4 PROLOG EN LA IA

Este capítulo lo dedicaremos a estudiar dos aplicaciones de Prolog en la IA: La resolución de problemas como una búsqueda en un espacio de soluciones; y la inducción de árboles de decisión a partir de datos. Ambos problemas tienen una noble tradición en nuestra área de estudio. La hipótesis de la búsqueda heurística de Newell y Simon [62], establece que las soluciones a los problemas se representan como estructuras simbólicas; y que un sistema simbólico físico ejerce su capacidad de resolver problemas inteligentemente, mediante una búsqueda –Esto es, generando y modificando progresivamente estructuras simbólicas hasta producir una estructura solución. La primer sección del capítulo aborda una serie de algoritmos que buscan soluciones en espacios incrementalmente más ricos, hasta llegar al conocido algoritmo A* [37].

Por otra parte, la inducción de árboles de decisión a partir de datos, es una de las técnicas de aprendizaje automático más usadas en la práctica. La idea subyacente en los diferentes algoritmos de inducción es la de una búsqueda descendiente (de la raíz a las hojas) y egoísta (primero el mejor, sin reconsiderar), en el espacio de todos los posibles árboles de decisión. La segunda sección del capítulo introduce la técnica, con base en la presentación de Mitchell [60]. Posteriormente implementamos una versión de ID3 [71].

Esta presentación complementa el contenido de los cursos de Introducción a la Inteligencia Artificial y Aprendizaje Automático, donde estos algoritmos son revisados con mayor detalle. Al final del capítulo se incluyen las referencias pertinentes para abundar en ese sentido.

4.1 BÚSQUEDAS EN ESPACIOS DE SOLUCIONES

Primero abordaremos un esquema general de representación de problemas y sus soluciones, ampliamente utilizado en la IA. Consideremos el ejemplo mostrado en la figura 4.1. El problema a resolver consiste en encontrar un plan para colocar los cubos en una configuración determinada, partiendo de una configuración inicial. Sólo un bloque puede moverse a la vez y las acciones del brazo son del tipo "pon *A* en la *mesa*", "pon *B* en *C*", etc. Dos conceptos aparecen en esta descripción: i) **Estados** del problema; y ii) **Acciones** que transforman un estado del problema en otro.

Estados y Acciones

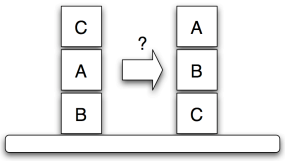

Figura 4.1: *El mundo de los bloques.*

Estados y acciones configuran un grafo dirigido conocido como **espacio de estados**. La figura 4.2, muestra el espacio de estados para tres bloques. El problema de encontrar un plan para acomodar los cubos es equivalente a encontrar un camino en este grafo, entre un nodo representado el estado inicial del problema y un nodo representando la solución final, un **nodo meta**. ¿Cómo podemos representar tal grafo en Prolog?

Espacio de estados

Nodo meta

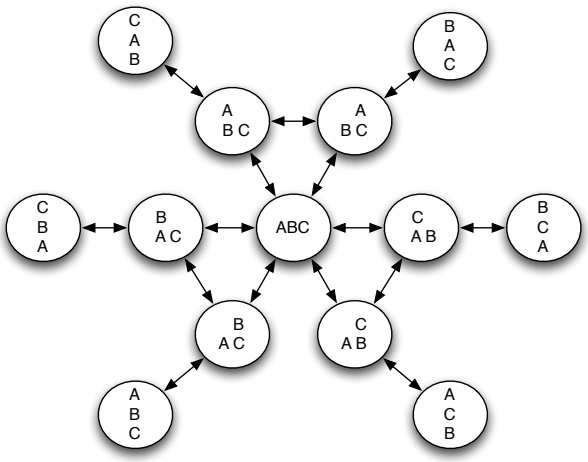

Figura 4.2: *Espacio de estados para el mundo de los bloques.*

El espacio de estados será representado por una relación $s(X, Y)$ que será verdadera si existe un movimiento válido en el espacio de estados del nodo X al nodo Y. El nodo Y recibe el nombre de **sucesor** de X. Si existe un costo asociado a las acciones esto puede representarse por un tercer argumento de s, $s(X, Y, Costo)$. *Sucesor*

Esta relación puede ser especificada extensionalmente por un conjunto de hechos. Sin embargo, para cualquier problema interesante, esto es irrealizable. La relación s es normalmente definida intensionalmente mediante un conjunto de reglas que computan el sucesor de un nodo dado.

Otro detalle importante tiene que ver con la representación de los estados del problema, los nodos. La representación debe de ser compacta y permitir la computación eficiente de los nodos sucesores; y posiblemente el costo asociado a las acciones.

Ejemplo 4.1. *Tomemos como ejemplo el mundo de los bloques. Cada estado del problema puede ser representado por una lista pilas. Cada pila a su vez puede ser representada por una lista de bloques. El tope de cada pila es el primer elemento de cada lista de bloques. La pila vacía está representada por la lista vacía. Así, el estado inicial mostrado en la figura 4.1 es la lista:* `[[c,b,a],[],[]]` *(suponemos, que en la mesa sólo hay espacio para 3 pilas de bloques).*

Una meta es cualquier arreglo con los bloques en el orden deseado. Existen tres soluciones en este caso:

- `[[a,b,c],[],[]]`
- `[[],[a,b,c],[]]`
- `[[],[],[a,b,c]]`.

Primero definiremos *quitar*/3:

```
1  quitar(X, [X|Ys], Ys).
2  quitar(X, [Y|Ys], [Y|Ys1]) :-
3          quitar(X,Ys,Ys1).
```

una relación que nos permite computar cual es la lista resultante de quitar un elemento a una lista:

```
1  ?- quitar(a,[a,b,c],R).
2  R = [b, c] ;
3  false.
4
5  ?- quitar(a,[b,a,c],R).
6  R = [b, c] ;
```

```
false.

?- quitar(a,[b,c,a],R).
R = [b, c] ;
false.
```

qué elemento puedo quitar de una lista y cual sería el resultado de ello:

```
?- quitar(X,[a,b,c],R).
X = a,
R = [b, c] ;
X = b,
R = [a, c] ;
X = c,
R = [a, b] ;
false.
```

O qué elemento debo quitar para obtener la lista deseada:

```
?- quitar(X,[a,b,c],[a,b]).
X = c ;
false.
```

Ahora podemos definir nuestra función sucesor *s*, en términos de quitar: el *Estado*2 es sucesor de *Estado*1 si hay dos pilas *Pila*1 y *Pila*2 en *Estado*1 y el tope de la pila *Pila*1 puede moverse a *Pila*2. Esto se traduce a Prolog como:

```
  s(Pilas, [Pila1, [Tope1|Pila2] | OtrasPilas ]) :-
          quitar([Tope1|Pila1], Pilas, Pilas1),
          quitar(Pila2, Pilas1, OtrasPilas).
```

La relación *s* nos permite verificar si un nodo es sucesor de otro, por ejemplo:

```
?- s([[b],[a,c],[]],[[],[b,a,c],[]]).
Yes
?- s([[b],[a,c],[]],[[],[a,b,c],[]]).
No
```

o computar quienes son sucesores de un estado dado:

```
?- s([[c,a,b],[],[]],Sucesores).
Sucesores = [[a, b], [c], []] ;
Sucesores = [[a, b], [c], []] ;
false.

?- s([[a],[b,c],[]],Sucesores).
Sucesores = [[], [a, b, c], []] ;
Sucesores = [[], [a], [b, c]] ;
Sucesores = [[c], [b, a], []] ;
Sucesores = [[c], [b], [a]] ;
false.
```

Para representar los estados meta usamos:

```
meta(Estado) :-
  member([a,b,c],Estado).
```

asumiendo que nuestra meta sea formar una pila con los bloques en ese orden, Si se desea una meta diferente, basta con modificar la lista `[a,b,c]` por la pila de bloques deseada.

4.1.1 Búsqueda primero en profundidad

Dada la formulación de un problema en términos de su espacio de estados, existen diversas estrategias para encontrar un camino solución. Dos estrategias básica son las búsquedas primero en profundidad y primero en amplitud. En esta sección implementaremos la búsqueda primero en profundidad.

Comenzaremos con una idea simple. Para encontrar un camino solución Sol, a partir de un nodo dado N a un nodo meta, ejecutar el siguiente **algoritmo**:

Algoritmo primero en profundidad

- Si N es un nodo meta, entonces $Sol = [N]$, o
- Si existe un nodo sucesor $N1$ tal que existe un camino $Sol1$ de $N1$ al nodo meta, entonces $Sol = [N|Sol1]$.

Lo cual traduce a Prolog como:

```
1  solucion(N,[N]) :-
2    meta(N).
3
4  solucion(N, [N|Sol1]) :-
5          s(N,N1),
6          solucion(N1,Sol1).
```

De forma que para computar la solución al problema de los bloques, preguntamos a Prolog:

```
1  ?- solucion([[c,b,a],[],[]],Sol).
2  Sol = [[[c, b, a], [], []],
3         [[b, a], [c], []],
4         [[a], [b, c], []],
5         [[], [a, b, c], []]]
6  Yes
```

La solución se computa como sigue. En un principio, el estado inicial $N = [[c, b, a][][]]$, por lo que el programa se pregunta si N es una meta. La cláusula $meta/1$ funciona verificando si la solución $[a, b, c]$ es miembro del estado N. Como esta meta falla, Prolog intentará satisfacer su meta inicial con la segunda cláusula $solucion/2$. Esto implica generar un sucesor de N (llamada a $s(N, N1)$). Así que se computa $N1 = [[b, a], [c], []]$ y se verifica si esto es una solución. Como la meta falla, se genera un sucesor de $N1$ y así hasta llegar a $[[], [a, b, c], []]$.

Este proceso puede seguirse detalladamente utilizando el *tracer* gráfico de SWI-Prolog. Para ello invoquen la meta `guitracer`. Al trazar una función verán una ventana como la mostrada en la figura 4.3. La ventana superior izquierda muestra las substituciones computadas, la derecha las pilas formadas, y la inferior muestra el código del programa que está siendo trazado.

Evidentemente, si $s/2$ falla, quiere decir que no hay más sucesores disponibles, y la meta no se ha alcanzado; por lo que la búsqueda en si falla, regresando falso como respuesta. Otra ejecución posible es que la memoria se nos agote, antes de alcanzar un nodo meta. En este caso, la salida será un mensaje de error avisando que la pila de ejecución ha desbordado la memoria. Observen que mientras lo primero, es un comportamiento deseable en nuestro programa, lo segundo no lo es. Dos **mejoras** pueden contribuir a la completez de nuestras búsquedas: Evitar ciclos y limitar la profundidad de la búsqueda.

