Appendix A

General Inductive Logic Programming System Manual

GILPS is a general Inductive Logic Programming framework which implements all the contributions of this dissertation. In particular, it includes TopLog, ProGolem and the sophisticated coverage engines and subsumption engine described in Chapters 3, 4 and 5.

A.1 Using GILPS

GILPS is free for academic usage and its latest version is available at http://ilp.doc.ic.ac.uk/GILPS. GILPS has been developed in Prolog, and requires YAP to be executed [Cos09]. YAP is a high-performance Prolog compiler that is particularly suitable for ILP implementations. YAP is also freely available for academic usage and its latest version is at http://www.dcc.fc.up.pt/∼vsc/Yap/.

GILPS has been tested with YAP 6. Earlier versions of YAP do not have all the necessary features to execute GILPS.

At GILPS' webpage there are a set of datasets which GILPS has been applied to. We suggest that the reader downloads these datasets in order to see how GILPS has been configured to solve these problems. The background knowledge and mode declarations definitions of GILPS

are identical to Aleph and Progol. We refer the reader to their manuals ([Sri07, MF01]) for a more thorough description of these common aspects.

A.2 Examples definition

The examples definitions in GILPS are slightly different than those in Aleph and Progol. In GILPS the examples are specified by the user in any background knowledge file through the special predicate example/2. The first argument is the example itself (a term) and the second is the example weight (a real number). A positive weight represents a positive example, a negative weight a negative example. E.g.

$$example(bind(p2BVW), 1)$$

specifies that the example 'bind(p2BVW)' is a positive example with weight 1. Optionally, the user can use example/3 in order to specify the fold of the example. E.g.

$$example(bind(p1T10), -1, 3)$$

specifies that 'bind(p1T10)' is a negative example with weight 1 belonging to fold 3. This is useful when using the same folds as other researchers have and allow directly comparing the cross-validation results. When the examples are loaded, they are assigned an integer example identifier incrementally, starting from 1.

A.3 Commands

Only a small set of commands are needed to use GILPS. GILPS commands are defined in module 'gilps.pl'.

build_theory/0 Starts the learning and possibly cross-validation of the problem read with *read_problem* using the current settings in the ILP engine specified by the *engine* setting.

example/{2,3} Specifies an example. The example definition can take either two or three arguments. See Section A.2.

modeb/{2,3} Specifies a mode body declaration, e.g. modeb(*, has_aminoacid(+pdb, -aminoacid_id, #aminoacid_name)). The first argument is an integer specifying the recall. The symbol '*' means indeterminate recall. The precise value assigned depends on the setting 'star_default_recall'. The third argument is optional and specifies whether the mode body declaration is commutative, e.g. modeb(1, diff_aminoacid(+aminoacid_id, +aminoacid_id),commutative).

modeh/{1,2} Specifies the mode head declaration, e.g. modeh(bind(+pdb)). Optionally, for compatibility with Aleph and Progol, there may be an integer specifying the recall, e.g. modeh(*, bind(+pdb)). However, the recall has no meaning in modeh declaration and is ignored by GILPS.

read_problem/1 Reads a Prolog file, which must define, or load files that define, the background knowledge, mode declarations and examples for the problem at hand. E.g. read_problem('hexose.pl').

sat/1 Display the variablized most-specific clause for the example with id (a positive integer) provided in the first argument. E.g. sat(1).

set/2 The first argument is the setting name, the second is the value. If the second argument is ground, the value of the setting is changed, otherwise it is bound to the variable. E.g. set(i, 4), set(clause_evaluation, X) ? X= 'smallest_predicate_domain'. See A.4 for a list of all possible GILPS settings.

ground_sat/1 Display the ground most-specific clause for the example with id (a positive integer) provided in the first argument. E.g. ground_sat(1).

evaluate_theory/1 Evaluates a previously constructed theory from a file after the background knowledge and examples are read with *read_problem*. E.g. evaluate_theory('theory.pl').

A.4 Settings

Below is a description of all the user-definable settings of GILPS. These settings are defined in module 'settings/settings.pl'. Unless otherwise stated these settings are valid on all ILP systems in GILPS.

bottom_early_stop (Default=false) If true and there are output variables in the modeh declaration, stops the construction of the most-specific clause in the earliest layer where all the output variables have already occurred. If false or there are no output variables in the modeh declaration, constructs the bottom clause normally up to depth 'i'. This setting is only applicable to ILP engines which use most-specific clauses (i.e. ProGolem and FuncLog).

clause_evaluation (Default=smallest_predicate_domain) Defines the clause_evaluation engine to use when computing the coverage of a clause. This setting is applicable to all ILP systems in GILPS but is particularly important to ProGolem, as ProGolem will generate much larger clauses than a top-down ILP system.

