Learning Recursive Relations with Randomly Selected Small Training Sets

David W. Aha
Navy AI Center
Naval Research Laboratory
Washington, DC
20375 USA
aha@aic.nrl.navy.mil

Stephane Lapointe
DREV
P.O. Box 8800
Courcelette, Quebec
G0A 1R0 Canada
lapointe@drev.dnd.ca

Charles X. Ling
Dept. of Computer Science
Univ. of Western Ontario
London, Ontario
N6A 5B7 Canada
ling@csd.uwo.ca

Stan Matwin
Dept. of Computer Science
University of Ottawa
Ottawa, Ontario
K1N 6N5 Canada
stan@csi.uottawa.ca

Abstract

We evaluate CRUSTACEAN, an inductive logic programming algorithm that uses inverse implication to induce recursive clauses from examples. This approach is well suited for learning a class of self-recursive clauses, which commonly appear in logic programs, because it searches for common substructures among the examples. However, little evidence exists that inverse implication approaches perform well when given only randomly selected positive and negative examples. We show that CRUSTACEAN learns recursive relations with higher accuracies than GOLEM, yet with reasonable efficiency. We also demonstrate that increasing the number of randomly selected positive and negative examples increases its accuracy on randomly selected test examples, increases the frequency in which it outputs the target relation, and reduces its number of outputs. We also prove a theorem that defines the class of logic programs for which our approach is complete.

1 MOTIVATION

This paper extends our previous work (Aha, Lapointe, Ling, & Matwin, 1994), which introduced an inductive logic programming (ILP) algorithm and its implementation in CRUSTACEAN. In that paper, we sketched a proof for a theorem that describes the class of relations for which our algorithm is complete, reported on CRUSTACEAN's ability to induce relations given manually selected inputs, and briefly summarized initial results of its learning behavior with randomly selected inputs. In this paper, we provide the proof for our theorem (Section 2.3), but focus on more extensive empirical analyses with randomly selected inputs (Section 3) after we added constraints (Section 2.2) that greatly increased CRUSTACEAN's efficiency. We describe evidence in Section 3 for the following hypotheses:

1. CRUSTACEAN's learning rates are faster than the learning rates of a system that does not have its strong target language bias.
2. Accuracy increases with increasing numbers of positive examples.
3. Accuracy increases with increasing numbers of negative examples.
4. The percentage of runs in which CRUSTACEAN outputs the target clause increases with more positive examples.
5. The number of relations output decreases with increasing numbers of examples.

Our initial reason for developing CRUSTACEAN was to extend LOPSTER (Lapointe & Matwin, 1992), which inputs two examples and outputs a recursive clause. It requires that one of these examples is the base clause of the recursive relation, or that they are different clauses on the same resolution chain. Thus, LOPSTER's users must know either the relation's base clause or recursive clause (i.e., so that they can select examples on the same resolution chain). CRUSTACEAN, a generalization of LOPSTER, avoids this problem by explicitly searching for a base clause. Thus, it shifts burden from the user to the algorithm, and in doing so increases the complexity of the learning task. Section 2 describes constraints, and their effects, that assist in reducing its search for recursive relations.

Most empirical studies of ILP algorithms focus on the types of relations that can be learned by an algorithm and the speed in which it learns them. Most such studies involve a dataset whose examples are hand-picked, which raises at least three concerns. First, the experimentor may have chosen inputs that were tailored to allow their algorithm to perform (comparatively) well. Second, this approach yields no insight on how the algorithm performs when the inputs are selected according to more realistic distributions. Third, such experiments lend little insight concerning the algorithm's learning rate (i.e., the number of examples it requires to learn the target relation). In this paper, we address these concerns by experimenting with randomly selected inputs. Cohen (1993) was perhaps the first to report experiments on an ILP algorithm (FORCE2) using randomly selected inputs. FORCE2 inputs, in addition to the examples, functions that bound the target relation's complexity and distinguish positive examples according to whether they instantiate the target base clause. This additional information assists in reducing the complexity of the learning task, and FORCE2's comparative results with FOIL are impressive. CRUSTACEAN instead inputs only the positive and negative examples.

Inverse implication approaches are often computationally expensive (Muggleton, 1992; Idestam-Almquist, 1993). This discourages their implementation and empirical evaluation. However, we will show that CRUSTACEAN has reasonable computational costs for small numbers of training examples.

