

AIBED
Algoritmo Inductivo Basado en Datos

Gildardo Medina Retamoza
Maestría en Inteligencia Artificial

Asesor Dr. Luciano García Garrido
Universidad de la Habana, Cuba

Revisor Dr. Manuel Martínez Morales
MIA U.V. Xalapa, Ver.

Revisor Dr. Christian Lemaitre León
LANIA Xalapa, Ver.

A mis padres por enseñarme a valerme por mi mismo.

Agradecimientos

Agradezco a todos mis amigos de la Maestría en Inteligencia Artificial por tantos momentos, por todas las risas, todas las lágrimas, todos los enojos y todos los reventones que juntos compartimos.

Especialmente agradezco a Luciano y Manuel por ser más que Maestros y Amigos para mí.

Índice General

Resumen	v
Introducción	vii
1 Programación Lógica	1
2 Programación Lógica Inductiva	4
2.1 Ejemplo de razonamiento Inductivo	6
2.2 Definición de PLI	8
2.3 Requerimientos de un sistema de PLI	9
2.4 Algoritmos de PLI Top-Down	9
2.5 Algoritmos de PLI Bottom-Up	11
3 AIBED	14
3.1 Objetivo	14
3.2 Intuición	15
3.2.1 Elementos para la ejecución	17
3.2.2 Ejemplo de archivos de entrada	18
3.3 Descripción de <i>AIBED</i>	19
3.4 Ciclo principal	20

3.4.1	Determinación del <i>Mejor Átomo</i>	22
3.4.2	Determinación del <i>Mejor Cuerpo</i>	23
3.4.3	Ejemplo ejecutado	25
4	Pruebas	27
4.1	Problema del Árbol Familiar	27
4.1.1	Aprendiendo la relación hermano	29
4.1.2	Aprendiendo la relación tio	32
4.1.3	Aprendiendo la relación primo	33
4.1.4	Aprendiendo la relación ancestro	35
4.2	Ejemplo de Animales	40
4.2.1	Resultados utilizando PROGOL 4.2	44
4.2.2	Resultados utilizando AIBED	45
5	Trabajo futuro y Conclusiones	46
5.1	Trabajo futuro	46
5.2	Conclusiones	47
A	Implementación de <i>AIBED</i>	51
A.1	Cargando <i>AIBED</i> en BinProlog	51
A.2	Llamando la ayuda de <i>AIBED</i>	52
A.3	Cargando un ejemplo	53
A.4	Disparando el proceso inductivo de <i>AIBED</i>	54

Resumen

Este trabajo de investigación tiene como objetivo proponer un algoritmo eficaz para la construcción de nuevas cláusulas a partir de un programa lógico.

Este objetivo se inscribe dentro de la *Programación Lógica Inductiva*, la cual ha sido definida como la intersección de la *Programación Lógica* y del *Aprendizaje Automático* [MR94], las cuales son dos áreas de intenso desarrollo dentro de la *Inteligencia Artificial*.

El algoritmo propuesto recibe como entrada un programa lógico P y da como salida otro programa lógico P' , donde P' incluye a lo menos todas las cláusulas de P y preferentemente incluye algunas cláusulas nuevas a partir de las cuales se deduzcan las cláusulas ejemplos de P , permitiendo su eliminación siendo las cláusulas nuevas una representación más compacta de las eliminadas.

Para la construcción de las nuevas cláusulas, se utiliza una heurística dirigida por los valores (constantes) que aparecen en las cláusulas que se presentan como ejemplos. La meta principal del algoritmo es encontrar un conjunto de cláusulas que expliquen un conjunto de ejemplos positivos (E^+) y negativos (E^-) –estos últimos no son indispensables– que se conocen como conjunto de entrenamiento (E). Dicho de otra forma, las nuevas cláusulas deben responder verdadero a todos los E^+ y falso a todos los E^- , de esta forma se busca mantener la consistencia entre P' y P .

Nuestro algoritmo trabaja sobre un lenguaje restringido, el cual contempla que todos los elementos de E^+ y E^- deben tener el mismo símbolo de predicados y la misma aridad, en el caso del conocimiento

previo o base de conocimientos (BC), las cláusulas que lo compongan solo pueden tener como argumento en su términos constantes y/o variables, los números son interpretados como símbolos. Además la Base de Herbrand de E deber ser un subconjunto de la Base de Herbrand de BC .

La implementación del algoritmo se probó con el problema de un árbol familiar, se escogió dicho problema por ser sumamente conocido y por resultar didáctico para mostrar algunos aspectos importantes de la implementación y finalmente se realizó una comparación con Prolog.

Introducción

Los mecanismos y técnicas que le permiten a una máquina adquirir cierto tipo de conocimiento son estudiados por una área de la *Inteligencia Artificial* (IA) que se ha denominado *Aprendizaje Automático* (AA).

Muchos de los procesos de AA pueden ser vistos como procesos de clasificación, es decir, como procesos que permiten particionar en clases las entidades u objetos de un universo de discurso. De modo que cuando se entrena a un sistema de AA lo que se busca es que el sistema sea capaz de encontrar una familia finita de clases, las cuales se definen mediante características distintivas de las entidades que las constituyen. Para ello, una estrategia es proporcionar al sistema de AA un conjunto de entrenamiento consistente en ejemplos *previamente clasificados*.

En el AA clásico el conjunto de entrenamiento y los procesos de aprendizaje utilizados típicamente ha sido representado por medio de lógica proposicional, lo cual constituye una limitante ya que existen ciertos dominios o universos que no pueden ser clasificados de manera efectiva a nivel proposicional.

Cuando un sistema de este tipo logra aprender a clasificar correctamente el conjunto de entrenamiento, se dice que ha logrado cierta capacidad de generalización si a su vez ha adquirido la habilidad de clasificar nuevos ejemplos como pertenecientes a alguna de las clases aprendidas.

Debido a que los sistemas de AA clásicos no utilizan ningún tipo de conocimiento previo cada vez que se ejecutan, es necesario que realicen el análisis completo de todo el conjunto de entrenamiento el cual

por lo general es amplio. Para reducir en cierta medida este problema la tendencia de algunos grupos de estudio del aprendizaje artificial buscan incorporar cierto tipo de conocimiento previo, además el uso de conocimiento previo es esencial para alcanzar un comportamiento inteligente [MR94].

La *Programación Lógica Inductiva* (PLI) integra características tanto del AA como de la *Programación Lógica* (PL), tomando de esta última su expresividad y riqueza de lenguaje para representar conocimiento que no puede ser representado por la lógica proposicional.

Este nuevo enfoque de aprendizaje artificial ha dado la pauta para el desarrollo de este trabajo.

Capítulo 1

Programación Lógica

En la lógica de predicados se denomina *fórmula elemental* a una fórmula de la forma $R(t_1, t_2, \dots, t_n)$ donde R es un símbolo de predicado n -ario y las t_i son términos. Se denomina *literal positivo* a una fórmula elemental y *literal negativo* a una fórmula elemental negada. Se denomina *cláusula* a una fórmula de la forma $A_1 \vee A_2 \vee \dots \vee A_n$, donde cada A_i es un literal. Una *cláusula definida* es una cláusula que contiene a lo sumo un literal positivo.

Un *programa lógico* P es un conjunto de cláusulas definidas, siendo una cláusula definida $A \vee \neg B_1 \vee \neg B_2 \vee \dots \vee \neg B_n$ representada de forma implicacional como $A \leftarrow B_1, B_2, \dots, B_n$.

Mediante esta notación, se denomina a A la *cabeza* de la cláusula y a B_1, B_2, \dots, B_n el cuerpo de la misma. Un *hecho* es una cláusula definida con el cuerpo vacío, esto es $n = 0$. Un *objetivo* es una cláusula definida con la cabeza vacía. Finalmente denominaremos *regla* a toda cláusula definida con cabeza y cuerpo no-vacío y *cláusula vacía* a la cláusula con cabeza y cuerpo vacíos.

A lo largo de este trabajo se asume que *cláusula* quiere decir cláusula definida y que todas las cláusulas son de *rango restringido*, esto es que todas las variables que ocurren en la cabeza de una cláusula deben ocurrir también en el cuerpo de la misma. Además se empleará *base de conocimiento (BC)* como sinónimo de programa lógico y denominaremos *cláusula básica* a toda cláusula que no contiene variables.

La *semántica declarativa* de un programa lógico interpreta los términos de una cláusula como denotando individuos de un dominio y a los predicados denotando relaciones entre los individuos del dominio. Sea L el lenguaje de un programa lógico P , el *Universo de Herbrand* (U_L) de P es el conjunto de todos los términos básicos de L que pueden construirse utilizando las constantes individuales y las funciones n -arias que aparecen en las cláusulas de P , y la *Base de Herbrand* (B_L) de P es el conjunto de todos los átomos básicos de L , es decir todas las fórmulas obtenidas aplicando los predicados de L a todos los elementos de U_L . Una *interpretación* de P en U_L es un subconjunto cualquiera (pudiendo ser incluso el conjunto vacío) de B_L . Se dice que una interpretación I de P es un *Modelo de Herbrand* de P si para todo ejemplo básico de cláusula $A \leftarrow B_1, \dots, B_n \in P$ se tiene que $A \in I$ si $B_1, \dots, B_n \in I$, la intersección de todas las $I \subseteq B_L$ tales que I es un modelo de P se denomina *Modelo Minimal* de P [GPM91]. Dado un programa lógico P y un objetivo O , O es una *consecuencia lógica* de P si todo modelo de P es también un modelo de O .

La *semántica operacional* de un programa lógico viene definida por una teoría de la prueba que consta de una regla de inferencia, el principio de resolución al cual esta asociado un algoritmo de unificación. El concepto de prueba asociado es un proceso de refutación el cual parte de la negación del objetivo o “teorema” que se pretende demostrar hasta generar, aplicando en cada paso *unificación + resolución*, la cláusula vacía. Es decir, dado un programa P y un objetivo O , O es deducible (ó es un teorema) a partir de P si a partir de $P \wedge \neg O$ se deduce la cláusula vacía.

El principio de resolución, como toda regla de inferencia deductiva aceptable, es:

sano: si O es deducible de P , entonces O es una consecuencia lógica de P .

completo: si O es una consecuencia lógica de P , entonces O es deducible de P .

