Un Experimento en Aprendizaje Automático y Evaluación Parcial

David A. Fuller †
Sacha Bocic

Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile
Casilla 6177
Santiago 22
Chile

Resúmen: En este trabajo se presenta un experimento en aprendizaje automático, y específicamente, en generalización basada en ejemplos. Primero, se presenta una introducción al tema de aprendizaje y al de generalización basada en ejemplos. Además, se explican los algoritmos conocidos y sus problemas. Luego, se presentan varias extensiones a dicho algoritmos con el fin de permitir el aprendizaje de ejemplos de gran tamaño. Finalmente, se prueba el nuevo algoritmo con un ejemplo del campo de los lenguajes de programación, y se entregan conclusiones y trabajo futuro.

† E-mail: dfuller@{dcc.puc.cl, pucing.bitnet}, fax: +(56-2) 552-4054.

1. Introducción

Todo sistema de aprendizaje realiza una transformación de la información provista por un "tutor" (o el medio ambiente) de tal forma de almacenarla para su posterior uso. El grado de complejidad en la transformación de la información determina el nivel de complejidad de la técnica. Así, a mayor grado de complejidad, menor es el esfuerzo que debe realizar el tutor y mayor el esfuerzo del sistema.

La técnica que menor grado de transformación de la información involucra es el "aprender de memoria", donde el sistema sólo se limita a aceptar y almacenar lo que el tutor indica. Otra técnica es la conocida como "aprender por instrucción", en la cual se hace alguna transformación de la información pero sólo a nivel sintáctico. En "aprendizaje por deducción", el sistema infiere a partir del conocimiento, conclusiones útiles las cuales almacena para el futuro. Estas deducciones tienen la característica de ser preservadoras de la verdad. En otras palabras, de un conjunto de reglas lógicamente verdaderas se deducen hechos que también son verdaderos. En "aprendizaje por analogía" tenemos una combinación de inferencia deductiva e inferencia inductiva. En esta técnica, descripciones de distintos dominios son comparados para determinar una estructura común, la cual sirve como base para transformar valores desde un dominio al otro. La determinación de la estructura común involucra un proceso de inferencia, y la transformación de un dominio al otro involucra un proceso de deducción. "Aprendizaje inductivo" es la técnica que más transformación de la información involucra y por lo tanto, es la técnica cuya implementación presenta mayor grado de dificultad. En esta técnica, la inferencia es inductiva, lo que quiere decir que la inferencia comienza con un conjunto de hechos y se produce una generalización de estos hechos. Este tipo de inferencia (al contrario de la inferencia deductiva) no es preservadora de la verdad, pues es posible inferir una regla falsa a partir de un conjunto de reglas cuyo valor de verdad es verdadero. Por ejemplo, si sabemos que mi gato tiene el pelo color negro, no es válido deducir que todos los gatos tienen el pelo color negro.

El aprendizaje inductivo se divide en dos sub-áreas, "aprendizaje basado en ejemplos" y "aprendizaje por observación y descubrimiento". En aprendizaje basado en ejemplos, el objetivo consiste en construir una generalización capaz de aceptar todos los ejemplos positivos de algún concepto objetivo y de rechazar todos los ejemplos negativos. Existe aprendizaje basado en múltiples ejemplos y

aprendizaje basado en un solo ejemplo. Por último, en aprendizaje por observación y descubrimiento, el sistema busca patrones o reglas que le permitan explicar todas o la mayoría, de las observaciones que éste hace. Este proceso se realiza sin la guia de un tutor.

Teniendo esto en cuenta, nos concentraremos en el estudio de una de las técnicas de aprendizaje conocida como "Generalización Basada en Ejemplos", a la cual nos refereriremos por su sigla en inglés **EBG** (Explanation Based Generalization), la que se encuentra en el área de aprendizaje inductivo. En este artículo, se presentarán algunas experiencias en aprendizaje basado en ejemplos, lo que llevará a proponer un nuevo algoritmo. Se demostrará la potencialidad de este nuevo algoritmo con un ejemplo en el área de lenguajes de programación. El sistema será capaz de aprender un lenguaje imperativo y procesar programas en dicho lenguaje. Hasta la fecha, los algoritmos tradicionales no son capaces de procesar ejemplos de esa complejidad. Además, se hará la conexión cada vez más evidente entre EBG y evaluación parcial, esta última, una técnica de evaluación simbólica [Fuller 89].

2. Generalización Basada en Ejemplos

El término EBG es mencionado por primera vez en [Mitchell 86]. Esta técnica ataca el problema de formular conceptos generales a partir de un solo ejemplo. El algoritmo que utiliza EBG consta de dos etapas. En la primera, se analiza el ejemplo de entrenamiento y se construye una explicación de porqué el ejemplo de entrenamiento es una instancia del concepto a generalizar. La segunda etapa consiste en generalizar la explicación para obtener una reformulación del concepto.

La explicación que construye EBG forma un árbol de prueba [Lloyd 84]. Antes de continuar, es necesario definir cuatro conceptos que son importantes en todo sistema EBG .

