la memoria de trabajo con un

patrón, y un nodo terminal:

Raíz
↓
Una-Entrada *[^element-type] = goal*
↓
Una-Entrada *[^status]= pending*
↓
Terminal: *(active-goal)*

Para ver como trabaja la red, supongamos primero que se añade el siguiente elemento a la memoria de trabajo:

(goal ^type interact ^status pending)

A causa de esta adición, se creará un testigo y se enviará al nodo raíz. El nodo raíz enviará un testigo idéntico al primer nodo de una entrada. Este nodo comprobará el valor del campo *element-type* y, encontrando que es el valor que estaba esperando -*goal*- creará una copia del testigo y lo enviará al siguiente nodo de una entrada. Este nodo comprobará el valor del campo *status* y, encontrando que es el valor que estaba esperando -*pending*- enviará una copia del testigo al nodo terminal. El nodo terminal añadirá un elemento al conjunto de reglas satisfechas. No envía ningún testigo.

5 Nodos de Dos Entradas

Las reglas seleccionadas por el compilador para más de un patrón no negado requieren un tipo adicional de nodo, denominado de *dos entradas*, y un tipo diferente de testigo denominado *beta*.

Los testigos beta consisten de dos o más punteros a elementos de la memoria de trabajo y una marca que indica si la lista de testigos se va a añadir o a eliminar del estado del sistema. La lista de testigos representa un reconocimiento de los patrones o un conjunto parcial de los patrones de alguna regla.

Los nodos para una regla con más de una condición se organizan como sigue:

1. Un único nodo raíz desde el que crece el grafo.
2. Una secuencia lineal de nodos de una entrada que se construye para cada patrón de la parte izquierda.
3. Un nodo de dos entradas que une dos caminos en uno. Si existen *P* patrones en una regla, se necesitarán un total de *P - 1* nodos de dos entradas.
4. Un nodo terminal para la regla, después de que los caminos se han unido completamente.

Los nodos de dos entradas hacen la mayoría del procesamiento necesario para determinar si se pueden verificar múltiples condiciones simultáneamente. Los nodos de una entrada realizan el procesamiento para determinar si las condiciones pueden satisfacerse independientemente. Los nodos de dos entradas determinan si las condiciones pueden verificarse a la misma vez. La siguiente regla es un ejemplo de como funciona:

```
(p find-colored-blk
   (goal ^type find-colored-blk  ^value <c>)
   (block ^color  <c>)
→
   ...)
```

Los nodos necesarios para esta parte se muestran en la figura 1. Los nodos de dos entradas realizan distintas funciones. Mantienen un registro de todos los testigos que satisfacen la

primera condición independientemente: los dos nodos de una entrada de la parte izquierda encuentran dichos testigos. Mantienen un registro de todos los registros que satisfacen la segunda condición por separado; dichos testigos se encuentran por el nodo de una entrada de la parte derecha. Y comparan los dos conjuntos de testigos para determinar qué parejas de testigos permiten que se satisfagan ambas condiciones simultáneamente.