Por último se introduce el concepto de *sub-sumción*: La cláusula c_1

θ -subsume a la cláusula c_2 si y sólo si existe una sustitución θ tal que $c_1\theta \subseteq c_2$. Donde c_1 es una generalización de c_2 (y c_2 es una especialización de c_1) bajo θ -subsumión [MR94].

Capítulo 2

Programación Lógica Inductiva

La *Programación Lógica Inductiva* es una extensión de la *Programación Lógica* (PL) en el sentido que a continuación se desarrolla.

Sea BC una base de conocimientos, que como ya fue advertido, en nuestro caso es un programa lógico. Sea E^+ un conjunto de hechos tal que E^+ no es deducible a partir de BC . Sea H un programa lógico al que denominaremos *hipótesis*, tal que a partir de BC y H se deduce (o implica lógicamente) E^+ . Diremos que en tal caso la hipótesis H *satisface* a E^+ , lo cual implica que H es una *compactación* de E^+ en el sentido de que pudiéndose deducir todos los elementos de E^+ a partir de H , entonces H los representa a todos. El proceso inferencial para hallar H a partir de BC y E^+ es un proceso inductivo: se realiza el hallazgo de H recorriendo un espacio de búsqueda que denominaremos el *espacio de las hipótesis*, el cual está estructurado por un orden parcial impuesto por la θ -subsumción, definido de la manera siguiente:

$H \leq H'$ si y solo si $H\theta \subseteq H'$, donde:

$H \leq H$ dado que $H\theta \subseteq H$

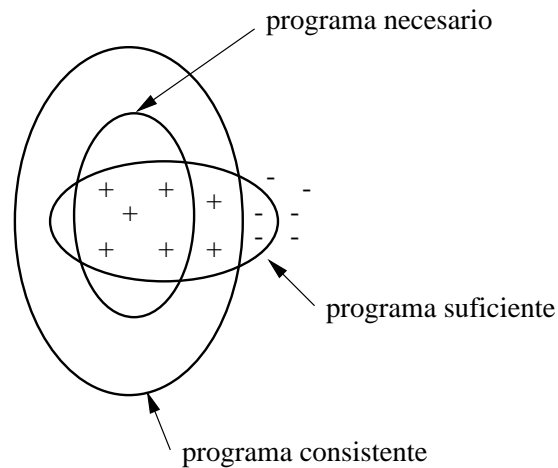
$H \leq H'$ y $H' \leq H''$ existe $H\theta \subseteq H'$ y $H'\theta \subseteq H''$ luego $H\theta \subseteq H''$

$H \leq H'$ y $H' \leq H$ dado $H\theta \subseteq H'$ y $H'\theta \subseteq H$

Las reglas de inferencia inductivas en general son **no** sanas y **no** completas [MR94], por lo tanto, los algoritmos asociados a los procesos inductivos no serán en general completos y los resultados que se obtengan pueden no ser sanos o la hipótesis H encontradas puede no satisfacer a E^+ , a esto hay que añadir que tanto la BC como el conjunto de entrenamiento E^+ pueden no contener suficiente información para verificar una hipótesis dada.

En el diseño de estos algoritmos se introduce en ocasiones además del conjunto E^+ , denominado conjunto de los ejemplos positivos, un conjunto E^- denominado conjunto de ejemplos negativos, tal que a partir de BC y H no sea deducible $\neg E^-$. La situación que se presenta con un algoritmo puede ser tal que alguna H seleccionada como candidato por el algoritmo puede satisfacer un subconjunto de E^- .

Se dice que un programa lógico P es suficiente si satisface todas las instancias positivas (E^+) y se dice que es necesario si no satisface ninguna instancia negativa (E^-). En el supuesto caso de que P pudiera satisfacer a ambas (o sea que fuese suficiente y necesario a la vez), se dice que P es consistente [Nil96].



Si un programa P es suficiente pero no necesario, se puede hacer que P implique menos ejemplos si se especializa, para ello se pueden sustituir variables por constantes, agregar más términos al cuerpo o

eliminar cláusulas. De modo inverso, si un programa P es necesario pero no suficiente, se puede hacer que P implique más ejemplos si se generaliza sustituyendo constantes por variables, eliminando términos del cuerpo o agregando nuevas cláusulas.

2.1 Ejemplo de razonamiento Inductivo

Se tratará de ilustrar este proceso con el siguiente ejemplo tomado de [MR94].

Imagine que una agente artificial está intentando aprender algunas relaciones familiares. Este agente sabe que un **abuelo** es el **papá** de algún "**progenitor**", pero aún no sabe lo que esto significa.

El agente conoce lo siguiente:

Una persona X es **abuelo** de una persona Y si algún X es **padre** de algún Z y Z es **progenitor** de Y .

Pedro es padre de María.

María es madre de Juan.

María es madre de Lupita.

Esto representado en Prolog queda de la siguiente forma:

$$BC = \begin{cases} \text{abuelo}(X,Y) :- \text{papa}(X,Z), \text{progenitor}(Z,Y). \\ \text{papa}(\text{pedro}, \text{maria}). \\ \text{mama}(\text{maria}, \text{juan}). \\ \text{mama}(\text{maria}, \text{lupita}). \end{cases}$$

Supóngase que ahora se le dan los siguientes hechos (ejemplos positivos) concernientes a ésta misma relación que estamos tratando que aprenda.

Pedro es abuelo de Juan.

Pedro es abuelo de Lupita.

Esto en Prolog puede ser representado como:

$$E^+ = \begin{cases} \text{abuelo}(\text{pedro}, \text{juan}). \\ \text{abuelo}(\text{pedro}, \text{lupita}). \end{cases}$$

También se dan algunos ejemplos que no son válidos para esta relación.

No es Juan abuelo de Pedro.

No es Lupita abuelo de Juan.

En Prolog esto se representa de la siguiente manera, donde :- representa el *NO ES*:

$$E^- = \begin{cases} \text{: - abuelo}(\text{juan}, \text{pedro}). \\ \text{: - abuelo}(\text{lupita}, \text{juan}). \end{cases}$$

Basado únicamente en el conocimiento previo *BC* y los ejemplos positivos E^+ y negativos E^- aquí presentados, puede advertirse la siguiente relación hipotética.

Una persona X es **progenitor** de una Y si X es **mama** de Y .

Que en Prolog se representa:

$$H = \{ \text{progenitor}(X,Y) \text{ :- mama}(X,Y) .$$

Esta H satisface todos los E^+ y a ningún E^- . Sin embargo, el agente no fue capaz de inducir que progenitor era cualquier **madre** o cualquier

padre debido a que los ejemplos presentados *no son* lo suficientemente representativos. Este es un problema ante el cual cualquier sistema que intenta aprender se tiene que enfrentar.

Informalmente podemos decir que los sistemas de PLI construyen cláusulas que definen un predicado a partir de un conjunto de entrenamiento E y un BC . Este tipo de sistemas no requiere de una BC completa y puede ser aplicado en dominios en donde el concepto a aprender no puede ser fácilmente descrito como un vector de entidades *atributo-valor* que es el método de representación usual en el AA clásico.

La PLI basada en la lógica clausal de la PL proporciona un formalismo de representación de conocimiento que es más expresivo que la representación *atributo-valor*, permitiendo no sólo el aprendizaje de conceptos definidos proposicionalmente sino también la generación de relaciones n -arias en general.

Desde el punto de vista de la Programación Lógica, la PLI abre una nueva y gran área al extender las capacidades deductivas fundamentales con técnicas para la síntesis inductiva de conocimiento.

2.2 Definición de PLI

La PLI yace en la intersección del Aprendizaje Inductivo y la Programación Lógica. Por ello la PLI emplea técnicas de ambas [MR94].

- La PLI además de superar el limitado formalismo de representación del AA facilita el empleo del conocimiento previo en el proceso de aprendizaje.
- De la Programación Lógica, la PLI toma su formalismo de representación del conocimiento, el cual brinda un elegante y poderoso mecanismo para representar en un mismo lenguaje y de forma declarativa el conjunto de ejemplos de entrenamiento, el conocimiento previo y las hipótesis que se buscan.

El contexto usual de la PLI es el siguiente: “Al agente que aprende se le da un conocimiento previo BC , ejemplos positivos E^+ y ejemplos negativos E^- y construye una hipótesis H ”. Donde BC , E^+ , E^- y H son programas lógicos [Mug94].

2.3 Requerimientos de un sistema de PLI

Las condiciones para construir H según [Mug94] y [MR94] son:

- Necesidad previa: $BC \not\models E^+$
Se refiere a que no debe ser posible deducir E^+ a partir de BC .
- Consistencia previa: $BC \wedge E^- \not\models \text{falsedad}$
Se refiere a que con la BC y los E^- no debe deducirse contradicción alguna (cláusula vacía).
- Suficiencia posterior: $BC \wedge H \models E^+$
 $BC \wedge \overline{E^+} \models \overline{H}$
Se refiere a que a partir de la BC y la H propuesta debe ser posible deducir E^+ .
- Consistencia débil: $BC \wedge H \not\models \text{falsedad}$
Se refiere a que con la BC y la H no debe deducirse contradicción alguna.
- Consistencia posterior: $BC \wedge H \wedge E^- \not\models \text{falsedad}$
Se refiere a que con la BC , la H y los E^- no debe ser posible deducir contradicción alguna.

2.4 Algoritmos de PLI Top-Down

Los algoritmos basados en este método, aprenden cláusulas buscando en el espacio de soluciones de lo general a lo específico de forma análoga

a como lo hacen los métodos tradicionales del Aprendizaje Automático tales como los árboles de decisión. El algoritmo más conocido de este método es FOIL de Quinlan el cual utiliza una heurística de ganancia de información para buscar las cláusulas en el espacio de soluciones.

FOIL aprende una cláusula a la vez utilizando el algoritmo que a continuación se bosqueja.

Inicialización:

hipótesis := programa nulo
resto := todos los ejemplos positivos

Mientras *resto* no este vacío

/* crear una nueva cláusula */
cláusula := R(A, B, ...) :-
 Mientras *cláusula* satisfaga ejemplos negativos
 /* Especialización de la cláusula */
 Buscar las literales *L* apropiadas
 Agregar *L* al cuerpo de *cláusula*
 Eliminar de *resto* los ejemplos positivos que sean satisfechos
cláusula
 Agregar *cláusula* a *hipótesis*

La creación de una cláusula nueva se lleva a cabo mediante un proceso iterativo de especialización partiendo con la cabeza de cláusula más general y agregando literales al cuerpo hasta que no satisfaga a ningún ejemplo negativo [Qui96].