Definición 2.1: Concepto objetivo corresponde a una definición no operacional del concepto a aprender. ❑

Definición 2.2: Criterio operacional corresponde a la especificación del tipo de definiciones de predicados que se consideran operacionales. En [Mitchell 86], se

representa este criterio como una lista de predicados observables (hechos) o fácilmente calculables. Otra forma de especificar los predicados operacionales consiste en especificar en una lista, los predicados que se consideran no operacionales. Una descripción de un predicado se dice que es operacional si y sólo si es expresada en términos de predicados operacionales. ❑

Definición 2.3: Un ejemplo de entrenamiento corresponde a la descripción de un objeto, el cual es una instancia del concepto objetivo. ❑

Definición 2.4: Teoría de dominio corresponde a un conjunto de reglas que definen el dominio desde el cual el ejemplo de entrenamiento y el concepto objetivo pueden ser derivados. Las reglas deben ser capaces de demostrar que el ejemplo de entrenamiento es una instancia del concepto objetivo. ❑

Como se mencionó, el algoritmo consta de dos etapas, la primera que es la construcción del árbol de prueba y la segunda que consiste en la generalización, tomando como base el árbol de prueba.

La construcción del árbol de prueba se hace con el ejemplo de entrenamiento en la raiz y la construcción de los nodos se hace utilizando las reglas que conforman la teoría de dominio, el cual no necesariamente es un hecho. La expansión (o ramificación) del árbol de prueba se termina al llegar a un predicado operacional. Para determinar si un predicado es o no operacional, se utiliza el criterio operacional. Si para un ejemplo de entrenamiento no es posible construir un árbol de prueba, entonces el ejemplo de entrenamiento no es válido, ya que no es posible demostrar su validez utilizando la teoría de dominio y por lo tanto no es posible generalizar.

La generalización consiste en determinar un conjunto de condiciones suficientes bajo las cuales la explicación se sustenta en términos de predicados que cumplan con el criterio operacional. Esto, en términos del árbol de prueba, consiste en la conjunción de los nodos operacionales (hojas del árbol) pero con variables frescas. Este proceso es explicado en [Kedar-Cabelli y McCarthy 87] como la forma en que se propagan las sustituciones de la reglas, pero ignorando las sustituciones de los hechos cuando se crea el árbol de prueba.

Esto nos hace pensar que si bien es cierto que las dos etapas del algoritmo son independientes, las podríamos realizar simultaneamente. Es decir, podríamos construir el árbol de prueba y al mismo tiempo podríamos ir construyendo nuestra generalización, pero teniendo cuidado de no instanciar variables en los predicados operacionales.

Supongamos que utilizamos el conjunto de reglas de la Figura 2.5 como una teoría de dominio, el predicado `puedecomprar(juan,ferrari)` como un ejemplo de entrenamiento y buscamos la generalización de `puedecomprar(X,Y)` considerando como predicados operacionales a los hechos (`esauto`, `sevende`, `gusta`, `costo`, `ahorro` y `prestamo`).

```
puedecomprar(Persona, Auto) :-
                    gusta(Persona,Auto),
                    sevende(Auto),
                    pagable(Persona, Auto).
pagable(Persona, Auto) :-
                    costo(Auto, X),
                    ahorro(Persona, Xa),
                    Xa > X.
pagable(Persona, Auto) :-
                    costo(Auto, X),
                    ahorro(Persona, Xa),
                    Xa < X,
                    prestamo(Persona, X-Xa).

esauto(ferrari).
sevende(ferrari).
gusta(juan, ferrari)
costo(ferrari,300000).
ahorro(juan,0).
prestamo(juan,300000).
```

Figura 2.5: Ejemplo con teoría de dominio.

Así,en una primera etapa, EBG comienza a construir el árbol de prueba para el ejemplo de entrenamiento. Simultáneamente, construimos una "copia" del árbol de prueba, pero con variables sin instanciar (es decir, variables libres). A esta copia le llamaremos árbol generalizado. Inicialmente, se crea el árbol de prueba y el generalizado con un sólo nodo, tal como lo muestra la Figura 2.6.

Figura 2.6: Arboles parcial y generalizado al iniciar el proceso.

Utilizando la teoría de dominio, es posible expandir los árboles de prueba y generalizado, como se muestra en la Figura 2.7, donde se ha utilizado para dicho efecto una representación de árboles AND/OR [Bratko 90].

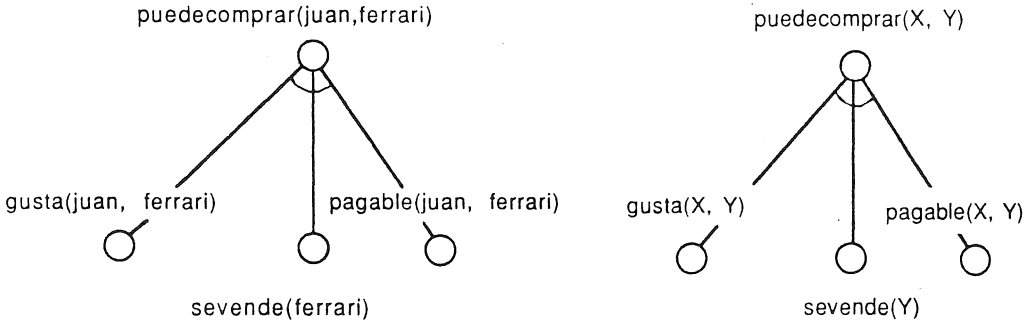

Figura 2.7: Arboles de prueba y generalizado durante el proceso.