Possible choices for the clause_evaluation engine are:

- 'left_to_right': uses Prolog built-in left-to-right, depth-first search heuristic for SLD-resolution. This is only advisable for very short or determinate clauses.
- 'smallest_predicate_domain': SLD-resolution with a selection heuristic that selects at each moment the literal which has the fewest solutions. This heuristic has an overhead compared with 'left-to-right' but, by failing earlier in the search, handles non-determinate clauses better.
- 'smallest_variable_domain': SLD-resolution with a selection heuristic that selects at each moment the variable which has the fewest possible values. This heuristic has an overhead compared with 'left-to-right' and 'smallest_predicate_domain' but may pay off on non-determinate clauses when the number of distinct variables is much smaller than the number of literals in the clause.

- 'advanced': identical to 'smallest_variable_domain' but decomposes a clause recursively in independent sub-components. This engine has a greater overhead than 'smallest_variable_domain' but may pay off on decomposable non-determinate clauses.
- 'theta_subsumption': performs theta-subsumption, using Subsumer [SM10a], between hypothesis and example. Note that the background knowledge must be pure-Prolog.

A detailed explanation of these coverage engines was presented in Section 4.3.5. A benchmark on a representative dataset of ILP problems was also presented in Section 4.4.3.

clause_length (Default=4) Defines the maximum number of literals (including the head) of a valid clause. ProGolem ignores this setting but TopLog and FuncLog adhere to it.

cross_validation_folds (Default=1) Number of folds to perform cross-validation. A value of '1' instructs GILPS to use all examples for training. Remember that when an example is specified the user can pre-assign it to a specific fold (see Section A.2). If there is no pre-assigned fold for an example, it will be assigned to a random fold.

cut_transformation (Default=false) If 'true' applies the cut-transformation as specified in [CSC+03]. The cut-transformation can only be applied if 'clause_evaluation=left_to_right'. The purpose of this transformation is to speed-up clause evaluation by transforming the clause before coverage computation. Although still applicable, the cut-transformation has mostly been superseded by the more sophisticated clause evaluation engines available through the 'clause_evaluation' setting.

engine (Default=progolem) Defines which ILP engine GILPS should use. The possibilities are currently:

- 'toplog': TopLog is a top-down ILP system that uses a logic program instead of mode declarations to define the hypothesis space. See Chapter 3 for further details.
- 'progolem': ProGolem is a bottom-up ILP system using asymmetric relative minimal generalizations as the specialization operator. See Chapter 4 for further details.

- 'funclog': FuncLog is a specialized ILP learner for head output connected predicates (i.e. functions). See [STNM09] for further details.

evalfn (Default=compression) Defines which function to use when scoring a clause. Suppose this clause has NL literals and covers TP true positive examples, FP false positive examples, TN true negative examples and FN false negative examples. The total number of examples, E, is $TP + FP + TN + FN$. The most relevant scoring functions are:

- 'accuracy': $(TP + TN)/E$
- 'compression': $TP - FP - NL$
- 'compression_ratio': $(TP - FP)/NL$
- 'coverage': $TP - FP$
- 'novelty': $TP/N - ((TP + FN) * (TP + FP))/(E * E)$
- 'precision': $TP/(TP + FP)$

While defining the scoring of functions, we took into account the remarks of [LFZ99] and the several functions there mentioned (e.g. novelty). For a full list of available scoring functions see module 'hypotheses/score.pl', where you can also specify your own scoring function.

depth (Default=20) Maximum depth for the proof of any clause. This setting is important to ensure the interpreter does not enter an infinite loop when evaluating badly behaved recursive hypotheses or background knowledge.

example_inflation (Default=1) The weight of each example as specified in the examples definition is multiplied by this factor. Remember that when defining the examples, it is possible to assign a custom weight for each example, therefore allowing some examples to be more important than others (see Section A.2). Also notice that if 'example_inflation' is a negative number, the positive and negative examples swap places. See also *positive_example_inflation* and *negative_example_inflation*.

i (Default=3) Defines the number of layers of new variables when constructing the most-specific clause for an example (see Figure C.1). This setting is ignored by TopLog as it does not need to construct most-specific clauses.

maximum_singletons_in_clause (Default=inf) Maximum number of singleton variables (i.e. variables which just occur once) in a clause. This is a TopLog-specific setting.

maxneg (Default=inf) Maximum weight of negative examples that may be covered by a single clause.

