Combinación de técnicas para solución de problemas

Jesús González Boticario

Departamento de Inteligencia Artificial
Facultad de Ciencias
Universidad Nacional de Educación a Distancia
jgb@dia.uned.es

Resumen

Muchas veces, en Inteligencia Artificial (IA) se utilizan métodos aislados de inferencia y/o representación para abordar la solución de casos prácticos. No obstante, para solucionar problemas de cierta entidad en dominios concretos es necesario combinar distintos mecanismos (p.ej., búsqueda, marcos, lógica y aprenedizaje). Con este fin, se muestra el uso de una versión reducida de la arquitectura THEO [7], empleada en la construcción de sistemas aprendices reales [3]. Este modelo, directamente implementado en Lisp, puede servir para integrar los contenidos de asignaturas de programación, de aprendizaje y de fundamentos en IA.

1 Introducción

Muchas veces, en Inteligencia Artificial (IA) se utilizan métodos aislados de inferencia y/o representación para abordar la solución de casos prácticos. No obstante, para solucionar problemas de cierta entidad en dominios concretos es necesario combinar distintos mecanismos (p.ej., búsqueda, marcos, lógica y aprenedizaje). Por tanto, sería deseable disponer de un esquema de representación que permita conjugar de una forma estándar y homogénea las diferentes técnicas. Las llamadas *arquitecturas integradas*[1] proporcionan un entorno genérico de programación que cumple tres características básicas: *representación uniforme de todos sus elementos, disponibilidad de diferentes métodos de inferencia* que pueden ser combinados y *organización eficiente del conocimiento adquirido.*

En este trabajo se muestra el uso de una versión reducida de la arquitectura THEO [7], empleada en la construcción de sistemas aprendices reales [3]. En concreto, se analizará su aplicación en un problema que, además de requerir una formalización de una cantidad considerable de conocimiento del dominio, tiene por objeto la realización de una tarea cuya solución sólo se conoce parcialmente de antemano. Esto permite introducir de una forma natural el aprovechamiento de los métodos de aprendizaje de THEO para mejorar la solución inicialmente obtenida. El modelo propuesto está directamente implementado en Lisp y puede servir para integrar los contenidos de asignaturas de programación, de aprendizaje y de fundamentos en IA (en este último caso ya se ha ilustrado su uso [4]). De esta forma, se puede comprobar cómo se definen las entidades del dominio (aquí representadas mediante *marcos*), cómo se combinan los distintos mecanismos de inferencia existentes (incluyendo el aprendizaje) y cómo se puede ampliar el propio sistema mediante la especificación de nuevos procedimientos.

Una vez finalizada la introducción de los motivos principales que fundamentan este trabajo, se comenta el funcionamiento de la arquitectura propuesta, tanto desde el punto de vista estructural como operativo. A continuación, en la parte de aplicación se utiliza dicho esquema para resolver un problema dado; primero, aplicando algunos métodos de inferencia predefinidos, y segundo, realizando un planteamiento alternativo acorde con los métodos de aprendizaje seleccionados. Finalmente, se destacan algunas cuestiones de interés relacionadas.

[1] El término *arquitectura* proviene de la analogía existente entre este enfoque y las arquitecturas "hardware", donde hay ciertos componentes fijos de los cuales se parte y otros ajustables en el diseño.

2 Funcionamiento de una arquitectura

El esquema de representación y los métodos de inferencia que aquí se comentan son una versión reducida y personalizada de los originalmente disponibles en THEO [7] y en la aplicación del Calendario [3] (existe una introducción de los mismos en el último capítulo de un libro docente [4]).

Antes de nada conviene precisar que cualquier *arquitectura* no es un programa concebido para solucionar directamente un determinado tipo de problemas, sino más bien una serie de estructuras de datos y de proceso que proporcionan un marco general sobre el cual pueden desarrollarse sistemas concretos más facilmente y con una cierta estandarización.