Our final motivation for this study is to help justify the development ILP algorithms based on inverse implication. These algorithms should learn self recursive target relations more quickly than general ILP algorithms based on θ-subsumption because they search for shared substructure information among examples that can help define the recursive clause. We examine this in Section 3, where we provide evidence that, for a class of recursive clauses, CRUSTACEAN attains higher accuracies than GOLEM (Muggleton & Feng, 1990) when given small numbers of training instances.

Section 2 briefly reviews CRUSTACEAN, highlights the constraints it uses to attain reasonable efficiency, and includes the proof for our completeness theorem. Section 3 then summarizes our empirical investigations and discusses the results. Sections 4 describes related work and Section 5 summarizes our conclusions and some topics for future research.

2 Review of CRUSTACEAN

2.1 DESCRIPTION

Table 1 describes an idealized version of CRUSTACEAN's control algorithm and I/O behavior. CRUSTACEAN inputs a set of positive and a set of negative examples of a predicate, all of which are represented as Horn clauses. It outputs a set of singly recursive relations LP=(BC,RC) with base clause BC and recursive clause RC such that

1. LP resolves with each positive example and no negative example and
2. at least one of the positive example is not θ-subsumed by BC (i.e., RC needs to be used at least once to prove one of the positive examples).

Table 1: CRUSTACEAN inputs positive and negative examples and outputs a set of recursive relations

```
 Given: P: a set of positive examples
        N: a set of negative examples

CRUSTACEAN(P,N)
 1. Relations ← Induce_relations(P,N)
 2. Output: Filter_relations(Relations,P,N)

Induce_relations(P,N)
 1. S ← Subterms(P)
 2. G ← Generating_terms(S)
 3. C ← Combinations(S,G)
 4. ∀c ∈ C :
  IF Legal_match(c) THEN
      b ← Base_clause(c)
     IF Inconsistent(b,N) THEN
         r ← Recursive_clause(c)
         Relations ← Relations ⋃ {⟨b,r⟩}
 5. Output: Relations

Filter_relations(Relations,P,N)
 1. Discard all redundant relations
 2. Discard all infinitely recursive
    relations
 3. Discard all relations that resolve
    with any n ∈ N
 4. Discard all relations that cannot
    resolve with all p ∈ P
 5. Output: Remaining relations
```

It's target language SR is defined as the family of singly recursive logic programs LP=(BC, RC) of the following syntactic form:

$$\begin{aligned} BC &= p(a_1, a_2, ..., a_n). \\ RC &= p(b_1, b_2, ..., b_n) \leftarrow p(c_1, c_2, ..., c_n). \end{aligned}$$

where p is a predicate symbol, a_i, b_i, and c_i are arbitrarily complex terms, and, for every i $(1 \leq i \leq n)$, c_i is a subterm of b_i, where they differ for at least one i.

Induce_relations first computes all the *subterms* of each example (Step 1). It then computes, for each subterm, its set of *generating terms* (Step 2), which describe how the subterm was obtained from the example's argument. A subterm could be generated in more than one way. For example, the sublist beginning with the third element of a given list could be obtained by a single application of two list decompositions, or by two applications of list decomposition. We refer to the number of applications of a generating term as its *depth*, which is one and two respectively for these two generators.

Step 3 creates combinations of subterms. For each example, all possible lists of subterm/generating term pairs are created, where one pair is selected for each argument. However, a list is kept only if the depth is the same for every generating term in the list. Each of these lists is then matched with each such list created from the other examples. In Step 4, Legal_match keeps only those combinations for which, for every argument, the generating terms match across all examples (see Section 2.2). A base clause is induced for each matched list. Base_clause(c) outputs the lgg of the examples' list of subterms. If a base clause does not resolve with any negative example, then Recursive_clause is executed to compute its recursive clause. The arguments in the head of the recursive clause are determined by computing lggs from the generating term decompositions of the positive examples, where the number of decompositions each example contributes is equal to the depth of its generating terms. See (Aha, Lapointe, Ling, & Matwin, 1994) for more details, including a detailed example.

2.2 CONSTRAINTS

CRUSTACEAN's complexity is dominated by its search through the space of generating term combinations. If s is the maximum number of subterms per term, a is the arity of the target relation, and p is the number of positive examples, then the size of this space is $O(s^{a^p})$. Other algorithms based on inverse implication also suffer from doubly exponential complexities (e.g., Muggleton, 1992; Idestam-Almquist, 1993).

