Discrimination-Based Constructive Induction of Logic Programs

Boonserm Kijsirikul **Masayuki Numao** **Masamichi Shimura**
Department of Computer Science, Tokyo Institute of Technology
2-12-1 Oh-okayama, Meguro, Tokyo 152, JAPAN
Email: {boon, numao, shimura}@cs.titech.ac.jp

Abstract

This paper presents a new approach to constructive induction, *Discrimination-Based Constructive induction(DBC)*, which invents useful predicates in learning relations. Triggered by failure of selective induction, DBC finds a *minimal* set of variables forming a new predicate that discriminates between positive and negative examples, and induces a definition of the invented predicate. If necessary, it also induces subpredicates for the definition. Experimental results show that DBC learns meaningful predicates without any interactive guidance.

Introduction

Learning systems find a concept based on positive and negative examples by using given terms, such as features and predicates. Most learning systems employ selective induction, and find a concept description composed of only predefined terms. However, if such terms are not appropriate, constructive induction [Michalski, 1983] or shift of bias [Utgoff, 1986] are required to invent new terms. Several systems have been developed for term invention. Most systems use a feature-value language for representing examples and a concept description, and invent a new feature by combining the given features [Matheus, 1990; Pagallo, 1989].

Due to a lack of expressive power in feature-value languages, there has been increasing interest in systems which induce a first-order logic program from examples [Muggleton and Feng, 1990; Quinlan, 1990]. Some of these systems perform constructive induction. FOCL [Siverstein and Pazzani, 1991] invents new terms, i.e., new predicates, by combining existing subpredicates. Based on interaction with its user, CIGOL [Muggleton and Buntine, 1988] invents terms without any given subpredicate.

This paper presents Discrimination-Based Constructive induction (DBC) which invents a new predicate without any given subpredicate nor any user interaction. Triggered by failure of selective induction, DBC introduces a new predicate into a previously found incomplete clause. This is performed by searching for a *minimal* relevant variable set forming a new predicate that discriminates between positive and negative examples. If necessary, DBC also recursively invents subpredicates for the definition.

Experimental results show that, without interactive guidance, our system CHAMP can construct meaningful predicates on predefined ones or from scratch. Our approach is system independent and applicable to other selective learning systems such as FOIL [Quinlan, 1990].

Discrimination-Based Constructive Induction

As the purpose of learning a program is to discriminate between positive and negative examples, a new predicate is likely to be useful if it is based on discrimination between the examples. In this section, we describe our DBC problem and then provide an algorithm to solve the problem.

The DBC Problem

The problem we are interested in is as follows:

Given:
- an incomplete clause C
- positive and negative examples covered by C

Determine:
- a clause C' that is the result of adding a new literal R $(= P(...))$ to the body of C
- a definition of the predicate P

The incomplete clause and the examples are obtained by selective induction. To simplify the problem, we assume that (i) C' *completely* discriminates between positive and negative examples and (ii) R is composed *only of variables* and *all* of them occur in C.

In our approach, a new predicate is invented by searching for a *minimal* set of variables forming a relation R that discriminates between the examples. A set is minimal if it has no proper subset which forms such a relation. We do not look for non-minimal variable sets since they contain superfluous variables.

Definition 1 A clause C covers an atom e with respect to (w.r.t.) logic program K, denoted $C\theta\sigma >_K e$, or simply $C >_K e$, iff there are substitutions θ, σ such that $C_{head}\theta$ is identical to e and $C_{body}\theta\sigma \subseteq D^*(K)$; where C_{head}, C_{body} and $D^*(K)$ are the head of C, the body of C, and an atomic-derivation-closure [1] [Muggleton and Feng, 1990] of K, respectively.

The definition extends naturally to a set of atoms E, and is written $C >_K E$. Using the definition, the DBC problem is formalized as:

Given: $C >_K E^+$, $C >_K E^-$. (1)

Find: $\langle R, \Delta \rangle$ such that,

(i) $\langle R, \Delta \rangle$ is *complete* (covering all positive examples) and is *consistent* (not covering any negative example), i.e.,
$(C \vee \neg R) >_{(K \wedge \Delta)} E^+$, $(C \vee \neg R) \not>_{(K \wedge \Delta)} E^-$ (2)

(ii) $\langle R, \Delta \rangle$ is *minimal*, i.e., there is no other complete and consistent tuple $\langle R', \Delta' \rangle$ such that $\text{var}(R') \subset \text{var}(R)$.

where K is a set of background clauses and C is an incomplete clause. E^+, E^- are sets of positive and negative examples covered by C, respectively. R is a new literal. Δ and $\text{var}(R)$ are a set of ground instances of R and a set of variables occurring in R, respectively.