THEO es un sistema implementado en Lisp cuya principal característica es utilizar una representación uniforme basada en marcos (en contraposición a otras arquitecturas como SOAR [9] y PRODIGY [2], que utilizan espacios de problemas). Tanto los hechos o *creencias*, como los problemas y los métodos para solucionarlos, se representan a través de los valores asignados a las características de los marcos (a las que también denominaremos *slots*). El mecanismo de mantenimiento de verdad (*truth maintenance system*), es decir, de garantizar la coherencia entre todos los hechos existentes en cada momento, está basado en la continua creación de explicaciones para todas las creencias inferidas. Esto es, cada vez que se infiere el valor de un *slot*, se anota en un *subslot* todos aquellos *slots* de los cuales depende el valor obtenido. Esto permite utilizar dicha explicación como base de un aprendizaje deductivo basado en la explicación (tal y como se describe en otra parte [1]). Otros métodos de aprendizaje disponible son: el mero almacenamiento de la información obtenida (llamado *caching*, o *recogelotodo*); la creación de *macrométodos* (al igual que en SOAR, es un resumen que explica lo ocurrido); la obtención de reglas aprendidas estilo prolog, generadas a partir de árboles de decisión (véase el apartado 3.2) y la aplicación de un modelo básico de aprendizaje conexionista (el clásico Backpropagation [10]).

Por otro lado, otra de las características más notables del sistema es su capacidad de acceso y modificación de la mayoría de las estructuras que componen su propia estructura. De hecho, tanto los métodos propios como los construidos siguen el mismo esquema de representación que se detalla a continuación. Esta generalidad le ha permitido ser lo suficientemente flexible como para resolver tareas tan diversas como el control de un robot [6] o la gestión personalizada de una agenda [5].

2.1 Estructura

Las características estructurales de un sistema determinan sus características funcionales, aunque el paso de unas a otras esté lejos de ser obvio. Se introduce en primer lugar la descripción de las estructuras simbólicas básicas que permiten representar el conocimiento en el sistema.

La terna que refleja el mecanismo de representación básico del sistema es <marco slot valor>(p.ej., `(reunion-10-13-1994-1-24-19 asistentes (acebron bermejo))`). Partiendo de este esquema se pueden distinguir los siguientes principios:

1. Cada hecho o creencia se representa mediante la terna (<entidad><slot><valor>) (p.ej., `(dia-semana valores-posibles (lunes martes miercoles jueves viernes sabado domingo))`).

2. Un problema se describe con la pareja (<entidad><slot>), siendo la solución del problema el valor inferido (p.ej., la solución del problema `(bermejo puesto)` es `estudiante`).

3. Una clase de problemas se representa por (?x <slot>) o simplemente <slot>; y señala la relación <slot>restringida al subdominio especificado por el tipo de entidad especificado en ?x. Si sólo se indica el nombre de la relación, no se impone ningún tipo de restricción y se considera como problemas relacionados todas aquellas direcciones cuyo último elemento sea <slot>(p.ej., `(?persona reuniones)` está asociado con todas las personas que han asistido a alguna reunión).

4. Los elementos descritos en los puntos 2 y 3 pueden considerarse a la

vez <entidad>y ser sustituidos convenientemente en cualquiera de las expresiones anteriores (p.ej., `((bermejo puesto) reuniones)` señala a todos los estudiantes que han asistido a alguna reunión).

Considerando este esquema en la figura 1 se muestra la estructura general de un marco en el sistema. En este caso se ilustra la posibilidad de incluir la definición de un *slot slot-1* dentro de la definición del marco que lo contiene o de hacerlo aparte mediante el uso del *subslot* predefinido *dominio*. Esto, además de "descongestionar" el código, permite que un *subslot* sea aplicado a diversas entidades (las indicadas en *dominio*).

```
(nombre-marco "sin-valor"
  (generalizaciones
    (marco-1 marco-2... marco-n)
  (especializaciones
    (marco-1 marco-2... marco-n)
  (slot-1 "sin-valor"
      (slot-11 "sin-valor")
      (slot-12 "sin-valor"
        (slot-121 "sin-valor")
        (slot-122 "sin-valor")
            ... ... ...)))
(slot-1 "sin-valor"
    (dominio (nombre-marco))
  (.....))
```

Figura 1: Marco genérico

2.2 Inferencia

Dado que cualquier problema consiste en solicitar el valor de un *slot*, la esencia de la operatividad del sistema depende de los métodos de inferencia que se hayan definido. La asignación de un determinado método (o una serie de ellos) a un *slot* de una entidad sigue las mismas convenciones del epígrafe anterior. En concreto, se utiliza el subslot `metodos`, tal y como se muestra en la figura 2.

Algunos de los métodos disponibles son:

1. `herencia`: Consiste en inferir los valores de un *slot* a través de la relación jerárquica definida mediante los *slots* `generalizaciones` y `especializaciones`.