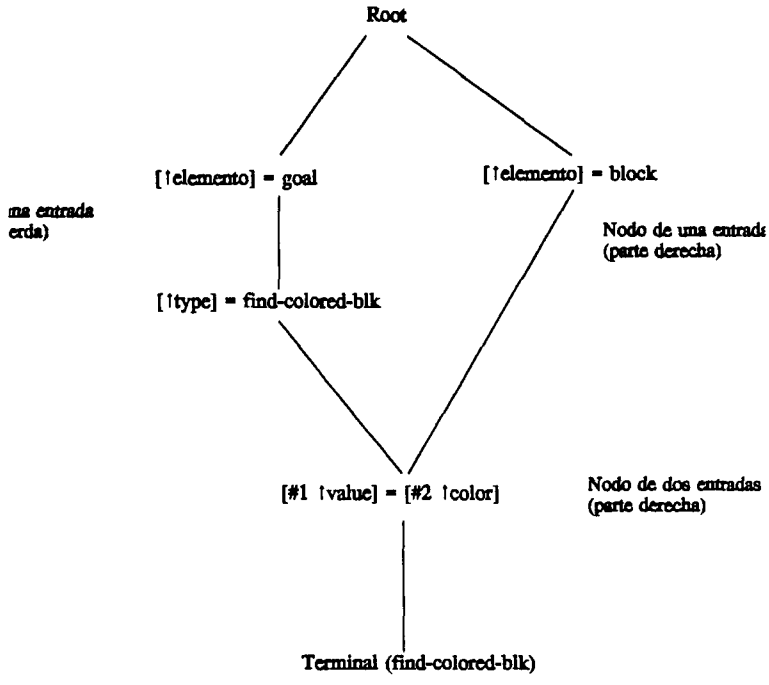


Figura 1

La única restricción es que el valor de la meta debe ser igual al color del bloque. Cuando el nodo de dos entradas encuentra una pareja de testigos que permite que las dos condiciones se satisfagan simultáneamente, construye un nuevo testigo que referencia a ambos tipos de elementos de la memoria de trabajo implicados y envía el nuevo testigo. Estas funciones se realizan incrementalmente: cuando llega un nuevo testigo por algún enlace, el nodo realiza el procesamiento necesario para actualizar la información almacenada y para determinar si se deben enviar nuevos testigos.

6 Eliminando Test Comunes

RETE intenta tener que realizar los mismos test una y otra vez para reglas distintas. En lugar de construir nodos redundantes en el grafo, comparte nodos entre reglas donde sea posible. Para poder utilizar los resultados computados por un nodo en más de un lugar, sólo necesita poner más de un enlace saliendo del nodo. La figura 2 muestra un ejemplo de un grafo con dos producciones.

Los ejemplos anteriores muestran qué pasa cuando se añade un elemento a la memoria de trabajo. También es necesario procesar las eliminaciones de los elementos. El procesamiento de las eliminaciones es bastante similar al de las adiciones. Los nodos raíz y de una entrada no hacen nada diferente. El nodo terminal utiliza la marca en el testigo que recibe para determinar si añadir o eliminar información del conjunto de reglas satisfechas. Los nodos de dos entradas realizan dos cosas con la marca del testigo: determinan si añadir o eliminar información del estado almacenado y qué marca dar al testigo que crean. Por lo demás, el procesamiento que realizan es idéntico al realizado cuando se añade un elemento a la memoria de trabajo.

Ciertamente, se podrían definir más tipos de nodos a parte de los descritos, aunque sólo se necesitan unos pocos más para completar un conjunto bastante útil. De hecho, sólo son necesarios dos nodos adicionales para completar el lenguaje OPS5.

Primero, un nodo *"not"*, muy similar al nodo de dos entradas, para los patrones negados. Segundo, una variante del nodo de una entrada para determinadas construcciones especiales del OPS5. Este segundo tipo de nodo se utiliza para procesar patrones que contienen dos o más tipos de ocurrencias de la misma variable; el tipo básico de nodo de una entrada comprueba características constantes de elementos de la memoria de trabajo.

Producciones:

```
(P Plus0x
    (Goal !type simplify !object <N>)
    (Expresión !name <N> !arg1 0 !op + !arg2 <X>)
--> ...)

(P Time0x
    (Goal !type simplify !object <N>)
    (Expresión !name <N> !arg1 0 !op * !arg2 <X>)
--> ...)
```

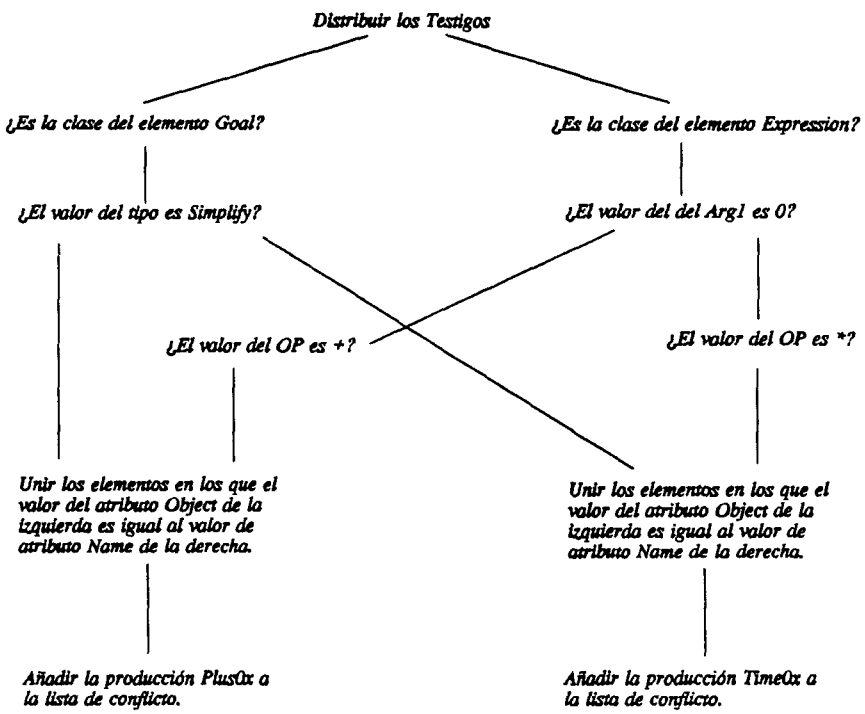


Figura 2

Bibliografía

Forgy, C. RETE: a fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligenc, Vol. 27, 1985 pp 219-227

Forgy, C.- Shepard, S. J. RETE: a fast match algorithm. AI Expert, january 1987, pp 34-40

Tuttle, S. Historical rete networks for debugging rule-based systems. Proc. of the 1991 IEEE. Int. conf. on tools for AI. San José CA.