Los predicados `gusta(juan,ferrari)` y `sevende(ferrari)` son considerados predicados operacionales, puesto que son hechos. Como sabemos, la expansión de una rama del árbol termina al llegar a un nodo o predicado operacional, por lo que sólo nos queda por expandir el nodo `pagable(juan, ferrari)`. La expansión de este nodo nos lleva al árbol final, ya que todos los nodos de las hojas son operacionales. El árbol generalizado final se muestra en la Figura 2.8 (sólo se grafican las expansiones exitosas).

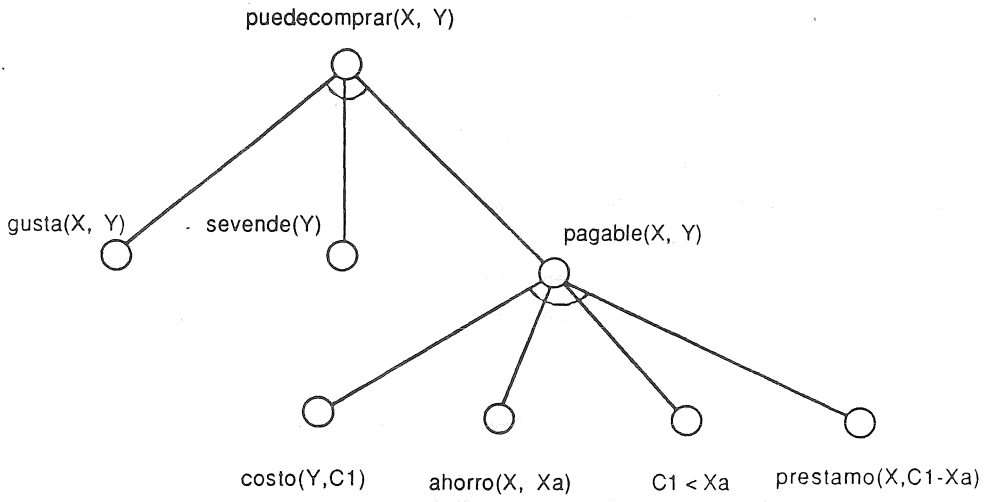

Figura 2.8: Arbol generalizado al final del proceso.

Una vez que se termina de construir el árbol de prueba, estamos en condiciones de generar la reformulación del concepto objetivo (`puedecomprar(X, Y)`) en términos de los predicados operacionales. Esto se hace formando la conjunción de los predicados operacionales del árbol generalizado, obteniendo el siguiente resultado:

Reformulación = `gusta(X,Y), sevende(Y), costo(Y, C1),`
`ahorro(X, Xa), Xa<C1, prestamo(X, C1-Xa)`

Resumiendo, hemos obtenido una reformulación del concepto objetivo en términos de predicados operacionales, guiados por el ejemplo de entrenamiento.

3. Implementación de EBG1

A continuación se presenta una implementación del algoritmo EBG tomada de [van Harmelen y Bundy 88] que le llamaremos **EBG1**.

```
ebg1( Hoja, HojaGen, HojaGen) :-
        operacional(Hoja), ! ,
        call(Hoja).[1]
ebg1( (Goal1,Goal2), (GenGoal1,GenGoal2), (Hojas1,Hojas2) ) :-
         ebg1(Goal1, GenGoal1, Hojas1),
         ebg1(Goal2, GenGoal2, Hojas2).
ebg1( Goal, GenGoal, Hojas) :-
         clause(GenGoal, GenClause),[2]
         copy( (GenGoal:- GenClause), (Goal:-Clause) ),[3]
         ebg1( Clause, GenClause, Hojas).
```

Figura 3.1: Implementación de `ebg1`.

El predicado `ebg1` toma como primer argumento un ejemplo de entrenamiento, el segundo argumento corresponde al concepto objetivo y entrega en el tercer argumento la reformulación del concepto objetivo en términos de predicados operacionales. En este algoritmo, la teoría de dominio y el criterio operacional se encuentran como entrada en forma implícita, ya que están en la base de datos de PROLOG.

Ejemplo 3.2: El primer ejemplo que analizaremos corresponde a un *clásico* de la literatura de aprendizaje y lo usaremos como entrada para la implementación de EBG1. Consideremos la siguiente base de datos en PROLOG:

```
% Concepto Objetivo
apilar (X,Y) :- mas_liviano(X,Y).
% Ejemplo de Entrenamiento
esta_sobre(caja1,mesa1).
volumen(caja1,10).
es_un(caja1,caja).
es_un(mesa1,mesa).
```

---

1. `call(Hoja)` es equivalente a un llamado al predicado que representa la variable `Hoja`.

2. `clause(GenGoal,GenClause)` es verdadero si existe en la base de datos de Prolog una definición de la forma `GenGoal :- GenClause`.

