The Alchemy System for Statistical Relational AI: Developer's Manual

Stanley Kok Parag Singla Matthew Richardson
Pedro Domingos Marc Sumner
Hoifung Poon
Department of Computer Science and Engineering, University of Washington

Jan 15, 2007

1 Introduction

Welcome to the Alchemy developer's manual. This is designed to help developers improve and extend Markov logic algorithms in Alchemy. We have strived to make Alchemy as modular as possible in order to encourage further development of the code. This effort is ongoing and it should be noted that Alchemy is still in a Beta stage.

This manual, along with the API provided in the package and on the website, should enable other developers to utilize the Alchemy classes in their own applications as well as allow them to extend Alchemy itself.

The development of Alchemy was partly funded by DARPA grant FA8750-05-2-0283 (managed by AFRL), DARPA contract NBCH-D030010 (subcontracts 02-000225 and 55-000793), NSF grant IIS-0534881, ONR grants N00014-02-1-0408 and N00014-05-1-0313, a Sloan Research Fellowship, and an NSF CAREER Award (both of these to Pedro Domingos). The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, NSF, ONR, or the United States Government.

2 Notes on Code Design

The C++ source code found in `ALCHDIR/src` is divided into six directories: `util/`, `parser/`, `logic/`, `learnwts/`, `learnstruct/` and `infer/`. Most of the code is found in `.h` files for convenient inlining. We avoided the use of polymorphism as much as possible, since virtual functions are not inlined and we would like to have as much inlining as possible for the code to run quickly. Most of the `.h` files have names that are the same as those of the classes they contain.

2.1 Utilities

The `util` directory contains "utility" classes. `Argument` is a class used to parse command line arguments. `Array` is a template class representing an array, and is used widely in the code. `HashArray` is similar to an `Array` except that it is backed up by a map so that its elements are unique. `HashList` is similar to a `HashArray` except that it is a list implementation. In `hashint.h` and `hashstring.h` are the definitions of `HashArrays` containing `int`s and `string`s. `ArraysAccessor` allows you to iterate through all combinations of items in several arrays. Both `DualMap` and `ConstDualMap` map `int`s to `string`s and vice versa. They are mainly used by `Domain` in `logic/` to hold predicates, types etc. `StrInt` is a data structure used by `DualMap` and `ConstDualMap`. `MeanVariance` is used to compute the mean and variance of a set of numbers. `MultDArray` represents a multi-dimensional array. `PowerSet` generates the powerset of {`0...n`} except the null set. `Timer` measures user time in seconds, and contains a function to print time. `util.h` is used to contain commonly used functions that can be shared across modules. `Random` is a random number generator.

2.2 Parser

In the `parser/` directory, `follex.y` and `fol.y` are the input files for Flex (lexical analyzer) and Bison (parser generator) respectively. `fol.y` contains the grammar rules that are used to parse first-order logic formulas, and the code that fires when each rule is encountered. `folhelper.h` contains our variables and functions that are used in `fol.y` and `follex.y`. All Flex and Bison variables and functions begin with the characters `yy`. Using a similar convention, all of our variables and functions that are used in `follex.y`, `fol.y` and `folhelper.h` begin with `zz`. The main function is `runYYParser()` that parses a `.mln` file and creates an `MLN` and `Domain` (see Section 2.3 below). If you want to add variables to be used in `fol.y` or `folhelper.h`, please see the note at the top of `folhelper.h`. You can also change the default weights given to hard clauses by setting `HARD_WEIGHT_MULTIPLIER`/`HARD_WEIGHT` at the top of `folhelper.h`. `StrFifoList` is a list used in `fol.y` to hold tokens in the order that they are extracted by Flex. `ListObj` contains the algorithm to convert a first-order formula to CNF. It approximates lisp in its use of lists to represent a prefix form of first-order logic. `replacefolcpp.pl` is a perl script that replaces certain code in `fol.cpp` (generated by Bison from `fol.y`) so that it is C++ compliant. If you are using a version of Bison that is less than 2.0, you may have to uncomment the lines at the bottom of the file. For debugging purposes, you can set the variables `follexDbg` (in `follex.y`) and `folDbg` (in `fol.y`) to see the order in which tokens are extracted, as well as the order in which the grammar rules are executed.