max_clauses_per_theory (Default=inf) Maximum number of hypotheses in the final induced theory. The default value of 'inf' allows the addition of whatever number of clauses may be needed, as long as there is an incremental gain in adding these clauses to the current theory. The incremental gain is measured according to the *evalfn* setting and also considers the positive and negative examples the theory covers so far.

max_resolutions (Default=10000) This setting is only applicable when the clause evaluation engine is 'left_to_right'. It defines the maximum number of resolutions allowed before failing a coverage test. The maximum resolutions may be set to 'inf' to ensure proper coverage computation (i.e. infinite resolutions). Keep in mind, though, that if the clause under evaluation is long and non-determinate, it is likely the ILP system may take too long. In this case it is better to use another clause evaluation engine. See setting 'clause_evaluation' for the possible options and trade-offs.

max_uncompressive_examples (Default=20) Maximum number of uncompressive examples (or negative score if other scoring function is being used) allowed before stopping the search and computing the current best theory. This setting is only applicable if 'theory_construction'=incremental.

minacc (Default=0) Minimum percentage accuracy a clause has to have on the training data to be considered a valid hypothesis.

mincov (Default=0) Minimum percentage of the positive examples a clause has to cover to be considered a valid hypothesis.

minimum_singletons_in_clause (Default=0) Minimum number of singleton variables (i.e. variables which just occur once) in a clause. This is a TopLog-specific setting.

minpos (Default=0) Minimum weight of positive examples a clause has to cover to be considered a valid hypothesis.

minprec (Default=0) Minimum percentage of corrected predicted positive examples a clause has to have to be considered a valid hypothesis.

negative_example_inflation (Default=1) Multiplies the weights of all negative examples by this factor. See also *example_inflation* and *positive_example_inflation*.

negative_reduction_measure (Default=precision) This is a ProGolem-specific setting. It defines which metric to maximize when performing negative reduction (see Section 4.3.4). The aim of negative reduction is to generalize a clause by keeping only the literals that block (i.e. prevent) negative examples from being proved. In the ILP systems Golem [MF92] and QG/GA [MTN07] only consistency negative reduction is performed. In 'consistency' mode it is ensured that the reduced clause entails no more negative examples than the original clause. However, this restriction may be too strict, as allowing a small extra negative coverage may significantly improve other clause evaluation metrics (e.g. compression, precision). All the possible scoring functions for *evalfn* are also accepted here. The most relevant possible values are:

- 'auto': maximize the same metric as defined by *evalfn*.
- 'consistency': the negatively reduced clause cannot entail more negative examples than the non-reduced clause.
- 'compression': maximize compression of the reduced clause
- 'precision': maximize coverage of the reduced clause

noise (Default=0.5) Maximum percentage of negative weights a clause may cover to still be considered a valid hypothesis.

nodes (Default=5000) Maximum number of hypotheses that may be derived by a single positive example. This setting is TopLog specific.

output_theory_file (Default='theory.pl') Filename where the induced theory is written. If this file already exists, it will be overwritten.

positive_example_inflation (Default=1) Multiplies the weights of all positive examples by this factor. See also *example_inflation* and *negative_example_inflation*.

print (Default=4) This setting controls the pretty printing of clauses to the stdout. It specifies the number of literals to be displayed per line when showing a clause.

progolem_beam_width (Default=3) Number of clauses (ARMGs) to carry forward to the next iteration of ProGolem's search. See Section 4.3.2 for more information. The bigger the beam-width the more likely it is that a better hypothesis will be found from a given example. However, the search will also take longer. The number of ARMGs that are constructed in each iteration of the search is $beam - width * iteration - sample - size$. See also *progolem_iteration_sample_size*. This setting is ProGolem specific.

progolem_iteration_sample_size (Default=20) Number of examples to randomly select to extend the $beam - width$ ARMGs of the current iteration. See Section 4.3.2 for more information. As with the beam-width, the bigger the iteration sample size, the more likely it is that a better hypothesis will be found from a given example. However, the search will also take longer. The number of ARMGs that are constructed in each iteration of the search is $beam - width * iteration - sample - size$. See also *progolem_beam_width*. This setting is ProGolem specific.

progolem_mode (Default=single) This setting controls the behaviour of the ProGolem algorithm. Possible values are:

- 'single': this is the default behaviour of ProGolem as explained in Section 4.3.2.
- 'pairs': *progolem_iteration_sample_size* pairs of randomly selected positive examples are constructed and the *progolem_beam_width* best are selected as seeds for the next

iteration. This is Golem's mode and is more efficient than 'single' but cannot be used with *theory_construction* = 'global'.