CAPÍTULO IV

Sistemas de Mantenimiento del Razonamiento

Índice

1 Introducción 1

2 Mantenimiento del Razonamiento 3
- 2.1 Incrementalidad 4
- 2.2 Selectividad 4
- 2.3 Gestión de la Consistencia 4
- 2.4 No-monotonía 5

3 Satisfacción de Restricciones 7
- 3.1 Problemas de Satisfacción de Restricciones 7
- 3.2 Algoritmos de Satisfacción de Restricciones 8
- 3.3 Procesamiento de Redes de Restricciones 9
- 3.4 Backmarking 11
- 3.5 Forward Checking 12

Bibliografía 15

1 Introducción

Sería fácil encontrar una aplicación de sistema experto que se caracterizara por tener que hacer suposiciones en un determinado momento. Ello puede ser debido a que o el sistema debe responder en tiempo real, y por lo tanto no tiene tiempo de buscar el dato necesario; o a que simplemente no se dispone de dicho dato cuando se necesita. Por lo tanto el sistema debe tomar decisiones (ej: construir un plan) en base a *información incompleta*. Podría utilizar *razonamiento por defecto*, es decir, suponer aquello que se necesario mientras no tenga alguna evidencia de que no sea cierto. Cuando en respuesta a una situación dinámica deba cambiar una hipótesis anteriormente establecida, tendrá que rehacer todo el proceso de razonamiento. Pero, evidentemente, esto requiere de una gran cantidad de procesamiento redundante, ya que muchas de las conclusiones intermedias anteriores siguen siendo válidas. Esto es especialmente grave en sistemas que deben proporcionar una respuesta en tiempo real.

Un Sistema de Mantenimiento del Razonamiento (SMR) proporciona la posibilidad de aumentar la velocidad de la búsqueda, de registrar los pasos de inferencia de forma que no tengan que repetirse, y de gestionar la actualización de la representación de una situación (real o hipotética). Esto se consigue manteniendo un registro de las dependencias lógicas entre las diferentes partes de una representación.

Registrar las dependencias implica notar que ciertas acciones o conclusiones *dependen* de otras acciones y conclusiones. Si, por ejemplo, estamos generando un plan compuesto de varios pasos, podríamos tener que

El paso P2 depende de que el paso P1 se haya completado

o que

El paso P4 depende de que los hechos H1,..,.Hn sean verdaderos

Adicionalmente, podemos darnos cuenta de que ciertas acciones y conclusiones dependen de que no se verifique otras.

El paso P1 depende de que no se realicen los pasos P1...Pn

2 Mantenimiento del Razonamiento

Un SMR mantiene una estructura de *dependencias de datos*, donde se mantiene un registro de cómo se ha alcanzado una conclusión. Si se demuestra que una fórmula es una consecuencia lógica de un conjunto de otras, entonces se puede informar al SMR de este hecho, el cual será almacenado como una *justificación*. También se informará al SMR cuando se detecte un conjunto de fórmulas inconsistentes, lo cual será registrado como *nobueno (nogood)*.

Como ejemplo consideremos un teoría consistente sólo de la sentencia P <- Q. Si añadimos Q, podremos concluir P. Por lo tanto, la conclusión P *depende* de la sentencia P <- Q. Si posteriormente descubrimos que Q deje de ser válido, tendremos que abandonar la certeza sobre P.

Disponer de un conjunto de registro de dependencias facilita el proceso de búsqueda (de la solución). Suponga que la tarea del sistema de resolución de problemas consiste en encontrar un conjunto de valores que verifiquen un conjunto determinado de características. Esta tarea puede considerarse como el problema de encontrar algún conjunto consistente de sentencias Δ tales que $T + \Delta \vdash G$, donde G describe el conjunto de restricciones y T el conjunto de reglas para el domino en cuestión. El sistema de resolución de problemas debe buscar eficientemente un subconjunto de bien-definido del espacio de posibles Δ. La información almacenada en un SMR facilita dicha búsqueda eficiente de dos maneras; ofreciendo *incrementalidad* y *selectividad*.