2.3 Logic

The `logic/` directory contains classes related to first-order logic. `PredicateTemplate` represents a predicate declaration, while a `Predicate` is its definition. Likewise for `FunctionTemplate` and `Function`. Observe that the code for `Predicate` and `Function` is similar, and we could

have made one the superclass of the other. However, we avoided polymorphism for the sake of inlining their functions. A `Term` represents a constant, a variable or a function. A `Predicate` contains one or more `Terms`. `Clause` is an array of `Predicate`s, and contains the important functions for counting the number of true groundings of a clause, and for finding unknown ground clauses. `ClauseFactory` creates clauses for structure learning. It includes a function `validClause()` in which you can specify rules to restrict the kinds of clauses created. `ClauseSampler` contains an algorithm that estimates the number of true groundings of a clause by sampling the clause's groundings. It uses `TrueFalseGroundingsStore` to store groundings of predicates. `MLN` represents a `Markov Logic Network` with a set of `Clauses`. `clausehelper.h` and `mlnhelper.h` contains the auxiliary data structures used by `Clause` and `MLN` respectively. `Database` provides the truth values of ground `Predicate`s (ground atoms), and keeps the truth values of all ground atoms in memory. `GroundPreds` is a data structure for holding ground `Predicate`s, and is mainly used for testing purposes. A `Domain` contains the declared types, constants, predicates, and functions, and provides information about them (e.g., the number of constants of a type). It also holds a pointer to a `Database`. The class `VariableState` represents the state of all predicates and clauses while performing learning or inference. Besides holding all ground predicates and clauses generated from its `MLN` and `domain`, it contains many data structures and indices which allow fast access to information about the state. It encapsulates the eagerness or laziness of the state (i.e. if all ground clauses are built upfront or just when needed). This allows us to implement inference and learning algorithms based on this state without worrying about the differences in a lazy and an eager implementation.

2.4 Weight Learning

The `learnwts` directory contains code for learning the weights of formulas. The mainline is in `learnwts.cpp`. Table 1 shows the options available when calling learnwts.

If you do not want to print the clauses as their number of true groundings are being counted during generative learning, you can set the variable `PRINT_CLAUSE_DURING_COUNT` to false at the top of `learnwts.cpp`. `learnwts.h` contains functions used in `learnwts.cpp` that can be shared with other modules. `PseudoLogLikelihood` computes the (weighted) pseudo-log-likelihood given the constants in one or more `Domains`, and clauses in an `MLN`. `LBFGSB` is an optimization routine that finds the optimal weights, i.e., the weights that give the highest (weighted) pseudo-log-likelihood. `DiscriminativeLearner` contains the various algorithms for discriminative learning, currently Voted Percptron, Conjugate Gradient and Newton's Method. `IndexTranslator` is used to translate between clause weights and the weights that are optimized. It is required when the CNF of a formula is different across multiple databases, e.g., when the formula has existentially quantified variables, or variables with mutually exclusive and exhaustive values.