- 'reduce': no ARMGs are generated. The hypothesis generated from an example is the negative reduction of the most-specific clause of that example.

random_seed (Default=7) This is an integer specifying the random seed to be used by the ILP system. If the seed is the same across runs, the same numbers will be generated by the random number generators and results can be reproduced.

randomize_recall (Default=false) This setting impacts the construction of the most-specific clause. For a given mode body declaration,*modeb*, if 'false' the atoms that will be added to the body of the most-specific clause are the first N matches of the *modeb* declaration against the background knowledge.

If *randomize_recall*='true' the N solutions are randomly selected from the set of all possible matches of the *modeb* declaration. This may be useful when the dataset has a degree of non-determinism higher than the *star_default_recall* and we do not want to introduce a bias to favour the first matches.

In ILP systems which do not have this setting, e.g. Aleph and Progol, it is possible to emulate it by shuffling the background knowledge file.

recall_bound_on_evaluation (Default=inf) This is an experimental setting. When evaluating a literal in a clause, only the first *recall_bound_on_evaluation* solutions for any literal are considered. The default value of 'inf' considers all the solutions, as is expected from Prolog semantics. A lower value would only consider the first solutions, which could wrongly conclude that a given clause does not cover an example when it does. Note that this setting is unrelated to *star_default_recall*.

remove_negatives (Default=false) This setting is only applicable when *theory_construction* = 'incremental'. If set to true, when asserting a new hypothesis to the theory, and in addition to remove the positive examples covered by this hypothesis clause, the negative examples the hypothesis covers are also removed.

sample (Default=1.0) This is a real number between 0.0 and 1.0 specifying the approximate percentage of the user-supplied examples (both positive and negative) to be used by the ILP system. In order to speed-up the learning in datasets where there are too many examples, it may be useful to use a small fraction of the total examples. Ideally the ILP system should already do some form of sample coverage and that is planned for a future version of GILPS (see Section 7.1.1).

smart_coverage (Default=true) If 'true' and the coverage of a prefix of the clause under evaluation is available, then the coverage of the clause under evaluation is computed on the subset of examples that its longest prefix clause covers. This setting speeds up coverage test considerably as examples not covered by a prefix of a clause are guaranteed not to be covered. This setting increases the memory footprint, however. Currently only TopLog is able to take advantage of *smart_coverage*.

star_default_recall (Default=10) The integer value specifying the recall to which a '*' ought to correspond in a modeb definition. The recall setting is an important setting that is used in the construction of the most-specific clause of an example. A higher recall implies a larger hypothesis space. See Section C.1 for further details on how recall is used in the construction of the most-specific clause.

theory_construction (Default=global) In 'global' construction mode the theory is constructed after all hypotheses have been generated. See Section B.2 for further details on how the final theory is constructed in 'global' mode. The other possibility is 'incremental' construction. In 'incremental' mode there is a loop where the best hypothesis generated from an example is asserted to the final theory and all the positive examples this hypothesis covers are retracted. Incremental theory construction therefore requires fewer CPU resources than global coverage but is example order dependent and may lead to weaker theories. This is exemplified in Section D.1. Aleph [Sri07] and Progol [Mug95b] only allow 'incremental' theory construction.

verbose (Default=1) An integer >= 0 controlling the verbosity of GILPS. The higher the verbose level, the more information is shown.

A.5 Sample problem

This section presents a complete, yet simple, problem to bring together the concepts presented so far. In order to run GILPS there must be at least two Prolog files.

The first file (e.g. 'sample.pl') defines, or loads files that define, the learning problem, i.e. mode declarations, background knowledge and examples. Figure A.2 shows the Prolog file that defines, in GILPS format, the mode declarations, background knowledges and examples for a sample problem. This is the same problem used to show the example order relevance issue with Aleph and Progol in Appendix D.1.

```
:-modeh(1, e(+int)). :-modeb(1, b(+int)). :-modeb(1, c(+int)).

b(1). b(2). c(2). b(3). c(3). c(4). c(5). b(6).

example(e(1),1). example(e(2),1). example(e(3),1).
example(e(4),1). example(e(5),1).

example(e(6),-1). example(e(7),-1). example(e(8),-1).
```

Figure A.1: Sample background knowledge, mode declarations and examples for a problem in GILPS

The second file (e.g. 'run.pl') is a simple Prolog program to load GILPS itself, read the problem file and build the theory.

```
:- ['GILPS/gilps.pl'].
:- read_problem('sample.pl').
%:- set(engine, toplog).
:- build_theory.
```

Figure A.2: Script to run GILPS on the sample program of Figure A.1

The settings to change GILPS behaviour can be defined in either file. If they are defined in the execution file they must be executed before the call to *build_theory*. Assuming YAP is available, the program can then be executed by typing in the command line: *yap -L run.pl*.