La otra función principal de un SMR tiene que ver con el registro de los *fallos* que encuentra el sistema de resolución de problemas. Un fallo durante este proceso puede indicar o una inconsistencia (ej. P y $\neg$ P) o un conflicto (ej. intentar asignar un valor a una variable que ya tiene uno). Un SMR puede ayudar al sistema de resolución de problemas ofreciendo una *gestión de la consistencia (o resolución de conflictos)* y la habilidad de razonar no-monótonamente.

2.1 Incrementalidad

Almacenar las justificaciones permite al sistema de resolución de problemas tener un acceso inmediato a las consecuencias lógicas que ya conoce, evitando así una gran cantidad de procesamiento redundante. Suponga que el sistema de resolución de problemas examina algún Δ donde $P \subset \Delta$, y determina que $T + P \vdash Q$ y por lo tanto que $T + \Delta \vdash Q$. Registrar $T + P \vdash Q$ como justificación posibilita el determinar directamente que $T + \Delta' \vdash Q$ a la hora de examinar Δ', donde $P \subset \Delta'$. De este forma, el movimiento en el espacio de posibles Δ es *incremental* en el sentido de que cualquier trabajo hecho al examinar un Δ se aprovechará al examinar otros Δ.

2.2 Selectividad

La gestión de la búsqueda puede efectuarse eficientemente registrando conjuntos inconsistentes de sentencias como *nogoods*. El conocimiento de que un determinado conjunto de sentencias es inconsistente puede utilizarse para prevenir la exploración de cualquier Δ que incluya dichas sentencias. Si el sistema examina algún Δ donde $P \subset \Delta$, y determina que $T + P \vdash \neg Q$, y por lo tanto que $T + \Delta \vdash \neg Q$. Si $P \wedge Q$ es un nogood (P es inconsistente con Q), el sistema de resolución no necesita considerar cualquier Δ' tal que $P \subset \Delta'$. Así, la exploración del espacio de Δ posibles es *selectivo*, en el sentido de que el sistema no examinará un Δ nuevo que haya demostrado no aportar nada a la solución.

2.3 Gestión de la Consistencia

Un Sistema de Mantenimiento del Razonamiento que almacena los nogoods permite que el sistema razone con inconsistencias. Esto es importante, ya que según las reglas de la lógica clásica, una teoría inconsistente posee una deducción inútil porque cualquier cosa puede deducirse de una teoría inconsistente. Supongamos que encontramos un conjunto de sentencias Δ tal que $T + \Delta$ es inconsistente. Necesitamos de algún medio de gestionar la inconsistencia, ya sea mediante la recuperación de la consistencia, o proporcionando algún mecanismo que asegure que el razonamiento se realiza sobre una base de datos consistente. Podríamos localizar un elemento P de Δ, tal que la eliminación de P dando Δ' restaura la consistencia.

Registrar las dependencias de los datos proporciona una forma de aislar los datos que contribuyen a la inconsistencia, a la vez que proporciona la información necesaria para recuperarse de un estado inconsistente.

2.4 No-monotonía

Los sistemas de Mantenimiento del Razonamiento proporcionan la posibilidad de razonar no monótonamente, permitiendo que el sistema modifique su conjunto de creencias a la recepción de información nueva, y posiblemente contradictoria. Necesitamos la habilidad de eliminar parte de nuestra base de datos, y encontrar como se ven afectadas nuestras creencias por dicha eliminación. Si, por ejemplo, sabemos que $T + P \vdash Q$, pero descubrimos que P ha cambiado en algún aspecto, necesitamos ser capaces de determinar como afecto esto a nuestras creencias en Q.

3 Satisfacción de Restricciones

Se han desarrollado una gran cantidad de trabajos sobre determinado tipo de problemas de satisfacción de restricciones, los cuales tienen numerosas aplicaciones, entre las que se encuentran el coloreado de regiones, problemas de n-queens, y criptoaritmética, etc... A continuación se describirán algunos algoritmos desarrollados para resolver estos problemas.