<-i <string>>	Comma-separated input .mln files. (With the -multipleDatabases option, the second file to the last one are used to contain constants from different databases, and they correspond to the .db files specified with the -t option.)
[-cw <string>]	Specified non-evidence atoms (comma-separated with no space) are closed world, otherwise, all non-evidence atoms are open world. Atoms appearing here cannot be query atoms and cannot appear in the -o option.
[-ow <string>]	Specified evidence atoms (comma-separated with no space) are open world, while other evidence atoms are closed-world. Atoms appearing here cannot appear in the -c option.
[-infer <string>]	Specified inference parameters when using discriminative learning. The arguments are to be encapsulated in "" and the syntax is identical to the infer command (run infer with no commands to see this). If not specified, MaxWalkSat with default parameters is used.
[-d [bool]]	Discriminative weight learning.
[-g [bool]]	Generative weight learning.
<-o <string>>	Output .mln file containing formulas with learned weights.
<-t <string>>	Comma-separated .db files containing the training database (of true/false ground atoms), including function definitions, e.g. ai.db,graphics.db,languages.db.
[-ne <string>]	First-order non-evidence predicates (comma-separated with no space), e.g., cancer,smokes,friends. For discriminative learning, at least one non-evidence predicate must be specified. For generative learning, the specified predicates are included in the (weighted) pseudo-log-likelihood computation; if none are specified, all are included.
[-noAddUnitClauses [bool]]	If specified, unit clauses are not included in the .mln file; otherwise they are included.
[-multipleDatabases [bool]]	If specified, each .db file belongs to a separate database; otherwise all .db files belong to the same database.
[-withEM [bool]]	If set, EM is used to fill in missing truth values; otherwise missing truth values are set to false.
[-dNumIter <integer>]	[100] (For discriminative learning only.) Number of iterations to run voted perceptron.
[-dLearningRate <double>]	[0.001] (For discriminative learning only) Learning rate for the gradient descent in voted perceptron algorithm.
[-dMomentum <double>]	[0.0] (For discriminative learning only) Momentum term for the gradient descent in voted perceptron algorithm.

[-queryEvidence [bool]]	If this flag is set, then all the groundings of query preds not in db are assumed false evidence.
[-dRescale [bool]]	(For discriminative learning only.) Rescale the gradient by the number of true groundings per weight.
[-dZeroInit [bool]]	(For discriminative learning only.) Initialize clause weights to zero instead of their log odds.
[-gMaxIter <integer>]	[10000] (For generative learning only.) Max number of iterations to run L-BFGS-B, the optimization algorithm for generative learning.
[-gConvThresh <double>]	[1e-5] (For generative learning only.) Fractional change in pseudo-log-likelihood at which L-BFGS-B terminates.
[-gNoEqualPredWt [bool]]	(For generative learning only.) If specified, the predicates are not weighted equally in the pseudo-log-likelihood computation; otherwise they are.
[-noPrior [bool]]	No Gaussian priors on formula weights.
[-priorMean <double>]	[0] Means of Gaussian priors on formula weights. By default, for each formula, it is the weight given in the .mln input file, or fraction thereof if the formula turns into multiple clauses. This mean applies if no weight is given in the .mln file.
[-priorStdDev <double>]	[1 for discriminative learning. 100 for generative learning] Standard deviations of Gaussian priors on clause weights.
[-dMaxSec <double>]	[-1] Maximum number of seconds to spend learning
[-dMaxMin <double>]	[-1] Maximum number of minutes to spend learning
[-dMaxHour <double>]	[-1] Maximum number of hours to spend learning
[-dPW [bool]]	[false] (For voted perceptron only.) Per-weight learning rates, based on the number of true groundings per weight.
[-dVP [bool]]	[false] (For discriminative learning only) Use voted perceptron to learn the weights.
[-dNewton [bool]]	[false] (For discriminative learning only) Use diagonalized Newton's method to learn the weights.
[-dCG [bool]]	[true] (For discriminative learning only) Use rescaled conjugate gradient to learn the weights.
[-cgLambda <double>]	[100] (For CG only) parameter to limit step size
[-cgPrecond [bool]]	[true] (For CG only) precondition with the diagonal Hessian

Table 1: Command line options for learnwts

2.5 Structure Learning

The `learnstruct/` directory contains code for the generative learning of MLN structure. The mainline is in `learnstruct.cpp`. Table 2 shows the options available when calling learnstruct.