2. `perdida-contexto`: Devuelve el valor de "la cola" de la dirección en la que se halla inmerso el *slot*. Por ejemplo, la pérdida de contexto de `(persona puesto metodos)` es el valor contenido en la dirección `(herederos metodos)`.

```
(puesto-asistente "sin-valor"
  (generalizaciones (sol-slot))
  (dominio solicitudes)
  (numero-valores 1)
  (valores-posibles (secretaria profesor investigador
                     alumno ...)
  (metodos (prolog))
  (clausulas-prolog "sin-valor" (
    ((puesto-asistente ?s ?categoria):-
      (sol-asistentes ?r ?a)
      (puesto ?a ?categoria))))))
```

Figura 2: Asignación expresa de un método

3. `valor-por-defecto`: Accede al valor del *subslot* `valor-por-defecto` del slot dado.

4. `lisp`: en algunas ocasiones conviene aplicar una función para calcular el valor de un slot. De esta forma, las acciones realizadas por la función permanecen "ocultas" en su código. Así, se logra una mayor eficiencia, pero se impide el acceso explícito a dichas acciones (ver siguiente punto).

5. `prolog`: determina el valor de un campo aplicando un conjunto de reglas lógicas asignadas al *slot* hasta que alguna infiera el valor requerido. En la figura 3 se contrapone la definición explícita e implícita del *slot* `grado-ocupacion`. Las reglas prolog se ejecutan una tras otra, siguiendo el orden en que están escritas, hasta que se consiga inferir el valor deseado. Por tanto, la disyunción `OR` en el antecedente de la regla, $A \vee B \rightarrow C$, se traduce en la especificación de las reglas $A \rightarrow C$ y $B \rightarrow C$.

Cualquier método lo que hace es apuntar a un conjunto de direcciones donde se espera encontrar el valor buscado. Esto se ve claramente en las reglas `prolog`, cuyos antecedentes señalan de forma explícita algunas de dichas direcciones. En realidad, todo el funcionamiento del sistema consiste en ir recorriendo las diferentes estructuras (*marcos* formados por listas de listas) siguiendo lo indicado por los métodos asignados. Cuando se llama a la función básica del sistema `(dame-valor <slot><entidad>)`, bien directamente o a través de algún interfaz (como ocurre en el siguiente epígrafe), en el caso de

```
(persona *sin-valor*
  (...
  (grado-ocupacion-explicito *sin-valor*
    (metodos (prolog))
    (clausulas-prolog (
          ((grado-ocupacion ?persona ?valor) :-
            (horas-trabajadas ?persona ?horas)
            (eval (>= ?horas 0))
            (eval (<= ?horas 6))
            (eval 'bajo ?valor))
          ((grado-ocupacion ?persona ?valor) :-
            (horas-trabajadas ?persona ?horas)
            (eval (>?horas 6))
            (eval (<= ?horas 8))
            (eval 'medio ?valor))
          ((grado-ocupacion ?persona ?valor) :-
            (horas-trabajadas ?persona ?horas)
            (eval (> ?horas 8))
            (eval (<= ?horas 15))
            (eval 'alto ?valor))))))
  (grado-ocupacion-implicito *sin-valor*
          (metodos (lisp))
          (lisp grado-ocupacion))) ...)
```

Figura 3: Cálculo explícito e implíto de un valor

que no haya ningún valor asignado se pregunta por la dirección (<slot><entidad>dar). Debido a que dar es otra entidad del sistema con sus propios métodos de inferencia, y gracias a la aplicación primero del comentado perdida-contexto (para así poder preguntar directamente por (dar)), se garantiza un proceso homogéneo de inferencia para todos aquellos *slots* que no tengan un valor ya asignado. La incorporación de los métodos de aprendizaje (véase 3.2) se realiza, una vez más, preguntando por el valor de una dirección concreta: (<slot><ent>reglas-prolog-aprendidas).

3 Aplicación: el Calendario

El Calendario, también denominado CAP (nombre que proviene de Aprendiz del Calendario, o *Calendar Apprentice*), fue el primer *sistema aprendiz*[2] que se utilizó desde el principio en condiciones reales, sin requerir un periodo de formación guiada [3]. En este documento se introduce una versión simplificada y adaptada del mismo.