Mejoras

La primer mejora consiste en evitar que los nodos visitados vuelvan a ser expandidos, evitando así caer en ciclos. La idea es llevar un registro de los nodos visitados:

```
1  solucion2(Nodo,Sol) :-
2          primeroProfundidad([],Nodo,Sol).
3
```

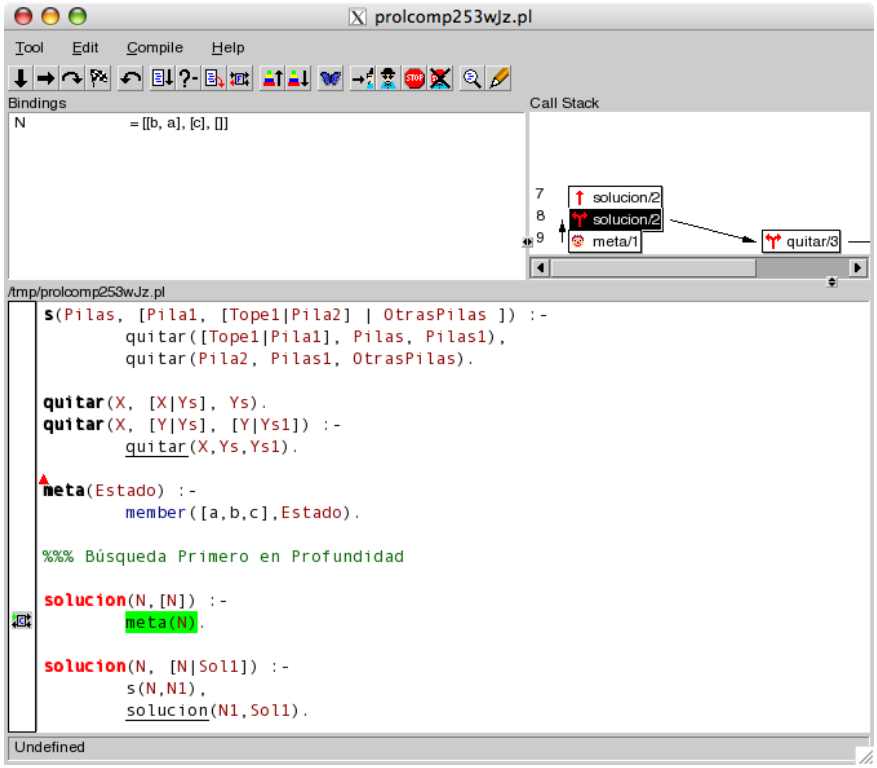

Figura 4.3: *Traza gráfica de SWI-Prolog.*

```
4   primeroProfundidad(Camino, Nodo, [Nodo|Camino]) :-
5           meta(Nodo).
6
7   primeroProfundidad(Camino, Nodo, Sol) :-
8           s(Nodo,Nodo1),
9           \+ member(Nodo1, Camino),
10          primeroProfunidad([Nodo|Camino],Nodo1,Sol).
```

Finalmente, para evitar caer en búsquedas infinitas sobre ramas no cíclicas, es posible establecer un límite a la profundidad de la búsqueda. Para ello definiremos *primeroProfundidad2/3*, donde el tercer argumento es la profundidad máxima de la búsqueda.

```
1   solucion3(Nodo,Sol,MaxProf) :-
2           primeroProfundidad2(Nodo,Sol,MaxProf).
3
4   primeroProfundidad2(Nodo,[Nodo],_) :-
5           meta(Nodo).
6
7   primeroProfundidad2(Nodo,[Nodo|Sol],MaxProf):-
8           MaxProf > 0,
9           s(Nodo,Nodo1),
10          Max1 is MaxProf-1,
11          primeroProfundidad2(Nodo1,Sol,Max1).
```

4.1.2 Búsqueda primero en amplitud

En contraste con la búsqueda primero en profundidad, la estrategia de búsqueda primero en amplitud elige visitar primero los nodos que están más cerca de la raíz, por lo que el árbol de búsqueda crece más en amplitud, que en profundidad. Esta estrategia de búsqueda es más complicada de programar. La razón de ello es que debemos mantener un conjunto de **caminos candidatos** alternativos, no únicamente

Caminos candidatos

un camino como lo hacíamos al buscar en profundidad. De forma que:

```
?- primeroEnProfundidad(Caminos,Sol).
```

es verdadera, si y sólo si algún camino miembro del conjunto de candidatos *Caminos*, puede extenderse hasta un nodo meta. *Sol* es el camino solución.

El conjunto *Caminos* será representado como listas de caminos, donde cada uno de ellos se representará como una lista de nodos en el orden inverso en que fueron visitados. Esto es, la cabeza de la lista que representa un camino será el último nodo generado; y el último elemento de la lista, será el estado inicial de la búsqueda. Al iniciar *Caminos* tiene un sólo camino candidato: *[[NodoInicial]]*.

Algoritmo primero en amplitud

El **algoritmo** de la búsqueda primero en amplitud puede describirse como sigue, dado un conjunto de caminos candidatos:

- Si el primer camino tiene como cabeza un nodo meta, entonces esta es la solución al problema. De otra forma
- Eliminar el primer camino del conjunto de caminos candidatos y generar el conjunto de todas las posibles extensiones de un paso de ese camino. Agregar este conjunto de extensiones al final del conjunto de candidatos. Ejecutar la búsqueda primero en amplitud en este nuevo conjunto de caminos candidatos.

Para generar las extensiones de un sólo paso, dado un camino, podemos usar el predicado predefinido *bagof/3* Veamos el programa:

```
%%% solucion(Inicio,Sol) Sol es un camino (en orden
%%% inverso)  de Inicio  a una meta

solucion(Inicio,Sol) :-
        primeroEnAmplitud([[Inicio]],Sol).

%%% primeroEnAmplitud([Camino1,Camino2,...],Sol)
%%% Sol es una extensión hacía la meta de alguno
%%% de los caminos

primeroEnAmplitud([[Nodo|Camino]|_],[Nodo|Camino]) :-
        meta(Nodo).

primeroEnAmplitud([Camino|Caminos],Sol) :-
        extender(Camino,NuevosCaminos),
        append(Caminos,NuevosCaminos,Caminos1),
        primeroEnAmplitud(Caminos1,Sol).

extender([Nodo|Camino],NuevosCaminos) :-
  bagof([NuevoNodo,Nodo|Camino],
        (s(Nodo,NuevoNodo), \+ member(NuevoNodo, [Nodo|Camino])),
        NuevosCaminos),
  !.

%%% Si extender falla, Camino no tiene sucesores (lista vacía)

extender(Camino_,[]).
```

Si aplicamos este programa de búsqueda al programa del mundo de los cubos, obtendremos:

```
?- solucion([[c,b,a],[],[]],Sol).
Sol = [[[], [a, b, c], []], [[a], [b, c], []],
       [[b, a], [c], []], [[c, b, a], [], []]]
Yes
```

Si queremos buscar en el espacio del grafo de la figura 4.4, codificamos los sucesores y las metas como sigue:

```
s(a,b).
s(a,c).
s(b,d).
s(b,e).
s(d,h).
s(e,i).
s(e,j).
s(c,f).
s(c,g).
s(f,k).

meta(j).
meta(f).
```

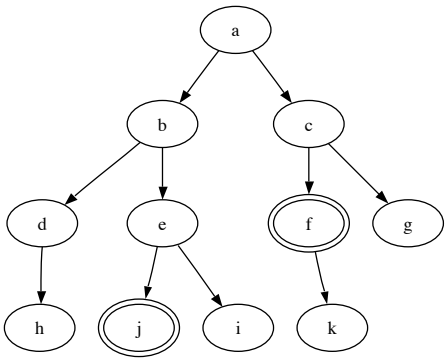

Figura 4.4: Gráfico de un espacio de estados: las metas son f y j.

y buscamos las soluciones:

```
?- solucion(a,Sol).
Sol = [f, c, a] ;
Sol = [j, e, b, a] ;
No
```

Observen que al realizarse la búsqueda primero en amplitud, la primer solución encontrada involucra la meta f más cercana al nodo raíz.

4.1.3 Búsqueda primero el mejor

Un programa de búsqueda primero el mejor, puede verse como una mejora a la búsqueda primero en amplitud. El algoritmo de primero el mejor comienza también con un nodo inicial y mantiene una lista de caminos candidato. La búsqueda por amplitud siempre elige para expandir el camino candidato más corto y la búsqueda primero el mejor afina esta estrategia.

costo

Asumamos que una función **costo** es definida para los arcos de un espacio de estados de un problema. De forma que $c(n, n')$ es el costo de moverse de un nodo n al nodo n' en el espacio de estados.

Estimador heurístico

Sea el **estimador heurístico** una función f tal que para cada nodo n en el espacio de estados, $f(n)$ estima la "dificultad" de llegar a n. De acuerdo a esto, el nodo más promisorio será aquel que minimice f. Usaremos aquí una forma especial de la función f que nos llevará al bien documentado **algoritmo A*** [37]. $f(n)$ será construida para estimar el costo del mejor camino solución entre un nodo inicial s y un nodo meta, con la restricción de que el camino pase por el nodo n. Supongamos

*A**

que tal camino existe y que un nodo meta que minimiza su costo es t. Entonces el estimado de $f(n)$ puede calcularse como la suma de dos términos:

$$f(n) = g(n) + h(n)$$

donde $g(n)$ es el estimado del costo de un camino óptimo de s a n; y $h(n)$ es el estimado del costo de un camino óptimo de n a t (Fig. 4.5).

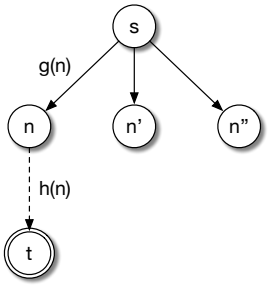

Figura 4.5: Estimado heurístico $f(n) = g(n) + h(n)$.

Cuando un nodo n es encontrado en el proceso de búsqueda, tenemos la siguiente situación: un camino de s a n debió ser encontrado, y su costo puede computarse como la suma del costo de cada arco en el camino. Este camino no es necesariamente un camino óptimo entre s y n (puede haber caminos mejores no cubiertos aún por la búsqueda), pero su costo puede servir como un estimador $g(n)$ del costo mínimo de ir de s a n. El otro término, $h(n)$, es más problemático, porque el espacio entre n y t no ha sido explorado aún, por lo que su valor es una verdadera adivinanza heurística, resuelta con base en el conocimiento general del algoritmo sobre la estructura particular del problema a resolver. Como h depende del dominio del problema, no existe un método universal para su construcción. Asumamos por el momento que una función h nos es dada y concentrémonos en los detalles del programa primero el mejor.

Como ejemplo consideren el siguiente problema. Dado un mapa (Fig. 4.6), la tarea es encontrar la ruta más corta entre una ciudad inicial s y una ciudad meta t. Al estimar el costo del resto del camino de la ciudad X a la meta, usamos simplemente la distancia lineal denotada por $dist(X, t)$. Entonces:

$$f(X) = g(X) + h(X) = g(X) + dist(X, t)$$

En este ejemplo, podemos imaginar la búsqueda de primero el mejor consistente de dos procesos, cada uno de ellos explorando uno de los dos caminos alternativos: El proceso 1 para el camino vía a y el proceso 2 para el camino vía e. En los pasos iniciales el proceso 1 está más activo porque los valores f en ese camino son más bajos que los del otro. En el momento en que el proceso 1 llega a c y el proceso 2 sigue en e, la situación cambia:

$$f(c) = g(c) + h(c) = 6 + 4 = 10$$

$$f(e) = g(e) + h(e) = 2 + 7 = 9$$

De forma que $f(e) < f(c)$ y ahora el proceso 2 procede al nodo f y el proceso 1 espera. Pero entonces:

$$f(f) = 7 + 4 + 11$$

$$f(c) = 10$$

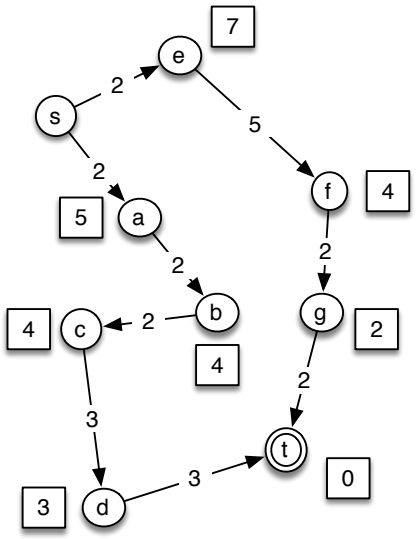

Figura 4.6: Mapa entre ciudades, sus distancias por carretera, y sus distancias líneales a la meta (cuadros).

$$f(c) < f(f)$$

por lo que el proceso 2 es detenido y se le permite al proceso 1 continuar, pero sólo hasta el nodo d ya que $f(d) = 12 > 11$. El proceso 2 continua corriendo hasta llegar a la meta t (Fig. 4.7).