We remind the reader that a set of more interesting problems is available at GILPS' webpage: http://ilp.doc.ic.ac.uk/GILPS.

A.6 Final Theory and Statistics

After constructing the theory with the command *build_theory/0*, irrespective of which particular engine was used to construct it, GILPS displays the final theory, its confusion matrix and a set of statistics measuring the quality of the induced theory.

For instance, the final theory generated for the protein-hexose binding problem of Chapter 6 is shown in Figure A.3.

```
Hypothesis 1/2:
#Literals=6, PosScore=37 (37 new), NegScore=4 (4 new) Prec=90.2% (90.2% new)
bind(A):-
   has_aminoacid(A,B,asp), atom_to_center_dist(B,'CG',5.4,0.5),
   has_aminoacid(A,C,asn), has_aminoacid(A,D,asn), diff_aminoacid(D,C).

Hypothesis 2/2:
#Literals=5, PosScore=30 (22 new), NegScore=1 (1 new) Prec=96.8% (95.7% new)
bind(A):-
   has_aminoacid(A,B,glu), atom_to_atom_dist(B,B,'N','CB',2.4,0.5),
   atom_to_center_dist(B,'CD',5.7,0.5), atom_to_center_dist(B,'CB',7.8,0.5).
```

Figure A.3: Final theory induced for the protein-hexose binding dataset with the aminoacid representation. See Chapter 6.

Notice that, for each hypothesis, GILPS identifies how many of the positive and negative examples it covers and how many of them are new when taking into account the examples covered by the rules before. For instance, the second hypothesis covers 30 positive examples; of these 30, 22 are novel and 8 were previously covered by the first hypothesis. The confusion matrix and statistics measures about this theory are shown in Figure A.4.

If cross-validation is enabled, a similar theory is constructed N times, once for each fold. An average of this n-fold cross-validation is also shown, together with the respective standard deviations, as in Figure A.5.

```
************************************************************
* Train theory statistics (using all examples as training) *
************************************************************

           |                 Actual                |
 Predicted |           Positive|           Negative|             Totals
-----------|-------------------|-------------------|-------------------
   Positive|             59+/-0|              5+/-0|             64+/-0
-----------|-------------------|-------------------|-------------------
   Negative|             21+/-0|             75+/-0|             96+/-0
-----------|-------------------|-------------------|-------------------
 Totals    |             80+/-0|             80+/-0|            160+/-0

Default accuracy: 50% +/-0.0%
Classifier accuracy: 83.8% +/-0.0%
Recall/Sensitivity: 73.8% +/-0.0% (% of correctly class. positive examples)
Specificity: 93.8% +/-0.0% (% of correctly class. negative examples)
Precision: 92.2% +/-0.0% (% of correctly predicted positive examples)
CorPredNeg: 78.1% +/-0.0% (i.e. % of correctly predicted negative examples)
F1-score: 0.819 +/-0.00 (i.e. 2*Precision*Recall/(Precision+Recall)
Matthews correlation: 0.689 +/-0.00
```

Figure A.4: Confusion matrix and statistical measures for the performance of the induced theory on the training set

```
*********************************
* Average test theory statistics *
*********************************

           |                 Actual                |
 Predicted |           Positive|           Negative|             Totals
-----------|-------------------|-------------------|-------------------
   Positive|              6+/-1|              0+/-1|              6+/-2
-----------|-------------------|-------------------|-------------------
   Negative|              2+/-1|              8+/-1|             10+/-2
-----------|-------------------|-------------------|-------------------
 Totals    |              8+/-2|              8+/-1|             16+/-3

Default accuracy: 50% +/-0.0%
Classifier accuracy: 83.1% +/-6.6%
Recall/Sensitivity: 72.5% +/-12.9% (% of correctly class. positive examples)
Specificity: 93.8% +/-8.8% (% of correctly class. negative examples)
Precision: 93.3% +/-9.0% (% of correctly predicted positive examples)
CorPredNeg: 78.1% +/-7.8% (i.e. % of correctly predicted negative examples)
F1-score: 0.807 +/-0.09 (i.e. 2*Precision*Recall/(Precision+Recall)
Matthews correlation: 0.687 +/-0.12
```

Figure A.5: Confusion matrix and statistical measures for the average performance of the induced theory on the test set of all folds