We restrict a set of instances Δ to a set that consists of only bindings in substitutions making C cover E^+. Let $\text{var}(C_{head}) = \{Y_1, \ldots, Y_m\}$ and $\text{var}(C_{body}) = \{Y_{m+1}, \ldots, Y_n\}$ be a set of variables occurring in C_{head} and only in C_{body}. Let $\{\theta_i\sigma_j\} = \{Y_1/a_{1,i}, \ldots, Y_m/a_{m,i}, Y_{m+1}/a_{m+1,j}, \ldots, Y_n/a_{n,j}\}$ be a set of substitutions that make C cover the ith positive example e_i^+ w.r.t. background knowledge $K(C\theta_i\sigma_j >_K e_i^+)$. The *initial hypothesis* of a solution $\langle R_0, \Delta_0 \rangle$ is defined as:

$$\langle R_0, \Delta_0 \rangle = \langle \$(Y_1, \ldots, Y_n), \bigwedge_i \bigwedge_j \$(a_{1,i}, \ldots, a_{m,i}, a_{m+1,j}, \ldots, a_{n,j}) \rangle,$$

where \$ is a predicate not appearing in K and C. R_0 and Δ_0 are composed of all variables in C and all bindings of those variables.

Theorem 2 Given (1), if every positive example is different from every negative one, $\langle R_0, \Delta_0 \rangle$ is complete and consistent. (For proof of the theorems see [Kijsirikul *et al.*, 1992].)

The initial hypothesis $\langle R_0, \Delta_0 \rangle$ may contain superfluous variables which must be removed.

Definition 3 $\langle R, \Delta \rangle_{-\{Z_1, \ldots, Z_m\}}$ denotes a pair $\langle R', \Delta' \rangle$ obtained by removing $\{Z_1, \ldots, Z_m\}$ from R and the corresponding terms from Δ.

[1] $D^*(K) = (D^0(K) \cup D^1(K) \cup \ldots)$, where $D^0(K)$ is the set of unit clauses in K and $D^n(K) = D^{n-1}(K) \cup \{A\theta_1 \ldots \theta_n \mid A \leftarrow B_1, \ldots, B_n \in K$ and for each B_i there exists $B_i' \in D^{n-1}(K)$ such that θ_i is the mgu of B_i and $B_i'\}$

Given $\langle R_0, \Delta_0 \rangle$, the search space for solving the DBC problem is: $H_{All} =$
$\{\langle R_0, \Delta_i \rangle | \Delta_i \subseteq \Delta_0\} \cup \{\langle R_0, \Delta_i \rangle_{-A} | A \subset \text{var}(R_0)\}$

Example 1 Let $\langle R_0, \Delta_0 \rangle$ = ⟨p(X,Y,Z),{p(2,1,4), p(2,1,5)}⟩.$\{\langle R_0, \Delta_i \rangle | \Delta_i \subseteq \Delta_0\}$ = {⟨p(X,Y,Z),{p(2,1,4), p(2,1,5)}⟩, ⟨p(X,Y,Z),{p(2,1,4)}⟩, ⟨p(X,Y,Z),{p(2,1,5)}⟩}. $\langle R_0, \Delta_0 \rangle_{-\{X\}}$ = ⟨p(Y,Z),{p(1,4),p(1,5)}⟩, etc. $\{\langle R_0, \Delta_i \rangle_{-A} | A \subset \{X, Y, Z\}\}$ = {⟨p(Y,Z),{p(1,4),p(1,5)}⟩, ⟨p(Y,Z),{p(1,4)}⟩, ⟨p(Y,Z),{p(1,5)}⟩, ⟨p(X,Z),{p(2,4), p(2,5)}⟩, ⟨p(X,Z),{p(2,4)}⟩, ⟨p(X,Z),{p(2,5)}⟩, ⟨p(X,Y),{p(2,1)}⟩, ⟨p(Z),{p(4),p(5)}⟩, ⟨p(Z),{p(4)}⟩, ⟨p(Z),{p(5)}⟩, ⟨p(Y),{p(1)}⟩, ⟨p(X),{p(2)}⟩}. □

The DBC Algorithm

To find a solution, one straightforward algorithm would enumerate $\langle R, \Delta \rangle$ in H_{All} from small to big until it finds such a pair that is complete. However, this is intractable since its time complexity is exponential in the number of variables in the clause. Fortunately, there is a DBC algorithm that accomplishes the task whose time complexity is a linear function of the number of variables. The following *greedy* removal algorithm has been adopted.