La tarea encomendada es la de ayudar a organizar una agenda personalizada interactiva que se acomode a los criterios de organización seguidos por el usuario. Una situación real podría ser la siguiente: Juan quiere reunirse con Luis esta semana y para ello llama por teléfono a la secretaria de Luis; en el transcurso de la conversación la secretaria tiene que decidir el siguiente conjunto de cuestiones: ¿cuándo debería celebrarse la reunión?, ¿en qué sala sería conveniente celebrarla?, ¿cuál es su duración estimada?, ¿debería hacerse hueco en la presente semana para esta reunión?, ¿sería mejor retrasarla una semana o acortarla para poder atenderla lo antes posible?, ¿debería confirmar con Luis esta reunión?, ¿convendría reservar la sala donde tendrá lugar?, ¿debería enviar un mensaje de correo electrónico que le recordara a Juan la reunión?, etc. El Calendario pretende ayudar a gestionar este tipo de decisiones (en último caso, llegará a automatizarlas, tal y como se comentará en 3.2) a partir del aprendizaje realizado sobre las propias decisiones del usuario.

La interfaz consiste en un editor (de tipo emacs) pensado para gestionar lo más cómodamente posible los eventos del Calendario. La figura 4 muestra una de las pantallas básicas del sistema.

El funcionamiento consiste en un bucle contínuo de espera de introducción de alguno de los tipos de solicitud[3]: reunión, salida, seminario y curso. Cada uno de estos tipos contiene un formulario adecuado a las necesidades de un profesor de universidad, que es el destinatario inicial de esta versión del sistema. Las preguntas principales que hay que contestar, con la ayuda del consejo del sistema (mostrado entre corchetes en la figura 4) son: lugar, fecha y duración del evento.Otras preguntas, como los ponentes de un seminario, dependen de cada tipo de solicitud.

3.1 Sin aprendizaje

La respuesta a cada una de las preguntas del formulario comentado previamente depende del conocimiento inicial sobre dicha tarea. Ocurre que los criterios que determinan dichos valores dependen de cada usuario. Es más, ningún usuario sabe de antemano todos los factores que hay que considerar y, mucho menos, la importancia de cada uno en cada situación. Por ejemplo, el tiempo asignado a una reunión con los

[2] Estos sistemas son capaces de aprender de sus usuarios a partir de la observación directa de su comportamiento.

[3] También existen comandos para la propia gestión de la agenda.

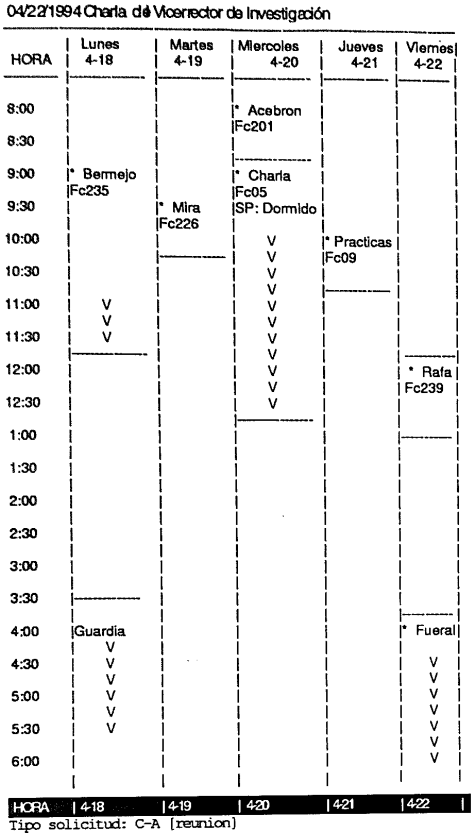

Figura 4: Una de las pantallas del interfaz

alumnos depende de la época del año en curso, ya que en períodos no lectivos la disponibilidad de tiempo (el principal recurso que hay que gestionar) aumenta sensiblemente. No obstante, difícilmente puede decirse de antemano cuál debe ser la duración aconsejada, ya que el aumento estará en función de los propios gustos del usuario y de otros factores variables difíciles de predecir: número de alumnos con reuniones regulares, ausencia del puesto de trabajo por causas diversas (conferencias, reuniones, charlas, etc.), fechas límites de trabajos pendientes, etc.