3. El predicado `copy` se usa para que el árbol generalizado sea una copia exacta del árbol de prueba pero con variables frescas.

```
color(mesa1,azul).
color(caja1,rojo).
densidad(caja1,10).
% Teoría de Dominio
mas_liviano(X,Y) :- peso(X,W1),
peso(Y,W2),
menor(W1,W2).
peso(X,500) :- es_un(X,mesa).
peso(X,Y)  :- volumen(X,V),
densidad(X,D),
multiplica(V,D,Y).
% Criterio Operacional
operacional(Goal) :- member(Goal, [multiplica(_,_,_), menor(_,_)
                                   esta_sobre(_,_), volumen(_,
                                   es_un(_,_), color(_,_),
                                   densidad(_,_)] ).
```

Figura 3.3: Ejemplo clásico.

El predicado `menor(X,Y)` es cierto si el valor de la variable `X` es menor que el valor de la variable `Y`. En forma análoga, el predicado `multiplica(X,Y,Z)` es verdadero si el valor de la variable `Z` es igual al producto de los valores de las variables `X` e `Y`. Estos predicados los consideramos predefinidos por el lenguaje.

Así, para generalizar el predicado `apilar` usamos la implementación de EBG1 descrita antes, de la siguiente forma:

```
ebg1(apilar(caja1,mesa1), apilar(X,Y), Reformulacion).
```

donde `apilar(caja1,mesa1)` corresponde al ejemplo de entrenamiento, `apilar(X,Y)` es el concepto objetivo y `Reformulacion` es un variable inicialmente desinstanciada que entregará la reformulación del concepto objetivo, es

4. El predicado `member` recibe como primer argumento un elemento y como segundo una lista. Si el elemento pertenece a la lista, entonces el predicado es verdadero.

decir, una nueva definición del predicado `apilar` en función de los predicados operacionales. El resultado obtenido es:

```
Reformulacion =  volumen(X,Vx), densidad(X,Dx), multiplica(Vx,Dx,Mx),
                 es_un(Y,mesa),  menor(Mx,500).
```

Es decir, cualquier cosa de menos de 500 kilos se puede poner sobre una mesa. Nótese que en la reformulación, todos los predicados tienen variables sin instanciar salvo el predicado `menor`, que tiene una constante (500). Esta constante proviene de la instanciación que se produce al expandir el árbol de prueba y el árbol generalizado según la primera definición del predicado `peso`. Como este predicado no es operacional, la variable se instancia.

4. Extensiones a EBG

En el Ejemplo 3.2, los predicados operacionales eran casi todos hechos ("facts"), salvo los predicados `multiplica` y `menor` que son predicados predefinidos por el lenguaje. Además, el ejemplo de entrenamiento contenía suficiente información como para "guiar" al algoritmo por el espacio de búsqueda, de tal forma de encontrar una solución única. Estos dos hechos no son siempre verdaderos, especialmente cuando se trabaja con ejemplos de mayor tamaño. A continuación, se propondrán dos extensiones al algoritmo EBG1, las que se incorporarán en una nueva versión del algoritmo EBG.

El algoritmo EBG1 expande cada rama del árbol de prueba hasta llegar a un predicado operacional. Como se sabe, en este punto no se producen instanciasiones en las variables del árbol generalizado y el efecto en la reformulación del predicado original se limita a la incorporación de éste a dicha reformulación. Gráficamente:

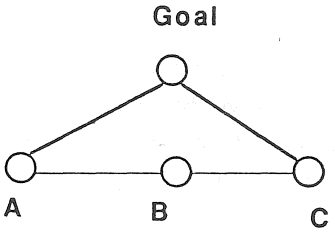

Figura 4.1: Primer problema de EBG1.

Supóngase que para el árbol de prueba de la figura, los predicados `A`, `B` y `C` son predicados operacionales. En consecuencia, la reformulación del predicado `Goal` es la conjunción de los predicados operacionales de dicho árbol, de la forma `Goal :- A, B, C.`

Ahora supongamos que no se desea que este predicado `B` aparezca en la reformulación del concepto objetivo. Si es este el caso, se tiene que eliminar al predicado `B` de los predicados operacionales. Al hacer esto, el algoritmo tendrá que expandir totalmente este predicado y se va a llegar a una conjunción de predicados que corresponde a la generalización de este predicado `B`.

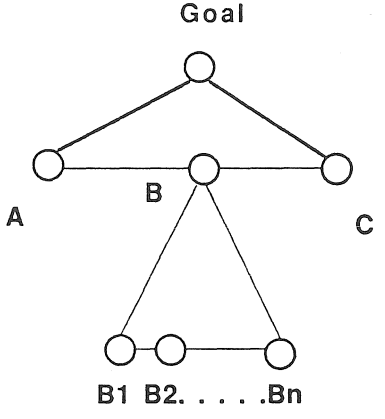

Figura 4.2: Expansión de `B`.

De esta forma nos encontramos con que la solución es peor que el problema, pues si no queremos al predicado `B`, menos queremos a la generalización de

este predicado[5]. Además, no se puede optar por sacar a este predicado de la reformulación del concepto objetivo pues se estaría generando una reformulación inválida. En efecto, los predicados reciben variables como argumento, esto implica que, o se instancian variables o se verifica que se cumpla alguna condición. Si se elimina a este predicado "no deseado" estamos eliminando estas instanciaciones o verificaciones de condiciones y por lo tanto la reformulación no será válida.