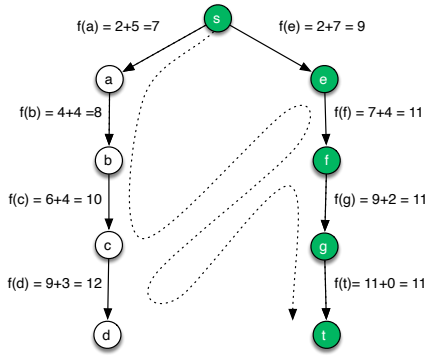

Figura 4.7: La búsqueda en el mapa de las ciudades.

Este proceso de búsqueda parte del nodo inicial (la ciudad s) y genera nuevos nodos sucesores, expandiéndose siempre en la dirección más promisoria de acuerdo a los valores de la función f. Esto genera un árbol que crece hasta llegar a un nodo meta. Este **árbol** será representado en nuestro programa de búsqueda primero el mejor por términos de dos formas:

Árbol de búsqueda

1. $l(N, F/G)$ representa una hoja del árbol, donde N es un nodo en el espacio de estados; G es $g(N)$, el costo de haber llegado a N; y F es el estimador heurístico $f(N) = G + h(N)$. A F lo llamaremos en lo subsecuente f-valor.

2. $t(N, F/G, Subs)$ representa un nodo interno del árbol, con una lista de subárboles $Subs$ no vacíos. F es el f-valor del sucesor más prometedor de N. La lista $Subs$ se ordena de forma decreciente de acuerdo al f-valor.

Por ejemplo, consideren nuevamente la búsqueda ilustrada en la figura 4.7. En el momento en que el nodo s es expandido, el árbol de búsqueda consta de tres nodos: el nodo s y sus hijos a y e. En nuestro programa, este árbol se representa como:

```
1 t(s,7/0,[l(a,7/2),l(e,9/2)]
```

El valor f para s es 7, esto es, el valor más promisorio de los hijos de s (a). El árbol crece expandiendo el nodo más promisorio. El más cercano competidor de a es e con un f valor de 9. Se permite que a crezca mientras su f valor no exceda 9. Por lo tanto los nodos b y c son generados, pero c tiene un f valor de 10, lo cual excede el umbral de crecimiento fijado en 9. En ese momento no se permite que a crezca más. En ese momento el árbol es:

```
t(s,9/0,[l(e,9/2),t(a,10/2,[t(b,10/4,[l(c,10/6)])])])
```

Observen que ahora el valor f del nodo a es 10, mientras que el del nodo e es 9. Estos valores se actualizaron porque fueron generados los nodos b y c. Ahora el nodo sucesor más promisorio de s es s es e con un valor f de 9.

La actualización de los f valores es necesaria para permitir al programa reconocer el subárbol más promisorio en cada nivel del árbol de búsqueda (esto es, el subárbol que contiene la hoja más promisoria). Esta modificación de los estimados de f, nos lleva a la generalización de la definición de f que extiende su definición de nodos a árboles. Para una hoja n del árbol, mantenemos la definición original:

$$f(n) = g(n) + h(n)$$

Para un subárbol T, cuya raíz es n y tiene como subárboles $S_1, S_2, \ldots$:

$$f(T) = \min_i f(S_i)$$

El programa que implementa la búsqueda primero el mejor es como sigue. Primero definimos una función interfaz, que encuentre la solución *Sol* a partir de un estado inicial *Inicio*. Para ello *solucion*/2 llama a *expandir*/6:

```
solucion(Inicio,Sol) :-
  expandir([],l(Inicio,0/0),9999,_,si,Sol).
```

El predicado *expandir*/6 se encarga de hacer crecer el árbol de búsqueda. Sus argumentos incluyen:

- El *Camino* recorrido, inicialmente vacío;
- El *Arbol* actual de búsqueda, inicialmente una hoja con el nodo *Inicio* y valor de 0 para F y G;
- El *Umbral* o limite para la expansión del árbol (f-valor máximo), para este ejemplo 9999 es suficiente (ningún costo en el árbol será mayor que este valor);
- El *Arbol*1 expandido bajo el *Umbral* (en consecuencia el f-valor de este árbol es mayor, al menos que se halla encontrado la solución). Originalmente se pasa una variable anónima en la llamada;
- La bandera *Resuelto* que puede tomar los valores *si*, *no*, o *nunca*;
- y la solución, si existe, al problema regresado en la variable *Sol*.

El crecimiento del árbol se programa por casos. El caso más simple corresponde a aquel donde árbol de búsqueda es una hoja, y su *Nodo* es una meta del espacio de estados. En ese caso $[Nodo|Camino]$ es la solución *Sol* buscada. Observen la bandera $Resuelto = si$.

```
expandir(Camino,l(Nodo,_),_,_,si,[Nodo|Camino]) :-
  meta(Nodo).
```

El segundo caso corresponde a un árbol de búsqueda que es una hoja, cuyo *Nodo* no es una meta del espacio de estados y tiene un f-valor menor (o igual) que el *Umbral*. Para ello se generan los árboles sucesores del árbol de búsqueda actual (*Arboles*) usando el predicado *listaSucc*/3. El árbol debe *expandir*/6 o fallar con *Resuelto* = *nunca*.

```
expandir(Camino,l(Nodo,F/G),Umbral,Arbol1,Resuelto,Sol) :-
  F  =<  Umbral,
  (bagof( Nodo2/C,(s(Nodo,Nodo2,C),not(member(Nodo2,Camino))),Succ),
  !,
  listaSuccs(G,Succ,As),
  mejorF(As,F1),
  expandir(Camino,t(Nodo,F1/G,As),Umbral,Arbol1,
           Resuelto,Sol)
  ;
  Resuelto = nunca).
```

El tercer caso es parecido, pero el *Nodo* es interno.

```
expandir(Camino,t(Nodo,F/G,[Arbol|Arboles]),Umbral,Arbol1,Resuelto,Sol) :-
  F =< Umbral,
  mejorF(Arboles,MejorF),
  min(Umbral,MejorF,Umbral1),
  expandir([Nodo|Camino],Arbol,
           Umbral1,Arbol1,Resuelto1,Sol),
  continuar(Camino,t(Nodo,F/G,[Arbol1|Arboles]),
            Umbral,Arbol1,Resuelto1,Resuelto,Sol).
```

El caso cuatro cubre los puntos muertos, cuando no hay solución al problema:

```
expandir(_,t(_,_,[]),_,_,nunca,_) :- !.
```

El caso cinco define la situación cuando el f-valor es mayor que el *Umbral* y se inhibe el crecimiento del árbol:

```
expandir(_,Arbol,Umbral,Arbol,no,_):-
  f(Arbol,F),F>Umbral.
```

continuar/7 decide como procede la búsqueda de acuerdo al árbol expandido. Si una solución *Sol* se ha encontrado, se regresa este valor. En cualquier otro caso, la expansión continua dependiendo del valor de *Resuelto* (no o nunca).

```
continuar(_,_,_,_,si,si,Sol).

continuar(Camino,t(Nodo,F/G,[A1|Arboles]),
          Umbral,Arbol1,no,Resuelto,Sol) :-
  insert(A1,Arboles,NodoArboles),
  mejorF(NodoArboles,MejorF),
  expandir(Camino,t(Nodo,F/G,NodoArboles),
           Umbral,Arbol1,Resuelto,Sol).

continuar(Camino,t(Nodo,F/G,[_|Arboles]),
          Umbral,Arbol1,nunca,Resuelto,Sol) :-
  mejorF(Arboles,MejorF),
  expandir(Camino,t(Nodo,MejorF/G,Arboles),
           Umbral,Arbol1,Resuelto,Sol).
```

Las siguientes funciones son auxiliares:

```
  listaSucc(_,[],[]).
  listaSucc(G0, [N/C|NCs], Arboles) :-
          G is G0+C,
```

```
4           h(N,H),
5           F is G+H,
6           listaSucc(G0,NCs,Arboles1),
7           inserta(l(N,F/G,Arboles1),Arboles).
8
9   inserta(Arbol,Arboles,[Arbol|Arboles]) :-
10    f(Arbol,F), mejorF(Arboles,F1),
11    F =< F1, !.
12  inserta(Arbol,[Arbol1|Arboles], [Arbol1|Arboles1]) :-
13    inserta(Arbol,Arboles,Arboles1).
14
15  f(l(_,F/_),F).
16  f(t(_,F/_,_),F).
17
18  mejorF([Arbol|_],F) :-
19          f(Arbol,F).
20  mejorF([],9999).
21
22  min(X,Y,X) :-
23          X =< Y, !.
24  min(_,Y,Y).
```

4.2 INDUCCIÓN DE ÁRBOLES DE DECISIÓN

En esta sección abordaremos la solución a un problema en el contexto del aprendizaje automático, ejemplificado con el algoritmo **ID3** (*Inductive Dicotomizer*) [71]. Este algoritmo induce árboles de decisión a partir de ejemplos conformados como un conjunto de pares atributo–valor, para predecir el valor de uno de los atributos, conocido como la clase. El aprendizaje de árboles de decisión es una de las técnicas de inferencia inductiva más usadas. Se trata de un método para aproximar funciones de valores discretos, capaz de expresar hipótesis disyuntivas y robusto al ruido en los ejemplos de entrenamiento. La descripción que se presenta en este capítulo, cubre una familia de algoritmos para la inducción de árboles de decisión que incluyen ID3 y C4.5 [73].

ID3

Estos algoritmos llevan a cabo su búsqueda de hipótesis en un espacio completamente expresivo, evitando así los problemas asociados a los espacios de hipótesis incompletos. Como veremos, el sesgo inductivo en este caso, consiste en la preferencia por árboles pequeños, sobre árboles grandes. Un árbol así aprendido, puede representarse también como un conjunto de reglas *si-entonces*, más fáciles de entender para un usuario.

4.2.1 Representación de los árboles de decisión

La figura 4.8 muestra un árbol de decisión. Cada nodo del árbol está conformado por un **atributo** y puede verse como la pregunta: ¿Qué valor tiene este atributo en el caso que deseamos clasificar? Las ramas que salen de los nodos, corresponden a los posibles **valores** del atributo correspondiente.