La dificultad de la tarea no supone que no se puedan tener criterios razonablemente válidos desde el principio. Por ejemplo, el usuario podría asignar una duración de 30 minutos desde junio a septiembre y en los períodos de vacaciones, dejando tan sólo 15 minutos para el resto del año . Todos estas creencias sobre la solución parcial de la tarea pueden constituir el conjunto de reglas de tipo `prolog` asignadas a cada uno de los elementos del formulario. De esta forma, la respuesta inicial del sistema sería lo suficientemente acertada como para ser tenida en cuenta. Este es un factor crítico en la mayoría de los *sistemas aprendices* debido a que, si el comportamiento no fuera útil desde el principio dejaría de utilizarse y no podría aprender de su propio uso. En el caso del Calendario se da además la circunstancia de que la propia interfaz supuso un gran avance con respecto a la herramienta utilizada previamente.

El caso concreto de la determinación de la fecha de los eventos o solicitudes tiene una dificultad especial, ya que la infintud de valores posibles de dicho parámetro impide plantearlo directamente como una tarea de clasificación. En su lugar, se utilizan dos valores intermedios a los que se les denomina `dias-entre` y `dia-semana`. El primero señala el número de días que suele transcurrir entre dos eventos de las mismas características y el segundo refleja la observación de la coinicidencia en el día de celebración de las reuniones de un determinado tipo (p.ej., todos los lunes se realiza una reunión del grupo de trabajo de un determinado proyecto). En ambos casos, el número de valores posibles es limitado. Finalmente, considerando que la regularidad en el día de la semana es más fiable que la de los días transcurridos entre reuniones, el cálculo definitivo de la fecha estimada podría realizarse buscando el día más cercano al sugerido por `dia-semana` que resulta de sumar el tiempo calculado por `dias-entre` a la fecha en que se celebró la última reunión del tipo dado.

La figura 5 muestra parte del contenido del campo `dia-semana`. Como puede apreciarse, en las reglas definidas se realiza la invocación de funciones lisp como `encuentra-reuniones-previas`, que realizan todas las acciones necesarias para concluir cuál es el día de la semana en que suelen celebrarse un determinado tipo de reuniones.

3.2 Con aprendizaje

Las reglas de inferencia descritas en el epígrafe anterior no resuelven satisfactoriamente el problema. De hecho, se ha comprobado que sólo tienen un acierto que se aproxima al 30%. Por tanto, considerando que el planteamiento coin-

```
(dia-semana *sin-valor*
  (generalizaciones (sol-slot))
  (dominio solicitudes)
  (valores-posibles (lunes martes miercoles jueves
                     viernes sabado domingo))
  (metodos (valor-asignado valor-aconsejado))
  (valor-aconsejado *sin-valor*
   (numero-valores 1)
   (clausulas-prolog
    (((valor-aconsejado (?s dia-semana) viernes):-
          (unica-persona ?s si)
          (sol-asistente ?s marisa))
     ((valor-aconsejado (?s dia-semana) martes):-
          (unica-persona ?s si)
          (sol-asistente ?s angel))
     ((valor-aconsejado (?s dia-semana) viernes):-
          (unica-persona ?s no)
          (sol-asistente ?s angel))
     ((valor-aconsejado (?s dia-semana) viernes):-
          (sol-asistente ?s pedro))
      ((valor-aconsejado (?s dia-semana) ?dia-semana):-
          (unica-persona ?s si)
          (sol-asistente ?s ?a)
          (eval (encuentra-reuniones-previas ?s ?a
                (fecha-actual) 2) ?previas)
          (eval (dia-eventos ?previas) ?dia-semana))
     ((valor-aconsejado (?s dia-semana) ?dia-semana):-
          (sol-seminar-tipo ?s ?seminario)
          (sol-ponentes ?s ?ponente)
          (eval (encuentra-seminarios-previos ?s
                ?ponente (fecha-actual) 2) ?previos)
          (eval (dia-eventos ?previos) ?dia-semana))
                    ... ... ... ... ... ... ... ... ...))))
```

Figura 5: Reglas a priori de `dia-semana`

cide con el esquema básico de una tarea de clasificación, se ha optado por aplicar las técnicas de aprendizaje clásicas disponibles en el sistema. En concreto, se ha utilizado una versión evolucianada del conocido ID3 [8].

La selección del conjunto de atributos más predictivos del valor de clase buscado (`dia-semana`) es uno de los factores críticos que determinan la bondad de las reglas obtenidas a partir del árbol de decisión generado por la estrategia de aprendizaje aplicada. Se ha verificado que las mejores soluciones se obtienen con 15 atributos aproximadamente, tanto para esta tarea como para el resto de los parámetros del interfaz. La figura 6 muestra dos de los conjuntos de atributos seleccionados.