La cabeza de la cláusula es siempre la literal que identifica la relación *R* que se quiere aprender y que contiene únicamente variables como argumentos. En cada paso, se evalúan las posibles literales que pueden ser adicionadas, estas deben contener a lo menos una variable que este

en la cabeza de la cláusula o en alguna literal previamente agregada al cuerpo y se escoge la que maximice la heurística de ganancia de información sobre conjunto de ejemplos, o sea la que satisfaga más ejemplos positivos y menos negativos [QCJ94].

Adicionalmente a lo aquí descrito, FOIL incluye algunas características adicionales como: heurísticas para la poda del espacio de búsqueda de literales, métodos para incluir igualdad y negación por fallo, así como mecanismos para la incorporación de literales que no dan a ganar información de forma inmediata (literales determinantes) y métodos para asegurar que el programa resultado termine [ZM97].

2.5 Algoritmos de PLI Bottom-Up

Estos sistemas parten de una hipótesis muy específica basándose en los ejemplos y la base de conocimientos e iterativamente intentan generalizarla aplicando reglas de inferencia inductiva. Durante este proceso deben cuidar que las hipótesis postuladas no implique lógicamente ejemplos negativos [MR94].

En la programación lógica, una cláusula general se utiliza para probar consecuencias específicas mediante el teorema de prueba denominado resolución. La inducción Bottom-Up invierte dicho proceso de resolución al derivar cláusulas generales a partir de consecuencias específicas. El efecto global de dicho proceso puede ser visto como la compresión de la definición de un concepto definido por las cláusulas que se derivan, debido a que se reemplazan muchas instancias específicas por estas cláusulas generales mediante las cuales dichas instancias pueden ser generadas [ZM97]. A continuación mencionamos algunos aspectos del funcionamiento de dos algoritmo de este tipo.

GOLEM (Muggleton y Feng) aprende a partir de ejemplos positivos tomando la menor generalización general (laest general generalization -lgg-) de las cláusulas, donde el átomo g es la generalización común de a y b si y solo si existe una substitución α'_g y β'_g tales que $a = g\alpha'_g$ y $b = g\beta'_g$. El átomo $lgg(a, b)$ es la menor generalización general de a y b si existe una substitución δ_g tal que $lgg(a, b) = g\delta_g$ [Mug95].

PROGOL (Muggleton) construye la cláusula más específica utilizando *Inverse Entailment* (Resolución Inversa) que se define: Dado una base de conocimiento BC y un conjunto de entrenamiento E , encontrar la hipótesis H más simple y consistente tal que $B \wedge H \models E$, donde H y E son Cláusulas de Horn [Mug95].

Durante la ejecución Progol sigue un procedimiento que puede ser descrito en 4 pasos:

1. **Elección de ejemplo.** Esto es, elegir un ejemplo para ser generalizado. Si no existen ejemplos termina, de lo contrario continuar.
2. **Construir la cláusula más específica.** Construir la cláusula más específica que implique lógicamente el ejemplo seleccionado según las restricciones de lenguaje proporcionadas. Usualmente ésta es una cláusula definida con muchas literales, y es llamada *cláusula de fondo*, este paso es llamado en ocasiones paso de saturación.
3. **Búsqueda.** Buscar la cláusula más general a partir de la cláusula de fondo. Esto se realiza buscando un subconjunto de literales con mejor puntuación en la cláusula de fondo, este paso puede ser llamado paso de reducción.
4. **Eliminar redundancia.** La cláusula con mejor puntuación es agregada a la base de conocimiento y todos los ejemplos que sean redundantes son eliminados. Regresar al paso 1.

El lenguaje de las hipótesis utilizado por PROGOL está restringido por la declaración de modos definidas por el usuario. La declaración de modos especifica si un átomo puede ser utilizado como cabeza de cláusula o como parte del cuerpo en la cláusula hipótesis. Para cada átomo, la declaración de modo indica el tipo de argumentos, si el argumento puede ser utilizado como variable de entrada, salida o con una constante.

PROGOL ofrece un serie de parametros para controlar el proceso de generalización, los cuales especifican la cardinalidad máxima de una hipótesis y el número máximo de variables nuevas, entre otras.

Capítulo 3

Algoritmo Inductivo Basado En Datos (*AIBED*)

A continuación se somete a consideración en el marco de la PLI el algoritmo *AIBED*, el cuenta con una estrategia de búsqueda Bottom-Up.

3.1 Objetivo

El objetivo del algoritmo es construir un programa lógico denominado Hipótesis (H) que pretende ser una explicación consistente del conjunto de entrenamiento compuesto por ejemplos positivos (E^+) y/o ejemplos negativos (E^-) los cuales son expresados en forma de hechos básicos en un programa lógico donde todos los átomos del conjunto de entrenamiento deben tener el mismo símbolo de predicado, la misma aridad y todos sus argumentos deben de ser constantes. Para la construcción de H además del conjunto de entrenamiento se considera un programa lógico denominado conocimiento previo o base de conocimiento (BC), donde dicho conocimiento además de ser necesario para la construcción de H enriquece su expresividad. Los términos de las cláusulas de BC y H pueden tener únicamente constantes y/o variables como argumentos (los números son interpretados como símbolos), no se permiten

cláusulas negados ni cláusulas recursivas. Así mismo es indispensable que la Base de Herbrand del conjunto de entrenamiento sea un subconjunto de la Base de Herbrand de BC .

En el mejor de los casos H será un conjunto de cláusulas nuevas que deben ser una compactación consistente de E^+ , si en H aparece algún elemento de E^+ se entiende que el algoritmo asume que dicho elemento no puede ser compactado y se toma como la explicación de si mismo. En el peor de los casos H será igual a E^+ .

3.2 Intuición

Ya que todos los elementos del conjunto de entrenamiento tiene el mismo símbolo de predicado para cada ejecución del algoritmo, entonces podemos ignorarlo y enfocarnos en las constantes individuales que aparecen en los elementos de E^+ , por ejemplo, si tomamos un elemento de E^+ que pudiera ser $f(b, c)$ lo que tenemos que hacer es encontrar en BC la cláusula o conjunción de cláusulas H que expliquen $\{b, c\}$.

Imaginemos ahora que BC contiene $\{p(a, b), p(b, d), p(b, e), p(a, c), h(X, Y) \leftarrow p(Y, X)\}$, si lo que buscamos en un H consistente para $\{b, c\}$ a partir de BC podemos construir algunos cuerpos de cláusula para $f(b, c)$ como $Cuerpo = p(a, b) \wedge p(a, c)$, ó $Cuerpo = p(a, b) \wedge h(c, a)$, incluso se pueden considerar cosas tan complicadas como $Cuerpo = p(b, d) \wedge p(a, c) \wedge p(b, d) \wedge p(a, b)$. Note que todos los *Cuerpos* aquí mostrados son una posible explicación consistente de $f(b, c)$ en base a BC . También se puede ver que algunos *Cuerpos* son más simples (o menos complejos) que otros. Se entiende que mientras menos cláusulas contenga un cuerpo, es más simple.

Por cuestiones prácticas se considera como el mejor *Cuerpo* aquel que resulte más simple, cuando exista más de un *Cuerpo* igualmente simple, como en este caso $\neg p(a, b) \wedge p(a, c)$ y $p(a, b) \wedge h(c, a)$ -, se consideran como igualmente buenos y se toma el primero en aparición.

Hasta aquí lo único que hemos encontrado es una conjunción de

cláusulas que explican el elemento tomado de E^+ . Una vez que hemos seleccionado el mejor *Cuerpo* construimos una hipótesis H la cual contendrá como cabeza de cláusula el elemento tomado de E^+ y como cuerpo de cláusula el *Cuerpo* seleccionado de donde obtenemos $H = f(b, c) \leftarrow p(a, b) \wedge p(a, c)$ y a continuación se generaliza la H que hemos construido haciendo una sustitución de constantes por variables manteniendo siempre una relación $\{c_1/v_1, c_2/v_2, \dots, c_m/v_m\}$ donde c_m son los distintos valores de las constantes individuales de cada elemento de E^+ y cada v_m es una variable nueva. De acuerdo a lo anterior, la relación de sustitución para este caso sería $\{b/X_1, c/X_2\}$, de modo que la cabeza de H queda únicamente con variables, obteniendo $H = f(X_1, X_2) \leftarrow p(a, X_1) \wedge p(a, X_2)$, a continuación verificamos que esta cláusula generalizada **no** implique lógicamente ningún elemento de E^- (si es que este último fue definido) de ser el caso podemos esperar que esta misma cláusula nos sirva para explicar a más de un elemento de E^+ . De no ser el caso, entonces dejamos a H sin generalizar.

Para este ejercicio consideraremos que efectivamente la H generalizada no implica ningún elemento de E^- , entonces lo que hacemos es eliminar todos los elementos de E^+ que sean implicados a partir de dicha H y se guarda la H en el conjunto resultado R .

Si E^+ no queda vacío tomamos el elemento en turno que en esta ocasión es $f(d, e)$, al realizar el mismo proceso que en caso anterior concluimos que $H = f(d, e) \leftarrow p(b, d) \wedge p(b, e)$ y al igual se generaliza y valida obteniendo que $H = f(X_1, X_2) \leftarrow p(b, X_1) \wedge p(b, X_2)$ y guardamos en R .

$$R = \left\{ \begin{array}{l} f(X_1, X_2) \leftarrow p(a, X_1) \wedge p(a, X_2) \\ f(X_1, X_2) \leftarrow p(b, X_1) \wedge p(b, X_2) \end{array} \right\}$$

Una vez que tenemos más de una cláusula en R es factible analizarlas para ver si podemos compactarlas encontrando una nueva cláusula que subsuma mediante una sustitución de constantes por variables a más de una. En este caso es factible encontrar una cláusula que subsuma a las dos ya creadas, esta es $f(X_1, X_2) \leftarrow p(X_3, X_1) \wedge p(X_3, X_2)$, validamos que esta no implique ningún elemento de E^- y en tal caso podemos

postular que esta nueva cláusula es la explicación de ambos ejemplos. En el caso contrario no se hace nada, esto es, se deja R como esta.

