

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]



Université d'Ottawa • University of Ottawa

LEARNING RELATIONAL CLICHÉS WITH CONTEXTUAL GENERALIZATION

© Johanne Morin

Thesis
Submitted to the School of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

June, 1999

Ottawa-Carleton Institute for Computer Science
School of Information Technology and Engineering
University of Ottawa
Ottawa, Ontario, Canada



National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-46535-7

Canada

ABSTRACT

Inductive logic programming (ILP) is concerned with the problem of inducing concepts represented as logic programs (or Horn clauses) from examples. Top-down inductive learners such as FOIL (Quinlan 1990; Cameron-Jones *et al.* 1993) learn Horn clauses adding one literal at a time using a hill-climbing search. These learners suffer from local plateaus, where the selection of a conjunction of literals, rather than a single literal, would improve the accuracy of the clause. The problem becomes the search for combinations of literals rather than just single literals. A mechanism to search efficiently through the space of combinations of literals is needed. The FOCL system (Pazzani *et al.* 1991) solved this problem by giving the concept learner hand-made “relational clichés” which are combinations of literals to consider while learning. The problem is that these clichés are hard to derive and often specific to a domain. So, it would be desirable to learn them automatically.

As a part of this thesis, an inductive learner called CLUSE (Clichés Learned and USEd) has been developed that learns combinations of literals called relational clichés. The underlying idea is to learn clichés from examples of a concept and to use them with a hill-climbing learner to escape local plateaus. Clichés are learned from a concept in one domain and used to learn concepts within the same domain as well as across domains. Assuming that clichés are learned and used in the same domain, literals used to express different concepts overlap. Consequently clichés learned from one concept should provide appropriate lookahead to learn concepts in the same domain. On the other hand, these clichés probably have few literals in common with concepts across domains, hence the need for more general clichés. To solve this, CLUSE learns two kinds of clichés: Domain Dependent Clichés expressed as a conjunction of literals specific to a domain, and Domain Independent Clichés where literals have variable predicate symbols.

CLUSE is a bottom-up inductive relational learner based on Relative Least General Generalization (RLGG). To remedy the inefficiency and the overgeneralization problems of RLGG, a modified version of RLGG has been

developed that exploits the context in which LGG is applied. The modified RLGG is called Contextual Least General Generalization (CLGG).

Empirical experiments with CLUSE reveal that clichés learned with CLUSE provide appropriate lookahead to escape local plateaus of a hill-climbing learner both within and across domains. For the purpose of the evaluation, FOIL has been extended to learn concepts with or without clichés. In two domains of application, clichés have proven to be useful. One domain is the real-life application defining structures for the finite element methods (FEM). The other domain is the synthetic domain of blocks, which offers a wide variety of problems (or concepts). Other domains of application such as drug design, text categorization, and detecting traffic problems are also discussed.

ACKNOWLEDGEMENTS

Special thanks to my supervisor Dr. Stan Matwin for all his support. He always had time for me and always had confidence in me. He pushed me to excel and the thesis is better for it.

Thanks to my other colleagues and friends, who have helped and encouraged me in different ways.

Thanks to the members of my thesis committee:

- Dr. Franz Oppacher: School of Computer Science, Carleton University;
- Dr. Caroline Barrière: School of Information Technology and Engineering, University of Ottawa;
- Dr. Robert Holte: School of Information Technology and Engineering, University of Ottawa;
- Dr. Charles X. Ling (external examiner): Department of Computer Science, University of Western Ontario;
- Dr. Stan Matwin (thesis supervisor): School of Information Technology and Engineering, University of Ottawa.

Thanks also to l'Université du Québec à Trois-Rivières and le Département de mathématiques et d'informatique for their support.

Finalement, un gros merci tout spécial à Dr Ken Barker pour ses précieux conseils sur tous les aspects de cette thèse. Je le remercie aussi pour sa présence, sa complicité, ses conversations, ses silences et le plus important: son amour pour moi.

CONTENTS

1	Introduction	13
1.1	Contributions.....	18
	1) CLGG.....	18
	2) CLUSE.....	19
	3) Empirical evaluation of the use of learned clichés	19
	4) GENEX and GENTAX.....	20
	5) xFOIL-CLICHÉS	20
1.2	Organization of the thesis	20
2	CLUSE vs. other related concept learners	22
2.1	Related concept learners.....	23
	2.1.1 CIA-CLINT	23
	2.1.2 Silverstein	25
	2.1.3 RDT/MAT-MOBAL.....	26
	2.1.4 Semantic Grammars.....	27
	2.1.5 Lazy Macros	28
	2.1.6 Recursive-Schema	28
	2.1.7 Sketches	28
	2.1.8 Skeletons and Techniques.....	29
2.2	CLUSE vs. related concept learners under different dimensions.....	29
3	Domains of application	32
3.1	The FEM domain	33
	3.1.1 Learning problem.....	34
3.2	The <i>blocks</i> domain.....	36
	3.2.1 <i>Blocks</i> concepts requiring clichés.....	37
	3.2.2 Generated concepts	37
	Concept B1	38
	Concept B2	39
	Concept B3	39
	Concept B4	40
4	GENEX: generator of examples	41
4.1	Description of GENEX.....	43
	Rule templates	45
	Literal templates	45
	Argument templates	47
4.2	Algorithm	47
	Generate example for RuleTemplate.....	48
	Generate literal for LiteralTemplate	48
	Generate Argument from Arg	49
4.3	Applications of GENEX.....	49
	4.3.1 GENEX generates <i>Trains</i> examples	49

4.3.2	GENEX generates <i>Blocks</i> examples	53
4.4	Description of GENTAX.....	55
4.5	Conclusion.....	56
5	CLGG overcomes the shortcomings of LGG	57
5.1	Theory underlying CLGG	58
5.1.1	Preliminary definitions	59
5.1.2	Generalization under θ_F -subsumption	60
	θ_F -subsumption.....	60
	Testing for θ_F -subsumption	61
	Least general generalization under θ_F -subsumption: LGG_F	64
	Existence of LGG_F	65
	Reduction under θ_F -subsumption	65
	θ_F -subsumption lattice	67
5.1.3	CLGG: the operator to find the LGG_F of two clauses in the lattice	68
5.2	Similarity measure evaluates bindings	72
5.2.1	Similarity measure of constants	72
5.2.2	Similarity measure of clauses	75
5.3	CLGG vs. LGG	76
5.4	CLGG relative to a BK vs. RLGG	80
5.5	Conclusion.....	83
6	CLUSE learns clichés	85
6.1	Examples are split into chains.....	86
6.2	CLUSE's algorithm to learn clichés	88
6.2.1	DDCs are generalizations of chains using CLGG	89
	Pruning of generalizations using coverage frequencies	90
6.2.2	DICs are generalizations of DDCs	91
6.3	CLUSE learns clichés in $O(mn^3)$	92
6.4	Examples of learning clichés	93
6.5	Sensitivity of CLUSE to its parameters.....	98
6.6	DDCs represent syntactic restrictions underlying examples of a concept... ..	102
6.7	Conclusion.....	104
7	xFOIL-CLICHÉS learns concepts with clichés	106
7.1	FOIL	108
7.2	xFOIL-CLICHÉS	113
7.2.1	xFOIL parameters and their effect on learning concepts	115
7.2.2	Algorithm for learning concepts with clichés	115
	(1) Clichés used with xFOIL-CLICHÉS	116
	(2) Instantiations of clichés	117
	(3) Variabilization of literals and clichés	122
	(4) Information gain of an instantiation of a cliché.....	123
	(5) Information gain of a cliché.....	123
	(6) Add a cliché to a clause	124
	(7) Add an instantiation of a cliché to a clause	124

7.2.3	Restrictions on the search space for clichés.....	124
7.3	Learning concepts in the <i>blocks</i> domain	125
7.3.1	xFOIL learns no rules for concepts B1 and B2	125
7.3.2	xFOIL-CLICHÉS learns specific rules for concept B1	125
7.3.3	xFOIL-CLICHÉS exploits background knowledge to learn concept B2	127
7.4	Conclusion.....	128
8	Empirical results	130
8.1	Design of empirical testing	131
8.2	Dimensions of evaluation	132
8.2.1	<i>k</i> -fold cross-validation to compare learning algorithms	133
8.2.2	Difference in error of two learning algorithms	134
8.2.3	Cost of an algorithm	135
8.2.4	Compactness of hypothesis.....	135
8.3	Empirical results in the FEM domain	135
	Number of training and testing examples	136
	Number of edges described in definitions.....	136
	Constraints on the rules	137
8.3.1	xFOIL	137
8.3.2	xFOIL-DDC	140
8.3.3	xFOIL-DDC with more specific DDCs.....	142
8.3.4	DICs from the <i>blocks</i> domain	144
8.3.5	xFOIL-DIC.....	145
8.3.6	xFOIL-DIC-10 with the most frequently used literals	149
8.4	Empirical results in the <i>blocks</i> domain	151
8.5	Conclusion.....	153
9	Other domains of application	154
9.1	Drug design – Mutagenesis discovery.....	155
9.2	Text categorization.....	156
9.3	Detecting traffic problems	157
10	Future work	159
10.1	Extensions and improvements to the existing implementation of CLUSE..	159
	Efficiency	159
	Numerical values.....	160
	Generalization and pruning	160
	Coverage in terms of examples instead of chains	160
	Coverage frequencies for DICs as well as DDCs	161
	Noisy data.....	161
	Recursivity	161
10.2	Framework modifications	162
10.2.1	Using CLUSE to create library of clichés	162
10.2.2	Using CLUSE to create a library of concept hierarchies.....	162
	CLUSE learns concept hierarchies.....	162
	CLUSE learns second-order concept hierarchies.....	163

CLUSE creates concept lattices	163
10.3 Other strategies for xFOIL-CLICHÉS.....	164
10.4 CLUSE combined with other learners	164
11 Conclusion	165
12 References	169
<u>Appendix I. Rule templates for blocks examples</u>	<u>177</u>
<u>Appendix II. Generated trains examples</u>	<u>178</u>
<u>Appendix III. The taxonomy form/1</u>	<u>179</u>
<u>Appendix IV. The cost of generalizing literals</u>	<u>180</u>
<u>Appendix V. Examples and background literals for mesh-6</u>	<u>181</u>
<u>Appendix VI. Clichés learned for mesh-4</u>	<u>183</u>
<u>Appendix VII. DDCs vs. rule templates</u>	<u>185</u>
<u>Appendix VIII. Rules learned by xFOIL for mesh-4</u>	<u>187</u>
<u>Appendix IX. Rules learned by xFOIL-DDC for mesh-4</u>	<u>188</u>
<u>Appendix X. Rules learned by xFOIL-DIC for mesh-4</u>	<u>189</u>
<u>Appendix XI. Rules learned by xFOIL-DIC for mesh-4 (10 literals)</u>	<u>190</u>
<u>Index</u>	<u>191</u>

ALGORITHMS

Algorithm 6.1. CLUSE generalizes the two most similar chains.....	89
Algorithm 6.2. Insertion of a generalization in the structure.	90
Algorithm 6.3. CLUSE passes through the structure of generalizations	90
Algorithm 6.4. DDCs are generalized into DICs.	92
Algorithm 7.1. An algorithmic overview of FOIL.....	109
Algorithm 7.2. Algorithm for learning concepts with clichés.....	115
Algorithm 8.1. Cross-validation procedure.....	133

FIGURES

Figure 4.1. Required input to generate trains with GENEX.	51
Figure 4.2. Substitutions and rules given to GENEX to generate blocks examples.	53
Figure 4.3. GENEX generates positive and negative examples.....	55
Figure 4.4. The taxonomy <i>fact</i> generated by GENTAX.....	56
Figure 5.1. θ_F -difference of two expressions E1 and E2.....	62
Figure 5.2. CLGG of two clauses C1 and C2.....	69
Figure 5.3. θ_F -difference of two clauses C1 and C2.	78
Figure 5.4. CLGG is less expensive to apply than LGG.....	79
Figure 5.5. CLGG vs. RLGG.	82
Figure 6.1. Example E expressed in terms of chains (c1 and c2).	87
Figure 6.2. DDCs are learned from examples and optionally, BK.	88
Figure 6.3. The hierarchy of generalizations learned (in bold) for concept B4.	95
Figure 6.4. CLUSE generalizes chains into a structure.....	96
Figure 6.5. GENEX is used to evaluate clichés learned with CLUSE.....	103
Figure 7.1. FOIL's information gain of a literal.	110
Figure 7.2. The information gain in terms of extended tuples	111
Figure 7.3. The information gain computed in terms of the examples	114
Figure 7.4. A cliché is a conjunction of literals.	118
Figure 7.5. Rules learned by xFOIL-CLICHÉS for concept B1.	126
Figure 8.1. CLUSE learns clichés and xFOIL learns hypotheses.....	131
Figure 8.2. Mesh-3 learned rules with xFOIL.	138
Figure 8.3. Learned rules for mesh-3 with xFOIL-DIC.....	146

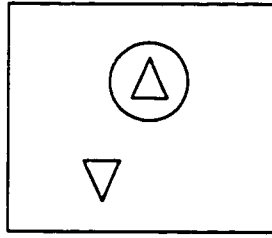
TABLES

Table 2.1. Comparisons of concept learners that learn biases.	30
Table 3.1. The number of edges with the same number of FEs.	35
Table 3.2. Attributes and relations for FEM concepts.	36
Table 3.3. Abbreviations used in experiments with blocks.	38
Table 3.4. Rule descriptions for concept B1.	38
Table 3.5. Rule descriptions for concept B2.	39
Table 3.6. Rule descriptions for concept B3.	40
Table 3.7. Rule descriptions for concept B4.	40
Table 4.1. Literal and argument templates.	46
Table 4.2. Definitions of cars in the trains domain.	50
Table 5.1. List of occurrences for constants.	74
Table 5.2. Similarity bindings of constants.	75
Table 5.3. Similarity of two clauses.	76
Table 6.1. Positive and negative chains for examples generated for concept B4.	94
Table 6.2. Learned DDCs with their coverage frequencies.	98
Table 6.3. DICs generalized from DDCs.	98
Table 6.4. Statistical results of CLUSE's sensitivity to SimCh parameter settings.	100
Table 6.5. Statistical results of CLUSE's sensitivity to PCov parameter settings.	101
Table 6.6. Rule templates used to generate examples of B4-chains.	104
Table 7.1. Rule descriptions for examples of concepts B1 and B2.	107
Table 7.2. Rule descriptions of examples for concept B3.	107
Table 7.3. The effect of a new variable on the current set of tuples.	111
Table 7.4. DDCs are expressed with first-order predicates.	116
Table 7.5. DICs learned with CLUSE for concept B3 (DICs-B3).	117
Table 7.6. Instantiations of DDCs' literals.	118
Table 7.7. DDCs-B3 and their instantiations.	119
Table 7.8. Instantiations of DICs' literals.	120
Table 7.9. DICs' from DICs-B3 and their instantiations.	120
Table 7.10. DICs-B3 of Table 7.5 and their number of instantiations.	121
Table 7.11. Restrictions on the search space of clichés.	122
Table 7.12. The two positive gain instantiations of DDC-17 and DDC-17 itself.	123
Table 7.13. Instantiations of clichés and the literal learned.	126
Table 7.14. Rules learned for the concept B2.	127
Table 8.1. The number of positive and negative examples.	136
Table 8.2. The maximum branching factor for xFOIL.	138
Table 8.3. Statistical results for xFOIL on learning and testing each FEM concept.	139
Table 8.4. The number of DDCs learned with CLUSE for each FEM concept.	141
Table 8.5. The difference in error between xFOIL and xFOIL-DDC.	141
Table 8.6. Statistical results for xFOIL-DDC.	142
Table 8.7. The number of DDCs learned with CLUSE for each FEM concept.	143
Table 8.8. The difference in error between xFOIL and xFOIL-DDC-1%.	143
Table 8.9. Statistical results for xFOIL-DDC-1%.	144
Table 8.10. DICs learned from a blocks concept.	145
Table 8.11. The branching factor for clichés with xFOIL-DIC.	145

Table 8.12. Statistical results for xFOIL-DIC.	147
Table 8.13. The difference in error $\pm$ the standard deviation	148
Table 8.14. The cost of xFOIL and xFOIL-DIC for each concept.	148
Table 8.15. The branching factor for clichés with xFOIL-DIC-10.....	149
Table 8.16. Statistical results for xFOIL-DIC-10.	150
Table 8.17. The difference in error between xFOIL-DIC-10 and xFOIL.....	151
Table 8.18. Statistical results on learning concepts B1 and B2	152

1 Introduction

A number of learning tasks of interest in Artificial Intelligence cannot be solved by standard methods with attribute-value learners (also called propositional learners). This is because of the inherent difficulties of propositional languages to represent structural relationships. Consider the *Bongard* problems which describe geometric puzzles used in simple pattern recognition and classification tasks (Bongard 1970), often applied in standard IQ tests. A Bongard problem consists of geometric figures (or objects) related to each other, where each object has features (or attributes). In an attribute-value representation an object must be described by its values for a fixed set of attributes. These values are given as a conjunction or a disjunction of simple propositional literals. There are no variables, no quantifiers, and no relations among components. Suppose the following Bongard problem is given: a triangle pointing upward is inside a circle and another triangle pointing downward is under the circle and the first triangle.



One way to represent these objects with an attribute-value representation is:

<code>obj1 = circle,</code>	<code>obj2 = tri_up,</code>	<code>obj3 = tri_down,</code>
<code>inside_12 = false,</code>	<code>inside_13 = false,</code>	<code>inside_21 = true,</code>
<code>inside_23 = false,</code>	<code>inside_31 = false,</code>	<code>inside_32 = false,</code>
<code>under_12 = false,</code>	<code>under_13 = false,</code>	<code>under_21 = false,</code>
<code>under_23 = false,</code>	<code>under_31 = true,</code>	<code>under_32 = true</code>

This representation requires that a fixed number of features be used to describe a problem. If one new object is added the number of literals grows exponentially.

On the other hand, examples with a variable number of objects and relevant relations among these objects (such as Bongard problems) can be represented naturally in the first-order formalism as a conjunction of literals, which are represented with k -ary predicates.

For instance, the previous Bongard problem can be represented as :

```
circle(obj1), tri_up(obj2), tri_down(obj3), inside(obj2, obj1),
under(obj3, obj1), under(obj3, obj2)
```

The literal `circle(obj1)` denotes that the object `obj1` is a circle and `inside(obj2, obj1)` denotes that object `obj2` is inside `obj1`. This is one example of the expressive power of a first-order representation, which is the representation used in relational learners.

The relational formalism has been applied successfully in different domains in which the concept to be learned cannot be easily represented in an attribute-value language. These applications include the finite element method (Dolsak *et al.* 1992), traffic problem detection (Dzeroski *et al.* 1998), text categorization (Cohen 1995) and drug design (Finn *et al.* 1998).

Inductive learners that use a first-order language to express examples, background knowledge and concept descriptions (or hypotheses) are called inductive relational learners. Because they induce hypotheses in the form of *logic programs* they are also called *inductive logic programming (ILP)* systems.

The ILP problem can be defined as follows: given a set of positive and negative examples of a concept and possibly some background knowledge, find a hypothesis expressed in a given concept description language such that every positive example is covered by the hypothesis and no negative example is covered by the hypothesis. The training examples, the background knowledge and the induced hypothesis are all expressed as a logic program, with additional restrictions imposed on each one. For instance, training examples are typically represented as ground facts of the target predicate, and most often background knowledge is restricted to the same form.

Inductive relational learners search for the hypothesis either in a bottom-up or in a top-down manner. Bottom-up approaches search from specific examples to a general hypothesis. They start from training examples and search the hypothesis space using generalization operators. GOLEM (Muggleton *et al.* 1992b) uses a bottom-up approach based on Plotkin's notion of *relative least general generalization (RLGG)* (Plotkin 1970). Top-down approaches search from the most general hypothesis to a specific hypothesis using specialization operators. This kind of search can easily be guided by a heuristic. Learning systems using a top-down approach include FOIL (Quinlan 1990), LINUS (Lavrac *et al.* 1991), and FOCL (Pazzani *et al.* 1992; Brunk *et al.* 1991). FOIL and LINUS upgrade attribute-value learners from the ID3 (Quinlan 1986) and AQ (Michalski *et al.* 1986) families toward a first-order logical framework. FOCL combines empirical ILP and explanation based learning (Mitchell *et al.* 1990) to extend FOIL.

Top-down ILP learners such as FOIL and FOCL learn Horn clauses one literal at a time until no negative examples are covered. Each clause is generated by adding one literal at a time using a hill-climbing search. At each step the coverage of the rule after adding a literal is tested on training examples. The literal that best discriminates the remaining positive and negative examples is added to the current clause. The clause is complete when no negative examples are covered by the clause. These systems are vulnerable to *local plateaus*. This arises when the best discrimination would be obtained by adding more than one literal at once. Consider the concept of *cup* which is something that has a handle, a flat bottom, and a concavity pointing upward.

```

cup(X):- partof(X, Y), handle(Y),
        partof(X, Z), bottom(Z), flat(Z),
        partof(X, W), upward_concavity(W).

```

Based on a greedy search for a single literal, the relation `partof(X, Y)` by itself is not likely to distinguish *cups* from *non cups* (both *cups* and *non cups* have parts) and hence would not be added by a hill-climbing algorithm to the definition of a cup. On the other hand, the conjunction of literals `partof(X, Y), handle(Y)` is likely to distinguish some *cups* from *non cups* and should be added to the definition. The problem becomes the search for combinations of literals rather than just single literals. Unfortunately, trying all possible combinations of literals can be intractable. A mechanism to search efficiently through the space of combinations of literals is needed. A learner can be provided with such a mechanism in form of a special-purpose bias.

There is general agreement in the ILP community on the need for biases to restrict the hypothesis space of relational learners. Language schemata in CLINT (De Raedt *et al.* 1992a), rule models in MOBAL (Kietz *et al.* 1992) and relational clichés in FOCL (Silverstein *et al.* 1993) are examples of biases used to learn concepts. Moreover, these learners start with initial biases to learn concept descriptions from which they derive other biases, which are in turn used to learn other concepts. CLINT provides language schemata ordered according to the growing expressiveness of the language they describe. These schemata allow CLINT to search from one subspace to the next (“shifting language”) until it finds a concept description. Then it derives second-order schemata (different from language schemata) from the concept description by turning predicate names into variables while interacting with the user. MOBAL’s user-supplied rule models are ordered in terms of their generality. MOBAL interacts with the user to derive all possible rules to characterize¹ the concept. It relies on user-supplied rules to derive new rule models by turning predicate names into variables. FOCL starts with an unconstrained cliché, which contains no predicate or variabilization restrictions. The unconstrained cliché can be instantiated to all conjunctions of two literals to learn concept descriptions. From the concept description, FOCL derives clichés specific to the domain

¹ Rules are not required to cover all instances or may cover some more than once.

(i.e. with restrictions on predicates and variables) and derives more general clichés by performing a restriction-by-restriction generalization of two clichés at a time until it reaches the unconstrained cliché. In order to start learning relational clichés, FOCL must be restricted to a small number of domains each having a small number of predicates.

Although these learners can derive new biases, they must start with initial biases to guide their search. Initial biases are user-supplied and therefore often ad hoc. Moreover, concept descriptions induced from these learners depend on the initial biases. Since biases are hard to define it would be desirable to learn them automatically.

I have developed and implemented an inductive learner called CLUSE (Clichés Learned and USEd) that learns combinations of literals as a particular type of bias. These combinations of literals are called *relational clichés*. Unlike Silverstein's relational clichés (Silverstein *et al.* 1993), my clichés implicitly represent both the pattern of predicates and variables and restrictions on them. The underlying idea is to learn clichés from examples of a concept and to use them with a hill-climbing learner to escape local plateaus. Clichés are learned from a concept in one domain and used to learn concepts within the same domain as well as across domains. Assuming that in the same domain literals used to express different concepts overlap, then clichés learned from one concept should provide appropriate lookahead to learn other concepts in the same domain. On the other hand, these clichés probably have few literals in common with concepts in other domains, hence the need for more general clichés. To solve this, CLUSE learns two kinds of clichés: Domain Dependent Clichés (DDCs) expressed as a conjunction of literals specific to a domain, and Domain Independent Clichés (DICs) where literals have variable predicate symbols. More formally, DDCs are partially expressed in second-order logic and DICs are totally expressed in that formalism. This will be further discussed in Chapter 6.

CLUSE is a bottom-up inductive relational learner based on Relative Least General Generalization (RLGG). To remedy the inefficiency and the overgeneralization problems of RLGG, I have also developed a modified version of RLGG that exploits the context in

which LGG is applied. The modified RLGG is called *Contextual Least General Generalization (CLGG)* (Chapter 5).

Empirical evaluation has shown that clichés provide appropriate lookahead to learn concepts across domains. I extended FOIL (Quinlan 1990) to learn concepts with or without clichés. The extended system is called xFOIL-CLICHÉS (Chapter 7). I use cross-validation with xFOIL-CLICHÉS to compare the hypothesis accuracy of concepts learned with and without clichés (Chapter 8). Two domains of application are used for the experimentation (Chapter 3). Clichés have proven to be useful in both of them. One domain is the real-life application of the *finite element methods (FEM)*. The other domain is the synthetic domain of *blocks*, which offers a wide variety of problems (or concepts). Concepts in that domain are automatically generated using a domain-independent examples generator called GENEX (Chapter 4). Clichés are learned from a concept in one domain and used to learn concepts within and across domains.

1.1 Contributions

The main contributions of this thesis are:

1) CLGG

CLGG overcomes the inefficiency and the overgeneralization problems of LGG/RLGG. More specifically:

- CLGG produces a generalization for “similar” literals only, limiting the danger of combinatorial explosion and of possible overgeneralization.
- The similarity measure of literals handles literals with embedded functors (*i.e.* nested terms).
- Unmatched literals found to be similar are either generalized using the background knowledge (BK) or returned as part of the generalization, instead of being dropped.

- CLGG changes the representation of the generalization from first-order to second-order logic, or – more exactly – to a subset of second-order logic with variable functor symbols (but not variable predicate symbols).
- The new θ_F -subsumption lattice is used to show the theoretical foundations of CLGG.

The work on CLGG is reported in Morin *et al.* (1999b).

2) CLUSE

CLUSE learns clichés in a bottom-up manner using CLGG.

- Chains are a conjunction of literals sharing at least one constant in the instances on which the generalization is based.
- The similarity measure of clauses (or chains) is based on the similarity of literals used in CLGG, so it handles nested terms (*i.e.* full Horn clauses) as well.
- The choice of which chains to generalize first is based on their similarity (and their cost in the presence of the BK).
- The algorithm generalizes positive chains in a hierarchy of generalizations and prunes this hierarchy according to the coverage of negatives.
- Two kinds of clichés are learned: DDCs and DICs.
- Clichés implicitly represent the pattern of predicates and restrictions on predicates and variabilization.

CLUSE is also reported in Morin *et al.* (1999c).

3) Empirical evaluation of the use of learned clichés

The empirical evaluation of learning with clichés is also of value to the ILP community. It compares the hypothesis accuracy of a concept learned with FOIL to the same concept learned with xFOIL-CLICHÉS, where clichés are either DDCs or DICs. FEM concepts are learned using DDCs (*i.e.* clichés learned from a FEM concept) or using DICs (clichés

learned from a concept in the *blocks* domain). Experimentation with the FEM concepts gives a better understanding of using that dataset.

Empirical testing in the *blocks* domain shows that for some concepts FOIL can learn a definition only in the presence of clichés (assuming they are appropriate for the concept).

4) GENEX and GENTAX

GENEX is a domain-independent tool for ILP systems that I developed for two purposes:

- 1) to generate random examples of concepts easily according to user-supplied concept descriptions;
- 2) to evaluate the similarities between the concept descriptions and the clichés learned with CLUSE. Clichés learned from generated examples are compared to concept descriptions given to GENEX to generate these examples.

GENTAX is a random taxonomy generator. According to user-supplied width and depth values, it generates a hierarchy of literals that can be used as the BK.

GENEX and GENTAX are also described in Morin *et al.* (1999a).

5) xFOIL-CLICHÉS

xFOIL-CLICHÉS is my extended version of FOIL for learning with or without clichés. xFOIL-CLICHÉS searches for clichés when no single literal can improve the accuracy of the clause under learning, yet the clause still covers negative examples.

CLGG, CLUSE, GENEX, xFOIL-CLICHÉS and a driver for the empirical evaluation were implemented in Quintus Prolog version 3.2 and experiments were run on a Sparc Ultra.

1.2 Organization of the thesis

The remainder of the thesis is organized as follows:

Chapter 2 presents a literature review. Chapter 3 describes concepts in the two domains of application used throughout the thesis. Chapter 4 describes GENEX, the random generator of examples. Chapter 5 describes CLGG and its underlying theory. Chapter 6 describes the CLUSE system, gives examples of learning clichés and presents an evaluation of CLUSE. Chapter 7 describes XFOIL-CLICHÉS and examples of learning with clichés. Chapter 8 shows the empirical evaluation of learning concepts across domains with and without clichés. Appendix I lists GENEX rule templates used to generate examples of *blocks* concepts. Appendix II lists the *trains* examples generated with GENEX. Appendix III shows the taxonomy form used with concepts in the *blocks* domain. Appendix IV describes how to compute the cost of generalizing two literals that belong to the BK. Appendix V shows positive and negative examples and background literals for one of the FEM concepts. The remaining appendices show experimental results in the FEM domain.

2 CLUSE vs. other related concept learners

This chapter describes other related concept learners that learn or use biases. Studies on biases used in ILP can be found in (Stahl 1994; Tausend 1994). Particular attention is given to CLINT (De Raedt *et al.* 1992a; De Raedt *et al.* 1992b), MOBAL (Kietz *et al.* 1992; Morik 1993), and Silverstein's system (Silverstein *et al.* 1993), since they learn biases similar to relational clichés (*i.e.* conjunction of literals). These systems are presented with a description of the biases they learn and how they learn and use them. To show the uniqueness of the strategy underlying CLUSE, these systems are compared under different dimensions and CLUSE's strategy is briefly described.

2.1 Related concept learners

2.1.1 CIA-CLINT

The interactive concept learner CLINT (Concept Learning in an INTeractive way) (De Raedt *et al.* 1992a; De Raedt *et al.* 1992b) learns a concept description (*i.e.* a single rule or clause) from a set of positive and negative instances. The learned concept covers the positive instances and rejects the negative instances, assuming that such a concept description exists.

A learning problem solved by CLINT is defined more formally as:

Given:

- a set of positive and negative examples;
- a series of concept description languages: $L_1, \dots, L_n$;
- background knowledge (BK) of concept descriptions;
- an oracle willing to answer questions;
- a novel example or an integrity constraint supplied by the oracle;

Find: an adapted BK such that all positive and no negative examples are covered.

CLINT works with predefined and possibly infinite series of concept description languages (or language schemata) ($L_1, \dots, L_n$), ordered according to the growing expressiveness of the language they describe. L_1 describes a language where all variables that belong to the body of the clause must also belong to the head and vice versa; L_2 describes a language where variables in the body of the clause, except one, belong to the head, and there is only one occurrence of that variable in the body; L_3 allows multiple occurrences of the variable not occurring in the head, etc. In other words, more existentially quantified variables and relations are allowed from one language schema to next (De Raedt *et al.* 1989c).

Language schemata allow CLINT to consider only a subspace of the BK at any time. Given a yet uncovered positive instance, the BK and the language schema L_i , a subspace corresponds to the most specific clause covering the instance with respect to the BK and L_i . Such a subspace is called an explanation. The explanation is then generalized into the starting clause by turning all constants into unique variables. The explanation becomes the body of that clause, while the instance becomes the head. If the starting clause covers negative instances then CLINT shifts its bias to $L_i + 1$ and explores the next subspace. Otherwise, the starting clause is generalized by dropping conditions. Then CLINT generates instances and asks membership questions to the user. This process continues until it is impossible to generalize the starting clause any further without covering negative examples. At this point the generalized clause (or the concept description) is added to the BK.

A constructive inductive analogy technique (called CIA) is developed in the context of CLINT to learn and/or invent new concepts (De Raedt *et al.* 1989a; De Raedt *et al.* 1989b). CIA-CLINT derives *second-order schemata* (different than language schemata) from rules of the concept description learned by CLINT by replacing predicate symbols by variables and asking questions to the user. Once such a schema is derived, it is stored to be matched with clauses when new concepts are learned. To learn new concepts, CIA matches each second-order schema in the BK with the starting clause produced by CLINT. A schema S matches a starting clause C if S subsumes C with a substitution θ (*i.e.* $S\theta \subseteq C$). When a match is found, CIA asks the user if $S\theta$ is the target concept or a new concept. If $S\theta$ is the target concept, CIA asks the user to continue or to stop. If $S\theta$ is a new concept, it asks the user to name the new concept and inserts it in the BK. When the target concept is not found, CIA invokes CLINT to learn the concept and then CIA will learn a new schema.

CLINT can be adapted to restrict the learned BK with user-supplied *integrity constraints* (ICs) (De Raedt *et al.* 1991). User-supplied ICs consist of general first-order logic clauses (*i.e.* more than one literal in the head of the clause). Assuming that ICs are correct, CLINT adds them to the BK and ensures that the learned predicates (in the BK) satisfy

them. When the learned BK violates an IC, CLINT analyses the violation and generates membership questions to the user. Answers either provide additional instances or allow CLINT to remove an incorrect clause in the BK.

2.1.2 Silverstein

(Silverstein *et al.* 1993) proposes a strategy to learn relational clichés used by FOCL (Silverstein *et al.* 1991). *Relational clichés* explicitly define a pattern and a set of restrictions. The pattern abstracts the description of the conjunction of predicates. The restrictions constrain the instantiations of predicates and the *variabilization* (*i.e.* choice of variables for a predicate (Pazzani *et al.* 1992)).

FOCL uses clichés to expand the search space of a relational learner like FOIL (Quinlan 1990; Quinlan *et al.* 1993) with conjunctions of literals potentially useful while learning. FOCL computes the information gain of extensionally and intensionally defined predicates. *Extensional predicates* represent a set of positive and negative instances. *Intensional predicates* represent clauses that define predicates in terms of other predicates (Silverstein *et al.* 1991)² (*e.g.* domain theories and clichés). The information gain is a function of the number of instances covered by the conjunction of the literals (Pazzani *et al.* 1992). FOCL adds whichever of the following has the maximum information gain to the current concept description:

1. a single literal by checking all variabilizations of extensionally defined predicates;
2. a conjunction of extensional literals by applying a modified version of the Explanation-Based Learning (EBL)³ method on the domain theory;
3. a conjunction of literals instantiating a cliché.

Given a set of clichés and a set of domains (instances and domain theories), Silverstein proposes to produce a revised set of clichés. Initially the set of clichés contains the

2 In FOCL the terms "extensionally defined" and "operational" are equivalent. Similarly the terms "intensionally defined" and "non-operational" are equivalent (Pazzani *et al.* 1991).

3 The modified EBL operates with a set of positive and negative instances instead of a single instance.

unconstrained cliché (UC), which can be instantiated to all conjunctions of two literals. His strategy is split in five steps:

1. Learning a concept description (resulting in a learned concept description or LCD) for each domain using FOCL and the set of clichés. Initially the set of clichés contains the UC.
2. Selecting a set of *cliché instantiation candidates* (called CICs), for each LCD. Considering all possible conjunctions derived from a LCD, select the conjunctions that:
 - a) have the maximum information gain and
 - b) could not be generated by induction, *i.e.* at least one of its literals when added in succession does not have maximum information gain.
3. Producing the most specific cliché for each CIC (*i.e.* collect the predicate and the relevant variabilization restrictions).
4. Building/updating the clichés subsumption hierarchy. The top-level cliché corresponds to the UC. A new cliché is inserted under the most specific cliché that subsumes it and is generalized with other clichés also subsumed by the same cliché.
5. Finding a frontier to prune the clichés subsumption hierarchy with a heuristic that evaluates the coverage *vs.* the efficiency of each cliché.

2.1.3 RDT/MAT-MOBAL

MOBAL is a multistrategy learning system that integrates knowledge acquisition tools. Two of the tools are relevant to CLUSE: the *Rule Discovery Tool* (RDT) and the *Model Acquisition Tool* (MAT) (Kietz *et al.* 1992; Morik 1993). MOBAL uses rule models as biases to add rules to the BK and to learn new rule models.

Rule models consist of rules in which predicate variables are used instead of predicates of an application domain. A predicate variable is a variable that can be instantiated only to a

predicate symbol with the same arity. Rule models are organized along a kind of θ -subsumption lattice and ordered in terms of their generality.

RDT generates all possible rules given a set of instances and a set of rule models. The rules are not required to cover all instances and may cover some instances more than once (*i.e.* rules characterize the concept). Generated rules correspond to the most general inductive generalization of instances. These rules can be interpreted by the user as part of the concept description or as useful concepts to insert in the BK.

Prior to the induction process, RDT saturates the BK in the same manner as GOLEM (Muggleton *et al.* 1992b) (generative⁴ with a depth-limit to reduce the expense that such a process requires). The search is breadth-first along the rule models hierarchy with pruning of specializations that do not satisfy the user-supplied acceptance criterion. From the most general to the most specific rule model, RDT instantiates given rule models systematically and tests their instantiations (*i.e.* rules) with instances. Rules that satisfy the acceptance criterion are suggested to the user.

Rule models are either provided by the user or derived by MAT. MAT generates new (non-redundant) rule models from a set of user-supplied rules and a set of existing rule models. Rule models are abstracted from given rules by turning predicate names of the application domain into variables.

2.1.4 Semantic Grammars

CHILL⁵ (Zelle *et al.* 1993a; Zelle *et al.* 1993b; Zelle *et al.* 1994) learns *semantic grammars* (which incorporate syntactic and semantic constraints) to parse English sentences. Given sentences and their correct parses CHILL produces a shift-reduce parser that maps sentences into parse trees in two steps. First, an overly-general shift-reduce parser capable of producing parses from sentences is formulated from the set of instances. This step consists of producing general operator clauses from distinct actions represented by instances using *least general generalization*. Operator clauses are ordered according to their frequency of occurrences in the instances. Second, the syntactic parser is specialized

4 Variables occurring in the head of the clause must also occur in the body of the clause.

by introducing conditions (*i.e.* semantic constraints) that limit the context in which operator clauses can be applied.

2.1.5 Lazy Macros

Cohen extends the GRENDEL⁶ (Cohen 1993a) learning system with *lazy macros*. This extension allows a rapid prototyping of ILP systems using explicit biases to explore the hypothesis space of recursive rules to be searched. Given a set of instances and a description of the desired bias written in BRL (bias representation language), GRENDEL uses an antecedent description grammar (ADG) to define the space of possible clauses that the learner can generate. Cohen's basic idea for a rapid prototyping is to vary GRENDEL's bias and keep the rest of the learning system fixed. To do so, he uses lazy macros to produce ADGs of common ILP biases used by GRENDEL. Lazy macros consist of second-order rules that are an extension of GRENDEL's BRL.

2.1.6 Recursive-Schema

Yokomori (1985) shows that a specific logic program schema called *recursive-schema* captures the common structural property of logic programs. Moreover, he recursively defines a class of recursive-schema programs and shows that they can compute any recursively enumerable language. The recursive-schema programs were applied to transform programs for database design and to synthesize programs for generating new predicates by analogical reasoning.

2.1.7 Sketches

Predefined sketches are refined into clauses in SKIL⁷ (Brazdil *et al.* 1994) to improve the performance of a top-down ILP learning system (like FOIL (Quinlan 1990)). Given *sketches* consist of derivations (*i.e.* derivation trees) generalized by dropping elements and abstracting predicate names into predicate variables. A sketch is associated with the target program (*i.e.* the goal) for which it provides a plan (*i.e.* partial information about the execution). Sketches are used as biases for top-down covering induction. They are

5 CHILL: Constructive Heuristics Induction for Language Learning
 6 GRENDEL: Grammatically Restricted NonDeductive Learner
 7 SKIL: Sketch-based Inductive Learner.

refined by replacing all their predicate variables by suitable BK predicates that cover many positive instances in general, but exclude the negatives. If there is no given sketch, an instance is selected and transformed into a sketch.

2.1.8 Skeletons and Techniques

Kirschenbaum and Sterling (1991) use the Model Inference System (MIS) (Shapiro 1983) with a new refinement operator in MISST⁸. Given input and output behaviour of Prolog programs, the refinement operator generates all clauses of uncovered positive instances and checks them. Clauses are generated based on the decomposition of Prolog programs into *skeletons* (control flow) and *techniques* (standard programming practices). It uses skeletons to find the appropriate control flow of the program (*i.e.* it matches the necessary recursive data structure with skeletons). It uses knowledge to restrict the number of skeletons generated and for pruning redundant clauses. The knowledge is given as relations between the number of elements in lists, or which list is used for recursion. Then skeletons are specialized by applying one of the *techniques* to them.

2.2 CLUSE vs. related concept learners under different dimensions

Table 2.1 shows some dimensions under which CLUSE differs from other learners described earlier in this chapter (*i.e.* CLINT, MOBAL and Silverstein's).

Unlike the other learners CLUSE can learn full Horn clauses (*i.e.* with embedded functions). The other learners restrict the hypothesis space to function-free Horn clauses (*i.e.* no embedded functions allowed). CLUSE is the only system that does not rely on either a user-supplied or predefined bias (taxonomies are optional for CLUSE). Biases learned from the other three learners depend on the user and on predefined biases. Their user must have a good knowledge of the domain of application and also of the learner. CLUSE learns a first kind of bias (DDCs) from combinations of literals that *characterize*

8 MISST: Model Inference System with Skeletons and Techniques

*the concept*⁹. DDCs are analogous to the “new concepts” in CLINT or the CICs in Silverstein’s that are learned from concept descriptions and added to the BK, since they represent conjunctions of literals specific to the domain. In MOBAL, conjunctions of literals are also added to the BK, but they are instantiations of rule models that satisfy the acceptance criterion and are admitted by the user. The main difference between DDCs and these other kinds of conjunctions of literals is that DDCs are the only ones learned based on *lgg* (in this case, CLGG).

	CIA_CLINT	Silverstein/FOCL	RDT/MAT MOBAL	CLUSE
Knowledge representation	function-free HC	function-free HC	function-free HC	full HC
Fully automatic	no	yes	no	yes
Given	BK +/- examples integrity constraints language schemata	domain theories +/- examples set of clichés	BK +/- examples acceptance criterion rule models	taxonomies +/- examples Cov% (+/-)
Predefined bias	language schemata	clichés	rule models	<i>none</i>
User-supplied bias	integrity constraints	domain theories	rule models	<i>none</i>
Domain dependent generalization	concept description	concept description	instantiations of rule models	DDCs
Generalization	predicate variables	dropping restrictions	predicate variables	intensional / extensional predicate variables
Domain independent generalization	2 nd -order schemata	relational clichés	rule models	DICs

Table 2.1. Comparisons of concept learners that learn biases.
HC stands for Horn clauses.

Similar to CLINT and MOBAL, CLUSE generalizes DDCs by replacing predicates by predicate variables. Unlike second-order schemata and rule models, the information as to whether a predicate of a DDC is intensional or extensional is preserved by CLUSE. When DICs are used to learn a concept across domains, this information can be recovered and used to instantiate second-order predicates with intensional or extensional predicates of

⁹ Rules that characterize the concept are not required to cover all instances or may cover some more than once.

the new domain. Unlike Silverstein' clichés, DDCs and DICs (as combinations of literals) implicitly represent restrictions on predicates and variabilization.

CLUSE uses CLGG to learn clichés. It generalizes a set of instances into a hierarchy of generalizations. To preserve as many literals as high as possible into the hierarchy, examples are split into their shortest (most general) chains and a similarity measure of chains is used. CLUSE generalizes positive chains into a hierarchy of generalizations and prunes this hierarchy according to the coverage of negatives and user-supplied coverage thresholds. Preserved generalizations are returned as DDCs. DDCs are further generalized into DICs.