Algorithm DBC
Input: A clause C, a set of background clauses K and sets of positive and negative examples E^+, E^-, where $\{Y_1, \ldots, Y_n\}$ is a set of variables in C.
begin
  $\langle R, \Delta \rangle \leftarrow \langle R_0, \Delta_0 \rangle$
  for i := 1 to n do **begin**
    $\langle R', \Delta' \rangle \leftarrow \langle R, \Delta \rangle_{-\{Y_i\}}$;
    $C' \leftarrow C \vee \neg R'$;
    if there exists Δ^+ such that
      $(\Delta^+ \subseteq \Delta') \bigwedge (C' >_{(K \wedge \Delta^+)} E^+)$
      $\bigwedge (C' \not>_{(K \wedge \Delta^+)} E^-)$
    then
      $\langle R, \Delta \rangle \leftarrow \langle R', \Delta' \rangle$ /* Y_i is irrelevant */
  end
  Let $var(R) = \{Z_1, \ldots, Z_l\}$;
  Find a set of possible bindings of $(Z_1, \ldots, Z_l)$ such that $C >_K E^-$ and use them as a set of negative instances, Δ^-, of R
end
Output: $\langle R, \Delta^+ \rangle, \Delta^-, C' = C \vee \neg R$

Figure 1: The DBC Algorithm

The basic idea of the algorithm is to remove irrelevant variables one by one. If a variable is removed and there exists $\langle R', \Delta^+ \rangle$ that enables the clause to still cover all positive examples but no negative examples, then that variable is considered to be irrelevant. Although $K \wedge \Delta_0$ derives no negative examples, $K \wedge \Delta'$ could derive some of them since Y_i is removed. To avoid such derivation, Δ^+ is calculated by removing from Δ' all its elements that derive any negative examples.

Theorem 4 The DBC algorithm produces a complete, consistent and minimal $\langle R, \Delta^+ \rangle$.

Note that a minimal hypothesis $\langle R, \Delta^+ \rangle$ produced by the algorithm does not necessarily contain the minimum number of variables, i.e., the number of variables in R is a local minimum.

Overview of CHAMP

The authors have constructed a learning system, CHAMP, to test the DBC algorithm. It consists of a selective induction component, CHAM, and a predicate invention component based on the algorithm.

Selective Induction Component: CHAM

CHAM [Kijsirikul *et al.*, 1991] is a logic program learning component that employs refinement operators in MIS [Shapiro, 1983] to specialize a clause by: (i) instantiating a head variable to a function, (ii) unifying variables, or (iii) adding a background predicate to a body. It employs the *Merit* heuristic which is a combination of *Gain* in FOIL and *likelihood* to avoid exhaustive search [Kijsirikul *et al.*, 1991].

Failure Driven Constructive Induction in CHAMP

CHAMP constructs a new predicate when the selective induction component fails, i.e.:

- There is no specialized clause that covers positive examples, or
- The number of bits encoding the specialized clause exceeds the number of bits encoding positive examples covered by the clause, and the clause does not achieve reasonable accuracy.

The second condition follows the argument in [Quinlan, 1990]. If a clause is reasonably accurate, the inexact clause is retained as a final clause. This restriction on encoding bits is used to avoid overfitting noisy examples.

The number of bits encoding a clause is defined recursively as follows: the number of bits required to code the *most general term* [Kijsirikul *et al.*, 1991] is 0:

$$bits(MGT) = 0.$$

The number of bits required to code each specialized clause C_i is:

- if C_i is specialized by instantiating a head variable in C to a function,
  $$bits(C_i) = bits(C) + log_2(3) + log_2(Ni),$$
  where Ni is the number of specialized clauses obtained by instantiation;
- if C_i is specialized by unifying variables in C,
  $$bits(C_i) = bits(C) + log_2(3) + log_2(Nu),$$
  where Nu is the number of specialized clauses obtained by unification; or
- if C_i is specialized by adding a predicate to the body of C,
  $$bits(C_i) = bits(C) + log_2(3)$$
  $$+log_2(\text{number of predicates} + 1)$$
  $$+log_2(\text{number of possible arguments}).$$

An extra 1 in log_2(number of predicates + 1) is for a new predicate. $Log_2(3)$ indicates the number of bits to specify one operation out of the above three.