Es importante destacar que el ejemplo aquí presentado está premeditadamente construido para mostrar el razonamiento de *AIBED*. Como ya se menciono anteriormente, en el peor de los casos *AIBED* dará como resultado el conjunto E^+ cuando no sea capaz de explicar ningún elemento de E^+ a partir de BC . En el caso de que pueda explicarlos pero no compactar R obtendremos como resultado el conjunto E^+ donde cada elemento tendrá un cuerpo que explique a dicho elemento respecto a BC .

Cada elemento de E^+ que va siendo analizado por el algoritmo es eliminado de E^+ , si se encuentra una cláusula que lo explique esta es añadida a R , de no ser el caso se guarda el elemento analizado en R asumiendo que se trata de una excepción. De esta forma, una vez que E^+ queda vacía el algoritmo termina y obtenemos en R el conjunto de cláusulas solución.

3.2.1 Elementos para la ejecución

Para la ejecución de *AIBED* se requiere de los siguientes archivos:

- Un `*.pl` que contenga las cláusulas del conocimiento previo. Los términos de dichas cláusulas únicamente pueden tener constantes y/o variables como argumentos, en el caso de contar con números estos serán interpretados como símbolos. No se permiten términos negados ni cláusulas recursivas.
- Un `*.pos` que contenga el conjunto de ejemplos positivos. Los cuales son expresados en forma de hechos básicos en un programa lógico y todos sus átomos deben tener el mismo símbolo de predicado, la misma aridad y todos sus argumentos deben de ser constantes.
- Opcionalmente un archivo `*.neg` que debe contener el conjunto de ejemplos negativos expresado teniendo las mismas restricciones que los ejemplos positivos.

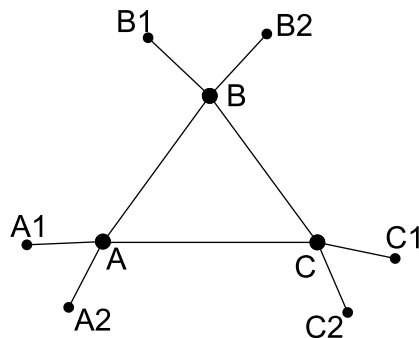
Nota. Es indispensable que la Base de Herbrand del conjunto de entrenamiento sea un subconjunto de la Base de Herbrand de BC .

Los archivos que pertenezcan a una misma entrada o problema deben tener el mismo nombre de archivo con las diferentes extensiones, de esta forma es como los identifica *AIBED*.

3.2.2 Ejemplo de archivos de entrada

A continuación se muestra un ejemplo (tomado de [Nil96]) con la sintaxis que debe tener un archivo que sirve como entrada a este sistema:

El ejemplo trata el problema de una línea aérea, para ello se considera una porción del mapa de rutas en donde las ciudades A , B y C son ciudades centrales, A_1 , A_2 , B_1 , B_2 , C_1 y C_2 son ciudades satélite respectivas a cada ciudad central tal y como se ilustra a continuación.



Los archivos aquí mostrados son la representación del problema de las líneas aéreas en forma de programa lógico que sirven como entrada a la implementación de *AIBED*.

```

% fly.pl - Ejemplo tomado del libro de Nilsson
% conocimiento previo

```

```

hub(a).
hub(b).

```

```
hub(c).

satellite(a1,a).
satellite(a2,a).
satellite(b1,b).
satellite(b2,b).
satellite(c1,c).
satellite(c2,c).

% fly.pos - ejemplos positivos

nonstop(a,b).
nonstop(b1,b).
nonstop(c,c2).
nonstop(c,b).
nonstop(a2,a).
nonstop(b,b2).

% fly.neg - ejemplos negativos

nonstop(a1,a2).
nonstop(a,b1).
nonstop(c2,b).
nonstop(b2,b1).
nonstop(b,a1).
```

3.3 Descripción de *AIBED*

Dado un programa lógico BC que sirve como base de conocimientos donde, como ya se mencionó, sus términos pueden tener únicamente constantes y/o variables como argumentos (los números son interpretados como símbolos) y no se permiten términos negados ni cláusulas recursivas. Dado un programa lógico E^+ el cual puede contener únicamente hechos básicos con constantes como argumentos, y opcionalmente dado un programa lógico E^- estructurado igual a E^+ , en donde el primero representa el conjunto de ejemplos positivos y el segundo el conjunto

de ejemplos negativos, y partiendo de la premisa de que el Universo de Herbrand de E^+ y E^- son un subconjunto del Universo de Herbrand de BC , donde $|x|$ denota el número de elementos del conjunto x :

3.4 Ciclo principal

1. **Tomar e_1 de E^+ .** Donde e_1 es el primer elemento que aparece en E^+ .
2. **Crear C y D a partir de e_1 .** Donde C es el conjunto de todas las constantes c_i que aparecen como argumentos en e_1 y D es la sustitución $[c_1/v_1, c_2/v_2, \dots, c_m/v_m]$, donde las v_i son variables (para $1 \leq i \leq m$).
3. **Construir C_p a partir de C y BC .** Donde C_p es un subconjunto $[a_1, a_2, \dots, a_n]$ del Modelo Minimal de Herbrand de BC , cuyos átomos básicos a_j (para $1 \leq j \leq n$) contienen como argumento al menos una constante de C .
4. **Construir una lista de cuerpos de cláusula LC a partir de C y C_p .** Donde LC es un conjunto de cuerpos $[cuerpo_1, cuerpo_2, \dots, cuerpo_p]$ y cada $cuerpo_k$ (para $1 \leq k \leq p$) es una conjunción de átomos básicos $[a_1, a_2, \dots, a_q]$ tomados de C_p y acotados por un valor de cardinalidad (por default es 10) considerando las restricciones que a continuación se mencionan.

$$LC = \{\}$$

mientras $C_p \neq \{\}$

(a) *atomo* = mejor átomo básico a_j de C_p

(b) Eliminar *atomo* de C_p

(c) $C_{ciclo} = C$

(d) $cuerpo_k = \{\}$

mientras $C_{ciclo} \neq \{\}$ y valor de cardinalidad $> |cuerpo_k|$

i. Agregar *atomo* a $cuerpo_k$

- ii. C_{atomo} = el conjunto de constantes que aparecen como argumentos en *atomo*
- iii. $NC = C_{ciclo} - C_{atomo}$
- iv. $NC_{atomo} = C_{atomo} - C_{ciclo}$
- v. $C_{ciclo} = NC \cup NC_{atomo}$
- si** $C_{ciclo} \neq \{\}$ **entonces**
 - A. construir Cp_{ciclo} a partir de C_{ciclo} y BC
 - B. *atomo* = mejor átomo básico a_j de Cp_{ciclo} que no esta en *cuerpo_k*

fin_si

fin_mientras

- (e) guarda *cuerpo_k* en LC

fin_mientras

5. **Tomar el mejor cuerpo MC de LC .** Tomar el mejor cuerpo y eleminarlo de LC . Si LC está vacío entonces $MC = \{\}$.
6. **Construir una cláusula H y guardarla en R .** Donde H es una cláusula cuya cabeza es e_1 y cuyo cuerpo es MC y R es el conjunto solución. Antes de guardar H , esta se generaliza haciendo la sustituciones de constantes según D y se verifica que H no implique lógicamente ningún elemento de E^- en BC , en tal caso se eliminan todos los elementos de E^+ que resulten implicados lógicamente y se guarda la H generalizada, de lo contrario guardar H sin generalizar.
7. **Si R cuenta con más de dos elementos “compactar” R .** Donde:

$$\begin{aligned} H_x &= \text{Cabeza}_x \text{ :- } \text{Cuerpo}_x \\ H_{actual} &= \text{Cabeza}_{actual} \text{ :- } \text{Cuerpo}_{actual} \end{aligned}$$

Caso 1 $\text{Cabeza}_x = \text{Cabeza}_{actual}$ y $\text{Cuerpo}_x \neq \text{Cuerpo}_{actual}$ únicamente en constantes, p.e.

$$H_x = f(A, B) :- p(a, B), p(a, A). \\ H_{actual} = f(A, B) :- p(b, B), p(b, A).$$

en tal caso, buscar una cláusula que subsuma a ambas (por medio de una sustitución de constantes por variables) y verificar que dicha cláusula no implique lógicamente ningún elemento de E^- , si este fue definido. De ser el caso, eliminar de R todas las cláusulas subsumidas y guardar la nueva cláusula genérica en R . De lo contrario continua.

Caso 2 Cabeza $_x \neq$ Cabeza $_{actual}$ únicamente en constantes y Cuerpo $_x =$ Cuerpo $_{actual}$, p.e.

$$H_x = f(A, a) :- p(c, A). \\ H_{actual} = f(A, b) :- p(c, A).$$

en tal caso, cambiar el cuerpo de H_{actual} por el mejor cuerpo que se encuentre en LC y para H_x recalculer su LC y tomar el mejor cuerpo que no sea el anteriormente seleccionado. Para ambos casos generalizar y verificar que no impliquen lógicamente ningún elemento de E^- , de no ser el caso, buscar un cuerpo que satisfaga esta condición, de no encontrar ninguno, se dejan como estaban.

8. Si $E^+ \neq \{\}$ regresar al paso 1, de lo contrario parar.

3.4.1 Determinación del Mejor Átomo

Para poder determinar cual es el mejor átomo de Cp es necesario asignarle a cada uno de ellos una métrica de bondad que en nuestro caso denominamos $peso_j$ el cual se asocia a cada a_j de Cp al momento de construirlo. El cálculo de dicho $peso_j$ esta basado en la siguiente heurística:

Considerar para cada átomo a_j : C_{a_j} =conjunto de constantes que aparecen como argumentos de a_j ; $aridad = |C_{a_j}|$; $valores = |C|$ y $cubiertas = |C \cap C_{a_j}|$.

$$p_i = \frac{\text{cubiertas}}{\text{valores}} * \frac{\text{cubiertas}}{\text{aridad}}$$

Si estamos buscando el mejor átomo para explicar $C = \{x, y, z\}$ nos podemos encontrar con muy diversos candidatos que contenga al menos uno de los valores que buscamos, como por ejemplo:

Átomo	peso	Ordenando →	Átomo	peso
p(x)	0.33		p(z,x,y)	1.00
p(a,x,y)	0.44		p(x,a,y,z)	0.75
p(a,b,z)	0.11		p(x,z)	0.67
p(x,z)	0.67		p(x,a,y,b,z)	0.60
p(x,a,y,z)	0.75		p(a,x,y)	0.44
p(z,x,y)	1.00		p(x)	0.33
p(x,a,y,b,z)	0.60		p(a,b,z)	0.11

La idea es seleccionar el átomo que contenga más elementos de los que buscamos y menos elementos nuevos, esto nos permite construir *cuerpos_k* más simples. En caso de que *AIBED* se encuentre con dos o más átomos con el mismo *peso_j* se considera que son igualmente buenos y se toma el primero en aparición.