Constraints are applied to reduce the amount of search required through this space. Their explanation requires some understanding of Legal_match. Two lists of generating terms match if their corresponding pairwise generating terms match for each argument. Two generating terms match if they denote the same pattern of decomposition. Also, any list whose depth is zero (i.e., its example is an instantiation of the induced base clause) matches any other list. Three constraints can be applied to reduce search through this space of subterm combinations:

1. **Example constraint**: The depths of the generating terms must be the same across all arguments of a given example (i.e., the same number of recursive decompositions must be applied to all arguments of a given example before it is reduced to an instantiation of the base clause).
2. **Argument constraint**: For a given argument, the decomposition pattern must be the same for all positive examples.
3. **Depth constraint**: At least one of the positive examples must have a generating depth d where $d \geq 2$ (i.e., to help prevent outputting overly-specific recursive clauses).

These constraints greatly reduce the amount of time required to locate matched lists of generating terms. For example, when tested on the manually-selected inputs (see Aha, Lapointe, Ling, & Matwin, 1994) for the ten relations listed in Table 2, these constraints reduced average summed cpu times by 93.2%. Applying these constraints reduces the space of combinations that satisfy Legal_match to 1% of its original size (i.e., 143 of 14225).

2.3 COMPLETENESS THEOREM

We (Aha, Lapointe, Ling, & Matwin, 1994) previously sketched a proof showing that our algorithm is complete for SR (i.e., it induces all *least general* logic programs of SR that are consistent with the training examples). Here we present its proof, which uses the notion of n^{th} powers and roots (Muggleton, 1992).

Theorem 1. *Given a set of positive examples P and a set of negative examples N,* CRUSTACEAN *will output every logic program LP=(BC,RC) in SR with the following properties:*

1. *LP is complete and consistent with respect to P and N, and there exists a $p \in P$ and a $d \geq 1$ such that BC θ-subsumes the resolvent of p with the d^{th} power of RC.*
2. *LP is a lgg under implication of P, or recursive clause RC is the least general d^{th} root of RC' for a given $d \geq 2$, where LP'=(BC,RC') is a lgg under implication of P.*

Proof of Theorem 1:
We first prove that CRUSTACEAN will output the base clause BC and then prove that it will also output the recursive clause RC.

Property 1 ensures that RC must be used at least once to prove at least one of the positive examples. Let LP=(BC,RC) be a logic program in SR that satisfies Properties 1 and 2 and defines a predicate ps with arity n. Let $\{p_1, p_2, \ldots, p_k\}$ be the set of all $p \in P$ for which there exist $d_i \geq 1$ such that BC θ-subsumes BC_i, where BC_i is the resolvent of p_i with the d_i^{th} power of RC (computed by Recursive_clause(c) in Table 1). From Property 1, $k \geq 1$. Since LP is complete with respect to P, we deduce that any $p_i \in \{p_{k+1}, p_{k+2}, \ldots, p_m\} = P - \{p_1, p_2, \ldots, p_k\}$ is θ-subsumed by BC. Let BC' = lgg(BC_1, BC_2,..., BC_k, p_{k+1}, p_{k+2},..., p_m). CRUSTACEAN builds potential base clause BC' using Base_clause (Table 1) with subterms $\mathrm{BC}_{i,j}$ $(1 \leq i \leq k)$ and $p_{i,j}$ $(k+1 \leq i \leq m)$, for every argument j $(1 \leq i \leq n)$. By construction, BC θ-subsumes BC'. Suppose BC' does not θ-subsume BC. Then LP=(BC,RC) is not a least general generalization of P since (BC', RC) is more specific than LP and complete with respect to P. Moreover, there exists no RC_1 such that RC is a non-trivial root of RC_1 and (BC,RC_1) is a least general generalization of P, since (BC',RC_1) is more specific than (BC,RC_1). This contradicts Property 2. Therefore, BC' θ-subsumes BC. We then conclude BC' = BC (up to a variable renaming). Thus, CRUSTACEAN will induce BC.

