PCE User Guide

Sam Owre <owre@csl.sri.com>

Revision History

Revision 1.0	July 2009	SO

Table of Contents

1. Introduction

The Probability Consistency Engine (PCE) is a system for performing probabilistic inference in Markov Logic using Markov Chain Monte Carlo (MCMC) methods. PCE uses the MCSat and Lazy MCSat algorithms and was originally developed as part of the CALO project, where it was used to provide marginal probabilities for the weighted assertions and rules provided by diverse machine learners and classifiers.

The PCE system currently has three different APIs. A future version will also support PCE as a library, allowing more direct control of the system. The most basic API is the standalone `MCSAT`, which runs as an interactive application taking commands from the user or files and providing feedback directly. The version used for CALO, referred to below as `OAAPCE`, is an Open Agent Architecture OAA agent, and assumes there are some specific CALO agents available. As OAA is a bit heavyweight with nonstandard datatypes, a new version was created, called `XPCE`. It is based on XML-RPC and uses JSON for the data encoding. We describe MCSAT and XPCE below, and describe OAAPCE in Appendix A.

2. PCE Description

As stated above, PCE uses the MCSAT and Lazy MCSAT methods to compute marginal probabilities based on Markov Chain Monte Carlo (MCMC) sampling. The object language of PCE consists of

sorts, atomic predicates, constants, and propositional formulas. Predicates are either *direct* or *indirect*, which is indicated when the predicate is declared. A direct predicate is one that is observable, e.g., that email was received from a specific email address, with a specific content. An indirect predicate is one that can only be inferred, e.g., that the email is associated with a particular project.

The direct atoms, i.e., atoms with direct predicates are only true when they have been added as hard facts. The closed world assumption applies, so unasserted direct atoms are treated as false. Given the truth assignment to the hard facts, the MCSAT procedures construct an initial model, i.e., an assignment of truth values to atoms, satisfying the hard facts. From this model, PCE generates a sequence of sampled models using MCMC sampling. In each step, clauses are selected that are satisfied in the current model. A subset of these clauses is chosen based on the weights so that clauses with higher weights are picked with higher probability. The sampling algorithm SampleSAT is then used to construct a random model of the chosen subset of clauses. This algorithm starts with a random truth assignment and interleaves *simulated annealing* with *WalkSAT* steps to converge on an assignment that satisfies all the chosen clauses. The simulated annealing step picks a random variable and flips its truth value if this reduces the weight of the unsatisfied clauses. With some probability, the flip is executed even when the weight is not reduced so as to move the search out of a local minimum. In the WalkSAT step, a random unsatisfied clause is chosen and the atom corresponding to a random literal is flipped. The algorithm implemented in PCE is an online variant of the one used by the Alchemy system. PCE also employs a lazy version of the algorithm that creates atoms only when they become relevant.

The input to PCE consists of sorts which are arranged in a subsort hierarchy, constants of specific sorts, direct and indirect predicates over these sorts, atomic facts asserting a direct predicate of specific constants, and rules which are universally quantified formulas in both the direct and indirect predicates along with their associated weights. PCE is then queried for formula patterns, new samples are run, and the formula instances are output, along with their marginal probabilities.

3. Obtaining and Executing PCE

Obtaining PCE. PCE is currently available only on request; contact owre@csl.sri.com [mailto:owre@csl.sri.com] for details.

Executing PCE. The PCE standalone MCSAT is associated with the `mcsat` executable, XPCE is associated with `xpce`, and OAAPCE with `pce`. These are available for Windows (under cygwin, with `.exe` extensions), MacOS X, and Linux. Below is the information needed to run each one.

4. MCSAT

The interactive system is called `mcsat` (or `mcsat.exe`) in the `bin` subdirectory. Commands are then entered at the `mcsat>` prompt, followed by a semicolon. To quit, enter `quit;`, and enter `help;` for a brief description of the available commands.