3.4.2 Determinación del Mejor Cuerpo

Para obtener el valor de optimización val_k asociado a cada *cuerpo_k* considerar: C_{cuerpo} es el conjunto de todas las constantes que ocurren como argumento en los átomos básicos a_l (donde $1 \leq l \leq q$) del *cuerpo_k*; $cub = |C - C_{cuerpo}|$; $adic = |C_{cuerpo} - C|$ y $long = |C_{cuerpo}| \cup |cuerpo_k|$.

$$val_i = \left[\frac{cub}{|C|} * \frac{5}{10} \right] + \left[\frac{1}{(adic + 1)} * \frac{4}{10} \right] + \left[\frac{1}{long} * \frac{1}{10} \right]$$

Lo que se busca con esta heurística es ordenar los cuerpos en base a los siguientes factores: 1) el número de elementos que cubre, es por ello que tiene $\frac{5}{10}$ asociados a este factor, 2) el número de constantes adicionales que utiliza, a mayor número de adicionales menor puntuación y a este factor se le asocia $\frac{4}{10}$ del valor total y 3) considerado como el

más débil y asociado a $\frac{1}{10}$ del total, es la longitud del *cuerpo_k*, donde a mayor longitud menor puntuación.

A continuación se somete a consideración dicha heurística presentado un ejemplo analizado por *AIBED*.

- Ejemplo positivo analizado: $e_i = \mathbf{tio}(\mathbf{b}, \mathbf{f})$
- Lista de cuerpos de cláusulas *LC* construidos:
 - [padre(a,b),padre(a,c),padre(c,f)]/0.911111,
 - [padre(b,d),padre(c,f),padre(a,c),padre(a,b),hijo(d,b)]/0.906667,
 - [padre(b,e),padre(c,f),padre(a,c),padre(a,b),hijo(e,b)]/0.906667,
 - [padre(c,f),hermano(b,c)]/0.916667,
 - [padre(f,l),padre(a,b),padre(a,c),padre(c,f),hijo(l,f)]/0.906667,
 - [padre(f,m),padre(a,b),padre(a,c),padre(c,f),hijo(m,f)]/0.906667,
 - [hijo(d,b),padre(b,d),padre(a,b),padre(a,c),padre(c,f)]/0.906667,
 - [hijo(e,b),padre(b,e),padre(a,b),padre(a,c),padre(c,f)]/0.906667,
 - [hijo(f,c),hermano(b,c)]/0.916667,
 - [hijo(l,f),padre(a,b),padre(a,c),padre(c,f),padre(f,l)]/0.906667,
 - [hijo(m,f),padre(a,b),padre(a,c),padre(c,f),padre(f,m)]/0.906667,
 - [hermano(b,c),padre(c,f)]/0.916667,
 - [hermano(c,b),padre(c,f)]/0.916667,
 - [hermano(f,g),padre(a,b),padre(a,c),padre(c,g)]/0.908333,
 - [hermano(g,f),padre(a,b),padre(a,c),padre(c,g)]/0.908333,
- *LC* ordenados según el valor de optimización val_k asociado a ellos:

```

[padre(c,f),hermano(b,c)]/0.916667,
[hijo(f,c),hermano(b,c)]/0.916667,
[hermano(b,c),padre(c,f)]/0.916667,
[hermano(c,b),padre(c,f)]/0.916667,
[padre(a,b),padre(a,c),padre(c,f)]/0.911111,
[hermano(f,g),padre(a,b),padre(a,c),padre(c,g)]/0.908333,
[hermano(g,f),padre(a,b),padre(a,c),padre(c,g)]/0.908333,
[padre(b,d),padre(c,f),padre(a,c),padre(a,b),hijo(d,b)]/0.906667,
[padre(b,e),padre(c,f),padre(a,c),padre(a,b),hijo(e,b)]/0.906667,
[padre(f,l),padre(a,b),padre(a,c),padre(c,f),hijo(l,f)]/0.906667,
[padre(f,m),padre(a,b),padre(a,c),padre(c,f),hijo(m,f)]/0.906667,
[hijo(d,b),padre(b,d),padre(a,b),padre(a,c),padre(c,f)]/0.906667,
[hijo(e,b),padre(b,e),padre(a,b),padre(a,c),padre(c,f)]/0.906667,
[hijo(l,f),padre(a,b),padre(a,c),padre(c,f),padre(f,l)]/0.906667,
[hijo(m,f),padre(a,b),padre(a,c),padre(c,f),padre(f,m)]/0.906667]

```

3.4.3 Ejemplo ejecutado

El ejemplo de las líneas aéreas presentado anteriormente y procesado por *AIBED* nos da como resultados:

```

aibed> load('Fly/fly').
consulting(Ejemplos/Fly/fly.pl)
consulted(Ejemplos/Fly/fly.pl)
time(consulting = 0,quick_compiling = 0,static_space = 0)
consulting(Ejemplos/Fly/fly.pos)
consulted(Ejemplos/Fly/fly.pos)
time(consulting = 0,quick_compiling = 0,static_space = 0)
Validacion de Necesidad Previa SATISFACTORIA
consulting(Ejemplos/Fly/fly.neg)
consulted(Ejemplos/Fly/fly.neg)
time(consulting = 0,quick_compiling = 0,static_space = 0)
aibed> learn.
ei=nonstop(a,b)
    H=nonstop(_x5846,_x5847) :- hub(_x5846),hub(_x5847)
ei=nonstop(b1,b)
    H=nonstop(_x5846,_x5847) :- satellite(_x5846,_x5847)
ei=nonstop(c,c2)

```

```
H=nonstop(_x5846,_x5847) :- satellite(_x5847,_x5846)
```

Hipotesis formuladas

```
% nonstop/2:
```

```
nonstop(A,B) :-
```

```
    hub(A),
```

```
    hub(B).
```

```
nonstop(A,B) :-
```

```
    satellite(A,B).
```

```
nonstop(B,A) :-
```

```
    satellite(A,B).
```

```
aibed>
```

Capítulo 4

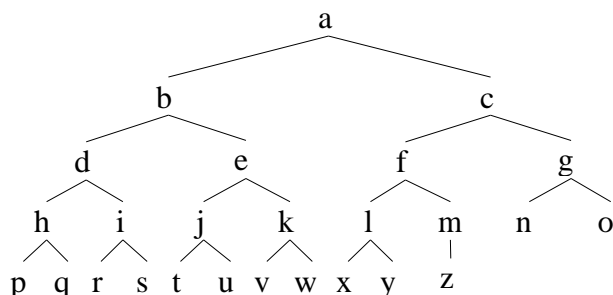
Pruebas

A consinuación se describen algunos resultados obtenidos al presentarle diferentes problemas a *AIBED*, principalmente se presentan pruebas con el problema de un árbol familiar, se escogió dicho problema por ser sumamente conocido y presentar una amplia variedad de situaciones didacticas que nos son útiles para ilustrar el comportamiento de *AIBED*.

Con el objeto de analizar de una forma más minuciosa el comportamiento del algoritmo se agregaron unos parametros en la implementación, estos son, `generaliza/1` que puede tomar los valores de `yes/no`, esto le indica al algoritmo cuando intentar hacer una generalización al momento de compactar el conjunto de hipótesis resultado y `profundidad/1` el cual le indica al algoritmo el número máximo de elementos del Modelo Minimal que deben ser considerados en el proceso de construcción de hipótesis. El empleo de este parametro es un tanto delicado, ya que tiene repercusiones importantes en los resultados obtenido, por lo cual se deja para la parte final de este análisis.

4.1 Problema del Árbol Familiar

Las pruebas se basan en el árbol presentado a continuación:



Este árbol es representado en forma de programa lógico utilizando las siguientes cláusulas en las cuales aparecen dos nombres de predicado `padre/2` e `hijo/2`, este programa junto con el conjunto de entrenamiento sirve como entrada a *AIBED*.

```

% padre/2:
padre(a,b).
padre(a,c).
padre(b,d).
padre(b,e).
padre(c,f).
padre(c,g).
padre(d,h).
padre(d,i).
padre(e,j).
padre(e,k).
padre(f,l).
padre(f,m).
padre(g,n).
padre(g,o).
padre(h,p).
padre(h,q).
padre(i,r).
padre(i,s).
padre(j,t).
padre(j,u).
padre(k,v).
padre(k,w).
padre(l,x).

```

```
padre(l,y).
padre(m,z).

% hijo/2:
hijo(B,A) :-
    padre(A,B).
```

La idea del ejercicio es ver como *AIBED* puede ir construyendo nuevas cláusulas y las puede ir incorporando a la base de conocimientos utilizando los comandos `incluye_hipo/0` y `save/2` para tener un aprendizaje incremental. En este capítulo presentaremos ejemplos de `hermano`, `tio`, `primo`, y `ancestro`. Para cada ejercicio se discutirán los resultados según algunas variantes como pueden ser el evitar que *AIBED* intente hacer la compactación de las hipótesis mediante la generalización de las mismas. Además se ponen en evidencia algunas deficiencias de la implementación del algoritmo.