3 Domains of application

Concepts from two domains of application are used throughout the thesis. One domain is the synthetic domain of *blocks* and the other one is the real domain of *Finite Element Methods (FEM)* Design. Concepts in the *blocks* domain represent geometric problems and are similar to the *Bongard* problems used in simple pattern recognition problem (Bongard 1970) and classification tasks, often used in standard intelligence quotient (IQ) tests. Concepts in that domain offer a wide variety of problems that can be automatically generated using GENEX (Chapter 4). Concepts from the FEM have been used by several ILP researchers (Dolsak *et al.* 1994; Lavrac *et al.* 1994) as a kind of benchmark for ILP systems. The FEM domain illustrates the need for clichés in relational learning problems. In particular, the ILP community has identified a need for some kind of lookahead to learn the relation *neighbour* (“... important improvement is possible if the information about neighbouring edges is taken into account properly.” (Lavrac *et al.* 1994)). Clichés provide such a lookahead.

3.1 The FEM domain

This section briefly describes the problem of the finite element method (FEM) from a practical point of view and as a learning task. For more details about FEM see (Dolsak *et al.* 1992), (Dolsak *et al.* 1994), and (Lavrac *et al.* 1994).

FEM is a numerical method to analyze stresses and deformations in physical structures (or models). A model is described as a collection of edges (or meshes). An edge is characterized by the number of finite elements (FEs) it has on it. The basic problem during manual FEM design is the selection of the number of FEs given an edge. This number is a property of the edge as well as of the structure of the model (other edges). Determining the number of FEs, therefore, is a relational problem. Considerable expertise is required in determining the appropriate number of FEs since edges are affected by several factors, including the shape of the model, loads and supports. So, to design a new model, it would be useful to have design rules that determine the number of FEs on its edges. Machine learning can be used to learn these design rules. A rule can be learned from edges with the same FEs occurring in known models. Rules could then be saved in a library and used in a FEM design expert system to determine the class of edges of a new unseen model.

Results from different experiments with FEM models (*e.g.* Lavrac *et al.* 1994; Emde 1994; and Geibel *et al.* 1996), were encouraging enough to believe that the derivation of a knowledge base using automatic learning is a feasible approach for this domain. On the other hand it was observed that:

- 1) more structures are required since each edge has some unique features, which cannot be captured when learning from other structures;
- 2) some kind of lookahead is needed; the relation *neighbour* is not induced in clauses, which means that essential information is not taken into account (Lavrac *et al.* 1994).

3.1.1 Learning problem

Examples are derived from ten models. Last five models are introduced in (Dolsak *et al.* 1994). Each model is described as a collection of edges. Each edge has a corresponding positive example stating the number of FEs on it. Examples are described with the relation: $\text{mesh}(E, N)$, where E is an edge labelled with a combination of a letter and a number. The letter denotes the FEM model while the consecutive numbers denote individual edges in the model. N is the recommended number of FEs along edge E . For example, $\text{mesh}(a14, 7)$ states that 7 is an appropriate number of FEs for edge 14 of model a .

The FEM design problem can be stated as learning a design rule for edges with the same number of FEs occurring in known models. In other words, edges with the same number of FEs make one class and a design rule is learned for one class at a time. All edges of the same class make up the positive examples for that concept and positive examples of all other classes become the negative examples for that concept. For example, $\text{mesh}(E, 1)$ are the positive examples for the concept of edges with 1 FE on them, and $\text{mesh}(E, 2)$, $\text{mesh}(E, 3)$, ..., $\text{mesh}(E, 8)$ are the negative examples for that concept. The design rule learned describes a class of edges with a set of disjunctive (Prolog) rules. An example of one of the rules describing a class of edges is:

```
mesh(X, Y):- usual(X), free(X), opposite(X, Z), usual(Z), short(Z).
```

Y represents the number of FEs on the edge x of that class, while z is some other edge in the model. Edge x is usual and free, edge z is usual and short, and x and z are opposite to each other.

Table 3.1 shows the number of edges with the same number of FEs in each FEM model. Rows represent the ten FEM models ($a - j$) and columns represent the number of FEs 1 - 17. A cell represents the number of edges in the model that have the number of FEs. For instance, model a has 21 edges with 1 FE. Each column represent a difference class (or concept) to learn. Only the first eight classes are used for experiments, since deviations

appear for $N > 8$ (Dolsak 1994)¹⁰. The total number of positive examples for each class (or concept) is shown in the last row. For instance, there are 19 positive examples for the class of edges with 5 FEs on them. Appendix V illustrates positive and negative examples for an edge with six FEs (*i.e.* the concept mesh-6).

		Number of FEs																
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Model	a	21	9	3	2	3			1	1	2	15	15					1
	b	9	9	4	1		11	4	4	4								
	c	6	6	2					14	14								
	d	14	13		2							28	28					
	e	23	36	11	5	9	4		6	7	7	2	1					
	f	12	4	4		2	5			4	4	4					10	
	g	10	10	4	8		2	4	20	20	2	2	2					
	h	16	21	4	3	5	2	6	5	12	7	7		1	1			
	i	8			12				2	2	2	2						2
	j	4	2	6	14		2		2	2	2							
Total		123	110	38	47	19	26	16	54	66	26	58	46	1	1	10	2	1

Table 3.1. The number of edges with the same number of FEs.
Rows represent FEM model. Columns represent the number of FEs on edges.

Each edge of a FEM model has several attributes (called factors), each represented by a literal with a single argument. Attributes influence the description of a concept and are provided as **background literals** (*i.e.* extensional predicates). They are taken into account when inducing a concept definition for a class of edges.

Table 3.2 shows the distribution of each attribute and relation for each FEM concept. There are three classes of attributes:

type: long, usual, short, circuit, half_circuit, quarter_circuit, short_for_hole, long_for_hole, circuit_hole, half_circuit_hole, not_important;

supports: free, one_side_fixed, two_side_fixed, fixed;

loads: not_loaded, one_side_loaded, two_side_loaded, cont_loaded.

¹⁰ Difference in the number of FEs are allowed, *i.e.* for each example mesh(E, N) additional examples

		FEM concept							
		1	2	3	4	5	6	7	8
Predicate name	Type	long			2		2	4	
		usual	6	54	29	36	15	3	6
		short	53	45	9	2			
		circuit							10
		half_circuit					6	2	26
		quarter_circuit						4	10
		short_for_hole	10	4					
		long_for_hole		4		2	2		
		circuit_hole					5		4
		half_circuit_hole				5	4	8	2
		not_important	54	3					
	Supports	free	31	39	12	7	9	7	1
		one_side_fixed	3	17	6	17	2	1	2
		two_side_fixed				3	2	10	1
		fixed	89	54	20	20	6	8	12
	Loads	not_loaded	82	84	28	25	17	22	11
		one_side_loaded	14	5	2	14			
		two_side_loaded		1		1		1	2
		cont_loaded	27	20	8	7	2	4	4
	Geom	Neighbour	483	434	149	189	72	86	64
		Opposite	29	51	15	17	8	18	20

Table 3.2. Attributes and relations for FEM concepts.

The ten most frequently used literals in the FEM domain are shaded¹¹. Geom stands for geometry relations.

The BK also includes two geometric relations between edges: opposite and neighbour. These predicates are symmetric *i.e.* if A is opposite of B then B is also opposite of A. Appendix V illustrates some of the attributes used for the FEM concept mesh-6 (*i.e.* edges with 6 FEs).

3.2 The blocks domain

Concepts in the *blocks* domain represent Bongard (or geometric) problems. Examples for these concepts are generated using GENEX. The next section characterizes examples of *blocks* concepts used to show that clichés can help learners escape local plateau problems.

mesh(E, N1) where $N1 = N \pm 1$ are added.

¹¹ The ten most frequently used literals will be used for one experiment (Section 8.3.6).

3.2.1 *Blocks* concepts requiring clichés

The *blocks* concepts requiring clichés are those for which a top-down learner like FOIL (Quinlan 1990; Cameron-Jones *et al.* 1993) requires some lookahead to improve its performance. FOIL requires a lookahead when learning a rule that still covers negative examples and no single literal can be added that will remove coverage of the negative examples. It cannot find a literal either because there are no more literals that help describe the positives and discriminate the negatives or the candidate literals have no gain, *i.e.* they cover as many negatives as positives.

To facilitate the comparison of learning with clichés to learning without clichés, concepts that cause local plateau problems early in the learning are required. An early local problem arises when the first few objects of positive and negative examples are described with almost the same literals¹². Moreover, to reduce the search for literals¹³ during the learning, only one variable occurs in the head of the rules (there may be other variables in the body of the rules¹⁴). For all *blocks* concepts the literal to learn is `block(x)`. With this kind of example, FOIL would require a lookahead early in learning rules.

Moreover, working with BK means that instantiations of literals are taken from taxonomies (when a literal belongs to a taxonomy and is not at a leaf level). Consequently literals chosen to describe examples appear at low levels in taxonomies. This limits the number of instantiations of literals used to generate examples and to learn with the BK. A small number of instantiations of literals means that it is likely that each positive example is generated at least once in a set of a hundred examples. In learning with clichés, a small number of instantiations considerably reduces the search space when the taxonomies are available.

3.2.2 Generated concepts

Four concepts in the *blocks* domain are generated with GENEX (called B1, B2, B3, and B4). These concepts are used throughout the thesis for examples and experiments. For the

¹² Hence negative examples are not simply anything that is not positive.

¹³ This also results in small *variabilization* (choice of variables, see Section 7.2).

¹⁴ The variable in the head of a rule corresponds to the first object in the instance.

first two concepts FOIL cannot learn a single rule to describe them. These examples are used to show that clichés are useful for learning concepts in the presence or in the absence of the BK. The last two concepts are a bit more complex and are used to learn clichés.

cir = circle	quad = quadripede	con = convex
ell = ellipse	rhombus = rhombus	ush = u-shape
sq = square	para = parallelogram	rect = rectangle
tri = triangle	con_sh = coned shape	poly = polygon
iso = isosceles triangle	equil = equilateral triangle	
rangl = right-angled triangle	iso_rangl = isosceles right-angled triangle.	

Table 3.3. Abbreviations used in experiments with blocks.

For all experiments with *blocks* concepts the abbreviations in Table 3.3 are used.

Examples are generated with different instantiations of literals provided by the form taxonomy (Appendix III). The same taxonomy will be used as the BK to learn these concepts with clichés. Unless stated otherwise, one hundred positive and one hundred negative examples are generated for each concept.

I will now describe each of the four *blocks* concepts in general terms. The reader should refer to Appendix I to see the exact form of the rules given to GENEX to generate the *blocks* concepts.

Concept B1

		Arguments						
		(X)	(X, Y)	(Y)	(Y)	(Y, Z)	(Z)	(Z)
Rules	+	cir	above	rect	± black	above	rangl / iso_rangl	± black
	-	cir	above	rect / sq	red	above	ell / cir	red
		cir	above	rect / sq	red	above	rect / sq	red
		cir	above	rangl / iso_rangl	red	above	ell / cir	red

Table 3.4. Rule descriptions for concept B1.

Rows represent rule descriptions to generate positive and negative examples. Columns represent arguments. A cell represents the predicate associated with the corresponding arguments. '/' represents 'or'. '±' denotes optional.

Table 3.4 shows the rule descriptions to generate positive and negative examples for concept B1. The first rule describes positive examples as a *circle above a rectangle*; this

rectangle is above a *right-angled triangle*. The *rectangle* and the *triangle* are colorless or *black*. One possible example generated from that rule would be:

```
block(X):- cir(X), above(X, Y), rect(Y), black(Y), above(Y,Z), rangl(Z).
```

Notice that a rule description may be defined with more than one rules for GENEX: concept B1 has 3 rules templates for generating negative examples.

Concept B2

Rules	Arguments						
	(X)	(X, Y)	(Y)	(Y)	(Y, Z)	(Z)	(Z)
+	cir	above	rect / sq	± white	above	rangl / iso_rangl	± white
-	cir	above	rect / sq	red	above	ell / cir	red
	cir	above	rect / sq	red	above	rect / sq	red
	cir	above	rangl / iso_rangl	red	above	ell / cir	red

Table 3.5. Rule descriptions for concept B2.

Rows represent rule descriptions to generate positive and negative examples. Columns represent arguments. A cell represents the predicate associated with the corresponding arguments. '/' represents 'or'. '±' denotes optional.

Rule descriptions for examples of concept B2 (Table 3.5) are very similar to B1, but concept B2 is slightly more general than B1. There are two differences with the positive examples: 1) the second object *Y* can be a *rectangle* or a *square*, and 2) the colour of an object in positive examples is optionally *white* instead of optionally *black*¹⁵. Negative examples are defined with exactly the same rules as with concept B1.

Concept B3

Table 3.6 shows the rule descriptions to generate positive and negative examples of concept B3. Concept B3 describes three objects as 1) an *ellipse* above a *rectangle*, which is *left* of a *parallelogram*; or 2) an *ellipse* above a *parallelogram*, which is *right* of a *rectangle*. In the first case the *rectangle* may be *blue*, and the *parallelogram* may be *small*, *medium*, or *large*. In the second case the *parallelogram* may be *red* and the *rectangle* may be *small*, *medium*, or *large*. In both cases the *ellipse* can also be a *circle*, the *rectangle* a *square*, and the *parallelogram* a *rhombus*, *rectangle* or a *square*. Rule for negative examples are similar.

¹⁵ The difference in color is simply used to make the concepts easier to recognize.

		Arguments					
		(X)	(X, Y)	(Y)	(Y, Z)	(Z)	(Z)
Rules	+	ell / cir	above	rect	± blue	leftof	para
		ell / cir	above	para	± red	rightof	rect
	-	ell / cir	above	rect	red	leftof / above	ell / cir
		ell / cir	above	rect	red	leftof / above	iso / iso_rang / equi
		ell / cir	above	para	blue	rightof / above	ell / cir
		ell / cir	above	para	blue	rightof / above	iso / iso_rang / equi
		ell / cir	above	para	blue	above	rect
		ell / cir	above	ell / cir	above	para	Size
		ell / cir	above	ell / cir	above	ell / cir	Size
							Size

Table 3.6. Rule descriptions for concept B3.

Rows represent rule descriptions to generate positive and negative examples. Columns represent arguments. A cell represents the predicate associated with the corresponding arguments. '/' represents 'or'. '±' denotes optional. Size represents the class of predicates for small, medium, and large.

Concept B4

		Arguments							
		(X)	(X)	(X, Y)	(Y)	(Y, Z)	(Z, Y)	(Z)	(Z)
Rules	+	ell / cir	±small	above	rect/sq	±large	leftof	iso	±red
		ell / cir	±small	above	iso/iso_rangl	±red	leftof	rect/sq	±large
	-	ell/cir		above	ell/cir	above		ell/cir	
		rect/sq		above	rect/sq	above		rect/sq	
		iso/iso_rangl		above	iso/iso_rangl	above		iso/iso_rangl	
		ell/cir		leftof	ell/cir	leftof		ell/cir	
		rect /sq		leftof	rect/sq	leftof		rect/sq	
		iso/iso_rangl		leftof	iso/iso_rangl	leftof		iso/iso_rangl	
		ell/cir/rect/sq/						ell/cir/rect/	
		iso/iso_rangl		above	ell/cir/iso/iso_rangl	leftof		sq/iso/iso_rangl	
		ell/cir/rect/sq/						ell/cir/rect/	
		iso/iso_rangl		above	ell/cir/rect/sq		leftof	sq/iso/iso_rangl	

Table 3.7. Rule descriptions for concept B4.

Rows represent rule descriptions to generate positive and negative examples of concept B4. Columns represent predicates associated with the corresponding arguments. '/' represents or. '±' denotes an optional literal.

Table 3.7 shows the syntactic restrictions underlying examples of concept B4. The concept is more complex than the other three *blocks* concepts and will be used mainly to show examples of learning clichés with CLUSE. Concept B4 describes three objects as 1) an *ellipse above a rectangle*, which is *left of an isosceles triangle*; or 2) an *ellipse above an isosceles triangle*, and a *rectangle is left of the triangle*. In both cases the *ellipse* may be *small*, the *rectangle* may be *large* and the *triangle* may be *red*. Moreover, the *ellipse* can also be a *circle*, the *rectangle* a *square*, and the *isosceles triangle* a *right-angled isosceles triangle*. Rules for negative examples are similarly represented.

The exact rules that are given to GENEX for *blocks* concepts are listed in Appendix I.

4 GENEX: generator of examples

The topic of testing results of ILP learners still awaits discussion by the ILP community. Quite often, results of an ILP learner, *e.g.* a Prolog program, are manually checked for correctness by the user, and are either accepted or rejected (see (Aha *et al.* 1994; Cohen 1993b) for exceptions to this approach). This test-free verification works well when the objective of learning is to learn simple Prolog definitions which are usually known to the user, at least implicitly. It will not work, however, when the learning targets are more complex programs, or when the user wants to know how far from being correct the programs are when they are not totally correct. There is no reason for not applying in ILP the standard testing methodologies of machine learning *e.g.* cross-validation (Mitchell 1997; Henery 1994).

To evaluate a logic program learned by an ILP system, one needs to use a variety of datasets. For instance, different datasets are needed to evaluate the performance of the learner with function-free examples, with embedded functions (*i.e.* nested structures), or with recursive examples. Also different datasets are needed to vary the number of literals

per example, the number of arguments per literal, the number of class labels, etc. Although manually constructed datasets allow comparisons between learners that learn the same definition, they are insufficient to test the full potential of a learner. Examples from existing datasets are often handpicked and limited in number. Datasets are known to be correct, since the underlying constraints of examples are known. However, the ILP program that corresponds to them may not be known. For instance, we know the constraints underlying the addition examples, but we may not know the ILP program for the addition. Manual generation of examples can be intractable.

Muggleton proposes a domain-dependent East-West *trains* generator that embodies the attribute constraints initially suggested by Ryszard Michalski (Section 4.3.1). As a consequence, this generator is unable to generate examples in other domains.

To address these problems, I have developed a domain-independent ***generator of examples*** called GENEX to produce examples of concepts for ILP systems. Concepts are generated with GENEX from user-supplied rule templates and substitutions allowed for these templates.

If one is to use GENEX to generate examples for CLUSE, then one way to evaluate CLUSE is to evaluate the similarities between rule templates given to GENEX and the clichés learned with CLUSE. Like rule templates, clichés represent syntactic restrictions. Rule templates are given to GENEX to generate examples of a concept. From these examples CLUSE learns clichés. Consequently, one expects the CLUSE-generated clichés to resemble the rules that define the instances from which the clichés were learned.

GENEX can generate instances with two formats. The first format is used throughout this chapter unless otherwise mentioned.

- 1) ground Horn clauses (or simply clauses). This representation is useful for bottom-up learners like Plotkin's *lgg* (Plotkin 1970; Plotkin 1971), Buntine's generalization (Buntine 1988), GOLEM (Muggleton *et al.* 1992b), inverse resolution (Muggleton *et*

al. 1992a), and inverse implication with LOPSTER (Lapointe *et al.* 1992). In this thesis, this is the representation used with CLUSE.

- 2) ground facts to represent examples and BK, where examples are function-free ground facts. The BK consists of extensional definitions given by a finite set of function-free ground facts. These predicates are used to make the clauses for the predicate being learned. For instance, the predicate `linked-to(X, Y)` can be useful in learning the concept `can-reach(X, Y)`. These predicates must be defined extensionally. This representation is useful to top-down systems like FOIL (Cameron-Jones *et al.* 1993; Quinlan 1990), ML-SMART (Bergadano *et al.* 1988), FOCL (Pazzani *et al.* 1992), MIS (Shapiro 1983), FILP (Bergadano *et al.* 1990), mFOIL (Dzeroski *et al.* 1991), LINUS (Lavrac *et al.* 1991), and MARKUS (Grobelnik 1992). In this thesis, this representation is used to learn with xFOIL-CLICHÉS.

I also have developed a **generator of taxonomies**, called GENTAX that can be used to test the influence of different shapes of taxonomy on the learning. Both generators are implemented in Prolog and are available at

<http://www.site.uottawa.ca/~jmorin/Programs/Generator>

Section 4.1 describes GENEX. Section 4.2 describes the algorithm of GENEX. Section 4.3 illustrates the application of GENEX with *trains* and with *blocks* concepts. A set of *trains* is generated and compared with the well-known set of *trains* from (Michie *et al.* 1994). The generation of *trains* shows some basic functionalities of GENEX. Examples of a concept in the *blocks* domain are also generated to illustrate other possibilities offered by the generator. Section 4.4 describes GENTAX.

4.1 Description of GENEX

To apply thorough empirical testing to a learning system means to test all its parts and their limits. This requires many sets of examples. These examples need to be easily modified (or generated) in terms of:

- the number of literals per example;
- the structure of the literals: function-free, with functions (*i.e.* nested structures), recursive, etc.;
- the use of constants, nominal values¹⁶, classes of predicates (Vrain 1990), taxonomies (Kodratoff *et al.* 1986; Vrain 1990);
- the number of arguments per literal;
- the arguments common to different literals;
- the number of examples per class label;
- the number of different class labels.

Without a tool, generation of examples driven by all parameters above can become intractable. GENEX uses rule templates and substitutions for constants, function symbols, and predicate symbols to generate concepts represented as examples. These examples are generated as ground facts or ground clauses. In terms of an interpretation over the Herbrand Universe, an *example* is an assignment of constants, function symbols, and predicate symbols (*i.e.* is a subset of the base) (Lloyd 1984). A **rule template** is a conjunction of literal templates which define the syntactic restrictions of literals (predicates and their arguments) in the base. *Substitutions* define the universe of the constants, function symbols, and predicate symbols to use when filling the literal templates. For instance, suppose the general format of a literal template is *predicate(function(constant))* and substitutions for *predicate* are *p* and *q*, the substitutions for *function* are *f* and *g*, and the substitution for *constant* is *x*. Then the universe over the substitutions is $\{p(x), q(x), p(f(x)), q(f(x)), p(g(x)), q(g(x)), p(f(f(x))), q(f(f(x))), p(f(g(x))), q(f(g(x))), \dots\}$.

¹⁶ Nominal values are constants that belong to a set of values. For instance, the set of nominal values for *size* could be $\{small, medium, large\}$.

Rule templates

A **rule template** is made of a head and a body (*head:- body*) or simply a body. The head represents a single literal and the body a list of at least one literal. A rule template can take one of the two following formats:

- 1) `rule(RuleId, (literal1:- [literal2, literal3, ...])).`
- 2) `rule(RuleId, [literal1, literal2, ...]).`

where the `RuleId` is a unique number that identifies rules.

Format 1 is used to generate examples of concepts as Horn clauses or as ground facts. An instance generated as a clause has the format: `'Label(Clause).'`. For instance,

```
positive((block(v1, v2):- [on(v1, v2), blue(v1), red(v2)])).
```

where `positive` is the user-assigned class label to the rule template that generated the instance, and `(block(v1, v2):- [on(v1, v2), blue(v1), red(v2)])` is a Horn clause that has the same format as the rule template. The same rule template can generate ground facts in the following way: `block(v1, v2).` is a positive example, and `on(v1, v2).`, `blue(v1).`, and `red(v2).` are three facts.

GENEX can generate a number of examples from each rule template or generate a given number of examples from the whole set of rules. All choices made with GENEX are random. A class label can be assigned to the rule template. For instance, a rule template labeled *positive* generates examples labeled *positive*. The applications in Section 4.3 illustrate these options.

Literal templates

Literal templates syntactically define predicates and their arguments to be used in generating examples. There are five literal templates defined in GENEX. For each literal template used in a rule, the user must provide the associated set of substitutions. According to the literal template and its number of arguments (*i.e.* arity), GENEX chooses a predicate from the user-supplied set of substitutions.

	Templates	Associated substitutions
Literal	vp#(ListArg) or lp(N, ListArg) np#(ListArg) cp#(ClassName, ListArg) taxo(TaxoName, TopPred(ListArg), MaxDist) rl(From, To, PredName) <pred>(ListArg)	NbArg, ListPred NbArg, ListNPred ClassName/NbArg, ListCPred TaxoName, TopPred, Pred, Dist ListLiterals <i>not applicable</i>
Argument	constants nominal value or n#(Name) f#(ListArg) <variable>	ListConstants Name, ListNominal NbArg, ListFunctions <i>not applicable</i>

Table 4.1. Literal and argument templates and the associated substitutions. The “#” represents a number for a single identification of a predicate, a function or an argument in a rule template. “< >” represents first-order predicates.

Table 4.1 shows the format of literal and argument templates, and the associated substitutions required. A literal template with a variable predicate like {vp#, np#, cp#, taxo, rl, lp} is replaced in the examples with a predicate from the appropriate set of user-supplied substitutions. Other predicate names (*i.e.* first-order predicates) are simply reproduced in the examples. For a predicate of the form vp# (for variable predicate) encountered in a rule, GENEX chooses a predicate from the list of predicates with the same arity¹⁷. For instance, a literal template vp1(x) requires a set of substitutions with the number of arguments (*NbArg*) equal to 1 and a list of predicate names (say [p1, p2]) to substitute for vp1. In this case, either p1(x) or p2(x) would be generated. Similarly, for a predicate of the form np# (for nested predicates), GENEX chooses a predicate from the list of nested predicates. For a predicate cp# (for class predicates), GENEX chooses a predicate from the list of predicates that belong to the class Name. For literal template with the predicate name taxo, GENEX generates a predicate from the taxonomy named TaxoName. It randomly chooses a predicate at a maximum number of levels MaxDist of the predicate TopPred in the taxonomy. Literal templates with a predicate lp (list of predicates) or rl (random literals) generate more than one literal in the examples. These literals describe the number of literals to generate their arguments and from which list to choose their predicate and arguments.

¹⁷ The arity of the literal template determines the list of substitutions to choose from when more than one list is defined for a predicate or a function.

The “#” at the end of a predicate¹⁸ represents a number and allows a user to generate the same predicate more than once in an example. Every occurrence of the same predicate $vp\#$ (with the same arity), in a rule is replaced with the same predicate in the example. For instance, the rule $(vp1(x, z) :- [vp2(x, y), vp1(y, z)])$ defines recursive examples, since the same predicate (with the same arity) is used to described the head and the last literal in the body of that rule. On the other hand, a rule like $(vp1(x, z) :- [vp2(x, y), vp3(y, z)])$ describes examples with three different predicates. The “#” works the same for predicates $np\#$, $cp\#$, and for arguments of the form $n\#$ and $f\#$.

Argument templates

There are four kinds of arguments in a rule (Table 4.1). Arguments that belong to the list of constants or to a list of nominal values are reproduced in examples. An argument of the form $n\#(\text{Name})$ (for nominal) is replaced with a nominal value chosen from a list Name of nominal values. An argument of the form $f\#$ (for function) is replaced with a function name (or simply a function) that belongs to the predefined list of functions (with the same arity). Any other arguments will be replaced with an argument of the form $v\#$. This last case ensures that each example will be *standardized apart* (Nienhuys-Cheng *et al.* 1996), *i.e.* different variable names will be used in every example.

4.2 Algorithm

There are two ways to invoke GENEX:

1. Generate N examples from each rule RuleId between FromRuleNb and ToRuleNb

```

for RuleId from FromRuleNb to ToRuleNb do
  get the RuleTemplate for RuleId
  for 1 to N do
    generate example for RuleTemplate

```

¹⁸ The “#” is similarly used at the end of an argument.

2. Generate a total of N examples randomly from rules between FromRuleNb and ToRuleNb

```

for 1 to N do
  choose a RuleId between FromRuleNb and ToRuleNb
  get the RuleTemplate for RuleId
  generate example for RuleTemplate

```

The following procedures describe the generation of examples.

Generate example for RuleTemplate

```

for each literal L of the RuleTemplate do
  generate LiteralTemplate from L

```

Generate literal for LiteralTemplate

```

find arity Arity of LiteralTemplate
case LiteralTemplate of
  . rl(From, To, PredName/Arity):
    for From to To do
      choose literal L from defined literals
      with the predicate PredName and Arity
      generate Argument from Arg
  . lp(N, Arg):
    for 1 to N do
      choose predicate in the list of predicates with Arity
      generate Argument from Arg
  . cp#(Name, Arg):
    choose predicate in class of predicates called Name with Arity
    generate Argument from Arg
  . np#(Arg):
    choose predicate in the list of nested predicates with Arity
    generate Argument from Arg
  . taxo(Name, TopPred(Arg), MaxDist):
    choose predicate in the taxonomy Name with Arity
    maximum number of levels of MaxDist from TopPred
    generate Argument from Arg
  . vp#(Arg):
    choose predicate in the list of predicates with Arity
    generate Argument from Arg
  . otherwise (i.e. <pred>(Arg))
    preserve the predicate
    generate Argument from Arg

```

Generate Argument from Arg

```

for each argument A in Arg do
  if A belongs to one of the lists of constants or nominal values then
    reproduce A in the example
  else
    case A of
      . f#(Arg) :
        choose a function name from the list of functions
        generate Argument with Arg
      . n#(Name) :
        generate a nominal value from the list of
        nominal values called Name
      . otherwise:
        generate a variable name of the form v#

```

4.3 Applications of GENEX

This section illustrates two applications of GENEX. The first application is in the *trains* domain. A set of *trains* is generated and compared with the well-known set of *trains* from (Michie *et al.* 1994). Since the generation of *trains* covers only a few of the functionalities of GENEX, GENEX is also applied to generate concepts in the *blocks* domain.

4.3.1 GENEX generates *Trains* examples

The East-West Challenge requires generation of a large number of random generated examples. As an exercise, GENEX is used to generate a set of 100 *trains* and compare them with Michie's well-known set of *trains*. Unlike Michie's *trains* generator, GENEX is domain-independent. Therefore attribute-value constraints suggested by Michalski (Michalski *et al.* 1977) are user-supplied, rather than embodied in the generator itself. These constraints are:

1. A train has two, three or four cars, each of which can either be long or short.
2. A long car can have two or three axles; a short car must have exactly two axles.
3. A short car can be rectangular, u-shaped, bucket-shaped, hexagonal, or elliptical, while a long car must be rectangular.

4. A hexagonal or elliptical car is necessarily closed, while any other car can be either open or closed.
5. The roof of a long closed car can be either flat or jagged.
6. The roof of a hexagonal car is necessarily flat; the roof of an elliptical car is necessarily an arc. Any short closed car can have either a flat or a peaked roof.
7. If a short car is rectangular then it can either be double sided or not double sided.
8. A long car can be empty or it can contain one, two or three replicas of one of the following kinds of load: circle, inverted-triangle, hexagon, rectangle.
9. A short car contains either one or two replicas of the following kinds of load: circle, triangle, rectangle, and diamond.

Like in Michalski's original version a possible distinction between hollow and solid wheels was ignored. Michie's syntax is used *i.e.* *c* for cars, *l* for load, etc.

Car	Shape	Size	Side	Roof	Axles	Load shape	Load nb	nb
1	rect	long	not_double	none / flat / jagged	2 / 3	cir / inv-tri / hexa / rect	0 / 1 / 2 / 3	96
2	rect	short	not_double / double	none / flat / peaked	2	cir / tri / rect / diam	1 / 2	48
3	u-shaped / b-shaped	short	not_double	none / flat / peaked	2	cir / tri / rect / diam	1 / 2	48
4	hexa	short	not_double	flat	2	cir / tri / rect / diam	1 / 2	8
5	ell	short	not_double	arc	2	cir / tri / rect / diam	1 / 2	8

Table 4.2. Definitions of cars in the trains domain.

According to these constraints, cars could be represented as shown in Table 4.2. There are five ways to represent cars in terms of constants and alternative attribute-values following Michalski's constraints. The number of cars in the last column is the product of the number of attributes offered in each row. For instance, the first three attribute-values of the first car are constants, *i.e.* they represent cars that are *rectangle*, *long*, and *single sided* (*i.e.* *not_double*). The next four attribute-values represent alternative attribute-values to complete the description of these cars. That is, each of these cars either has a *flat*, a *jagged* or *none* (*i.e.* no roof). They can either have 2 or 3 wheels (*axles*). Their

load's shape is either a *circle*, an *inverted-triangle*, an *hexagon* or a *rectangle*. And they can have between 0 and 3 loads of the same kind (i.e. $3 \times 2 \times 4 \times 4 = 96$).

Following Table 4.2, it is easy now to define rule templates and the associated substitutions to generate *trains*. Each line of the table defines a possible car (i.e. given as a fact to GENEX). *Trains* are defined in a single rule, as a combination of two to four of these cars. The substitutions required to complete the example consist of 1) a list of all constants that occur in definitions of a car, and 2) as many lists of nominal values as the number of different attributes that offer alternative values.

```
% Substitutions
constants([rect,hexa,ell,long,short,not_double,flat,arc,2]).
nominal(side,[double,not_double]).           % car's side
nominal(lg_rf_sh,[none,flat,jagged]).         % roof's shape for a long car
nominal(shrt_rf_sh,[none,flat,peaked]).        % roof's shape for a short car
nominal(lg_axles,[2,3]).                      % nb. of axles for long car
nominal(lg_l_sh,[cir,utri,hexa,rect]).         % load shape for a long car
nominal(shrt_l_sh,[cir,tri,rect,diam]).        % load shape for a short car
nominal(lg_l_nb,[0,1,2,3]).                   % load number for a long car
nominal(shrt_l_nb,[1,2]).                     % load number for a short car

% car definitions
c(rect, long, not_double, n1(lg_rf_sh), n2(lg_axles), l(n3(lg_l_sh),
n4(lg_l_nb))).
c(rect, short, n1(side), n2(shrt_rf_sh), 2, l(n3(shrt_l_sh),
n4(shrt_l_nb))).
c(n1(shrt_sh), short, not_double, n2(shrt_rf_sh), 2, l(n3(shrt_l_sh),
n4(shrt_l_nb))).
c(hexa, short, not_double, flat, 2, l(n1(shrt_l_sh), n2(shrt_l_nb))).
c(ell, short, not_double, arc, 2, l(n1(shrt_l_sh), n2(shrt_l_nb))).

% rule template
rule(1, [r1(2, 4, c/6)]).                    % randomly choose 2 to 4 cars

:- gen_nb_ex_for_each_rule(train, 1, 1, 100).
```

Figure 4.1. Required input to generate trains with GENEX.
c for cars and l for load.

Figure 4.1 shows an example of GENEX being used to generate 100 *trains*. Appendix II provides a subset of unclassified *trains* generated with GENEX from rule 1. rule 1 defines a train as a combination of two to four cars. Facts *c/6* define five ways to generate a car based on the constraints in Table 4.2. Facts *nominal/2* define nominal values to choose from whenever a nominal argument is encountered in the definition of a

car. The fact `constants/1` enumerates arguments of definitions of cars that are reproduced in examples. For instance, suppose that the first fact `c/6` is chosen by GENEX to generate one of the cars for a train defined with `rule 1`. GENEX reproduces the predicate and the first three arguments in the example, since the predicate `c` is not a variable predicate (*i.e.* not of the form `vp#`, `np#`, `cp#`, `lp`, `rl`, `taxo`) and the first three arguments: `rect`, `long`, and `not_double` belong to the list of constants. The last three arguments of that car force GENEX to choose a nominal value from alternatives for each argument. `n1(lg_rf_sh)` causes GENEX to choose a nominal value in the list named `lg_rf_sh` (*i.e.* `none`, `flat`, `jagged`). `n2(lg_axles)` causes GENEX to generate another nominal value from the list of values for `lg_axles` (*i.e.* `2` or `3`), and the last argument `l(n3(lg_l_sh), n4(lg_l_nb))` causes it to generate a nested structure with the predicate name `l` (for *load*) and two nominal values as arguments. The first argument must be chosen from the list of `lg_l_sh` values and the second one from the list of `lg_l_nb`.

There are roughly 1.9×10^9 possible *trains*¹⁹, since 208 (*i.e.* $96 + 48 + 48 + 8 + 8$) cars could be generated and a combination of two to four cars makes a train. With such a large number of *trains*, it is unlikely *trains* common to both Michie's set and mine would be found. On the other hand, nine *trains* from my set share no cars with Michie's *trains*, seventeen share exactly one car and six share exactly two.

The generation of *trains* illustrates:

- the generation of a given number of examples from a set of rules (here only one rule);
- the generation of unclassified examples, represented with a single class label for all examples;
- the use of rules with the *body* only (*i.e.* format 2 of a rule template in Section 4.1);

¹⁹ $208^2 + 208^3 + 208^4 \approx 1.9 \times 10^9$

- the use of a predicate `r1` to generate a list of literals in examples and the definitions of the literals required;
- the use of arguments as constants and nominal values.

4.3.2 GENEX generates *Blocks* examples

This section illustrates abilities of GENEX not illustrated by the *trains* example. Examples generated here do not represent a meaningful *blocks* world scene. Functionalities illustrated here include:

- the generation of a total number of examples from a set of rules;
- the use of different class labels assigned to examples;
- the use of rules of the form (*head*:- *body*);
- the use of the different predicates: `lp`, `cp#`, `np#`, `vp#`, `taxo`, and constant predicates, and their associated substitutions required;
- the use of nested arguments and variables.

```
% User-supplied substitutions
substitution(variable,1,[big,long,large]).      % for vp# and lp pred.
substitution(nested,2,[on,left,right]).         % for np# pred.
substitution(class,color/1,[yellow,red,blue]). % for cp# pred.

% rule templates
rule(1,(bl(x,y):- [np1(taxo(form,con(x),5),taxo(form,poly(y),4)),
                  cp1(color,x),cp1(color,y)]])).
rule(2,(bl(x,y):- [under(sq(x),ell(y)),vp1(x),lp(2,y)]])).
rule(3,(bl(x,y):- [np1(cp1(color,x),cp2(color,y)),
                  taxo(form,tri(x),2),quad(y)]])).

:-      gen_tot_ex_from_rules(pos,1,2,10),
        gen_tot_ex_from_rules(neg,3,3,5).
```

Figure 4.2. Substitutions and rules given to GENEX to generate *blocks* examples.
10 positive and 5 negative examples in the *blocks* domain.

Figure 4.2 shows the substitutions and the rule templates to generate examples as Horn clauses in the *blocks* domain. Ten positive examples are generated from the first two rules

and five negative examples from the third one. To generate examples from these rules, the user must supply the taxonomy called *form/1* (Appendix III)²⁰, the substitutions for literal templates with a predicate of the form *vp#* and *lp*, the substitutions for literal templates with embedded functions (*i.e.* of the form *np#*), and a class of predicates named *color*.

Rule 1: the first literal `np1(taxo(form,con(x),5),taxo(form,poly(y),4))` defines a variable predicate with a ***nested structure*** (*i.e.* with embedded functions) of two arguments. A predicate is chosen from the list of predicates (with the same arity) and produced in the examples. The functions for both of its arguments come from the taxonomy called *form*. The first generated function must be at a number of levels of at most 5 from the predicate *con*, and the second one must be at a maximum number of levels of 4 from the predicate *poly*. Arguments *x* and *y* are replaced with a variable symbol of the form *v#* in the examples, since they do not occur in any lists of constants or nominal values. Occurrences of these arguments in rule 1 are replaced with the same variable symbols in the example. The next two literals `cp1(color,x)` and `cp1(color,y)` force GENEX to generate the same predicate name twice from the list of predicates in the class *color*, each one with the variable names previously assigned to arguments *x* and *y*.

Rule 2: the predicate and functions of the first literal `under(sq(x),ell(y))` are reproduced in the examples (since it is not of the form *vp#*, *cp#*, etc.). Similar to the first rule, variable symbols are generated for arguments *x* and *y*. The second literal causes GENEX to choose a predicate from the list of substitute predicates with the previously assigned variable symbol for *x* as argument. The last literal `lp(2,y)` of that rule, generates a list of at least one and at most 2 literals, where each literal is made of a predicate that belongs to the list of substitute predicates and the variable symbol for the argument *y*. Rule 3 uses different combinations of the same features, but generates negative examples.

²⁰ The taxonomy is shown as a tree for clarity. In the actual implementation, the taxonomy is represented as a set of the number of levels between pairs of predicates in the taxonomy.

```

1. pos((bl(v1,v2):-[left(tri(v1), equil(v2)), yellow(v1), yellow(v2)]))).
2. pos((bl(v3,v4):-[right(quad(v3), iso_r_angled(v4)), blue(v3), blue(v4)]))).
3. pos((bl(v5,v6):-[right(equil(v5), iso(v6)), yellow(v5), yellow(v6)]))).
4. pos((bl(v7,v8):-[left(equil(v7), para(v8)), blue(v7), blue(v8)]))).
5. pos((bl(v9,v10):-[on(r_angled(v9), quad(v10)), blue(v9), blue(v10)]))).
6. pos((bl(v11,v12):-[under(sq(v11), ell(v12)), long(v11), large(v12)]))).
7. pos((bl(v13,v14):-[on(quad(v13), equil(v14)), yellow(v13), yellow(v14)]))).
8. pos((bl(v15,v16):-[under(sq(v15), ell(v16)), big(v15), large(v16), big(v16)]))).
9. pos((bl(v17,v18):-[under(sq(v17), ell(v18)), long(v17), big(v18), long(v18)]))).
10. pos((bl(v19,v20):-[left(r_angled(v19), sq(v20)), blue(v19), blue(v20)]))).
11. neg((bl(v21,v22):-[on(red(v21), blue(v22)), r_angled(v21), quad(v22)]))).
12. neg((bl(v23,v24):-[left(red(v23), yellow(v24)), iso(v23), quad(v24)]))).
13. neg((bl(v25,v26):-[right(red(v25), yellow(v26)), equil(v25), quad(v26)]))).
14. neg((bl(v27,v28):-[left(red(v27), yellow(v28)), r_angled(v27), quad(v28)]))).
15. neg((bl(v29,v30):-[right(yellow(v29), blue(v30)), r_angled(v29), quad(v30)]))).

```

Figure 4.3. GENEX generates positive and negative examples from the input of Figure 4.2.

Figure 4.3 shows the examples of *blocks* generated from rule templates in Figure 4.2. Positive examples 1 to 5, 7, and 10 are generated from rule 1, positive examples 6, 8, and 9 are generated from rule 2, and negative examples number 11 to 15 from rule 3. Notice that the generated examples are standardized apart (Section 4.1). Their predicate name corresponds to the class label (the same as the rule they are generated from). Notice also that the frequency of using either rule 1 or 2 to generate positive examples is random (rule 1 was used seven times and rule 2 was used three times).

4.4 Description of GENTAX

Using a single taxonomy to generate examples does not shed any light on the effect of the depth and breadth of the taxonomy on the learner. What is needed is a facility that allows the user to vary these attributes of the taxonomy. A *generator of taxonomies* called GENTAX is implemented to solve this problem. GENTAX automatically generates taxonomies²¹ that could be used as part of the substitutions to generate examples with GENEX.

²¹ GENTAX generates complete trees, but branches can be easily removed if necessary.

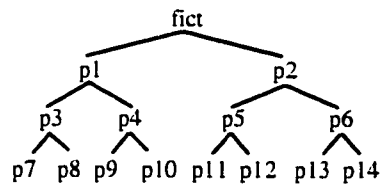


Figure 4.4. The taxonomy *fict* generated by GENTAX with a branching factor of 2 and a depth of 4.

The user gets to specify the depth and the branching factor to control the shape of the taxonomy. A taxonomy is expressed in terms of the number of levels between pairs of predicates (*P1* and *P2*), where *P1* is a (direct or indirect) parent of *P2* in the taxonomy. Figure 4.4 shows a taxonomy named *fict* with a branching factor of 2 and a depth of 4. The number of levels between the predicates *p1* and *p10* in the taxonomy is 2, whereas the number of levels between *p2* and *p10* is undefined.

4.5 Conclusion

GENEX randomly generates examples of concepts from user-supplied rule templates and substitutions.

GENEX and GENTAX are independent of the learner and could be used by machine learning community at large as domain-independent tools for making (complete) sets of examples and taxonomies for thorough empirical testing of learners.