Atributo

Valor

Un árbol de decisión clasifica cada caso que se le presenta, filtrándolo de manera descendente (Observen que el árbol está dibujado de cabeza), hasta encontrar una hoja, que corresponde a la **clase** buscada.

clase

Ejemplo 4.2. *Consideren el proceso de clasificación del siguiente caso, que describe un día en particular:*

$$\langle\, cielo = soleado, temperatura = caliente, humedad = alta, viento = fuerte \,\rangle$$

Como el atributo cielo, *tiene el valor* soleado *en el caso, procedemos por la rama de la izquierda. Como el atributo* humedad, *tiene el valor* alta, *procedemos nuevamente por rama*

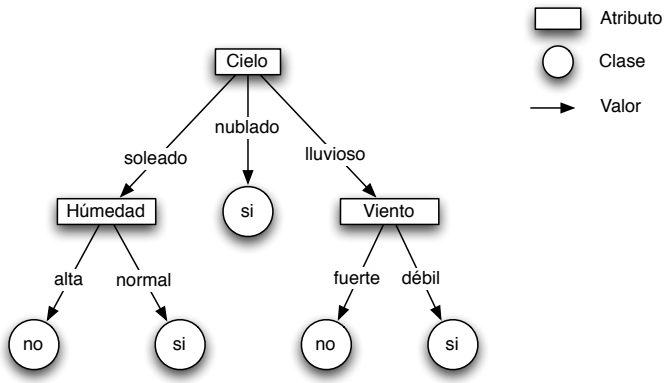

Figura 4.8: *Un ejemplo de árbol de decisión para el concepto "buen día para jugar tenis". Adaptado del artículo original de Quinlan [71].*

de la izquierda, lo cual nos lleva a la hoja que indica la clase del caso: buen día para jugar tenis = no.

El Algoritmo 4.1, define esta idea. La función `tomaValor` encuentra el valor de un atributo, en el caso que se está clasificando. El predicado `hoja` es verdadero si su argumento es un nodo terminal del árbol y falso si se trata de un nodo interno. La función `subArbol` se mueve por la rama del árbol que corresponde al valor del atributo probado en el caso. De esta forma, obtiene un sub-árbol. En nuestro ejemplo, a partir del nodo raíz *cielo*, esta función obtiene el sub-árbol que resulta de moverse por la rama *soleado*, etc.

Algoritmo 4.1 El algoritmo clasifica, para árboles de decisión

```
1: function CLASIFICA(Caso, Arbol)
Require: Caso: un caso a clasificar, Arbol: un árbol de decisión
Ensure: Clase: la clase del caso
2:     Clase ← tomaValor(raiz(Arbol), Caso);
3:     if hoja(raíz(Arbol)) then
4:         return Clase
5:     else
6:         clasifica(Caso, subArbol(Arbol, Clase));
7:     end if
8: end function
```

En general, un árbol de decisión representa una **disyunción de conjunciones** de restricciones en los posibles valores de los atributos de los casos. Cada rama que va de la raíz del árbol a una hoja, representa una conjunción de tales restricciones y el árbol mismo representa la disyunción de esas conjunciones.

Carácter disyuntivo-conjuntivo

Ejemplo 4.3. *Por ejemplo, la hipótesis de un buen día para jugar tenis, contenida en el árbol de la figura 4.8, puede expresarse como sigue:*

$$\begin{aligned} & (cielo = soleado \ \wedge \ humedad = normal) \\ \vee \ & (cielo = nublado) \\ \vee \ & (cielo = lluvia \ \wedge \ viento = débil) \end{aligned}$$

4.2.2 Problemas apropiados

Aun cuando se han desarrollado diversos métodos para la inducción de árboles de decisión, y cada uno de ellos ofrece diferentes capacidades, en general estos algoritmos son apropiados para solucionar problemas de aprendizaje conocidos como **problemas de clasificación**. Estos problemas presentan las siguientes características:

Clasificación

- Ejemplos representados por pares atributo-valor. Los casos del problema están representados como un conjunto fijo de atributos, por ejemplo *cielo* y sus valores. El caso más sencillo es cuando cada atributo toma valores de un pequeño conjunto discreto y cada valor es disjunto, por ejemplo *{soleado, nublado, lluvia}*. Existen extensiones para trabajar con atributos de valores reales, por ejemplo, *temperatura* expresado numéricamente.
- La función objetivo tiene valores discretos. El árbol de decisión de la figura 4.8, asigna una clasificación binaria, por ejemplo *si* o *no* a cada caso. Un árbol de decisión puede ser extendido fácilmente, para representar funciones objetivo con más de dos valores posibles. Una extensión menos simple consiste en considerar funciones objetivo de valores discretos, por ello la aplicación del método en dominios discretos es menos común.
- Se necesitan descripciones disyuntivas. Como se mencionó, los árboles de decisión representan naturalmente conceptos disyuntivos.
- Ruido en los ejemplos de entrenamiento. El método es robusto al ruido en los ejemplos de entrenamiento, tanto errores de clasificación, como errores en los valores de los atributos.
- Valores faltantes en los ejemplos. El método puede usarse aún cuando algunos ejemplos de entrenamiento tengan valores desconocidos para algunos atributos. Al igual que en el punto anterior, esto se debe a que el algoritmo computa estadísticas globales que minimizan el impacto del ruido o falta de información de un ejemplo.

4.2.3 El algoritmo básico de inducción

Búsqueda descendente egoísta

La mayoría de los algoritmos para inferir árboles de decisión son variaciones de un algoritmo básico que emplea una **búsqueda descendente** (*top-down*) y **egoísta** (*greedy*) en el espacio de posibles árboles de decisión. En lo que sigue, nos basaremos en los algoritmos ID3 y C4.5.

Mejor atributo

El algoritmo básico ID3, construye el árbol de decisión de manera descendente, comenzando por preguntarse: ¿Qué atributo debería ser colocado en la raíz del árbol? Para responder esta pregunta, cada atributo es evaluado usando un test estadístico para determinar que tan bien clasifica él solo los ejemplos de entrenamiento. El **mejor atributo** es seleccionado y colocado en la raíz del árbol. Una rama y su nodo correspondiente es entonces creada para cada valor posible del atributo en cuestión. Los ejemplos de entrenamiento son repartidos en los nodos descendentes de acuerdo al valor que tengan para el atributo de la raíz. El proceso entonces se repite con los ejemplos ya distribuidos, para seleccionar un atributo que será colocado en cada uno de los nodos generados. Generalmente, el algoritmo se **detiene** si los ejemplos de entrenamiento comparten el mismo valor para el atributo que está siendo probado. Sin embargo, otros criterios para finalizar la búsqueda son posibles: i) Cobertura mínima, el número de ejemplos cubiertos por cada nodo está por abajo de cierto umbral; ii) Pruebas de significancia estadística usando χ^2 para probar si las distribuciones de las clases en los sub-árboles difiere significativamente. Aunque, como veremos, la poda del árbol se prefiere a las pruebas de significancia. El algoritmo que consideraremos, lleva a cabo una búsqueda egoísta de un árbol de decisión aceptable, sin reconsiderar nunca las elecciones pasadas (*backtracking*). Una versión simplificada de él se muestra en el Algoritmo 4.2.

Criterio de paro

¿Qué atributo es el mejor clasificador?

La decisión central de ID3 consiste en seleccionar qué atributo colocará en cada nodo del árbol de decisión. En el algoritmo presentado, esta opción la lleva a cabo la

Algoritmo 4.2 El algoritmo ID3.

```
1: function ID3(Ejs, Atbs, Clase)
2:     Arbol ← ∅; Default ← claseMayoria(Ejs);
3:     if Ejs = ∅ then
4:         return Default;
5:     else if mismoValor(Ejs, Clase) then
6:         return Arbol ← tomaValor(first(Ejs).Clase);
7:     else if Atbs = ∅ then
8:         return Arbol ← Default;
9:     else
10:        MejorPrtn ← mejorPrtn(Ejs, Atbs);
11:        MejorAtrb ← first(MejorPrtn);
12:        Arbol ← MejorAtrb;
13:        for all Prtn ∈ rest(MejorPrtn) do
14:            ValAtrb ← first(Prtn);
15:            SubEjs ← rest(Prtn);
16:            Rama ← ID3(SubEjs, {Atbs \ MejorAtrb}, Clase);
17:            agregaRama(Arbol, ValAtrb, Rama);
18:        end for
19:        return Arbol
20:    end if
21: end function
```

función `mejorPrtn`, que toma como argumentos un conjunto de ejemplos de entrenamiento y un conjunto de atributos, regresando la partición inducida por el atributo, que sólo, clasifica mejor los ejemplos de entrenamiento. Considere los ejemplos de entrenamiento del Cuadro 4.1 para el concepto objetivo: buen día para jugar tenis? El encabezado del cuadro indica los atributos usados para describir estos ejemplos, siendo *jugar-tenis?* el atributo clase.

día	cielo	temperatura	humedad	viento	jugar-tenis?
1	soleado	calor	alta	débil	no
2	soleado	calor	alta	fuerte	no
3	nublado	calor	alta	débil	si
4	lluvia	templado	alta	débil	si
5	lluvia	frío	normal	débil	si
6	lluvia	frío	normal	fuerte	no
7	nublado	frío	normal	fuerte	si
8	soleado	templado	alta	débil	no
9	soleado	frío	normal	débil	si
10	lluvia	templado	normal	débil	si
11	soleado	templado	normal	fuerte	si
12	nublado	templado	alta	fuerte	si
13	nublado	calor	normal	débil	si
14	lluvia	templado	alta	fuerte	no

Cuadro 4.1: *Conjunto de ejemplos de entrenamiento para la clase jugar-tenis? en ID3, adaptado de Quinlan [71].*

Si queremos particionar este conjunto de ejemplos con respecto al atributo *temperatura*, obtendríamos:

```
?- partition(temperatura, Ejemplos).
Ejemplos= [[temperatura [frio 5 6 7 9]
```

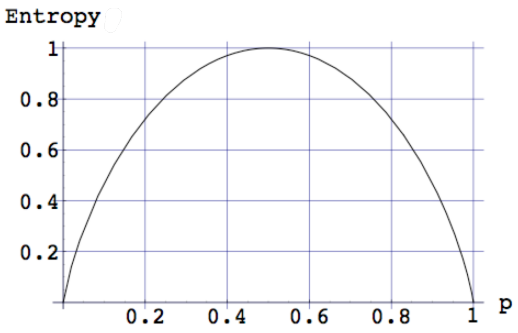

Figura 4.9: *Gráfica de la función entropía para clasificaciones booleanas.*

```
3          [caliente 1 2 3 13]
4          [templado 4 8 10 11 12 14]]
```

Lo que significa que el atributo *temperatura* tiene tres valores diferentes en el conjunto de entrenamiento: *frío*, *caliente*, y *templado*. Los casos 5,6,7 y 9 tienen como valor del atributo *temperatura= frío*. La función *mejorPrtn* encuentra el atributo que mejor separa los ejemplos de entrenamiento de acuerdo al atributo objetivo. En qué consiste una buena medida cuantitativa de la bondad de un atributo? Definiremos una propiedad estadística llamada ganancia de información.

