Scaling Up ILP to Large Examples: Results on Link Discovery for Counter-Terrorism

Lappoon R. Tang, Raymond J. Mooney, and Prem Melville

Department of Computer Sciences,
University of Texas at Austin,
Austin, TX 78712, U.S.A.
{rupert, mooney, melville}@cs.utexas.edu
http://www.cs.utexas.edu/users/ml/

Abstract. Inductive Logic Programming (ILP) has been shown to be a viable approach to many problems in multi-relational data mining (e.g., bioinformatics). Link discovery (LD) is an important task in data mining for counter-terrorism and is the focus of DARPA’s program on Evidence Extraction and Link Discovery (EELD). Learning patterns for LD is a novel problem in relational data mining that is characterized by having an unprecedented number of background facts. As a result of the explosion in the amount of background knowledge, the efficiency of existing ILP algorithms becomes a serious limitation. This paper presents a new ILP algorithm that integrates top-down and bottom-up search in order to reduce search when processing large examples. Experimental results on EELD data confirm that it significantly improves efficiency over existing ILP methods.

1 Introduction

The terrible events of September 11, 2001 have sparked increased development of information technology that can aid intelligence agencies in detecting and preventing terrorism. The Evidence Extraction and Link Discovery (EELD) program of the Defense Advanced Research Projects Agency (DARPA) is one attempt to develop new computational methods for addressing this problem. More precisely, Link Discovery (LD) is the task of identifying known, complex, multi-relational patterns that indicate potentially threatening activities in large amounts of relational data. Some of the input data for LD comes from Evidence Extraction (EE), which is the task of obtaining structured evidence data from unstructured, natural-language documents (e.g., news reports), other input data comes from existing relational databases (e.g., financial and other transaction data). Finally, Pattern Learning (PL) concerns the automated discovery of new relational patterns for detecting potentially threatening activities in large amounts of multi-relational data.

Scaling to large datasets in data mining typically refers to increasing the number of training examples that can be processed. Another measure of complexity that is particularly relevant in multi-relational data mining is the size
of individual examples, i.e. the number of facts used to describe each example. To our knowledge, the challenge problems developed for the EELD program are the largest ILP problems attempted to date in terms of the number of facts in the background knowledge. Relational data mining in bioinformatics [5] was probably the previously largest ILP problem in this sense. Table 1 shows a comparison between link discovery and, to our knowledge, the largest problem in bioinformatics.

<table>
<thead>
<tr>
<th>Domain</th>
<th># Bg. preds.</th>
<th>Avg. Arity</th>
<th># Bg. facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link Discovery</td>
<td>52</td>
<td>2</td>
<td>≈ 568k</td>
</tr>
<tr>
<td>Bioinformatics</td>
<td>36</td>
<td>4.9</td>
<td>≈ 24k</td>
</tr>
</tbody>
</table>

Table 1. Link Discovery versus Bioinformatics (e.g. carcinogenesis). # Bg. preds. is the number of different predicate names in the background knowledge, Avg. Arity is the average arity of the background predicates, and # Bg facts is the total number of background facts.

Scaling up ILP to efficiently process large examples like those encountered in EELD is a significant problem. Section 2 discusses the problems existing ILP algorithms have scaling to large examples and presents our general approach to controlling the search for multi-relational patterns by integrating top-down and bottom-up search. Section 3 presents the details of our new algorithm, BETH. Section 4 presents some theoretical results on our approach. Experimental results are presented and discussed in Section 5, followed by concluding remarks in Section 6.

2 Combining Top-down and Bottom-up Approaches in BETH

The two standard approaches to ILP are bottom-up and top-down [6]. Bottom-up methods start with a very specific clause generated from an individual positive example and generalize it as far as possible without covering negative examples. Top-down methods start with the most general (empty) clause and repeatedly specialize it until it no longer covers negative examples. Both approaches have problems scaling to large examples.

Most current bottom-up methods are based on the concept of inverse entailment [2]. The most popular approach to implementing inverse entailment is a two-stage process: 1) saturation which builds up the most specific clause (a.k.a. bottom clause) describing a positive example, and 2) truncation which finds solutions that generalize the bottom clause [3]. This approach is implemented in PROGOL [2] and its successor ALEPH.¹

¹ The Aleph Manual can be accessed via http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph.html.
Given a positive example and background knowledge, the bottom clause can be infinite, and practically one has to bound it. In PROGOL, it is bounded by five parameters: $i$, $r$, $M$, $j^-$, and $j^+$ (please refer to [2] for more details). Unfortunately, the complexity of PROGOL's bottom clause is exponential w.r.t. the variable depth $i$, which results in a hypothesis space that is doubly exponential! (The size of the subsumption lattice is two to the power of the size of the bottom clause.)