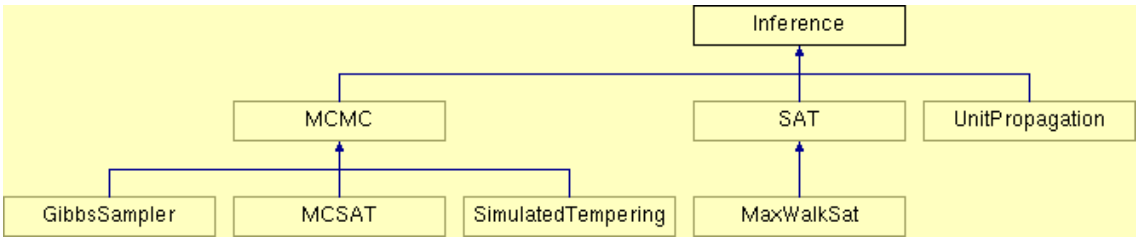
`structlearn.h` contains most of the structure learning code. `structlearn.cpp` contains the code that handles formulas with variables that are existentially quantified, or have mutually exclusive and exhaustive values.

2.6 Inference

The `infer/` directory contains code for performing inference. The mainline is in `infer.cpp`. Table 3 shows the options available when calling infer.

`infer.h` contains functions used in `infer.cpp` that can be shared with other modules. `GroundPredicate` and `GroundClause` are the counterparts of `Predicate` and `Clause` in `logic/`. We created separate classes for inference in order to save space since most of the instance variables in `Predicate` and `Clause` are not needed during inference, and inference requires us to ground the MLN to create a Markov random field that may take up a lot of memory. `MRF` represents the Markov random field and contains the code for Gibbs sampling. `GelmanConvergenceTest` is used to determine convergence during burn-in, and `ConvergenceTest` is used to determine convergence during Gibbs sampling.

All inference algorithms in Alchemy are implemented as subclasses of the abstract class `Inference`. Currently, the class hierarchy contains two large classes of inference algorithms: SAT-solvers and MCMC algorithms. These two classes hold the parameters which are common among all flavors of SAT-solvers and MCMC algorithms, respectively. `SAT` and `MCMC` are also implemented as abstract classes as they should only serve as superclasses for various implementations.



<-i <string>>	Comma-separated input .mln files. (With the -multipleDatabases option, the second file to the last one are used to contain constants from different domains, and they correspond to the .db files specified with the -t option.)
<-o <string>>	Output .mln file containing learned formulas and weights.
<-t <string>>	Comma-separated .db files containing the training database (of true/false ground atoms), including function definitions, e.g. ai.db,graphics.db,languages.db.
[-ne <string>]	[all predicates] Non-evidence predicates (comma-separated with no space), e.g., cancer,smokes,friends.
[-multipleDatabases [bool]]	If specified, each .db file belongs to a separate domain; otherwise all .db files belong to the same domain.
[-beamSize <integer>]	[5] Size of beam in beam search.
[-minWt <double>]	[0.01] Candidate clauses are discarded if their absolute weights fall below this.
[-penalty <double>]	[0.01] Each difference between the current and previous version of a candidate clause penalizes the (weighted) pseudo-log-likelihood by this amount.
[-maxVars <integer>]	[6] Maximum number of variables in learned clauses.
[-maxNumPredicates <integer>]	[6] Maximum number of predicates in learned clauses.
[-cacheSize <integer>]	[500] Size in megabytes of the cache that is used to store the clauses (and their counts) that are created during structure learning.
[-noSampleClauses [bool]]	If specified, compute a clause's number of true groundings exactly, and do not estimate it by sampling its groundings. If not specified, estimate the number by sampling.
[-delta <double>]	[0.05] (Used only if sampling clauses.) The probability that an estimate a clause's number of true groundings is off by more than epsilon error is less than this value. Used to determine the number of samples of the clause's groundings to draw.
[-epsilon <double>]	[0.2] (Used only if sampling clauses.) Fractional error from a clause's actual number of true groundings. Used to determine the number of samples of the clause's groundings to draw.
[-minClauseSamples <integer>]	[-1] (Used only if sampling clauses.) Minimum number of samples of a clause's groundings to draw. (-1: no minimum)
[-maxClauseSamples <integer>]	[-1] (Used only if sampling clauses.) Maximum number of samples of a clause's groundings to draw. (-1: no maximum)