We now prove that, given the subterms considered above, CRUSTACEAN will output RC. Let A_j be the arguments of the head of RC and B_j be the arguments of its recursive literal. Based on the syntactic form of RC (see definition of SR), we define $A_j - B_j$ as the difference between both terms. For example, if A_j = f(a,f(b,X)) and B_j = f(b,X) then $A_j - B_j$ = f(a,Y). Let $GT_j = A_j - B_j$, for every $1 \leq j \leq n$, be a term decomposition operator (computed by Generating_terms(S)). Let $ET_{i,j}$ be the embedding term needed to build term $p_{i,j}$ from term $BC_{i,j}$, for every $1 \leq i \leq k$ and $1 \leq j \leq n$ (computed by Subterms(P)). Let $ET_{i,j,d}$ be the embedding term $ET_{i,j}$ decomposed d times ($0 \leq d \leq d_i - 1$) by the operator GT_j. Let A_j' = lgg($ETD_{1,j}$, $ETD_{2,j}$, ..., $ETD_{k,j}$) for every $1 \leq j \leq n$, where $ETD_{i,j} = \{ET_{i,j,0}, ET_{i,j,1}, \ldots, ET_{i,j,d_i-1}\}$, for every $1 \leq i \leq k$ (computed by Recursive_clause(c)). Let B_j' be the term A_j' decomposed *once* by the operator GT_j. Let RC' = ps(A_1', A_2',..., A_j') ← ps(B_1', B_2', ..., B_j'). By construction, A_j θ-subsumes A_j' and B_j θ-subsumes B_j', for every $1 \leq j \leq n$. Therefore, RC θ-subsumes RC'. When considering generating terms GT_j, for every $1 \leq j \leq n$, CRUSTACEAN will build the recursive clause RC'. Suppose RC' does not θ-subsume RC. Then LP = (BC,RC) is not a least general generalization of P since (BC, RC') is more specific than LP and complete with respect to P. Moreover, there exists no RC_1 and $d \geq 2$ such that RC is the least general d^{th} root of RC_1, since RC' is a more specific d^{th} root of RC_1. This contradicts Property 2. Therefore, RC' θ-subsumes RC. We then conclude RC' = RC (up to a variable renaming). This proves that CRUSTACEAN will induce RC.

Since LP is complete and consistent with respect to P and N (Property 1, guaranteed by Steps 3 and 4 of Filter_relations), CRUSTACEAN's filters will not reject it and it will be output. ■

CRUSTACEAN's output consists of all the least general logic programs of SR under implication and all their least general n^{th} roots ($n > 1$). Since the θ-subsumption notion of generality is only incomplete for self-recursive clauses (i.e., for expressing generality relations between different powers or roots of a clause) (Muggleton, 1992), no output is more general under θ-subsumption than any other output, although one may be more general than another under implication.

3 EVALUATION

We empirically evaluated CRUSTACEAN's ability to induce target relations when given randomly selected positive and negative examples. The hypotheses explored were summarized in Section 1.[1] Our experiments are with the ten relations whose target clauses are detailed in Table 2. The instance space was defined as the set of examples such that no argument had recursive structures of depth greater than four. Such arguments (e.g., lists, successors) were generated randomly according to a uniform distribution on structure depth in the range $[0, 4]$. Constant terms and list subterms were randomly generated from a uniform distribution on the 26 lower case letters. Relation-dependent constraints were used to ensure that all generated instances were correctly labeled as positive or negative. For example, when generating a positive example for the *member* relation, the first argument was selected randomly from the list's elements. All sampling was done with replacement.

[1] Initial evidence for Hypotheses 2 and 5 were discussed in (Aha, Lapointe, Ling, & Matwin, 1994).

Table 2: Target relations used in the experiments

append([],A,A). append([A|B],C,[A|D]) :- append(B,C,D).
delete(A,[A|B],B). delete(A,[B,C|D],[B|E]) :- delete(A,[C|D],E).
extractNth(s(0),[A|B],A). extractNth(s(s(A)),[B,C|D],E) :- extractNth(s(A),[C|D],E).
fact(0,s(0)). fact(s(0),s(0)). fact(s(s(A)),s(s(A))*B) :- fact(s(A),B).
last_of(A,[A]). last_of(A,[B,C|D]) :- last_of(A,[C|D]).
member(A,[A|B]). member(A,[B,C|D]) :- member(A,[C|D]).
noneIsZero([]). noneIsZero([s(A)|B]) :- noneIsZero(B).
plus(0,A,A). plus(s(A),B,s(C)) :- plus(A,B,C).
reverse([],[]). reverse([A|B],append(C,[A])) :- reverse(B,C).
split([],[],[]). split([A,B|C],[A|D],[B|E]) :- split(C,D,E).