3.1 Problemas de Satisfacción de Restricciones

Un Problema de Satisfacción de Restricciones (PSR) se define como sigue. Una asignación (binding) para una variable v se escribe t/v. Una *sustitución* para un conjunto de variables $\{v_1, \ldots, v_n\}$ es un conjunto de n asignaciones, una para cada v_i. Una sustitución $\{t_1/v_1 \ldots. t_n/v_n\}$ está incluida en una sustitución $\{s_1/w_1 \ldots. s_n/w_n\}$ si y sólo si $t_i = s_j$ para todo $v_i = w_j$. Una restricción sobre un conjunto de variables $\{v_1, \ldots, v_n\}$ es un conjunto de sustituciones,

$$\{\{t_{1,1}/v_1 \ldots. t_{1,n}/v_n\} \ldots.\{t_{m,1}/v_1 \ldots. t_{m,n}/v_n\}\}$$

Tal restricción se podría escribir:

$$C(v_1, \ldots, v_n) = \{<t_{1,1}/v_1 \ldots. t_{1,n}/v_n> \ldots.<t_{m,1}/v_1 \ldots. t_{m,n}/v_n>\}$$

Una sustitución B para un conjunto J reúne una restricción C en J si y sólo si $B \in C$. De otra forma violaría dicha restricción. Un dominio para una variable v es un conjunto finito de etiquetas S, y esto podría escribirse $D(v) = S$. Un problema de satisfacción de restricciones sobre un conjunto $J = \{X_1 \ldots. X_n\}$ incluye un dominio para cada X_i y un conjunto P de restricciones en subconjuntos de J. Una sustitución $B = \{t_1/X_1 \ldots. t_n/X_n\}$ es una solución a tal problema si y sólo si cada t_i está en el dominio de X_i y B verifica todas las restricciones en P. Un ejemplo podría ser el siguiente:

$D(X_1) = D(X_2) = \{a, b, c\}$
$D(X_3) = D(X_4) = D(X_5) = \{a, b\}$
$C(X_1, X_3) = \{<b,a>, <c,a>, <c,b>\}$
$C(X_2, X_3) = \{<b,a>, <c,a>, <c,b>\}$
$C(X_3, X_4) = \{<a,b>\}$
$C(X_3, X_5) = \{<a,b>\}$
$C(X_4, X_5) = \{<a,b>\}$

Las sustituciones se escriben normalmente como tuplas cuando está claro el conjunto de variables implicadas y como dichas variables van ordenadas. Una tupla que incluya una o más variables sin asignar es una *tupla parcial.* Las variables sin asignar se especifican mediante un guión (-). Ejemplo: <a, c, -, d, ->.

Dado un PSR sobre un conjunto J y una tupla parcial T, podría dividirse J en el conjunto de variables asignadas en T, denominadas el conjunto pasado P, y el conjunto de variables sin asignar en T, llamado el conjunto futuro F. Cualquier tupla T' cuyo conjunto futuro sea F - {v} y cuyo conjunto pasado sea P ∪ {v} para algún v ∈ F, tal que las asignaciones para P en T' son las mismas que en T, es una *extensión* de T.

3.2 Algoritmos de Satisfacción de Restricciones

El algoritmo más evidente y más ineficiente consiste en generar una a una todas las combinaciones de asignaciones posibles y comprobar si alguna de ellas es una solución. Este algoritmo es el *generar y testar.* Dado un PSR sobre un conjunto S, suponga que se especifica una ordenación para los miembros de S y que dicha ordenación es impuesta sobre los dominios de cada miembro de S. Entonces es posible generar cada asignación posible en S y comprobada con las restricciones especificadas. Con este método, todas las variables reciben su asignación antes de que se compruebe cualquier restricción.

El árbol de búsqueda explorado por generar y testar puede ser mucho más grande que el explorado por la vuelta atrás cronológica. En lugar de generar una asignación para cada variable en S antes de comprobar las restricciones, la vuelta atrás cronológica comprueba las restricciones tan pronto como se asignen las variables que intervienen en la restricción. Dado un PSR con un conjunto de restricciones P, el árbol de búsqueda tiene la siguiente forma: cada nodo T tiene un conjunto de hijos que se corresponden con el conjunto de tuplas parciales T' que son extensiones de T y que verifican las restricciones P. Sin embargo, este método puede ser ineficiente. La siguiente lista se corresponde con las tuplas parciales que violan alguna de las restricciones del ejemplo anterior.