[-noSampleAtoms [bool]]	If specified, do not estimate the (weighted) pseudo-log-likelihood by sampling ground atoms; otherwise, estimate the value by sampling.
[-fractAtoms <double>]	[0.8] (Used only if sampling ground atoms.) Fraction of each predicate's ground atoms to draw.
[-minAtomSamples <integer>]	[-1] (Used only if sampling ground atoms.) Minimum number of each predicate's ground atoms to draw. (-1: no minimum)
[-maxAtomSamples <integer>]	[-1] (Used only if sampling ground atoms.) Maximum number of each predicate's ground atoms to draw. (-1: no maximum)
[-noPrior [bool]]	No Gaussian priors on formula weights.
[-priorMean <double>]	[0] Means of Gaussian priors on formula weights. By default, for each formula, it is the weight given in the .mln input file, or fraction thereof if the formula turns into multiple clauses. This mean applies if no weight is given in the .mln file.
[-priorStdDev <double>]	[100] Standard deviations of Gaussian priors on clause weights.
[-tightMaxIter <integer>]	[10000] Max number of iterations to run L-BFGS-B, the algorithm used to optimize the (weighted) pseudo-log-likelihood.
[-tightConvThresh <double>]	[1e-5] Fractional change in (weighted) pseudo-log-likelihood at which L-BFGS-B terminates.
[-looseMaxIter <integer>]	[10] Max number of iterations to run L-BFGS-B when evaluating candidate clauses.
[-looseConvThresh <double>]	[1e-3] Fractional change in (weighted) pseudo-log-likelihood at which L-BFGS-B terminates when evaluating candidate clauses.
[-numClausesReEval <integer>]	[10] Keep this number of candidate clauses with the highest estimated scores, and re-evaluate their scores precisely.
[-noWtPredsEqually [bool]]	If specified, each predicate is not weighted equally. This means that high-arity predicates contribute more to the pseudo-log-likelihood than low-arity ones. If not specified, each predicate is given equal weight in the weighted pseudo-log-likelihood.
[-startFromEmptyMLN [bool]]	If specified, start structure learning from an empty MLN. If the input .mln contains formulas, they will be added to the candidate clauses created in the first step of beam search. If not specified, begin structure learning from the input .mln file.

[-tryAllFlips [bool]]	If specified, the structure learning algorithm tries to flip the predicate signs of the formulas in the input .mln file in all possible ways
[-bestGainUnchangedLimit <integer>]	[2] Beam search stops when the best clause found does not change in this number of iterations.

Table 2: Command line options for learnstruct

All inference algorithms are based on a `VariableState` (see dir /src/logic/) which encodes the state of the propositional variables and clauses. There are two different methods to build the state: lazily and eagerly. An eager state builds a Markov random field based on the queries, the MLN and the domain of constants. Inference is then run on the clauses and variables in the MRF. A lazy state makes the assumption of all variables being false in the beginning and activates variables and clauses as needed by the inference algorithm. In sparse domains, this can lead to large savings in memory usage. The laziness or eagerness of a state is encapsulated in the class `VariableState` and is set with the constructor.

In addition, all inference algorithms can be instantiated with a seed for the random number generator, if needed. If the algorithm contains no randomness, this is ignored. The ability to set the seed is useful when debugging and comparing different parameter settings of an algorithm.

2.6.1 Implementing a New Inference Algorithm

Any new inference algorithm in Alchemy must fit into the existing class hierarchy (i.e. it must be a subclass of `Inference`). Therefore, it must implement the methods `init()`, `infer()`, `printProbabilities`, `printTruePreds()` and `getProbability()`, although it could be that no initialization is required (see, for example, `UnitPropagation`). The constructor of the new class should call the constructor of `Inference` so that the state and seed are initialized.