In problems with large examples like EELD, the background knowledge contains many facts using numerous predicates that describe each complex object or event. Typically, many of these facts are irrelevant to the task. However, PROGOL's bottom clause includes every piece of background knowledge (within the recall bound $r$) in its body. This leads to intractably large bottom-clauses which generates an exponentially larger hypothesis space when learning a clause. This leads one to wonder if it is possible to bound the bottom-clause differently so that it contains only a relevant subset of background facts.

A strength of the top-down approach is that the generation of literals is inherently directed by the heuristic search process itself: only the set of literals that make refinements to clauses in the search beam are generated. Clauses with insufficient heuristic value are discarded, saving the need to generate literals for them. So, there is a tangible link between the entire set of literals that could be included in a bottom-clause and the heuristic search for a good clause. Therefore, perhaps it is possible to employ the heuristic search as a guide to selecting a relevant subset of background facts for inclusion in an alternative bottom-clause.

A major weakness of the top-down approach (as far as literal generation is concerned) is that the enumeration of all possible combination of variables generates many more literals than necessary; some literals generated by the algorithm are not even guaranteed to cover one positive example. The complexity of enumerating all such combinations in FOIL [7] (and mFOIL [6]) is exponential w.r.t. the arity of the predicates [8]. A corresponding strength of the bottom-up approach is that a literal is created using a ground atom describing a known positive example. The advantages are: 1) specializing using this literal results in a clause that is guaranteed to cover at least the seed example, and 2) the set of literals generated are constrained to those that satisfy 1).

Given the strengths and weaknesses of typical top-down and bottom-up approaches, it seems that one can take advantage of the strength of each approach by combining them into one coherent approach. More precisely, we no longer build the bottom clause using a random seed example before we start searching for a good clause. Instead, after a seed example is chosen, one generates literals in a top-down fashion (i.e. guided by heuristic search) except that the literals generated are constrained to those that cover the seed example. Based on this idea, we have developed a new system called **Bottom-clause Exploration**.

---

2 Enforcing argument type restrictions can help lower the complexity but cannot completely solve the problem.
Through Heuristic-search (BETH) in which the bottom clause is not constructed in advance but “discovered” during the search for a good clause.\footnote{Progol and ALEPH are really, more precisely, “Subsumption lattice exploration through heuristic-search”. Here, we explore the bottom clause and the subsumption lattice simultaneously.}

## 3 The Algorithm

BETH’s bottom clause is virtual in the sense that the algorithm does not have to construct it to work, unlike Progol/ALEPH; it is, nonetheless, constructed to facilitate collection of statistics. However, the virtual bottom clause is a real bound on the subsumption lattice (see Section 4).

Let us begin with some basic definitions. A predicate specification takes the form PredName/Arity where PredName is the name of the predicate in concern and Arity its arity. A list of predicate specifications for the background knowledge is given to the ILP system before learning starts. The function predname(P) returns the predicate name of the predicate specification of the background predicates P and arity(P) returns its arity. Likewise, predname(L) returns the predicate name of the literal L and arity(L) returns its arity.

### 3.1 Constructing a Clause

The outermost loop of BETH is a simple set covering algorithm like that of any typical ILP algorithm: 1) find a good clause which covers a non-empty subset of positive examples, 2) remove the positive examples covered by the clause from the entire set of positive examples, 3) add the clause found to the set of clauses being built (a.k.a. theory) which was initially empty, 4) repeat step 1) to step 3) until the entire set of positive examples are covered by the theory, 5) return the entire set of clauses found.

The way a clause is constructed in BETH is very similar to a traditional top down ILP algorithm like FOIL; the search for a good clause goes from general to specific. It starts with the most general clause $\Box$ which is specialized by adding a literal to its body. The most general clause $\Box = T \leftarrow true$ where T is a literal such that predname(T) = predname(e) and arity(T) = arity(e), where e is a randomly chosen seed example from the set of positive examples. A beam of potentially good clauses is kept while searching for refinements of each clause in the beam. The construction of a clause terminates when there is a clause in the beam which is sufficiently accurate (i.e. its $m$-estimate is greater than or equal to a certain threshold).

In addition, we also compute the bottom clause which bounds the search space. The initial bottom clause is set to the smallest (i.e. $e \leftarrow true$ which has an empty set of literals in the body of this initial bottom clause and e is from the set of positive examples) in which case the search space contains only the most general clause (a.k.a. the empty clause). The bound is expanded incrementally during the search for a good clause. The bound is fixed when a sufficiently good
clause is found, at which point both the clause and the bound are returned as solutions to the search. The algorithm which constructs a clause is outlined in Figure 1.