On the other hand, the number of bits required to encode positive examples is:

$$bits(p, p+n) = log_2(p+n) + log_2\left(\left(\begin{array}{c} p+n \\ p \end{array}\right)\right),$$

where p and n are the number of positive examples and negative examples, respectively.

The number of bits encoding a program P which contains a set of clauses, $\{C_1, \ldots, C_m\}$, is:

$$Bits(P) = bits(C_1) + \cdots + bits(C_m).$$

Learning Algorithm of CHAMP

In this section we describe the learning algorithm of CHAMP. The algorithm first tries to learn a clause composed of only given predicates. If no such clause is found, the algorithm then invents a new predicate that is represented in terms of its instances. These instances are then fed into the algorithm so that the algorithm can learn their definition, as shown in Figure 2.

When the selective induction component fails to discover a clause, as candidates for introducing new predicates the algorithm selects n incomplete clauses that have higher scores in the following heuristic:

$$Score(C) = Merit(C) \times (bits(CovPos, Total) - bits(C)),$$

where C, $CovPos$ and $Total$ are a clause, the number of positive examples covered by the clause and the number of all examples, respectively. The algorithm outputs either programs or instances of new predicates which minimize the encoding bits. In other words, the number of bits encoding positive examples covered by a clause added by a predicate should be greater than the number of bits encoding the clause and a definition (P and/or Is) of that predicate.

There are two criteria for stopping predicate invention: (1) the number of bits encoding all clauses exceeds that encoding the positive examples of the target concept; and (2) the instances of a new predicate $(NewPs_j, NewNs_j)$ are the same as examples $(PosExs, NegExs)$ except for the name of the predicate.

Learning Sort without Background Knowledge

Below we demonstrate that a sort program is learned without background knowledge by inventing subpredicates. Training examples are selected from all lists of length up to three, each containing non-repeated atoms drawn from the set {0,1,2}. We use a predicate with symbol $ to indicate that it is created by the system, and name it appropriately for readability.

Precedure CHAMP($PosExs$,$NegExs$,$Clauses$,$Instances$);
/* **input:** $PosExs, NegExs$ are sets of positive and negative examples */
/* **output:** $Clauses, Instances$ are sets of clauses and positive instances */
begin
$Clauses \leftarrow \{\}$; $Instances \leftarrow \{\}$;
while $PosExs \neq \{\}$ **do begin**
Choosc a seed example e from $PosExs$ and call CHAM to learn a clause C which covers e;
if succeed **then** add C to $Clauses$ and remove positive examples covered by it from $PosExs$
else begin
Select n previously found incomplete clauses(C_i) by CHAM that have higher scores in $Score(C_i)$;
for i := 1 to n **do begin**
Use DBC algorithm to construct a clause $NC_i(= C_i \vee \neg R_i)$ whose body is added by a new predicate(literal) R_i;
Calculate positive and negative instances of the new predicate($NewPs_i$, $NewNs_i$);
$CovPosBits_i \leftarrow$ bits(|positive examples covered by NC_i|,|$PosExs$| + |$NegExs$|)
end;
Select the best new clause NC_j that has the maximum value of
$CovPosBits_j$ - (bits(NC_j) + bits(|$NewPs_j$|,|$NewPs_j$| + |$NewNs_j$|));
Call CHAMP($NewPs_j, NewNs_j, P, Is$) rccursivcly to learn a definition (P and/or Is) of the new predicate;
if $CovPosBits_j \geq$ (bits(NC_j) + Bits(P) + bits(|Is|,|$NewPs_j$| + |$NewNs_j$|)) **then begin**
Add NC_j and P to $Clauses$, Is to $Instances$;
Remove positive examples covered by NC_j from $PosExs$
end
else begin Add $PosExs$ to $Instances$; $PosExs \leftarrow \{\}$ **end**
end
end
end.