Entropía y ganancia de información

Una manera de cuantificar la bondad de un atributo en este contexto, consiste en considerar la cantidad de información que proveerá este atributo, tal y como ésto es definido en teoría de información de Shannon y Weaver [83]. Un bit de información es suficiente para determinar el valor de un atributo booleano, por ejemplo, si/no, verdadero/falso, 1/0, etc., sobre el cual no sabemos nada. En general, si los posibles valores del atributo v_i, ocurren con probabilidades $P(v_i)$, entonces el contenido de información, o **entropía**, E de la respuesta actual está dado por: *Entropía*

$$E(P(v_i),\ldots,P(v_n)) = \sum_{i=1}^{n} -P(v_i)\ \log_2\ P(v_i)$$

Ejemplo 4.4. *Consideren nuevamente el caso booleano, aplicando esta ecuación a un volado con una moneda confiable, tenemos que la probabilidad de obtener águila o sol es de* $1/2$ *para cada una:*

$$E(\frac{1}{2},\frac{1}{2}) = -\frac{1}{2}\ \log_2\ \frac{1}{2} - \frac{1}{2}\ \log_2\ \frac{1}{2} = 1$$

Ejecutar el volado nos provee 1 bit de información, de hecho, nos provee la clasificación del experimento: si fue águila o sol. Si los volados los ejecutamos con una moneda cargada que da 99 % de las veces sol, entonces $E(1/100, 99/100) = 0{,}08$ *bits de información, menos que en el caso de la moneda justa, porque ahora tenemos más evidencia sobre el posible resultado del experimento. Si la probabilidad de que el volado de sol es del 100 %, entonces* $E(0,1) = 0$ *bits de información, ejecutar el volado no provee información alguna.*

La gráfica de la función de entropía se muestra en la figura 4.9.

Consideren ahora los ejemplos de entrenamiento del Cuadro 4.1. De 14 ejemplos, 9 son positivos (si es un buen día para jugar tenis) y 5 son negativos. La entropía de este conjunto de entrenamiento es:

$$E(\frac{9}{14},\frac{5}{14}) = 0{,}940$$

Si todos los ejemplos son positivos o negativos, por ejemplo, pertenecen todos a la misma clase, la entropía será 0. Una posible interpretación de ésto, es considerar

la entropía como una medida de ruido o desorden en los ejemplos. Definimos la **ganancia de información** como la reducción de la entropía causada por particionar un conjunto de entrenamiento S, con respecto a un atributo A:

Ganancia de información

$$Ganancia(S, A) = E(S) - \sum_{v \in A} \frac{|S_v|}{|S|} E(S_v)$$

Observen que el segundo término de *Ganancia*, es la entropía con respecto al atributo A. Al utilizar esta medida en ID3, sobre los ejemplos del cuadro 4.1, obtenemos:

```
Ganancia de informacion del atributo CIELO : 0.24674976
Ganancia de informacion del atributo TEMPERATURA : 0.029222548
Ganancia de informacion del atributo HUMEDAD : 0.15183544
Ganancia de informacion del atributo VIENTO : 0.048126936
Maxima ganancia de informacion: 0.24674976
Particion:
[cielo [soleado 1 2 8 9 11] [nublado 3 7 12 13]
       [lluvia 4 5 6 10 14]]
```

Esto indica que el atributo con mayor ganancia de información fue *cielo*, de ahí que esta parte del algoritmo genera la partición de los ejemplos de entrenamiento con respecto a este atributo. Si particionamos recursivamente los ejemplos que tienen el atributo *cielo* = *soleado*, obtendríamos:

```
Ganancia de informacion del atributo TEMPERATURA : 0.5709506
Ganancia de informacion del atributo HUMEDAD : 0.9709506
Ganancia de informacion del atributo VIENTO : 0.01997304
Maxima ganancia de informacion: 0.9709506
Particion:
[humedad [normal 11 9] [alta 8 2 1]]
```

Lo cual indica que en el nodo debajo de *soleado* deberíamos incluir el atributo *humedad*. Todos los ejemplos con *humedad* = *normal*, tienen valor *si* para el concepto objetivo. De la misma forma, todos los ejemplos con valor *humedad* = *alta*, tiene valor *no* para el concepto objetivo. Así que ambas ramas descendiendo de nodo *humedad*, llevarán a clases terminales de nuestro problema de aprendizaje. El algoritmo terminará por construir el árbol de la figura 4.8.

4.2.4 Espacio de hipótesis

Como los otros métodos de aprendizaje, ID3 puede concebirse como un proceso de búsqueda en un espacio de hipótesis, para encontrar aquella hipótesis que se ajusta mejor a los datos de entrenamiento. El espacio de hipótesis explorado por ID3 es el espacio de todos los árboles de decisión posibles. El algoritmo lleva a cabo una búsqueda de lo simple a lo complejo, comenzando por el árbol vacío, para considerar cada vez hipótesis más complejas. La medida *ganancia de información* guía esta búsqueda de ascenso de colina (*hill-climbing*), como ejemplificamos en la sección anterior.

Considerando ID3 en términos de su espacio y estrategias de búsqueda, es posible analizar sus **capacidades y limitaciones**:

Capacidades y limitaciones

- El espacio de hipótesis de ID3 es **completo** con respecto a las funciones de valores discretos que pueden definirse a partir de los atributos considerados. De manera que no existe el riesgo que la función objetivo no se encuentre en el espacio de hipótesis.
- ID3 mantiene sólo **una hipótesis** mientras explora el espacio de hipótesis posibles. Esto contrasta, por ejemplo, con el algoritmo eliminación de candidatos,

que mantiene el conjunto de todas las hipótesis consistentes con el conjunto de entrenamiento. Es por ello que ID3 es incapaz de determinar cuantos árboles de decisión diferentes son consistentes con los datos.

- El algoritmo básico ID3 no ejecuta vuelta atrás (*backtracking*) en su búsqueda. Una vez que el algoritmo selecciona un atributo, nunca reconsiderará esta elección. Por lo tanto, es susceptible a los mismos riesgos que los algoritmos estilo ascenso de colina, por ejemplo, caer **máximos o mínimos locales**. Como veremos, la vuelta atrás puede implementarse con alguna técnica de poda.
- ID3 utiliza todos los ejemplos de entrenamiento en cada paso de su búsqueda guiada por el estadístico *Ganancia*. Esto contrasta con los métodos que usan los ejemplos incrementalmente, por ejemplo, eliminación de candidatos. Una ventaja de usar propiedades estadísticas de todos los ejemplos es que la búsqueda es menos sensible al **ruido** en los datos.

4.2.5 Sesgo inductivo

Recuerden que el sesgo inductivo es el conjunto de afirmaciones que, junto con los datos de entrenamiento, justifican deductivamente la clasificación realizada por un sistema de aprendizaje inductivo sobre casos futuros. Dado un conjunto de entrenamiento, por lo general hay muchos árboles de decisión consistentes con éste. Describir el sesgo inductivo de ID3 equivale a explicar porqué este algoritmo prefiere ciertos árboles a otros, qué árbol elegirá.

Puesto que ID3 encontrará el primer árbol consistente con el conjunto de entrenamiento, producto de una búsqueda de ascenso de colina, de lo simple a lo complejo, el algoritmo tiene preferencia por: i) árboles pequeños sobre árboles grandes, que indican que la búsqueda termino en proximidad a la raíz del árbol; y ii) debido a su carácter egoísta, árboles que colocan atributos más informativos cerca de la raíz del árbol. Sin embargo, observen que este sesgo es aproximado. Un algoritmo que tuviera un sesgo idéntico al descrito aquí, tendría que realizar una búsqueda primero en amplitud y preferir los árboles de menor profundidad. ID3 busca primero en profundidad.

Sesgo por restricción y sesgo por preferencia

Existe una diferencia interesante entre los sesgos que exhiben ID3 y el algoritmo eliminación de candidatos. El sesgo de ID3 es producto de su estrategia de búsqueda, mientras que el sesgo de eliminación de candidatos es resultado de la definición del espacio de búsqueda. Por lo tanto, el sesgo de ID3 exhibe una preferencia por ciertas hipótesis, sobre otras, por ejemplo, hipótesis compactas. Este tipo de sesgo, que no impone restricciones sobre las hipótesis que serán eventualmente consideradas, recibe el nombre de **sesgo por preferencia**. Por otra parte, el sesgo de eliminación de candidatos que restringe el conjunto de hipótesis a considerar, recibe el nombre de **sesgo por restricción**. En general, es preferible trabajar con un sesgo por preferencia, puesto que éste permite al sistema de aprendizaje explorar un espacio de hipótesis completo, asegurando que la representación del concepto objetivo se encuentra ahí.

Sesgo por preferencia

¿Porqué preferir hipótesis más compactas?

¿Es el sesgo inductivo de ID3, preferir las hipótesis más compactas, lo suficientemente robusto para generalizar más allá de los datos observados? Este es un debate no resuelto iniciado por William de Occam[1] *circa* 1320. Un argumento intuitivo es

1 El enunciado exacto de la navaja de Occam es: *Non sunt multiplicanda entia prater necessitatem* (las entidades no deben multiplicarse más allá de lo necesario).

que existen mucho menos hipótesis compactas que extensas, por lo que es más difícil que una hipótesis compacta coincida accidentalmente con los datos observados. En cambio, hay muchas hipótesis extensas que se pueden, ajustar a los datos de entrenamiento, pero fallarán al generalizar.

4.2.6 Consideraciones

Algunas consideraciones sobre la aplicación práctica del algoritmo básico de ID3 presentado aquí, incluyen: Mecanismos para determinar que tanto debe crecer el árbol en profundidad; para procesar atributos con valores continuos; para procesar ejemplos de entrenamiento con valores faltantes; para introducir costos diferentes asociados a los atributos; así como para determinar una buena métrica de selección de los atributos y mejorar la eficiencia computacional del algoritmo. Cabe mencionar que, muchos de estos aspectos han sido incorporados en el sistema C4.5 [73].

Evitando un sobre ajuste con los datos de entrenamiento

El algoritmo básico de ID3 expande cada rama del árbol en profundidad hasta que logra clasificar perfectamente los ejemplos de entrenamiento. Esta estrategia es razonable, pero puede introducir dificultades si los datos de entrenamiento presentan ruido, o cuando el conjunto de entrenamiento es demasiado pequeño, como para ofrecer un muestreo significativo del concepto objetivo. En estos casos, ID3 puede producir árboles que se sobre ajustan a los datos de entrenamiento. Formalmente definimos el **sobre ajuste** como: *Sobre ajuste*

Definición 4.1. *Dado un espacio de hipótesis H, se dice que una hipótesis $h \in H$ está sobre ajustada a los ejemplos de entrenamiento, si existe una hipótesis alternativa $h' \in H$, tal que h' tiene un error de clasificación más pequeño que h sobre la distribución completa de los casos del problema.*

Es común observar que a medida que el **tamaño del árbol** crece, en término del número de nodos usado[2], su precisión sobre el conjunto de entrenamiento mejora monotónicamente, pero, sobre el conjunto de prueba primero crece y luego decae. *Tamaño del árbol*

¿Cómo es esto posible, que un árbol h que tiene mayor precisión que h' sobre el conjunto de entrenamiento, luego tenga un desempeño menor sobre el conjunto de prueba? Una situación en la que esto ocurre, se da cuando el conjunto de entrenamiento contiene **ruido**, por ejemplo, elementos mal clasificados. Consideren agregar el siguiente caso mal clasificado (clasificado como *jugar-tenis? = no*) al conjunto de entrenamiento del Cuadro 4.1: *Ruido*

$$\langle\, cielo = soleado, temperatura = alta, humedad = normal, viento = fuerte \,\rangle$$

Al ejecutar ID3 sobre el nuevo conjunto de entrenamiento, éste construirá un árbol más complejo. En particular, el ejemplo con ruido será filtrado junto con los ejemplos 9 y 11 (*cielo = soleado* y *humedad = normal*), que son ejemplos positivos. Dado que el nuevo ejemplo es negativo, ID3 buscará refinar el árbol a partir del nodo *humedad*, agregando un atributo más al árbol. Este nuevo árbol h' tiene mayor precisión sobre los ejemplos de entrenamiento que h, puesto que se ha ajustado al ejemplo con ruido. Pero h tendrá mejor desempeño al clasificar nuevos casos, tomados de una misma distribución que los ejemplos de entrenamiento.