4.1.1 Aprendiendo la relación hermano

Primeramente presentaremos a *AIBED* el ejercicio de `hermano`, para ello consideramos el siguiente conjunto de entrenamiento:

```
% Ejemplos positivos de la relacion hermano

%hermano/2:
hermano(b,c).
hermano(d,e).
hermano(f,g).
hermano(h,i).
hermano(j,k).
hermano(l,m).
hermano(n,o).
hermano(p,q).
hermano(r,s).
hermano(t,u).
hermano(v,w).
hermano(x,y).
```

En este caso se presenta un conjunto de E^+ completo en un sentido (lo cual no es necesario), esto es, se están omitiendo todos los ejemplos simétricos de esta relación p.e. `hermano(c,b)`, `hermano(e,d)`, etc.

```
% Ejemplos negativos de la relacion hermano
```

```
% hermano/2:
hermano(a,b).
hermano(e,f).
hermano(h,o).
```

Los ejemplos negativos son solo una pequeña muestra aleatoria, pues carecen de importancia en este caso para el ejercicio. Los resultados obtenidos, considerando el parametro `generaliza(yes)` son:

```
Hipotesis formuladas
```

```
% hermano/2:
hermano(A,C) :-
    padre(B,A),
    padre(B,C).
```

En la solución no aparece la relación `hijo` porque como ya se menciono en el capítulo anterior cuando *AIBED* se encuentra con dos o más átomos igualmente buenos al construir un cuerpo, considera el primero en aparición.

Este mismo resultado es el que se obtiene cuando agregamos a E^+ alguno o todos los ejemplos simétricos de dicha relación, sin embargo, al ejecutarlo de nuevo pero con el parametro `generaliza(no)`, obtenemos:

```
Hipotesis formuladas
```

```
% hermano/2:
hermano(A,B) :-
    padre(a,A),
    padre(a,B).
hermano(A,B) :-
    padre(b,A),
```

```

        padre(b,B) .
hermano(A,B) :-
        padre(c,A) ,
        padre(c,B) .
hermano(A,B) :-
        padre(d,A) ,
        padre(d,B) .
hermano(A,B) :-
        padre(e,A) ,
        padre(e,B) .
hermano(A,B) :-
        padre(f,A) ,
        padre(f,B) .
hermano(A,B) :-
        padre(g,A) ,
        padre(g,B) .
hermano(A,B) :-
        padre(h,A) ,
        padre(h,B) .
hermano(A,B) :-
        padre(i,A) ,
        padre(i,B) .
hermano(A,B) :-
        padre(j,A) ,
        padre(j,B) .
hermano(A,B) :-
        padre(k,A) ,
        padre(k,B) .
hermano(A,B) :-
        padre(l,A) ,
        padre(l,B) .

```

En este punto, resulta claro que al no compactar el conjunto de las hipótesis generadas –con el parametro *generaliza(no)*- lo que se obtiene es únicamente la explicación más simple para cada uno de los elementos de E^+ . Al presentarle los ejemplos simétricos con esta opción se obtienen los mismos resultados, ya que al explicar el primer ejemplo p.e. *hermano(b,c)* *AIBED* elimina de E^+ todos los elementos que sean implicados lógicamente por dicha hipótesis, que en este caso

sería `hermano(c,b)`. Algo similar ocurre si *AIBED* encontrara estos hechos en orden de aparición inverso, la diferencia radica en que la cláusula en vez de ser: `hermano(A,B) :- padre(b,A), padre(b,B) .`, sería: `hermano(B,A) :- padre(b,A), padre(b,B) .`

Ahora analizamos el comportamiento de *AIBED* con un nuevo conjunto de E^- . La única modificación con respecto al E^- anterior es la adición de un hecho que indica que un hermano no puede ser hermano de si mismo (`hermano(b,b)`).

```
% hermano/2:
hermano(a,b).
hermano(b,b).
hermano(e,f).
hermano(h,o).
```

Esta simple adición impide que *AIBED* realice la compactación de las hipótesis –aun y cuando este generaliza(yes)-, esto se debe a que la implementación **no** es capaz de generar cláusulas negativas para especificar que $A \neq C$ por lo que se obtiene como resultado el mismo conjunto solución que en caso anterior. La cláusula esperada y que la implementación de *AIBED* no es capaz de construir es:

```
Hipotesis formuladas
% hermano/2:
hermano(A,C) :-
    padre(B,A),
    padre(B,C),
    A \== C.
```

4.1.2 Aprendiendo la relación tío

En esta ocasión presentamos el comportamiento de *AIBED* con un conjunto de entrenamiento no completo de la relación `tío`. El conjunto de entrenamiento está compuesto únicamente por ejemplos positivos, los cuales son una muestra aleatoria que resulta lo suficientemente representativo para que *AIBED* logre aprender dicha relación.

```
% conjunto de ejemplos positivos de la relacion tio

% tio/2:
tio(b,f).
tio(c,d).
tio(d,j).
tio(e,h).
tio(f,n).
tio(g,l).
tio(h,r).
tio(j,w).
tio(l,z).
```

Los resultados obtenidos utilizando el parametro `generaliza(yes)` son los siguientes:

```
Hipotesis formuladas
% tio/2:
tio(B,A) :-
    padre(C,A),
    hermano(B,C).
```

Este ejemplo deja en claro que *AIBED* es capaz de trabajar eficazmente con un conjunto de entrenamiento no completo que resulte representativo, más adelante se verá un caso donde aparentemente no es capaz de aprender correctamente debido a la falta de representación del conjunto de entrenamiento.

4.1.3 Aprendiendo la relación primo

Este ejemplo nos sirve para ilustrar que *AIBED* es capaz de utilizar adecuadamente los predicados que se han ido construyendo para generar las hipótesis más simples. En esta oportunidad presentamos el siguiente conjunto de E^+ y omitimos el conjunto de E^- .

```
% primo/2:
```

```

primo(d,f).
primo(d,g).
primo(e,f).
primo(e,g).
primo(h,j).
primo(h,k).
primo(i,j).
primo(i,k).
primo(l,n).
primo(l,o).
primo(m,n).
primo(m,o).
primo(p,r).
primo(p,s).

```

Los resultados obtenidos son:

```

Hipotesis formuladas
% primo/2:
primo(A,C) :-
    padre(B,A),
    tio(B,C).

```

Hasta este momento todos los ejemplos se han realizando con el parámetro de `profundidad(30)`, este parámetro nos indica cuantos elementos del Modelo Minimal de Herbran pueden ser considerados en cada paso de inducción, si este parámetro es muy pequeño se corre el riesgo de que no se puedan construir cláusulas lo suficientemente simples, si por ejemplo hacemos que *AIBED* dispare de nuevo el proceso de aprendizaje para el mismo ejemplo, pero con `profundidad(10)` nos arroja los siguientes resultados:

```

% primo/2:
primo(A,B) :-
    padre(b,A),
    padre(a,b),
    padre(a,c),

```

```

        padre(c,B) .
primo(A,B) :-
    padre(d,A) ,
    padre(b,d) ,
    padre(a,b) ,
    padre(a,c) ,
    padre(c,f) ,
    padre(e,B) .
primo(A,B) :-
    padre(f,A) ,
    padre(c,f) ,
    padre(a,c) ,
    padre(a,b) ,
    padre(b,d) ,
    padre(d,h) ,
    padre(g,B) .
primo(A,B) :-
    padre(h,A) ,
    padre(d,h) ,
    padre(b,d) ,
    padre(a,b) ,
    padre(a,c) ,
    padre(c,f) ,
    padre(f,l) ,
    padre(i,B) .

```

Los resultado obtenidos son poco satisfactorio, pero esto se debe a que forzamos a *AIBED* a realizar un análisis muy superficial y *AIBED* no logra encontrar explicaciones concisas. En el siguiente ejemplo se explica el porque se este comportamiento. Queda claro además, que mientras menor es el valor de *profundidad/1* más rápido será el análisis.

4.1.4 Aprendiendo la relación ancestro

Este ejercicio resulta interesante por dos aspectos fundamentales, 1) Muestra como *AIBED* puede aprender parcialmente una relación debido a la falta de información en el conjunto de ejemplos positivos y 2)

Muestra el patrón de solución que se puede alcanzar con una cláusula que en esencia es recursiva. Como ya se menciono, ésta implementación de *AIBED* no es capaz de generar hipótesis recursivas, sin embargo, este ejemplo nos servirá de referencia para bosquejar cómo sería una posible adición para hacer tratables estos casos por el algoritmo, dicho desarrollo se realiza en el siguiente capítulo.

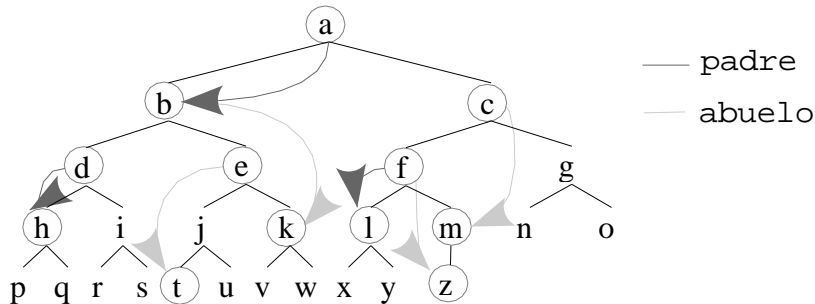
Considerar el siguiente conjunto de ejemplos positivos:

```
ancestro(a,b).
ancestro(b,k).
ancestro(c,m).
ancestro(d,h).
ancestro(e,t).
ancestro(f,l).
ancestro(f,z).
```

Los resultados obtenidos con una profundidad(30) son los siguientes:

```
Hipotesis formuladas
% ancestro/2:
ancestro(A,B) :-
    padre(A,B).
ancestro(A,C) :-
    padre(A,B),
    padre(B,C).
```

Esta solución resulta aceptable considerando que los ejemplos presentados muestran únicamente la relación de padre y abuelo como se ilustra a continuación.



Si dejamos que *AIBED* realice un análisis más exhaustivo con profundidad(50), lo que obtenemos es:

```
Hipotesis formuladas
% ancestro/2:
ancestro(A,B) :-
    padre(A,B).
ancestro(A,B) :-
    abuelo(A,B).
```

Estos resultados son igualmente consistentes a los anteriores, solo que en este caso logró simplificar más la segunda cláusula al especificar *abuelo/2* en lugar de *padre/2*, *padre/2*. Esto es debido a que dada la forma en la que están organizadas tanto la *BC* como los E^+ el predicado *abuelo/2* necesario para simplificar la hipótesis no aparece en los primeros 30 términos de *Cp*. Al ampliar la profundidad a 50 le estamos dando la oportunidad de aparecer, por lo cual es considerado y aplicado en la segunda solución que presentamos.