Figure 2: Learning Algorithm of CHAMP

Because useful background predicates such as **partition, append**, <, >= are not given, the selective induction component fails to discover a clause. At this step, incomplete clauses with high scores are:

```
C1:  sort([X|Y],Z) :-        sort(Y,V)
C2:  sort([X|Y],Z) :-
C3:  sort([X|Y],[Z|U]) :-    sort(Y,W)
                ...
```

CHAMP introduces **$insert** into the first clause as follows:
var($C1$) = {X,Y,Z,V}, R = **$insert(X,Y,Z,V)**. First, Δ is calculated:

$$\Delta = \left\{ \begin{array}{l} \texttt{\$insert}(2,[1,0],[0,1,2],[0,1]) \wedge \\ \texttt{\$insert}(2,[0,1],[0,1,2],[0,1]) \wedge \\ \texttt{\$insert}(2,[1],[1,2],[1]) \wedge \\ \texttt{\$insert}(2,[0],[0,2],[0]) \wedge \\ \qquad\qquad ... \end{array} \right\}$$

To remove irrelevant variables, $\langle R', \Delta' \rangle$ is computed and tested.

1st loop:
$\langle R', \Delta' \rangle = \langle R, \Delta \rangle_{-\{X\}} =$
$\langle$**$insert**(Y,Z,V), {**$insert**([1,0],[0,1,2],[0,1])...}$\rangle$
C' = **sort([X|Y],Z) :- sort(Y,V), $insert(Y,Z,V)**
There is no $\Delta^+ \subseteq \Delta'$ that makes $\langle R', \Delta^+ \rangle$ complete.
$\langle R, \Delta \rangle = \langle$**$insert**(X,Y,Z,V), $\Delta\rangle$

2nd loop:
$\langle R', \Delta' \rangle = \langle R, \Delta \rangle_{-\{Y\}} =$
$\langle$**$insert**(X,Z,V), {**$insert**(2,[0,1,2],[0,1])...}$\rangle$
C' = **sort([X|Y],Z) :- sort(Y,V), $insert(X,Z,V)**
There exists a $\Delta^+ = \Delta'$ that makes $\langle R', \Delta^+ \rangle$ complete.
$\langle R, \Delta \rangle = \langle$**$insert**(X,Z,V), $\Delta'\rangle$
Removing Z or V is a similar process. The obtained clause is:
sort([X|Y],Z) :- sort(Y,V), $insert(X,Z,V).
with instances of **$insert** listed below:

positive instances(Δ^+):

```
$insert(2,[1,2],[1])     $insert(2,[0,1,2],[0,1])
$insert(2,[0,2],[0])     $insert(2,[2],[])
$insert(1,[1,2],[2])     $insert(1,[0,1,2],[0,2])
                ...
```

negative instances(Δ^-):

```
$insert(0,[],[])         $insert(0,[0],[1])
$insert(0,[0],[2])       $insert(0,[0,1],[])
$insert(0,[0,1],[1,2])   $insert(0,[0,1],[2])
                ...
```

CHAMP also tries to introduce new predicates into the other clauses but the first clause is selected since it has the maximum value.

Positive and negative instances of **$insert** are passed to the selective induction component to learn a definition of **$insert**. The component fails again and then CHAMP attempts to introduce a new predicate into the clause **$insert(X,[Y|Z],[Y|V]) :- $insert(X,Z,V)**, and finds the first clause of **$insert**:

```
$insert(X,[Y|Z],[Y|V]) :-
        $insert(X,Z,V), $greaterThan(X,Y).
```

with instances of **$greaterThan** listed below.

Positive instances:

```
$greaterThan(1,0).       $greaterThan(2,0).
$greaterThan(2,1).
```

Negative instances:

```
$greaterThan(0,1).       $greaterThan(0,2).
```

```
$greaterThan(1,2).
```

While CHAMP learns a definition of **$greaterThan**, it needs a new predicate, and constructs **$greaterThan(X,Y) :- $p(X,Y)**. Since examples of **$p** are the same as those of **$greaterThan**, CHAMP terminates the predicate invention.

Finally, CHAMP learns the following program:

```
sort([],[]).
sort([X|Y],Z) :-
    sort(Y,V), $insert(X,Z,V).
$insert(X,[Y|Z],[Y|V]) :-
    $insert(X,Z,V), $greaterThan(X,Y).
$insert(X,[X,Z|U],[Z|U]) :- $greaterThan(Z,X).
$insert(X,[X],[]).
$greaterThan(1,0).      $greaterThan(2,0).
$greaterThan(2,1).
```

Experimental Results

An experiment was made on various learning problems in order to show that CHAMP invents meaningful predicates which lead to increased classification accuracies. Table 1 shows the results.