5 CLGG overcomes the shortcomings of LGG

Much of the existing work in learning in the first-order logic setting is based on Plotkin's Least General Generalization LGG (Plotkin 1970), and on its extension – called *Relative LGG* (RLGG). RLGG compiles additional knowledge into the generalization process. From the machine learning perspective, however, there are certain practical shortcomings of the LGG approach to generalization. First, in the worst case, the *cost* of applying LGG on two clauses is equal to the length of the first clause times the length of the second one. So the cost of applying LGG on a set of clauses becomes exponential in the number of literals to generalize. Second, LGG may overgeneralize, because important relationships between literals can be lost during the generalization process (Wirth *et al.* 1991). Third, additional knowledge (*e.g.* taxonomical hierarchies) is often available during generalization. Many learning methods take such knowledge into account in the generalization process (Bisson 1990; Clark *et al.* 1993; De Raedt *et al.* 1992a; Kodratoff 1990; Muggleton *et al.* 1992b; Rouveirol 1991). LGG does not use any background knowledge (BK) in the generalization process. RLGG is supposed to address knowledge-

driven generalization, but since RLGG compiles all the knowledge during generalization in the form of additional literals, this compounds the inefficiency problems of LGG.

This chapter presents an alternative to LGG that exploits the context in which LGG is applied. The *context* is meant here to include both the additional knowledge available during generalization, as well as the similarity of literals in the context of the clauses being generalized. I refer to this generalization technique as *Contextual LGG* (CLGG).

To extend LGG so that context is taken into account during generalization, the similarity between constants (or arguments) occurring in the two clauses being generalized is computed before the generalization. The similarity is a measure between arguments of literals that takes into account their occurrences in clauses. Constant bindings with a similarity higher than a threshold are bound to a variable. The constants and the variable constitute the *similarity bindings*, which are then passed to the CLGG to limit its search. When generalizing two clauses, literals that *match*²² (even with multiple occurrences) must have at least one similarity binding in order to be generalized. Moreover, to take into account the context in which the generalization of two clauses takes place and to address the shortcomings of RLGG, the BK is used in a lazy manner. Only unmatched literals restricted with similarity bindings that find a generalization in the BK are generalized.

The new concept of CLGG is presented and its underlying theory is shown in Section 5.1. The similarity measure is introduced in Section 5.2. CLGG is compared to Plotkin's LGG in Section 5.3. CLGG relative to the BK is compared to Plotkin's RLGG in Section 5.4.

5.1 Theory underlying CLGG

The main difference between CLGG and Plotkin's LGG (Plotkin 1970; Plotkin 1971) is that CLGG can generalize functors to functor variables. *Functor variables* allow CLGG to preserve relationships between arguments in the generalized clause and avoid

²² Two literals match if they have the same predicate and arity.

overgeneralization. This moves the representation of the generalization from first-order to second-order logic, or – more exactly – to a subset of second-order logic with variable functor symbols (but not variable predicate symbols).

This section first introduces preliminary definitions for first-order logic. These definitions are then extended to apply to variable functor symbols. Extended definitions are used to define θ_F -subsumption lattice, and the LGG under θ_F -subsumption (LGG_F). That lattice is then used to show that the CLGG of clauses corresponds to their LGG_F , and to show that CLGG is as specific as Plotkin's LGG.

I assume the reader is familiar with the basic notions and notations in Logic Programming (Lloyd 1984) or Automatic Theorem Proving (Chang *et al.* 1987), although I give necessary definitions when needed. The language used is restricted to clauses and sets of clauses. To a large extent I follow definitions in (Lloyd 1984; Chang *et al.* 1987).

5.1.1 Preliminary definitions

Definition A *simple expression* is either a term or a literal. An *expression* is either a simple expression or a finite set of simple expressions.

Definition The set of variables occurring in any syntactic object o , is denoted by $vars(o)$. A *substitution*, $\theta = \{v_1/t_1, \dots, v_n/t_n\}$ uniquely maps terms in o to variables v_i , where each v_i is a variable, each t_i is a term distinct from v_i and the variables $v_1, \dots, v_n$ are distinct. A substitution is applied to a term by replacing all occurrences of each v_i by the corresponding term t_i . The set of variables $\{v_1, \dots, v_n\}$ is called the *domain*(θ).

Definition Let $E1$ and $E2$ be expressions. $E1$ and $E2$ are *variants*, denoted $E1 \cong E2$, iff they differ only in the names of their variables, *i.e.* iff there exist substitutions λ and σ such that $E1 = E2\lambda$ and $E2 = E1\sigma$. In that case $E1$ is a variant of $E2$ ($E1 \cong E2$) and $E2$ is a variant of $E1$ ($E2 \cong E1$).

Definition Let $\lambda = \{x_1/y_1, \dots, x_n/y_n\}$ be a substitution, E an expression and V the set of variables occurring in E (*i.e.* $vars(E)$). Then λ is a *renaming substitution* for E iff $y_1, \dots, y_n$ are distinct variables and $(V - \{x_1, \dots, x_n\}) \cap \{y_1, \dots, y_n\} = \{\}$.

Clauses that are variants are not regarded as distinct since they are equal up to variable renaming.

Definition Let $\lambda = \{v_1/t_1, \dots, v_n/t_n\}$ and $\sigma = \{u_1/s_1, \dots, u_m/s_m\}$ be substitutions. Then the **composition** $\lambda\sigma$ of λ and σ is the substitution obtained from

$$\{v_1/t_1\sigma, \dots, v_n/t_n\sigma, u_1/s_1, \dots, u_m/s_m\}$$

by deleting any binding $v_i/t_i\sigma$ for which $v_i = t_i\sigma$ and deleting any binding u_j/s_j for which u_j belongs to $\{v_1, \dots, v_n\}$ ($1 \leq i \leq n$, $1 \leq j \leq m$).

Example:

$$\begin{aligned} \lambda &= \{X/f(Y), Y/Z\} \\ \sigma &= \{X/a, Y/b, Z/Y\} \\ \lambda\sigma &= \{X/f(Y)\sigma, Y/Z\sigma, X/a, Y/b, Z/Y\} \\ &= \{X/f(b), \cancel{Y/Y}, \cancel{X/a}, \cancel{Y/b}, Z/Y\} \\ &= \{X/f(b), Z/Y\} \end{aligned}$$

Definition Let $C1$ and $C2$ be two clauses. If $C1$ and $C2$ have no variables in common, then they are said to be **standardized apart**.

5.1.2 Generalization under θ_F -subsumption

θ_F -subsumption

To apply to variable functor symbols, the above definition of a substitution is extended to accept terms with functor variables as follows.

Definition A **substitution** $\theta_F = \{v_1/t_1, \dots, v_n/t_n\}$ uniquely maps terms in $\mathcal{o}$ to variables, where each v_i is a variable, each t_i is a term distinct from v_i and the variables $v_1, \dots, v_n$ are distinct. Function variables and terms with functor variables are also accepted as terms.

The definition for variant expressions stays unchanged, but here is a new example of variant expressions with functors.

Example:

$C1:$ $p(f(X, Y), G(Z), a)$
 $C2:$ $p(f(U, V), H(W), a)$
 $C3:$ $p(F(X, X))$
 $C4:$ $p(F(X, Y))$

$C1$ is a variant of $C2$ since $C1\{X/U, Y/V, Z/W, G/H\} = C2$ and $C2\{U/X, V/Y, W/Z, H/G\} = C1$. But $C3$ is not a variant of $C4$ since there is no substitution σ such that $C3\sigma = C4$.

Now, the generality relation θ_F -subsumption can be defined as follows.

Definition A clause $C1$ θ_F -subsumes a clause $C2$, denoted $C1 \leq_F C2$, iff there exists a substitution θ_F such that $C1\theta_F \subseteq C2$. We say that $C1$ is a **generalization under θ_F -subsumption** of $C2$.

θ_F -subsumption is reflexive and transitive.

Example: Consider the following clauses:

$C1:$ $\leftarrow q(F(X)), q(Z)$
 $C2:$ $p(X) \leftarrow q(F(X))$
 $C3:$ $p(a) \leftarrow q(f(a)), r(b)$

$C1 \leq_F C1$ since every clause is a subset of itself. Also $C1 \leq_F C2$ since $C1\theta_F \subseteq C2$ where $\theta_F = \{Z/f(X)\}$, and $C2 \leq_F C3$ since $C2\sigma \subseteq C3$ where $\sigma = \{F/f, X/a\}$. Then $C1 \leq_F C3$ since $C1\theta_F\sigma \subseteq C3$.

Subsumption is the most commonly used form of generalization of clauses (Idestam-Almquist 1993). Buntine (1988) describes an algorithm aimed at computing the generality relationship for arbitrary pairs of theories (set of clauses), which he terms **generalized subsumption**. For a fuller discussion of the subject of generality the reader is referred to (Niblett 1988).

Testing for θ_F -subsumption

Given two expressions $E1$ and $E2$ which have no variables in common, the substitution θ_F is a **θ_F -difference** when $E1\theta_F = E2$ and $\text{domain}(\theta_F) \subseteq \text{vars}(E1)$. The θ_F -difference in

undefined otherwise. When it is defined it is written as the infix relation $\theta_F = E1 -_{\theta_F} E2$. In fact, the θ_F -difference between two expressions $E1$ and $E2$ is unique and can be defined recursively.

• $v_l -_{\theta_F} T = \{v_l/T\}$	if v_l is a variable and T is a term;
• $f(r_1, \dots, r_n) -_{\theta_F} f(s_1, \dots, s_n) = \cup(r_i -_{\theta_F} s_i)$ for $1 \leq i \leq n$	if f is a predicate or a functor symbol with arity n ;
• $F(r_1, \dots, r_n) -_{\theta_F} g(s_1, \dots, s_n) =$ $\{F/g\} \cup (r_i -_{\theta_F} s_i)$ for $1 \leq i \leq n$	if F is a functor variable and g a functor symbol with arity n ;
• $E1 -_{\theta_F} E2$ is undefined	if none of the above.

Figure 5.1. θ_F -difference of two expressions $E1$ and $E2$.

To test if a clause $C1$ θ_F -subsumes another clause $C2$, the definition of θ_F -difference presented in (Muggleton *et al.* 1992a) is extended to manipulate functor variables in Figure 5.1. When $E1 -_{\theta_F} E2$ is defined then $E1$ θ_F -subsumes $E2$.

Theorem 1 Decidability of θ_F -subsumption between clauses

Let $C1$ and $C2$ be two clauses. Then there exists a procedure to decide if $C1$ θ_F -subsumes $C2$.

Proof:

Suppose $C1$ and $C2$ are two Horn clauses and $C1 = \{C1_1, \dots, C1_n\}$ where $C1_i$ for $1 \leq i \leq n$ are literals $\in C1$, and $C2 = \{C2_1, \dots, C2_m\}$ where $C2_j$ for $1 \leq j \leq m$ are literals $\in C2$. There are two cases for the θ_F -difference of two clauses.

1) Suppose $C1$ and $C2$ are in a θ_F -subsumption relationship.

For every $CI_i \in C1$, there exists a $C2_j \in C2$ such that $CI_i -_{\theta_F} C2_j$ is defined and used throughout. So for any other pair CI_i and $C2_j$, either there exists already a $CI_i -_{\theta_F} C2_j$ of their terms or $CI_i -_{\theta_F} C2_j$ is defined. Then $C1 -_{\theta_F} C2$ is defined and is equal to $\{\cup (CI_i -_{\theta_F} C2_j) \mid CI_i \in C1 \text{ and } C2_j \in C2\}$.

2) Suppose $C1$ and $C2$ are not in a θ_F -subsumption relationship.

For at least one $CI_i \in C1$ there exists no $C2_j$ such that $C2_j \in C2$ and $CI_i -_{\theta_F} C2_j \neq \text{undefined}$. Then $C1 -_{\theta_F} C2$ is *undefined*. •

The θ_F -difference of two expressions is used to test if a clause θ_F -subsumes another clause. The **θ_F -subsumption test** is performed by checking if each literal in the first clause has a θ_F -difference with some literals in the second clause, using the same substitution throughout. The θ_F -difference of two expressions is unique up to variable renaming.

The relation θ_F -subsumption between clauses is *decidable*, which means that it is always possible to determine whether a clause θ_F -subsumes another clause. If clauses contain no recursion, termination is guaranteed (Buntine 1988). In fact, generalization under implication seemed more appropriate than generalization under subsumption to learn recursive clauses (Idestam-Almquist 1993; Idestam-Almquist 1995; Nienhuys-Cheng *et al.* 1996). The subproblem of testing whether a clause subsumes another clause has been shown to be *NP*-complete (Garey *et al.* 1979; Kapur *et al.* 1986). Subsumption algorithms are studied in (Gottlob 1985).

Example:

$E1: \quad p(F(a, Z), b, Z)$
 $E2: \quad p(f(a, g(Y)), b, g(Y))$
 $E3: \quad p(g(a, b), c, d)$

Although the three expressions ($E1$, $E2$, and $E3$) are *compatible*²³, the only θ_F -subsumption found between them is that $E1$ θ_F -subsumes $E2$, since $E1 -_{\theta_F} E2 =$

²³ Literals are *compatible* if they have the same predicate symbol and the same arity (*i.e.* number of arguments).

$\{F/f, Z/g(Y)\}$. There exist no substitutions such that $E1$ θ_F -subsumes $E3$ or that $E2$ θ_F -subsumes $E3$.

Least general generalization under θ_F -subsumption: LGG_F

Definition A clause $C1$ is a **generalization under θ_F -subsumption** of a set of clauses $S = \{C2_1, \dots, C2_n\}$ iff, for every $C2_i$, $1 \leq i \leq n$, $C1 \leq_F C2_i$. A generalization under θ_F -subsumption $C1$ of S is a least general generalization under θ_F -subsumption (LGG_F) of S iff, for every generalization under θ_F -subsumption $C1'$ of S , $C1' \leq_F C1$.

Example: consider the following clauses:

- $C1: \quad p(a) \leftarrow q(f(a)), q(f(b))$
- $C2: \quad p(b) \leftarrow q(f(b)), q(f(X))$
- $C3: \quad p(X) \leftarrow q(f(X)), q(f(b))$
- $C4: \quad p(Y) \leftarrow q(f(Y)), q(f(b)), q(f(Z)), q(f(W))$

Both clauses $C3$ and $C4$ are LGG_F of $\{C1, C2\}$, since $C3\{X/a\} \subseteq C1$, $C3\{X/b\} \subseteq C2$, and $C4\{Y/a, Z/a, W/b\} \subseteq C1$, and $C4\{Y/b, Z/X, W/X\} \subseteq C2$.

In general, an LGG_F is not unique, as shown by examples above.

Definition Two clauses are **equivalent under θ_F -subsumption**, denoted $C1 \sim C2$, iff $C1 \leq_F C2$ and $C2 \leq_F C1$.

Example: Consider the following clauses:

- $C1: \quad p(X) \leftarrow q(F(X, Y)), q(f(Y, X)), q(f(X, X))$
- $C2: \quad p(X) \leftarrow q(f(X, X))$

$C1 \leq_F C2$ since $C1\lambda \subseteq C2$ where $\lambda = \{F/f, Y/X\}$, $C2 \leq_F C1$ since $C2$ is a subset of $C1$. Hence, $C1 \sim C2$ and still $C2 \not\cong C1$.

In the example above, the two clauses are equivalent under θ_F -subsumption, but are not variants. In clause $C1$ literals $q(f(Y, X))$ and $q(f(X, X))$ are redundant. An algorithm to remove redundant literals in a clause will be described later.

Existence of LGG_F

For every set of clauses there exists a clause that θ_F -subsumes all clauses in the set, since the empty clause θ_F -subsumes every clause. The empty clause is the only clause that is not θ_F -subsumption equivalent to any other clause, and thus it is the only clause that can be a unique LGG_F .

Example: Consider the following clauses

$$C1: \quad p(a) \leftarrow q(a)$$

$$C2: \quad p(b) \leftarrow r(b)$$

$$E: \quad s(X) \leftarrow r(X)$$

The empty clause is an LGG_F of $\{C1, C2, E\}$.

Theorem 2 Uniqueness of LGG_F

Let S be a set of clauses (or sets of equivalent clauses) partially ordered under θ_F -subsumption, and let $C1 \in S$ and $C2 \in S$. Then if $C1$ and $C2$ have a LGG_F (or the *least upper bound*) E , then they have a unique LGG_F up to the equivalence. Similarly, the *greatest specific specializations* (GSS_F) (or the *greatest lower bound*) are unique.

Proof:

Without loss of generality, clauses in S are standardized apart. Suppose that E_1 and E_2 are both LGG_F for $C1$ and $C2$. Using that with the definition of a generalization, $E_1 \leq_F C1$ and $E_1 \leq_F C2$, and also $E_2 \leq_F C1$ and $E_2 \leq_F C2$. With the definition of LGG_F , $E_1 \leq_F E_2$ and $E_2 \leq_F E_1$ (taking $C1$ first as E_1 , then as E_2). Then I can conclude that $E_1 \sim E_2$ because of the definition of equivalent clauses. So, LGG_F is unique up to the equivalence induced under θ_F -subsumption. •

The proof for uniqueness of GSS_F 's is dual to that just given. •

Reduction under θ_F -subsumption

Reducing clauses under θ_F -subsumption removes redundancies and will prove useful for CLGG (Section 5.3).

Definition The *reduced clause* $C1$ of a clause $C2$ is a minimal subset of literals such that $C1$ is equivalent to $C2$. In other words, $C1$ is reduced if it is equivalent to any proper subset of itself. A clause $C1$ is a *reduction* under θ_F -subsumption of a clause $C2$ iff $C1 \subseteq C2$, $C1 \sim C2$ and $C1$ is reduced under θ_F -subsumption.

Example Consider the following clauses:

$C1:$ $p(X) \leftarrow q(F(X, X)), q(F(X, Y))$

$C2:$ $p(X) \leftarrow q(F(X, X))$

$E:$ $p(X) \leftarrow q(F(X, X)), r(F(X, Y))$

Clauses $C2$ and E are reduced under θ_F -subsumption. Clause $C2$ is also a reduction under θ_F -subsumption of $C1$, since $C1 \sim C2$ (i.e. $C1\{Y/X\} = C2$). But $C2$ is not a reduction under θ_F -subsumption of E since $C2 \not\sim E$.

The reduction algorithm is not needed to remove unconnected literals in a generalization, since these literals are avoided by the use of similarity bindings. On the other hand, the reduction algorithm may be useful when CLGG generalizes clauses with the same duplicate relations. $C1$ (above) is an example of such resulting generalization. For this thesis, CLGG is used (in CLUSE) to generalize clauses with only one relation (i.e. chains). Therefore the reduction algorithm will not be used.

An algorithm for reducing a clause is given below in Theorem 3. The algorithm is the same as Plotkin's (Plotkin 1970), but it is based on θ_F -subsumption rather than θ -subsumption. As also described in (Buntine 1988), each expression needs to be considered for reduction, only once in the entire course of the algorithm. As with θ -subsumption, the algorithm inherits termination problems for the θ_F -subsumption tests performed in Step 2. Because the input is finite in length, the algorithm is guaranteed to terminate with a clause $C2$ set to a reduced clause form of $C1$ if all subsumption tests terminate. Other algorithms to remove redundancy from a clause can be found in (Muggleton *et al.* 1992b; Gottlob 1993).

Theorem 3 Reduction under θ_F -subsumption

Suppose $C1$ and $C2$ are two clauses. If $C1 \sim C2$, and $C1$ and $C2$ are reduced, then $C1 \cong C2$. The following algorithm gives a reduced subset E , of $C1$ such that $E \sim C1$, assuming all θ_F -subsumption tests terminate.

1. Set E to $C1$.
2. For each atom A in the body of $C1$, if $E - (\neg A) \theta_F E$, set E to $E - (\neg A)$.

A proof of theorem can be found in (Plotkin 1970).

The complexity of this algorithm depends on the complexity of subsumption.

 θ_F -subsumption lattice

The relation $\leq_F$ introduces a **lattice** on the set of reduced clauses. This means that any two clauses have an LGG_F (and a GSS_F). Both of them are unique up to variable renaming.

As in Plotkin's lattice there exist infinite ascending chains in the θ_F -subsumption lattice.

For example:

$$\begin{aligned} p(X1) &\leftarrow q(X1, X2) \\ p(X1) &\leftarrow q(X1, X2), r(X2, X3) \\ p(X1) &\leftarrow q(X1, X2), r(X2, X3), s(X3, X4) \quad \dots \end{aligned}$$

There also exist infinite descending chains in the lattice.

For example:

$$\begin{aligned} p(X1) &\leftarrow q(X1, X2), r(X2, X1) \\ p(X1) &\leftarrow q(X1, X2), r(X2, X3), s(X3, X1) \\ p(X1) &\leftarrow q(X1, X2), r(X2, X3), s(X3, X4), t(X4, X1) \quad \dots \end{aligned}$$

Unlike Plotkin's lattice, for each of the above chains, there exist infinite chains for different nestings of arguments.

For example, for the above chain $p(X1) \leftarrow q(X1, X2)$, there exist infinite chains of the form:

$$\begin{aligned} p(X1) &\leftarrow q(X1, F(X2)) \\ p(X1) &\leftarrow q(X1, F(G(X2))) \quad \dots \end{aligned}$$

All clauses in both infinite series θ_F -subsume the clause $p(X) \leftarrow q(X, X)$ and are θ_F -subsumed by the clause $p(X) \leftarrow q(X, Y)$.

5.1.3 CLGG: the operator to find the LGG_F of two clauses in the lattice

CLGG of two terms:

1. $CLGG(t, t) = t$ where t represents a constant
2. $CLGG(s_n, t_n) = V$ where s_n and t_n represent constants and V is a variable²⁴
and $\varphi_1 = \{V/s_n\}$, $\varphi_2 = \{V/t_n\}$
3. $CLGG(V1, t) = V2$ where V represents a variable and t represents a term, and $V2$ is a variable
and $\varphi_1 = \{V2/V1\}$, $\varphi_2 = \{V2/t\}$
4. $CLGG(f(s_1, \dots, s_n), f(t_1, \dots, t_n))$ where f represents a functor symbol
 $= f(CLGG(s_1, t_1), \dots, CLGG(s_n, t_n))$ with arity n
and $\varphi_1 = \cup \varphi_{1j}$ for $i = \{1, 2\}$ and $1 \leq j \leq n$
5. $CLGG(f(s_1, \dots, s_n), g(t_1, \dots, t_n))$ where f and g represent functor symbols with
 $= F(CLGG(s_1, t_1), \dots, CLGG(s_n, t_n))$ arity n , and F is a functor variable
and $\varphi_1 = \{F/f\} \cup \varphi_{1i}$ for $1 \leq i \leq n$
 $\varphi_2 = \{F/g\} \cup \varphi_{2i}$
6. $CLGG(F(s_1, \dots, s_n), f(t_1, \dots, t_n))$ where F represents a functor variable and
 $= G(CLGG(s_1, t_1), \dots, CLGG(s_n, t_n))$ f represents a functor
and $\varphi_1 = \{G/F\} \cup \varphi_{1i}$ symbol with arity n for $1 \leq i \leq n$
 $\varphi_2 = \{G/f\} \cup \varphi_{2i}$
7. if none of the above cases apply then $CLGG = \{\}$

²⁴ A variable V (or F) is chosen distinct from any in φ_1 and φ_2 , and represents this pair of terms (or functions) throughout the generalization of the two clauses.

CLGG of two literals

8. $CLGG(p(s_1, \dots, s_n), p(t_1, \dots, t_n))$ where p is a predicate symbol with arity n
 $= p(CLGG(s_1, t_1), \dots, CLGG(s_n, t_n))$ for $i = \{1, 2\}$ and $1 \leq j \leq n$
 and $\varphi_i = \cup \varphi_{ij}$
9. if none of the above cases apply then $CLGG = \{\}$

CLGG of two clauses:

Suppose $C1$ and $C2$ are two Horn clauses and

$C1 = \{C1_1, \dots, C1_n\}$ where $C1_i$ for $1 \leq i \leq n$ are literals $\in C1$, and
 $C2 = \{C2_1, \dots, C2_m\}$ where $C2_j$ for $1 \leq j \leq m$ are literals $\in C2$.

10. $CLGG(C1, C2) = \{CLGG(C1_i, C2_j) \mid C1_i \in C1 \text{ and } C2_j \in C2\}$
 and substitutions $\varphi_{C1} = \cup \varphi_{1i}$ and $\varphi_{C2} = \cup \varphi_{2j}$
11. if none of the above cases apply then $CLGG = \{\}$

Figure 5.2. CLGG of two clauses $C1$ and $C2$.
 Only cases 5 and 6 differ from Plotkin.

The CLGG algorithm (Figure 5.2) computes the LGG_F of two clauses under θ_F -subsumption (*i.e.* where functor symbols can be variables). This procedure differs from Plotkin's LGG only when intermediate levels of literals (*i.e.* functor symbols) are different (cases 5 and 6).

Example:

$$\begin{array}{lcl}
 C1: & & p(X) \leftarrow p(f(a, g(X)), b, g(X)) \\
 C2: & & p(Y) \leftarrow p(g(a, g(Y)), b, g(Y)) \\
 \hline
 CLGG(C1, C2) = & p(Z) \leftarrow p(F(a, g(Z)), b, g(Z)) \text{ and } & \varphi_{C1} = \{Z/X, F/f\}, \\
 & & \varphi_{C2} = \{Z/Y, F/g\}
 \end{array}$$

A computed CLGG of two Horn clauses is always a Horn clause, since a positive literal can only be a generalization of positive literals. Hence there exists a CLGG that is a Horn clause generalization of any two Horn clauses. Although the above definition could easily be extended to generalize a set of clauses, CLUSE uses CLGG to generalize two clauses at a time (Chapter 6).

Theorem 4 proves that:

- 1) CLGG G of two clauses corresponds to their LGG_F in the lattice;
- 2) if there exists another CLGG G' of the same two clauses then $G' \leq_F G$.

Theorem 4 The CLGG of two clauses is also their LGG_F .

I will show that $\text{CLGG}(C1, C2) = \text{LGG}_F(C1, C2)$, where $C1$ and $C2$ are two Horn clauses.

Proof:

Without loss of generality, clauses $C1$ and $C2$ are standardized apart. Suppose that $G = \text{CLGG}(C1, C2)$ is not their LGG_F . This means that $G \not\rightarrow_{\leq_F} C1$ or $G \not\rightarrow_{\leq_F} C2$, which implies that there exist no substitutions λ and σ such that $G\lambda \subseteq C1$ or $G\sigma \subseteq C2$. From the definition of the CLGG of two clauses:

$$\text{CLGG}(C1, C2) = \{\text{CLGG}(l_1, l_2) \mid l_1 \in C1 \text{ and } l_2 \in C2\} = G$$

with the substitutions φ_{C1} and φ_{C2} .

From the definition of the CLGG of compatible literals:

$$\text{CLGG}(l_1, l_2) = L \text{ with substitutions } \varphi_1 \text{ and } \varphi_2, \text{ where}$$

$$\begin{aligned} L &= \text{CLGG}(p(s_1, \dots, s_n), p(t_1, \dots, t_n)) \\ &= p(\text{CLGG}(s_1, t_1), \dots, \text{CLGG}(s_n, t_n)) \end{aligned}$$

$$\varphi_1 = \{v_1/s_1, \dots, v_n/s_n\}$$

$$\varphi_2 = \{v_1/t_1, \dots, v_n/t_n\}$$

where v_i are variables in L , s_i and t_i are terms that are distinct in l_1 and l_2 respectively. Using the substitution φ_1 , it is possible to restore l_1 from L with $L\varphi_1 = l_1$ (and similarly for l_2 from L with $L\varphi_2 = l_2$). Only compatible literals are generalized (*i.e.* unmatched literals are ignored during the generalization). At the level of clauses this means that if $C1$ and $C2$ have no compatible literals their LGG_F is *undefined* and $\text{CLGG}(C1, C2)$ returns the *empty clause*. If $C1$ and $C2$ have compatible literals then there exists only one substitution λ such that $G\lambda \subseteq C1$, where $\lambda = \varphi_{C1}$. Similarly, there exists only one

substitution σ such $G\sigma \subseteq C2$, where $\sigma = \varphi_{C2}$. Moreover, G is reduced with the reduction algorithm (Theorem 3). Therefore there exists a unique substitution (up to the variable renaming) between G and C_i , $i = 1, 2$, which contradicts $G \not\leq_F C1$ or $G \not\leq_F C2$. Hence $G \leq_F C1$ or $G \leq_F C2$, so restoring l_1 from L with $L\varphi_1 = l_1$ and l_2 from L with $L\varphi_2 = l_2$ results in their reduced generalization (*i.e.* their LGG_F). •

Theorem 5 The CLGG of two clauses is as specific as their Plotkin's LGG

The CLGG of two clauses $C1$ and $C2$ is as specific as their generalization using Plotkin's LGG.

Proof:

Without loss of generality, clauses $C1$ and $C2$ are standardized apart. Suppose that $CLGG(C1, C2)$ is more general than $LGG(C1, C2)$, then $CLGG(C1, C2) \leq_F LGG(C1, C2)$. So there must exist a substitution θ_F such that $CLGG(C1, C2)\theta_F \subseteq LGG(C1, C2)$. A generalization exists if there exists at least two compatible literals l_1 and l_2 , where $l_1 \in C1$ and $l_2 \in C2$. Otherwise the *empty clause* is returned as for their LGG. I distinguish two cases.

1) CLGG differs from LGG. Using the algorithm, CLGG differs from LGG when:

a) l_1 and l_2 have different functor symbols with the same arity n :

$$l_1: \quad p(f(s_1, \dots, s_n))$$

$$l_2: \quad p(g(t_1, \dots, t_n))$$

$$CLGG(l_1, l_2) = G \text{ with substitutions } \varphi_1 \text{ and } \varphi_2, \text{ where}$$

$$G = p(F(CLGG(s_1, t_1), \dots, CLGG(s_n, t_n)))$$

$$\varphi_1 = \{F/f\} \cup \varphi_{1i}$$

$$\varphi_2 = \{F/g\} \cup \varphi_{2i} \quad \text{for } 1 \leq i \leq n$$

$$LGG(l_1, l_2) = L \text{ with substitutions } \delta_1 \text{ and } \delta_2, \text{ where}$$

$$L = p(X)$$

$$\delta_1 = \{X/f(s_1, \dots, s_n)\}$$

$$\delta_2 = \{X/g(t_1, \dots, t_n)\}$$

In that case, there exists no substitution θ_F such that $p(F(CLGG(s_1, t_1), \dots, CLGG(s_n, t_n)))\theta_F \subseteq p(X)$, so $G\theta_F \not\subseteq L$, which contradicts that $CLGG(C1, C2) \leq_F LGG(C1, C2)$.

b) l_1 and l_2 have the same arity n and one of the two literals l_1 and l_2 has a functor variable:

$$\begin{aligned}
 l_1: & \quad p(F(s_1, \dots, s_n)) \\
 l_2: & \quad p(f(t_1, \dots, t_n)) \\
 CLGG(l_1, l_2) &= G \text{ with substitutions } \varphi_1 \text{ and } \varphi_2, \text{ where} \\
 G &= p(G(CLGG(s_1, t_1), \dots, CLGG(s_n, t_n))) \\
 \varphi_1 &= \{G/F\} \cup \varphi_{1i} \\
 \varphi_2 &= \{G/g\} \cup \varphi_{2i} \quad \text{for } 1 \leq i \leq n \\
 LGG(l_1, l_2) &= \text{undefined}
 \end{aligned}$$

Therefore, there exists no substitution θ_F such that $G\theta_F \subseteq L$, which also contradicts that $CLGG(C1, C2) \leq_F LGG(C1, C2)$ and completes the proof.

2) CLGG is the same as LGG, *i.e.* all other cases not covered in 1):

Assuming that the previous cases (in 1) do not appear. The CLGG of two clauses is unique (theorem 2); the LGG of two terms is unique (Idestam-Almqvist 1993). Since the procedures to generalize these expressions are exactly the same in CLGG and LGG, then $G \cong L$. Therefore, this also contradicts that $CLGG(C1, C2) \leq_F LGG(C1, C2)$ and completes the proof. •

5.2 Similarity measure evaluates bindings

5.2.1 Similarity measure of constants

The similarity measure limits the search performed during computation of CLGG, by giving to it the similarity bindings to take into account while generalizing clauses. Similarity bindings consist of a binding's constants and a variable. Constants come from

each clause and result in a variable when their similarity value is higher than the preset similarity threshold²⁵.

My *similarity measure* evaluates bindings of clauses that may have terms with functors (i.e. nested terms). Although this is a new similarity measure, the underlying idea comes from (Bisson 1990; Vrain 1990; Bisson 1992). It counts common predicates of constants belonging to two clauses. Bindings for numeric or nominal values are ignored, although if included they would simply increase the actual measure. Moreover, arguments that are user-supplied as nominal values are ignored in the computation of the similarity of bindings.

To count common predicates, lists of occurrences for each constant are used. To handle nested terms, these lists are made of triples instead of pairs. For each constant a_i , a list of occurrences (denoted by $occ(a_i)$) is made of triples ($predicate-a_i$, $position-of-a_i$, $level-of-a_i$) for each literal where the constant occurs. The $predicate-a_i$ is the literal's predicate, the $position-of-a_i$ is the term's position among the arguments of $predicate-a_i$, and the $level-of-a_i$ is the nesting level to reach the constant. For each literal, each nested term is recursively enumerated in the same way, by the triple ($functor-a_i$, $position-of-a_i$, $level-of-a_i$), where $functor-a_i$ is the functor symbol of the nested term. This enumeration is repeated until the constant is reached.

Table 5.1. shows lists of occurrences for each constant of clauses $C1$ and $C2$. For example $occ(a)$ shows that the constant a occurs in the literal with the predicate *on*, in its first argument's term, three levels deep (hence *on13*). There are two nested terms in the first argument term to reach the constant a : 1) a occurs in the first argument of functor *size* two levels deep (hence *size12*), and 2) a occurs in the first argument of functor *rectangle* one level deep (hence *rectangle11*). The constant a also occurs in the first argument of predicate *red* (hence *red11*).

²⁵ The default value for the similarity threshold is 0.5. This parameter can be modified by the user.

C1: on(size(rectangle(a), large), size(circle(b), small)),
red(a), blue(rectangle (c))

C2: on(size(triangle(e), giga), size(ellipse(d), micro)),
red(e), blue(triangle(f))

Clause	A_i	$occ(A_i)$
C1	a	(on13, size12, rectangle11),(red11)
	b	(on23, size12, circle11)
	c	(blue12, rectangle11)
C2	d	(on23, size12, ellipse11)
	e	(on13, size12, triangle11),(red11)
	f	(blue12, triangle11)

Table 5.1. List of occurrences for constants.

Nominal values small and micro are ignored for bindings. The example is adapted from (Bisson 1990).

Two constants match if they occur in two literals whose predicates are identical. Matching constants can occur in terms whose functors are different²⁶. The similarity between two constants from two clauses is the ratio of the length of the lists of common occurrences to the maximum length of constants' occurrences in the two clauses. It results in a value between [0..1], where the closer the value gets to 1, the more similar the constants are. The *similarity measure* formula²⁷ is:

$$sim(a_i, a_j) = \frac{length(occ(a_i) \cap occ(a_j))}{MAX(length(occ(a_i)), length(occ(a_j)))}$$

The overall idea of this definition is that constants occurring in two clauses are similar if they occur in a similar enough context. It is required that occurrences of constants in relations (*i.e.* literals with more than one argument) must match, otherwise their similarity binding is zero²⁸.

²⁶ Since CLGG generalizes functors to functor variables when they differ.

²⁷ To take predicate weights into account as suggested in Bisson (1992), *length* in the formula is simply changed to $\sum weight$.

a_i	a_j	$sim(a_i, a_j)$	Variable binding
a	d	0	-
	e	1	V1
	f	0	-
b	d	1	V2
	e	0	-
	f	0	-
c	d	0	-
	e	0	-
	f	1	V3

Table 5.2. Similarity bindings of constants.

A variable binding is associated with constants with a similarity value higher than the similarity threshold.

Table 5.2 shows the similarity measure with nested terms of previous example (Table 5.1.). The similarity value for the binding a and d is 0, because their positions in the relation *on/2* differ. Recall that matching of constants may occur in terms whose functors are different (as long as the positions and the levels are the same). The similarity value of the binding (a, e) is 1, because both constants occur in the first position of the predicate *on*, three levels deep: both also occur in the same argument position and at the same level in functors *size* and *rectangle* for the constant a , and *size* and *triangle* for the constant e . Their occurrences in the literal *red* also match. So a and e match on both of the two literals in the predicates *on* and *red*, giving a similarity of 1.

5.2.2 Similarity measure of clauses

When generalizing a set of clauses, it may be useful to know the similarity between clauses. CLUSE uses the *similarity of clauses*²⁹ to choose the two most similar chains to generalize first (Chapter 6). The similarity measure of two clauses is computed from similarity bindings of constants. This measure is used in CLUSE, prior to the application of CLGG. The similarity of two clauses equals the product of their non-zero similarity bindings of constants.

²⁸ This avoids similarity of constants that represent attributes being equal to similarity of constants that occur in a relation. CLGG is used to learn relational clichés where relations are more important than attributes.

²⁹ The similarity of clauses is used prior to CLGG. Since it is computed from similarity bindings of constants, it is appropriate to this chapter.

The formula is:

$$\text{sim}(C1, C2) = \prod_{i=1}^n \prod_{j=1}^m \text{sim}(a1_i, a2_j) \quad \text{for } \text{sim}(a1_i, a2_j) \neq 0$$

where $a1_i \in C1$ and $a1_j \in C2$, n is the number of constants in $a1$, and m is the number of constants in $a2$.

Table 5.3 shows that $C1$ and $C3$ are the most similar clauses with a similarity 1 compared to the other two pairings with a similarity $1/9$.

C1: on(a, b), blue(a), square(a), triangle(b)
 C2: on(c, d), blue(d), square(d), triangle(c)
 C3: on(f, e), blue(f), square(f), triangle(e)

Clause	A_i	$\text{occ}(A_i)$
C1	a	(on11),(blue11),(square11)
	b	(on21),(triangle11)
C2	c	(on11),(triangle11)
	d	(on21),(blue11),(square11)
C3	e	(on21),(triangle11)
	f	(on11),(blue11),(square11)

$\text{sim}(C1, C2)$	$\text{sim}(C1, C3)$	$\text{sim}(C2, C3)$
(a, c) = 1/3	(a, e) = 0	(c, e) = 0
(a, d) = 0	(a, f) = 1	(c, f) = 1/3
(b, c) = 0	(b, e) = 1	(d, e) = 1/3
(b, d) = 1/3	(b, f) = 0	(d, f) = 0
1/9	1	1/9

Table 5.3. Similarity of two clauses
 is the product of non-zero similarity bindings of constants.

5.3 CLGG vs. LGG

As with Plotkin's LGG (Plotkin 1970), the CLGG of two clauses $C1$ and $C2$ (denoted $\text{CLGG}(C1, C2)$) is the *least upper bound* (or the *least general generalization*) of $C1$ and $C2$ in a θ -subsumption lattice. Unlike Plotkin's θ -subsumption lattice, the

θ_F -subsumption lattice allows functor variables (Section 5.1), and therefore has a finer grain.

This section describes the procedure to compute the CLGG of two clauses³⁰. It presents the differences between the use of CLGG and that of LGG, and illustrates the differences with concrete examples.

Figure 5.2 (Section 5.1) describes the CLGG procedure to generalize two clauses. There are two main differences between CLGG and Plotkin's LGG.

- 1) LGG *overgeneralizes* literals where differences occur at the intermediate levels of nested structures (*i.e.* at the functor levels). Unlike LGG, CLGG generalizes functors to functor variables. Furthermore, rather than generalizing all possible matches of literals, CLGG limits its search to matching literals with similarity bindings.
- 2) CLGG preserves the structure of nested terms.

Figure 5.3 shows that LGG overgeneralizes $C1$ and $C2$. It also shows that because CLGG preserves relationships between variables (*i.e.* the structure of nested terms), the CLGG of $C1$ and $C2$ is more specific than their LGG (*i.e.* $LGG(C1, C2)$ θ_F -subsumes $CLGG(C1, C2)$).

LGG overgeneralizes $C1$ and $C2$ because it loses the relationships between arguments and functors occurring in a clause. For example, the relationship between arguments in $rectangle(a)$ and $red(a)$ in $C1$, and $triangle(e)$ and $red(e)$ in $C2$ disappears when generalizing $rectangle(a)$ and $triangle(e)$ into $V6$. Furthermore, the relationship between the functors $rectangle$ and $triangle$ also disappears. In fact, the first time they match, they are generalized in $V6$ and the second time in $V8$, since they occur with different arguments. Consequently the relationship between $V6$ and $V8$ is lost.

³⁰ The generalization of a set of clauses is the subject of the next chapter.

C1: `on(size(rectangle(a), large), size(circle(b), small)),
red(a), blue(rectangle(c))).`

C2: `on(size(triangle(e), giga), size(ellipse(d), micro)),
red(e), blue(triangle(f))).`

LGG(C1, C2):

$\Rightarrow$ `on(size(V6, V3), size(V7, V4)), red(V1), blue(V8)`

• bindings:

V1 = (a, e)
V2 = (b, d)
V3 = (large, giga)
V4 = (small, micro)
V5 = (c, f)
V6 = (rectangle(V1), triangle(V1))
V7 = (circle(V2), ellipse(V2))
V8 = (rectangle(V5), triangle(V5))

CLGG(C1, C2):

$\Rightarrow$ `on(size(F1(V1), V4), size(F2(V2), V5)), red(V1), blue(F1(V3)))`

• bindings:

V1 = (a, e) {similarity binding}
V2 = (b, d) {similarity binding}
V3 = (c, f) {similarity binding}
V4 = (large, giga)
V5 = (small, micro)
F1 = (rectangle, triangle)
F2 = (circle, ellipse)

Figure 5.3. θ_F -difference of two clauses C1 and C2.

For CLGG, variables *V1*, *V2*, and *V3* are associated with similarity bindings (Table 5.2). Variables *V4* and *V5* are added to the bindings while generalizing because the functor *size* relates nominal values *large* and *giga* to the similarity binding *V1*, and *small* and *micro* to *V2*. CLGG preserves the relationships lost by the LGG by generalizing the functors *rectangle* and *triangle* in the functor variable *F1*, keeping the argument's variable *V1* for *a* and *e* (hence *F1(V1)*). This way, the relationship of the argument variables *V1* between *rectangle(a)* and *red(a)* in *C1*, and *triangle(e)* and *red(e)* in *C2* still exists. Moreover, even though functors *rectangle* and *triangle* occur in the generalization with different arguments, they are replaced by the same functor variable *F1*.

Figure 5.4 illustrates to what extent similarity bindings limit CLGG search when there exist multiple instances of the same predicate in clauses.

C1: scene(a, b):- above(a, b), triangle(a), rectangle(b), small(b), black(a), black(b). C2: scene(c, d):- above(c, d), square(c), square(d), black(c), black(d).
LGG(C1, C2): $\Rightarrow$ scene(V1, V2):- above(V1, V2), black(V1), black(V2), black(V3), black(V4). - unmatched literals: triangle(a), rectangle(b), small(b), square(c), square(d) - bindings: V1 = (a, c) V2 = (b, d) V3 = (a, d) V4 = (b, c)
CLGG(C1, C2): $\Rightarrow$ scene(V1, V2):- above(V1, V2), black(V1), black(V2). - unmatched literals: triangle(a), rectangle(b), small(b), square(c), square(d) - bindings: V1 = (a, c) {similarity binding} V2 = (b, d) {similarity binding}

Figure 5.4. CLGG is less expensive to apply than LGG.

The predicate *black* occurs twice in clauses *C1* and *C2*. LGG generalizes all possible matches of that predicate, resulting in four *black* predicates in the LGG. With LGG, unmatched literals are dropped. In contrast, CLGG limits the generalization of the *black* predicate to those with one similarity binding (*i.e.* *V1* and *V2*). In CLGG, unmatched literals are used in the generalization relative to BK (Section 5.4), or returned (to CLUSE) with the generalization. As a consequence of using similarity bindings, literals of a generalization are *connected*. The *connection principle* states that variables in the body of a clause must either appear in the head of the clause, or be linked to a variable of the head of a clause through a path of predicates. This ensures that all literals in the body of a clause are related to its head, or that all the conditions for an instance to belong to a

concept are related to the concept itself. Literals produced by CLGG constitute a subset of the literals resulting from LGG. Again, the generalization from the CLGG is more specific than the generalization from LGG (*i.e.* $LGG(C1, C2) \theta_F$ -subsumes $CLGG(C1, C2)$).