Every inference algorithm is called in the same manner in infer/infer.cpp. A `VariableState` is initialized and the pointer `inference` is set to the inference algorithm as specified on the command line. If a new inference algorithm is added, this should be extended (along with the command line options) to accomodate the new algorithm, i.e.: `inference = new NewAlgorithm(state, seed, params);` where `params` is a `struct` to hold the parameters specific to the new algorithm. Finally, `init()` and `infer()` are called which perform the inference and the probabilities (or best state) of the ground atoms are output to file by calling `printProbabilities()`. Of course, the header file of the new algorithm must be included in infer.cpp. The laziness or eagerness of the inference algorithm should be encapsulated in `VariableState`.

<-i <string>>	Comma-separated input .mln files.
[-cw <string>]	Specified non-evidence atoms (comma-separated with no space) are closed world, otherwise, all non-evidence atoms are open world. Atoms appearing here cannot be query atoms and cannot appear in the -o option.
[-ow <string>]	Specified evidence atoms (comma-separated with no space) are open world, while other evidence atoms are closed-world. Atoms appearing here cannot appear in the -c option.
[-m [bool]]	Run MAP inference and return only positive query atoms.
[-a [bool]]	Run MAP inference and show 0/1 results for all query atoms.
[-p [bool]]	Run inference using MCMC (Gibbs sampling) and return probabilities for all query atoms.
[-ms [bool]]	Run inference using MC-SAT and return probabilities for all query atoms
[-simtp [bool]]	Run inference using simulated tempering and return probabilities for all query atoms
[-seed <integer>]	[random] Seed used to initialize the randomizer in the inference algorithm. If not set, seed is initialized from the current date and time.
[-lazy [bool]]	[false] Run lazy version of inference if this flag is set.
[-lazyNoApprox [bool]]	[false] Lazy version of inference will not approximate by deactivating atoms to save memory. This flag is ignored if -lazy is not set.
[-memLimit <integer>]	[-1] Maximum limit in kbytes which should be used for inference. -1 means main memory available on system is used.
[-mwsMaxSteps <integer>]	[1000000] (MaxWalkSat) The max number of steps taken.
[-tries <integer>]	[1] (MaxWalkSat) The max number of attempts taken to find a solution.
[-targetWt <integer>]	[the best possible] (MaxWalkSat) MaxWalkSat tries to find a solution with weight ¡= specified weight.
[-hard [bool]]	[false] (MaxWalkSat) MaxWalkSat never breaks a hard clause in order to satisfy a soft one.
[-heuristic <integer>]	[1] (MaxWalkSat) Heuristic used in MaxWalkSat (0 = RANDOM, 1 = BEST, 2 = TABU, 3 = SAMPLESAT).
[-tabuLength <integer>]	[5] (MaxWalkSat) Minimum number of flips between flipping the same atom when using the tabu heuristic in MaxWalkSat.