In all problems, the system was given positive examples and it generated negative examples under the closed-world assumption. In the **arch** problem, examples are given in a specific form of the arch as in [Muggleton and Buntine, 1988]. The test examples for all problems are randomly selected from supersets of training sets.

The program for the **sort** problem is the same as the above except that the given predicate '>' is used in **$insert** clauses instead of **$greaterThan**. The rest of the programs for problems in Table 1 are as follows:

grandmother:

```
grandmother(X,Y) :-
    parent(X,Z), parent(Z,Y), $female(X).
```

reverse:

```
reverse([],[]).
reverse([X|Y],Z) :- reverse(Y,V), $concat(X,V,Z).
$concat(X,[],[X]).
$concat(X,[Y|V],[Y|Z]) :- $concat(X,V,Z).
```

union:

```
union([],X,X).
union([X|Y],Z,U) :- union(Y,Z,U), $member(X,U).
union([X|Y],Z,[X|V]) :-
    union(Y,Z,V), $not-member(X,V).
$member(X,[X|Y]).
$member(X,[Y|Z]) :- $member(X,Z).
$not-member(X,[]).
$not-member(X,[Y|Z]) :-
    not-equal(Y,X), $not-member(X,Z).
```

arch:

```
arch(X,Y,X) :- $column-beam(X,Y).
$column-beam([X|Y],Z) :-
    $column-beam(Y,Z), $brick-or-block(X).
$column-beam([],X) :- $beam(X).
$brick-or-block(brick).  $brick-or-block(block).
$beam(beam).
```

The results reveal that CHAMP learns meaningful predicates when background knowledge is inappropriate or insufficient. There are two reasons for predicate invention. One reason is that CHAMP invents predicates for increasing classification accuracies as shown in Table 1. In the **grandmother** problem, accuracy with predicate invention did not reach 100% because **$female(X)** is defined by its instances and there are some people in the test examples that are not given in the training examples and are female. The other reason is to reduce the size of programs by avoiding overly complex clauses. During an experiment by the authors, FOIL discovered a **reverse** program without inventing predicates as follows:

```
reverse(A,B) :-
   components(A,C,D), components(B,E,F),
   reverse(D,G), components(G,E,H), reverse(H,I),
   reverse(F,J), components(J,C,I).
```

where "components([A|B],A,B)" is given as background knowledge. CHAMP learns a more compact program by inventing **$concat**.

Although we have not tested the effect of noise on predicate invention yet, we believe that the restriction on encoding bits could avoid overfitting examples with noise, as in FOIL.

Related Work

The DBC algorithm searches for a minimal relevant variable set forming a new predicate based on discrimination between the examples. This distinguishes it from other constructive induction algorithms. For instance, the intra-construction operator in CIGOL [Muggleton and Buntine, 1988] searches for relevant variables of a new predicate by considering the structure of terms sharing variables. Given resolvent clauses, for example, "`insert(1,[0],[0,1]) :- insert(1,[],[1])`" and "`insert(2,[1,0],[1,0,2]) :- insert(2,[0],[0,2])`", intra-construction constructs a clause "`insert(A,[B|C],[B|D]) :- insert(A,C,D),$greaterThan(A,B,C,D)`" with instances "`$greaterThan(1,0,[],[1])`" and "`$greaterThan(2,1,[0],[0,2])`", where C and D are not really relevant variables.[2] On the other hand, CHAMP, as demonstrated in the example, learns **$greaterThan(A,B)** consisting of only relevant variables A and B. Another advantage of CHAMP, compared to CIGOL, is that, as training examples are given as a batch, invention of predicates needs less user interaction with the system.

FOCL [Siverstein and Pazzani, 1991] is a constructive induction system extending FOIL. To construct a new predicate, FOCL uses *clichés* to constrain combination patterns of predefined predicates. The clichés help heuristic used in FOCL to search for a useful pred-

[2] If we give examples in an appropriate sequence, CIGOL learns the meaningful `$greaterThan(A,B)` which contains only relevant variables.

problem	background knowledge	number of training data (positive)	invented predicate	CPU(sec) (a)	number of test data (positive)	accuracy(%) (b)	(c)
sort	>, =<	256(16)	$insert	19.9	64(32)	100	67.2
grand-mother	parent, child, nephew-niece, uncle-aunt, couple, sibling	529(6)	$female	68.6	16(8)	99.7	98.6
arch	—	507(7)	$column-and-beam, $brick-or-block, $beam	7.9	28(14)	100	50.0
reverse	components	256(16)	$concat	24.2	64(32)	100	50.0
union	not-equal	3400(100)	$member, $not-member	413.4	400(200)	100	62.5

(a): CPU time for SICStus Prolog version 0.7 on SPARC station 2.
(b): Classification accuracy with predicate invention.
(c): Classification accuracy without predicate invention.