1. <a, a, a, -, - >
2. <a, a, b, -, - >
3. <a, b, a, -, - >
4. <a, b, b, -, - >
5. <a, c, a, -, - >
6. <a, c, b, -, - >
7. <b, a, a, -, - >
8. <b, a, b, -, - >
9. <b, b, a, a, - >
10. <b, b, a, b, a >
11. <b, b, a, b, b >
12. <b, b, b, -, - >
13. <b, c, a, a, - >
14. <b, c, a, b, a >
15. <b, c, a, b, b >
16. <b, c, b, -, - >
17. <c, a, a, -, - >
18. <c, a, b, -, - >
19. <c, b, a, a, - >
20. <c, b, a, b, a >
21. <c, b, a, b, b >
22. <c, b, b, -, - >
23. <c, c, a, a, - >
24. <c, c, a, b, a >
25. <c, c, a, b, b >
26. <c, c, b, a, - >
27. <c, c, b, b, - >

Este ejemplo típico ilustra la principales fuentes de ineficiencia de la vuelta atrás cronológica. En ocasiones se descubren las mismas incompatibilidades varias veces; la incompatibilidad entre X_1 = a y X_3 = a es descubierta primero con la tupla 1 y otra vez (redundantemente) con las tuplas 3 y 5. El fallo de las tuplas 1 y 2 elimina la posibilidad de que X_1 = a porque no existe un valor de X_3 que sea compatible con él, y aún así este valor se vuelve a intentar otra vez en las tuplas 3, 4, 5, y 6.

Con la tupla 7, se revela la incompatibilidad entre X_2 = a y X_3 = a y la misma otra vez en la tupla 17. Hacer notar que esta incompatibilidad no se descubre con la tupla 1, aunque X_2 = a y X_3 = a porque la tupla no verifica las restricciones sobre X_1 y X_3 que se comprueban primero. La tupla 1 no facilita la posibilidad de descubrir ambas incompatibilidades al mismo tiempo. Este ejemplo está lejos de ser patológico. Es muy fácil construir casos donde la redundancia crezca exponencialmente con el número de variables en el PSR. A causa de esto, el comportamiento ineficiente de la vuelta atrás cronológica se denomina a veces *basura* (trashing). Las siguientes técnicas intentan evitar este tipo de comportamiento.

3.3 Procesamiento de Redes de Restricciones

La idea del preprocesamiento en la satisfacción de restricciones es la de conseguir una reducción en el espacio de tuplas que debe explorarse, con un coste de refinamiento del problema en una fase inicial. Consideremos un PSR sobre un conjunto $I = \{X_1 \ldots X_n\}$. Para cada variable X_i en I existe un nodo en la red etiquetado con el conjunto $D(X_i)$. Cada restricción unaria $C(X_i)$ se representa con un bucle sobre el nodo X_i etiquetado con el conjunto $C(X_i)$. Cada restricción binaria $C(X_i,X_j)$ se representa con un arco entre los nodos

X_i y X_j etiquetado con el conjunto $C(X_i,X_j)$. Un ejemplo se muestra en la figura siguiente:

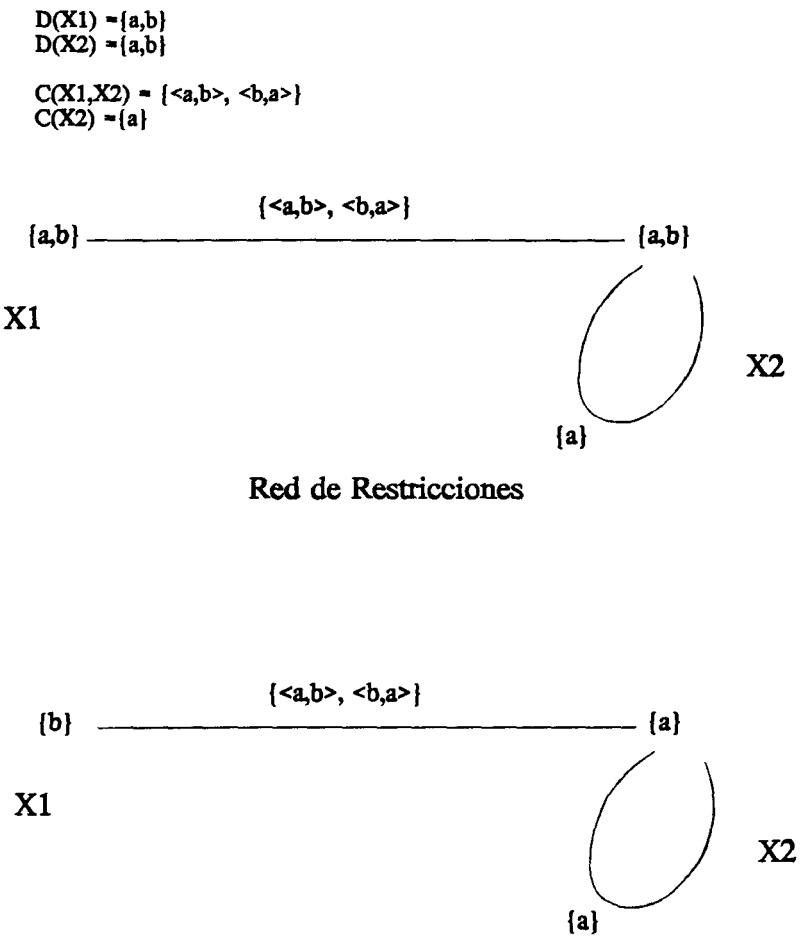


Figura 1

Es posible identificar tres tipos de inconsistencias dentro de la red: nodo, arco, y camino. Los algoritmos actuales pueden eliminar tales inconsistencias, dejando una forma refinada del problema, la cual, cuando se aplica la vuelta atrás cronológica no provocará el comportamiento penoso comentado anteriormente.

La inconsistencia de nodo surge si y solo si existe un bucle en algún nodo X_i que posea algún valor en $D(X_i)$ que no esté presente en $C(X_i)$.

La inconsistencias de arcos surgen cuando existe un arco entre alguna pareja de nodos X_i y X_j etiquetado con el conjunto $C(X_i,X_j)$ de forma que existe algún valor x de $D(X_i)$ para el que no existe el valor correspondiente y en $D(X_j)$ de forma que $<x,y> \in C(X_i,X_j)$. La figura anterior muestra una red original y otra con las inconsistencias de nodo y arco eliminadas.

La inconsistencias de caminos se producen cuando existe algún valor x de $D(X_i)$ y un valor y en $D(X_j)$ de forma que $<x,y> \in C(X_i,X_j)$, para el que existe un camino desde X_i hasta X_j con más de un arco y sin una asignación de valores a las variables a lo largo del camino que incluya las asignaciones X_i = x y $X_j = y$ y que satisfaga todas las restricciones a lo largo del camino.

3.4 Backmarking

El Backmarking es una extensión del chronological backtraking que consigue mejorar su eficiencia mediante dos arrays extras de información. Para cada variable X_i y etiqueta t, el valor de Mark[i,t] es el menor j tal que el (entonces) valor actual de X_j era incompatible con la asignación t/X_i la última vez que dicha asignación se intentó. Ahora, podría ser que el valor de alguna otra variable X_k donde $k \leq Mark[i,t]$, haya cambiado desde que se intentó por última vez t/X_i, en cuyo caso el valor de Mark[i,t] estará desfasado. Así, para cada variable X_i el valor de *Low[i]* es el menor j tal que el valor de X_j ha cambiado desde que la columna X_i del vector Mark fue actualizado por última vez.

La búsqueda se realiza como para la vuelta atrás cronológica. Pero cada vez que se intenta una asignación t/X_i, pueden surgir uno de los tres escenarios que se muestran en la figura 2. Si Low[i] > Mark[i,t], entonces no es necesario intentar t/X_i porque la asignación que lo hacía incompatible no ha cambiado. Si Low[i] ≤ Mark[i,t], entonces es necesario comprobar t/X_i contra el resto de asignaciones actuales. Notar que no es necesario comprobar las asignaciones de cualquier variable anterior a Low[i]. Esto es porque ninguna variable anterior a Low[i] ha cambiado su valor desde que se intentó t/X_i por última vez, de forma que todos esas comprobaciones tendrían éxito.