Existe la posibilidad de sobreajuste, aún cuando el conjunto de entrenamiento esté libre de ruido, por ejemplo, si el conjunto de entrenamiento tiene **pocos elementos**. En conjuntos de entrenamiento pequeños, es fácil encontrar regularidades accidentales en donde un atributo puede particionar muy bien los ejemplos dados, aunque no esté relacionado con el concepto objetivo. *Pocos ejemplos*

2 Observen que esto refleja el número de atributos usado en la hipótesis, esto es, árboles más grandes imponen más restricciones.

Puesto que el sobreajuste puede reducir la precisión de un árbol inducido por ID3 entre un 10 a 25 %, diferentes enfoques han sido propuestos para evitar este fenómeno. Los **enfoques** pueden agruparse en dos clases: *Soluciones*

- Enfoques que detienen el crecimiento del árbol anticipadamente, antes de que alcance un punto donde clasifique perfectamente los ejemplos de entrenamiento.
- Enfoques en donde se deja crecer el árbol para después podarlo.

Aunque el primer enfoque parezca más directo, la poda posterior del árbol ha demostrado tener más éxito en la práctica. Esto se debe a la dificultad de estimar en que momento debe detenerse el crecimiento del árbol. Independientemente del enfoque usado, una pregunta interesante es: ¿Cual es el **tamaño correcto** de un árbol? Algunos enfoques para responder a esta pregunta incluyen: *Tamaño correcto*

- Usar un conjunto de ejemplos, diferentes de los usados en el entrenamiento, para evaluar la utilidad de eliminar nodos del árbol. Por ejemplo, poda.
- Usar los ejemplos disponibles para el entrenamiento, pero aplicando una prueba para estimar cuando expandir el árbol (o detener su crecimiento), podría producir una mejora al clasificar nuevos casos. Por ejemplo, usar el test χ^2.
- Usar explícitamente una medida de complejidad para codificar los ejemplos de entrenamiento y el árbol de decisión, deteniendo el crecimiento del árbol cuando el tamaño de la codificación sea minimizado. Por ejemplo, el principio de descripción de longitud mínima (*MDL*).

Reduciendo el error por poda

¿Como podemos usar un conjunto de ejemplos de validación para prevenir el sobre ajuste? Un enfoque llamado *reduced-error pruning* [71], consiste en considerar cada nodo del árbol como candidato a ser podado. La poda consiste en eliminar todo el subárbol que tiene como raíz el nodo en cuestión, convirtiéndolo así en una hoja, cuya clase corresponde a valor más común de los casos asociados a ese nodo.

Un nodo solo es eliminado si el árbol podado que resulta de ello, no presenta un desempeño peor que el árbol original sobre el conjunto de validación. El efecto de esto, es que los nodos que se han colocado en el árbol por **coincidencias fortuitas** en los datos del entrenamiento, generalmente son eliminados debido a que las coincidencias suelen no estar presentes en el conjunto de validación. *Coincidencias*

Este método es únicamente efectivo si contamos con **suficientes ejemplos**, de tal forma que el conjunto de entrenamiento y el conjunto de validación sean significativos estadísticamente. De otra forma, tomar ejemplos para el conjunto de validación reduce aún más el tamaño del conjunto de entrenamiento, aumentando así la posibilidad de sobre ajuste.

Poda de reglas

En la práctica, un método exitoso para encontrar el árbol de mayor precisión se conoce como *rule post-prunning* [73] y está incorporado en el sistema C4.5. El procedimiento es el siguiente:

1. Inducir el árbol de decisión permitiendo sobre ajuste, por ejemplo, con nuestro algoritmo básico ID3.
2. Convertir el árbol aprendido en un conjunto de reglas equivalente, esto es, una conjunción por cada rama del árbol que va de la raíz a una hoja.
3. Podar (generalizar) cada regla, eliminando cualquier condición que resulte en una mejora de la precisión estimada.

4. Ordenar las reglas por su precisión estimada, y aplicarlas en ese orden al clasificar nuevos casos.

Cabe mencionar que el método aplicado por C4.5 no es estadísticamente valido, aunque ha demostrado ser una heurística útil.

Incorporando valores continuos

En el algoritmo básico de ID3 tanto la clase, como los atributos usados para describir los casos, deben tener valores discretos. La segunda restricción puede ser eliminada fácilmente, permitiendo el uso de atributos con valores continuos. Esto se logra definiendo dinámicamente **nuevos atributos** discretos a partir de los atributos continuos originales. Para un atributo continuo A, el algoritmo puede crear dinámicamente un atributo discreto A_c que es verdadero si $A > c$ y falso en cualquier otro caso. La única consideración es cómo seleccionar el mejor valor para el umbral c. Supongan que el atributo *temperatura* toma valores discretos y que su relación con la clase es la siguiente:

Nuevos atributos discretizados

temperatura	40	48	60	72	80	90
jugar-tenis?	No	No	Si	Si	Si	No

¿Qué atributo booleano basado en un umbral debemos definir para el atributo temperatura? Obviamente, necesitamos un umbral c, tal que éste produzca la mayor ganancia de información posible. Es posible generar candidatos a umbral, ordenando los ejemplos de acuerdo a su valor en el atributo *temperatura* e identificando ejemplos adyacentes que difieren en el valor de su atributo objetivo. Se puede demostrar que los umbrales c que maximiza la ganancia de información, se encuentran en estos sitios. Para el ejemplo presentado, dos umbrales pueden localizarse en los puntos $(48 + 60)/2$ y $(80 + 90/2)$. La ganancia de información puede entonces calcularse para los atributos $temperatura_{>54}$ y $temperatura_{>85}$. El atributo con mayor ganancia de información, en este caso el primero, puede ser usado entonces para competir con otros atributos en la construcción del árbol de decisión. Por supuesto, es posible también mantener ambos atributos dinámicamente creados, usando múltiples intervalos.

Medidas alternativas para la selección de atributos

Existe un sesgo natural en la medida de ganancia de información, que favorece a aquellos atributos con muchos valores. Por ejemplo, un atributo *fecha*, tendría mayor ganancia de información que cualquiera de los atributos en nuestro ejemplo. Esto se debe a que este atributo predice perfectamente el valor del atributo objetivo. El problema es que este atributo tiene tantos valores distintos que tiende a separar perfectamente los ejemplos de entrenamiento en pequeños subconjuntos, que se ajustan al concepto buscado. Por esto, el atributo *fecha* tiene una ganancia de información elevada, a pesar de ser un predictor pobre.

Una solución a este problema es usar una métrica alternativa a la ganancia de información. Quinlan [71], propone una medida alternativa que ha sido usada con éxito, *gain ratio*. Esta métrica penaliza atributos como *fecha* incorporando un término conocido como *split information*, que es sensible a qué tan amplia y uniforme es la partición que un atributo induce en los datos:

Split information

$$splitInformation(S, A) = -\sum_{i=1}^{c} \frac{|S_i|}{|S|} \, log_2 \frac{|S_i|}{|S|}$$

Observen que este término es la entropía de S con respecto al atributo A. La medida *gain radio* está definida como:

Gain ratio

$$gainRatio(S, A) = \frac{gain(S, A)}{splitInformatio(S, A)}$$

Problemas

Un problema práctico con este enfoque es que el **denominador** de esta medida puede ser cero o muy pequeño, si $|S_i| \approx |S|$, lo cual hace que la medida sea indefinida para atributos que tienen el mismo valor para todos los ejemplos.

4.2.7 Implementación

En esta sección implementaremos una versión de ID3 en SWI-Prolog. El código será comentado y ejemplificado a lo largo de la presentación.

Interfaz principal

La llamada principal al algoritmo es *id*3, si queremos que el umbral para el mínimo de casos por hoja sea uno, su valor por defecto. El umbral indica cual es el número mínimo de ejemplos en una hoja, para volver a aplicar ID3 de manera recursiva. Si el número de ejemplos asociados a la hoja es menor que el umbral, el proceso se detiene y la clase asignada será la clase mayoritaria. De forma que *id*3, en realidad llama a *id*3/1 cuyo argumento es un valor para el umbral. La implementación de ambos predicados es como sigue:

```
:- dynamic
     ejemplo/3,
     nodo/3.

id3 :- id3(1).        % Umbral = 1, por default.

id3(Umbral) :-
  reset,
  write('Con que archivo CSV desea trabajar: '),
  read(ArchCSV),
  cargaEjs(ArchCSV,Atrs),
  findall(N,ejemplo(N,_,_),Inds), % Obtiene indices Inds de los ejemplos
  write('Tiempo de inducción:'),
  time(inducir(Inds,raiz,Atrs,Umbral)),
  imprimeArbol, !.
```

Con la llamada a *id*3/1 se eliminan los nodos y ejemplos que hayan existido en el espacio de trabajo de Prolog, vía *reset*. Por ello, *ejemplo*/3 y *nodo*/3 son declarados como dinámicos al principio de nuestro programa. Posteriormente se le pide al usuario el nombre del archivo CSV con el que desea trabajar. La respuesta debe estar entrecomillada y terminar con un punto, como es común al usar *read*/1. El predicado *cargaEjs*/3 lee un archivo separado por comas y agrega al espacio de trabajo los ejemplos en el incluidos. También obtiene la lista de atributos. En realidad, nuestra implementación manipulará el conjunto de índices de los ejemplos de entrenamiento para computar las particiones típicas de ID3. Por ello, se crea una lista de índices vía *findall*/3. Posteriormente se lleva a cabo la inducción del árbol de decisión con *inducir*/4 y se imprime el árbol obtenido.

Ejemplos de entrenamiento

Los ejemplos de entrenamiento se definen mediante el predicado *ejemplo*/3 cuyos argumentos son su índice, su valor para la clase, y la lista de sus pares atributo-valor. Para el ejemplo de jugar tenis, al archivo de entrenamiento `tenis.pl` incluye las siguientes líneas:

```
ejemplo(1,no,[cielo=soleado,temperatura=alta,humedad=alta,viento=debil]).
ejemplo(2,no,[cielo=soleado,temperatura=alta,humedad=alta,viento=fuerte]).
ejemplo(3,si,[cielo=nublado,temperatura=alta,humedad=alta,viento=debil]).
ejemplo(4,si,[cielo=lluvioso,temperatura=templada,humedad=alta,viento=debil]).
ejemplo(5,si,[cielo=lluvioso,temperatura=fresca,humedad=normal,viento=debil]).
```

```
ejemplo(6,no,[cielo=lluvioso,temperatura=fresca,humedad=normal,viento=fuerte]).
ejemplo(7,si,[cielo=nublado,temperatura=fresca,humedad=normal,viento=fuerte]).
ejemplo(8,no,[cielo=soleado,temperatura=templada,humedad=alta,viento=fuerte]).
ejemplo(9,si,[cielo=soleado,temperatura=fresca,humedad=normal,viento=debil]).
ejemplo(10,si,[cielo=lluvioso,temperatura=templada,humedad=normal,viento=debil]).
ejemplo(11,si,[cielo=soleado,temperatura=nublado,humedad=normal,viento=fuerte]).
ejemplo(12,si,[cielo=nublado,temperatura=templado,humedad=alta,viento=fuerte]).
ejemplo(13,si,[cielo=nublado,temperatura=alta,humedad=normal,viento=debil]).
ejemplo(14,no,[cielo=lluvioso,temperatura=templada,humedad=alta,viento=fuerte]).
```

en ese caso, basta con cargar el archivo en Prolog para poder llevar a cabo la inducción de un nuevo árbol de decisión.