Ahora consideremos un conjunto de entrenamiento que es lo suficientemente representativo como para generar un concepto más amplio que en el caso anterior.

```
% ancestro/2:
ancestro(a,b).
ancestro(a,d).
ancestro(a,h).
ancestro(a,l).
ancestro(a,p).
ancestro(a,t).
ancestro(a,x).
```

Resultados obtenidos con profundidad(10).

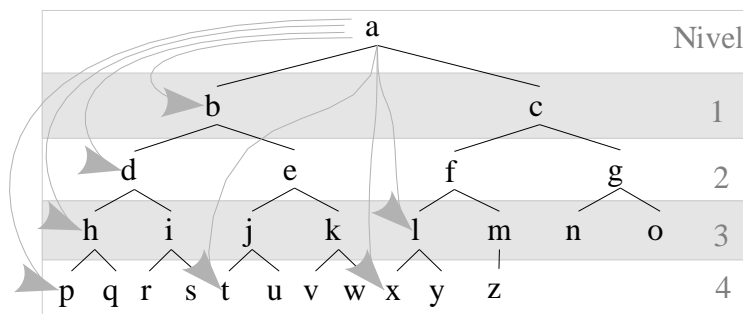
```
Hipotesis formuladas
% ancestro/2:
a10. ancestro(A,B) :-
```

```

        padre(A,b),
        padre(b,B).
b10. ancestro(A,B) :-
        padre(A,B).
c10. ancestro(A,D) :-
        padre(A,B),
        padre(B,C),
        padre(C,D).
d10. ancestro(A,E) :-
        padre(A,B),
        padre(B,C),
        padre(C,D),
        padre(D,E).
    
```

Los resultados aquí mostrados son una explicación consistente del conjunto de entrenamiento, además en todas las cláusulas construidas excepto en la primera (a10) no aparecen constantes.

El hecho por el cual aparece la constante **b** en **a10** es porque sólo existe un único ejemplo que explica el segundo nivel del árbol como se muestra a continuación. En este mismo esquema se ve que en el caso del primer nivel la generalización es completa para **b10**, esto se debe a que la relación **padre/2** explica con una sola cláusula a **ancestro/2** para tal ejemplo.



Resultados obtenidos con profundidad(30).

Hipotesis formuladas

```

% ancestro/2:
a30. ancestro(A,B) :-
    padre(A,c),
    padre(c,f),
    padre(f,B).
b30. ancestro(B,A) :-
    padre(d,A),
    abuelo(B,d).
c30. ancestro(A,B) :-
    abuelo(A,B).
d30. ancestro(A,B) :-
    padre(A,B).
e30. ancestro(A,E) :-
    padre(A,B),
    padre(B,C),
    padre(C,D),
    padre(D,E).

```

Al aumentar la profundidad(30) para forzar a *AIBED* a realizar un análisis más exhaustivo que el caso anterior, vemos que comienza a aparecer la relación de *abuelo/2* en algunas cláusulas, esto se debe a la forma en la cual estan estructurados los datos de *BC*, ya que cuando se construye *Cp* (paso 3 del ciclo ppal) realmente no se están considerando todos los átomos básicos de *BC* que contengan a lo menos una constante de *C*, sino que, realmente se esta acotando el tamaño de *Cp* por el parámetro *profundidad/1*. Además, a diferencia del conjunto solución anterior, este nuevo conjunto cuenta con una cláusula adicional (4 en el caso anterior y 5 en este caso) donde se observan las siguientes equivalencias lógicas: $a_{10} = c_{20}$, $b_{10} = d_{20}$, $c_{10} = b_{20} + a_{20}$, y $d_{10} = e_{20}$.

Resultados obtenidos con profundidad(50).

```

Hipotesis formuladas
% ancestro/2:
ancestro(A,B) :-
    abuelo(A,B).
ancestro(A,B) :-

```

```

        padre(A,B) .
ancestro(A,C) :-
        padre(A,B) ,
        abuelo(B,C) .
ancestro(A,D) :-
        padre(A,B) ,
        padre(B,C) ,
        abuelo(C,D) .

```

Resultados obtenidos con profundidad(100).

```

Hipotesis formuladas
% ancestro/2:
ancestro(A,C) :-
        padre(A,B) ,
        abuelo(B,C) .
ancestro(A,B) :-
        padre(A,B) .
ancestro(A,B) :-
        abuelo(A,B) .
ancestro(A,C) :-
        abuelo(A,B) ,
        abuelo(B,C) .

```

Como podemos observar, mientras más grande es el valor de `profundidad/1` más compacto se vuelve el conjunto solución. En el capítulo siguiente se expondrá una posible solución para que los resultados no resulten tan dependientes del parametro `profundidad` manteniendo un límite para acotar con ello el espacio de búsqueda.

4.2 Ejemplo de Animales

El siguiente ejemplo referente a la clasificación de animales por especie fue tomado de Progol4.2 de Muggleton para probarlo con *AIBED*. En este caso se presentan el programa lógico que sirve como base de

conocimiento, y el conjunto de ejemplos tanto positivo como negativos, los resultados obtenidos por Progol4.2 y los resultados obtenidos por *AIBED*.

```
% Base de conocimientos
% has_covering/2:
has_covering(dog, hair).
has_covering(dolphin, none).
has_covering(platypus, hair).
has_covering(bat, hair).
has_covering(trout, scales).
has_covering(herring, scales).
has_covering(shark, none).
has_covering(eel, none).
has_covering(lizard, scales).
has_covering(crocodile, scales).
has_covering(t_rex, scales).
has_covering(snake, scales).
has_covering(turtle, scales).
has_covering(eagle, feathers).
has_covering(ostrich, feathers).
has_covering(penguin, feathers).

% has_legs/2:
has_legs(dog, 4).
has_legs(dolphin, 0).
has_legs(platypus, 2).
has_legs(bat, 2).
has_legs(trout, 0).
has_legs(herring, 0).
has_legs(shark, 0).
has_legs(eel, 0).
has_legs(lizard, 4).
has_legs(crocodile, 4).
has_legs(t_rex, 4).
has_legs(snake, 0).
has_legs(turtle, 4).
has_legs(eagle, 2).
has_legs(ostrich, 2).
```

```
has_legs(penguin,2).

% has_milk/1:
has_milk(dog).
has_milk(dolphin).
has_milk(bat).
has_milk(platypus).
has_milk(cat).

% homeothermic/1:
homeothermic(dog).
homeothermic(dolphin).
homeothermic(platypus).
homeothermic(bat).
homeothermic(eagle).
homeothermic(ostrich).
homeothermic(penguin).
homeothermic(cat).

% habitat/2:
habitat(dog,land).
habitat(dolphin,water).
habitat(platypus,water).
habitat(bat,air).
habitat(bat,caves).
habitat(trout,water).
habitat(herring,water).
habitat(shark,water).
habitat(eel,water).
habitat(lizard,land).
habitat(crocodile,water).
habitat(crocodile,land).
habitat(t_rex,land).
habitat(snake,land).
habitat(turtle,water).
habitat(eagle,air).
habitat(eagle,land).
habitat(ostrich,land).
habitat(penguin,water).
```

```
% has_eggs/1:
has_eggs(platypus).
has_eggs(trout).
has_eggs(herring).
has_eggs(shark).
has_eggs(eel).
has_eggs(lizard).
has_eggs(crocodile).
has_eggs(t_rex).
has_eggs(snake).
has_eggs(turtle).
has_eggs(eagle).
has_eggs(ostrich).
has_eggs(penguin).

% has_gills/1:
has_gills(trout).
has_gills(herring).
has_gills(shark).
has_gills(eel).
```

Conjunto de ejemplos positivos:

```
% Ejemplos positivos
% class/2:
class(eagle,bird).
class(bat,mammal).
class(dog,mammal).
class(ostrich,bird).
class(shark,fish).
class(cat,mammal).
class(eel,fish).
class(crocodile,reptile).
class(t_rex,reptile).
class(platypus,mammal).
class(penguin,bird).
class(trout,fish).
```

```
class(lizard,reptile).
class(herring,fish).
class(snake,reptile).
class(dolphin,mammal).
class(turtle,reptile).
```

Conjunto de ejemplos negativos:

```
%Ejemplos negativos
% class/2:
class(bat,reptile).
class(dolphin,reptile).
class(turtle,mammal).
```

4.2.1 Resultados utilizando PROGOL 4.2

```
class(A,bird) :- has_covering(A,feathers).
class(A,mammal) :- has_milk(A).
class(A,fish) :- has_gills(A).
class(A,reptile) :- habitat(A,land).
class(A,reptile) :- has_legs(A,4).
```

Estos son los resultados obtenidos por Progol considerando el mismo conjunto de entrenamiento, Progol requiere de ciertos parámetros adicionales para dirigir su búsqueda como lo son la declaración de modos y tipos aquí presentados.

```
% Mode declarations

:- modeh(1,class(+animal,#class))?
:- modeb(1,has_gills(+animal))?
:- modeb(1,has_covering(+animal,#covering))?
:- modeb(1,has_legs(+animal,#nat))?
:- modeb(1,homeothermic(+animal))?
:- modeb(1,has_eggs(+animal))?
:- modeb(1,not has_gills(+animal))?
```

```

:- modeb(1,nhas_gills(+animal))?
:- modeb(*,habitat(+animal,#habitat))?
:- modeb(1,has_milk(+animal))?
:- modeh(1,false)?
:- modeb(1,class(+animal,#class))?

% Types

animal(dog). animal(dolphin). animal(platypus). animal(bat).
animal(trout). animal(herring). animal(shark). animal(eel).
animal(lizard). animal(crocodile). animal(t_rex). animal(turtle).
animal(snake). animal(eagle). animal(ostrich). animal(penguin).

class(mammal). class(fish). class(reptile). class(bird).

covering(hair). covering(none). covering(scales). covering(feathers).

habitat(land). habitat(water). habitat(air). habitat(caves).

```

4.2.2 Resultados utilizando AIBED

Hipotesis formuladas

```

% class/2:
class(A,bird) :-
    homeothermic(A).
class(A,mammal) :-
    has_milk(A).
class(A,fish) :-
    has_eggs(A).
class(A,reptile) :-
    has_covering(A,scales),
    has_covering(trout,scales),
    has_eggs(trout).
class(A,fish) :-
    has_gills(A).