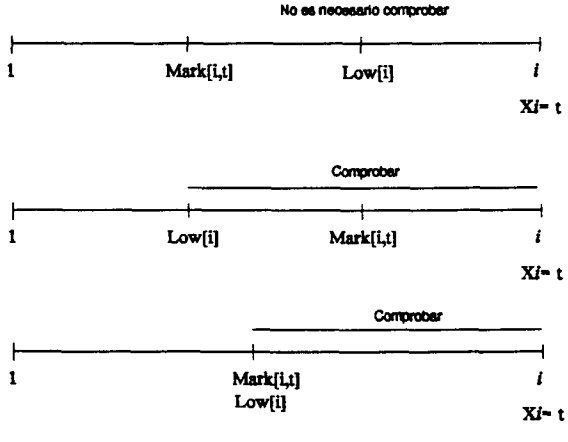


Figura 2

Claramente el backmarking ofrece un ahorro sobre la vuelta atrás cronológica con un coste mínimo de almacenamiento (los dos vectores). Pero el algoritmo está diseñado para problemas con restricciones binarias que deben comprobarse una cada vez, con lo que no sería utilizable cuando la aplicación requiera de restricciones n-arias que deban comprobarse simultáneamente. El algoritmo para encontrar todas las soluciones es el siguiente.

```
PROCEDIMIENTO Backmark (i,Mark, Low)

  PARA cada t ∈ D(Xi)
    SI Mark[i,t] ≥ Low[i]
    ENTONCES
      Asignar t a Xi
      fallo = falso
      j = Low[i]
      MIENTRAS j<i y no fallo
        fallo = t/Xi es incompatible con el valor actual de Xj
        SI no fallo
        ENTONCES  j = j+1
        FIN SI
      FIN MIENTRAS
      Mark[i,t] = j
      SI no fallo
      ENTONCES
        SI i<n
        ENTONCES Backmark(i+1,Mark,low)
        SINO Solución Encontrada
        FIN SI
      FIN SI
      desasignar Xi
    FIN SI
    Low[i] = i - 1
    PARA cada j desde i+1 hasta n
      Low[j] = el mínimo de Low[j] y i-i
    FIN PARA
  FIN PARA
```

3.5 Forward Checking

La base de este método consiste en comprobar la compatibilidad de un conjunto de asignaciones *antes* de que aparezca en una tupla.

Se mantiene una tabla de marcas, uno para cada valor de cada dominio, que indica si dicho

valor está disponible o eliminado para dicho dominio. Cada nodo en el árbol de búsqueda *T* tiene un conjunto de hijos que se corresponden con el conjunto de todas las tuplas parciales que son extensiones de T y cuya extensión *t/v* es tal que *t* no se ha eliminado del dominio de *v*. Cada hijo hereda una tabla en la que cada valor disponible para cualquier dominio de las posibles extensiones es compatible con cada una de las asignaciones hechas hasta la fecha.

La búsqueda es como sigue. En un determinado nodo en el árbol de búsqueda en el nivel *n*, se asigna a X_n el primer valor disponible de su dominio. Se comprueba que este valor es compatible con cada uno de los valores de cada variable del conjunto futuro, y cualquier valor incompatible se elimina. Así se garantiza que cada valor seleccionado sea siempre compatible con el valor asignado a cada variable de la tupla parcial actual. Si se eliminan todos los posibles valores para una determinada variable, entonces la asignación t/X_n no puede contribuir a la solución, y por lo tanto no se incluirá en la tupla actual de trabajo para proseguir la búsqueda con ella. Esto se conoce con el nombre de *row wipe-out*. En caso contrario, la búsqueda continúa (al nivel n+1 del árbol de búsqueda) con la asignación t / X_n. Cuando esto se haya completado, la búsqueda se retoma (al nivel n del árbol), después de la desasignación de X_n y de la restauración de la tabla al estado que tenía antes de dicha asignación, continuando con el siguiente valor disponible del dominio de X_n. El algoritmo que encuentra todas las soluciones se muestra a continuación:

```
PROCEDIMIENTO buscar (n)
  SI todas las variables tienen valor
  ENTONCES solución encontrada
  SINO
    PARA cada t en el dominio de Xn
      Asignar t a Xn
      CheckForward(n, RowWipeOut, acutalizaciones)
      SI not RowWipeOut
      ENTONCES
        buscar (n+1)
        deshacerActualizaciones (actualizaciones)
      FIN SI
    FIN PARA
  FIN SI


PROCEDIMIENTO CheckForward (n,RowWipeOut,actualizaciones)

  actualizaciones = {}
  RowWipeOut = false
  PARA  cada i > n
    PARA  cada t disponible en el dominio de Xn
      Asignar t a Xi
      Comprobar restricciones
      SI  violación
      ENTONCES
```

```
            actualizaciones = actualizaciones U {t / Xi}
            Hacer t eliminado para Xi
          FIN SI
          Desasignar Xi
        FIN PARA
        SI todo t del dominio de Xi está eliminado
        ENTONCES RowWipeOut = true
        FIN SI
      FIN PARA


PROCEDIMIENTO  deshacerActualizaciones (actualizaciones)

    PARA cada t/v en actualizaciones
      Hacer t disponible para v
    FIN PARA
```

Tanto datos experimentales como estadísticos aportados por varios investigadores sugieren que el forward checking es superior que la vuelta atrás cronológica y que el backmarking. Sin embargo otros autores consideran que cuando sólo se necesite una solución mucho del trabajo necesario para calcularla es redundante. Al mismo tiempo, si las soluciones están concentradas en una zona del espacio de búsqueda, las comprobaciones redundantes tendrán éxito y por lo tanto no eliminarán ramas del árbol. Para cierto tipo de aplicaciones, el rendimiento de forward checking puede ser peor que el de la vuelta atrás cronológica.

Bibliografía

Shanahan, Murray. Search, Inference amd Dependencies in Artificial Intelligence. ed. Ellis Horwood 1989