Most ILP systems are typically evaluated according to whether they output the *exact* target relation, which is a central reason why hand-picked training examples are often used. Since we instead use randomly selected training examples, we can no longer expect this to always occur. Instead, we expect that, as more examples are provided, both the prediction accuracy on randomly generated test examples and the frequency with which the target relations are output will increase. We believe that this evaluation methodology is more realistic and critical for ILP systems.

Accuracy of an output (i.e., a paired base and recursive clause) is defined as its percentage of correct classifications for a given test set. The accuracy of a set of outputs is their average accuracy. We report this average itself averaged across ten runs for each relation, where a different training set is generated for each run. All test sets included 100 randomly selected instances, where positive and negative instances were generated with equal probability.

Hypothesis #1: CRUSTACEAN**'s learning rates for relations in SR are faster than the learning rates of systems that do not have** CRUSTACEAN**'s strong target language bias.** This first hypothesis explores whether the algorithm underlying CRUSTACEAN is of any use. CRUSTACEAN embodies strong constraints on the target language, as detailed in Section 2.3. Most popular ILP systems can learn larger classes of relations. Therefore, this restriction is only useful if it yields some comparative performance improvement. We compared CRUSTACEAN with GOLEM (Muggleton & Feng, 1990), a popular ILP system. We fixed the number of negative inputs to ten and varied the number of positive examples from between two and five. Table 3 lists the average results when using five positive examples.[2] These results strongly support our hypothesis; the approach embodied in CRUSTACEAN allows it to attain higher accuracies with fewer randomly selected instances. We found similar differences in accuracies when using two, three, and four positive instances.

Hypothesis #2: Accuracy increases with more positive examples. We fixed the number of negative input examples to ten and varied the number of positive examples between two and six. We found strong evidence for our hypothesis; accuracy increased monotonically with the number of positive examples. Table 4 displays the results for the extremes of our range. These accuracies are imperfect because the randomly generated examples can share structure or constants that yield overly-specific outputs. Also, the randomly generated negative examples do not always resolve with the overly general outputs.

Hypothesis #3: Accuracy increases with more negative examples. Using only four positive input examples, we varied the number of negative inputs between 2 and 25. Table 5 displays the results for the extrema. These results again provide strong evidence for our hypothesis. Negative examples help to eliminate overly general outputs, which increases accuracy.

Hypothesis #4: CRUSTACEAN **outputs the target clause more frequently with more positive examples.** We found some evidence for this hypothesis in the experiment whose results are described in Table 4. For this same experiment, Table 6 lists the percentage of runs in which the correct target relation is among CRUSTACEAN's outputs. In general, increases occurred because overly specific relations are not as frequently consistent with larger sets of randomly selected positive examples. The percentage of runs containing the target output increased or remained the same for eight of the ten relations, and decreased substantially for *factorial*, whose target relation is not in SR (i.e., it has two base clauses). Table 6 reports the percentage of times that *factorial*'s second base clause and recursive clause is output. However, whenever the positive example "fact(0,s(0))" is given as input, CRUSTACEAN overgeneralizes the base clause's first argument when it computes lggs with the other inputs, yielding "fact(A,s(0))," which we didn't consider to be the correct target relation.

Table 3: Average accuracies for GOLEM and CRUSTACEAN when given five randomly selected positive examples and ten randomly selected negative examples

Name	GOLEM	CRUSTACEAN
append	74.2%	86.0%
delete	66.8%	85.3%
extractNth	60.5%	88.2%
last_of	53.9%	97.6%
member	68.0%	93.7%
noneIsZero	63.6%	92.8%
plus	50.7%	87.3%
split	76.2%	94.2%

Table 4: CRUSTACEAN's average accuracies (Acc) and number of outputs (#Out) given two or six randomly selected positive examples with ten randomly selected negative examples

Name	2 Pos Inputs		6 Pos Inputs	
	Acc	#Out	Acc	#Out
append	60.0%	7.8	86.8%	7.1
delete	64.5%	9.2	91.5%	6.6
extractNth	59.3%	8.9	88.1%	2.3
factorial	83.4%	5.8	97.0%	1.3
last_of	89.1%	1.8	97.8%	1.8
member	70.9%	1.8	93.7%	1.0
noneIsZero	75.3%	3.0	94.6%	2.7
plus	69.7%	6.1	93.3%	2.9
reverse	79.0%	5.0	98.9%	1.2
split	69.2%	11.8	94.2%	3.5

[2]The results for two of the relations are missing because it is unfair to require GOLEM to use the representations for the examples that we used with CRUSTACEAN. Similarly, it is unfair to give GOLEM the typical background knowledge it normally requires to induce rules for those relations since CRUSTACEAN cannot yet process such background knowledge.