```

Comparando los resultados de ambos sistemas queda claro que se puede llegar a diversas soluciones validas.

Capítulo 5

Trabajo futuro y Conclusiones

5.1 Trabajo futuro

En el desarrollo de las pruebas se pusieron en evidencia algunas limitantes del algoritmo, como lo son la dependencia de las soluciones al parámetro `profundidad/1`, el hecho de no poder utilizar negaciones y la incapacidad de trabajar con cláusulas recursivas.

Para solucionar el problema de la dependencia de la `profundidad` se puede forzar a la implementación de tal forma que el parámetro `profundidad/1` en vez de interpretarse como “*considera los primeros N términos (no importando el símbolo de predicados) del Modelo Minimal que contengan o que se instancien con a lo menos un elemento de C* ” se interprete como “*considera los primeros N términos de cada símbolo de predicado diferente del Modelo Minimal que contengan o que se instancien con a lo menos un elemento de C* ”. Este cambio puede evitar casos en los cuales existen una gran cantidad de posibles términos para la construcción de cuerpos con un mismo símbolo de predicados y esto nos haga que se excluyan del análisis otros símbolos de predicado como ocurrió con el ejemplo de `ancestro` con profundidades inferiores a 30.

En el caso de las negaciones, se tendría que incluir un módulo que considere los símbolos de predicado que van siendo utilizados para cada hipótesis y de este modo poder detectar al final si existe alguno que no se utiliza en ninguna hipótesis, en tal caso se podría intentar incorporarlo a las hipótesis en forma negada cuidando que no se logre implicar lógicamente ejemplo negativo alguno e intentar la compactación de las mismas.

Respecto a las cláusulas recursivas, resultan interesantes los resultados obtenidos al aprender la relación *ancestro*, ya que se ve claramente en ellos un patrón de recursividad, la adición en este sentido sería ampliar el módulo de compactación de hipótesis para que este pudiera detectar patrones recursivos e intentara construir cláusulas recursivas.

Adicionalmente a lo ya mencionado, es necesario que el *AIBED* pueda trabajar con listas para que con ello sea más robusto y pueda extender un poco las restricciones de su lenguaje.

5.2 Conclusiones

Este trabajo se enfoca básicamente en la idea de que la construcción de los cuerpos de las hipótesis debe ser dirigida 100% por los mismos datos que se encuentran representados en la Base de Conocimientos expresados ya sea en forma de términos básicos (hechos) o en forma de cláusulas definidas, sin embargo nos enfrentamos a algunos problemas intrínsecos a la Programación Lógica Inductiva como lo es la no sanidad y no completos de la inducción y la riqueza de expresividad de la Programación Lógica que si bien enriquece la expresividad de las soluciones, también dificulta su construcción dado el espacio de búsqueda tan grande, por lo que es necesario utilizar cotas como la profundidad de la búsqueda.

En base al objetivo de la tesis que es *construir un algoritmo eficaz para la construcción de nuevas cláusulas a partir de un programa lógico*, remitiendonos a los resultados obtenidos y al análisis realizado en las pruebas, considero que es factible hacer que *AIBED* logre un nivel de robustez, considerando los planteamientos del trabajo a futuro, que le permita incursionar en problemas reales que tengan un alto grado de

complejidad y que a la vez sea capaz de dar resultados satisfactorios.

Bibliografía

- [GPM91] Luciano García, Olga Padrón, y René Moreno. *Programación Lógica: notas de curso*. Ministerio de educación superior, julio 1991.
- [MR94] Stephen Muggleton y Luc De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19 & 20:629–680, mayo 1994.
- [Mug94] Stephen Muggleton. Inductive logic programming: derivations, successes and shortcomings. *SIGART Bulletin*, 5(1):5–11, 1994.
- [Mug95] S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.
- [Nil96] Nils J. Nilsson. *Introduction to machine learning*. September 26 1996.
- [QCJ94] J. R. Quinlan y R. M. Cameron-Jones. Efficient top-down induction of logic programs. *SIGART Bulletin*, 5(1):33–42, 1994. .
- [Qui96] J.R. Quinlan. Learning first-order definitions of functions. *The Journal of Artificial Intelligence*, páginas 139–161, 1996.
- [Rob] S. Roberts. *An introduction to Progol*.
- [Yam97] Akihiro Yamamoto. Which hypotheses can be found with inverse entailment? *The full paper is submitted to the ILP-97 Workshop*, 1997. Extended Abstract.

- [ZM97] Jhon M. Zelle y Raymond J. Mooney. An inductive logic programming method for corpus-based parser construction. marzo 1997.

Apéndice A

Implementación de *AIBED*

AIBED fue implementado en BinProlog y probado en una Sun microsystem ULTRA 5.

A.1 Cargando *AIBED* en BinProlog

Llamar a BinProlog desde la línea de comandos con la instrucción bp:

```
asterion% bp aibed.pl
```

```
BinProlog 7.50 Copyright (C) Paul Tarau 1992-98,BinNet  
Corp. 1998-99  
http://www.binnetcorp.com  
E-MAIL to : binnetcorp@binnetcorp.com  
For Internet talk type ?-listen. and ?-chat. in 2 windows.  
(C-ified standalone)  
(with heap GC enabled)  
Detected hostname: undetected (type bp -p10 to detect host)  
Evaluation copy: 19 days left.  
Finished loading system C-code (49250 instructions).  
Finished loading user C-code (4 instructions).  
compiling(to(mem),aibed.pl,...)  
bytes_used(code(16456),strings(3604),symbols(1272),
```

```
htable(4632),total(25964))
compile_time(700)
```

Algoritmo Inductivo Basado En Datos - AIBED
Maestria en Inteligencia Artificial - Universidad
Veracruzana / LANIA

```
aibed>
```

A.2 Llamando la ayuda de *AIBED*

Una vez en el interprete de *AIBED* se pueden ejecutar diversos comandos, como lo es `help/1`. el cual nos muestra una lista con los comandos validos para el sistema.

```
aibed> help.
```

```
== COMANDOS DE ARCHIVOS ==
```

```
*load(FILE). Carga los archivos con nombre FILE y estension  
.pl, .pos y .neg.
```

```
*save(BD,NAME). Guarda una Base de Datos BD en un archivo  
llamado NAME.
```

```
*path(PATH). Cambia el subdiretorio de trabajo segun PATH.  
*path. Muestra el valor actual de PATH.
```

```
== COMANDOS DE BASE DE DATOS ==
```

```
*list(BD). Muestra el contenido de la Base de Datos DB,  
donde BD puede ser: bc, ep, en, hipo.
```

```
*incluye_hipo. Agrega a la BD bc (base de conocimiento)  
las clausulas de hipo (hipotesis generadas).
```

```
== PARAMETROS DE AIBED ==
```

```
*cardinalidad(N). Especifica el maximo de terminos que
```

pueden tener las hipótesis construidas, donde N es un número entero.

*cardinalidad. Muestra el valor actual N (de cardinalidad).

*limite(N). Especifica un límite de bondad para la métrica de mejor hipótesis (es un límite inferior) y está definido 0.60. Donde $0 \leq N \leq 1$.

*limite. Muestra el valor actual N (de límite).

*profundidad(N). Especifica el máximo de elementos de BC que pueden ser utilizados en el análisis para la construcción de una hipótesis. Donde N es un número entero..

*profundidad. Muestra el valor actual N (de profundidad).

*generaliza(Val). Indica si está activo el proceso de generalización. Donde Val=yes/no.

*generaliza. Muestra el valor actual Val (de generaliza).

*learn. Una vez que se ha cargado un archivo, learn dispara el proceso de aprendizaje.

*help. Muestra esta ayuda.

*exit. Salir del sistema.

aibed>

A.3 Cargando un ejemplo

Para cargar un archivo es necesario que se encuentren en el mismo subdirectorio los archivos .pl, .pos y opcionalmente el .neg, e indicar a *AIBED* con el comando `load/1` el nombre de los archivos y opcionalmente especificar la ubicación de los archivos con `path/1`. A continuación se presenta los pasos a seguir para cargar el ejemplo denominado *hermano* que se encuentra en el subdirectorio de `Ejemplos/Arbol/`, para ello, con el comando `path/0` se verifica en cual directorio se encuentra actualmente. Con el comando `path/1` se cambia al directorio

correcto, y con el comando `load/1` se carga el archivo.

```
aibed> path.
  Path-> Ejemplos/
aibed> path('Ejemplos/Arbol/').
aibed> load(hermano).
consulting(Ejemplos/Arbol/hermano.pl)
consulted(Ejemplos/Arbol/hermano.pl)
time(consulting = 10,quick_compiling = 0,static_space = 0)
consulting(Ejemplos/Arbol/hermano.pos)
consulted(Ejemplos/Arbol/hermano.pos)
time(consulting = 10,quick_compiling = 0,static_space = 0)
Validacion de Necesidad Previa SATISFACTORIA
consulting(Ejemplos/Arbol/hermano.neg)
consulted(Ejemplos/Arbol/hermano.neg)
time(consulting = 10,quick_compiling = 0,static_space = 0)
aibed>
```

Una vez que el ejemplo ha sido cargado se podrían listar las bases de datos `bc`, `ep`, `en` e `hipo` con el comando `list/1` para ver cual es la información con la que va a trabajar el algoritmo. En el caso de `hipo` aparecera siempre vacío antes de llamar a `learn/1` y `ep` aparecera siempre vacío después de ejecutar `learn`.

A.4 Disparando el proceso inductivo de *AIBED*

Para que *AIBED* inicie el proceso inductivo es necesario indicarlo con el comando `learn`.

```
aibed> learn.
ei=hermano(b,c)
  H=hermano(_x7114,_x7115) :- padre(a,_x7114),padre(a,_x7115)
ei=hermano(f,g)
  H=hermano(_x7114,_x7115) :- padre(c,_x7114),padre(c,_x7115)
```

```
--> Intenta generalizar  
==> Generalizacion
```

Hipotesis formuladas

```
hermano/2:  
hermano(A,C) :-  
    padre(B,A),  
    padre(B,C).
```

```
aibed>
```

Esta implementación de *AIBED* tiene un “bug”, en ocasiones después de cargar varios ejemplos en una misma sesión comienza a marcar errores de memoria, estos se eliminan ejecutando el comando `clean/0`. desde el intérprete de *AIBED*.