To sum up, CLGG offers the following advantages over LGG:

- CLGG avoids overgeneralization of clauses when intermediate levels of literals differ. It preserves relationships between arguments and functors in the generalized clause. CLGG preserves (possibly with variable functor symbols) the structure of nested terms (*i.e.* functors in the arguments) of the generalized literals³¹.
- CLGG may be less expensive to apply than LGG for clauses with repetitive predicates, since it limits the generalization to matching literals with at least one similarity binding. This addresses a very important problem occurring in practice, where examples to learn are often described with few but repetitive literals (Dolsak *et al.* 1992).

5.4 CLGG relative to a BK vs. RLGG

Plotkin (1971) also introduces a notion of LGG relative to some BK, called the *relative least general generalization (RLGG)* to take into account available knowledge during generalization. The **RLGG of two clauses** $C1$ and $C2$ is their *least general generalization* LGG relative to the BK. If the BK is given as arbitrary clauses, LGG is applied on the initial clauses plus all their logical consequences (the Herbrand models (Lloyd 1984; Chang *et al.* 1987)) with the BK. The resulting LGG increases the amount of search during generalization. It may be intractably large, and may even contain an infinite number of literals (Aha 1992; Buntine 1988).

The ILP learning system GOLEM (Muggleton *et al.* 1992b) is based on RLGG but reduces the cost by constraining the BK to a finite Herbrand model. Even using a finite

Herbrand model, the length of the LGGs is exponential in the number of given examples. As a consequence further restrictions are required on the hypotheses. When building the RLGG of a set of examples, GOLEM considers only *ij*-determinate clauses. A clause is *ij*-determinate if it is determinate and each literal in the body of that clause has a maximum depth of *i* and a maximum degree *j* (Muggleton *et al.* 1992b; Dzeroski *et al.* 1992). Since *ij*-determinism does not completely avoid redundant literals in hypotheses, GOLEM uses negative examples and input-output mode of variables as a post-processing step (Muggleton *et al.* 1992b). By using the *ij*-determinism and syntactically *generative*³² background clauses, the length of the LGG of a set of example no longer depends on the number of examples (Bergadano *et al.* 1995).

Other learning systems use BK, but none of them is based on RLGG. FOCL (Silverstein *et al.* 1991) uses explanation-based learning (EBL) to operationalize the target concept. MARVIN, CIGOL, and IRES (Sammur *et al.* 1986; Muggleton *et al.* 1992a; Rouveirol *et al.* 1989) use a destructive saturation, which generalizes clauses instead of completing them with additional literals. ITOU (Rouveirol 1991), CLINT (De Raedt *et al.* 1992a), and Kodratoff's system (Kodratoff 1990) encounter the same problem of inefficiency as RLGG, generating the Herbrand models of the BK (by an exhaustive saturation process).

Unlike other learning systems, CLGG exploits an intensional BK in a lazy way. It first generalizes clauses. It then uses the BK to generalize unmatched literals that have at least one similarity binding.

Appendix III illustrates an example BK taxonomy form. The BK is intensionally defined as, for example `ellipse(X) :- circle(X)`.

Figure 5.5 shows that CLGG generalizes the two clauses *C1* and *C2* more efficiently than RLGG. RLGG which corresponds to LGG of *C1* and *C2* in presence of the BK, is built as follows.

³¹ Notice that CLGG treats clauses with differences at the constant level the same way as LGG.

³² A clause is said to be *syntactically generative* if the variables in its head are a subset of the variables in its body.

C1: scene(a, b):- above(a, b), triangle(a), rectangle(b), small(b), black(a), black(b).
C2: scene(c, d):- above(c, d), square(c), square(d), black(c), black(d).

RLGG(C1, C2):

- ground model of C1 and C2:

C1': scene(a, b):- above(a, b), triangle(a), rectangle(b), small(b), black(a), black(b), polygon(a), convex(a), form(a), polygon(b), convex(b), form(b).

C2': scene(c, d):- above(c, d), square(c), square(d), black(c), black(d), rectangle(c), polygon(c), convex(c), form(c), rectangle(d), polygon(d), convex(d), form(d).

RLGG(C1', C2'):

⇒ scene(V1,V2):- above(V1, V2), black(V1), black(V2), black(V3), black(V4), rectangle(V2), rectangle(V4), polygon(V1), polygon(V2), polygon(V3), polygon(V4), convex(v1), convex(V2), convex(V3), form(V1), form(V2), form(V3), form(V4).

- unmatched literals:
triangle(a), small(b), square(c), square(d)
- bindings:
V1 = (a, c)
V2 = (b, d)
V3 = (a, d)
V4 = (b, c)

CLGG(C1, C2) with BK:

⇒ scene(V1,V2):- above(V1, V2), Polygon(V1), Rectangle(V2), black(V1), black(V2).

- generalized literals:
Polygon(V1) = (triangle(a), square(c))
Rectangle(V2) = (rectangle(b), square(d))
- unmatched literals:
small(b)
- bindings:
V1 = (a, b)
V2 = (b, d)

Figure 5.5. CLGG vs. RLGG.

Predicates with a capital letter are learned using the taxonomy *Form*³³. *Italic literals represent logical consequence relative to the BK. Underlined literals represent taxonomical redundancy.*

³³ From now on, I will use this convention.

In RLGG, the BK is first converted into a *ground model* (all logical consequences are generated), which adds literals shown in *italic* in $C1'$ and $C2'$. LGG is then applied on matching literals of $C1'$ and $C2'$. The resulting generalization has many literals which need to be reduced. It contains taxonomical redundancies (underlined) that represent overgeneralizations with respect to the context. It also contains unconnected literals (literals with argument variables $V3$ and $V4$).

As in Section 5.3, CLGG generalizes literals with at least one similarity binding. It results in a generalization with unmatched literals and bindings. When the BK is available to CLGG, unmatched literals with at least one similarity binding are generalized using the BK. Moreover, a single access is required to find the generalization of literals (Appendix IV). Unmatched literals that are generalized using the BK are dropped and the literal from the BK is added to the generalization. CLGG produces a connected clause³⁴ without logical (or taxonomical) redundancy.

The advantages of CLGG over RLGG are:

- No literals are added to clauses prior to generalization in CLGG
- CLGG restricts its search to matching literals with similarity bindings.
- CLGG produces connected clauses without taxonomical redundancy.

5.5 Conclusion

This chapter presented an extension of LGG (and RLGG) that exploits additional knowledge available during generalization and the similarity of literals in the context of the clauses being generalized. It defined the new θ_F -subsumption lattice with a generality relation θ_F -subsumption and a technique to compute LGG under θ_F -subsumption (LGG_F). It showed that θ_F -subsumption between clauses is decidable, that there exists a LGG_F for every finite set of clauses, and that there exists a unique reduction under θ_F -subsumption

³⁴ A clause is connected if literals in its body are connected.

of every clause. It also showed that there exists a unique LGG_F reduced under θ_F -subsumption of every finite set of clauses. It described CLGG to compute the LGG_F of clauses, and finally showed that CLGG is at least as specific as Plotkin's LGG.

CLGG offers several advantages over LGG and RLGG. CLGG is less expensive to apply. No literals are added to clauses prior to the generalization, and matching is restricted to literals with similarity bindings. CLGG also avoids overgeneralization of clauses when intermediate levels of literals differ and preserves relationships between literals. Finally, CLGG produces connected clauses without logical redundancy.

6 CLUSE learns clichés

There is a general agreement in the ILP community about the need for biases to restrict the hypothesis space of relational learners: languages in CLINT (De Raedt *et al.* 1992a), rule models in MOBAL (Kietz *et al.* 1992) and clichés in FOCL (Silverstein *et al.* 1991) are but a few examples of biases. Furthermore, since biases are often specific to domains of application and are hard to define, it is useful to be able to learn them automatically. None of the existing relational learners learn their biases based on the notion of LGG (or RLGG). CIA-CLINT (Constructive Induction by Analogy) (De Raedt *et al.* 1989b) learns languages by analogy. MAT-MOBAL (model acquisition tool) (Morik 1993) abstracts rule models from rules supplied by the user by turning predicate symbols from the application domain into predicate variables. Silverstein (Silverstein *et al.* 1993) uses a top-down approach in which general clichés with a fixed length (called the unconstrained clichés) produce more specific clichés useful in specific domains.

The goal of this thesis is to learn relational clichés to escape local plateaus encountered by a top-down hill-climbing learner (*e.g.* FOIL (Quinlan 1990)). A **relational cliché** is a

combination of literals to escape local plateaus. I have developed and implemented the learning system called CLUSE to learn such clichés from examples. Examples are expressed in terms of their shortest (or most general) chains and are generalized in a bottom-up manner using CLGG (Chapter 5). Resulting generalizations are expressed with the vocabulary (or literals) specific to the domain. These generalizations are returned as *domain-dependent clichés (DDCs)*. Additionally, CLUSE can generalize DDCs into *domain-independent clichés (DICs)*, where literals are no longer specific to the domain. They are expressed with predicate variables. Unlike Silverstein's clichés (Silverstein *et al.* 1993) restrictions on predicates and variabilization are implicitly expressed in the clichés.

This chapter introduces the notion of chains (Section 6.1). It describes the algorithm to learn clichés (Section 6.2). It also presents results of different evaluations of CLUSE:

- 1) its complexity (Section 6.3);
- 2) its application to the synthetic domain of *blocks* and to the real domain of FEM design (Section 6.4);
- 3) its sensitivity to parameter settings (Section 6.5);
- 4) its capacity to learn syntactic restrictions implicit in examples (Section 6.6).

6.1 Examples are split into chains

The generalization problem of learning clichés consists of finding common parts in examples. In relational domains, examples are represented with different relations and features, which makes it difficult to find a generalization process that will succeed in finding common parts of a set of examples. Moreover, in most of these domains, important concepts are represented by a small number of chains among constants defining examples. For these reasons, CLUSE splits examples into their shortest chains. This is similar to the idea of relational pathfinding (Richards *et al.* 1992). Intuitively, a

chain is a pattern showing how objects are related to one another and how their features are used in examples. So, each relation of an example (which provides the structural information between objects) and all features of the related objects form a chain. Every relation and feature of an example are preserved and some features may occur in more than one chain. Syntactically, a chain from an example consists of a list of ground literals sharing at least one constant³⁵. A chain is thus defined as a connected conjunction of literals where one and only one literal is a relation.

```

E:    on(x, y), cir(x), rect(y), leftof(y, z), iso(z)

c1:   on(x, y), cir(x), rect(y)

c2:   leftof(y, z), rect(y), iso(z)

```

Figure 6.1. Example *E* expressed in terms of chains (*c1* and *c2*).

Figure 6.1 shows the shortest chains *c1* and *c2* made from the example *E*. They are connected combinations of literals with a single relation (i.e. *on*(*x*, *y*) and *leftof*(*y*, *z*)).

The idea of splitting examples in some way is not new. The learning system COSIMA (Herrmann *et al.* 1994) learns floorplan rules from examples representing structure descriptions. Examples are split into *important parts* (predicates with high weight, or relevance), and *additional parts* (other predicates). The first step matches only important parts as the most specific generalizations (MSGs). The second step matches additional parts and completes the initial MSGs adding further generalized facts.

For the purpose of this thesis and because relevant relations are not always connected to the head of examples, chains are used without the example's head. In Figure 6.1, both relation *on*(*x*, *y*) and *leftof*(*y*, *z*) would be related to the head *block*(*x*, *y*). On the other hand, *leftof*(*y*, *z*) would not be connected if the head is *block*(*x*).

³⁵ A **chain** from a generalization consists of lists of literals sharing at least one variable.

6.2 CLUSE's algorithm to learn clichés

Figure 6.2 illustrates the general algorithm of learning DDCs and DICs with CLUSE. Positive and negative examples and optionally the BK are given to CLUSE.

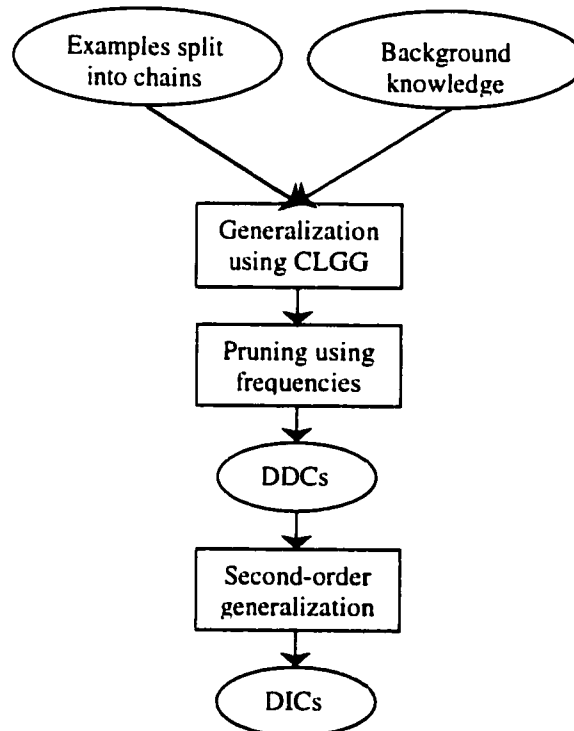


Figure 6.2. DDCs are learned from examples and optionally, BK. CLUSE generalizes chains using CLGG. Generalizations are pruned according to their coverage frequencies. Preserved generalizations are returned as domain-dependent-clichés (DDCs). DDCs can be further generalized into domain-independent clichés (DICs).

CLUSE splits examples into chains. In a bottom-up manner CLUSE generalizes positive chains using CLGG (and the BK when available) (Chapter 5) into a hierarchy of generalizations. Then, in a top-down manner CLUSE prunes generalizations according to their coverage frequencies of positive and negative chains. CLUSE returns the remaining generalizations with their coverage frequencies as learned DDCs. These DDCs can be further generalized to variable predicates to make the DICs.

6.2.1 DDCs are generalizations of chains using CLGG

CLUSE learns a hierarchy (or a tree structure) of generalizations in a bottom-up manner from chains (Algorithm 6.1).

```

Gen ← Chains from positive examples as roots
compute similarity of pair of roots in Gen
repeat
    find most similar roots Ch1 and Ch2 in Gen
    G ← CLGG(Ch1, Ch2)
    Gen ← Gen + G + Ch1 + Ch2 {connect Ch1,Ch2 to their CLGG in Gen}36
    compute similarity of G with other roots in Gen
until one root left or no similarity between roots

```

Algorithm 6.1. CLUSE generalizes the two most similar chains at a time using CLGG until no generalization is possible and returns a structure of generalizations.

First, examples are split into chains (Section 6.1). At the beginning, positive chains are considered roots of the structure. CLUSE evaluates the similarity of each pair of roots. It chooses the two most similar roots and generalizes them using CLGG. The resulting generalization becomes the parent (and the new root) of the two generalized chains. The similarity of this root with each other root is evaluated. CLUSE repeats this process until no more generalization is possible.

When taxonomies (BK) are available to CLUSE, the similarity of pairs and the cost of generalizing them are computed. The *cost* expresses the distance between predicates in a taxonomy (Appendix III). For instance, the cost of generalizing *square* and *rectangle* from the taxonomy *form/1* is one, since they are only one level apart in the taxonomy. Then the most similar chains with the lowest cost are generalized first with CLGG (Chapter 5). The remainder of the algorithm stays unchanged.

A generalization is added to the structure as the parent of the two chains that it generalizes (Algorithm 6.2).

³⁶ The generalization G is connected as the parent of Ch1 and Ch2. If Ch1 or Ch2 are ground chains (*i.e.* from examples) then they are removed and only the G is added to *Gen*.

```

Gen ← G (where G parent of Ch1 and Ch2)
if G exactly subsumes Chain (Ch1 and/or Ch2) or
    Chain is a ground chains (i.e from a positive example) then
    remove Chain
for each unmatched literal of G (G.UnmLit) do
    if G and G.UnmLit exactly subsume Ch1 or Ch2 then
        remove G.UnmLit

```

Algorithm 6.2. Insertion of a generalization in the structure.

To avoid a tree with duplicate generalizations (when many chains result in the same generalization), chains that are exactly subsumed by their generalization are removed. Chains are *exactly subsumed* by their generalization if the only differences between them are the argument names; i.e. the exact subsumption is tested efficiently³⁷. Every unmatched literal (when combined with the generalization) that is exactly subsumed by the generalization it accompanies, is also removed.

Pruning of generalizations using coverage frequencies

Pruning the structure preserves generalizations with good coverage of instances and discards others (Algorithm 6.3).

```

doPruning Gen
compute frequencies of Gen
if Gen covers negatives then
    for each child Child of Gen do
        compute frequencies of Child
        if Child covers fewer negatives than Gen and
            Child frequencies satisfy Coverage% then
            doPruning Child
        else
            remove Child
else
    remove all children
    remove all unmatched literals of G

```

Algorithm 6.3. CLUSE passes through the structure of generalizations and prunes generalizations using coverage of frequencies. Coverage% corresponds to parameter values for positive and negative coverage of chains (PCov and NCov respectively).

CLUSE traverses a structure in depth-first manner and computes coverage frequencies for positive and negatives examples. *Coverage frequencies* correspond to the number of

³⁷ There exists a procedure to decide the θ_F -subsumption between chains (Figure 5.1).

positive (or negative) instances subsumed by the generalization divided by the total number of positive (or negative) instances. A generalization is preserved when it covers fewer negative examples than the generalization that subsumes it, and satisfies the user-defined coverage thresholds. CLUSE has two parameters that allow the user to define the level of pruning according to coverage frequencies. These parameters are the *positive coverage threshold* $PCov$ and the *negative coverage threshold* $NCov$ (Section 6.5). Preserved generalizations along with their coverage frequencies are returned as learned DDCs.

The frequency of coverage gives more flexibility than a measure like information gain (Quinlan 1990). Unlike information gain, the coverage frequencies of a generalization explicitly represent the proportion of subsumed positive and negative examples. This allows the user to fix two different coverage thresholds for choosing generalizations. For instance, the user may choose to preserve only generalizations that cover at least 50% of the positive and at most 25% of the negatives.

6.2.2 DICs are generalizations of DDCs

DDCs are useful for learning concepts in the same domain where they are learned, since their first-order predicates are specific to the domain. Assuming that concepts in different domains are described with different predicates, clichés independent of the domain are required. To learn such clichés, first-order predicates of DDCs are replaced with second-order predicates giving DICs (Algorithm 6.4).

Furthermore, it might be useful to know if a predicate is generalized using the BK or not. So, the information as to whether a predicate of a DDC is intensional or extensional is preserved within the predicate name in the DIC. An intensional predicate is generalized to a predicate variable of the form $IntP\#$, and an extensional predicate to a predicate variable of the form $ExtP\#$. When DICs are used to learn a concept in a new domain, this information can be recovered and used to instantiate second-order predicates with intensional or extensional predicates of the new domain.

```

for each cliché C of DDC do
  start a new DIC
  for each literal L of C do
    case predicate P of L of
      intensional:    VP <- IntP#
      extensional:    VP <- ExtP#
    add VP (with argument of L) to the current DIC
eliminate redundant DICs

```

Algorithm 6.4. DDCs are generalized into DICs.

Intensional predicates have children in a taxonomy. Extensional predicates belong to instances. All occurrences of a predicate in a DDC is converted with the same predicate in the DIC.

6.3 CLUSE learns clichés in $O(mn^3)$

In the worst case, CLUSE learns clichés in $O(mn^3)$, where m is the number of instances and n is the number of chains from positive instances. For the purpose of the complexity analysis, we can equate the number of chains with the number of literals, as in the worst case there will be one chain per literal (*i.e.* when all literals are relations)³⁸.

Instances are split into chains in the following way. A new chain Ch is started for each relation R in an instance. For each constant argument Arg of R , literals (*i.e.* attributes) that share Arg are added to Ch . In the worst case, the number of relations, the number of constant arguments and the number of attributes are equal to the number of literals n . So, the complexity of making the chains is $O(mn^3)$.

CLUSE learns clichés in two steps from chains: DDCs are learned first in a bottom-up manner and then from these DDCs are learned the DICs. First, CLUSE evaluates the similarity between each pair of chains. This requires $(n * (n - 1)) / 2$ comparisons. CLUSE finds the best matching chains $C1$ and $C2$, which may be observed chains from examples or, on later calls, generalizations for a set of instances. The algorithm combines (cost $n * n$) the two chains into a new generalization G , storing $C1$ and $C2$ as its children in the hierarchy (constant cost). Next CLUSE checks to see if any chains remain to be generalized. If not, CLUSE halts, returning the entire hierarchy it has generated along the

³⁸ In the FEM domain, more than half of the literals are relations.

way. If chains remain, it removes all pairs containing *C1* and *C2* (since *G* now covers them) and calculates all pairwise similarity between *G* and the remaining chains (the worst case would be $n * n$). CLUSE then calls itself recursively on the new set of pairs, combining the best matching chains, adding a new node to the hierarchy, and so forth until it has combined all chains into a single hierarchy. Since this is done $n - 1$ times (*i.e.* only one pair is generalized at a time), the complexity to learn the DDCs is $O(n^3)$.

CLUSE learns DICs by further generalizing learned DDCs. Although chains are generalized into a hierarchy, learned DDCs are returned as a flat list. In the worst case, there exists a DDC for every generalization of matching chains, *i.e.* $n - 1$ DDCs. A DIC is made by going through every literal of a DDC. Therefore, the complexity to learn the DICs is $O(n^2)$.

The overall complexity to learn clichés is due to the splitting instances into chains $O(mn^3)$ in the worst case³⁹.

6.4 Examples of learning clichés

This section illustrates an example of CLUSE learning a concept in the *blocks* domain. It shows the chains used for learning, the structure of generalizations, the DDCs and the DICs learned for the concept. Examples for the concept are generated using GENEX, and the taxonomy *form/1* as the BK (Appendix III). The concept B4 describes two scenes 1) an *ellipse above a rectangle*, which is *left of an isosceles triangle*; 2) an *ellipse above an isosceles triangle* and a *rectangle left of the triangle*. In both cases the ellipse may be *small*, the rectangle may be *large* and the triangle may be *red*. Moreover, the ellipse can also be a *circle*, the rectangle a *square*, and the isosceles triangle a *right-angled isosceles triangle*.

³⁹ The cost of learning clichés is polynomial, but the use of clichés will be worse than polynomial (Chapter 8).

Ten positive and ten negative examples of concept B4 are generated using GENEX. Table 6.1 lists the corresponding chains. There are twice as many chains as examples, since each example has two relations.

Ex.	Ch.	Positive chains
1	1	leftof(x1, x2), large(x1), sq(x1), equil(x2), red(x2)
1	2	above(x3, x4), cir(x3), small(x3), equil(x4), red(x4)
2	3	leftof(x5, x6), large(x5), rect(x5), iso_rangl(x6), red(x6)
2	4	above(x7, x8), ell(x7), small(x7), iso_rangl(x8), red(x8)
3	5	leftof(x9, x10), large(x9), rect(x9), iso(x10), red(x10)
3	6	above(x11, x12), ell(x11), small(x11), large(x12), rect(x12)
4	7	leftof(x13, x14), sq(x13), equil(x14), red(x14)
4	8	above(x15, x16), cir(x15), equil(x16), red(x16)
5	9	leftof(x17, x18), large(x17), rect(x17), iso(x18), red(x18)
5	10	above(x19, x20), ell(x19), small(x19), iso(x20), red(x20)
6	11	leftof(x21, x22), rect(x21), iso(x22), red(x22)
6	12	above(x23, x24), ell(x23), small(x23), iso(x24), red(x24)
7	13	leftof(x25, x26), sq(x25), iso_rangl(x26), red(x26)
7	14	above(x27, x28), ell(x27), sq(x28)
8	15	leftof(x29, x30), large(x29), rect(x29), iso(x30)
8	16	above(x31, x32), ell(x31), iso(x32)
9	17	leftof(x33, x34), sq(x33), equil(x34), red(x34)
9	18	above(x35, x36), cir(x35), small(x35), sq(x36)
10	19	leftof(x37, x38), sq(x37), equil(x38)
10	20	above(x39, x40), ell(x39), equil(x40)
Ex.	Ch.	Negative chains
11	21	leftof(x41, x42), equil(x41), rect(x42)
11	22	above(x43, x44), iso_rangl(x43), equil(x44)
12	23	leftof(x45, x46), iso(x45), rect(x46)
12	24	above(x47, x48), iso_rangl(x47), iso(x48)
13	25	leftof(x49, x50), cir(x49), cir(x50)
13	26	above(x51, x52), cir(x51), cir(x52)
14	27	leftof(x53, x54), iso_rangl(x53), rect(x54)
14	28	above(x55, x56), iso_rangl(x55), rect(x56)
15	29	leftof(x57, x58), iso(x57), equil(x58)
15	30	above(x59, x60), iso(x59), iso(x60)
16	31	leftof(x61, x62), cir(x61), rect(x62)
16	32	above(x63, x64), cir(x63), rect(x64)
17	33	leftof(x65, x66), sq(x65), cir(x66)
17	34	above(x67, x68), rect(x67), cir(x68)
18	35	leftof(x69, x70), sq(x69), ell(x70)
18	36	above(x71, x72), rect(x71), ell(x72)
19	37	leftof(x73, x74), equil(x73), ell(x74)
19	38	above(x75, x76), equil(x75), equil(x76)
20	39	leftof(x77, x78), ell(x77), rect(x78)
20	40	above(x79, x80), ell(x79), ell(x80)

Table 6.1. Positive and negative chains for examples generated for concept B4.
The first column is the example number, the second is the chain number and the third is the chain (body only).

Figure 6.3 shows the structure of generalizations learned with CLUSE for concept B4⁴⁰. The bottom level represents the subsumed positive chains (which are not preserved in the hierarchy). For instance, chains 16 and 20 are generalized into generalization G50. G50 and chains 6 and 14 are generalized into the generalization G57, etc. The same structure is represented in more details in the following paragraphs.

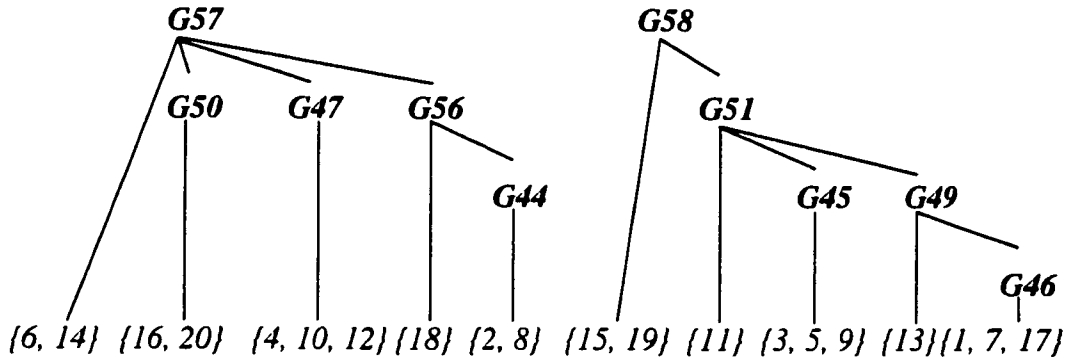


Figure 6.3. The hierarchy of generalizations learned (in bold) for concept B4.
Numbers between brackets represent subsumed positive chains.

In the first step of Algorithm 6.1, CLUSE builds the structure of generalizations in a bottom-up manner using CLGG (Figure 6.4).

⁴⁰ CLUSE parameters setup for this example are: *SimCh* = 0, *SimBind* = 0, *PCov* = 0, *NCov* = 1, and *BK* = on.

	F+	F-
G58: leftof(V1,V2),Rect(V1), iso(V2). Rect(V1) -> (sq(x37), rect(V25)). Iso(V2) -> (equil(x38), iso(V26)). large(x29) {15} Subs. chains: {1, 3, 5, 7, 9, 11, 13, 15, 17, 19}	0.5	0.25
G51: leftof(V21,V22), Rect(V21), iso(V22), red(V22). Rect(V21) -> (sq(V17), rect(V15)). Subs. chains: {1, 3, 5, 7, 9, 11, 13, 17}	0.4	0
G49: leftof(V17,V18), eq(V17), iso(V18), red(V18). Iso(V18) -> (equil(V12), iso_rangl(x26)). Subs. chains: {1, 7, 13, 17}		
G46: leftof(V11,V12), eq(V11), equil(V12), red(V12). large(x1) {1} Subs. chains: {1, 7, 17}		
G48: leftof(V9,V10), large(V9), rect(V9), iso(V10), red(V10). Iso(V10) -> (iso(V1), iso_rangl(x8)). Subs. chains: {3, 5, 9}		
G57: above(V35,V36), Ell(V35), Poly(V36). Ell(V35) -> (cir(V33), ell(V29)). Poly(V28) -> (iso(V14), rect(x12)). Poly(V24) -> (iso(V20), sq(x28)). large(x12) {6} red(V14) {4, 10, 12} Subs. chains: {2, 4, 6, 8, 10, 12, 14, 16, 18, 20}	0.5	0.15
G56: above(V33,V34), cir(V33), Poly(V34). Poly(V34) -> (equil(V8), sq(x36)). red(V8) {2, 8} Subs. chains: {2, 8, 18}	0.15	0.1
G44: above(V7,V8), cir(V7), equil(V8), red(V8). Subs. chains: {2, 8}	0.1	0
G47: above(V13,V14), ell(V13), small(V13), iso(V14), red(V14). Iso(V14) -> (iso(V5), iso_rangl(x8)). Subs. chains: {4, 10, 12}	0.15	0
G50: above(V19,V20), ell(V19), iso(V20). Iso(V20) -> (iso(x32), equil(x40)). Subs. chains: {16, 20}	0.1	0.1

Figure 6.4. CLUSE generalizes chains into a structure and prunes the structure according to generalizations' coverage frequencies. Generalizations appear in bold, followed by predicate bindings, unmatched literals, and subsumed positive chains. Shaded generalizations are pruned.

At the lowest level, generalization G46 subsumes chains 1, 7 and 17. `large(x1)` belongs to the positive chain `{1}` and is an unmatched literal for this generalization. G49 subsumes G46 (and therefore chains 1, 7, and 17) and chain 13, since it knows from the taxonomy form that an *equilateral triangle* and an *isosceles right-angled triangle* are also *isosceles triangles*. So, `equil(V12)` from G46, and `iso_rangl(x26)` from chain 13 are

generalized into `iso(V18)` in G49. CLUSE learns a structure of generalizations with two roots for this concept since it cannot generalize them.

In the second step CLUSE prunes the structure in a top-down manner according to the coverage frequencies of generalizations, starting with G58. G58 covers 50% of the positives and 25% of the negatives ($F+ = 0.5$ and $F- = 0.25$). CLUSE preserves G58, since it is a top-level generalization. The combination of the unmatched literal `large(x29)` with G58 covers 5% of the positives and none of the negatives. It covers fewer negatives than G58 itself, so the unmatched literal is preserved. CLUSE continues with G51 and finds the coverage frequencies to be 40% for the positives and 0% for the negatives. G51 covers fewer negative examples than G58, so CLUSE preserves it. CLUSE knows that no generalizations under G51 can cover fewer negatives than G51 (because lower generalizations are more specific). Therefore, CLUSE prunes all descendants of G51 (*i.e.* G49, G46, and G45). For the same reason, CLUSE would also remove unmatched literals of G51. Similarly, CLUSE passes through the second major branch G57 and evaluates coverage frequencies. In this case, none of the generalizations or unmatched literals are pruned, since they always cover fewer negatives than the generalization that subsumes them.

Generalizations left after the pruning are returned with their coverage frequencies as learned DDCs (Table 6.2)⁴¹. A generalization with each of its unmatched literals makes a DDC. There are three DDCs (7, 8, and 9) created from generalization G57. DDC 7 corresponds to G57 itself. DDCs 8 and 9 correspond to G57 with each of its unmatched literals `large(Y)` and `red(Y)` respectively.

⁴¹ Variable names are changed for simplification only.

#	Gen	DDC	F+	F-
1	44	<i>above(X, Y), cir(X), equil(Y), red(Y)</i>	0.1	0
2	47	<i>above(X, Y), ell(X), small(X), iso(Y), red(Y)</i>	0.15	0
3	50	<i>above(X, Y), ell(X), iso(Y)</i>	0.1	0.1
4	51	<i>leftof(X, Y), Rect(X), iso(Y), red(Y)</i>	0.4	0
5	56	<i>above(X, Y), cir(X), Poly(Y)</i>	0.15	0.1
6	56	<i>above(X, Y), cir(X), Poly(Y), red(Y)</i>	0.1	0
7	57	<i>above(X, Y), Ell(X), Poly(Y)</i>	0.5	0.15
8	57	<i>above(X, Y), Ell(X), large(Y), Poly(Y)</i>	0.05	0
9	57	<i>above(X, Y), Ell(X), Poly(Y), red(Y)</i>	0.15	0
10	58	<i>leftof(X, Y), Rect(X), iso(Y)</i>	0.5	0.25

Table 6.2. Learned DDCs with their coverage frequencies.

Table 6.3 lists DICs generalized from DDCs following Algorithm 6.4. For example, DIC-1 is a generalization of DDC-1 where all predicates are extensional. The extensional relation *ExtP1(X, Y)* in DIC-2 generalizes the relation *above(X, Y)* in DDC-2. Literals *IntP1(X)* and *ExtP1(X)* in DIC-2 generalize literals *ell(X)* and *small(X)* of DDC-2; literals *IntP2(Y)* and *ExtP2(Y)* generalize literals *iso(Y)* and *red(Y)*. Notice that DIC-3 is a generalization for both DDC-7 and DDC-10.

#	DIC
1	<i>ExtP1(X, Y), ExtP2(X), ExtP3(Y), ExtP4(Y)</i>
2	<i>ExtP1(X, Y), IntP1(X), ExtP2(X), IntP2(Y), ExtP3(Y)</i>
3	<i>ExtP1(X, Y), ExtP2(X), IntP1(Y)</i>
4	<i>ExtP1(X, Y), IntP1(X), IntP2(Y), ExtP2(Y)</i>
5	<i>ExtP1(X, Y), IntP1(X), IntP2(Y)</i>
6	<i>ExtP1(X, Y), ExtP2(X), IntP1(Y), ExtP3(Y)</i>
7	<i>ExtP1(X, Y), ExtP2(X), IntP1(X), IntP2(Y)</i>

Table 6.3. DICs generalized from DDCs.

CLUSE has also been used to learn clichés in the real-life domain of the FEM. DDCs and DICs for the FEM concept mesh-4 learned with CLUSE are listed in Appendix VI.

6.5 Sensitivity of CLUSE to its parameters

This section introduces CLUSE parameters and evaluates their effect on the behaviour of CLUSE.

CLUSE has five parameters that can be set by the user:

- 1) **Similarity of chains (*SimCh*)** (value between 0 and 1): minimum value for the similarity of two chains; used to prune the search space.
- 2) **Similarity of bindings (*SimBind*)** (value between 0 and 1): minimum value for the similarity of two arguments; used to prune the search space.
- 3) **Positive coverage threshold (*PCov*)** (value between 0 and 1): minimum frequency for positive examples. Used to prune DDCs.
- 4) **Negative coverage threshold (*NCov*)** (value between 0 and 1): maximum frequency for negatives. Used to prune DDCs.
- 5) **Background knowledge (*BK*)** (value *on* or *off*): taxonomies available (or not) for learning.

SimCh and *SimBind* affect the structure of generalizations during the generalization process. They allow the user to cut at the top of the structure (*i.e.* to preserve more specific generalizations). *PCov* and *NCov* affect the structure during the pruning process. They allow the user to cut at the bottom of the structure (*i.e.* to preserve more general generalizations). *BK* affects the generalizations themselves. In the presence of *BK*, unmatched literals may be generalized.

In this section I present results of an experiment to evaluate CLUSE's sensitivity to the first four parameters. This experiment consisted of varying the value of one parameter at a time and evaluating its effect on learning a concept. The first three parameters vary from 0 to 1 by step of 0.25. The fourth parameter (*NCov*) varies by step of 0.25 from 1 down to 0.25.

The concept used in the experiment was mesh-5⁴² from the FEM domain. Mesh-5 has 19 positive and 414 negative examples. These examples were split into 160 positive and 3568 negative chains. These large numbers are due to the number of occurrences of symmetric relations *opposite/2* and *neighbour/2*. For instance, the example for

⁴² Results with the same tendency are obtained with block concept B4 (with or without the *BK*).

`mesh(h14, 5)` has one chain for the relation `neighbor(h14, h13)`, and one chain for the relation `neighbour(h13, h14)`.

Two of the parameters had no effect on the generalization or the pruning. The similarity of binding arguments (*SimBind*) had no influence because all generalizations occurred with at least one binding of arguments with a value close to 1. *NCov* had no effect on the pruning, since most of the generalizations had a negative coverage frequency close to 0.

Table 6.4 shows statistical results of learning `mesh-5` with different settings of the *SimCh* parameter⁴³. The top part represents statistics on the generalization process. The second part represents the number of generalizations and literals that are dropped during the pruning process. The third part represents statistics on learned DDCs. CPU time is reported for the overall process.

		<i>SimCh</i>				
		0	0.25	0.5	0.75	1
Generalization	Generalizations	153	151	144	130	104
	Literals	6	6	7	7	7
	Unmatched literals	1	86	81	64	36
	F+ (%)	3	3	2	2	1
	F- (%)	2	2	1	1	0
Pruning	Pruned generalizations	16	26	39	60	78
	Pruned literals	10	10	10	4	0
Clichés	DDCs	137	125	105	70	26
	Literals	6	6	6	7	7
	F+ (%)	4	4	4	3	3
	F- (%)	2	2	1	1	1
Execution	CPU time (in minutes)	17	17	19	20	20

Table 6.4. Statistical results of CLUSE's sensitivity to *SimCh* parameter settings.
Rows represent statistics for comparisons and columns represent executions.

The first execution (first column) represents *SimCh* value of 0 resulting in no search space pruning. During the generalization process CLUSE made a total of 153 generalizations. On average these generalizations had 6 literals, had one unmatched literal, they covered 3% of positive chains and 2% of the negatives. 16 generalizations were dropped during the pruning process. Of the remaining generalizations, 10 unmatched literals are dropped. A generalization or literal is dropped when it does not

cover fewer negative chains than the generalization that subsumes it. CLUSE learned 137 DDCs, which covered on average 4% of the positive chains and 2% of the negatives. The generalization and pruning processes took 17 minutes of CPU time.

Increasing of the *SimCh* parameter causes the generalization process to stop earlier: it prunes the top level of the structure of generalizations. Compared to the first execution fewer DDCs were learned and they were more specific. Hence their coverage frequency for positive chains ultimately decreased by 1% for the positives (from 4 to 3) and the negatives (from 2 to 1).

The next parameter *PCov* only affects the structure during the pruning process: it has no effect during the generalization process. Table 6.5 shows results of learning mesh-5 with different values of the *PCov* parameter. The higher the *PCov* value gets, the more CLUSE prunes generalizations and unmatched literals. Only generalizations and unmatched literals that cover fewer negatives than the generalization that subsumes it are preserved. The others are dropped. Moreover, they also need to satisfy the *PCov* to be preserved. This pruning decreases the CPU time from 17 to 10 minutes.

		<i>PCov</i>				
		0	0.25	0.5	0.75	1
<i>Pruning</i>	Pruned generalizations	16	145	147	147	147
	Pruned literals	10	18	16	16	16
<i>Clichés</i>	DDCs	137	8	6	6	6
	Literals	6	4	3	3	3
	F+ (%)	4	23	17	17	17
	F- (%)	2	16	17	17	17
<i>Execution</i>	CPU time (in minutes)	17	11	10	10	10

Table 6.5. Statistical results of CLUSE's sensitivity to *PCov* parameter settings. Rows represent statistics for comparisons and columns represent executions.

No generalization satisfied the values of *PCov* higher than 0.25. In principle, no DDCs should have been learned for the last three executions, except that CLUSE preserves generalizations at the top level (*i.e.* generalizations not subsumed by any other generalizations). The idea is that although learned DDCs do not satisfy *PCov*, the user can adjust the parameter setting according to the best generalizations that CLUSE *could*

⁴³ There is no BK used for this experiment.

learn. Therefore, in the worst case with mesh-5, CLUSE returned 6 (out of 153 generalizations) DDCs, which covered 17% of the positive chains and 17% of the negatives.

To summarize, CLUSE's parameters⁴⁴ allow the user to modify the hierarchy of generalizations, and hence what clichés are learned. CLUSE learns combinations of literals for as few as two chains (or even one chain with unmatched literals), and generalizes them into a hierarchy. The pruning process discards generalizations that cover as many negatives as the generalizations that subsume them. CLUSE's parameters allow the user to modify the hierarchy of generalizations with two of the five parameters. The mesh-5 experiment showed that the similarity of chains parameter (*SimCh*) influences the structure of the hierarchy during the generalization process. An increase of that value cuts the top of the hierarchy (*i.e.* the most general generalizations), by stopping the generalization process earlier. The positive coverage of chains (*PCov*) also modifies the hierarchy of generalizations, but during the pruning process. An increase of that value cuts the lower parts of the structure (*i.e.* more specific generalizations). The presence of BK (parameter BK is *on*) provides DDCs with more information. This parameter has influence on the generalizations themselves, rather than on the hierarchy. In the presence of the BK, generalizations are expressed with more literals (provided from the BK).

6.6 DDCs represent syntactic restrictions underlying examples of a concept

This section describes an experiment in which GENEX is used to evaluate the similarities between its user-supplied rule templates and clichés (or DDCs) learned with CLUSE (Figure 6.5).

DDCs and rule templates both represent syntactic restrictions on examples of a concept. Rule templates express syntactic restrictions known (by the user) prior to the generation of examples. DDCs represent syntactic restrictions that CLUSE learns from examples. So

how do the clichés learned from examples and the rule templates used to generate these examples compare?

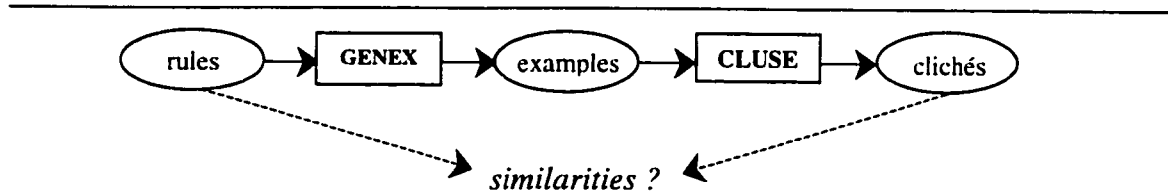


Figure 6.5. GENEX is used to evaluate clichés learned with CLUSE.

The experiment was run on concept B4 from the *blocks* domain in the presence of BK (the form taxonomy). The steps in the experiment were as follows:

- 1) generate examples using GENEX;
- 2) learn clichés with CLUSE from these generated examples.
- 3) compare the learned clichés and the rule templates given to GENEX in step 1.

To compare syntactic restrictions on chains (since clichés are chains), rule templates of concept B4 were split into “*chain templates*” (*i.e.* rule templates are split on each relation). I refer to these chain templates as B4-chains. Table 6.6 shows the rule templates (split into chains) given to GENEX to generate examples of concept B4-chains⁴⁵. 12 positive rules represent a total of 64 positive examples and 15 negative rules represent 74 negative examples.

CLUSE was able to reproduce all the syntactic underlying restrictions expressed in examples of the B4-chains from 104 generated examples (*i.e.* less than twice the number of possible examples). The complete hierarchy of generalizations is listed in Appendix VII. With fewer examples CLUSE learned incomplete restrictions. For instance, with 64 examples, three rule templates (9, 10, and 11) were exactly learned. Other templates were incomplete (*e.g.* *ell* does not occur as a generalization of *cir*).