Table 5: CRUSTACEAN's average accuracies (Acc) and number of outputs (#Out) given four randomly selected positive examples with either two or 25 randomly selected negative examples

Name	2 Neg Inputs		25 Neg Inputs	
	Acc	#Out	Acc	#Out
append	70.7%	12.3	84.8%	4.8
delete	75.6%	25.7	83.7%	4.8
extractNth	65.8%	21.0	88.9%	1.7
factorial	86.6%	3.4	91.7%	1.3
last_of	95.8%	1.9	97.8%	1.5
member	77.3%	3.3	92.5%	1.1
noneIsZero	82.4%	5.7	88.7%	2.4
plus	77.1%	8.9	84.4%	4.0
reverse	91.9%	4.4	95.9%	2.3
split	81.3%	12.0	87.2%	4.7

Table 6: Percentage of runs in which CRUSTACEAN output the target relation, where the inputs are two or six randomly selected positive examples with ten randomly selected negative examples

Name	2 Pos Inputs	6 Pos Inputs
append	30%	90%
delete	50%	60%
extractNth	10%	70%
factorial	100%	30%
last_of	30%	40%
member	10%	70%
noneIsZero	60%	100%
plus	40%	100%
reverse	100%	100%
split	70%	60%

Hypothesis #5: The number of outputs decreases with additional examples. This also decreased monotonically with increasing numbers of positive (Table 4) and negative input examples (Table 5). This occurred because the only lists of generating terms that can match for $n+1$ positive examples must be a subset of those that match for the first n examples, and additional negative examples assist in filtering overly-general relations.

In summary, our experiments provided confirming evidence for our hypotheses; CRUSTACEAN has comparably fast learning rates for relations in SR, its accuracy increases with more randomly selected positive and negative examples, the target clause is output more frequently with more positive examples, and the number of outputs decreases with more examples.

CRUSTACEAN's search constraints (Section 2.2) proved effective. Table 7 displays its average cpu times when given two through four positive and ten negative examples, plus the number of combinations and matches found for four positive examples. While these (Sun SPARCstation 10, QProlog) runtimes still increase exponentially with the number of positive examples, these constraints allowed for reasonable runtimes. We plan to analyze the utility of additional pragmatic constraints.

4 RELATED WORK

This paper extends (Aha, Lapointe, Ling, & Matwin, 1994) by empirically evaluating several hypotheses using randomly selected inputs and by providing a proof for our theorem. CRUSTACEAN is derived from LOPSTER (Lapointe & Matwin, 1992). LOPSTER and Muggleton's (1992) theoretically complete algorithm for inverse implication require that the two input examples be on the same resolution chain. This constraint is removed in CRUSTACEAN.

Cohen (1993) described one of the few evaluations of an ILP algorithm (FORCE2) using randomly selected inputs, and also proved it is a PAC-learning algorithm. FORCE2's class of learnable theories and techniques are similar to CRUSTACEAN's. It is a model-driven (enumeration) system, while CRUSTACEAN is data-driven. Cohen reported that FORCE2 has faster learning rates than FOIL (Quinlan, 1991), and that its speed scales linearly with the number of examples. FORCE2 inputs, in addition to the examples, knowledge concerning which examples are instantiations of the base clause, an upper bound on the target relation's depth complexity, and a depth bound i (i.e., it outputs ij-determinate logic programs). CRUSTACEAN is not given this information, and could presumably benefit from it.

Our work also relates to previous work on program synthesis from examples (e.g., Kodratoff & Jouannaud, 1984)). This involves transforming an input-output sequence $\{x_i, F(x_i)\}$ into computational traces of the form $\{p_i(x), f_i(x)\}$, where $p_i(x)$ is a predicate and is true if x has the same structure as x_i. The *fragment* $f_i(x)$ describes how each atom in $F(x_i)$ is constructed from a combination of basic functions. After this transformation, a search is conducted to find a recursive relation between f_i and f_{i+1}. If a consistent recursion is found (for all i), then $f_{i+1}(x)$ is defined by $f_i(x)$. In ILP, there are often no specific input or output arguments in predicates, and "fragments" are represented by term structures. Also, there is normally no "ordering" of positive examples. Seeking recursive relations between f_i and f_{i+k} is similar to how LOPSTER searches for subterm relations be-