1. Given a set of predicate specifications \( \mathcal{P} \) of the background predicates, a beam width \( b \), a bound on the clause length \( n \), variable depth bound \( i \), recall bound \( r \), a non-empty set of positive examples \( Pxs \) and a set (possibly empty) of negative examples \( Nxs \).
2. Randomly choose a seed example \( e \in Pxs \).
3. \( \bot_0 := e \leftarrow \text{true} \).
4. \( Q_0 := \{ \bot \} \).
5. \( Q := Q_0 \).
6. \( \bot := \bot_0 \).
7. REPEAT
   \begin{align*}
   & \text{generate refinements}(Q, \mathcal{P}, b, n, i, r, Pxs, Nxs, Q', \bot, \bot') \text{,} \\
   & Q := Q', \\
   & \bot := \bot',
   \end{align*}
UNTIL
there is a clause \( C \in Q \) which is sufficiently accurate, \( Q' \) and \( \bot' \) are output variables and the rest in \( \text{generate refinements} \) are input variables.
8. Return \( C \) and \( \bot \).

Fig. 1. The construction of a clause in BETH

3.2 Generating Refinements for a Clause

To find all the refinements of a given clause \( C_i \), first find a substitution \( \theta \), which satisfies the body of the clause (a “successful proof” of the clause, given the background facts); then construct a literal (with dummy variables) \( R_j \) for a predicate specification in \( \mathcal{P} \), and find a substitution \( \beta \) that makes \( R_j \beta \) a ground atom such that \( C_i \theta \) and \( R_j \beta \) satisfy the following conditions we call refinement constraints: 1) the link constraint: one of the arguments of \( R_j \beta \) has to appear in \( C_i \theta \) (this is to make sure that the resulting clause is still a linked clause), 2) the unique-literal constraint: \( R_j \beta \notin C_i \theta \) (to avoid making two identical literals). We try to find pairs of \( \theta \) and \( \beta \) satisfying the refinement constraints, but at most \( r \) distinct ground atoms \( R_j \beta \) will be used. For example, suppose \( C_i \theta = f(a, b) \leftarrow g(a, c), h(c, d), g(b, e) \) and \( R_j \beta_1 = g(e, f) \) and \( R_j \beta_2 = g(a, c) \), then only \( R_j \beta_1 \) satisfies all the refinement constraints, as \( R_j \beta_2 \) fails the unique-literal constraint. So, only \( R_j \beta_1 \) will be used to make literals for the clause \( C_i \).

To avoid repeatedly finding a successful proof of a given clause by theorem proving, we make a set of “cached proofs” for each clause in the beam (similar to the way variable bindings are stored in extensional FOIL) by starting with the initial proof \( e \leftarrow \text{true} \), where \( e \) is a randomly chosen seed example, and we incrementally update the cache of proofs of each clause by adding to the end of each proof a ground atom satisfying all the refinement constraints. A bound is also given to the cache size. When finding a satisfying substitution \( \theta \) for a clause \( C_i \) in the beam, we will simply unify \( C_i \) with a proof in its cache. If there is no \( R_j \beta \) satisfying the refinement constraints, which can happen if the first chosen example \( e \) was a “bad” one, a new example \( e' \neq e \) will be randomly chosen
from the remaining set of positive examples to be covered. The clause $C_i$ will be replaced (in the beam) by the most general clause such that its cache of proofs will contain only $e' \leftarrow \text{true}$. The idea is that if a clause cannot be refined, then we will just restart with a different seed example.

One can also take advantage of type declarations (if available) to further restrict the number of predicate specifications needed to be considered for a given clause — one needs only to consider those which contain at least one argument type which is the same as at least one of the types of all the variables in the current clause (so that a linked clause that satisfies the type constraints is possible).

One can also make use of mode declarations (if available) by substituting arguments with "input" mode for constants which appear in the clause, provided that the argument type and the constant type are the same (similar to the way the bottom clause is built in Progol). One needs to find satisfying substitutions for $R_j$, for each unique way of substituting arguments with input mode for constants in the clause. The algorithm for generating refinements to a clause is outlined in Figure 2.

1. Given a set of predicate specifications $\mathcal{P}$ of the background predicates, a beam width $b$, a bound on the clause length $n$, variable depth bound $i$, recall bound $r$, a non-empty set of positive examples $Pzs$ and a set (possibly empty) of negative examples $Nzs$, the current bottom clause $\bot$ (i.e. the current bound on the search space).
2. For each clause $C_i \in Q$ and for each $P_j \in \mathcal{P}$, make a literal $R_j$ with dummy variables such that $\text{predname}(R_j) \equiv \text{predname}(P_j)$ and $\text{arity}(R_j) \equiv \text{arity}(P_j)$.
3. Find substitutions $\theta, \beta$ such that 1) $\theta$ satisfies $C_i$, 2) $\beta$ satisfies $R_j$, and 3) $C_i \theta$ and $R_j \beta$ satisfy all the refinement constraints.
4. Collect at most $r$ such ground atoms $R_j \beta$ for different $\theta$ and $\beta$.
5. For each pair of $C_i \theta$ and $R_j \beta$ satisfying all the refinement constraints, make $\text{literals}(C_i \theta, R_j \beta, \text{Lits})$ and add $R_j \beta$ to the body of $\bot$.
6. For each $L \in \text{Lits}$, add $L$ to the body of $C_i$ to make $C'_i$ and let the set of all $C'_i$'s be $Q_i$.
7. Eulate each clause in $\bigcup_{C_i \in Q} C_i$ by a heuristic (e.g. m-estimate) given $Pzs$ and $Nzs$.
8. Put only the best $b$ clauses into $Q'$.
9. Let $\bot'$ be the resulting bottom clause after adding all the ground atoms $R_j \beta$'s to the body of $\bot$ for each $C_i$ and $R_j$ (such that there exists $\theta$ and $\beta$ satisfying all the refinement constraints).
10. Return $Q'$ and $\bot'$.