⁴⁴ The default setting for these parameters corresponds to not having parameters, *i.e.* SimCh = SimBind = PosCov = 0, NCov = 1. The user does not require any knowledge of the system to set them.

⁴⁵ Complete rule templates given to GenEx can be found at:

		Arguments					Nb. ex
		Rule	(X)	(X)	(X, Y)	(Y)	
+		1	ell / cir		above	rect / sq	4
		2	ell / cir	small	above	rect / sq	4
		3	ell / cir		above	rect / sq	large
		4	ell / cir	small	above	rect / sq	large
		5	ell / cir		above	iso / iso_rangl / equil	6
		6	ell / cir	small	above	iso / iso_rangl / equil	6
		7	ell / cir		above	iso / iso_rangl / equil	red
		8	ell / cir	small	above	iso / iso_rangl / equil	red
		9	rect / sq		leftof	iso / iso_rangl / equil	6
		10	rect / sq	large	leftof	iso / iso_rangl / equil	6
		11	rect / sq		leftof	iso / iso_rangl / equil	red
		12	rect / sq	large	leftof	iso / iso_rangl / equil	red
-	Total						64
		51	ell / cir		above	ell / cir	4
		52	rect / sq		above	rect / sq	4
		53	rect / sq		above	iso / iso_rangl / equil	6
		54	rect / sq	small	above	ell / cir	4
		55	iso / iso_rangl / equil	small	above	rect / sq	6
		56	iso / iso_rangl / equil		above	ell / cir	large
		57	iso / iso_rangl / equil	large	above	rect / sq	6
		59	ell / cir		leftof	rect / sq	red
		60	ell / cir	small	leftof	rect / sq	large
		61	rect / sq	small	leftof	rect / sq	red
		62	rect / sq	large	leftof	ell / cir	red
		63	iso / iso_rangl / equil		leftof	rect / sq	6
		64	iso / iso_rangl / equil		leftof	rect / sq	6
		65	iso / iso_rangl / equil		leftof	ell / cir	6
	Total						74

Table 6.6. Rule templates used to generate examples of B4-chains.
The last column represents the number of examples that can be generated from the rule.

6.7 Conclusion

This chapter showed the underlying algorithm of CLUSE and its evaluation. CLUSE uses CLGG and the notion of chains (Section 6.1) to learn clichés in a bottom-up manner into a hierarchy of generalizations (Section 6.2). CLUSE prunes this hierarchy according to the generalizations' coverage frequencies of chains. Preserved generalizations and their coverage are returned as learned DDCs. DDCs are further generalized into DICs with variable predicates (Section 6.2.2). DDCs are considered domain-dependent, since they

are expressed with predicates specific to the domain, whereas DICs are domain-independent.

Subsequent sections in this chapter showed different evaluations of CLUSE. Section 6.3 showed that CLUSE learns clichés with a complexity of $O(mn^3)$ in the worst case. The cost of learning clichés is polynomial, but the use of clichés will be worse than polynomial (Chapter 8). Section 6.4 showed that CLUSE learns clichés for *blocks* as well as for FEM concepts. Section 6.5 described an experiment where CLUSE was sensitive to two of its parameters *SimCh*, during the generalization process and *PCov* during the pruning process. *SimCh* is used to cut the most general generalizations, and *PCov* to cut the most specific generalizations. Finally, Section 6.6 showed that CLUSE is able to reproduce all the underlying restrictions represented in examples of a concept when enough of examples are provided.

7 xFOIL-CLICHÉS learns concepts with clichés

This chapter gives a brief overview of FOIL. FOIL is a well-known hill-climbing learner vulnerable to the local plateau problem and can be used to show that clichés provide appropriate lookahead to such learners. This chapter introduces xFOIL-CLICHÉS, which is my implementation of FOIL extended to learn with clichés. I have applied xFOIL-CLICHÉS to learn concepts in the *blocks* domain. To illustrate xFOIL-CLICHÉS learning in the absence or in the presence of background knowledge (BK). The knowledge used is the taxonomy *form/1* (Appendix III).

Three concepts in the *blocks* domain are used in this chapter. Examples for each one were generated with GENEX. Concepts B1 and B2 are very similar and represent a small set of instances for the sake of simplicity (Table 7.1).

	Arguments						
	(X)	(X, Y)	(Y)	(Y)	(Y, Z)	(Z)	(Z)
B1 +	cir	above	rect	± black	above	rangl / iso_rangl	± black
B2 +	cir	above	rect / sq	± white	above	rangl / iso_rangl	± white
B1-	cir	above	rect / sq	red	above	ell / cir	red
&	cir	above	rect / sq	red	above	rect / sq	red
B2 -	cir	above	rangl / iso_rangl	red	above	ell / cir	red

Table 7.1. Rule descriptions for examples of concepts B1 and B2.

Rows represents rule descriptions to generate positive and negative examples. Columns represent predicates associated with the corresponding arguments. '/' represents and or. '±' denotes an optional literal.

Each concept describes three objects, *X*, *Y* and *Z*. In both concepts *X* is above *Y* and *Y* is above *Z*. In *B1*, *X* is a *circle*, *Y* is a *rectangle* that may be *black* and *Z* is a *right-angled* or *isosceles right-angled triangle* that may be *black*. In *B2*, *X* is a *circle*, *Y* is a *rectangle* or *square* that may be *white* and *Z* is a *right-angled triangle* or *isosceles right-angled triangle* that may be *white*. The negative examples for both concepts are the same: *X* is a *circle*, *Y* is a *red rectangle* or *square* and *Z* is a *red ellipse*, *circle*, *rectangle*, or *square*. Alternatively, *Y* may be a *red right-angled triangle* or *isosceles right-angled triangle* when *Z* is a *red ellipse* or *circle*.

	Arguments						
	(X)	(X, Y)	(Y)	(Y)	(Y, Z)	(Z)	(Z)
Rules	+	ell / cir	above	rect	± blue	leftof	para
		ell / cir	above	para	± red	rightof	rect
	-	ell / cir	above	rect	red	leftof / above	ell / cir
		ell / cir	above	rect	red	leftof / above	iso / iso_rang / equi
		ell / cir	above	para	blue	rightof / above	ell / cir
		ell / cir	above	para	blue	rightof / above	iso / iso_rang / equi
		ell / cir	above	para	blue	above	rect
		ell / cir	above	ell / cir	above	para	Size
		ell / cir	above	ell / cir	above	ell / cir	Size
		ell / cir	above	ell / cir	above	ell / cir	Size
		ell / cir	above	ell / cir	above	ell / cir	Size
		ell / cir	above	ell / cir	above	ell / cir	Size

Table 7.2. Rule descriptions of examples for concept B3.

Rows represents rule descriptions to generate positive and negative examples. Columns represent predicates associated with the corresponding arguments. '/' represents and or. '±' denotes an optional literal.

Concept B3 is more complex than the other two concepts and is used to provide clichés to learn the previous two concepts (Table 7.2). Concept B3 describes three objects as 1) an *ellipse* above a *rectangle*, which is *left* of a *parallelogram*; or 2) an *ellipse* above a *rectangle*, which is *right* of a *rectangle*. In the first case the *rectangle* may be *blue*, and the *parallelogram* may be *small*, *medium*, or *large*. In the second case the *parallelogram* may be *red* and the *rectangle* may be *small*, *medium*, or *large*. In both cases the *ellipse*

can also be a *circle*, the *rectangle* a *square*, and the *parallelogram* a *rhombus*, *rectangle* or a *square*. Negative examples are similarly described. Clichés are learned using CLUSE and then passed to xFOIL-CLICHÉS to learn concepts B1 and B2. For simplicity, both DDCs and DICs will be generated from concept B3. DICs will be provided as if they were learned from another domain.

7.1 FOIL

The FOIL learning system (Quinlan 1990; Quinlan 1991; Quinlan *et al.* 1993) has been widely used in the ILP community and by Machine Learning researchers at large. For complete examples of applying FOIL, see (Lavrac *et al.* 1994; Bergadano *et al.* 1995).

FOIL learns function-free (and constant-free) Horn clause theories that serve as intensional definitions of a concept (or a predicate). The definition of a concept consists of a set of clauses, where each clause represents an alternative method of proving that an example is an instance of the concept. The clauses consist of a conjunction of literals, where each literal is composed of a predicate and an ordering of variables for the predicate. Variables of the literal are classified as *new* and *old* as follows: a variable of a literal is called “*new*” if it does not appear in the head of the current clause or in any literal to the left of the current literal; otherwise, the variable is called “*old*”.

An example in FOIL is represented as a *tuple*, which contains values for the variables of the predicates to be learned. For example, when learning the definition of a mesh, the literal `mesh(X, Y)` denotes that edge `x` has `y` finite elements on it. Training examples for this problem consist of a set of 2-tuples whose elements correspond to the edge name and its number of FEs. Each tuple is identified as a positive or negative instance of `mesh(X, Y)`. `[g1, 9]` and `[g2, 9]` would be positive instances of a mesh with 9 FEs, whereas `[f1, 5]` and `[h3, 7]` would be negative instances.

FOIL starts out with a set of extensionally defined predicates, one of which is identified as the concept to learn, and others as *background literals*⁴⁶. FOIL learns a set of clauses comprised of literals derived from these extensionally defined predicates. For example, the predicates `neighbour(X, Y)` (edge `x` is the neighbour of edge `y`) can be useful in learning the concept `mesh(X, Y)`. In FOIL, these predicates must be defined extensionally.

Algorithm 7.1 gives FOIL's main algorithm.

```

Input
Pred: Name of the predicate to learn
Vars: An ordered tuple of variable names for the predicate
Pos:  A set of tuples for the positive examples of the predicate
Neg:  A set of tuples for the negative examples of the predicate
Preds:      A set of extensionally defined predicates (BK)

set Clauses to empty
until Pos is empty
    set NewClause to empty
    set Old to Vars
    until Neg is empty
        for each Predicate in Preds
            for each V in variabilization(Predicate, Old)
                create a Literal from Predicate and V
                compute_gain(Literal, Pos, Neg)
            conjoin the literal having the maximum gain to NewClause
            add any new variables in the literal to Old
            set Pos to the extensions of Pos satisfied by Literal
            set Neg to the extensions of Neg satisfied by Literal
        remove from Pos all tuples that satisfy the NewClause
        reset Neg to the original negative tuples
        add NewClause to Clauses
return Clauses

```

*Algorithm 7.1. An algorithmic overview of FOIL.
Preds represents the background literals.*

FOIL starts the search with a new empty clause. It adds a literal to the end of the current clause until no negative example is covered by the clause. FOIL starts new clauses until all positive examples are covered by the clauses already built. FOIL computes the

⁴⁶ The *background literals* are also called the BK for FOIL and they are strictly extensional (*i.e.* ground facts with a single literal).

information gain of the **legal variabilization**⁴⁷ of each extensionally defined predicate in order to determine which literal to add to the end of the clause. A **variabilization** is a particular ordering of new and old variables.

The information gain of the addition of a new literal to the current clause is shown in Figure 7.1.

$$\text{gain}(\text{Literal}) = T_i^{++} \times (I(T_i) - I(T_{i+1}))$$

$$I(T_i) = -\log_2(T_i^+ / (T_i^+ + T_i^-))$$

$$I(T_{i+1}) = -\log_2(T_{i+1}^+ / (T_{i+1}^+ + T_{i+1}^-))$$

Figure 7.1. FOIL's information gain of a literal.

T_i^{++} is the number of current positive tuples that, after adding the next literal, lead to another positive tuple. T_i^+ and T_i^- are the current number of positive and negative tuples. T_{i+1}^+ and T_{i+1}^- are the number of positive and negative tuples that would remain after adding the literal (Quinlan 1990). If a positive literal with a new variable is added to the body of the clause, the size (arity) of the tuples in the local set of tuples increases. A tuple from the current set of tuples may also give rise to more than one tuple in the next set of tuples.

Suppose the current clause to complete for meshes with 9 FEs on them is:

```
mesh(X, Y) :- fixed(X)
```

Table 7.3 illustrates the current set of tuples T_i (with variables $\{X, Y\}$) for this rule, and the extended set of tuples T_{i+1} (with variables $\{X, Y, Z\}$) when the literal $\text{neighbour}(X, Z)$ is added.

⁴⁷ A legal variabilization must include at least one old variable, and not cause infinite recursion (Quinlan 1990).

Examples	T_i	T_{i+1}
+	[g1, 9]	[g1, 9, g1]
		[g1, 9, g22]
		[g1, 9, g20]
		[g1, 9, g41]
	[g2, 9]	[g2, 9, g19]
		[g2, 9, g40]
		[g2, 9, g20]
		[g2, 9, g41]
	[g3, 9]	[g3, 9, g18]
		[g3, 9, g39]
		[g3, 9, g19]
		[g3, 9, g40]
-	[f1, 5]	[f1, 5, f2]
		[f1, 5, f24]
	[f3, 5]	[f3, 5, f12]
		[f3, 5, f14]
	[f5, 5]	[f5, 5, f4]
		[f5, 5, f5]
		[f5, 5, f21]
		[f5, 5, f22]

Table 7.3. The effect of a new variable on the current set of tuples.

T_i represents the current set of tuples $[X, Y]$, where X is an edge and Y is the number of FEs on it. T_{i+1} represents the extended set of tuples $[X, Y, Z]$ where X and Y are the same values as in the current set, and Z is the new edge introduced in the clause with the literal neighbour(X, Z).

Each of the positive tuples in T_i^+ have four successors in T_{i+1}^+ , for a total of twelve 3-tuples in T_{i+1}^+ . The first tuple in T_i^+ [g1, 9] has the successors [g1, 9, g1], [g1, 9, g22], [g1, 9, g20], and [g1, 9, g41] because g1 with 9 FEs on it has g1, g22, g20, and g41 as *neighbours*. The first two negative tuples in T_i^- have two successors each, and the third one has four successors in T_{i+1}^- . The tuples in T_{i+1} inherit the label of their parent in T_i . The information gain of adding such a literal is computed in Figure 7.2.

$$I(T_i) = -\log_2(3 / (3 + 3)) = 1.0$$

$$I(T_{i+1}) = -\log_2(12 / (12 + 8)) = 0.74$$

$$\text{gain}(\text{neighbour}) = T_i^{++} \times (I(T_i) - I(T_{i+1})) = 3 \times 0.74 = 0.79$$

Figure 7.2. The information gain in terms of extended tuples when neighbour(X, Z) is added in the example from Table 7.3.

To prevent unnecessary search, FOIL restricts the literals in the search space to satisfy the following two conditions:

- 1) The literal must contain at least one old variable, to be linked to previous literals in the clause.
- 2) If a relation to be added to the body of the clause is the same as the relation in the head of the clause, arguments are restricted to prevent some problematic recursion. The relation cannot have the same arguments as in the head of the clause. Arguments occurring both in the head and in the body must satisfy a partial ordering. This prevents the construction of useless definitions that cause infinite recursion.

Variable names are ordered *e.g.* x, y, z, and w. When FOIL requires a new variable, it takes it from that list (following the order). The argument modes for a relation are input for the first argument and output for the second one. The argument mode for a literal with a single argument is input (*i.e.* the argument must be bound). For FOIL, this means that a relation can only be added to a rule if the first argument already exists in the rule and that an attribute can only be added if its argument already exists in the rule. Moreover, arguments of a relation must appear in the same ordering as in the partial ordering list to be valid.

Following these restrictions, the example

```
block(X):- cir(X), above(X, Y), rect(Y)
```

is valid. It respects the partial ordering and the modes given to FOIL. On the other hand, the example

```
block(X):- cir(X), above(Y, X), rect(Y)
```

is invalid, because the arguments of `above/2` violate the partial ordering given for them.

Partial orderings are used to avoid constructing recursive clauses which could invoke themselves with the same arguments they were called with (thus causing infinite recursion). If a literal $p(Y_1, Y_2, \dots, Y_n)$ appears in the body of a clause that has a head $p(X_1, X_2, \dots, X_n)$, then a terminating recursion is guaranteed if the partial ordering $X_i < Y_i$ holds for some i . For more detailed treatment of this subject see (Quinlan 1990).

A newer development in the use of orderings to avoid infinite recursion is used in (Cameron-Jones *et al.* 1993). Cameron-Jones has an ordered list of variables that is preprocessed once at the beginning of a learning session for each literal. For example, $[x, y, z, w]$ ⁴⁸ establishes a partial ordering $x < y$, $y < z$, and $z < w$. Thereafter to prevent infinite recursion, it is simply a matter of keeping track of the partial orderings that have been established so far among the variables.

7.2 xFOIL-CLICHÉS

To show how clichés can be used with FOIL and to evaluate their significance on learning concepts, I implemented a version of FOIL called xFOIL in Prolog and extended it to learn concepts with clichés. xFOIL follows the same main FOIL algorithm presented in Algorithm 7.1. In this chapter I will refer to my implementation as xFOIL when not using clichés and xFOIL-CLICHÉS when using clichés to learn. When necessary to distinguish using DDCs and DICs, I will use the names xFOIL-DDC and xFOIL-DIC.

One of the differences between xFOIL and FOIL is the way the information gain is computed. xFOIL computes the information gain directly from the examples in the training set, in contrast to the count of *extended tuples* in FOIL. The same modification was used in mFOIL (Lavrač *et al.* 1994), and in FLIPPER (Cohen *et al.* 1997). In non-determinate domains (*e.g.* the FEM domain), these counts can be quite large when new variables are introduced in the clause. Consider the twelve extended tuples of T_{i+1}^+ in Table 7.3. When the new variable z is introduced in the clause with `neighbour(X, Z)`, the twelve extended tuples represent 3 positive examples (*i.e.* $(g1, 9)$, $(g2, 9)$, and $(g3, 9)$) and also represent 3 negative examples (*i.e.* $(f1, 5)$, $(f3, 5)$, and $(f5, 5)$). In xFOIL the information gain is computed only in terms of these initial tuples as shown in Figure 7.3.

⁴⁸ The same partial ordering and argument modes will be used to learn FEM and *blocks* concepts.

$$\begin{aligned}
I(T_i) &= -\log_2(3 / (3 + 3)) = 1.0 \\
I(T_{i+1}) &= -\log_2(3 / (3 + 3)) = 1.0 \\
\text{gain}(\text{neighbour}) &= T_i^{++} \times (I(T_i) - I(T_{i+1})) = 3 \times 0 = 0
\end{aligned}$$

Figure 7.3. The information gain computed in terms of the examples for the example of Table 7.3.

So the gain for $\text{neighbour}(X, Z)$ is null instead of 0.79 as it was with extended tuples. This avoids adding a literal that covers as many examples as the clause so far, hence reducing the need of a post-processing reduction of clauses.

In mFOIL, this decision is justified by the following argument. Suppose a single noisy training example, erroneously classified as positive, is covered by a clause. In FOIL, this example may be extended to enough positive tuples to yield a score high enough for the clause to be accepted. Thus a clause might be built that covers a single erroneous example.

Unlike FOIL, xFOIL makes no distinction between *determinate*⁴⁹ and non-determinate literals, since concepts with non-determinate literals are relevant to CLUSE. For example, the relation *neighbour/2* in the FEM domain is never determinate, since each edge usually has two neighbours.

Although no recursive rules are learned with xFOIL, the idea of partial ordering is used to restrict the search space of concepts with non-determinate literals (e.g. *neighbour/2* and *opposite/2* in FEM concepts). This reduces the search space and prevents symmetric literals being learned in the same clause, for example, $\text{neighbour}(X, Z)$, and $\text{neighbour}(Z, X)$.

The stopping criterion⁵⁰ from FOIL is not used in xFOIL. This allows xFOIL to learn more clauses. Lavrac *et al.* (1994) point out that one of the problems in FOIL that causes

⁴⁹ *Determinate literals* are literals that introduce new variables for which each positive tuple has exactly one extension and each negative tuple has at most one extension in the new training set.

⁵⁰ The *stopping criterion* decides whether there are sufficient data to support adding a literal to a clause (or creating a new clause). The stopping criterion compares the number of bits explicitly needed to

poor performance in learning FEM concepts is that the large number of background relations increases the number of bits needed to encode a clause. FOIL's stopping criterion prevents important clauses from being built.

7.2.1 xFOIL parameters and their effect on learning concepts

xFOIL-CLICHÉS has four parameters that can be set by the user.

- 6) **CLUSE** (on / off): allows xFOIL to use clichés (or not).
- 7) **BK** (on / off): allows xFOIL to use background taxonomies (or not) for learning.
- 8) **FreqPos** (value between 0 and 1): specifies the minimum DDC frequency for xFOIL-DDC; DDCs with a frequency less than *FreqPos* are not used in learning.
- 9) **FreqNeg** (value between 0 and 1): specifies the maximum DDC frequency for negative examples in xFOIL-DDC.

7.2.2 Algorithm for learning concepts with clichés

```

for each Cliché in Clichés                                     (1)
  for each instantiation Inst of Cliché                       (2)
    for each V in variabilization (Inst, Old)
      create a VariabilizedInst from Inst and V               (3)
      compute_gain(VariabilizedInst, Pos, Neg)                 (4)
    preserve the VariabilizedInst with the maximum gain
    compute_gain(Cliché, Pos, Neg)                             (5)
    if Cliché's gain is higher than any of its instantiations' gains then
      add Cliché to the clause                                 (6)
    else
      add instantiation to the clause                           (7)

```

Algorithm 7.2. Algorithm for learning concepts with clichés.

Clichés are used when xFOIL cannot find a literal with a positive gain and the clause still covers negative examples.

Clichés are learned when xFOIL cannot find a single literal with a positive gain and the clause still covers negative examples. Algorithm 7.2 shows part of the algorithm for

encode the data to the number of bits required to encode the new literal. It is a kind of minimum description length.

learning with clichés⁵¹. Each numbered line of the algorithm is described in the following sections.

(1) Clichés used with xFOIL-CLICHÉS

Clichés are learned with CLUSE and are given to xFOIL-CLICHÉS as input. Only DDCs with frequency values that satisfy the frequency parameters of xFOIL-CLICHÉS (*FreqPos*, *FreqNeg*) are used. This allows the user to eliminate specific DDCs, either because they cover too few positive examples or cover too many negative examples.

Table 7.4 shows DDCs-B3 (DDCs learned with CLUSE for concept B3). When xFOIL-DDC's parameters *FreqPos* (denoting the minimum frequency of positives) and *FreqNeg* (denoting the maximum frequency of negatives) are set to 0, all DDCs are preserved. If *FreqPos* were set to 0.05, the first six DDCs would be removed prior to the instantiation process.

Id	DDC	FP	FN
1	leftof(X, Y), blue(X), rect(X), para(Y), small(Y)	0.02	0
2	leftof(X, Y), blue(X), rect(X), large(Y), para(Y)	0.025	0
8	rightof(X, Y), para(X), large(Y), rect(Y)	0.035	0.05
9	rightof(X, Y), para(X), rect(Y), small(Y)	0.04	0.03
10	leftof(X, Y), rect(X), large(Y), para(Y)	0.04	0.1
3	above(X, Y), cir(X), para(Y), red(Y)	0.045	0.22
11	leftof(X, Y), rect(X), para(Y), small(Y)	0.05	0.055
5	leftof(X, Y), blue(X), rect(X), para(Y)	0.055	0
6	above(X, Y), cir(X), blue(Y), rect(Y)	0.055	0
4	above(X, Y), ell(X), para(Y), red(Y)	0.055	0.41
7	above(X, Y), ell(X), blue(Y), rect(Y)	0.075	0.0
12	above(X, Y), cir(X), para(Y)	0.105	0.32
13	above(X, Y), cir(X), rect(Y)	0.14	0.175
14	rightof(X, Y), para(X), rect(Y)	0.23	0.85
15	leftof(X, Y), rect(X), para(Y)	0.27	0.135
16	above(X, Y), ell(X), rect(Y)	0.27	0.34
17	above(X, Y), ell(X), para(Y)	0.5	0.59

Table 7.4. DDCs are expressed with first-order predicates and are provided to xFOIL-DDC with their coverage frequencies of positive and negative examples (FP and FN respectively) when learned with CLUSE.

Table 7.5 shows DICs-B3 (DICs learned for concept B3). Unlike DDCs, DICs are provided to xFOIL-DIC without coverage frequency, and are all preserved for learning.

⁵¹ Notice that for efficiency, the variabilization of clichés is done only once at the beginning.

DICs	
1	ExtP1 (X, Y) , ExtP2 (X) , IntP3 (X) , ExtP4 (Y) , IntP5 (Y)
2	ExtP1 (X, Y) , ExtP2 (X) , ExtP3 (Y) , IntP4 (Y)
3	ExtP1 (X, Y) , IntP2 (X) , ExtP3 (Y) , IntP4 (Y)
4	ExtP1 (X, Y) , ExtP2 (X) , IntP3 (X) , IntP4 (Y)
5	ExtP1 (X, Y) , ExtP2 (X) , IntP3 (Y)
6	ExtP1 (X, Y) , IntP2 (X) , IntP3 (Y)

Table 7.5. DICs learned with CLUSE for concept B3 (DICs-B3).
DICs are expressed with second-order predicates.

(2) Instantiations of clichés

xFOIL-CLICHÉS learns a concept using either DDCs or DICs. DDCs are used when the concept to learn is in the same domain as the one in which the DDCs were learned. DDCs are expressed with predicates that belong to that domain. When no taxonomy is provided to xFOIL-DDC (or in the absence of BK), each DDC becomes its own instantiation. When BK is available, each predicate that is a generalization (in a taxonomy) of other predicates is instantiated to itself and to all its children in the taxonomy.

DICs are used to learn concepts in a domain different from the one in which they are learned. They are described with variable predicates whose names tell xFOIL-DIC where to take their instantiations: either from examples (when the predicate has the format ExtP#) or from the BK (when the predicate has the format IntP#). ExtP# predicates are instantiated to every predicate (with the same arity)⁵² that is used in examples (positive or negative). IntP# predicates are instantiated to every predicate in the taxonomy.

A cliché is simply a conjunction of literals (with first and second-order predicates). Associated with every cliché is a list of instantiations. Instantiations of a cliché are made with all combinations (without repetitions) of its literals' instantiations (one instantiation for each literal at a time) (Figure 7.4). Literals in clichés are instantiated with literals in the domain of the target concept. Instantiations of literals depend on 1) the kind of clichés (DDC or DIC), 2) the literal itself, 3) the availability of taxonomies, and 4) the concept to learn.

⁵² For simplicity, I assume all instantiations are made with literals of the same arity. Moreover, predicates are instantiated with the same arguments as the literals in clichés.

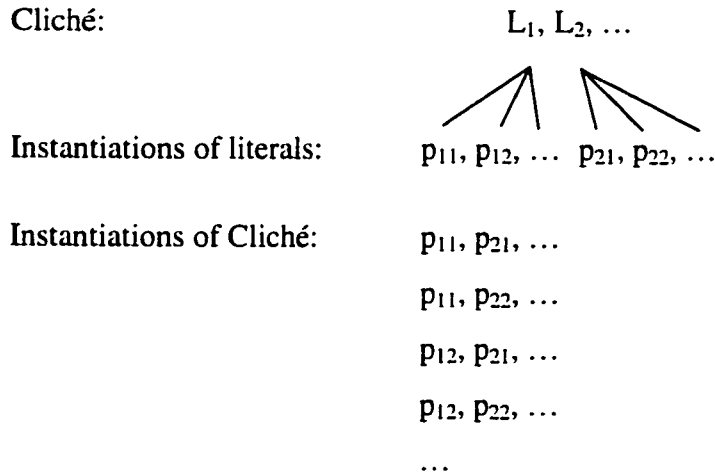


Figure 7.4. A cliché is a conjunction of literals.
Instantiations of a cliché correspond to all combinations of its instantiated (first-order) literals.

Instantiations of literals in DDCs

Table 7.6 shows the instantiations of literals for DDCs. A literal in a DDC is instantiated to itself unless if it is an intensional literal and BK is available. In this case, the literal is instantiated to itself and to its children in the taxonomy. Although DDCs are learned from a concept in the same domain, they may be expressed with literals that are not used to describe examples of the current concept. Since such DDCs cannot have a positive gain during the learning, they are discarded⁵³.

Predicate format	No BK	BK
<intensional>	<i>itself</i>	intensional
<extensional>	<i>itself</i>	<i>itself</i>

Table 7.6. Instantiations of DDCs' literals
depend on the predicate name and on the availability of the BK.

For example, from DDCs-B3 (Table 7.4), only two DDCs—DDC-13 and DDC-16—are preserved for learning the concept B1. In the absence of the BK, a DDC is its own instantiation. Other DDCs are discarded, since they are expressed with literals that do not belong to the target concept.

⁵³ This also considerably reduces the search space, without affecting learning.

Table 7.7 shows the preserved DDCs-B3 and their instantiations for learning the concept B1 when the BK is available. Literals *above/2*, *cir/1*⁵⁴ and *red/1* in DDC-3 are extensional. *Para/1* is an intensional literal and is instantiated to *rect/1* and to *sq/1* (its only children that are extensional). Other DDCs-B3 not appearing in Table 7.7 are discarded because they have a literal that is neither extensional nor intensional⁵⁵ for the concept B1 (e.g. *leftof/2*, *rightof/2*, *blue/1*).

	DDC	Instantiations
3	<i>above(X,Y), cir(X), Para(Y), red(Y)</i>	1. <i>above(X,Y), cir(X), rect(Y), red(Y)</i> 2. <i>above(X,Y), cir(X), red(Y), sq(Y)</i>
4	<i>above(X,Y), Ell(X), Para(Y), red(Y)</i>	1. <i>above(X,Y), cir(X), rect(Y), red(Y)</i> 2. <i>above(X,Y), cir(X), red(Y), sq(Y)</i> 3. <i>above(X,Y), ell(X), rect(Y), red(Y)</i> 5. <i>above(X,Y), ell(X), red(Y), sq(Y)</i>
12	<i>above(X,Y), cir(X), Para(Y)</i>	1. <i>above(X,Y), cir(X), rect(Y)</i> 2. <i>above(X,Y), cir(X), sq(Y)</i>
13	<i>above(X,Y), cir(X), Rect(Y)</i>	1. <i>above(X,Y), cir(X), rect(Y)</i> 2. <i>above(X,Y), cir(X), sq(Y)</i>
16	<i>above(X,Y), Ell(X), Rect(Y)</i>	1. <i>above(X,Y), cir(X), rect(Y)</i> 2. <i>above(X,Y), cir(X), sq(Y)</i> 3. <i>above(X,Y), ell(X), rect(Y)</i> 4. <i>above(X,Y), ell(X), sq(Y)</i>
17	<i>above(X,Y), Ell(X), Para(Y)</i>	1. <i>above(X,Y), cir(X), rect(Y)</i> 2. <i>above(X,Y), cir(X), sq(Y)</i> 3. <i>above(X,Y), ell(X), rect(Y)</i> 4. <i>above(X,Y), ell(X), sq(Y)</i>

Table 7.7. DDCs-B3 and their instantiations
for learning the concept B1 when BK is available. Predicate with a capital letter represent intensional predicates.

Instantiations literals in DICs

All literals in DICs are variables (or second-order predicates) (Table 7.8). Each literal is instantiated to extensional literals in the absence of BK. In the presence of the BK, literals with variable predicates of form *ExtP#* are also instantiated to extensional literals, but literals with variable predicates of the form *IntP#* are instantiated to intensional literals (i.e. literals in the taxonomy and their children).

⁵⁴ *cir/1* also belongs to a taxonomy, but at the lowest level (i.e. the only instantiation is itself), so it is not intensional.

⁵⁵ Literals that were extensional (or intensional) when DDCs are learned from concept B3 may not be extensional (or intensional) for the concept B1.

Predicate format	No BK	BK
ExtP#	extensional	extensional
IntP#	extensional	intensional

Table 7.8. Instantiations of DICs' literals depend on the predicate name and on the availability of the BK.

In the absence of BK, all variable literals in a DIC are instantiated to extensional literals. This is equivalent to replacing predicates of the form IntP# in DICs with variable predicates of the form ExtP#. Table 7.9 shows such a replacement for DICs from Table 7.5. For example, DIC-1 in Table 7.5 becomes DIC-1 in Table 7.9 with IntP3(X) and IntP4(Y) replaced by ExtP3(X) and ExtP4(Y). DIC-3 and DIC-6 from Table 7.5 are discarded because they become duplicates of DIC-2 and DIC-5 in Table 7.9.

	DICs	Number of instantiations
1	ExtP1(X, Y), ExtP2(X), ExtP3(X), ExtP4(Y), ExtP5(Y)	4096
2	ExtP1(X, Y), ExtP2(X), ExtP3(Y), ExtP4(Y)	512
4	ExtP1(X, Y), ExtP2(X), ExtP3(X), ExtP4(Y)	512
5	ExtP1(X, Y), ExtP2(X), ExtP3(Y)	64
	Total	5184

Table 7.9. DICs' from DICs-B3 and their instantiations. Without the BK, variable literals are instantiated to extensional literals.

The size of the search space for clichés (especially for DICs) can easily become intractable. For the concept B1, there is only one relation above/2, and eight attributes: cir/1, ell/1, iso_rangl/1, rangl/1, black/1, red/1, sq/1, and rect/1. The total number of instantiations for DICs is 5184 (Table 7.9). To get an idea of the size of the search space, multiply the number of instantiations by the number of possible variabilizations (which is six; see the next section for a discussion of the variabilization of clichés). With these four DICs, xFOIL-DIC searches through over 30 000 variabilized instantiations of clichés.

For concept B1, the number of instantiations for DICs-B3 is smaller when BK is available (Table 7.10). Predicates IntP# are instantiated to six predicates (intensional

predicates and their children in the taxonomy form/1) instead of eight predicates. The total number of instantiations is less than with the BK, but still large.

	DICs	Number of instantiations
1	ExtP1(X, Y), ExtP2(X), IntP3(X), ExtP4(Y), IntP5(Y)	2304
2	ExtP1(X, Y), ExtP2(X), ExtP3(Y), IntP4(Y)	384
3	ExtP1(X, Y), IntP2(X), ExtP3(Y), IntP4(Y)	288
4	ExtP1(X, Y), ExtP2(X), IntP3(X), IntP4(Y)	288
5	ExtP1(X, Y), ExtP2(X), IntP3(Y)	48
6	ExtP1(X, Y), IntP2(X), IntP3(Y)	36
	Total	3348

Table 7.10. DICs-B3 of Table 7.5 and their number of instantiations.

To reduce xFOIL-CLICHÉS' search space (such that experiments will take a limited time), the number of instantiations per cliché is decreased in the following way.

Table 7.11 shows the restrictions on instantiations in the search space introduced at the beginning of the instantiation process. The number of instantiations allowed per cliché depends on the number of clichés and the number of variable predicates per cliché. The number of instantiations only applies to attributes (*i.e.* of the form literal/1) and not to relations⁵⁶.

If the number of clichés is less than or equal to 10 then xFOIL-CLICHÉS allows a maximum of approximately 500 instantiations. Otherwise the maximum is 100. For example, DIC-1 in Table 7.9 has four variable predicates ExtP2(X), ExtP3(Y), ExtP4(Y), ExtP5(Y) with eight instantiations each. This is over the 500 instantiations allowed. According to the Table 7.9, the four variable predicates should be restricted to five instantiations each for a total of 625 instantiations (5^4).

⁵⁶ Usually only two or three relations were encountered in individual experiments with xFOIL-CLICHÉS.

Conditions	Apply restrictions when instantiations are >	Number of variable predicates	Restrict number of instantiations per variable predicate to	Total
Nb. clichés ≤ 10	500	2	20	400
		3	8	512
		4	5	625
		5	3	243
		$N > 5$	2	2^N
Nb. clichés > 10	100	2	9	81
		3	4	64
		4	3	81
		$N > 4$	2	2^N

Table 7.11. Restrictions on the search space of clichés.

The number of instantiations allowed per variable predicate in clichés depends on the number of clichés and the number of variable predicates per cliché.

Restricting instantiations means that only the first instantiations of literals are preserved, according to the order in which literals are given to the learner. A list with a specific ordering of the literals can also be defined to modify the ordering of the instantiations. The list of the *most frequently used* literals in the FEM domain will be used for one of the experiments (Section 8.3.6).

(3) Variabilization of literals and clichés

A FOIL-type learner requires a list of variables and a partial ordering for these variables. In all experiments, given variables and the partial ordering correspond to the ordered list $[x, y, z, w]$. This affects the number of variables used in rules (the number of edges in the FEM domain and the number of objects in the block domain) and the variabilization.

The **variabilization** for literals with an arity of one and a predicate p is: $p(X), p(Y), p(Z), p(W)$. According to the partial ordering, the variabilization for literals with an arity of two is: $p(X, Y), p(X, Z), p(X, W), p(Y, Z), p(Y, W),$ and $p(Z, W)$. All other combinations are eliminated because they violate the partial ordering.

The variabilization of a cliché is the variabilization of its relation and the propagation of the bindings of these arguments to other literals of the cliché. For example, the DDC: $\text{above}(A, B), \text{cir}(A), \text{rect}(B)$ has the following variabilizations over the ordered list $[X, Y, Z, W]$:

```

above(X, Y), cir(X), rect(Y)
above(X, Z), cir(X), rect(Z)

```

```
above(X, W), cir(X), rect(W)
above(Y, Z), cir(Y), rect(Z)
above(Y, W), cir(Y), rect(W)
above(Z, W), cir(Z), rect(W)
```

In xFOIL, preference is given to literals with old variables over literals with new variables in the following way. Literals with old variables are evaluated first, and the first literal with the highest gain encountered is chosen.

(4) Information gain of an instantiation of a cliché

The information gain of a variabilized instantiation of a cliché is computed of the conjunction of literals rather than of a single literal.

(5) Information gain of a cliché

xFOIL-CLICHÉS first evaluates the information gain of instantiations of a cliché and then evaluates the gain of the cliché itself (*i.e.* as the union of the coverage of examples of all its instantiations) and takes the one with the highest gain. xFOIL-CLICHÉS repeats this process for all clichés and chooses the instantiation or the cliché with the highest gain over all clichés.

For example, DDC-17 `above(X, Y), ell(X), para(Y)` and its four instantiations (Table 7.12) are used to learn the concept B2 (Table 7.14). xFOIL-DDC searches for a cliché to complete the empty clause having the head `block(X)`. xFOIL-DDC computes the information gain for each instantiation (only the two with a positive gain are shown) and then for DDC-17. In this case DDC-17 has a higher gain than any of its instantiations.

	Temporary clause	Gain
I1	<code>block(X) :- above(X,Y), cir(X), rect(Y)</code>	13.74
I2	<code>block(X) :- above(X,Y), cir(X), sq(Y)</code>	6.25
DDC-17	<code>block(X) :- above(X,Y), Ell(X), Para(Y)</code>	19.93

Table 7.12. The two positive gain instantiations of DDC-17 and DDC-17 itself with their information gain when they are added to the empty clause `block(X):-`.

(6) Add a cliché to a clause

Each instantiation of a cliché is a combination of literals and is added to the current rule, possibly resulting in several clauses. Each new clause is a copy of the rule learned so far with the cliché instantiation added.

Following the previous example, adding DDC-17 to a clause corresponds to adding each of its instantiations (having positive gain) to the current rule (*i.e.* `block(X)`) as disjunctive clauses as follows:

```
block(X) :- above(X,Y), cir(X), rect(Y) .
block(X) :- above(X,Y), cir(X), sq(Y) .
```

Since the general cliché had a higher gain than either of its instantiations, both instantiations are added to the rule.

(7) Add an instantiation of a cliché to a clause

An instantiation of a cliché is added to the rule as a conjunction of literals. Duplicate literals are removed.

For example, if an instantiation of a cliché (say `above(X,Y)`, `blue(X)`, `cir(X)`, `rect(Y)`) has the highest gain, then only this instantiation is added to the current rule (say `block(X) :- blue(X)`) and duplicate literals are removed:

```
block(X) :- blue(X), above(X,Y), cir(X), rect(Y) .
```

7.2.3 Restrictions on the search space for clichés

xFOIL-CLICHÉS searches for variabilized instantiations of clichés in the same way that xFOIL searches for variabilized literals to add to a clause. After a few experiments, particularly in the FEM domain, restrictions on the search space for clichés were added to xFOIL-CLICHÉS. These restrictions decrease the CPU time, and have very little affect on learned rules⁵⁷. The restrictions are as follows:

⁵⁷ The first two restrictions express design decisions for xFOIL-CLICHÉS whereas the last three are implementation decisions.

- 1) Only one instantiation of a cliché is learned per clause.
- 2) xFOIL-CLICHÉS stops learning a clause when a cliché (not just one of its instantiation) is added to the rule.
- 3) Clichés (and their instantiations) must introduce at least one new variable, since literals with old variables are evaluated before xFOIL-CLICHÉS searches for clichés.
- 4) After an instantiation of a cliché is added to a clause, xFOIL-CLICHÉS searches only for literals with the new variable introduced by the cliché.
- 5) The target literal is not allowed to occur in any instantiation of a cliché. This avoids introducing recursive calls within a cliché.

7.3 Learning concepts in the *blocks* domain

This section shows the results of xFOIL-CLICHÉS of learning with DDCs or DICs, and with or without BK. Clichés are learned from concept B3 and used to learn concepts B1 and B2.

7.3.1 xFOIL learns no rules for concepts B1 and B2

Without clichés, xFOIL cannot learn any rule for concept B1 or B2. This is because the literal to learn is `block(X)` which describes only one object, and the object `cir(X)` appears in all examples (both positive and negative) as the second object. Moreover, the literal `above/2` (which relates the first object to the second one) is the only literal that could introduce a new variable in the rule, but it also appears in all positive and negative examples. So xFOIL alone cannot learn any literal to discriminate positive and negative examples.

7.3.2 xFOIL-CLICHÉS learns specific rules for concept B1

With clichés learned from concept B3, xFOIL-CLICHÉS learns the same two rules for the concept B1 (Figure 7.5) whether BK is available or not.

```

R1:    block(X)    :-    above(X,Y) , cir(X) , rect(Y) , black(Y) .
R2:    block(X)    :-    above(X,Y) , cir(X) , rect(Y) .

```

Figure 7.5. Rules learned by xFOIL-CLICHÉS for concept B1.

The same instantiation of a cliché `above(X,Y) , cir(X) , rect(Y)` is learned in each rule. In the first rule, the instantiation introduces the variable `Y`, which allows xFOIL-DDC to learn the literal `black(Y)` to discriminate the negative examples covered by this rule. In the second rule, the same instantiation is learned to cover remaining positive examples. Since there exist no other literals to discriminate negative examples, xFOIL-CLICHÉS stops learning.

	No BK	BK
Rule#	DDCs-B3 (Table 7.4)	DDCs-B3 (Table 7.7)
R1	inst-13 & black(Y)	inst-12 & black(Y)
R2	inst-13	inst-12
	DICs-B3 (Table 7.9)	DICs-B3 (Table 7.10)
R1	inst-5 & black(Y)	inst-5 & black(Y)
R2	inst-5	inst-5

Table 7.13. Instantiations of clichés and the literal learned in R1 and R2 of Figure 7.5, with DDCs-B3 or DICs-B3, and with or without the BK.

More precisely, Table 7.13 shows which instantiation was used to learn rules R1 and R2 under each condition (DDCs or DICs, and with or without the BK). For example, in the absence of BK, R1 is learned from the instantiation of DDC-13 `above(X,Y) , cir(X) , rect(Y)` (labelled inst-13 in Table 7.7) and the literal `black(Y)`. xFOIL-DIC is able to learn the same rule R1 from DIC inst-5 (Table 7.9).

Rule R2 is learned with the same instantiation inst-13. Similarly for others.

This example shows that the same definition can be learned under different conditions. It also shows that xFOIL-CLICHÉS does not overgeneralize, even in the presence of the BK. In fact, the knowledge that a *square* is also a *rectangle* is too general to learn this concept, since none of the positive examples describe a *circle* above a *square* and some of the negatives do.

7.3.3 xFOIL-CLICHÉS exploits background knowledge to learn concept B2

xFOIL-CLICHÉS learns different definitions of the concept B2 depending on the kind of clichés used and the availability of the BK (Table 7.14).