Table 1: Predicates invented by CHAMP and classification accuracies with and without predicate invention.

icate, whereas our approach uses DBC to construct a new predicate when the heuristic fails to discover a clause. It may be useful to combine the approaches of DBC and the clichés.

SIERES [Wirth and O'Rorke, 1991] is another constructive induction system. It constrains the search of a clause and a new predicate by using a limited sequence of *argument dependency graphs*. Our approach is more general than SIERES in the sense that we have no such constraints.

Our work is also related to CW-GOLEM [Bain, 1991]. Based on the Closed-World Specialization method, CW-GOLEM introduces a new predicate *not(R)* into the body of an over-general clause for handling *exceptions*. Our method can handle such exceptions by including *not(R)* into the hypotheses H_{All}.

Conclusion

We have described a new approach to constructive induction. The experimental results show that our system CHAMP can successfully construct meaningful predicates on existing ones or from scratch. The predicates are invented for increasing classification accuracies and reducing the size of programs. One reason for the successful results is that a minimal variable set that forms a new predicate simplifies the learning process. Simplification of the learning of a concept by good interaction between constructive and selective induction, is achieved by recursively inventing subpredicates.

Limitations of DBC are that a new predicate must completely discriminate between positive and negative examples, and a new predicate cannot contain new variables, constants and functions.

Our implementation of DBC is combined with a selective induction system CHAM. However, since DBC is system independent, it is applicable to other selective learning systems that search clauses in a top-down manner.

Acknowledgements

We would like to thank Seiichirou Sakurai, Surapan Meknavin and Somkiat Tangkitvanich for helpful discussions, and Niall Murtagh, who rewrote the paper.

References

Bain, M. 1991. Experiments in Non-Monotonic Learning. In *Proc. 8th Intl. Workshop on Machine Learning.* 380–384.

Kijsirikul, B.; Numao, M.; and Shimura, M. 1991. Efficient Learning of Logic Programs with Non-determinate, Non-discriminating Literals. In *Proc. the 8th Intl. Workshop on Machine Learning.* 417–421.

Kijsirikul, B.; Numao, M.; and Shimura, M. 1992. Predicate Invention Based on Discrimination. Technical report, Tokyo Institute of Technology.

Matheus, C. J. 1990. Adding Domain Knowledge to SBL Through Feature Construction. In *AAAI90.* 803–808.

Michalski, R. S. 1983. A Theory and Methodology of Inductive Learning. *Artificial Intelligence* 20:111–161.

Muggleton, S. and Buntine, W. 1988. Machine Invention of First Order Predicates by Inverting Resolution. In *Proc. the 5th Intl. Workshop on Machine Learning.* 339–352.

Muggleton, S. and Feng, C. 1990. Efficient Induction of Logic Programs. In *Proc. the First Intl. Workshop on Algorithmic Learning Theory*, Tokyo. OHMSHA. 368–381.

Pagallo, G. 1989. Learning DNF by Decision Trees. In *IJCAI89*, Detroit,MI. 639–644.

Quinlan, J. R. 1990. Learning Logical Definitions from Relations. *Machine Learning* 5:239–266.

Shapiro, E. Y. 1983. *Algorithmic Program Debugging.* The MIT Press, Cambridge, MA.

Siverstein, G. and Pazzani, M. J. 1991. Relational Clichés: Constraining Constructive Induction during Relational Learning. In *Proc. the 8th Intl. Workshop on Machine Learning.* 203–207.

Utgoff, P. E. 1986. Shift of Bias for Inductive Concept Learning. In Michalski, R.S.; Carbonell, J.G.; and Mitchell, T.M., editors 1986, *Machine learning: An artificial intelligence approach (Vol.2).* Morgan Kaufmann. 107–148.

Wirth, R. and O'Rorke, P. 1991. Constraints on Predicate Invention. In *Proc. the 8th Intl. Workshop on Machine Learning.* 457–461.