Fig. 2. Generate Refinements

### 3.3 Making Literals

The idea behind making literals given a clause $C$, a satisfying substitution $\theta$ of the clause, and a particular ground atom $R_j \beta$ is that we want to replace arguments in the ground atom by variables in the clause which are instantiated (in $\theta$) to these arguments in the ground atom, provided that the ground atom is not already part of $C \theta$ and the resulting literal observes the restriction on the variable depth bound.

For example, consider a clause $C : f(A,B) \leftarrow g(B,D), h(A,E), l(D,E)$, $\theta = \{A/a, B/b, D/h, E/e\}$ (thus, $\theta^{-1} = \{a/A, b/B, h/D, e/E\}$) and two ground
atoms \(a_1 = p(b,e)\) and \(a_2 = p(e,f)\). We can make two literals using \(a_1\): 1) \(p(B,E)\) (since \(b/B, e/E \in \theta^{-1}\)), and 2) \(p(D,E)\) (since \(b/D, e/E \in \theta^{-1}\)). We can make one literal using \(a_2\): \(p(E,F)\) (since \(e/E \in \theta^{-1}\), but the constant \(f\) is not bound to any variable in \(\theta^{-1}\)). However, if the variable depth bound is one, then the literal \(p(E,F)\) will be rejected because the depth of \(F\) is two. The variable depth \(d(V)\) of variable \(V\) is defined in Linus [6]. The algorithm for making literals is outlined in Figure 3.

1. Given a clause \(C\theta\) of the form \(e \leftarrow a_1, \ldots, a_n\) (where \(C\) is the current clause being refined, i.e. specialized, and \(\theta\) is a substitution that satisfies \(C\) and \(e \in Ps\) and background knowledge \(BK \models a_i\) for each ground atom \(a_i\) in the body of \(C\theta\)) and a ground atom \(a_{n+1}\) such that \(BK \models a_{n+1}.
2. Make a set of literals \(Lits\) such that each literal \(L \in Lits\) satisfies: 1) \(\text{prename}(L) = \text{prename}(a_{n+1})\), 2) \(\text{arity}(L) = \text{arity}(a_{n+1})\), 3) suppose the constant \(c_i\) is the \(i\)th argument of \(a_{n+1}\) and the variable \(V_i\) is the \(i\)th argument of \(L\). If \(c_i\) appears in \(C\theta\), then \(a_i / V_i \in \theta^{-1}\); otherwise, \(V_i\) is a new variable not appearing in \(C\), 4) there is no variable \(V\) in \(L\) such that \(d(V) > i\) where \(i\) is the variable depth bound.
3. Return \(Lits\).

**Fig. 3. Make Literals**

### 3.4 A Concrete Example

We can see how the algorithm works through a simple example from the family-relation domain. Suppose we want to learn the concept \(uncle(X,Y)\), which is true iff \(X\) is an uncle of \(Y\) (blood uncle).

Suppose we have the following set of background facts (Figure 4):

1. \(\text{male}(Bob), \text{male}(Tom), \text{male}(Tim)\)
2. \(\text{female}(Ann), \text{female}(Mary), \text{female}(Susan), \text{female}(Betty), \text{female}(Joyce)\)
3. \(\text{parent}(Tom, Mary), \text{parent}(Tom, Betty), \text{parent}(Tom, Bob), \text{parent}(Mary, Ann), \text{parent}(Joyce, Susan), \text{parent}(Tom, Tim)\)
4. \(\text{friend}(Mary, Susan), \text{friend}(Susan, Mary), \text{friend}(Joyce, Betty), \text{friend}(Betty, Joyce)\)

and \(P = \{\text{male}/1, \text{female}/1, \text{friend}/2, \text{parent}/2\}\) (exactly in this order from left to right) is our set of predicate specifications. We will use `'+'` to denote the output mode and `'-'` the input mode here. The following is the set of mode specifications for each predicate specification:

\(\text{male}(-), \text{female}(-), \text{parent}(+, +), \text{parent}(+, -), \text{friend}(+, -), \text{friend}(-, +)\)

and the following is the set of type specifications for each predicate specification:

\(\text{male(person), female(person), parent(person, person), friend(person, person)}\)

Suppose we have this set of training examples:

1. Positive: \(\text{uncle}(Bob, Ann)\)
Fig. 4. A simple family relation domain

2. Negative:
\[ \text{uncle}(Bob, Susan), \text{uncle}(Betty, Ann), \text{uncle}(Tim, Susan), \text{uncle}(Tom, Betty), \text{uncle}(Susan, Betty), \text{uncle}(Joyce, Ann), \text{uncle}(Tim, Joyce), \text{uncle}(Tom, Mary) \]

We present a trace of how our algorithm discovers a good clause, given a beam size and a recall bound of one, and a clause length of four. It starts by choosing a random seed example from the set of positive examples. This has to be \( \text{uncle}(Bob, Ann) \) since there is only one positive example. When generating refinements to a clause, it considers each predicate specification in \( P \) (from left to right). We will show the specialized clause before its set of cached proofs. The literal added to the clause currently being built is generated from the new ground atom added to the body of the cached proof of the current clause.

The algorithm starts with:

1. The most general clause which covers every pair of people: \( \text{uncle}(X,Y) :- \text{true} \)
2. The set of cached proofs for this clause: \{\( \text{uncle}(bob,ann) :- \text{true} \)\}
3. The empty bottom clause: \( \text{uncle}(bob,ann) :- \text{true} \)

It considers \textit{male/1} and generates the following:

1. The specialized clause: \( \text{uncle}(X,Y) :- \text{male}(X) \)  
   \( \text{m-est} = 0.153 \)
2. The set of cached proofs for this clause: \{\( \text{uncle}(bob,ann) :- \text{male}(bob) \)\}
3. The updated bottom clause: \( \text{uncle}(bob,ann) :- \text{male}(bob) \)

Next, the algorithm considers \textit{female/1}, and the literal \( \text{female}(Y) \) is generated (in the same way as \textit{male/1}), the new ground atom \( \text{female}(arm) \) is added to the current bottom clause. The specialized clause \( \text{uncle}(X,Y) :- \text{female}(Y) \)
has an $m$-estimate of 0.111. Next, it considers $parent/2$ (using $parent(+,-)$) and generates the following:

1. The specialized clause: $\text{uncle}(X,Y) \leftarrow \text{parent}(Z,X)$  
   ($m$-est = 0.136)

2. The set of cached proof of this clause: $\{\text{uncle}(\text{bob, ann}) \leftarrow \text{parent}(\text{tom, bob})\}$

3. The updated bottom clause:
   $\text{uncle}$(bob, ann) $\leftarrow$ male(bob), female(ann), parent(tom, bob)

Similarly $parent/2$ (using $parent(+,-)$) is used to generate another specialized clause $\text{uncle}(X,Y) \leftarrow \text{parent}(W,Y)$ ($m$-estimate = 0.122) using the ground atom $\text{parent}(\text{mary, ann})$.

The predicate specification $\text{friend/2}$ was considered but no ground atom was found to satisfy all the refinement constraints; the link constraint could not be satisfied, because neither Bob nor Ann has a friend. There are totally four different refinements to the most general clause. The clause with the best $m$-estimate is:

$$\text{uncle}(X,Y) \leftarrow \text{male}(X)$$

Since the beam size is just one, only this clause is retained in the beam. This clause is still covering negative examples: $\text{uncle}(\text{Bob, Susan})$, $\text{uncle}(\text{Tom, Betty})$, $\text{uncle}(\text{Tim, Susan})$, $\text{uncle}(\text{Tim, Joyce})$, and $\text{uncle}(\text{Tom, Mary})$. So, it still needs to be refined. Next, $\text{male}/1$ is considered but no ground atom is found to satisfy all the refinement constraints; the unique-literal constraint could not be satisfied ($\text{male}(\text{Bob})$ is already in the cached proof of the clause). The current bottom clause is

$$\text{uncle}$(bob, ann) $\leftarrow$ male(bob), female(ann), parent(tom, bob).

Next, it considers $\text{female}/1$ and generates the following:

1. The specialized clause: $\text{uncle}(X,Y) \leftarrow \text{male}(X), \text{female}(Y)$  
   ($m$-est = 0.153)

2. The set of cached proof of this clause:
   $\{\text{uncle}(\text{bob, ann}) \leftarrow \text{male}(\text{bob}), \text{female}(\text{ann})\}$

3. The updated bottom clause:
   $\text{uncle}$(bob, ann) $\leftarrow$ male(bob), female(ann), parent(tom, bob),  
   parent(mary, ann)

Next, it considers $parent/2$ (using $parent(+,-)$) and generates the following:

1. The specialized clause: $\text{uncle}(X,Y) \leftarrow \text{male}(X), \text{parent}(Z,X)$  
   ($m$-est = 0.204)

2. The set of cached proof of this clause:
   $\{\text{uncle}(\text{bob, ann}) \leftarrow \text{male}(\text{bob}), \text{parent}(\text{tom, bob})\}$

3. The updated bottom clause:
   $\text{uncle}$(bob, ann) $\leftarrow$ male(bob), female(ann), parent(tom, bob),  
   parent(mary, ann)

$parent/2$ (using $parent(+,-)$) can be used to generate another specialized clause $\text{uncle}(X,Y) \leftarrow \text{male}(X), \text{parent}(W,Y)$ ($m$-estimate = 0.175) using the ground atom $\text{parent}(\text{mary, ann})$.

The predicate specification $\text{friend/2}$ was considered but no ground atom was found to satisfy all the refinement constraints (the link constraint cannot
be satisfied). There are totally three different refinements to \( uncles(X,Y) \leftarrow male(X) \). The clause with the best \( m \)-estimate is:

\[
uncle(X,Y) \leftarrow male(X), parent(Z,X)
\]

This clause still covers a non-empty set of negative examples:

\( uncle(Bob, Susan), uncles(Tim, Susan), uncles(Tim, Joyce) \).

The algorithm continues in exactly the same manner for the last two steps (omitted to save space). The clause \( uncles(X,Y) \leftarrow male(X), parent(Z,X) \) has four different refinements. The clause with the best \( m \)-estimate is:

\[
uncle(X,Y) \leftarrow male(X), parent(Z,X), parent(W,Y)
\]

which is still covering a non-empty set of negative examples: \( uncles(Bob, Susan) \) and \( uncles(Tim, Susan) \).

There are totally eight different refinements to

\[
uncle(X,Y) \leftarrow male(X), parent(Z,X), parent(W,Y).
\]

The clause with the best \( m \)-estimate is:

\[
uncle(X,Y) \leftarrow male(X), parent(Z,X), parent(W,Y), parent(Z,W)
\]

which covers all the positive examples and no negative examples. At this point, the algorithm has found the target concept. Both the bottom clause discovered and the consistent clause found are returned. Notice that the bottom clause found by BETH is:

\[
uncle(bob, ann) :- male(bob), female(ann), parent(tom, bob), parent(mary, ann),
\]

\[
male(tom), female(mary), parent(tom, mary), friend(mary, susan),
\]

\[
friend(susan, mary)
\]

whereas, PROGOL’s bottom clause is:

\[
uncle(bob, ann) :- male(bob), female(ann), parent(tom, bob), parent(mary, ann),
\]

\[
male(tom), female(mary), parent(tom, mary), friend(mary, susan),
\]

\[
friend(susan, mary), female(susan)
\]

which is bigger than BETH’s bottom clause.

4 Analysis

Let \( \bot (b, n, P, r, i) \) be the bottom clause constructed by the algorithm outlined in Section 3 given the parameters \( b, n, P, r, \) and \( i \) which are the beam width, the maximum clause length, the set of predicate specifications, the recall bound, and the variable depth bound respectively.

**Theorem 1.** Suppose \( B \) is a beam of clauses produced by BETH, for any clause \( C \in B, C \preceq \bot (b, n, P, r, i) \).

**Proof.** Suppose \( C_j \) is a clause in \( B \) such that \( C_j = H \leftarrow L_1, \ldots, L_m \) where
Each $L_k$ is produced from some ground atom $a_k$ and $H$ is produced from a particular seed example $e$. Obviously, $e \in \bot(b, n, \mathcal{P}, r, i)$. For any $k : 1 \leq k \leq m$, $a_k \in \bot(b, n, \mathcal{P}, r, i)$, since each ground atom which satisfies all the refinement constraints is added to the current bottom clause and only ground atoms satisfying all the refinement constraints are used to make literals for any given clause. Therefore, there exists a substitution $\theta$ which satisfies $C_j$ such that

$$C_j \theta = e \leftarrow a_1, \ldots, a_m.$$  

Therefore, $C_j \theta \subseteq \bot(b, n, \mathcal{P}, r, i)$. In other words, we have $C_j \subseteq \bot(b, n, \mathcal{P}, r, i)$. Hence we have $C \subseteq \bot(b, n, \mathcal{P}, r, i)$ for any clause $C \in B$. □

**Theorem 2.** The worst case length of $\bot(b, n, \mathcal{P}, r, i)$ is $O(n|\mathcal{P}|r)$.

**Proof.** The maximum number of ground atoms that 1) satisfy the refinement constraints and 2) make literals observing the variable depth bound $i$ for any clause in the search beam at the point the clause is being refined are $|\mathcal{P}|r$. Therefore, the maximum number of ground atoms satisfying the refinement constraints after adding $n$ literals to the body of the most general clause are $n|\mathcal{P}|r$. Since there are at most $b$ clauses in the search beam at any time, the maximum number of ground atoms satisfying the refinement constraints are $bn|\mathcal{P}|r$. Thus, the worst case complexity of the bottom clause $\bot(b, n, \mathcal{P}, r, i)$ is $O(n|\mathcal{P}|r)$. □

The length of Progol’s bottom clause is $O((r|M[j^+=j^-]^j^+)^2)$ (which makes a hypothesis space doubly exponential w.r.t. $i$) while the length of Beth’s bottom clause is only linear w.r.t. $n$ (which gives rise to a much smaller hypothesis space).