No BK	DDCs-B3 (Table 7.7)
block(X) :- above(X,Y), cir(X), rect(Y), white(Y).	inst-13 & white(Y)
block(X) :- above(X,Y), cir(X), rect(Y).	inst-13
BK	DICs-B3 (Table 7.9)
block(X) :- above(X,Y), cir(X), white(Y).	inst-5
block(X) :- above(X,Y), cir(X), rect(Y).	inst-5
block(X) :- above(X,Y), cir(X), sq(Y).	inst-5
BK	DDCs-B3 (Table 7.7)
block(X) :- above(X,Y), cir(X), rect(Y).	DDC-12
block(X) :- above(X,Y), cir(X), sq(Y).	
BK	DICs-B3 (Table 7.10)
block(X) :- above(X,Y), cir(X), rect(Y), white(Y).	DIC-2
block(X) :- above(X,Y), cir(X), sq(Y), white(Y).	
block(X) :- above(X,Y), ell(X), rect(Y), white(Y).	
block(X) :- above(X,Y), ell(X), sq(Y), white(Y).	
block(X) :- above(X,Y), cir(X), rect(Y).	DIC-5
block(X) :- above(X,Y), cir(X), sq(Y).	

Table 7.14. Rules learned for the concept B2
with DDCs-B3, DICs-B3, and with or without the BK.

In the absence of BK, xFOIL-CLICHÉS learns the concept B2 with instantiations of clichés. With DDCs, an instantiation of DDC-13 is used in two rules. This time the literal `white(X)` is added to the first rule to discriminate negative examples still covered with the DDC. Since the only given DDC described the second object as a `rect(Y)`, xFOIL-DDC stops learning without covering all the positives. With DICs, three different instantiations of DIC-5 are learned, resulting in three clauses. No other literals are added. Notice the flexibility provided with DICs. Unlike DDCs, DICs allow xFOIL-DIC to choose the best combinations of literals to describe a concept. For example, xFOIL-DIC chooses `above(X, Y), cir(X), white(Y)` for the first rule, since `white/1` only occurs in positive examples and some of the *squares* and *rectangles* are *white*. Therefore, this rule covers all the positives with a *circle* above a *white square* or *rectangle*, and discriminates all negative examples. It is the shortest instantiation with the highest gain for this concept.

When the taxonomy *form/1* is available, xFOIL-DDC uses the knowledge that a *square* is also a *rectangle* to learn concept B2. For example, both instantiations of DDC-12 are added to the (empty) current rule, resulting in two new rules. xFOIL-CLICHÉS then stops learning according to the second restrictions in Section 7.2.3. xFOIL-DIC also exploits BK when it uses DIC-2 and DIC-5. DIC-2 contains the knowledge that a *circle* is also an *ellipse*, and that a *square* is also a *rectangle*. The fact that a *circle* is also an *ellipse* is more general than what is needed to cover positive examples of concept B2, since the first object of the positives is always a *circle*. On the other hand, none of the negatives describe the first object as an *ellipse*. Such instantiations have a positive gain and are learned as well. DIC-5 gives the knowledge that a *square* is also a *rectangle* and results in the last two rules.

7.4 Conclusion

DDCs are expressed with predicates that belong to the domain in which they are learned. In the absence of BK, only the provided DDCs are used to learn the concept. In the presence of BK, intensional predicates supply additional instantiations of DDCs to the learner. DDCs are expressed with extensional or intensional predicates that belong to the current domain.

DICs are expressed with variable predicates that are instantiated to predicates in the target domain. In the absence of BK, DIC predicates are instantiated to extensional predicates. In the presence of BK, DIC predicates of the form *IntP#* are instantiated to intensional predicates and their children in the taxonomy. DIC predicates of the form *ExtP#* are instantiated to extensional predicates of the target domain.

When xFOIL-CLICHÉS learns a rule that still covers negative examples and it cannot find a single literal with a gain, it searches for a cliché. xFOIL-CLICHÉS adds a cliché when DDCs provide a conjunction of literals that have a gain for the concept in the same domain in which the DDCs were learned. xFOIL-CLICHÉS also adds a cliché when DICs

provide a pattern of variables that—when instantiated to literals in the target domain—have a gain for the concept in the target domain.

xFOIL-CLICHÉS does not overgeneralize the definition of a concept, even in the presence of knowledge. DDCs influence the choice of the vocabulary to describe the rules. To be useful, a DDC must be expressed with a vocabulary (or specific predicate names) that gives a gain to learn the concept. In the absence of BK, knowledge is explicitly expressed with a conjunction of literals in the DDCs. For instance, rules learned for concepts B1 and B2 are mostly described in terms of the objects' shape, because DDCs are described using these literals. Different rules could have been learned with DDCs describing objects by their color and by their shape and color. Moreover, no other DDCs described the additional knowledge required (*i.e.* *circle* is above a *square*) to cover the remaining positive examples for concept B2. Therefore, xFOIL-DDC covers only a subset of the positives. In the presence of BK, a DDC is useful as long as it is at least as general as what is needed to learn a rule. For example, the knowledge that a *square* is also a *rectangle* is useless for learning concept B1, but useful for learning concept B2. Therefore, only an instantiation (where a *rectangle* does not imply a *square* too) of a DDC is learned for B1. On the other hand, that knowledge is useful to B2 and the DDC itself (*i.e.* with both instantiations) is learned.

DICs are more flexible than DDCs (but are also more expensive). They only provide patterns of variables, not specific literals. The vocabulary is chosen with the shortest combination of literals that gives the highest gain on examples. They are useful as long as at least one of the instantiations of a combination results in a gain. For example, in the absence of BK, xFOIL-DIC learns one rule for concept B2 with the shape of the first object and the color of the second one rather than a rule with both objects' shape. In the presence of knowledge, xFOIL-DIC learns the best combination of literals using the available knowledge.

8 Empirical results

This chapter gives empirical results showing that clichés are useful to learn concepts within and across domains. Clichés are learned with CLUSE and used with xFOIL-CLICHÉS. For each kind of cliché (DDCs and DICs), the definition of a concept learned from xFOIL is compared to the definition of the same concept learned with xFOIL-CLICHÉS. Hypotheses learned with xFOIL and xFOIL-CLICHÉS are compared in different dimensions.

The first section presents the design of the experiments. The second section describes the dimensions in which definitions learned with xFOIL and xFOIL-CLICHÉS are compared. Sections 8.3 and 8.4 present empirical results in the FEM and the *blocks* domains.

8.1 Design of empirical testing

Independently from each other, CLUSE learns clichés and xFOIL learns hypotheses (Figure 8.1).

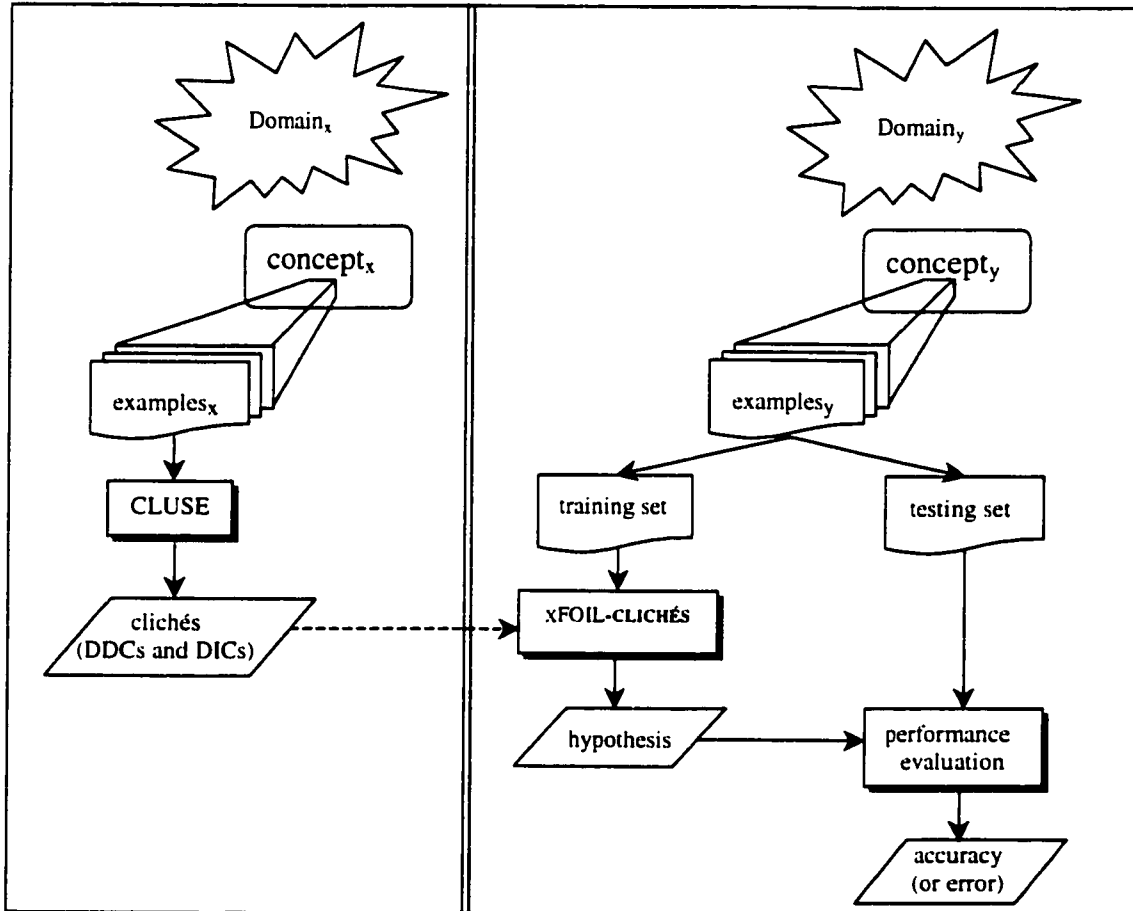


Figure 8.1. CLUSE learns clichés and xFOIL learns hypotheses.

xFOIL uses a training set to learn the hypothesis, and a separate testing set to evaluate the accuracy (or error). xFOIL learns with or without clichés. Clichés can either be DDCs or DICS. xFOIL uses DDCs when it learns a concept in the same domain as the concept from which clichés are learned. When domains are different xFOIL uses DICS.

Both systems learn from examples of a concept. xFOIL uses a training set to learn the hypothesis, and a separate testing set to evaluate the **accuracy** (or **error**)⁵⁸ of this hypothesis over subsequent data. xFOIL learns with or without clichés. Clichés can either be DDCs or DICS. xFOIL uses DDCs when it learns concepts in the same domain as the

⁵⁸ The **accuracy** is equal to $1 - \text{error}$. Both measures will be used in this chapter.

concept from which clichés are learned. It is more likely that concepts in the same domain are described using the same vocabulary (or predicate names). When domains are different xFOIL uses DICs. Clichés are learned with CLUSE from examples and used with xFOIL whenever they are needed. For each concept a set of DDCs and a set of DICs are learned⁵⁹. To evaluate the significance of using clichés with a FOIL type learner, learning a concept with xFOIL (without clichés) is compared to learning the same concept with xFOIL-CLICHÉS (where clichés are either DDCs or DICs).

To show that clichés learned in one domain are useful to learn concepts in another domain, two domains of application are needed. It has been stated already that a FOIL type learner would benefit from learning with a kind of lookahead with FEM data (see Chapter 3). FEM data consists of a large number of examples described with relations and factors (or attributes)⁶⁰. Relations need to be added to the definition of a concept in order to introduce new variables in a rule. Because relations occur in both positive and negative examples, they are rarely learned in rules. Such relations are provided by clichés. In this chapter DDCs learned from one concept in the FEM domain are used to learn other concepts in that domain. Furthermore, DICs from the *blocks* domain are used to learn FEM concepts.

8.2 Dimensions of evaluation

This section describes the dimensions in which the two learning algorithms are compared. Compared algorithms learn on the same training examples and are tested on the same testing (or validation) examples of a concept. The procedure to compare two learning algorithms computes the statistical significance of the difference in the error of the two algorithms. The cost of applying the two algorithms and the compactness of learned hypotheses (or definitions) are also compared.

⁵⁹ One could imagine that clichés are preserved in a KB and used with xFOIL whenever they are needed to learn concepts in different domains.

⁶⁰ I will use relations to refer to literals with more than one argument and to attributes for literals with a single argument. When no distinction is needed, I will refer to them as literals.

8.2.1 *k*-fold cross-validation to compare learning algorithms

To show that clichés provide an appropriate lookahead to improve the *accuracy*⁶¹ of xFOIL in different domains, a procedure to estimate the difference in error of two learning algorithms and to determine whether the observed difference is statistically significant is used.

The difference in error between two learning algorithms (or methods) is estimated with a cross-validation procedure (Algorithm 8.1).

Partition the available data D_0 into k disjoint subsets $T_1, T_2, \dots, T_k$ of equal size, where this size is at least 30.

for i from 1 to k ,

use T_i for the test set and the remaining data for training set S_i

$S_i \leftarrow \{D_0 - T_i\}$

$h_A \leftarrow L_A(S_i)$ (h_A is the hypothesis learned by L_A)

$h_B \leftarrow L_B(S_i)$ (h_B is the hypothesis learned by L_B)

$\delta_i \leftarrow \text{error}_{T_i}(h_A) - \text{error}_{T_i}(h_B)$

return the value $\bar{\delta}$ where

$$\bar{\delta} \equiv \frac{1}{k} \sum_{i=1}^k \delta_i$$

(F1)

*Algorithm 8.1. Cross-validation procedure
to estimate the difference in error
between two learning algorithms L_A and L_B .*

This procedure first partitions the data into k disjoint subsets of equal size. It then *trains* and tests the learning algorithms k -times, using each of the k subsets in turn as the test set, and using all remaining data as the training set. Test sets generated by k -fold cross-validation are independent, since each instance is included in only one test set. This way the learning algorithms are tested on k independent test sets, and the mean difference in errors $\bar{\delta}$ is returned as an estimate of the difference between the two learning algorithms.

⁶¹ *Accuracy* = $1 - \text{error}$. The sample error of a hypothesis with respect to some sample S of instances, is the fraction of S that it misclassifies:

$$\text{error}_S(h) = \frac{\# \text{ misclassified positives} + \# \text{ misclassified negatives}}{\# \text{ positives} + \# \text{ negatives}}$$

The underlying conditions required to apply the procedure are:

- 1) to have enough data to partition them into k disjoint subsets of equal size, where this size is at least thirty⁶²;
- 2) to draw examples independently of the hypothesis and of one another⁶³.

Statistically, the above procedure corresponds to a *two-sided paired t-test*, since hypotheses are evaluated on identical test sets (a paired test), and the hypotheses is that performances differ (two-sided test). The *t test* is a significance test of differences in a collection of independent, identically and Normally distributed random variables. So any differences in observed errors are due to differences between hypotheses, instead of differences in the makeup of the samples.

8.2.2 Difference in error of two learning algorithms

The approximate $N\%$ confidence interval for estimating the average difference in error between the two learning methods using $\bar{\delta}$ is given by

$$(F2) \quad \bar{\delta} \pm t_{N,k-1} S_{\bar{\delta}}$$

where $t_{N,k-1}$ is the value for two-sided confidence intervals. N is the level of confidence. $k-1$ is the number of degrees of freedom related to the number of times the procedure (or number of independent random events) is repeated to produce the variable $\bar{\delta}$. $S_{\bar{\delta}}$ is an estimate of the standard deviation of the distribution governing $\bar{\delta}$. $S_{\bar{\delta}}$ is defined as:

$$S_{\bar{\delta}} \equiv \sqrt{\frac{1}{k(k-1)} \sum_{i=1}^k (\delta_i - \bar{\delta})^2}$$

For the purpose of this work, unless stated otherwise all differences in error are averaged over all partitions of the data under learning. The level of confidence is presented as a

⁶² Because of this, the individual δ_i will each follow an approximately Normal distribution (due to the Central Limit Theorem). And when individual Y_i each follows a Normal distribution, then the sample mean $\bar{Y}$ follows a Normal distribution as well.

⁶³ All examples of a concept are put in a bag and randomly chosen to make the training and testing sets.

percentage. The confidence intervals are the approximate 95% confidence intervals. Any difference with confidence level of at least 95% (or smaller than 0.05 level of significance) is considered *statistically significant* or simply *significant*.

8.2.3 Cost of an algorithm

Another useful dimension for comparing algorithms is the *cost* of execution. The cost of an algorithm measures the number of search steps and the CPU time required to learn a concept. The number of search steps depends on the number of variabilized elements (Section 7.2.2) in the search space. Elements are either literals for xFOIL, or literals and clichés (DDCs or instantiations of DICs) for xFOIL-CLICHÉS. For example, xFOIL-CLICHÉS searches among the literals: `neighbour(X, Y)`, `neighbour(X, Z)`, and `neighbour(Y, Z)`, in order to add the literal `neighbour(X, Y)` to the rule `mesh(X, Y):- not_loaded(X), short(X)`.

Although time is proportional to the search steps, the *CPU time* is also given to show the absolute time cost of experiments. For simplicity and because experiments are long the CPU time appears in minutes unless otherwise mentioned.

8.2.4 Compactness of hypothesis

The *compactness* of a hypothesis is another dimension by which two learning algorithm are compared. The number of rules per hypothesis, the number of literals and the number of variables per rule define its compactness. Again, these are averages computed for all partitions of the data during the experiment.

8.3 Empirical results in the FEM domain

In the FEM domain, the experiment consists of learning eight design rules (or one definition). One rule is learned for each class of edges, *i.e.* edges with the same number of FEs on them. Each class of edges is a FEM concept (Chapter 3).

The experiment compares xFOIL to xFOIL-DDC, and xFOIL to xFOIL-DIC. According to the procedure to compare learning algorithms (Section 8.2.1), this corresponds to

applying the algorithm first with L_A representing xFOIL and L_B representing xFOIL-DDC, and then with L_A representing xFOIL and L_B representing xFOIL-DIC. Since results from xFOIL are needed for both comparisons, xFOIL alone (*i.e.* without clichés) is executed first (Section 8.3.1). For each partition a hypothesis is learned from the training set and the hypothesis error is evaluated on the test set. In Section 8.3.2, xFOIL-DDC is executed on the same training and testing sets. This returns errors as well, which can be compared with the xFOIL errors to evaluate the difference in learning with DDCs. The difference in error between xFOIL and xFOIL-DIC is evaluated in Section 8.3.4.

Number of training and testing examples

Table 8.1 shows the number of examples used for the 3-fold cross-validation on FEM concepts. This corresponds to learning on 36 positive examples and 253 negatives, and to testing on 18 positive examples and 126 negatives (on average).

Mesh	Training		Testing	
	Positive	Negative	Positive	Negative
1	82	207	41	103
2	73	215	37	108
3	25	263	13	132
4	31	257	16	129
5	13	276	6	138
6	17	271	9	136
7	11	278	5	139
8	36	253	18	126
Average	36	253	18	126

Table 8.1. The number of positive and negative examples in the training and testing sets, for each FEM concept.

Number of edges described in definitions

A maximum of three edges can be described in learned definitions for a FEM concept, since four variables (x , y , z , and w) are given to xFOIL (and xFOIL-Clichés). The first two variables are used with the literal to learn (*i.e.* the head of the rule). The literal to learn is `mesh( $x$ ,  $y$ )`, where x represents an edge and y the number of FEs on that edge. The other two variables can be introduced with the relation `opposite` or `neighbour`, which links two edges together. When added to a rule such a relation links an existing edge (one already in the rule) to a new edge. Since there is already one edge in the head

of the rule and there are a total of four available variables, the maximum number of edges per rule is three. For each additional variable, the search space grows exponentially⁶⁴.

Constraints on the rules

Each edge in a rule should have at least one attribute describing it. This constraint is expressed by two of the three conditions in Dolsak *et al.* (1994)⁶⁵. Rules that violate one of them are removed before testing. These constraints are:

- (C1) *specificity of an existing edge*: an existing (or old) edge needs to be described with at least one literal not referring to a geometric relation with other edges, such as:

```
mesh(A, B):- fixed(A), neighbour(A, C).
```

- (C2) *specificity of a new edge*: a new edge introduced with a relation `neighbour` (or `opposite`) needs to be described with at least one literal and not only refer to a geometric relation with other edges, like:

```
mesh(A, B):- neighbour(C, A), fixed(C).
```

8.3.1 xFOIL

This section shows the FEM concepts learned by xFOIL. The branching factor and an example of learned rules for one of the concepts are shown. This section also introduces statistical results that are used in the following sections to compare xFOIL to xFOIL-CLICHÉS.

The first experiment consists of learning FEM concepts with xFOIL alone. xFOIL uses a training set to learn a hypothesis and uses a separate testing set to evaluate the hypothesis' accuracy. The cross-validation is applied for each FEM concept. ($\text{concept}_m = \text{mesh-1 to mesh-8}$). Appendix VIII lists sets of rules learned with xFOIL for mesh-4.

Table 8.2 shows the maximum branching factor for xFOIL for a FEM concept. FEM concepts are described with 2 relations and 19 attributes. The maximum number of

⁶⁴ Experiments show that three variables is usually sufficient since all rules were learned with at most 3.

⁶⁵ The third condition is omitted, since it never occurs in experiments with FEM concepts.

variabilizations is six for relations and four for attributes (Section 7.2.2). The number of literals in the search space equals the sum of the number of literals times their corresponding variabilizations. Therefore, the maximum⁶⁶ branching factor is $2 * 6 + 19 * 4 = 88$ variabilized literals.

Literal/arity	Nb. predicates	Nb. variabilizations	Total
relation/2	2	6	12
attribute/1	19	4	76
Total	21	<i>non applicable</i>	88

Table 8.2. The maximum branching factor for xFOIL.

Literals are either relations or attributes. The summation of literals times the corresponding variabilization gives the total number of variabilized literals in the search space.

Figure 8.2 shows the rules that xFOIL learns for the first partition of mesh-3. Seven rules (2, 3, 4, 5, 7, 9, and 10) describe only one edge (denoted by variable x). In the other rules (1, 6, and 8) the relation *opposite* introduces a second edge z ⁶⁷. Similar rules are learned for the other two partitions of that concept.

-
- 1 *mesh*(X, Y) :- *usual*(X), *not_loaded*(X), *opposite*(X, Z), *short_for_hole*(Z).
 - 2 *mesh*(X, Y) :- *usual*(X), *not_loaded*(X), *free*(X).
 - 3 *mesh*(X, Y) :- *usual*(X), *fixed*(X), *cont_loaded*(X).
 - 4 *mesh*(X, Y) :- *usual*(X), *not_loaded*(X), *fixed*(X).
 - 5 *mesh*(X, Y) :- *one_side_fixed*(X), *short*(X), *not_loaded*(X).
 - 6 *mesh*(X, Y) :- *one_side_fixed*(X), *not_loaded*(X), *opposite*(X, Z), *usual*(Z).
 - 7 *mesh*(X, Y) :- *one_side_fixed*(X), *not_loaded*(X), *usual*(X).
 - 8 *mesh*(X, Y) :- *short*(X), *opposite*(X, Z), *cont_loaded*(Z), *short*(Z).
 - 9 *mesh*(X, Y) :- *free*(X), *one_side_loaded*(X).
 - 10 *mesh*(X, Y) :- *cont_loaded*(X), *usual*(X), *free*(X).
-

Figure 8.2. Mesh-3 learned rules with xFOIL.

In the FEM domain, xFOIL is cheap and compact. Table 8.3 shows that on average xFOIL learns concepts in 4 minutes of CPU time and requires 178 steps (*i.e.* the number of variabilized literals tried) to search for literals. Rules learned with xFOIL are general.

⁶⁶ This is the upper bound because xFOIL looks for variabilized literals that introduce at most one new variable to the rule.

⁶⁷ Variable Y describes the number of FEs on edges, not an edge.

xFOIL learns (on average) 13 rules of 3 literals describing 2 variables (*i.e.* only one edge). On the testing sets, learned rules cover 82% of the positive examples, and 29% of the negative examples, giving 26% error. The error is high because xFOIL has difficulty using relations *opposite* and *neighbour*. These relations occur in all positive and negative examples so they do not discriminate between positive and negative examples. The relations are also the only literals that could introduce a new variable (or edge) in a rule. In the whole experiment xFOIL uses the relation *opposite* rarely, and almost never learns the relation *neighbour*. Moreover, the relation *neighbour* is learned only if the relation *opposite* is already used in a rule (Appendix VIII).

Mesh	Learning						Testing		
	Neg.left%	Rules	Lit/rule	Var/rule	Steps	CPU	PC%	NC%	Err%
1	44	17	3	2	138	3	98	45	33
2	50	24	4	3	216	11	86	49	41
3	53	10	3	2	184	2	84	29	28
4	54	10	3	2	189	2	85	20	20
5	44	6	3	2	169	1	70	16	16
6	36	11	3	2	171	2	66	17	18
Avg.	47	13	3	2	178	4	82	29	26
7	5	7	3	2	160	2	47	6	8
8	6	10	2	2	129	2	89	8	8

Table 8.3. Statistical results for xFOIL on learning and testing each FEM concept.

The first column represents the FEM concepts. The second column represents the training negative examples covered by learned rules. The next three columns represent the compactness and the next two columns represent the cost of the rules learned. The last three columns show the positive and negative coverage and the error of learned rules on testing sets.

Table 8.3 shows that xFOIL performs very well (with only 8% error on testing sets) for mesh-7 and mesh-8. For these concepts and the way that the significance is computed, it is almost impossible to significantly improve that performance. In fact, xFOIL-CLICHÉS would need an error rate of at most 1% on average. And even a 0% error rate for mesh-8 would not make a significant difference. Therefore, these two concepts are removed from the experiment with xFOIL-CLICHÉS.

Notice that the number of rules that satisfy the specificity constraints (C1) and (C2) on average for all FEM concepts is only one. These numbers are not shown.

8.3.2 xFOIL-DDC

To show that clichés learned in one domain provide appropriate lookahead to improve xFOIL's accuracy to learn concepts in the same domain, xFOIL-CLICHÉS is first used with DDCs. DDCs are learned with CLUSE⁶⁸ for each FEM concept (mesh-1 to mesh-6). Then xFOIL-CLICHÉS uses DDCs from each concept to learn each FEM concept in turn (*i.e.* 36 executions). To show that DDCs improve xFOIL's accuracy, the difference in error of xFOIL and xFOIL-DDC is estimated using the procedure described in Algorithm 8.1 for each concept.

I believe that a DDC that covers less than 5% of the positive examples when learned with CLUSE is unlikely to perform well (or have a gain) when used with xFOIL-DDC on another concept. Although CLUSE's parameters are set to limit the number of clichés learned, many DDCs are obtained for each FEM concept. In the FEM domain, almost half of them are specific to the concept learned, since their frequency on covering positive examples is very small⁶⁹. To preserve only DDCs that covered at least 5% of positive examples when learned with CLUSE, xFOIL-DDC's *FreqPos* parameter is set to 0.05. Table 8.4 shows that this decreases on average the number of DDCs to 50% of the total available DDCs learned by CLUSE.

Table 8.5 shows that 47% of the time (17 times out of 36) DDCs significantly improve xFOIL's accuracy of a learned FEM concept⁷⁰. For example, xFOIL's accuracy of learning mesh-1 is significantly improved with DDCs learned from meshes 2, 3, 4, 5, and 6. Notice that only DDCs learned from three concepts (mesh-3, mesh-4, and mesh-6) significantly improve xFOIL's accuracy to learn the same concept. This is because clichés are learned after literals are added to a rule. Even though clichés are learned with CLUSE from the same concept, they often make a rule too specific to have a gain.

⁶⁸ Parameter values for CLUSE: SimCh = 0.01, similarity bindings = 0.01, PCov = 0.025, NCov = 1.0 and BK = off. For mesh concepts, none of the extensional predicates are also intensional, so the background knowledge is not used, and the BK is always set to off.

⁶⁹ Moreover in the FEM domain, the large number of DDCs learned is partially due to the symmetric chains that CLUSE learns from. These symmetric chains are introduced by the two relations *neighbour/2* and *opposite/2* and produce symmetric DDCs. For example, *neighbour(X, Y), fixed(X), not_loaded(X), free(Y), usual(Y)* is symmetric with *neighbour(X, Y), free(X), usual(X), fixed(Y), not_loaded(Y)*.

⁷⁰ Definitions learned with DDCs-2 for mesh-4 are listed in Appendix IX.

DDC	Learned with CLUSE	Used in xFOIL-DDC (FreqPos = 0.05)
1	45	13
2	54	20
3	80	29
4	60	40
5	69	35
6	56	26
Average	61	27

Table 8.4. The number of DDCs learned with CLUSE for each FEM concept vs. the number of DDCs used with xFOIL-DDC with FreqPos = 0.05.

xFOIL-DDC significantly improves upon xFOIL when enough DDCs are learned to improve rules with the lowest testing accuracy. For example, xFOIL learns on average 10 rules with 80% testing accuracy per partition for the concept mesh-4. For the same concept, xFOIL-DDC-5 (xFOIL-DDC with DDCs from mesh-5) learns 13 rules containing 7 DDCs with 88% testing accuracy. DDC-5 improves enough rules that otherwise would have the lowest testing accuracy to make an overall difference on the testing accuracy. On the other hand, insufficient DDCs are learned from mesh-1 to make a difference in learning mesh-4. xFOIL-DDC-1 learns 10 rules with only 2 DDCs and 81% testing accuracy. Because DDCs-1 improve only one of the rules with the lowest testing accuracy and the definition is a set of disjunctive rules, these DDCs are insufficient to improve the rules learned from xFOIL.

		DDCs					
		1	2	3	4	5	6
Mesh	1	0.07 ± 0.17	0.15 ± 0.06	0.15 ± 0.04	0.16 ± 0.04	0.16 ± 0.05	0.10 ± 0.09
	2	0.04 ± 0.09	0.09 ± 0.17	0.11 ± 0.13	0.11 ± 0.11	0.10 ± 0.19	0.11 ± 0.09
	3	0.04 ± 0.11	0.10 ± 0.08	0.08 ± 0.07	0.10 ± 0.09	0.09 ± 0.08	0.07 ± 0.12
	4	0.0 ± 0.0	0.04 ± 0.01	0.05 ± 0.03	0.08 ± 0.05	0.08 ± 0.04	0.02 ± 0.07
	5	0.03 ± 0.14	0.08 ± 0.16	0.06 ± 0.11	0.08 ± 0.15	0.09 ± 0.13	0.04 ± 0.14
	6	0.03 ± 0.11	0.09 ± 0.09	0.11 ± 0.14	0.09 ± 0.10	0.10 ± 0.14	0.09 ± 0.09

Table 8.5. The difference in error between xFOIL and xFOIL-DDC ± the standard deviation. Columns represent DDCs learned from FEM concepts with CLUSE. Rows represent FEM concepts learned with xFOIL-DDC. **Shaded** values represent a significant difference in favour of xFOIL-DDC.

xFOIL-DDC learns more specific rules than xFOIL but it is more expensive to apply (Table 8.6). xFOIL-DDC learns on average 16 rules with 5 literals describing three variables (*i.e.* two edges) per rule compared to 13 rules with 3 literals and one edge with

just xFOIL. Learned rules cover 29% of negative examples in the training set, compared to 47% with xFOIL. Learned rules cover 74 positive and 18 negative examples, compared to 82 and 29 for xFOIL, giving an error on positive examples of 18% instead of 26%. Finally, using DDCs almost doubles the CPU time.

Mesh	Learning									Testing		
	Neg.left%	Rules	Lit/rule	#Cli	Lit/cli	Dup.lit	Stp.lit	Stp.cli	CPU	PC%	NC%	Err%
1	39	14	4	4	3	1	184	17	5	81	25	22
2	29	16	5	9	4	1	202	36	9	73	17	17
3	25	16	5	10	5	1	206	47	9	71	17	17
4	25	17	5	13	4	1	201	69	10	72	15	16
5	27	17	5	11	5	2	206	58	10	71	16	16
6	30	16	5	9	4	1	194	42	8	77	21	19
Avg.	29	16	5	9	4	1	199	45	9	74	18	18

Table 8.6. Statistical results for xFOIL-DDC on learning and testing each FEM concept. The first column represents the FEM concepts. The second column represents the training negative examples covered by learned rules. The next three columns represent the compactness and the next two columns represent the cost of the rules learned. The last three columns show the coverage and the error of learned rules on testing sets.

As far as compactness of individual rules is concerned, this experiment reveals that an average of only one literal is duplicated between DDC's literals and literals in the rule before the DDC is added. It also reveals (after verification) that adding a DDC to a rule always improves the rule's testing accuracy (not shown). Consequently, if enough DDCs are learned to improve rules with the lowest accuracy, then the accuracy for the set of rules of the concept will improve.

8.3.3 xFOIL-DDC with more specific DDCs

The previous experiment shows that there remains some room for improvement. One possibility is that the more specific DDCs (those that were not used in the previous experiment) would provide better lookahead. The following experiment is the same as the previous experiment, but with DDCs that cover at least 1% (instead of 5%) of the positive chains when learned.

Table 8.7 shows the number of DDCs with a coverage frequency for positive chains of at least 1% from the total number of DDCs learned with CLUSE.

DDC	Learned with CLUSE	Used in xFOIL-DDC (FreqPos = 0.01)
1	45	45
2	54	51
3	80	80
4	60	59
5	69	69
6	56	56
Average	61	60

Table 8.7. The number of DDCs learned with CLUSE for each FEM concept vs. the number of DDCs used with xFOIL-DDC with FreqPos = 0.01.

Table 8.8 shows, as with DDCs-5%, DDCs-1% significantly improve xFOIL's accuracy on 17 of 36 FEM concepts. Although the number of DDCs that bring a significant improvement over xFOIL is the same as xFOIL-DDC-5%, some of the intervals differ. Mesh-1 with DDCs-6, mesh-2 with DDCs-4, mesh-6 with DDCs-2 are no longer significant improvements. A closer look to these results suggests that at 1% some rules overfit the training set. This also explains the increase of the variance instead of the accuracy. On the other hand DDCs-1 make a significant improvement for mesh-1 and mesh-3, and DDCs-5 for mesh-2.

		DDCs-1%					
		1	2	3	4	5	6
Mesh	1	0.11 ± 0.08	0.17 ± 0.07	0.18 ± 0.11	0.16 ± 0.08	0.15 ± 0.04	0.11 ± 0.12
	2	0.11 ± 0.14	0.11 ± 0.14	0.12 ± 0.15	0.12 ± 0.15	0.11 ± 0.10	0.10 ± 0.09
	3	0.09 ± 0.08	0.11 ± 0.05	0.08 ± 0.06	0.09 ± 0.01	0.09 ± 0.04	0.07 ± 0.12
	4	0.06 ± 0.09	0.03 ± 0.03	0.08 ± 0.04	0.08 ± 0.05	0.06 ± 0.04	0.04 ± 0.14
	5	0.07 ± 0.13	0.07 ± 0.14	0.08 ± 0.12	0.08 ± 0.15	0.09 ± 0.14	0.04 ± 0.14
	6	0.09 ± 0.11	0.10 ± 0.14	0.10 ± 0.14	0.09 ± 0.10	0.10 ± 0.14	0.08 ± 0.03

Table 8.8. The difference in error between xFOIL and xFOIL-DDC-1% ± the standard deviation. Columns represent DDCs learned from FEM concepts with CLUSE. Rows represent FEM concepts learned with xFOIL-DDC-1%. Shaded intervals represent a significant difference in favour of xFOIL-DDC-1%.

Table 8.9 shows statistical results for xFOIL-DDC-1%. Although on average it learns one more rule than xFOIL-DDC-5% the number of literals and the number of variables per rule are the same. It learns more specific clichés (the number of literals per cliché is now 5 instead of 4), but the number of duplicate literals also increases by 1 (from 1 to 2). In terms of coverage, xFOIL-DDC-1% learns more specific rules than xFOIL-DDC-5%.

Learned rules with xFOIL-DDC-1% cover 26% of the negative examples, compared to 29% for rules learned with xFOIL-DDC-5%. On testing sets, rules learned with xFOIL-DDC-1% cover fewer positives (71% compared to 74%) and fewer negatives (16% instead of 18%), and have a lower error rate (17% instead of 18%). Learning more specific rules costs more for xFOIL-DDC-1%. It takes almost the same number of steps to search for literals (201 steps vs. of 199), but because it searches through twice as many DDCs, it takes twice as many steps to search for a cliché. That extra work done by xFOIL-DDC-1% requires 20% more CPU time on average.

Mesh	Learning									Testing		
	Neg.left%	Rules	Lit/rule	#Cli	Lit/cli	Dup.lit	Stp.lit	Stp.cli	CPU	PC%	NC%	Err%
1	30	18	5	11	5	1	200	67	10	74	18	18
2	24	17	5	11	5	2	200	76	10	69	15	16
3	22	17	5	12	5	2	205	118	12	70	14	15
4	25	17	5	12	4	1	201	94	11	72	15	16
5	25	17	5	13	5	2	203	105	11	68	15	16
6	30	17	5	11	5	1	195	84	10	75	20	19
Avg.	26	17	5	12	5	2	201	91	11	71	16	17

Table 8.9. Statistical results for xFOIL-DDC-1% on learning and testing each FEM concept. The first column represents the FEM concepts. The second column represents the training negative examples covered by learned rules. The next three columns represent the compactness and the next two columns represent the cost of the rules learned. The last three columns show the coverage and the error of learned rules on testing sets.

Note that neither xFOIL-DDC-1% nor xFOIL-DDC-5% could significantly improve learning mesh-5. I repeated the experiment with DDCs from all the FEM concepts at the same time and executed xFOIL-DDC to learn mesh-5. Again, there was no significant improvement over xFOIL. Recall however that xFOIL alone had an accuracy of 84% on mesh-5 (the highest of the six FEM concepts).

8.3.4 DICs from the *blocks* domain

The next experiment takes DICs learned from one *blocks* concept and uses them to learn FEM concepts. To evaluate the improvement of xFOIL-DIC over xFOIL, the difference of their error is estimated with the procedure described in Algorithm 8.1 for each FEM concept.

Table 8.10 shows DICs learned by CLUSE from a *blocks* concept. These DICs are used with xFOIL-DIC to learn FEM concepts. Since no BK is used, intensional predicates (*IntP#*) are treated as extensional predicates (*ExtP#*).

Id	DIC
1	<i>ExtP1</i> (X,Y) , <i>ExtP2</i> (Y)
2	<i>ExtP1</i> (X,Y) , <i>ExtP2</i> (Y) , <i>IntP3</i> (Y)
3	<i>ExtP1</i> (X,Y) , <i>ExtP2</i> (X) , <i>ExtP3</i> (Y)
4	<i>ExtP1</i> (X,Y) , <i>ExtP2</i> (X) , <i>ExtP3</i> (Y) , <i>IntP4</i> (Y)

Table 8.10. DICs learned from a *blocks* concept.

8.3.5 xFOIL-DIC

This section evaluates xFOIL-DIC in the FEM domain with DICs learned from the *blocks* domain⁷¹ and shows an example of rules learned with xFOIL-DIC for one of the concepts. Results of xFOIL-DIC and the experiment with xFOIL are compared (Section 8.2.2). Finally, the cost of using DICs is evaluated.

The search space for xFOIL-DIC contains the search space of variabilized literals and DICs. Table 8.11 shows the number of variabilized DICs for xFOIL-DIC. There are four DICs all described with variable predicates.

DICs	Nb. instantiations	Reduced	Nb. variabilizations
1	38	38	228
2	722	722	4332
3	722	722	4332
4	13718	1024	6144
Total	15544	2506	15036

Table 8.11. The branching factor for clichés with xFOIL-DIC for attribute's arguments. The number of instantiations before and after reduction, and the number of variabilizations after reduction.

There are two relations and nineteen attributes to describe an edge. The number of instantiations of a DIC is equal to the number of instantiations for a relation (only one per cliché) times the number of attributes (19) times the number of instantiations for these attributes. Taking into account the partial ordering of variables, the number of

⁷¹ Appendix X lists sets of rules learned with xFOIL-DIC for mesh-4.

variabilizations for each DIC is equal to six. For example, the fourth DIC has a relation with one literal describing the argument x , and two literals describing argument y . So there are 13718 instantiations ($2 * 19^3$). After the reduction procedure (Section 7.2.3) the number of instantiations becomes 1024 (*i.e.* $2 * 512$). Literals are ordered by the frequencies of their occurrences in all the concepts. Another heuristic used when the number of instantiations needs to be reduced (say to N literals) is to choose only the first N of the most frequently used literals for instantiations (Section 7.2.2). This reduces the search space to frequently occurring literals in FEM concepts. Finally the number of variabilizations for DIC-4 is 6144.

Figure 8.3 gives the set of rules learned by xFOIL-DIC for the first partition of the concept mesh-3.

-
- 1 **mesh(X,Y) :- usual(X), not_loaded(X), opposite(X,Z), short_for_hole(Z).**
 - 2 mesh(X,Y) :- neighbour(X,Z), free(Z), long(Z), usual(X), not_loaded(X), free(X).
 - 3 mesh(X,Y) :- neighbour(X,Z), cont_loaded(Z), short(Z), usual(X), fixed(X), cont_loaded(X).
 - ~~4 mesh(X,Y) :- neighbour(X,Z), fixed(Z), short(Z), usual(X), not_loaded(X), fixed(X), opposite(Z,W).~~
 - 5 mesh(X,Y) :- usual(X), not_loaded(X), opposite(X,Z), cont_loaded(Z), usual(Z), one_side_fixed(X).
 - 6 mesh(X,Y) :- neighbour(X,Z), free(Z), short(Z), usual(X), not_loaded(X), free(X).
 - 7 mesh(X,Y) :- usual(X), not_loaded(X), fixed(X), opposite(X,Z), usual(Z), cont_loaded(Z).
 - 8 mesh(X,Y) :- usual(X), not_loaded(X), fixed(X), opposite(X,Z), not_loaded(Z).
 - 9 **mesh(X,Y) :- one_side_fixed(X), short(X), not_loaded(X).**
 - 10 mesh(X,Y) :- neighbour(X,Z), free(Z), long(Z), usual(X), not_loaded(X), one_side_fixed(X).
 - 11 mesh(X,Y) :- neighbour(X,Z), cont_loaded(Z), short(Z), usual(X), free(X), cont_loaded(X).
 - 12 **mesh(X,Y) :- short(X), opposite(X,Z), cont_loaded(Z), short(Z).**
 - 13 mesh(X,Y) :- neighbour(X,Z), fixed(X), one_side_loaded(Z), usual(X), not_loaded(X), short(Z).
 - 14 mesh(X,Y) :- neighbour(X,Z), free(Z), half_circuit(Z), free(X), one_side_loaded(X).
 - 15 mesh(X,Y) :- neighbour(X,Z), one_side_fixed(Z), usual(Z), usual(X), not_loaded(X), free(X).
 - 16 mesh(X,Y) :- neighbour(X,Z), one_side_fixed(Z), short(Z), usual(X), fixed(X), not_loaded(X), not_loaded(Z),
neighbour(Z,W), long_for_hole(W), free(W), not_loaded(W).
-

Figure 8.3. Learned rules for mesh-3 with xFOIL-DIC.

Bold rules are also learned with xFOIL. Clichés are shown in boxes. Underlined literals are learned after adding the cliché. The ~~striethrough~~ clause violates the specificity of a new edge (C2) and is eliminated before testing.

Clichés are used in more than half of the rules. Instantiations from two of the four DICs are used. Instantiation `neighbour(X,Z)`, `fixed(X)`, `one_side_loaded(Z)` in rule 13 comes from DIC-3: `ExtP1(X, Y)`, `ExtP2(X)`, `IntP3(Y)`. Other instantiations come from the DIC-2: `ExtP1(X, Y)`, `ExtP2(Y)`, `ExtP3(Y)` (e.g. `neighbour(X,Z)`, `free(Z)`, `long(Z)` in rule 2). In rules 13 and 16, the DIC instantiation added to the rule allows xFOIL-DIC to learn more about the edge introduced by the cliché. Rule 13 describes two edges (x and z), where the cliché introduces z. Rule 16 describes three edges (x, z, and w), where the cliché introduces z and `neighbour(Z, w)` introduces w. Rule 4 is removed before testing, because it violates the condition for the *specificity of a new edge* (C2), i.e. `opposite(Z, w)` introduces the edge w, but no literals describe that edge (Section 8.3). xFOIL-DIC learns only three rules (7, 9 and 12) that are also learned with xFOIL. Moreover, compared to xFOIL's learned rules (Figure 8.2), xFOIL-DIC learns more rules and these rules are described with more literals (i.e. they are more specific).