[-lazyLowState [bool]]	[false] (MaxWalkSat) If false, the naive way of saving low states (each time a low state is found, the whole state is saved) is used; otherwise, a list of variables flipped since the last low state is kept and the low state is reconstructed. This can be much faster for very large data sets.
[-burnMinSteps <integer>]	[100] (MCMC) Minimun number of burn in steps (-1: no minimum).
[-burnMaxSteps <integer>]	[100] (MCMC) Maximum number of burn-in steps (-1: no maximum).
[-minSteps <integer>]	[-1] (MCMC) Minimum number of Gibbs sampling steps.
[-maxSteps <integer>]	[1000] (MCMC) Maximum number of Gibbs sampling steps.
[-maxSeconds <integer>]	[-1] (MCMC) Max number of seconds to run MCMC (-1: no maximum).
[-subInterval <integer>]	[2] (Simulated Tempering) Selection interval between swap attempts
[-numRuns <integer>]	[3] (Simulated Tempering) Number of simulated tempering runs
[-numSwap <integer>]	[10] (Simulated Tempering) Number of swapping chains
[-numStepsEveryMCSat <integer>]	[1] (MC-SAT) Number of total steps (mcsat + gibbs) for every mcsat step
[-numSolutions <integer>]	[10] (MC-SAT) Return nth SAT solution in SampleSat
[-saRatio <integer>]	[50] (MC-SAT) Ratio of sim. annealing steps mixed with WalkSAT in MC-SAT
[-saTemperature <integer>]	[10] (MC-SAT) Temperature (/100) for sim. annealing step in SampleSat
[-lateSa [bool]]	[false] Run simulated annealing from the start in SampleSat
[-numChains <integer>]	[10] (Gibbs) Number of MCMC chains for Gibbs sampling (there must be at least 2).
[-delta <double>]	[0.05] (Gibbs) During Gibbs sampling, probabilty that epsilon error is exceeded is less than this value.
[-epsilonError <double>]	[0.01] (Gibbs) Fractional error from true probability.
[-fracConverged <double>]	[0.95] (Gibbs) Fraction of ground atoms with probabilities that have converged.
[-walksatType <integer>]	[1] (Gibbs) Use Max Walksat to initialize ground atoms' truth values in Gibbs sampling (1: use Max Walksat, 0: random initialization).
[-samplesPerTest <integer>]	[100] Perform convergence test once after this many number of samples per chain.
<-e <string>>	Comma-separated .db files containing known ground atoms (evidence), including function definitions.
<-r <string>>	The probability estimates are written to this file.

[-q <string>]	Query atoms (comma-separated with no space) ,e.g., cancer,smokes(x),friends(Stan,x). Query atoms are always open world.
[-f <string>]	A .db file containing ground query atoms, which are are always open world.

Table 3: Command line options for infer

3 Online Alchemy

For many applications, the end user is not interested in performing learning and/or inference once in batch mode, but rather requires this for many time steps in an online mode. The class `OnlineEngine` in the directory `online` addresses this issue by performing online inference and (coming soon) learning. The class is designed to be used by an agent which initializes the engine with evidence and an MLN. In subsequent time steps, the agent adds, changes and deletes evidence and query atoms.

3.1 OnlineEngine

The interface of `OnlineEngine` which the agent can utilize can be found in the Alchemy API. In this section, we explain, using a small example, how to use the API to perform online inference.

An `OnlineEngine` is constructed based on a string containing the options for the underlying inference. The form of the string is the same as the options available in the batch mode executable infer (see Section 2.6). A simple example of an agent exists in the file `online.cpp` in the `online` directory.

After constructing the `OnlineEngine`, the agent should call the `init()` method which initializes the underlying inference procedure. Then, in each time step, the agent asks the `OnlineEngine` to perform inference and return the true atoms (if performing MAP inference) or the atoms with non-zero probability (if performing probabilistic inference). This is achieved with the `infer()` method. The agent can then change the evidence and/or query predicates and inform the `OnlineEngine` about these changes with the methods `addTrueEvidence()`, `addFalseEvidence()` and `removeEvidence()`. Evidence to be removed or added needs to be in string form (as in a .db file) and put in a `vector` of with elements of type `string`.

For most agents, it is often the case that the state will not change drastically from one time step to the next. If this is the case, we don't want our inference engine to perform full inference in each iteration. When using MaxWalkSat as the underlying inference procedure, the `OnlineEngine` can vary the number of maximum steps which MaxWalkSat will perform. For time step n, inference begins in the state in which time step $n-1$ left off. Therefore, it is advisable that the agent reduce the maximum number of inference steps after the first time step by using the `setMaxInferenceSteps()` method. For probabilistic inference, this

option is not available.

References

[1] S. Kok, P. Singla, M. Richardson, and P. Domingos. The Alchemy system for statistical relational AI. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 2005. http://www.cs.washington.edu/ai/alchemy/.