Las reglas aprendidas se almacenan siguiendo el formato `prolog` ya expuesto. Llama la atención la sencillez de su estructura (véanse algunas de estas reglas en la figura 7). No obstante, el porcentaje de aciertos llega a ser del 70%. En algunas tareas sencillas, como el aprendizaje del *slot* `sol-lugar`, la naturaleza sesgada de las

Atributos predictivos:
A_1: sol-tipo sol-asistentes
sol-seminar-tipo
sol-ponentes
unico-asistente?
puesto-asistente
sol-curso-nombre conocido?
departamento-asistente

A_2: sol-tipo unico-asistente?
sol-asistentes
puesto-asistente
conocido?

Figura 6: Conjuntos alternativos de atributos

ejemplos tratados (la mayoría de las reuniones se celebran en el despacho del propio usuario) permite superar el 90% de aciertos. Para estos casos se podría descargar al usuario de tomar la decisión correspondiente, obteniéndose una automatización de la tarea. En este contexto existen muchas labores que el sistema puede realizar de forma autónoma; por ejemplo, enviar un mensaje de aviso a todos los asistentes a una reunión.

Si se analiza cualquiera de las gráficas que se han obtenido sobre el aprendizaje de los parámetros del interfaz se observa una clara inflexión en las curvas en el período estival (tal y como se refleja en la figura 8). Esto se debe a que el conjunto de ejemplos de entrenamiento utilizados pertenecen a una época de mayor actividad académica. Como puede apreciarse en la figura 8, las reglas aprendidas generan un comportamiento que se estabiliza cuando el número de ejemplos de entrenamiento es suficiente (en torno a 120 o superior). La traza de "%Aplicables" indica el porcentaje de ejemplos a los que las reglas ID3 son aplicables (porcentaje de ejemplos para los que existe alguna regla aplicable). Por otro lado, la curva de "ID3-Aplicables" representa el porcentaje de aciertos de las reglas ID3 en los casos en que éstas son aplicables.

4 Cuestiones de interés

El hecho de haber utilizado un sistema genérico como THEO permite detectar ciertas cuestiones de interés, entre las que destacan:

- *¿Cuáles son los elementos del diseño que deben representarse explícitamente y cuáles*

```
(dia-semana *sin-valor*
  (generalizaciones (sol-slot))
  (dominio solicitudes)
  (valores-posibles (lunes martes miercoles jueves
          viernes sabado domingo))
  (metodos (valor-asignado valor-aconsejado))
  (reglas-prolog-aprendidas
        (((dia-semana ?s viernes):-
              (unica-persona ?s si)
              (sol-asistente ?s marisa))
         ((dia-semana ?s martes):-
              (unica-persona ?s si)
              (sol-asistente ?s angel))
         ((dia-semana ?s viernes):-
              (unica-persona ?s no))
         ((dia-semana ?s viernes):-
              (unica-persona ?s si)
              (sol-asistente ?s bermejo))
         ((dia-semana ?s lunes):-
              (sol-seminar-tipo ?s ?seminario)
              (sol-ponentes ?s jesus))
         ((dia-semana ?s miercoles):-
              (sol-seminar-tipo ?s ?seminario)
              (sol-ponentes ?s jose-luis)))))
```