Una forma de superar este problema consiste en reemplazar la expansión del árbol de prueba y del árbol generalizado por un meta-llamado a este predicado y su respectiva copia. De esta forma, todas las variables que se tienen que instanciar en este predicado lo hacen, así como también se verifican todas las condiciones, pero no se incluye este predicado no deseado en la reformulación. Para ello, el usuario debe definir los predicados que son posibles de ejecutar y para que el algoritmo modificado los ejecute.

Es posible especificar los predicados "ejecutables" de una forma similar a la que se usa para especificar los predicados operacionales, es decir, mediante una lista y a través del predicado `member` se detecta si el predicado pertenece a dicha lista. La forma de incorporar esta idea al algoritmo primitivo es mediante la extensión de la definición original del predicado `ebg1` y la definición del predicado `ejecutable`, de la siguiente forma:

```
ebg(Goal,GenGoal,[]) :-
                    ejecutable(Goal),
                    call(Goal),
                    call(GenGoal),!.

ejecutable(Goal) :- member(Goal, [lista de predicados ejecutables]).
```

Figura 4.3: Incorporación de la primera extensión.

En el Ejemplo 3.2, se vió que el ejemplo de entrenamiento nos guiaba por el espacio de búsqueda de tal forma de llevarnos a una única reformulación del concepto objetivo. En efecto, en este ejemplo para el predicado `peso` tenemos

5. Si el predicado `B` es un hecho, no hay forma de eliminarlo de la reformulación.

dos definiciones, lo que en términos de un árbol AND/OR constituye un nodo-or, sin embargo, el algoritmo escoge sólo una de ellas.

Este es precisamente el rol del ejemplo de entrenamiento, guiar al algoritmo a través del árbol de prueba, escogiendo algunos nodos para expandir y descartando otros. ¿Pero cómo enfrentamos el problema de no saber cuál nodo expandir si el ejemplo de entrenamiento no tiene la suficiente información? Una posible solución consiste en modificar el algoritmo EBG para que detenga la expansión de los árboles al encontrarse con nodos-or en los cuales se presenta este problema. Esta solución presenta el problema que en muchos casos la terminación anticipada de la expansión de los árboles nos lleva a obtener árboles muy poco profundos, con lo que la reformulación del concepto objetivo no presenta un gran avance.

La otra solución consiste en expandir cada una de las soluciones, de tal forma de obtener tantas reformulaciones del concepto objetivo como expansiones del árbol de prueba se realicen, pero ¿podemos estar seguros de que el número de expansiones (y reformulaciones) será finito? Ciertamente no, y como prueba de ello analicemos el siguiente ejemplo, en el que se desea reformular el predicado `largolista(Lista,N)`, cuya definición es la siguiente:

```
largolista([] , 0).[6]
largolista([Cabeza | Cola] , N) :-
                    largolista(Cola, Nc),
                    N is Nc + 1.
```

Figura 4.4: Definición de `largolista`.

Una de las formas de usar este predicado es recibiendo como primer parámetro una lista y entregando en el segundo parámetro el número de elementos que hay en la lista. Supongamos que el ejemplo de entrenamiento no tiene ninguna variable instanciada. Al aplicar el algoritmo EBG1 nos encontraremos con infinitas soluciones. En efecto, esto se produce ya que en el ejemplo de entrenamiento, la primera variable (la que representa a la lista) no está instanciada y por lo tanto puede ser unificada tanto con la lista vacía (`[]`) como con la lista `[Cabeza|Cola]`. Luego,

6. `[]` es la forma en que se denota una lista vacia.

existen dos definiciones del predicado `largolista` que pueden ser utilizadas para expandir el árbol de prueba. Al tener dos posibles expansiones para el árbol, tenemos dos posibles reformulaciones para el concepto objetivo, pero como una de las definiciones es recursiva, al expandirla, nuevamente tenemos dos posibles ramas y así sucesivamente. Gráficamente:

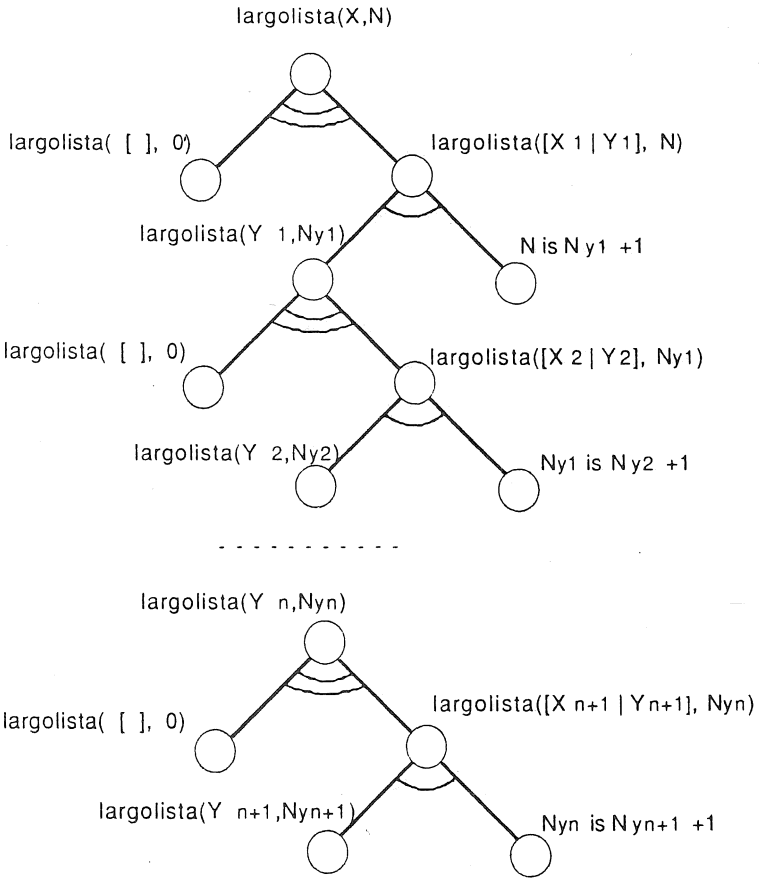

Figura 4.5: Infinitas soluciones en la expansión de `largolista`.

Esto hace imprecindible determinar cuales predicados son peligrosos de expandir al momento de construir el árbol de prueba para el ejemplo de entrenamiento. Una forma sencilla de hacer esto consiste (nuevamente) en dejar al usuario que especifique cuales predicados son peligrosos de expandir. Sin em-

bargo, aún nos queda planteada la duda de como hacerlo. Si lo hacemos en forma similar a la especificación de predicados operacionales, es decir, por medio de una lista, estaríamos siendo demasiado restrictivos ya que no todos los predicados `largolista(X,N)` son peligrosos de expandir. Por este motivo, lo más recomendable es especificar el predicado y la condición que lo hace peligroso. Por ejemplo, para `largolista` lo hacemos de la siguiente forma:

```
peligroso( largolista(X,N) ) :- \+ ground(X).
```
[7]

La incorporación de las extensiones mencionadas al algoritmo EBG1 nos lleva a la siguiente implementación para el algoritmo EBG2.

```
ebg2(Goal,GenGoal,[]) :- ejecutable(Goal),
                         call(Goal),
                         call(GenGoal),!.
ebg2(Goal,GenGoal,[GenGoal]) :- operacional(Goal),
                         peligroso(Goal),!.
ebg2(Goal,GenGoal,[GenGoal]) :- operacional(Goal),
                         call(Goal),!.
ebg2( (SubGoal,ListGoals), (SubgenGoal,ListgenGoal), Refor) :-
                         ebg2(SubGoal,SubgenGoal,Ref1),
                         ebg2(ListGoals,ListgenGoal,Ref2),
                         append(Ref1,Ref2,Refor).
ebg2( Goal, GenGoal, Ref) :-
                         clause(Goal,Body),
                         copy((Goal:-Body), (GenGoal:-GenBody)),
                         ebg2(Body,GenBody,Ref).
```
[8]

Figura 4.6: Implementación de EBG2.

7. `\+ ground(X)` es verdadero si X es una variable no instanciada

8. `append` recibe tres listas como argumentos, digamos `X`,`Y` y `Z`. El predicado es verdadero si el valor de `Z` es igual (sintacticamente) al valor de la concatenación de `X` e `Y`.

Nótese que se reemplazó la utilización del constructor de PROLOG (_ , _) del tercer argumento del predicado `ebg1` por la utilización de listas y el correspondiente predicado `append`.

5. Ejemplo

Hasta el momento, sólo se han presentado ejemplos bastante sencillos, en los cuales la reformulación del concepto objetivo no involucra mayores dificultades. A continuación, aplicaremos el nuevo algoritmo de EBG a un ejemplo más complejo, el que consiste en un intérprete de un lenguaje llamado Norma3, el cual recibe como entrada un programa que a su vez recibe datos de entrada. En este caso tenemos programas que trabajan en niveles de abstracción distintos. Gráficamente, esto lo vemos así:

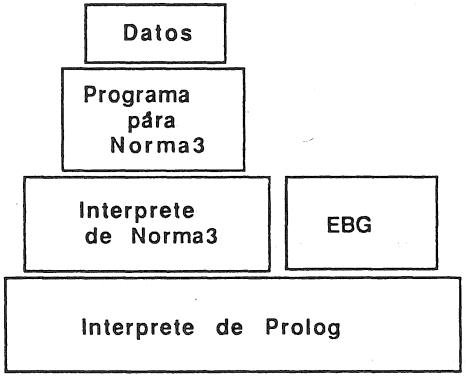

Figura 5.1: Niveles de abstracción en el ejemplo.

En este ejemplo, la teoría de dominio corresponde al conjunto de reglas y hechos que definen un interprete de Norma3, el que corresponde a un lenguaje imperativo que cuenta con cinco registros (identificados con las letras **v** a **z**), dos de ellos de entrada (**w** y **x**), dos internos o acumuladores (**v** e **y**) y uno de salida (**z**). En Norma3 sólo se manejan números enteros representados como una lista de símbolos "1" . Así, por ejemplo, el valor 3 será representado como [1, 1, 1]. El lenguaje permite ejecutar sentencias de asignación, operaciones matemáticas de suma, resta, saltos condicionales (if_then) y saltos incondicionales (goto´s). Además, Norma3 posee la capacidad de manejar listas de números. La cabeza, o

primer elemento de una lista, almacenada en un registro i se identifica con I, es decir, con la letra mayúscula que identifica el registro.