Table 7: Average cpu times (in seconds) for two, three, and four positive and ten negative examples, plus the average number of subterm combinations (Comb) and number of those whose lists of generating terms match (Match) for four positive and ten negative examples

Relation	Cpu Times			Comb	Match
	2	3	4		
append	0.7	1.3	3.0	2.0×10^8	409.8
delete	0.8	2.1	8.0	8.8×10^6	875.7
extractNth	0.7	1.2	3.3	3.0×10^5	465.1
factorial	0.5	0.4	1.0	1.2×10^6	150.8
last_of	0.2	0.4	0.8	1.9×10^3	202.8
member	0.2	0.4	1.0	2.3×10^3	244.0
noneIsZero	0.3	0.6	1.0	3.6×10^4	175.4
plus	0.7	1.4	6.4	4.5×10^5	736.3
reverse	0.6	0.9	2.3	1.9×10^7	328.1
split	0.8	1.1	2.5	1.6×10^8	296.6

tween two positive examples on the same resolution chain using a form of inverse implication.

5 CONCLUSION

We evaluated CRUSTACEAN, an inverse implicator that induces recursive relations from examples. We characterized the class SR of functions for which CRUSTACEAN is proved to be complete. We focused on CRUSTACEAN's empirical behavior with randomly selected examples, and described evidence that its strong target language bias does indeed allow it to attain higher learning rates than an algorithm without this bias, that its accuracy increases with increasing examples, that it outputs the target clause more frequently with more positive examples, and that its number of outputs decreases with more examples.

Our approach shifts the burden of knowledge from the user to the algorithm. CRUSTACEAN's users do not need to know the base clause, as required by LOPSTER, the recursive clause, as (implicitly) required for most approaches based on θ-subsumption, or knowledge that distinguishes instantiations of base clauses from other examples, as required by FORCE2. The trade off is that CRUSTACEAN's algorithm is more complex. While some constraints greatly improve its behavior, more research is needed to investigate how it can exploit other pragmatic constraints.

CRUSTACEAN would be useful for inducing the definition of new predicates since, in most cases, only a few new predicate instances can be collected (Lapointe, Ling, & Matwin, 1993). Inverse implication approaches should prove useful for such tasks.

Acknowledgements

Thanks to Cao Feng for his help in running experiments with GOLEM, to William Cohen and our reviewers for their useful comments, and to NSERC for its support for the last two authors.

References

Aha, D. W., Lapointe, S., Ling, C. X., & Matwin, S. (1994). Inverting implication with small training sets. In *Proceedings of the 1994 European Conference on Machine Learning* (pp. 31–48). Catania, Italy: Springer Verlag.

Cohen, W. (1993). Pac-learning a restricted class of recursive logic programs. In *Proceedings of the Eleventh National Conference on Artificial Intelligence* (pp. 86-92). Washington, DC: AAAI Press.

Idestam-Almquist, P. (1993). Recursive anti-unification. In *Proceedings of the Third International Workshop on Inductive Logic Programming* (pp. 241–254). Bled, Slovenia: J. Stefan Institute.

Kodratoff, Y, & Jouannaud, J.-P. (1984). Synthesizing LISP programs working on the list level of embedding. In A. Biermann, G. Guiho, & Y. Kodratoff (Eds.) (1984). *Automatic program construction techniques.* New York: Macmillian Publishing Company.

Lapointe, S., Ling, X. C., & Matwin, S. (1993). Constructive inductive logic programming. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence.* Chambery, France: Morgan Kaufmann.

Lapointe, S., & Matwin, S. (1992). Sub-unification: A tool for efficient induction of recursive programs. In *Proceedings of the Ninth International Conference on Machine Learning* (pp. 273–281). Aberdeen, Scotland: Morgan Kaufmann.

Muggleton, S. (1992). Inverting implication. In *Proceedings of the First European Workshop on Inductive Logic Programming.* Vienna, Austria: Unpublished.

Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. *Proceedings of the First International Workshop on Algorithmic Learning Theory* (pp. 368–381). Tokyo, Japan: Japanese Society for AI.

Quinlan, J. R. (1991). Determinate literals in inductive logic programming. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence* (pp. 746–750). Sydney, Australia: Morgan Kaufmann.