Figura 7: Reglas aprendidas

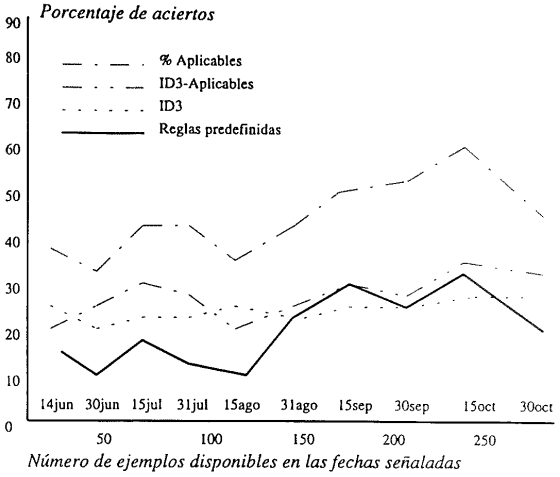

Figura 8: Evolución del aprendizaje

deberían ser inferidos? La resolución paulatina de problemas provoca la expansión continuada de la red de *slots* que reflejan las relaciones válidas entre las entidades del dominio. Así, cuantos más *slots* se definan y almacenen, menor número de inferencias habrá que realizar en el cálculo de nuevos *slots*. Suponiendo que se considerara el funcionamiento del sistema como el de una "tela de araña en construcción", al principio sólo habría unas pocas direcciones asignadas, que indicarían por dónde se puede expandir el sistema (la tela de araña). Según el sistema (la araña) fuera teniendo interés en crecer en alguna dirección (se le pregunte), éste empezaría a crecer en la misma. Si no se almacena el camino recorrido (la tela construida) habría que volverlo a recorrer en sucesivas demandas. Por tanto, la conveniencia o no de guardar una determinada entidad depende del número de peticiones que la soliciten y de la eficiencia requerida tanto para su almacenamiento (cantidad de memoria disponible) como para su gestión (cuanto más conocimiento haya almacenado más se tardará en comprobar cuál es el pertinente).

- *¿Cuándo y qué debe aprenderse?* No debe olvidarse que el aprendizaje es, al fin y al cabo, una forma alternativa de inferencia. En este sentido, cuando la inferencia del valor de un atributo tenga una complejidad considerable (como es el caso de la fecha de la reunión) puede optarse por aprender otros valores relacionados más simples, dejando que la decisión final dependa de una combinación ponderada de aquéllos.

- *¿Cuáles y cómo deben integrarse los distintos métodos de inferencia?* En el esquema de representación adoptado, cualquiera de los métodos de inferencia utilizados tiene por objeto señalar las direcciones donde se puede encontrar el valor del *slot* solicitado. Por tanto, su integración es inmediata; no importa la secuencia de pasos asociados a cada método (reglas, funciones, conocimiento aprendido, etc.) todos ellos revierten en accesos y asignaciones de valores a *slots* de entidades. La integración de métodos se realiza a través de la definición del *subslot* `metodos`.

En definitiva, la posibilidad real de experimentar con el sistema Calendario, desarrollado sobre THEO, permite estudiar la conjugación de diferentes formas de inferencia con técnicas alternativas de aprendizaje en la solución de tareas concretas. Esto, junto con la flexibilidad de un sistema construido en Lisp, justifica su interés para integrar los contenidos de asignat-

uras de programación, de aprendizaje y de fundamentos en IA.

Referencias

[1] Jesús G. Boticario. *Contribuciones a la Teoría Computacional del Aprendizaje: Ejemplos de Aplicación.* PhD thesis, Facultad de Ciencias de la Universidad Nacional de Educación a Distancia, July 1994.

[2] Jaime G. Carbonell, Craig A. Knoblock, and Steven Minton. Prodigy: An integrated architecture for planning and learning. In K. VanLehn, editor, *Architectures for Intelligence.* Erlbaum, Hillsdale, NJ, 1990. Also Technical Report CMU-CS-89-189.

[3] L. Dent, J. G. Boticario, J. McDermott, T. M. Mitchell, and D. T. Zabowski. A personal learning apprentice. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 96–103, San Jose, CA. Mit Press.

[4] S. F. Galán, J. G. Boticario, J. Mira, and F. J. Díez. *Problemas Resueltos de Inteligencia Artificial Aplicada: Búsqueda y Representación.* Addison-Wesley Iberoamericana España S.A., Madrid, 1998.

[5] T. M. Mitchell, R. Caruana, D. Freigat, J. McDermott, and D. Zabowski. Experience with a learning personal assistant. *Communications of the ACM*, 37:80–91, 1994.

[6] Tom M. Mitchell. Becoming increasingly reactive. In *Proceedings of Eighth National Conference on Artificial Intelligence*, pages 1051–1058, Cambridge, MA, 1990. AAAI Press/The MIT Press.

[7] Tom M. Mitchell, John Allen, Prasad Chalasani, John Cheng, Oren Etzioni, Marc Ringuette, and Jeffrey C. Schlimmer. Theo: A framework for self-improving systems. In K. VanLehn, editor, *Architectures for Intelligence.* Erlbaum, Hillsdale, NJ, 1990.

[8] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[9] Paul S. Rosenbloom, Allen Newell, and John E. Laird. Towards the knowledge level in SOAR: The role of the architecture in the use of knowledge. In K. VanLehn, editor, *Architectures for Intelligence.* Erlbaum, Hillsdale, NJ, 1990.

[10] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstucture of Cognition.* MIT Press, Cambridge, MA, 1986.