Normalmente, los conjuntos de entrenamiento no están en el formato presentado. Estos suelen almacenarse como archivos de valores separados por coma (CSV), de ahí que hayamos implementado *cargaEjs*/3 para obtener la representación deseada a partir de un archivo CSV. Nuestro conjunto de entrenamiento `tenis.csv` luce así:

```
cielo, temperatura, humedad, viento, jugarTenis
soleado, alta, alta, debil, no
soleado, alta,  alta,  fuerte, no
nublado, alta,  alta, debil, si
lluvioso, templada,  alta, debil, si
lluvioso, fresca, normal, debil, si
lluvioso, fresca, normal, fuerte, no
nublado, fresca, normal, fuerte, si
soleado, templada, alta, debil, no
soleado, fresca, normal, debil, si
lluvioso, templada, normal, debil, si
soleado, templada, normal, fuerte, si
nublado, templada, alta, fuerte, si
nublado, alta, normal, debil, si
lluvioso, templada, alta, fuerte, no
```

Para leer los archivos CSV utilizaremos las utilidades introducidas en la sección 3.5.5. Para recordar, su uso sería como sigue:

```
?- leerCSV("tenis.csv",Atrs,Ejs), dominios(Atrs,Ejs,Doms).
14 ejemplos de entrenamiento cargados.
Atrs = [cielo, temperatura, humedad, viento, jugarTenis], Ejs =
[[soleado, alta, alta, debil, no], [soleado, alta, alta, fuerte,
no], [nublado, alta, alta, debil, si], ...],
Doms = [[cielo, [soleado, nublado, lluvioso]], [temperatura, [alta,
templada, fresca]], [humedad, [alta, normal]], [viento, [debil,
fuerte]], [jugarTenis, [no, si]]].
```

donde hemos obtenido los atributos *Atrs*, los ejemplos *Ejs* como una lista de valores y el dominio de cada atributo *Doms*. Por supuesto que, si queremos agregar cláusulas *ejemplo*/3 debemos aún procesar estas salidas.

Primero definiremos dos predicados auxiliares. *last*/2 recibe como primer argumento una lista, y regresa en el segundo argumento el último elemento de la lista; *butlast*/2 regresa la lista recibida en su primer argumento, sin el último elemento:

```
%%% last(L,E): E es el último elemento de la lista L.

last([],[]).
last(L,E) :-
  append(_,[E],L).

  %%% butLast(L1,L2): L2 es L1 sin el último elemento.

butlast([],[]).
```

```
butlast(L1,L2) :-
  last(L1,Last),
  append(L2,[Last],L1).
```

La idea es procesar la información de *leerCSV*/3, para obtener una salida como la siguiente:

```
?- csv2ejs("tenis.csv",Ejs,Atrs).
14 ejemplos de entrenamiento cargados.
Ejs = [ejemplo(1, no, [cielo=soleado, temperatura=alta, humedad=alta,
viento=debil]), ejemplo(2, no, [cielo=soleado, temperatura=alta,
humedad=alta, viento=fuerte]), ejemplo(3, si, [cielo=nublado,
temperatura=alta, humedad=alta, viento=debil]), ...],
Atrs = [cielo, temperatura, humedad, viento, jugarTenis]
```

La definición de *csv2ejs*/3 es la siguiente:

```
% csv2ejs(ArchCSV,Ejs,Atrs): transforma las lista de salida de
% leerCSV/2 en una lista de ejemplos(Ind,Clase,[Atr=Val]) y otra
% lista de atributos.

csv2ejs(ArchCSV,Ejs,Atrs) :-
  leerCSV(ArchCSV,Atrs,EjsCSV),
  butlast(Atrs,AtrsSinUltimo),
  procEjs(1,AtrsSinUltimo,EjsCSV,Ejs).

procEjs(_,_,[],[]).

procEjs(Ind,AtrsSinUltimo,[Ej|Ejs],[ejemplo(Ind,Clase,EjAtrsVals)|Resto]) :-
  last(Ej,Clase),
  butlast(Ej,EjSinUltimo),
  maplist(procAtrVal,AtrsSinUltimo,EjSinUltimo,EjAtrsVals),
  IndAux is Ind + 1,
  procEjs(IndAux,AtrsSinUltimo,Ejs,Resto).

procAtrVal(Atr,Val,Atr=Val).
```

El predicado *csv2ejs*/3 lee el archivo CSV, con la ayuda del predicado *leerCSV*/3, previamente definido. Luego elimina el último valor de la lista de atributos *Atrs*, que es el atributo clase y procede a procesar cada ejemplo con el predicado *ProcEjs*/4. La salida de este último es una lista de ejemplos *Ejs* con el formato deseado, listos para ser agregados a la memoria de trabajo de Prolog.

Por cada ejemplo *Ej* que *procEjs*/3 procesa, se obtiene la clase del ejemplo (con la ayuda de *last*/2) y se elimina ésta del ejemplo mismo, que estaba representado como una lista e valores (con la ayuda de *butlast*/2). Luego se procesan los atributos y los valores para obtener pares de ellos, p. ej., *cielo* = *soleado*. El *ejemplo*/3 así formado se agrega al frente del resto de los ejemplos por procesar recursivamente.

Lo único que resta es un predicado que cargue la lista de ejemplos en memoria:

```
% cargaEjs(ArchCSV): carga ArchCSV como hechos estilo
% ejemplo(Ind,Clase,[Atr=Val]).

cargaEjs(ArchCSV,Atrs) :-
  csv2ejs(ArchCSV,Ejs,Atrs),
  maplist(assertz,Ejs).
```

que es el predicado usado para cargar el archivo CSV desde la interfaz *id*3/1. Es posible verificar que los ejemplos han sido cargados en memoria, ejecutando:

```
?- listing(ejemplo).
:- dynamic ejemplo/3.
```

```
ejemplo(1, no, [cielo=soleado, temperatura=alta, humedad=alta,
viento=debil]).
ejemplo(2, no, [cielo=soleado, temperatura=alta, humedad=alta,
viento=fuerte]).
...
```

lo que nos indica que nuestros 14 ejemplos han sido cargados correctamente.

El árbol de decisión

El árbol de decisión se representará como un conjunto de *nodo*/3 que se irán agregando a memoria al inducir el árbol. El caso de jugar tenis, produce los siguientes nodos:

```
?- listing(nodo).
:- dynamic nodo/3.

nodo(1, cielo=lluvioso, raiz).
nodo(2, viento=fuerte, 1).
nodo(hoja, [no/2], 2).
nodo(3, viento=debil, 1).
nodo(hoja, [si/3], 3).
nodo(4, cielo=nublado, raiz).
nodo(hoja, [si/4], 4).
nodo(5, cielo=soleado, raiz).
nodo(6, humedad=normal, 5).
nodo(hoja, [si/2], 6).
nodo(7, humedad=alta, 5).
nodo(hoja, [no/3], 7).
```

Para poder reportar el árbol inducido en un formato legible, haremos uso del predicado *imprimeArbol*, que a su vez hace uso de *imprimeLista*, cuyas definiciones son como sigue:

```
imprimeArbol :-
  imprimeArbol(raiz,0).

  imprimeArbol(Padre,_) :-
  nodo(hoja,Clase,Padre), !,
  write(' => '),write(Clase).
imprimeArbol(Padre,Pos) :-
  findall(Hijo,nodo(Hijo,_,Padre),Hijos),
  Pos1 is Pos+2,
  imprimeLista(Hijos,Pos1).

imprimeLista([],_) :- !.
imprimeLista([N|T],Pos) :-
  nodo(N,Test,_),
  nl, tab(Pos), write(Test),
  imprimeArbol(N,Pos),
  imprimeLista(T,Pos).
```

cuya salida es:

```
cielo=lluvioso
  viento=fuerte => [no/2]
  viento=debil => [si/3]
cielo=nublado => [si/4]
cielo=soleado
  humedad=normal => [si/2]
  humedad=alta => [no/3]
```

observen que junto con la clase, *imprimeArbol* reporta también cuantos ejemplos han sido colocados en la hoja en cuestión.

Inducción

En la inducción del árbol de decisión hay varios casos que cubrir. Comencemos por los casos terminales: Hay dos casos terminales a considerar, el primero es cuando el número de ejemplos disponibles es menor que el *Umbral* en ese caso se guarda un nodo *hoja* con la distribución de la clase para los ejemplos, p. ej., para el caso de jugar tenis cuantos ejemplos tienen clase *si* y cuantos *no*:

```
% Caso 1. El número de ejemplos a clasificar es menor que el umbral.
% Se crea un nodo hoja con las distribución Distr, apuntando al padre
% del nodo.

inducir(Ejs,Padre,_,Umbral) :-
  length(Ejs,NumEjs),
  NumEjs=<Umbral,
  distr(Ejs, Distr),
  assertz(nodo(hoja,Distr,Padre)), !. % Se agrega al final de los nodos.
```

El otro caso terminal es cuando todos los ejemplos pertenecen a la misma clase. En ese caso la distribución de la clase para los ejemplos tomara la forma $[Clase/NumEjs]$:

```
% Caso 2. Todos los ejemplos a clasificar son de la misma clase.
% Se crea un nodo hoja con la distribución [Clase], apuntando al padre
% del nodo.

inducir(Ejs,Padre,_,_) :-
  distr(Ejs, [Clase]),
  assertz(nodo(hoja,[Clase],Padre)).
```

Si no estamos en un caso terminal, es necesario elegir el mejor atributo y particionar los ejemplos de acuerdo a los valores para el atributo seleccionado:

```
% Caso 3. Se debe decidir que atributo es el mejor clasificador para
% los ejemplos dados.

inducir(Ejs,Padre,Atrs,Umbral) :-
  eligeAtr(Ejs,Atrs,Atr,Vals,Resto), !,
  particion(Vals,Atr,Ejs,Padre,Resto,Umbral).
```

Si esto no es posible, es que los datos son inconsistentes:

```
% Caso 4. Los datos son inconsistentes, no se pueden particionar.

inducir(Ejs,Padre,_,_) :- !,
  nodo(Padre,Test,_),
  write('Datos inconsistentes: no es posible construir partición de '),
  write(Ejs), write(' en el nodo '), writeln(Test).
```

Ya podrán adivinar que es *particion*/6 quien realmente lleva a cabo la construcción del árbol.