## 5 Experimental Evaluation

We compared our system, Beth, with two other leading ILP systems — Aleph and mFoil.

### 5.1 Domain

After the events of 9/11, the EELD project has been working on several Challenge Problems that are related to counter-terrorism. The problem that we choose to tackle is the detection of Murder-For-Hires (contract killings) in the domain of Russian Organized Crime. The data used in all EELD Challenge Problems include representations of people, organizations, objects, and actions and many types of relations between them. One can picture this data as a large graph of entities connected by a variety of relations. For our purposes, we represent these relational databases as facts in Prolog.

For the ease of generating large quantities of data, and to avoid violating privacy, the program currently only uses synthetic data generated by a simulator. The data for the Murder-For-Hire problem was generated using a Task-Based (TB) simulator developed by Information Extraction and Transport Incorporated (IET). The TB simulator outputs case files, which contain complete and unadulterated descriptions of murder cases. These case files are then filtered for observability, so that facts that would not be accessible to an investigator are eliminated. To make the task more realistic, this data is also corrupted, e.g., by misidentifying role players or incorrectly reporting group memberships. This
filtered and corrupted data form the evidence files. In the evidence files, facts
about each event are represented as ground facts, such as:

\texttt{murder(Murder714)}
\texttt{perpetrator(Murder714, Killer186)}
\texttt{crimeVictim(Murder714, MurderVictim996)}
\texttt{deviceTypeUsed(Murder714, PistolCzech)}

The synthetic dataset that we used consists of 632 murder events. Each murder
event has been labeled as either a positive or negative example of a murder-
for-hire. There are 133 positive and 499 negative examples in the dataset. Our
task was to learn a theory to correctly classify an unlabeled event as either
a positive or negative instance of murder-for-hire. The amount of background
knowledge for this dataset is extremely large; consisting of 52 distinct predicate
names, and 681,039 background facts in all.

5.2 Results

The performance of each of the ILP systems was evaluated using 6-fold cross-
validation. The total number of Prolog atoms in the data is so large that running
more than six folds is not feasible.\textsuperscript{4} The data for each fold was generated by sepa-
rate runs of the TB simulator. The facts produced by one run of the simulator,
only pertain to the entities and relations generated in that run; hence the facts
each fold are unrelated to the others. For each trial, one fold is set aside for
testing, while the remaining data is combined for training. To test performance
on varying amounts of training data, learning curves were generated by test-
ing the system after training on increasing subsets of the overall training data.
Note that, for different points on the learning curve, the background knowledge
remains the same; only the number of positive and negative training examples
given to the system varies.

We compared the three systems with respect to accuracy and training time.
Accuracy is defined as the number of correctly classified test cases divided by
the total number of test cases. The training time is measured as the CPU time
consumed during the training phase. All the experiments were performed on a
1.1 GHz Pentium with dual processors and 2 GB of RAM. \textsc{Beth} and \textsc{mFoil}
were implemented in Sestus Prolog version 3.8.5 and \textsc{Aleph} was implemented in
Yap version 4.3.22. Although different Prolog compilers were used, the Yap Pro-
log compiler has been demonstrated to outperform the Sestus Prolog compiler,
particularly in ILP applications \cite{4}.

In our experiments, we used a beam width of 4 for \textsc{Beth} and \textsc{mFoil}; and
limited the number of search nodes in \textsc{Aleph} to 5000. We used $m$-estimate
($m = 2$) as a search heuristic for all ILP algorithms. The clause length was
limited to 10 and the variable depth bound to 5 for all systems. The recall bound
was limited to 1 for \textsc{Beth} and \textsc{Aleph} (except for some mode declarations it was

\textsuperscript{4} The maximum number of atoms that the Sestus Prolog compiler can handle is
approximately a quarter million.
Fig. 5. Performance of the systems versus the percentage of training examples given

set to *). We modified mFOIL to be constrained by the maximum clause length and the variable depth bound, to ensure that it terminates. We refer to this version of mFOIL as *Bounded* mFOIL. All the systems were given 1 second of CPU time to compute the set of examples covered by a clause. If a specialized clause took more time than allotted, the clause was ignored; although the time it took to create the clause is still recorded.