Este intérprete no cuenta ni con analizador léxico ni sintáctico, por lo que los programas escritos en Norma3 deben, por el momento, mantener el formato de listas de PROLOG. El nombre Norma3 obedece a motivos históricos, pues este interprete es una derivación de Norma2, lenguaje de similares características pero más simple [Fuller 89].

El concepto objetivo corresponde a la interpretación de un programa escrito en Norma3 que ordena una lista de mayor a menor usando un algoritmo de inserción. El código correspondiente a este programa es el siguiente:

```
0    [ y:=W,                          y = Cabeza de lista del registro w
1    x:=w,                            x = w
2    [ifX<y,1,1,1,1]                  IF Cabeza lista en x < y goto 4
3    y:=X,                            y = Cabeza de lista de registro x
4    x--,                             x = x - primer elemento
5    [ifx=0,1,1,1,1,1,1,1,1],         IF x = 0 goto 7
6    [goto,1,1],                      goto 2
7    w-y,                             w = w-y
8    v:=+y,                           v = v concatenado con la lista
                                      que contiene y
9    [ifw=0,1,1,1,1,1,1,1,1,1,1,1,1], IF w = 0 goto 11
10   [goto],                          goto 0
11   y:=v   ]                         y = v.
```

Figura 5.2: Programa sort en NORMA3.

Recordemos que la teoría de dominio utilizada por EBG corresponde a un intérprete, por lo tanto, los predicados operacionales, peligrosos y ejecutables pertenecerán al conjunto de reglas y hechos que definen a este intérprete. A continuación se presenta la declaración de estos predicados para el ejemplo:

```
operacional(Goal) :-
                    member(Goal,[without(_,_,_),length(_,_),
                                 add_one(_,_),sub_one(_,_)]).
operacional(Goal) :-
                    builtin(Goal).
```

[9]

9. `builtin(Goal)` es verdadero si `Goal` es un predicado predefinido del lenguaje.

```
ejecutable(Goal) :- clause(Goal,true),
                    \+ operacional(Goal).[10]
ejecutable(Goal) :- member(Goal, [skip(_,_,_), copy_val(_,_,_)).

peligroso( length(X,_) ) :-
                    \+ ground(X).
peligroso( Goal ) :-
                    builtin(Goal),
                    Goal =.. [_|Vars],
                    allground(Vars).[11]
```

Figura 5.3: Definición de meta-predicados para el ejemplo.

Así, se ve que los predicados `without`, `length`, `add_one` y `sub_one`, definidos en el intérprete de NORMA3, así como los predicados predefinidos de PROLOG, se considerarán como operacionales. Por otro lado, los predicados `skip` y `copy` de NORMA3 fueron definidos como ejecutables. Aquí, además se definió como ejecutable, cualquier predicado que cumpla con ser un hecho y no ser operacional.

Utilizaremos dos ejemplos de entrenamientos. En el primero, daremos como entrada al algoritmo EBG2, el programa de ordenamiento y una lista a ordenar, totalmente instanciada. Esto lo hacemos mediante el predicado `ebg2` de la siguiente forma, donde la variable `Pgm` ha sido instanciada previamente al programa sort en NORMA3.

```
ebg2(execute(Pgm, [ [1,1,1], [1,1,1,1], [1]], _, Salida),
        execute(Pgm, ListaIN, _, ListaOut), Reformulacion)
```

10. Todo predicado de la base de datos de PROLOG que sea un hecho y que no sea operacional es considerado ejecutable.

11 Esta última definición del predicado `peligroso` nos dice que no se debe expandir un predicado predefinido por el lenguaje cuando todos los argumentos no están instanciados. Esto es para evitar la expansión de cosas como `X > Y` ó `X is Y+1`.

Con esto, el resultado que se obtiene (una vez renombradas las variables y eliminadas las variables inútiles de la definición de `execute`, tal como `Pgm` que corresponde a una constante), es el siguiente:

```
execute_1( [Cabeza | Resto], [Mayor, Medio, Menor] ) :-
                length(Cabeza,N1),length(Cabeza,N2), N1 >= N2,
                sub_one([Cabeza | Resto],[Mayor | L1]),
                length(Mayor,N3),length(Cabeza,N4), N3 >= N4,
                sub_one([Mayor | L1],[Cabeza2 | Resto2]),
                length(Cabeza2,N5),length(Mayor,N6), N5 < N6,
                sub_one([Cabeza2 | Resto2],[]),
                without(Mayor,[Resto | Resto],[Medio | L2]),
                length(Medio,N7),length(Medio,N8), N7 >= N8,
                sub_one([Medio | L2],[Cabeza3 | Resto3]),
                length(Cabeza3,N9),length(Medio,N10), N9 < N10,
                sub_one([Cabeza3 | Resto3],[]),
                without(Medio,[Medio | L2],[Menor | L3]),
                length(Menor,N11),length(Menor,N12), N11 >= N12,
                sub_one([Menor | L3],[]),
                without(Menor,[Menor | L3],[]).
```

Figura 5.4: Resultado del ejemplo.