Distribución de clases

¿Cual es la distribución inicial de la clase para los ejemplos de jugar tenis? Esto lo podemos consultar con:

```
?- findall(E,ejemplo(E,_,_),Ejs), distr(Ejs,Dist).
Ejs = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
Dist = [no/5, si/9].
```

lo que indica que tenemos 5 ejemplos de la clase *no* y 9 de la clase *si*. Los ejemplos 1,2,6 y 8 son todos miembros de la clase *no*, por lo que:

```
?- distr([1,2,6,8],Dist).
Dist = [no/4].
```

La implementación de *distr*/2 es la siguiente:

```
%%% distr(+Ejs,-DistrClaseEjs)
%%% Computa la Distributción de clases para el conjunto de ejemplos S.
%%% La notación X^Meta causa que X no sea instanciada al solucionar la Meta.

distr(Ejs,DistClaseEjs) :-
  % Extrae Valores de clase Cs de los ejemplos S
  setof(Clase,Ej^AVs^(member(Ej,Ejs),ejemplo(Ej,Clase,AVs)),Clases),
  % Cuenta la distribución de los valores para la Clase
  cuentaClases(Clases,Ejs,DistClaseEjs).

cuentaClases([],_,[]) :- !.
cuentaClases([Clase|Clases],Ejs,[Clase/NumEjsEnClase|RestoCuentas]) :-
  % Extrae los ejemplos con clase Clase en la lista Cuentas
  findall(Ej,(member(Ej,Ejs),ejemplo(Ej,Clase,_)),EjsEnClase),
  % Computa cuantos ejemplos hay en la Clase
  length(EjsEnClase,NumEjsEnClase), !,
  % Cuentas para el resto de los valores de la clase
  cuentaClases(Clases,Ejs,RestoCuentas).
```

El mejor atributo

El mejor atributo es el que maximiza la ganancia de información *Gain* con respecto a los ejemplos *Ejs* y atributos *Atrs*. El predicado *eligeAtr*/4 computa el mejor atributo y lo elimina de la lista de atributos disponibles para construir el árbol:

```
% eligeAtr(+Ejs,+Atrs,-Atr,-Vals,-Resto)
% A partir de un conjunto de ejemplos Ejs y atributos Atrs, computa
% el atributo Atr en Atrs con mayor ganancia de información, sus Vals
% y el Resto de atributos en Atrs.

eligeAtr(Ejs,Atrs,Atr,Vals,RestoAtrs) :-
  length(Ejs,NumEjs),
  contenidoInformacion(Ejs,NumEjs,I), !,
  findall((Atr-Vals)/Gain,
          (member(Atr,Atrs),
           vals(Ejs,Atr,[],Vals),
           separaEnSubConjs(Vals,Ejs,Atr,Parts),
           informacionResidual(Parts,NumEjs,IR),
           Gain is I - IR),
          Todos),
  maximo(Todos,(Atr-Vals)/_),
  eliminar(Atr,Atrs,RestoAtrs), !.

separaEnSubConjs([],_,_,[]) :- !.
separaEnSubConjs([Val|Vals],Ejs,Atr,[Part|Parts]) :-
  subconj(Ejs,Atr=Val,Part), !,
  separaEnSubConjs(Vals,Ejs,Atr,Parts).

informacionResidual([],_,0) :- !.
informacionResidual([Part|Parts],NumEjs,IR) :-
  length(Part,NumEjsPart),
  contenidoInformacion(Part,NumEjsPart,I), !,
  informacionResidual(Parts,NumEjs,R),
  IR is R + I * NumEjsPart/NumEjs.
```

```
30
31 contenidoInformacion(Ejs,NumEjs,I) :-
32   setof(Clase,Ej^AVs^(member(Ej,Ejs),ejemplo(Ej,Clase,AVs)),Clases), !,
33   sumaTerms(Clases,Ejs,NumEjs,I).
34
35 sumaTerms([],_,_,0) :- !.
36 sumaTerms([Clase|Clases],Ejs,NumEjs,Info) :-
37   findall(Ej,(member(Ej,Ejs),ejemplo(Ej,Clase,_)),EjsEnClase),
38   length(EjsEnClase,NumEjsEnClase),
39   sumaTerms(Clases,Ejs,NumEjs,I),
40   Info is I - (NumEjsEnClase/NumEjs)*(log(NumEjsEnClase/NumEjs)/log(2)).
41
42 vals([],_,Vals,Vals) :- !.
43 vals([Ej|Ejs],Atr,Vs,Vals) :-
44   ejemplo(Ej,_,AVs),
45   member(Atr=V,AVs), !,
46   (member(V,Vs), !, vals(Ejs,Atr,Vs,Vals);
47    vals(Ejs,Atr,[V|Vs],Vals)
48   ).
49
50 subconj([],_,[]) :- !.
51 subconj([Ej|Ejs],Atr,[Ej|RestoEjs]) :-
52   ejemplo(Ej,_,AVs),
53   member(Atr,AVs), !,
54   subconj(Ejs,Atr,RestoEjs).
55 subconj([_|Ejs],Atr,RestoEjs) :-
56   subconj(Ejs,Atr,RestoEjs).
```

Por ejemplo, la siguiente meta computa el mejor atributo (*cielo*), dados los ejemplos E y atributos conocidos:

```
?- findall(N,ejemplo(N,_,_),E),
   eligeAtr(E,[cielo,temperatura,humedad,viento],A,V,R).
E = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
A = cielo,
V = [lluvioso, nublado, soleado],
R = [temperatura, humedad, viento].
```

Como vimos en la primera parte de esta sección, para computar la ganancia de información necesitamos computar el contenido informacional de todos los ejemplos:

```
?- findall(N,ejemplo(N,_,_),E), length(E,L), contenidoInformacion(E,L,I).
E = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
L = 14,
I = 0.9402859586706309.
```

Y por cada atributo, computar la información residual para restársela al contenido informacional, y así obtener las ganancias de información:

```
  ?- findall(N,ejemplo(N,_,_),E),
     findall((A-Valores)/Gain,
            (member(A,[cielo,temperatura,humedad,viento]),
            vals(E,A,[],Valores),
            separaEnSubConjs(Valores,E,A,Ess),
            informacionResidual(Ess,14,R),
            Gain is 0.940286 - R),
            All).
  E = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
  All = [(cielo-[lluvioso, nublado, soleado])/0.24674986110380803,
         (temperatura-[fresca, templada, alta])/0.029222606988323685,
         (humedad-[normal, alta])/0.1518355426917104,
         (viento-[fuerte, debil])/0.048127071737638305].
```

Solo resta obtener el atributo de *All* con la máxima ganancia de información y eliminarlo de la lista de atributos disponibles regresada en *RestoAtrs*. La inducción del árbol de decisión es un proceso recursivo. Los valores del atributo elegido inducen una partición sobre los ejemplos. Esto se logra con el predicado *particion*/6.

```
% particion(+Vals,+Atr,+Ejs,+Padre,+Resto,+Umbral)
% Por acada Valor del atributo Atr en Vals, induce una partición
% en los ejemplos Ejs de acuerdo a Valor, para crear un nodo del
% árbol y llamar recursivamente a inducir.

particion([],_,_,_,_,_) :- !.
particion([Val|Vals],Atr,Ejs,Padre,RestoAtrs,Umbral) :-
  subconj(Ejs,Atr=Val,SubEjs), !,
  generaNodo(Nodo),
  assertz(nodo(Nodo,Atr=Val,Padre)),
  inducir(SubEjs,Nodo,RestoAtrs,Umbral), !,
  particion(Vals,Atr,Ejs,Padre,RestoAtrs,Umbral).
```

Ejecutando todo el experimento

Así la sesión para construir el árbol de decisión para jugar tenis es como sigue:

```
?- id3.
Con que archivo CSV desea trabajar: "tenis.csv".
14 ejemplos de entrenamiento cargados.
Tiempo de inducción:
% 5,264 inferences, 0.001 CPU in 0.001 seconds (95% CPU, 6326923 Lips)

cielo=lluvioso
  viento=fuerte => [no/2]
  viento=debil => [si/3]
cielo=nublado => [si/4]
cielo=soleado
  humedad=normal => [si/2]
  humedad=alta => [no/3]
true.
```

Predicados auxiliares

Algunos predicados auxiliares incluyen:

```
 genera_nodo_id(M) :-
   retract(id(N)),
   M is N+1,
   assert(id(M)), !.

 genera_nodo_id(1) :-
   assert(id(1)).

 eliminar(X,[X|T],T) :- !.

eliminar(X,[Y|T],[Y|Z]) :-
  eliminar(X,T,Z).

 subconjunto([],_) :- !.

 subconjunto([X|T],L) :-
   member(X,L), !,
   subconjunto(T,L).

 maximo([X],X) :- !.
```

```
21  maximo([X/M|T],Y/N) :-
22     maximo(T,Z/K),
23     (M>K,Y/N=X/M ; Y/N=Z/K), !.
24
25  % elimina las ocurrencias de ejemplo y nodo en el espacio de trabajo
26
27  reset :-
28    retractall(ejemplo(_,_,_)),
29    retractall(nodo(_,_,_)).
```

4.3 LECTURAS Y EJERCICIOS SUGERIDOS

La búsqueda en espacios de soluciones de problemas en IA, inicia con los trabajos fundacionales de Simon y Newell [86] y Newell y Simon [62]. Nilsson [63] provee los fundamentos teóricos de lo que se ha constituido como una sólida área de investigación en IA. El capítulo tres del libro de Russell y Norvig [77] aborda en detalle las búsquedas en espacios de soluciones. Stobo [89] presenta una aproximación a la solución de problemas basada en Prolog. Los algoritmos presentados en este capítulo han sido tomados y/o adaptados del libro de Bratko [10], capítulos 11, 12 y 13.

ID3 fue propuesto originalmente por Quinlan [71], quien posteriormente introdujo C4.5 [73]. En 1995 [72] propone mejoras a éste último algoritmo, para el manejo de atributos con valores continuos. El capítulo 18, sección 3, de Russell y Norvig [77] aborda la inducción de árboles de decisión a partir de datos. Rokach y Maimon [75] una revisión muy completa de esta técnica de aprendizaje, y sus variaciones, en el contexto de la minería de datos. Kotsiantis [47] ofrece otra revisión más compacta y reciente de las técnicas de inducción de árboles de decisión en la IA. Nuestra implementación es una extensión de la propuesta por Markov [3].

Ejercicios

Ejercicio 4.1. *Me estoy cambiando de casa y debo llevar a mi casa nueva a mi perro, mi gato y mi hamster que, sobra decirlo, no se llevan muy bien entre ellos. Mi mini auto solo me permite llevar a una mascota conmigo. De manera que, por ejemplo, puedo llevarme al gato, dejando solos al hamster y al perro, pero no puedo dejar juntos a éste último y al gato, ni al gato y al perro. Escribir un programa en Prolog para encontrar los movimientos válidos para pasar todas mis mascota de una casa a otra. Implemente una solución al problema mediante una búsqueda en el espacio de soluciones del problema.*

Ejercicio 4.2. *Modifique el caso inventado de la ruta entre ciudades para reflejar el caso de las principales ciudades del estado Veracruz (Veracruz, Boca del Río, Xalapa, Córdoba, Orizaba, Poza Rica, Tuxpam, Coatzacoalcos, Minatitlán). Extienda el programa de búsqueda adecuado para reportar la ruta más corta entre dos ciudades en términos de kilómetros recorridos y costo (casetas y gasolina).*

Ejercicio 4.3. *Pruebe el ejemplo presentado para ilustrar un caso donde se puede generar sobre ajuste (Definición 4.1). ¿Son consistentes los resultados con lo esperado? Justifique su respuesta.*

Ejercicio 4.4. *Escriba un procedimiento que convierta un árbol de decisión a un conjunto de reglas de clasificación. ¿En qué orden deben agregarse a la memoria de Prolog?*

Ejercicio 4.5. *Substituya la medida ganancia de información, por radio de ganancia en el algoritmo ID3. ¿Qué beneficios observa con el cambio?*

3 http://www.cs.ccsu.edu/~markov/