Table 8.12 shows statistics for xFOIL-DIC learning FEM concepts. Compared to xFOIL (Table 8.3), xFOIL-DIC learns more specific rules. The number of literals per rule increases from 3 to 5 literals. The number of training negative examples covered by rules drops from 47% to 15%. To compensate for learning more specific rules xFOIL-DIC learns in average 20 rules instead of 13. Again, learning more specific rules decreases the coverage of testing examples. The coverage of positive examples decreases from 82% to 65% and it decreases from 29% to 11% for the negatives. Overall the error rate decreases by 50% (from 26% to 13%).

Mesh	Learning				Testing		
	Neg.left%	Rules	Lit/rule	#Cli	PC%	NC%	Err%
1	11	31	5	23	76	12	16
2	13	38	6	23	67	23	25
3	16	16	6	22	68	14	16
4	19	12	5	21	81	9	10
5	17	9	6	16	42	5	7
6	15	11	5	13	57	4	7
Avg.	15	20	5	20	65	11	13

Table 8.12. Statistical results for xFOIL-DIC.

The number of negative examples covered by learned rules, the number of rules, the number of literals per rule, and the testing coverage of xFOIL-DIC for FEM concepts. The number of literals per cliché is constant (three) and is not shown here. The number of duplicate literals is zero and is not shown here.

Table 8.13 shows that xFOIL-DIC performs significantly better than xFOIL on testing examples for 50% of the FEM concepts. Using DICs improves xFOIL's accuracy, since it decreases the number of misclassified examples for each FEM concept. This is shown by positive differences in error between xFOIL and xFOIL-DIC. The data provide enough evidence to say that xFOIL-DIC performs significantly better than xFOIL on testing sets for concept mesh-1, mesh-3, and mesh-4 (shaded values in Table 8.13).

Mesh	Difference in error
1	0.17 ± 0.12
2	0.15 ± 0.18
3	0.12 ± 0.07
4	0.10 ± 0.02
5	0.09 ± 0.17
6	0.11 ± 0.15

Table 8.13. The difference in error $\pm$ the standard deviation for each FEM concept. **Shaded** intervals represent significant improvements in favour of xFOIL-DIC.

The improvement provided by DICs to xFOIL has a cost. This cost is evaluated in terms of search steps that directly influence the CPU time. Table 8.14 shows the number of search steps and the corresponding CPU time needed for xFOIL and xFOIL-DIC.

Mesh	xFOIL		xFOIL-DIC		
	Steps	literals	CPU	Steps	literals
1	138	3	163	2361	116
2	216	11	225	2032	146
3	184	2	222	2593	71
4	189	2	201	2285	45
5	169	1	201	2156	36
6	171	2	188	1835	19
Average	178	4	200	2210	72

Table 8.14. The cost of xFOIL and xFOIL-DIC for each concept. xFOIL searches only for literals whereas xFOIL-DIC searches for literals and instantiations of DICs.

xFOIL searches only for literals, while xFOIL-DIC searches for literals and instantiations of DICs. Moreover, after a DIC's instantiation is added to a rule, xFOIL-DIC continues to search for literals to describe the new edge introduced by the DIC. The number of search steps for DICs is related to the number of instantiations of DICs. So the number of search steps to find a literal increases by 13% (from 178 to 200 steps). Taking into

account the number of search steps to find a DIC as well, xFOIL-DIC takes almost fourteen times more search steps than xFOIL (2410 steps for xFOIL-DDC compared to 178 for xFOIL). As a consequence, the CPU time increases from 4 minutes to 72 when DICs are used.

8.3.6 xFOIL-DIC-10 with the most frequently used literals

One of the reasons that using DICs is so expensive in the FEM domain is that there is a large number of attributes that describe an edge used to instantiate variable literals of DICs. One way to reduce the size of the search space is to reduce that number of attributes. The next experiment reduces the number of literals to instantiate predicate variables of a DIC to ten (referred to as xFOIL-DIC-10), instead of using 19 attributes as in the previous xFOIL-DIC experiment

The ten attributes chosen are the most frequently used attributes to describe an edge in the FEM domain. The branching factor for xFOIL-DIC-10 is shown in Table 8.15 along with statistics of using xFOIL-DIC-10 instead of xFOIL-DIC.

DICs	Nb. instantiations	Reduced	Nb. variabilizations
1	20	20	120
2	200	200	1200
3	200	200	1200
4	2000	1024	6144
Total	2420	1444	8664

*Table 8.15. The branching factor for clichés with xFOIL-DIC-10.
The number of instantiations for attributes before and after reduction (Section 7.2.3), and the number of variabilizations after reduction*

The search space for xFOIL-DIC-10 contains the search space of variabilized attributes and the search space for variabilized DICs. Table 8.15 shows the number of variabilized DICs for xFOIL-DIC-10. There are four DICs all described with variable predicates. There are again two relations but this time there are only ten attributes to describe an edge instead of nineteen. For example, the fourth DIC has a relation with one attribute describing the argument x , and two literals describing the argument y . So there are 2000 instantiations ($2 * 10^3$). After the reduction procedure (Section 7.2.3) the number of instantiations becomes 1024 ($2 * 512$). This is the same number as in Table 8.15 (because

of the reduction procedure), but the instantiations now contain the most frequently used literals only.

The total number of instantiations for DICs with xFOIL-DIC-10 is only 8664 compared to 15036 for xFOIL-DIC.

xFOIL-DIC-10 learns the same number of rules (Table 8.16) as xFOIL-DIC (Table 8.12). Although xFOIL-DIC-10 learns an additional literal per rule on average, learned rules cover more positive and negative examples. xFOIL-DIC-10 covers 70% of the positive examples and 21% of the negatives, compared to 66% for positives and 9% for negatives with xFOIL-DIC. xFOIL-DIC-10 has a much lower cost of learning: the search steps for literals and DICs decreases by 37% compared to xFOIL-DIC, and the CPU time decreases by almost 50%.

Mesh	Learning							Testing		
	Neg.left%	Rules	Lit/rule	#Cli	PC%	NC%	Err%	Stp.lit	Stp.cli	CPU
1	13	29	5	19	89	16	14	159	1152	54
2	16	37	6	21	67	24	26	232	1000	85
3	19	15	6	21	63	17	19	225	1250	35
4	22	12	5	21	81	9	10	201	1116	24
5	19	9	6	15	47	6	8	207	1045	18
6	17	11	4	12	58	7	9	175	908	11
Avg.	18	19	5	18	68	13	14	200	1079	38

Table 8.16. Statistical results for xFOIL-DIC-10.

xFOIL-DIC-10 also improves on xFOIL's accuracy, since it decreases the number of misclassified examples for each FEM concept. This is shown with positive differences in error between xFOIL and xFOIL-DIC-10 (Table 8.17). Two thirds of the FEM concepts (shaded intervals) provide enough evidence to say that the decrease in error from xFOIL to xFOIL-DIC-10 is significant. In sum xFOIL-DIC-10 is less expensive than xFOIL-DIC, decreasing the branching factor of DICs by more than 50%. It also makes a significant improvement for one additional FEM concept⁷².

⁷² Notice that there is no significant improvement between results from xFOIL-DIC-10 and xFOIL-DIC.

Mesh	Difference in error
1	0.19 ± 0.11
2	0.14 ± 0.11
3	0.09 ± 0.05
4	0.10 ± 0.02
5	0.09 ± 0.15
6	0.09 ± 0.11

Table 8.17. The difference in error between xFOIL-DIC-10 and xFOIL ± the standard deviation. Shaded intervals represent significant improvement in favour of using DICs-10.

8.4 Empirical results in the *blocks* domain

This section describes experiments similar to the FEM experiments carried out for two concepts in the *blocks* domain. Each concept is learned with xFOIL, xFOIL-DDC and xFOIL-DIC. Two additional experiments are done with each concept: xFOIL-DDC and xFOIL-DIC in the presence of BK. A 3-fold cross-validation is applied and error rates are compared using the procedure described in Section 8.2.1. The two concepts are B1 and B2 which were described in Chapter 3 and used for examples in chapter 7. Briefly, instances of the concept in B1 start with a *circle* above a *rectangle* (but not a *square*) whereas instances of the concept B2 start with a *circle* above a *rectangle* or a *square*. Everything else is the same for the two concepts. DDCs are learned from the *blocks* concept B3 (also described in Chapter 3). For simplicity, DICs are also learned from concept B3 and used as if they are provided from another domain. Clichés learned for concept B3, as well as the definitions learned for each experiment are introduced in Chapter 7, and are not shown here.

As shown in Chapter 7, xFOIL is unable to learn any rules for concepts B1 and B2. This is because the literal to learn is `block(X)` which describes only one object (the variable `x`), and the object `cir(X)` appears in all examples as the first literal. The literal `above/2` (which relates the second object to the second one) is the only literal that could introduce a new variable in the rule, but it also appears in all examples. Consequently, xFOIL cannot find any literal that discriminates positive and negative instances.

Clichés are useful for learning concepts B1 and B2. This is true either in the presence or in the absence of the BK, and for DDCs or DICs. Table 8.18 shows some statistical results of learning concepts B1 and B2 with DDCs and DICs, with or without BK.

		Rules	Lit/rule	Err%	CPU	Steps literals	Steps clichés	Given clichés
Learning B1	xFOIL	0	0	NA	0	17	0	0
	xFOIL-DDC	2	4	18	0	17	2	17
	xFOIL-DDC & BK	2	4	18	0	17	18	17
	xFOIL-DIC	2	4	18	5	16	612	9
	xFOIL-DIC & BK	2	4	18	5	16	612	9
Learning B2	xFOIL	0	0	NA	0	17	0	0
	xFOIL-DDC	2	4	36	0	17	2	17
	xFOIL-DDC & BK	2	3	31	0	9	18	17
	xFOIL-DIC	3	3	31	6	14	612	9
	xFOIL-DIC & BK	6	4	31	7	14	612	9

Table 8.18. Statistical results on learning concepts B1 and B2 with clichés learned from concept B3. NA stands for not applicable⁷³.

Almost the same results are obtained when xFOIL-DDC learns B1 in the absence or in the presence of the BK. The knowledge provided with the BK (*e.g.* a *rectangle* is also a *square*) is too general and is not used. In this case, adding BK merely adds more instantiations of DDCs to search. Similar results are obtained when xFOIL-DIC learns B1. On the other hand, the BK is useful for learning the concept B2 though without the BK, xFOIL-DDC still learns where xFOIL cannot. xFOIL-DDC learns a definition with a better accuracy in the presence of BK. Since all instantiations of a cliché are added as disjunctive rules, xFOIL-DDC stops searching, hence the smaller number of steps to search for literals in the presence of the BK. Definitions learned with xFOIL-DIC have the same accuracy but are different rules. In this case, the presence of knowledge allows xFOIL-DIC to learn rules describing the first object as a *circle* (as in the positive examples), but also as an *ellipse* since no negative instances contradict that possibility. The six rules are learned from instantiations of two DICs in the presence of BK, whereas in the absence of the BK, each rule corresponds to one instantiation of a DIC.

⁷³ To a null program could be assigned the default error rate of the number of positive examples on the total number of examples (positives and negatives).

8.5 Conclusion

Empirical results show that clichés are useful for learning concepts across domains. Clichés are learned in every definition of a FEM concept and they increase the accuracy of learned rules 47% of the time. Experiments show that DDCs learned from a FEM concept make a significant improvement to learning other concepts in that domain half of the time. Similarly, DICs learned from a *blocks* concept are useful for learning at least half of the FEM concepts.

Rules learned by xFOIL-CLICHÉS are more specific than those learned by xFOIL. Rules are expressed with more literals and they cover fewer examples each. Hence xFOIL-CLICHÉS learns more rules to cover all positive examples. Learning with clichés improves the accuracy of rules, and only introduces rules that increase the overall accuracy.

DICs are more flexible than DDCs, but are more expensive to use. Because of this, the search space needs to be restricted in some way. The last experiment in the FEM domain showed that the search space restricted with the most frequently used literals of the FEM concepts increase the improvement in using xFOIL-DIC over xFOIL on two thirds of the FEM concepts. Other similar experiments should be done to find the best way to prune that search space.

This chapter did not address the issue of noisy data, but FEM concepts are known to be noisy. It is difficult to identify what kinds of noise (misclassification of examples, wrong description of examples, etc.) exist and in what proportion (in only few concepts or for every one). I performed some experiments to attempt to identify sources of noise, but observed no general tendency for all FEM concepts. One experiment preserved only rules with at least 75% training accuracy. Another experiment cut off the search for a literal (or a cliché) if a rule covered few (say 3) negative examples. More experiments of this kind could help group FEM concepts with similar behaviour and to characterize groups of concepts for which xFOIL-CLICHÉS succeed.

Experiments with *blocks* concepts show that for concept for which xFOIL cannot learn a definition, clichés provide appropriate lookahead 100% of the time.

9 Other domains of application

It is important that theory developments in ILP be validated by practical applications. In this thesis, applications of learning clichés with CLUSE and using them to learn concepts within and across domains were demonstrated using the *blocks* and the FEM domains. CLUSE (and/or xFOIL-CLICHÉS) might also be used in domains of application where other ILP learners have shown some potential. Domains of application selected for discussion here include drug design, detecting traffic problems, and text categorization. Similar to the FEM domain, these domains are characterized by the need of a relational representation that is highly non-determinate, and involve a large number of facts. In each domain of application, CLUSE could be used to provide clichés to another learner or to xFOIL-CLICHÉS as used in the FEM domain.

9.1 Drug design – Mutagenesis discovery

CLUSE could provide automated assistance in the process of scientific discovery in the biological domain. One of the problems in that domain is discovering rules for mutagenicity in nitroaromatic compounds (Srivasan *et al.* 1994). These compounds occur in automobile exhaust fumes and are also common intermediates in the synthesis of many thousands of industrial compounds (Debnath *et al.* 1991). Highly mutagenic nitroaromatics have been found to be carcinogenic, and often cause damage to DNA. It is of considerable interest in pharmaceutical industry to determine which molecular features result in compounds having mutagenic activity (Finn *et al.* 1998; Lee *et al.* 1998).

For an ILP learner, the task is to obtain structural descriptions that discriminate drugs with positive mutagenicity in small molecules from those which have zero or negative mutagenicity. The most primitive but still practical structural representation of molecules is in terms of the atomic and bonding properties of the molecules. At this level, feature-based algorithms are inapplicable, as it is usually impossible to know all relevant sub-structures for all molecules. Since a single molecule usually has several atoms, each of which can be associated in more than one bond, a relational learner without determinacy restriction is required. Bonding and atomic features are represented as follows.

`bound(compound, atom1, atom2, bondtype)`, stating that compound has a bond of bondtype between the atoms atom1 and atom2. For example, an aromatic bond between atoms d2_1 and d2_2 in drug d2 is represented⁷⁴ as `bond(d2, d2_1, d2_2, 7)`.

`atom(compound, atom, element, atomtype, charge)`, stating that in compound, atom has element element of atomtype and partial charge charge. For example, the fact that atom d2_1 in drug d2 is an aromatic carbon atom with partial charge 0.067 is represented by the fact `atom(d2, d2_1, c, 22, 0.067)`.

⁷⁴ These are representations supplied by a standard molecular graphics package (QUANTA).

Srivasan *et al.* (1994) used the ILP learner PROGOL (Muggleton 1995) to discover rules for mutagenic activity. Only a large number of facts (230 bond-and-atom molecular descriptions) that are largely non-determinate were used with no additional background knowledge. The experiments with PROGOL showed that:

- 1) PROGOL can discover concepts that are comprehensible and can usually assist statistical models devised by experts in the field.
- 2) In cases where statistical models fail, PROGOL alone can still work effectively as a theory constructor.

The authors claim that PROGOL's performance is largely due to its strategy of generalization, as opposed to a strategy of specialization (*c.f.* FOIL).

CLUSE could also be a potential assistant for mutagenesis discovery. As in PROGOL, CLUSE learns in the absence of background knowledge and from a large number of facts that are highly non-determinate (*c.f.* FEM domain, Chapter 3). Furthermore, as in PROGOL, CLUSE uses a strategy of generalization (although CLUSE is based on CLGG instead of inverse entailment).

9.2 Text categorization

CLUSE could be applied to text categorization (or classification), for use in e-mail filtering, news filtering, and automatic indexing of documents. Text categorization is the classification of textual documents into one or several predefined categories. Because categories are not mutually exclusive, binary classifiers are learned for each possible category, rather than formulating the problem as a single multi-class learning problem.

In text categorization, relational learners are more appropriate than attribute-value learners, because examples only use a subset of possible features and also formulate important properties such as word order in a document. Many properties of texts in natural language are conveniently expressed by relations (*e.g.* that a direct object in a

sentence is related to the verb in the same sentence). Cohen (1995) has suggested one possible representation of texts conducive to text categorization using relations. A word is represented by the literal $w_i(d, p)$ where word w_i occurs in the document d , and p is the position at which the word occurs. A word is related to the word at the next position with a relation *successor*, it is related to words at the previous and following three positions with the relation *nearN* (where $N \leq 3$). Any learning system based on an entropy measure would have difficulty learning these relations, since the complexity of a text makes it possible that words will have more than two relations *near* and *after*. Experiments done by Cohen (1995) confirm this.

Cohen (1995) evaluates FOIL on a series of text categorization problems. The results of the first experiment show that using a relational learner rather than a propositional one can lead to significantly better performance on real-world text categorization problems. This experiment also shows that relations are used only occasionally by FOIL, because they occur in both positive and negative examples. The second experiment investigated several *relation selection* methods (*i.e.* a first-order analog to feature selection) methods. The results of this experiment show that FOIL's performance can be dramatically improved by such methods. These last two experiments (Cohen 1995) and results are very similar to my experiments of comparing learning FEM concepts with xFOIL to xFOIL-CLICHÉS. Because of this and because the domain of text categorization shows similarity with the FEM domain, CLUSE (and/or xFOIL-CLICHÉS) could be applied to learn and use clichés in the text categorization domain.

9.3 Detecting traffic problems

ILP is well suited to problems in which relations between objects play an important role. One such real-life application is of detecting traffic problems. Dzeroski *et al.* (1998) explore the possibility of using inductive learning techniques (such as ILP) to generate knowledge on traffic problem detection from historical data that contains parameters recorded by sensors. The knowledge for incident (traffic problem) detection has been

formulated by domain experts in the first-order representation. Therefore, ILP is a suitable tool for learning to detect traffic problems in this context.

Two kinds of input are available to the learning process. The first type is the background knowledge on the road network to capture different types of road sections, the relations among them, and the placement of sensors on individual road sections. The second type is sensor readings in three basics quantities describing traffic behaviour: *speed* (kilometers/hour), *saturation* (vehicles/hour) and *occupancy* (percentage of time that the sensor is occupied by vehicles). Each section at a particular moment of time corresponds to an example, classified into one of the three classes: an accident critical section, a congestion critical section or a non critical section. Relations that allow access to the *previous* or *next* sections of a given section are part of the background knowledge. The complexity of the road makes it possible for a section to have more than two *previous* or *next* sections. The background knowledge also includes predicates that calculate the speed, the saturation and the occupancy between sections. So, whether the learning task is to learn a rule for each of the three classes mentioned above or to distinguish between non-critical and any type of critical road sections, all examples are described with relations for *previous* and *next* sections. This is similar to the relation *neighbour* in the FEM domain. Any learning system based on an entropy measure will have difficulty learning these relations.

10 Future work

This chapter suggests some extensions and alternate uses for both CLUSE and xFOIL-CLICHÉS. Future work on CLUSE can be divided into extensions to the existing implementation and conceptual modifications of the framework in which clichés are learned and used.

10.1 Extensions and improvements to the existing implementation of CLUSE

Efficiency

When I implemented CLUSE, I expected to work with concepts represented by few hundred chains. The efficiency of the program was not a priority. It turns out that in a real-life domain like the FEM domain, there were several thousand chains to learn clichés

from. Simple modifications such as asserting facts instead of passing very long lists of data as parameters would make a difference in the CPU time.

Parts of the programs would also be more efficient if they were implemented in another language such as C.

Numerical values

By default, numerical value arguments are treated as constants and therefore are generalized into variables. They could also be treated as nominal values. In that case, the user would need to classify them under a class name (Section 5.3), which would be used as the generalization symbol instead of variables. If the number of numerical values is large, treating them as nominals becomes impractical. Therefore, other ways to handle numerical values should be explored. For example, an averaging technique could be used; or instances themselves could be used to generate a probabilistic summary. Another solution would be to discretize numeric values beforehand, making the induction process more efficient (Blockeel *et al.* 1997). Blockeel's discretize algorithm finds a threshold that partitions a set of examples into two subsets such that the average class entropy of the subsets is as small as possible. This procedure is applied recursively on the two partitions until some stopping criterion is reached.

Generalization and pruning

All generalizations are inserted in the hierarchy of generalizations and then CLUSE prunes the hierarchy according to the generalization coverage frequencies (Section 6.2). During the generalization process, generalizations that do not satisfy the user-supplied coverage threshold ($PCov$) should be removed as soon as they are generalized. This would reduce the size of the hierarchy. During the pruning phase, generalizations that cover fewer positive than negative chains should also be removed.

Coverage in terms of examples instead of chains

Some comparisons should be done between generalizations with coverage frequencies in terms of examples instead of coverage frequencies in terms of chains. Generalizations are pruned according to their coverage frequencies. These coverage frequencies are

computed in terms of chains. I initially thought that computing coverage frequencies in terms of chains would be better at handling noise. Noisy chains would get very low coverage frequencies and be pruned. This may be less desirable if there is class noise. To handle class noise, a generalization that covers more examples should get more credit than a generalization that covers more chains (and not necessarily more examples).

Coverage frequencies for DICs as well as DDCs

Currently, coverage frequencies are only computed for DDCs. Coverage frequencies could also be added to DICs. These frequencies represent the generalization's coverage of the concept chains when it is learned. When clichés are used, these values can be used to select clichés that satisfy user-supplied thresholds (Section 6.2), hence avoiding (at least part of) the utility problem. The utility problem occurs when increasing knowledge decreases the learner's efficiency rather than improving it, often because the cost of matching learned knowledge more than offsets the savings in search. One way to add coverage frequencies to a DIC would be to use the coverage frequencies of the most general DDC (*i.e.* the one with the highest coverage frequencies) from which the DIC was generalized.

Noisy data

A thorough evaluation of CLUSE's tolerance to noise should be performed. This evaluation should compare the performance of CLUSE when learning from "perfect" examples of a concept to its performance when learning from examples of the same concept with noise. Examples with different kinds of noise as well as different levels of noise could be generated easily using GENEX.

Recursivity

CLUSE could be extended to learn recursive rules. One way to explore this extension of CLUSE would be as follows. Prior to the generalization process (Section 6.2) CLUSE could generalize chains within a single example first and then generalize chains from different examples.

10.2 Framework modifications

10.2.1 Using CLUSE to create library of clichés

CLUSE could be used as is to create a library of clichés. For each domain of application (Chapter 3) and for each concept clichés (DDCs and DICs) could be stored in a library made publicly available (on the web for example). Coverage frequencies could be used to avoid (at least part of) the utility problem. For instance, systems could use only clichés that satisfy coverage thresholds. Moreover, clichés learned for a specific concept or clichés learned for all concepts in one domain could be used. The task of choosing the appropriate library would be done manually by one who understand the domains.

10.2.2 Using CLUSE to create a library of concept hierarchies

CLUSE learns concept hierarchies

CLUSE could be used to build concept hierarchies, as described in Langley (Langley 1996).

CLUSE shares the same learning task as the formation of concept hierarchies, which is:

Given: A set of training instances and (possibly) associated classes.

Find: A concept hierarchy that makes accurate predictions about novel test instances. As with other induction tasks, this one assumes that the acquired knowledge structure will perform well on unseen instances, even at the expense of imperfect performance on the training set.

CLUSE's hierarchies of generalizations are in fact concept hierarchies. A *concept hierarchy* is composed of nodes and links. Each node represents a separate intensionally defined⁷⁵ concept. The links connecting a node to its children specify a subset relation. Typically, a node covers all of the instances covered by the union of its descendants, making the concept hierarchy a subgraph of the partial ordering by generality (Section 5.1.2).

⁷⁵ The user could also be asked to name the concept.

CLUSE constructs the hierarchy in an *agglomerative* manner (from terminal nodes upward)⁷⁶. The construction of the hierarchy of generalizations groups instances into successively larger clusters rather refining them (*c.f.* the divisive scheme). CLUSE follows the general algorithm of agglomerative formation of concept hierarchies described in (Langley 1996). CLUSE preserves only generalizations that pass coverage frequencies criteria, and returns these generalizations as DDCs. Instead of returning DDCs as flat lists (Section 6.2), DDCs (and DICs) would be returned preserving the (concept) hierarchy organization.

Concept hierarchies provide a better memory organization than flat lists of clichés and allow some pruning, giving a solution to the utility problem (Langley 1996). Classifying new instances with a concept hierarchy involves moving downward through the hierarchy. At each level, instantiate the cliché or use coverage frequencies on the alternative nodes to select one to expand, then recurse to the next level.

CLUSE learns second-order concept hierarchies

CLUSE could also be used to learn second-order concept hierarchies. DICs are second-order generalizations of DDCs. An easy way to build second-order concept hierarchies would be to preserve the hierarchy (or generality relation) of DDCs when generalizing them into DICs.

CLUSE creates concept lattices

CLUSE could be used to provide concept hierarchies for each concept in different domains, instead of providing simple lists of clichés. Concept hierarchies also provide better memory organization than flat lists and would help alleviate the utility problem.

More than one hierarchy could be learned for each concept, and concept hierarchies in the same domain could be combined to form a lattice.

- 1) Different hierarchies could be learned for the same concept. A hierarchy can be learned given user-supplied preferences on which predicates to choose to describe a concept. The user-supplied preferences are given through predicate weights (Section

⁷⁶ This is different than the specific-to-general method to search the concept description (Section 6.2).

- 5.2). More than one hierarchy can be learned this way. For instance, one concept hierarchy could give priority to the *shape* of objects, whereas another one would give the priority to their *size*. Hierarchies for the same concept could then be combined into a single hierarchy (or lattice).
- 2) Concept hierarchies for all concepts in one domain could be combined in a lattice. This lattice could be used as a concept hierarchy with multiple top concept descriptions.

10.3 Other strategies for xFOIL-CLICHÉS

Other strategies of learning clichés with xFOIL should be explored. One of the experiments in the FEM domain showed that DDCs learned from one concept and used to learn the same concept make a significant improvement for only half of the concepts (Section 8.3). A closer look at these results revealed that adding a cliché would make the rule too specific to have a gain (since clichés are learned after literals are added to the rules). These experiments suggest that other strategies should be tried with xFOIL-CLICHÉS. Since it would be too expensive to search for clichés or literals at every step in induction, xFOIL-CLICHÉS could start a rule by searching for a literal or a cliché and then look only for literals until it covers no negative examples or it cannot find a single literal. At this point, if there is not already a cliché in the rule, it could search for one.

10.4 CLUSE combined with other learners

CLUSE could be used as a module with other learners that already work with similar biases (and possibly with intensionally defined BK) such as CLINT (De Raedt *et al.* 1992a), MOBAL (Kietz *et al.* 1992) or FOCL (Silverstein *et al.* 1993).

11 Conclusion

This thesis is devoted to the problem of learning relational clichés. To address the problem, I have considered the following questions.

- How can clichés be learned?
- What kind of generalization (more gradual than *lgg*) is necessary to learn clichés?
- Once learned, what kind of lookahead do clichés offer:
 - a) within the same domain?
 - b) across domains?
- How does the distance between domains affect the utility of clichés?

CLUSE learns clichés as a combination of literals that escape the local plateau problem of a hill-climbing learner.

To answer these questions, I have implemented CLUSE. CLUSE learns clichés in a bottom-up manner from examples of a concept. Examples are expressed in terms of chains and are generalized into a hierarchy of generalizations using CLGG. CLUSE then prunes this hierarchy according to generalization coverage frequencies of chains. Preserved generalizations are expressed with predicates specific to the domain and are returned as DDCs. These DDCs are further generalized with variable predicates into DICs.

CLGG (Chapter 5) overcomes the overgeneralization and inefficiency problems of LGG/RLGG (Plotkin 1970). CLGG is used in CLUSE to generalize two clauses (or chains). It exploits additional background knowledge if available during generalization (taxonomical hierarchies) and the similarity of literals in the context of the clauses being generalized.

Different evaluations of CLUSE were presented in (Chapter 6).

- 1) CLUSE parameters allow the user to control some aspects of system performance. The sensitivity analysis evaluated how much control these parameters allow during the generalization process and during the pruning process (Section 6.5). The user can stop the generalization process before the resulting generalizations become too general. During the pruning process the user can also cut out the most specific generalizations. The user can also provide BK to generalize literals that would not be generalized otherwise.
- 2) The complexity analysis evaluated the cost of learning clichés with CLUSE (Section 6.3). The complexity of CLUSE learning clichés is in the worst case $O(mn^3)$, where m is the number of instances and n the number of positive chains. The cost of learning clichés is polynomial.
- 3) GENEX (Chapter 4) was used to evaluate the similarities between rule templates given to GENEX and clichés (or DDCs) learned with CLUSE. DDCs learned from

generated examples were compared to rule templates given to GENEX to generate these examples. This evaluation revealed that all syntactic restrictions underlying examples were accounted for in the hierarchy of generalizations when enough examples were provided to CLUSE.

- 4) CLUSE was applied to both synthetic and real-life domains. Applications of CLUSE showed that it learns clichés in the *blocks* and the FEM domains (Section 6.4).

GENEX (Chapter 4) also proved to be a useful tool in generating examples of a wide variety of concepts in the *blocks* domain.

Empirical evaluation revealed that clichés learned with CLUSE provide appropriate lookahead to escape local plateaus of a hill-climbing learner both within and across domains. xFOIL-CLICHÉS was used with a cross-validation method to compare hypotheses induced with clichés to hypotheses induced without clichés. DDCs expressed with first-order predicates are used to learn concepts in the same domain in which they are learned. DICs expressed with second-order predicates are used to learn concepts across domains. DICs proved to be more flexible than DDCs, but are more expensive to apply. DDCs provide a combination of literals with specific predicate names and an implicit pattern of variables (showing which variables are shared by literals). On the other hand, DICs only provide a pattern of variables. Experiments showed that clichés often significantly improve the accuracy of the hypotheses and in the worst case, the accuracy is never worse than if no cliché were provided. In the *blocks* domain, experiments showed that with concepts for which xFOIL learns no hypotheses, clichés always provide an appropriate lookahead. In the FEM domain, DDCs learned from a FEM concept were useful (*i.e.* make a significant improvement) to learn other concepts in that domain 47% of the time. Similarly, DICs learned from a *blocks* concept were useful to learn 50% of the FEM concepts. Since the size of the search space for clichés (especially for DICs) can easily become intractable, a procedure to restrict the number of instantiations was introduced (Section 7.2.2). This first experiment used an arbitrary ordering of literals (*i.e.* the same ordering of literals as was given to the system). The second experiment showed

that when the search space for DICs was restricted to the most frequently used literals in that domain, DICs were useful for 66% of the FEM concepts.

The results indicate that ILP learners could benefit from the use of learned clichés. Other domains of application such as drug design, text categorization, and detecting traffic problems were discussed in Chapter 9. Clichés would be learned from the domain at hand (DDCs), or clichés for a “similar” domain could be fetched from the library (Chapter 10). How to define and measure “similarity” of domains is an open problem. In general, CLUSE-like system could be used whenever an ILP learner fails to give practical results.

12 References

AHA, D. W. (1992). "Relating Relational Learning Algorithms." *Inductive Logic Programming*, Muggleton S. (ed.), Academic Press, 233-253.

AHA, D., S. LAPOINTE, C. X. LING AND S. MATWIN (1994). "Learning Recursive Relations with Randomly Selected Small Training Sets." *Proceedings of the Eleventh International Conference on Machine Learning (ICML'94)*, Rutgers University, New Brunswick, N. J., Morgan Kaufmann, 12-18.

BERGADANO, F. AND A. GIORDANA (1988). "A knowledge intensive approach to concept induction." *Proceedings of the Fifth International Conference on Machine Learning (ICML'88)*, San Mateo, CA, Morgan Kaufmann, 305-317.

BERGADANO, F. AND A. GIORDANA (1990). "Guiding induction with domain theories." *Machine Learning: An Artificial Intelligence Approach, volume III*, Kodratoff, Y and R. S. Michalski (eds.), San Mateo, CA, Morgan Kaufmann, 474-492.

BERGADANO, F. AND D. GUNETTI (1995). *Inductive Logic Programming: from Machine Learning to Software Engineering*, London, MIT Press.

BISSON, G. (1990). "KBG: A Knowledge Based Generalizer." *Proceedings of the Seventh International Conference on Machine Learning*, Austin, Texas, Morgan Kaufmann, 9-15.

- BISSON, G. (1992). "Learning in FOL with a Similarity Measure." *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose, CA, Morgan Kaufmann, 82-87.
- BLOCKEEL, H. AND L. DE RAEDT (1997). "Lookahead and discretization in ILP." *Proceedings of the Seventh International Workshop on Inductive Logic Programming (ILP-97)*, Prague, Czech Republic, Springer, 77-84.
- BONGARD, M. (1970). *Pattern Recognition*. Spartan Books.
- BRAZDIL, P. AND A. JORGE (1994). "Learning by Refining Algorithm Sketches." *Proceedings of the European Conference on Artificial Intelligence*, Amsterdam, The Netherlands, John Wiley and Sons, 443-447.
- BRUNK, C. AND M. PAZZANI (1991). "An investigation of noise-tolerant relational concept learning algorithms." *Proceedings of the Eighth International Workshop on Machine Learning (ICML'91)*, San Mateo, CA, Morgan Kaufmann, 389-393.
- BUNTINE, W. (1988). "Generalized Subsumption and its Applications to Induction and Redundancy." *Artificial Intelligence* 2(36):149-176.
- CAMERON-JONES, R. M. AND J. R. QUINLAN (1993). "Avoiding Pitfalls When Learning Recursive Theories." *Proceedings Thirteen International Joint Conference on Artificial Intelligence*, Chambéry, France, 1050-1055.
- CHANG, C. AND R. Lee (1987). *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York.
- CLARK, P. AND S. MATWIN (1993). "Using Qualitative Models to Guide Inductive Learning." *Proceedings of the Tenth International Conference on Machine Learning*, Amherst, Massachusetts, Morgan Kaufmann, 49-56.
- COHEN, W. W. (1993a). "Rapid Prototyping of ILP Systems Using Explicit Bias." *Proceedings of the IJCAI Workshop on Inductive Logic Programming (ILP-93)*, Chambéry, France, Morgan Kaufmann, 24-35.
- COHEN, W. W. (1993b). "Pac-learning a restricted class of recursive logic programs." *Proceedings of the Third International Conference on Artificial Intelligence*, Washington, DC, AAAI Press, 86-92.
- COHEN, W. W. (1995). "Text Categorization and relational Learning." *Proceedings of the Twelfth International Conference on Machine Learning (ICML'95)*, Tahoe City, CA, Morgan Kaufmann, 124-132.
- COHEN, WILLIAM W. AND P. DEVANHU (1997). "A comparative Study of Inductive Logic Programming Methods for Software Fault Prediction." *Proceedings of the Fourteen International Conference on Machine Learning (ICML'97)*, Nashville, Tennessee, Morgan Kaufmann, 66-75.

- DE RAEDT, L. AND M. BRUYNOOGHE (1989a). "Constructive Induction by Analogy." *Proceedings of the Sixth International Conference on Machine Learning*, Ithaca, New York, Morgan Kaufmann, 476-477.
- DE RAEDT, L. AND M. BRUYNOOGHE (1989b). "Constructive Induction by Analogy: a Method to learn how to learn." *Proceedings of the Fourth European Workshop Session on Learning*, Montpellier, Pitman, 189-199.
- DE RAEDT, L. AND M. BRUYNOOGHE (1989c). "Towards Friendly Concept-Learners." *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, Michigan, Morgan Kaufmann, 849-854.
- DE RAEDT, L., M. BRUYNOOGHE AND B. MARTENS (1991). "Integrity Constraints and Interactive Concept-Learning." *Proceedings of the Eighth International Conference on Machine Learning*, Evanston, Illinois, Morgan Kaufmann, 394-398.
- DE RAEDT, L. AND M. BRUYNOOGHE (1992a). "An Overview of the Interactive Concept-Learner and Theory Revisor CLINT." *Inductive Logic Programming*, Muggleton S. (ed.), Academic Press, 163-191.
- DE RAEDT, L. AND M. BRUYNOOGHE (1992b). "Interactive Concept-Learning and Constructive Induction by Analogy." *Machine Learning*, 8(2):107-150.
- DEBNATH, A. K., R. L. LOPEZ DE CAMPADRE, G. DEBNATH, A. J. SHUSTERMAN, AND C. HANSCH (1991). "Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds correlation with molecular orbital energies and hydrophobicity." *Journal of Medicinal Chemistry*, 34(2):786-797.
- DOLSAK, B. AND S. MUGGLETON (1992). "The Application of Inductive Logic Programming to Finite-Element Mesh Design." *Inductive Logic Programming*, Muggleton S. (ed.), Academic Press, 453-472.
- DOLSAK, B., I. BRATKO AND A. JEZERNIK (1994). "Finite Element Mesh Design: An Application Domain for ILP Application." *Proceedings of the Fourth International Workshop on Inductive Learning Programming (ILP-94)*, Bonn, Germany, 305-321.
- DZEROSKI, S. AND B. DOLSAK (1991). "A Comparison of relation learning algorithms on the problem of finite element mesh design." *Proceedings XXVI Yugoslav Conference of Society for ETAN*, Ohrid, Yugoslavia, In Slovenian.
- DZEROSKI, S., S. MUGGLETON, AND S. RUSSELL (1992). "Pac-learnability of Determinate Logic Programs." *Proceedings of the Fifth ACM Workshop on Computational Learning Theory*, Pittsburgh, 128-135.
- DZEROSKY, S. AND N. JACOBS, M. MOLINA, AND C. MOURE (1998). "ILP Experiments in Detecting Traffic Problems." *Proceedings of the Tenth European Conference on Machine Learning (ECML-98)*, Chemnitz, Germany, Springer, 60-66.

- EMDE, W. (1994). "Inductive Learning of Characteristic Concept Descriptions." *Proceedings of the Fourth International Workshop on Inductive Learning Programming (ILP-94)*, Bonn, Germany, 51-69.
- FINN, P., S. MUGGLETON, D. PAGE AND A. SRINIVASAN (1998). "Pharmacophore Discovery Using the Inductive Logic Programming System PROGOL." *Machine Learning*, 30:241-270.
- GAREY, M. R. AND D. S. JOHNSON (1979). *Computers and Intractability*, Freeman, San Francisco.
- GEIBEL, W. (1996). "Learning Relational Concepts with Decision Trees." *Proceedings of the Thirteenth International Conference on Machine Learning (ICML'96)*, Bari, Italy, Morgan Kaufmann, 166-174.
- GOTTLOB, G. (1985). "On the Efficiency of Subsumption Algorithms." *Journal of the ACM*, 32(2):280-295.
- GOTTLOB, G. (1993). "Removing Redundancy from a Clause." *Artificial Intelligence* 61(2):263-289.
- GROBELNIK, M. (1992). "MARKUS: An optimized Model Inference System." *Proceedings Workshop on Logical Approaches to Machine Learning, Tenth European Conference on Artificial Intelligence*, Vienna, Austria.
- HENERY, R. J. (1994). "Methods for comparison." *Machine Learning, Neural Networks and Statistical Classification*, Michie D., D. J. Spiegelhalter, and C. C. Taylor (eds.), Ellis Horwood, 107-124.
- HERRMANN, J., R. ACKERMANN, J. PETERS AND D. REIPA (1994). "A Multistrategy Learning System and its Integration into an Interactive Floorplanning Tool." *Proceedings of the European Conference on Machine Learning*, Catania, Italy, Springer-Verlag, 138-153.
- IDESTAM-ALMQUIST, P. (1993). "Generalization of Clauses." Ph.D. thesis, Stockholm University, Sweden.
- IDESTAM-ALMQUIST, P. (1995). "Generalization of Clauses under Implications." *Journal of Artificial Intelligence Research*, 3:467-489.
- KAPUR, D. AND P. NARENDHAN (1986). "NP-Completeness of the set unification and matching problems." *Proceedings of the Eighth International Conference on Automated Deduction, Lecture Notes in Computer Science*, Springer, Berlin, 230:489-495.
- KIETZ, J.-U. AND S. WROBEL (1992). "Controlling the complexity of learning in Logic Through Syntactic and Task-Oriented Models." *Inductive Logic Programming*, Muggleton S. (ed.), Academic-Press, 335-359.

- KIRSCHENBAUM, M. AND L. S. STERLING (1991). "Refinement Strategies for Inductive Learning of Simple Prolog Programs." *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, Sydney, Australia, Morgan Kaufmann, 757-761.
- KODRATOFF, Y. AND J.-G. GANASCIA (1986). "Improving the Generalization Step in Learning." *Machine Learning: An Artificial Intelligence Approach - II*, Michalski R. S., J. Carbonell and T. Mitchell (eds.), Morgan Kaufmann, 215-244.
- KODRATOFF, Y. (1990). "Learning Expert Knowledge by Improving the Explanations Provided by the System." *Machine Learning: An Artificial Intelligence Approach-III*, Michalski R. S. and Y. Kodratoff (eds.), Morgan Kaufmann, 433-473.
- LANGLEY, P. (1996). *Elements of Machine Learning*, Morgan Kaufmann.
- LAPOINTE, S. AND S. MATWIN (1992). "Sub-unification: a tool for efficient induction of recursive programs." *Proceedings of the International Conference on Machine Learning*, Aberdeen, Scotland, Morgan Kaufmann, 373-281.
- LAVRAC, N., S. DZEROSKI AND M. GROBELNIK (1991). "Learning nonrecursive definitions of relation with LINUS." *Proceedings Fifth European Working Session on Learning*, Springer, Berlin, 265-281.
- LAVRAC, N. AND S. DZEROSKI (1994). *Inductive Logic Programming Techniques and Applications*, Ellis Horwood.
- LEE, Y., B. BUCHANAN AND J. ARONIS (1998). "Knowledge-based learning in explanatory science: Learning rules to predict rodent carcinogenicity." *Machine Learning*, 30:217-240.
- LLOYD, J. W. (1984). *Foundations of Logic Programming*, Springer-Verlag.
- MICHALSKI, R. S. AND J. B. LARSON (1977). "Inductive inference of VL decision rules." Paper presented at Workshop in Pattern-Directed Inference Systems, Hawaii, 1977, SIGART Newsletter, ACM, 63:38-44.
- MICHALSKI, R. S., I. MOZETIC, J. HONG, AND N. LAVRAC (1986). "The multi-purpose incremental learning system AQ15 and its testing application on three medical domains." *Proceedings of the Fifth National Conference on Artificial Intelligence*, San Mateo, CA, Morgan Kaufmann, 1041-1045.
- MITCHELL, T. M., R. M. KELLER AND S. KEDAR-CABELLI (1986). "Explanation-based generalization: A Unifying view." *Machine Learning*, 1(1):47-80.
- MITCHELL, TOM M. (1997). *Machine Learning*, McGraw-Hill.
- MICHIE, D., S. MUGGLETON, D. PAGE AND A. SRINIVASAN (1994). "To the International Computing Community: A new East-West Challenge." Oxford University Computing Laboratory, Oxford, UK.