Nótese que esta redefinición del predicado `execute` en términos de los predicados operacionales, cumple con ordenar una lista de tres elementos siempre y cuando estén en el mismo orden de la lista del ejemplo de entrenamiento. En otras palabras, se aprendió como ordenar una lista con ciertas características. En este caso, el tipo de listas que se aprendió a ordenar es bastante específico.

Intentemos un segundo ejemplo de entrenamiento, que nos permita ordenar cualquier lista de tres elementos. Para ello, utilizaremos el mismo programa de ordenamiento, pero esta vez la lista a ser ordenada estará instanciada sólo en la estructura, es decir, aplicaremos

```
ebg2(execute(Pgm, [ X, Y, Z], _, Salida),
        execute(Pgm, ListaIN, _, ListaOut), Reformulacion)
```

donde `X`, `Y` y `Z` son variables que están sin instanciar. El resultado que se obtiene es un conjunto de 64 definiciones para el predicado `execute`, el que es capaz de ordenar cualquier lista de tres elementos. Es importante destacar que la reformulación del ejemplo anterior contiene seis comparaciones y las 64 redefiniciones corresponden a todas las posibles combinaciones que se producen al realizar seis comparaciones (pues 2 elevado a 6 es 64).

6. Conclusiones

Se ha presentado una extensión al algoritmo original de aprendizaje basado en ejemplos, obteniendo generalizaciones con menor número de clausulas y por lo tanto más eficientes. Las extensiones al algoritmo tradicional, incorporándoles meta-predicados para dirigir el proceso de construcción del árbol de prueba permitieron la experimentación con ejemplos de gran tamaño, lo que no es posible con los algoritmos actuales.

Sin embargo, es necesario profundizar el estudio de los criterios para determinar cuales predicados pueden ser considerados operacionales, ejecutables o peligrosos. En este artículo, dichos predicados deben ser definidos por el usuario, lo que no siempre es factible. En este último aspecto, es posible beneficiarse de las similitudes entre EBG y evaluación parcial [van Harmelen y Bundy 88] [Fuller y Hoppe 91] para automatizar la detección de predicados peligrosos [Fuller 91]. Específicamente, se estima que la definición de los predicados `peligroso` y `ejecutable` podrían ser hechas en forma automática por el mismo sistema de aprendizaje, basado en técnicas de interpretación abstracta [Abramsky y Hankin 84].

Con esta técnica de aprendizaje es posible obtener mejoras en la eficiencia de la ejecución de los programas producidos de un orden de magnitud, con lo que la utilidad de la técnica queda de manifiesto. Sin embargo, existe una fuerte limitación asociada a la representación del conocimento. En efecto, al utilizar una representación del conocimiento basada en lógica de primer órden, sólo se pueden hacer generalizaciones en este contexto y quedean excluidas representaciones asociadas al razonamiento no-monótono o al tratamiento del tiempo.

Otra aplicación de esta técnica en la que se está trabajando es en el campo del lenguaje natural. Ahí, se piensa aplicar la técnica de EBG para explorar la idea de "tutores inteligentes" del lenguaje. En este caso, el algoritmo de EBG sobre una interfaz de lenguaje natural y una sentencia para dicho lenguaje podría construir el árbol de prueba. En caso de que la prueba no logre llegar a las hojas del árbol, significa que la sentencia entregada por el usuario fue incorrecta, y el sistema podría corregirla partiendo de los nodos no expandidos del árbol.

Agradecimientos

Este trabajo ha sido financiado gracias al apoyo del Fondo Nacional de Ciencia y Tecnología de Chile, FONDECYT, proyectos 90-0688 y 89-0591 y al proyecto CHI/88/022 del Programa de las Naciones Unidas para el Desarrollo.

Bibliografía

1. Abramsky, S. y Hankin, C. (eds.), Abstract Interpretation of Declarative Languages, Wiley (1987).
2. Bratko, I., PROLOG Programming for Artificial Intelligence, segunda edición, Addison-Wesley (1990).
3. Fuller, D., Partial Evaluation and Mix Computation in Logic Programming, tésis de PhD, Department of Computing, Imperial College of Science and Technology, Londres, Inglaterra (1989).
4. Fuller, D., Replacing the Loop Detection Scheme in Partial Evaluation of Logic Programs, en preparación (1991).
5. Fuller, D. y Hoppe, U., Explanation Based Generalization and Partial Evaluation Revisited, enviado a Artificial Intelligence (1991).
6. van Harmelen, F. y Bundy A., Explanation-Based Generalisation = Partial Evaluation, Artificial Intelligence 36 (1988).
7. Kedar-Cabelli, S.T. y McCarthy, L.T., Explanation-based Generalization as a resolution theorem proving, Proceedings of the Fourth International Machine Learning Workshop, Irvine, EE.UU. (1987).
8. Lloyd, J., Foundations of Logic Programming, Springer-Verlag (1984).
9. Mitchell, T.M., Keller, R.M. y Kedar-Cabelli, S.T., Explanation-based Generalization: A unifying view, Machine Learning Vol I (1986).

10. Michalski R.S., Understanding the nature of learning, Machine Learning Vol II (1986).
11. Simon, H.A., Why Should Machines Learn?, en Machine Learning: An Artificial Intelligence Approach, R.S. Michalski, J.G. Carbonell y T.M. Mitchell (Eds). (1983).