The results of our experiments are summarized in Figure 5. A snapshot of the performance of the three ILP systems given 100% of the training examples is shown in Table 2. The following is a sample rule learned by BETH:

```
murder_for_hire(A) :- murder(A), event0ccursAt(A,H),
    geographicalSubRegions(I,H), perpetrator(A,B),
    recipientOfInfo(C,B), senderOfInfo(C,D), socialParticipants(F,D),
    socialParticipants(F,G), payer(E,G), toPossessor(E,D).
```

This rule covered 9 positive examples and 3 negative examples. The rule can be interpreted as: *A* is a murder-for-hire, if *A* is a murder event, which occurs in a city in a subregion of Russia, and in which *B* is the perpetrator, who received information from *D*, who had a meeting with and received some money from *G*.

5.3 Discussion of Results

On the full training set, BETH trains 25 times faster than ALEPH while losing only 2 percentage points in accuracy and it trains twice as fast as mFOIL while gaining 3 percentage points in accuracy. Therefore, we believe that its integration
of top-down and bottom-up search is an effective approach to dealing with the problem of scaling ILP to large examples. The learning curves for training time further illustrate that although BETH and mFOIL appear to scale linearly with the number of training examples, ALEPH’s training-time growth is super-linear.

The large speedup over ALEPH is explained by the theoretical analysis on the complexity of the bounds on the search space, i.e. the different sizes of the bottom clauses they construct. The size of the bottom clause for BETH is only linear w.r.t. n compared to that of ALEPH which is exponential w.r.t. to i (i ≤ n) even for small i. As a result, ALEPH’s search space is much larger than BETH’s. ALEPH’s bottom clause was on average 119x larger than BETH’s and the total number of clauses it constructed was 14x larger, although a theory of similar accuracy was learned.

Systems like BETH and ALEPH construct literals based on actual ground atoms in the background knowledge, guaranteeing that the specialized clause covers at least the seed example. On the other hand, mFOIL generates more literals than necessary by enumerating all possible combination of variables. Some such combinations make useless literals; adding any of them to the body of the current clause makes specialized clauses that do not cover any positive examples. Thus, mFOIL wastes CPU time constructing and testing these literals. Since the average predicate arity in the EELD data was small (2), the speedup over mFOIL was not as great, although much larger gains would be expected for data that contains predicates with higher arity.

Nevertheless, searching a smaller space comes at the cost of spending more time generating each literal for refining a clause. In ALEPH, all the necessary ground literals are generated before the search starts, while BETH must spend time computing a set of ground atoms satisfying the refinement constraints on literal generation, resulting in fewer clauses tested per unit time compared to both ALEPH and mFOIL.

From the experimental results obtained, we can conclude that 1) an approach like BETH, which emphasizes searching a much smaller space over testing hypotheses at a higher rate, can outperform (in terms of efficiency) an approach like PROGOL/ALEPH, which trades off the two factors the other way around, and 2) using ground atoms directly avoids testing useless literals, improving training time over a purely top-down approach like mFOIL.

<table>
<thead>
<tr>
<th>System</th>
<th>Accuracy</th>
<th>CPU Time (mins)</th>
<th># of Clauses</th>
<th>Bottom Clause Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>BETH</td>
<td>94.80% (+/- 2.3%)</td>
<td>23.39 (+/- 4.26)</td>
<td>4483</td>
<td>34</td>
</tr>
<tr>
<td>ALEPH</td>
<td>96.91% (+/- 2.8%)</td>
<td>598.92 (+/- 250.00)</td>
<td>63334</td>
<td>4061</td>
</tr>
<tr>
<td>mFOIL</td>
<td>91.23% (+/- 4.8%)</td>
<td>45.28 (+/- 5.40)</td>
<td>112904</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table 2. Results on classifying murder-for-hire events given all the training data. # of Clauses is the total number of clauses tested; and Bottom Clause Size is the average number of literals in the bottom clause constructed for each clause in the learned theory. The 90% confidence intervals are given for test Accuracy and CPU time.
6 Conclusions

An important under-studied aspect of scaling to large databases in multi-relational data mining concerns the size of examples rather than their number. For ILP methods, this issue involves scaling to large amounts of connected background knowledge associated with each example or set of examples. We have developed a new ILP algorithm that integrates top-down and bottom-up search in order to more efficiently learn in the presence of large amounts of background knowledge. Challenge problems constructed for DARPA’s program on Evidence Extraction and Link Discovery concern identifying potential threatening activities in large amounts of heterogeneous, multi-relational data. These problems contain relatively modest numbers of examples but involve large amounts of background knowledge. Experimental results on these problems demonstrate that our new hybrid approach substantially decreases training time compared to existing ILP methods.

7 Acknowledgments

This research is sponsored by DARPA and managed by Rome Laboratory under contract F30602-01-20571. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied of the Defense Advanced Research Projects Agency, Rome Laboratory, or the United States Government.

References