- MORIK, K. (1993). "Balanced Cooperative Modeling." *Machine Learning*, 11(2/3):217-235.
- MORIN, J. AND S. MATWIN (1999a). "GENEX: generator of examples". TR-99-03, School of Information Technology and Engineering, University of Ottawa.
- MORIN, J. AND S. MATWIN (1999b). "Extending LGG to work in context". *The Ninth International Workshop on Inductive Logic Programming (ILP-99)*, Bled, Slovenia. To appear.
- MORIN, J. AND S. MATWIN (1999c). "Relational Clichés: their learning and use". In preparation.
- MUGGLETON, S. AND W. BUNTINE (1992a). "Machine invention of first-order predicates by inverting resolution." *Inductive Logic Programming*, Muggleton S. (eds.), Academic-Press, 261-280.
- MUGGLETON, S. AND C. FENG (1992b). "Efficient Induction of Logic Programs." *Inductive Logic Programming*, Muggleton S. (eds.), Academic Press, 281-298.
- MUGGLETON, S. (1995). "Inverse entailment and PROLOG." *New Generation Computing*, 13:245-286.
- NIBLETT, T. (1988). "A Study of generalization in Logic Programs." EWSL-88, London, Pitman, 131-138.
- NIENHUYS-CHENG, S.-H. AND R. D. DE WOLF (1996). "Least Generalizations under Implication." *Proceedings of the Sixth International Workshop on Inductive Learning Programming (ILP-96)*, Stockholm, Sweden, Springer, 285-298.
- PAZZANI, M. J., C. A. BRUNK AND G. SILVERSTEIN (1991). "A Knowledge-intensive Approach to Learning Relational Concepts." *Proceedings of the Eighth International Conference on Machine Learning*, Evanston, Illinois, Morgan Kaufmann, 432-436.
- PAZZANI, M. AND D. KIBLER (1992). "The Utility of Knowledge in Inductive Learning." *Machine Learning*, 9(1):57-94.
- PLOTKIN, G. (1970). "A note on inductive generalization." *Machine Intelligence*, Meltzer, B. and D. Michie (eds.), Edinburg University Press, Edinburg, 5, 153-163.
- PLOTKIN, G. (1971). "A Further Note on Inductive Generalization." *Machine Intelligence*, Meltzer B. and D. Michie (eds.), 6, 101-124, Edinburgh.
- QUINLAN, R. (1986). "Induction of decision trees." *Machine Learning*, 1(1):81-106.
- QUINLAN, R. (1990). "Learning Logical Definitions from Relations." *Machine Learning*, 5:239-266.

- QUINLAN, R. (1991). "Knowledge Acquisition from Structures Data." *IEEE Expert*, 6(6):32-37.
- QUINLAN, J. R. and J. M. CAMERONE-JONES (1993). "FOIL: A Midterm Report." *Proceedings of the European Conference on Machine Learning (ECML-93)*, Vienna, Austria, Springer-Verlag, 3-20.
- RICHARDS, B. L. AND R. J. MOONEY (1992). "Learning Relations by Pathfinding." *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose, CA, AAAI Press, 50-55.
- ROUVEIROL, C. AND J.-F. PUGET (1989). "A Simple Solution for Inverting Resolution." *Proceedings of the Fourth European Working Session on Learning*, Pitman, 201-210.
- ROUVEIROL, C. (1991). "ITOU: Induction of First Order Theories." *Proceedings of the International Workshop on Inductive Logic Programming*, Vienna de Castelo, Portugal, 127-151.
- SAMMUT, C. AND R. B. BANERJI (1986). "Learning Concepts by Asking Questions." *Machine Learning: An Artificial Intelligence Approach-II*, Michalski R. S. (eds.), Morgan Kaufmann, 167-191.
- SHAPIRO, E. (1983). *Algorithmic Program Debugging*, Cambridge, MA, MIT Press.
- SILVERSTEIN, G. AND M. PAZZANI (1991). "Relational Clichés: Constraining constructive induction during relational learning." *Proceedings of the Eighth International Conference on Machine Learning*, Evanston, Illinois, Morgan Kaufmann, 203-207.
- SILVERSTEIN, G. AND M. PAZZANI (1993). "Learning Relational Clichés." *Proceedings of the IJCAI-93 Workshop on Inductive Logic Programming*, Chambery, France, 71-82.
- STAHL, I. (1994). "On the Utility of Predicate Invention in Inductive Logic Programming." *Proceedings of the European Conference on Machine Learning*, Catania, Italy, Springer-Verlag, 272-286.
- SRIVASAN, A , S. MUGGLETON, R. KING, AND M. STERNBERG (1994). "Mutagenesis: ILP Experiments in a Non-determinate biological Domain." *Proceedings of the Fourth International Workshop on Inductive Learning Programming (ILP-94)*, Bonn, Germany, 217-232.
- TAUSEND, B. (1994). "A Representation Language for Biases in Inductive Logic Programming." *Proceedings of the European Conference on Machine Learning*, Catania, Italy, Springer-Verlag, 427-430.
- VRAIN, C. (1990). "OGUST: A System that Learns using Domain Properties Expressed as Theories." *Machine Learning: An Artificial Intelligence Approach - III*, Kodratoff, Y. and R. S. Michalski (eds.), Morgan Kaufmann, 360-382.

WIRTH, R. AND P. O'RORKE (1991). "Constraints on Predicate Invention." *Proceedings of the Eighth International Conference on Machine Learning*, Evanston, Illinois, Morgan Kaufmann, 457-461.

YOKOMORI, T. (1985). "A Logic Schema and its Applications." *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, Morgan Kaufmann, 723-725.

ZELLE, J. M. AND R. MOONEY (1993a). "ILP Techniques for Learning Semantic Grammars." *Proceedings of the IJCAI Workshop on Inductive Logic Programming*, Chambéry, France, Morgan Kaufmann, 83-90.

ZELLE, J. M. AND R. MOONEY (1993b). "Learning Semantic Grammars with Constructive Inductive Logic Programming." *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Washington, DC., 817-822.

ZELLE, J. M. AND R. MOONEY (1994). "Inducing Deterministic Prolog Parses from Treebanks: A Machine Learning Approach." *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, Washington, AAAI Press, 748-753.

Appendix I. Rule templates for blocks examples

This appendix presents the rule templates given to GENEX to generate examples of the *blocks* concept B1. The file *taxoabbrev.pl* contains facts representing the number of levels between predicates in the taxonomy *form/1* (Appendix III). Files used to generate *blocks* examples with GENEX can be found at the address:

<http://www.site.uottawa.ca/~jmorin/Programs/Generator>

Rule templates for concept B1

% B1_rules.pl

```
:- ensure_loaded('cgen_rulesBlck').  
:- ensure_loaded('taxoabbrev.pl').      % taxonomy form - abbreviate names
```

% Rules for positive instances

```
rule(1,(block(x):- [above(x,y), cir(x), rect(y), black(y), above(y,z),  
                    taxo(form, rangl(z), 2)])).  
rule(2,(block(x):- [above(x,y), cir(x), rect(y), above(y,z), taxo(form,  
                    rangl(z), 2)])).  
rule(3,(block(x):- [above(x,y), cir(x), rect(y), black(y), above(y,z),  
                    taxo(form, rangl(z), 2), black(z)])).  
rule(4,(block(x):- [above(x,y), cir(x), rect(y), above(y,z), taxo(form,  
                    rangl(z), 2), black(z)])).
```

% Rules for negative instances

```
rule(30,(block(x):- [above(x,y), cir(x), taxo(form, rect(y), 2), red(y),  
                    above(y,z), taxo(form, ell(z), 2), red(z)])).  
rule(31,(block(x):- [above(x,y), cir(x), taxo(form, rect(y), 2), red(y),  
                    above(y,z), taxo(form, rect(z), 2), red(z)])).  
rule(32,(block(x):- [above(x,y), cir(x), taxo(form, rangl(y), 2), red(y),  
                    above(y,z), taxo(form, ell(z), 2), red(z)])).  
  
gen_b1:- genWithOptions(asLstLit, b1, 1, 4, 100, 30, 32, 100),  
         genWithOptions(asChains, b1, 1, 4, 100, 30, 32, 100), halt.
```

Appendix II. Generated trains examples

Only a subset of the 100 generated *trains* examples are listed here. The files can be found

at: <http://www.site.uottawa.ca/~jmorin/Programs/Generator>

```
train([c(1,hexa,short,not_double,flat,2,1(diam,1)),
      c(2,rect,short,double,peaked,2,1(rect,1))])).

train([c(1,rect,short,not_double,none,2,1(rect,1)),
      c(2,rect,short,not_double,none,2,1(tri,2)),
      c(3,rect,short,double,flat,2,1(diam,2)),
      c(4,ell,short,not_double,arc,2,1(rect,1))])).

train([c(1,rect,long,not_double,none,3,1(utri,0)),
      c(2,ell,short,not_double,arc,2,1(cir,2)),
      c(3,ell,short,not_double,arc,2,1(cir,2)),
      c(4,rect,short,not_double,none,2,1(tri,1))])).

train([c(1,u_shaped,short,not_double,peaked,2,1(cir,1)),
      c(2,ell,short,not_double,arc,2,1(rect,1)),
      c(3,u_shaped,short,not_double,flat,2,1(diam,1))])).

train([c(1,rect,short,double,flat,2,1(cir,2)),
      c(2,hexa,short,not_double,flat,2,1(tri,2))])).

train([c(1,u_shaped,short,not_double,flat,2,1(tri,1)),
      c(2,ell,short,not_double,arc,2,1(rect,2)),
      c(3,rect,long,not_double,flat,3,1(rect,1))])).

train([c(1,rect,long,not_double,jagged,3,1(utri,2)),
      c(2,hexa,short,not_double,flat,2,1(diam,2)),
      c(3,hexa,short,not_double,flat,2,1(rect,1))])).

train([c(1,rect,short,not_double,peaked,2,1(rect,2)),
      c(2,ell,short,not_double,arc,2,1(diam,2)),
      c(3,ell,short,not_double,arc,2,1(tri,2))])).

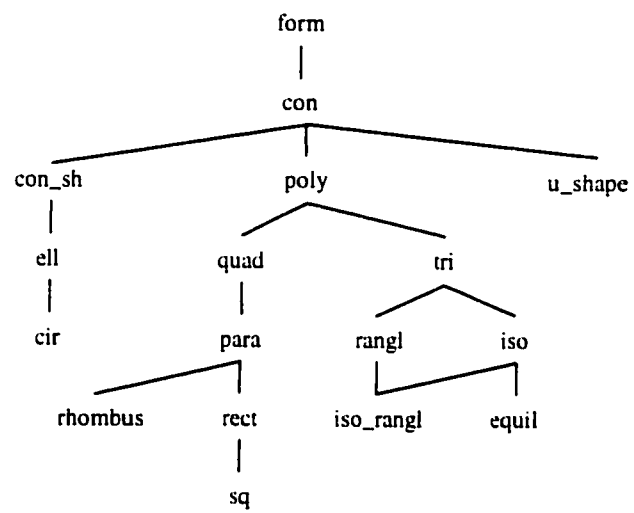
train([c(1,rect,short,double,peaked,2,1(diam,2)),
      c(2,ell,short,not_double,arc,2,1(tri,1))])).

train([c(1,rect,long,not_double,jagged,3,1(utri,2)),
      c(2,rect,long,not_double,flat,3,1(hexa,1)),
      c(3,ell,short,not_double,arc,2,1(rect,1))])).
```

etc...

Appendix III. The taxonomy form/1

The taxonomy form/1 used with concepts in the *blocks* domain.



Appendix IV. The cost of generalizing literals

In the presence of the background knowledge, the most similar and the cheapest clauses are generalized first. The similarity of clauses is presented in Section 5.2.2. The cost of generalizing two clauses is the summation of the cost to generalize literals of the two clauses. The cost of generalizing two literals (or predicates) corresponds to the sum of the generalization steps (hierarchical links to be traversed) required to generalize them into their least general generalization in their taxonomy. The idea is to avoid having a predicate like *square* being generalized first as a *polygon* instead of a *rectangle*.

The cost of generalizing two predicates in a taxonomy is computed only once (prior to the generalization process) and a single access is required to find the cost (and the generalization of two predicates) (Chapter 6). The first level of generalization is worth one, the second one is worth two, etc. For instance, the cost of generalizing the predicates *rhombus* and *rectangle* into a *parallelogram* in the taxonomy *form/1* of Appendix III is 2 (1 for *rhombus* to *parallelogram* and 1 for *rectangle* to *parallelogram*). On the other hand, the cost of generalizing the predicates *rhombus* and *square* is 4 (1 for *rhombus* to *parallelogram* and 3 for *square* to *parallelogram*).

Appendix V. Examples and background literals for mesh-6

Positive examples: (26 positive examples)

mesh(f37, 6).	mesh(f41, 6).	mesh(b41, 6).
mesh(b42, 6).	mesh(j24, 6).	mesh(b11, 6).
mesh(e41, 6).	mesh(b9, 6).	mesh(b14, 6).
mesh(e47, 6).	mesh(e39, 6).	mesh(g20, 6).
mesh(h71, 6).	mesh(f38, 6).	mesh(b6, 6).
mesh(f39, 6).	mesh(g41, 6).	mesh(f40, 6).
mesh(e79, 6).	mesh(h42, 6).	mesh(j29, 6).
mesh(b40, 6).	mesh(b12, 6).	mesh(b1, 6).
mesh(b3, 6).	mesh(b29, 6).	

Negative examples: (35 out of 407 examples are listed here)

mesh(e28, 5).	mesh(f7, 3).	mesh(b34, 2).
mesh(d3, 1).	mesh(e68, 1).	mesh(a35, 1).
mesh(h25, 1).	mesh(e45, 5).	mesh(e21, 3).
mesh(g12, 2).	mesh(h44, 2).	mesh(e33, 1).
mesh(a18, 1).	mesh(b31, 2).	mesh(h18, 2).
mesh(f3, 5).	mesh(f16, 2).	mesh(a29, 1).
mesh(h7, 2).	mesh(e50, 2).	mesh(g6, 4).
mesh(i3, 4).	mesh(d27, 2).	mesh(e59, 1).
mesh(j22, 4).	mesh(i1, 4).	mesh(f19, 3).
mesh(c10, 2).	mesh(h30, 2).	mesh(j11, 4).
mesh(h21, 3).	mesh(e70, 2).	mesh(h37, 4).
mesh(d26, 1).	mesh(i12, 8).	...

Background literals

long(a1).	short_for_hole(a16).	circuit_hole(c20).
long(a34).	short_for_hole(a18).	circuit_hole(c21).
long(a54).	short_for_hole(a23).	circuit_hole(c22).
long(e19).	short_for_hole(a33).	circuit_hole(c23).
long(e22).	short_for_hole(b28).	circuit_hole(f33).
...	...	...
usual(a3).	long_for_hole(a17).	half_circuit_hole(a38).
usual(a39).	long_for_hole(c2).	half_circuit_hole(a42).
usual(b11).	long_for_hole(e41).	half_circuit_hole(a43).
usual(b13).	long_for_hole(e79).	half_circuit_hole(a55).
usual(b15).	long_for_hole(h26).	half_circuit_hole(b1).
...	...	...
short(a6).	not_important(a2).	one_side_loaded(a34).
short(a9).	not_important(a4).	one_side_loaded(a35).
short(a11).	not_important(a5).	one_side_loaded(a40).
short(a13).	not_important(a7).	one_side_loaded(a41).
short(a15).	not_important(a8).	one_side_loaded(a54).
...	...	...
circuit(c15).	one_side_fixed(a34).	two_side_loaded(e37).
circuit(c16).	one_side_fixed(a35).	two_side_loaded(i2).
circuit(c17).	one_side_fixed(a41).	two_side_loaded(i6).
circuit(c18).	one_side_fixed(c2).	two_side_loaded(j2).
circuit(c19).	one_side_fixed(c3).	two_side_loaded(j6).
...	...	...
fixed(a1).	two_side_fixed(a36).	cont_loaded(a8).
fixed(a2).	two_side_fixed(a37).	cont_loaded(a9).
fixed(a3).	two_side_fixed(a38).	cont_loaded(a10).
fixed(a4).	two_side_fixed(a42).	cont_loaded(a11).
fixed(a5).	two_side_fixed(a43).	cont_loaded(a12).
...	...	...
free(a39).	quarter_circuit(b19).	neighbour(a1,a2).
free(a40).	quarter_circuit(b39).	neighbour(a1,a24).
free(c6).	quarter_circuit(b24).	neighbour(a1,a44).
free(c7).	quarter_circuit(b25).	neighbour(a1,a54).
free(c11).	quarter_circuit(e75).	neighbour(a2,a1).
...	...	...
not_loaded(a1).	half_circuit(a36).	opposite(a11,a3).
not_loaded(a2).	half_circuit(a37).	opposite(a3,a11).
not_loaded(a3).	half_circuit(a45).	opposite(a9,a3).
not_loaded(a4).	half_circuit(a46).	opposite(a3,a9).
not_loaded(a5).	half_circuit(a47).	opposite(a31,a25).
...	...	...

Source of files: ftp MachineLearning/ILP/public/data/mesh_design (April 94)

Appendix VI. Clichés learned for mesh-4

Clichés learned with CLUse⁷⁷ for mesh-4. DDCs and DICs for other mesh concepts can be found at:

<http://www.site.uottawa.ca/~jmorin/Programs/CLUse/Output/Mesh>.

22	4095	<i>opposite(x, y), not_loaded(x), two_side_fixed(x), not_loaded(y), two_side_fixed(y)</i>	0.005	0.010
60	4131	<i>opposite(x, y), free(x), free(y)</i>	0.015	0.021
1	4035	<i>opposite(x, y), fixed(x), not_loaded(x), usual(x), fixed(y), not_loaded(y), usual(y)</i>	0.029	0.007
20	4092	<i>neighbour(x, y), fixed(x), not_loaded(x), usual(x), not_loaded(y)</i>	0.029	0.092
21	4093	<i>neighbour(x, y), not_loaded(x), fixed(y), not_loaded(y), usual(y)</i>	0.029	0.092
40	4119	<i>neighbour(x, y), free(x), free(y)</i>	0.029	0.176
44	4123	<i>opposite(x, y), fixed(x), usual(x), fixed(y), not_loaded(y)</i>	0.032	0.000
45	4123	<i>opposite(x, y), fixed(x), fixed(y), not_loaded(y), usual(y)</i>	0.032	0.000
9	4069	<i>opposite(x, y), fixed(x), usual(x), fixed(y), not_loaded(y), usual(y)</i>	0.032	0.012
23	4101	<i>neighbour(x, y), not_loaded(x), free(y), not_loaded(y)</i>	0.036	0.169
4	4040	<i>neighbour(x, y), one_side_fixed(x), one_side_loaded(x), usual(x), cont_loaded(y), free(y), usual(y)</i>	0.046	0.002
5	4041	<i>neighbour(x, y), one_side_fixed(x), one_side_loaded(x), usual(x), fixed(y), not_loaded(y), usual(y)</i>	0.046	0.002
6	4042	<i>neighbour(x, y), cont_loaded(x), free(x), usual(x), one_side_fixed(y), one_side_loaded(y), usual(y)</i>	0.046	0.002
7	4043	<i>neighbour(x, y), fixed(x), not_loaded(x), usual(x), one_side_fixed(y), one_side_loaded(y), usual(y)</i>	0.046	0.002
25	4104	<i>neighbour(x, y), fixed(x), not_loaded(x), usual(x), half_circuit(y), two_side_fixed(y)</i>	0.049	0.000
26	4104	<i>neighbour(x, y), fixed(x), usual(x), half_circuit(y), not_loaded(y), two_side_fixed(y)</i>	0.049	0.000
28	4105	<i>neighbour(x, y), half_circuit(x), not_loaded(x), two_side_fixed(x), fixed(y), usual(y)</i>	0.049	0.000
29	4105	<i>neighbour(x, y), half_circuit(x), two_side_fixed(x), fixed(y), not_loaded(y), usual(y)</i>	0.049	0.000
2	4038	<i>neighbour(x, y), fixed(x), not_loaded(x), usual(x), half_circuit(y), not_loaded(y), two_side_fixed(y)</i>	0.049	0.018
3	4039	<i>neighbour(x, y), half_circuit(x), not_loaded(x), two_side_fixed(x), fixed(y), not_loaded(y), usual(y)</i>	0.049	0.018
49	4127	<i>neighbour(x, y), fixed(x), usual(x), two_side_fixed(y)</i>	0.053	0.000
50	4127	<i>neighbour(x, y), usual(x), half_circuit(y), two_side_fixed(y)</i>	0.053	0.000
52	4128	<i>neighbour(x, y), half_circuit(x), two_side_fixed(x), usual(y)</i>	0.053	0.000
53	4128	<i>neighbour(x, y), two_side_fixed(x), fixed(y), usual(y)</i>	0.053	0.000
24	4104	<i>neighbour(x, y), fixed(x), usual(x), half_circuit(y), two_side_fixed(y)</i>	0.053	0.026
27	4105	<i>neighbour(x, y), half_circuit(x), two_side_fixed(x), fixed(y), usual(y)</i>	0.053	0.026
43	4123	<i>opposite(x, y), fixed(x), fixed(y), not_loaded(y)</i>	0.056	0.041
48	4127	<i>neighbour(x, y), usual(x), two_side_fixed(y)</i>	0.058	0.038
51	4128	<i>neighbour(x, y), two_side_fixed(x), usual(y)</i>	0.058	0.038
31	4113	<i>neighbour(x, y), fixed(x), not_loaded(x), not_loaded(y)</i>	0.061	0.277
46	4124	<i>opposite(x, y), fixed(x), fixed(y)</i>	0.063	0.061
10	4070	<i>neighbour(x, y), one_side_fixed(x), one_side_loaded(x), usual(x), cont_loaded(y), free(y)</i>	0.066	0.005
11	4071	<i>neighbour(x, y), one_side_fixed(x), one_side_loaded(x), usual(x), fixed(y), not_loaded(y)</i>	0.066	0.005
12	4072	<i>neighbour(x, y), cont_loaded(x), free(x), one_side_fixed(y), one_side_loaded(y), usual(y)</i>	0.066	0.005

⁷⁷ SimCh = SimBind = 0.01, PosCov = 0.025, NegCov = 1.

13	4073	<i>neighbour(x, y), fixed(x), not_loaded(x), one_side_fixed(y), one_side_loaded(y), usual(y)</i>	0.066	0.005
47	4126	<i>neighbour(x, y), cont_loaded(x), cont_loaded(y)</i>	0.068	0.134
8	4044	<i>neighbour(x, y), fixed(x), not_loaded(x), usual(x), fixed(y), not_loaded(y), usual(y)</i>	0.083	0.017
35	4117	<i>neighbour(x, y), free(x), one_side_fixed(y), one_side_loaded(y)</i>	0.087	0.000
36	4117	<i>neighbour(x, y), free(x), one_side_loaded(y), usual(y)</i>	0.087	0.000
38	4118	<i>neighbour(x, y), one_side_fixed(x), one_side_loaded(x), free(y)</i>	0.087	0.000
39	4118	<i>neighbour(x, y), one_side_loaded(x), usual(x), free(y)</i>	0.087	0.000
16	4086	<i>neighbour(x, y), one_side_fixed(x), one_side_loaded(x), usual(x), free(y)</i>	0.087	0.006
17	4087	<i>neighbour(x, y), one_side_fixed(x), one_side_loaded(x), usual(x), not_loaded(y)</i>	0.087	0.006
18	4088	<i>neighbour(x, y), free(x), one_side_fixed(y), one_side_loaded(y), usual(y)</i>	0.087	0.006
19	4089	<i>neighbour(x, y), not_loaded(x), one_side_fixed(y), one_side_loaded(y), usual(y)</i>	0.087	0.006
55	4129	<i>neighbour(x, y), one_side_fixed(x), not_loaded(y)</i>	0.095	0.000
56	4129	<i>neighbour(x, y), usual(x), not_loaded(y)</i>	0.095	0.000
58	4130	<i>neighbour(x, y), not_loaded(x), usual(y)</i>	0.095	0.000
59	4130	<i>neighbour(x, y), one_side_fixed(y), usual(y)</i>	0.095	0.000
14	4075	<i>neighbour(x, y), fixed(x), not_loaded(x), usual(x), fixed(y), not_loaded(y)</i>	0.095	0.048
32	4115	<i>neighbour(x, y), one_side_fixed(x), usual(x), not_loaded(y)</i>	0.095	0.062
33	4116	<i>neighbour(x, y), not_loaded(x), one_side_fixed(y), usual(y)</i>	0.095	0.062
34	4117	<i>neighbour(x, y), free(x), one_side_loaded(y)</i>	0.097	0.015
37	4118	<i>neighbour(x, y), one_side_loaded(x), free(y)</i>	0.097	0.015
41	4120	<i>neighbour(x, y), free(x)</i>	0.126	0.223
15	4085	<i>neighbour(x, y), fixed(x), not_loaded(x), fixed(y), not_loaded(y)</i>	0.146	0.166
57	4130	<i>neighbour(x, y), usual(y)</i>	0.153	0.280
30	4110	<i>neighbour(x, y), not_loaded(x), fixed(y), not_loaded(y)</i>	0.206	0.277
42	4121	<i>neighbour(x, y), not_loaded(x), not_loaded(y)</i>	0.320	0.597
54	4129	<i>neighbour(x, y), not_loaded(y)</i>	0.415	0.639

DICs for mesh-4

- 1 *extp351(x, y), extp352(x), extp353(x), extp354(x), extp355(y), extp356(y), extp357(y)*
- 2 *extp407(x, y), extp408(x), extp409(x), extp410(y), extp411(y), extp412(y)*
- 3 *extp443(x, y), extp444(x), extp445(x), extp446(y), extp447(y)*
- 4 *extp483(x, y), extp484(x), extp485(y), extp486(y)*
- 5 *extp537(x, y), extp538(x), extp539(y)*
- 6 *extp562(x, y), extp563(x)*

Appendix VII. DDCs vs. rule templates

All generalizations learned with CLUSE⁷⁸ for B4-chains.

G310: <i>above</i> (S203,S204), <i>Ell</i> (S203), <i>Poly</i> (S204).	{more than 20 –not listed}
<i>large</i> (S101)	{9,19,41,70}
<i>small</i> (S102)	{9,19,41,70}
G305: <i>above</i> (S193,S194), <i>Ell</i> (S193), <i>rect</i> (S194).	{2,8,15,20,25,34,42,44,50,51,55,58,59,73,79,80,83,90,98,100}
G280: <i>above</i> (S143,S144), <i>ell</i> (S143), <i>small</i> (S143), <i>rect</i> (S144).	{15,55,73,90}
G253: <i>above</i> (S90,S89), <i>ell</i> (S90), <i>small</i> (S90), <i>large</i> (S89), <i>rect</i> (S89).	{15,55,90}
G249: <i>above</i> (S82,S81), <i>ell</i> (S82), <i>large</i> (S81), <i>rect</i> (S81).	{2,42,83}
G296: <i>above</i> (S175,S176), <i>cir</i> (S175), <i>Rect</i> (S176).	{8,20,25,34,44,50,51,58,59,79,80,98,100}
G241: <i>above</i> (S66,S65), <i>cir</i> (S66), <i>small</i> (S66), <i>rect</i> (S65).	{50,51}
G254: <i>above</i> (S92,S91), <i>cir</i> (S92), <i>large</i> (S91), <i>rect</i> (S91).	{25,79,80}
G285: <i>above</i> (S153,S154), <i>cir</i> (S153), <i>sq</i> (S154).	{20,34,44,58,59,98,100}
G264: <i>above</i> (S112,S111), <i>cir</i> (S112), <i>small</i> (S112), <i>sq</i> (S111).	{34,44,58,59,98,100}
G302: <i>above</i> (S187,S188), <i>Ell</i> (S187), <i>Iso</i> (S188).	{more than 20 –not listed}
G297: <i>above</i> (S177,S178), <i>Ell</i> (S177), <i>iso_rangl</i> (S178).	{10,23,24,28,60,69,76,85,86,92}
G290: <i>above</i> (S163,S164), <i>cir</i> (S163), <i>iso_rangl</i> (S164).	{10,23,24,69,76,86,92}
G252: <i>above</i> (S88,S87), <i>cir</i> (S88), <i>iso_rangl</i> (S87), <i>red</i> (S87).	{10,76,92}
G267: <i>above</i> (S117,S118), <i>cir</i> (S117), <i>small</i> (S117), <i>iso_rangl</i> (S118).	{23,24}
<i>red</i> (x48)	{24}
G227: <i>above</i> (S38,S37), <i>ell</i> (S38), <i>iso_rangl</i> (S37), <i>red</i> (S37).	{28,60}
G233: <i>above</i> (S50,S49), <i>ell</i> (S50), <i>small</i> (S50), <i>iso</i> (S49).	{36,46}
G269: <i>above</i> (S121,S122), <i>ell</i> (S121), <i>iso</i> (S122), <i>red</i> (S122).	{12,89,102}
<i>small</i> (x203)	{102}
G270: <i>above</i> (S123,S124), <i>cir</i> (S123), <i>iso</i> (S124).	{3,16,21,62}
G211: <i>above</i> (S6,S5), <i>cir</i> (S6), <i>iso</i> (S5), <i>red</i> (S5).	{3,62}
G291: <i>above</i> (S165,S166), <i>cir</i> (S165), <i>equil</i> (S166).	{27,38,39,43,52,56,65,95,103}
G281: <i>above</i> (S145,S146), <i>cir</i> (S145), <i>equil</i> (S146), <i>red</i> (S146).	{27,39,43,56,95}
G255: <i>above</i> (S94,S93), <i>cir</i> (S94), <i>small</i> (S94), <i>equil</i> (S93), <i>red</i> (S93).	{27,39,56}
G235: <i>above</i> (S54,S53), <i>cir</i> (S54), <i>small</i> (S54), <i>equil</i> (S53).	{38,52}
G268: <i>above</i> (S119,S120), <i>ell</i> (S119), <i>small</i> (S119), <i>equil</i> (S120).	{49,74}
<i>red</i> (x148)	{74}
G293: <i>above</i> (S169,S170), <i>ell</i> (S169), <i>sq</i> (S170).	{6,7,26,29,31,47,48,53,54,66,84,93,97,101}
G251: <i>above</i> (S86,S85), <i>ell</i> (S86), <i>small</i> (S86), <i>sq</i> (S85).	{7,29,66}
G271: <i>above</i> (S125,S126), <i>ell</i> (S125), <i>large</i> (S126), <i>sq</i> (S126).	{6,26,84,101}
G214: <i>above</i> (S12,S11), <i>ell</i> (S12), <i>small</i> (S12), <i>large</i> (S11), <i>sq</i> (S11).	{6,101}
G259: <i>above</i> (S102,S101), <i>cir</i> (S102), <i>small</i> (S102), <i>large</i> (S101), <i>sq</i> (S101).	{9,19,41,70}
G309: <i>leftof</i> (S201,S202), <i>Rect</i> (S201), <i>Iso</i> (S202).	{all}

⁷⁸ NbPosCh = NbNegCh = 104; SimCh = SimBind = PosCov = 0, NegCov = 1.

<i>red(S198)</i>	{5,37,61,64,68,96,104}
<i>large(S171)</i>	{1,18,22,30,57,63,91}
<i>G262: leftof(S108,S107), large(S108), rect(S108), iso(S107).</i>	{72,81,87}
<i>G265: leftof(S113,S114), rect(S113), iso(S114), red(S114).</i>	{11,67}
<i>large(x21)</i>	{11}
<i>G275: leftof(S133,S134), sq(S133), iso(S134).</i>	{4,45,77,78}
<i>G212: leftof(S8,S7), large(S8), sq(S8), iso(S7).</i>	{4,78}
<i>G288: leftof(S159,S160), rect(S159), iso_rangl(S160).</i>	{13,17,32,33,40,75,82,94}
<i>G274: leftof(S131,S132), large(S131), rect(S131), iso_rangl(S132).</i>	{33,40,75,82}
<i>G230: leftof(S44,S43), large(S44), rect(S44), iso_rangl(S43), red(S43).</i>	{33,82}
<i>G294: leftof(S171,S172), large(S171), Rect(S171), equil(S172).</i>	{1,18,22,30,57,63,91}
<i>red(x36)</i>	{18}
<i>G278: leftof(S139,S140), large(S139), sq(S139), equil(S140).</i>	{1,22,30,57,63}
<i>G248: leftof(S80,S79), large(S80), sq(S80), equil(S79), red(S79).</i>	{1,30,57}
<i>G307: leftof(S197,S198), sq(S197), Iso(S198), red(S198).</i>	{5,37,61,64,68,96,104}
<i>large(S84)</i>	{5,61,104}
<i>G282: leftof(S147,S148), sq(S147), iso_rangl(S148), red(S148).</i>	{37,64,68,96}
<i>large(x191)</i>	{96}
<i>G250: leftof(S84,S83), large(S84), sq(S84), iso(S83), red(S83).</i>	{5,61,104}

Appendix VIII. Rules learned by xFOIL for mesh-4

Rules for other FEM concepts can be found at:

<http://www.site.uottawa.ca/~jmorin/Programs/xFoil/Output/Mesh>.

- 1 *mesh(X,Y):- usual(X), one_side_loaded(X), one_side_fixed(X).*
- 2 *mesh(X,Y):- usual(X), fixed(X), not_loaded(X).*
- 3 *mesh(X,Y):- half_circuit_hole(X), not_loaded(X), two_side_fixed(X).*
- 4 *mesh(X,Y):- long(X), two_side_fixed(X).*
- 5 *mesh(X,Y):- long_for_hole(X), free(X).*
- 6 *mesh(X,Y):- half_circuit_hole(X), fixed(X), opposite(X,Z), half_circuit_hole(Z).*
- 7 *mesh(X,Y):- cont_loaded(X), usual(X), free(X).*
- 8 *mesh(X,Y):- long(X), free(X).*
- 9 *mesh(X,Y):- cont_loaded(X), fixed(X), opposite(X,Z), long(Z).*
- 10 *mesh(X,Y):- usual(X), cont_loaded(X), fixed(X), opposite(X,Z), usual(Z), fixed(Z), not_loaded(Z), neighbour(X,W), usual(W).*
- 11 *mesh(X,Y):- one_side_fixed(X), usual(X), not_loaded(X).*

Appendix IX. Rules learned by xFOIL-DDC for mesh-4

Rules for other FEM concepts can be found at:

<http://www.site.uottawa.ca/~jmorin/Programs/xFoil/Output/Mesh>.

- 1 *mesh(X, Y):- usual(X), one_side_loaded(X), one_side_fixed(X).*
- 2 *mesh(X, Y):- neighbour(X,Z), free(Z), not_loaded(X), not_loaded(Z), usual(X), usual(Z), fixed(X).*
- 3 *mesh(X, Y):- neighbour(X,Z), not_loaded(X), not_loaded(Z), usual(Z), usual(X), fixed(X), fixed(Z).*
- 4 *mesh(X, Y):- neighbour(X,Z), not_loaded(X), not_loaded(Z), one_side_fixed(Z), half_circuit_hole(X), two_side_fixed(X).*
- 5 *mesh(X, Y):- half_circuit_hole(X), not_loaded(X), two_side_fixed(X).*
- 6 *mesh(X, Y):- long(X), two_side_fixed(X).*
- 7 *mesh(X, Y):- long_for_hole(X), free(X).*
- 8 *mesh(X, Y):- half_circuit_hole(X), fixed(X), opposite(X,Z), half_circuit_hole(Z).*
- 9 *mesh(X, Y):- cont_loaded(X), usual(X), free(X).*
- 10 *mesh(X, Y):- long(X), free(X).*
- 11 *mesh(X, Y):- cont_loaded(X), fixed(X), opposite(X,Z), long(Z).*
- 12 *mesh(X, Y):- usual(X), cont_loaded(X), fixed(X), opposite(X,Z), usual(Z), fixed(Z), not_loaded(Z), neighbour(X, W), usual(W).*
- 13 *mesh(X, Y):- neighbour(X,Z), free(Z), not_loaded(X), not_loaded(Z), usual(X), usual(Z), one_side_fixed(X).*

DDCs-2 learned for mesh-4

Rule# DDC# DDC

- | | | |
|----|----|--|
| 2 | 11 | <i>neighbour(X,Z), free(Z), not_loaded(X), not_loaded(Z), usual(X), usual(Z)</i> |
| 3 | 26 | <i>neighbour(X,Z), not_loaded(X), not_loaded(Z), usual(Z)</i> |
| 4 | 21 | <i>neighbour(X,Z), not_loaded(X), not_loaded(Z), one_side_fixed(Z)</i> |
| 13 | 11 | <i>neighbour(X,Z), free(Z), not_loaded(X), not_loaded(Z), usual(X), usual(Z)</i> |

Appendix X. Rules learned by xFOIL-DIC for mesh-4

Rules for other FEM concepts can be found at:

<http://www.site.uottawa.ca/~jmorin/Programs/xFoil/Output/Mesh>.

- 1 *mesh(X,Y):- usual(X), one_side_loaded(X), one_side_fixed(X).*
- 2 *mesh(X,Y):- neighbour(X,Z), half_circuit(Z), two_side_fixed(Z), usual(X), fixed(X), not_loaded(X), opposite(Z,W), two_side_fixed(W), not_loaded(W), half_circuit(W).*
- 3 *mesh(X,Y):- neighbour(X,Z), fixed(Z), short(Z), half_circuit_hole(X), not_loaded(X), two_side_fixed(X).*
- 4 *mesh(X,Y):- neighbour(X,Z), free(Z), half_circuit(Z), usual(X), cont_loaded(X), free(X).*
- 5 *mesh(X,Y):- long(X), two_side_fixed(X).*
- 6 *mesh(X,Y):- long_for_hole(X), free(X).*
- 7 *mesh(X,Y):- usual(X), fixed(X), cont_loaded(X), opposite(X,Z), usual(Z), fixed(Z), not_loaded(Z), neighbour(X,W), usual(W).*
- 8 *mesh(X,Y):- half_circuit_hole(X), fixed(X), opposite(X,Z), half_circuit_hole(Z).*
- 9 *mesh(X,Y):- neighbour(X,Z), half_circuit(Z), two_side_fixed(Z), usual(X), fixed(X), not_loaded(X), opposite(Z,W), long(W).*
- 10 *mesh(X,Y):- long(X), free(X).*
- 11 *mesh(X,Y):- neighbour(X,Z), circuit(Z), free(Z), usual(X), fixed(X), not_loaded(X).*
- 12 *mesh(X,Y):- neighbour(X,Z), free(Z), usual(Z), usual(X), one_side_fixed(X), not_loaded(X).*
- 13 *mesh(X,Y):- fixed(X), cont_loaded(X), short(X), opposite(X,Z), long(Z).*
- 14 *mesh(X,Y):- neighbour(X,Z), half_circuit(Z), two_side_fixed(Z), usual(X), fixed(X), not_loaded(X).*

DICs learned for mesh-4

Rule#	DIC#	DIC
2	2	<i>neighbour(X,Z), half_circuit(Z), two_side_fixed(Z)</i>
3	2	<i>neighbour(X,Z), fixed(Z), short(Z)</i>
4	2	<i>neighbour(X,Z), free(Z), half_circuit(Z)</i>
9	2	<i>neighbour(X,Z), half_circuit(Z), two_side_fixed(Z)</i>
11	2	<i>neighbour(X,Z), circuit(Z), free(Z)</i>
12	2	<i>neighbour(X,Z), free(Z), usual(Z)</i>
14	2	<i>neighbour(X,Z), half_circuit(Z), two_side_fixed(Z)</i>

Appendix XI. Rules learned by xFOIL-DIC for mesh-4 (10 literals)

Rules for other FEM concepts can be found at:

<http://www.site.uottawa.ca/~jmorin/Programs/xFoil/Output/Mesh>.

- 1 *mesh(X,Y):- usual(X), one_side_loaded(X), one_side_fixed(X).*
- 2 *mesh(X,Y):- neighbour(X,Z), half_circuit(Z), two_side_fixed(Z), usual(X), fixed(X), not_loaded(X), opposite(Z,W), two_side_fixed(W), not_loaded(W), half_circuit(W).*
- 3 *mesh(X,Y):- neighbour(X,Z), fixed(Z), short(Z), half_circuit_hole(X), not_loaded(X), two_side_fixed(X).*
- 4 *mesh(X,Y):- neighbour(X,Z), free(Z), half_circuit(Z), usual(X), cont_loaded(X), free(X).*
- 5 *mesh(X,Y):- long(X), two_side_fixed(X).*
- 6 *mesh(X,Y):- long_for_hole(X), free(X).*
- 7 *mesh(X,Y):- usual(X), fixed(X), cont_loaded(X), opposite(X,Z), usual(Z), fixed(Z), not_loaded(Z), neighbour(X,W), usual(W).*
- 8 *mesh(X,Y):- half_circuit_hole(X), fixed(X), opposite(X,Z), half_circuit_hole(Z).*
- 9 *mesh(X,Y):- neighbour(X,Z), half_circuit(Z), two_side_fixed(Z), usual(X), fixed(X), not_loaded(X), opposite(Z,W), long(W).*
- 10 *mesh(X,Y):- long(X), free(X).*
- 11 *mesh(X,Y):- neighbour(X,Z), free(Z), not_loaded(Z), usual(X), fixed(X), not_loaded(X).*
- 12 *mesh(X,Y):- neighbour(X,Z), free(Z), usual(Z), usual(X), one_side_fixed(X), not_loaded(X).*
- 13 *mesh(X,Y):- fixed(X), cont_loaded(X), short(X), opposite(X,Z), long(Z).*
- 14 *mesh(X,Y):- neighbour(X,Z), half_circuit(Z), two_side_fixed(Z), usual(X), fixed(X), not_loaded(X).*

DICs learned for mesh-4 (10 literals)

Rule#	DIC#	DIC
2	2	<i>neighbour(X,Z), half_circuit(Z), two_side_fixed(Z)</i>
3	2	<i>neighbour(X,Z), fixed(Z), short(Z)</i>
4	2	<i>neighbour(X,Z), free(Z), half_circuit(Z)</i>
9	2	<i>neighbour(X,Z), half_circuit(Z), two_side_fixed(Z)</i>
11	2	<i>neighbour(X,Z), free(Z), not_loaded(Z)</i>
12	2	<i>neighbour(X,Z), free(Z), usual(Z)</i>
14	2	<i>neighbour(X,Z), half_circuit(Z), two_side_fixed(Z)</i>

Index

- A
 - accuracy, 140, 142
- B
 - background knowledge (BK), 114
 - background literal, 35, 117
 - Bongard, 13, 32
- C
 - chain, 93, 94
 - chain template, 111
 - characterization of a concept, 30
 - clause
 - connected, 91
 - equivalent, 67
 - reduced, 68
- CLGG
 - algorithm, 72
 - of clauses, 61, 72
- compactness, 144
- connection principle, 85
- context, 60
- Contextual Least General Generalization, 18
- cost, 59, 96, 144
- coverage
 - frequency, 97
 - threshold, 98
- D
 - distance, 96
- domain-dependent cliché (DDC), 93
- domain-independent cliché (DIC), 93
- E
 - exactly subsumed, 97
- extended tuple, 122
- extensional predicate, 25
- F
 - finite element methods (FEM), 18
- G
 - generalization
 - under θ -subsumption, 63, 67
- generator
 - of examples, 43
 - of taxonomies, 45, 57
- GSS_F, 68
- I
 - information gain, 117
- intensional predicate, 25
- L
 - LGG_F, 61, 67
- local plateau, 16, 114
- M
 - matching literals, 60
 - most frequently used literals, 131, 156
- N
 - nested structure, 42, 45, 54, 56, 82
- O
 - overgeneralization, 86
- P
 - partial ordering, 121
- R
 - relational cliché, 17, 92
 - renaming substitution, 62
 - rule template, 46
- S
 - similarity
 - of chains, 109
 - of clauses, 80
 - of constants, 81
- specificity constraint, 146, 157
- standardized apart, 49, 62
- statistically significant, 144
- subsumption test, 69
- V
 - variabilization, 25, 117, 131
- variable
 - new, 116
 - old, 116