4.1. Executing `mcsat`

Usage:

```
mcsat [OPTION]... [FILE]...
```

Option		Effect
`-h -?`	`--help`	Print a summary of the options
`-i`	`--interactive`	Run interactively
`-s NUM`	`--seed=NUM`	Run with the given pseudorandom seed
	`--lazy=BOOL`	Use the lazy version (default `false`)
	`--strict=BOOL`	Require declarations for all constants (default `true`)
`-v NUM`	`--verbosity=NUM`	Sets the verbosity level
`-V`	`--version`	Prints the version number and exits

4.2. MCSAT Syntax

The commands that follow refer to the syntax described here. This is in EBNF notation, with the following conventions: - Terminals are in single quotes, and are case-insensitive - Parentheses are used for grouping, `(^` is used for complementation (i.e., all characters except for…) - `?`, `*`, and `+` may be used to indicate zero or one, zero or more, and one or more, respectively. If these are preceded by a separator (e.g., a comma), it means the entries are separated accordingly. Thus `nt := term,+` is equivalent to `nt := term | term ',' nt`, and `LITERAL|+` means any number of literals separated by vertical bars.

```
FORMULA := FMLA
         | '[' VAR,* ']' FMLA

FMLA := ATOM
      | FMLA 'iff' FMLA
      | FMLA 'implies' FMLA
      | FMLA 'or' FMLA
      | FMLA 'and' FMLA
      | NOT FMLA
      | '(' FMLA ')'

CLAUSE := LITERAL|+
        | '(' VAR,+ ')' LITERAL|+

LITERAL := ATOM | NOT ATOM

ATOM := PREDICATE '(' ARGUMENT,* ')'

VAR := NAME
PREDICATE := NAME
ARGUMENT := NAME | NUM
NOT := 'not' | '~'

STRING := Anything between double quotes: "a" or between single quotes: 'a'

NAME := ALPHA (^ DELIM | CNTRL | COMMENT )*
NUM := ( DIGIT | '.' | '+' | '-' ) ( DIGIT | '.' )*
       must contain at least on DIGIT, at most one '.'

ALPHA := 'a' | ... | 'z' | 'A' | ... | 'Z'
DIGIT := '0' | ... | '9'
DELIM := ' ' | '(' | ')' | '[' | ']' | '|' | ',' | ';' | ':'
CNTRL := non-printing (control) characters
```

```
COMMENT := '#' to end of line
```

4.3. MCSAT Commands

MCSAT has a simple syntax. Names (identifiers) start with a letter, followed by any character other than parentheses, commas, semicolon, colon, vertical bar, space, or control characters. Numbers are simple floating point numbers: an optional plus or minus, followed by digits and an optional decimal point. Whitespace is ignored.

Atoms are predicates applied to constants, e.g., `p(x, y, z)`. As described below, predicates have an arity and a signature. Literals are atoms or negated atoms, where the tilde "~" is used for negation. Clauses are disjunctions of literals, separated by vertical bars "`|`".

Commands are case-insensitive, and terminated with a semicolon.

4.4. MCSAT Commands

4.4.1. sort

MCSAT uses simple sorts - sorts are pairwise disjoint or subsorts (see below) and introduced with the `sort` or `subsort` command.

Syntax:

```
sort NAME;
```

Example:

```
sort File;
```

4.4.2. subsort

Subsorts may be declared in PCE. The effect of this is that every constant belonging to a subsort also belongs to the supersort. Subsorts are technically not needed, but can make the algorithm run faster.

Syntax:

```
subsort NAME NAME ;
```

Example:

```
subsort File Entity;
```

4.4.3. predicate

MCSAT supports predicates of any arity, but the sort signature must be given. MCSAT also makes a distinction between direct (observable), and indirect predicates. Internally, direct predicates satisfy the closed world assumption, and indirect predicates do not.

Syntax:

```
predicate NAME ( NAMES ) IND ;
```

where `IND` is "`direct`", "`indirect`", or omitted, defaulting to "`direct`".

Example:

```
predicate fileHasTask(File, Task) indirect;
```

4.4.4. const

Sorts are empty initially, the `const` command is used to introduce elements of a given sort.

Syntax:

```
const NAMES : NAME ;
```

Example:

```
const fi8, fi22, fi23: File
```

4.4.5. assert

Similar to atom, but used to introduce facts. Note that negative literals may not be asserted, and the predicate must be direct.

Syntax:

```
assert ATOM ;
```

Example:

```
assert fileHasTask(fi8, ta1);
```

4.4.6. add

Add is used to introduce weighted formulas and rules. Rules include variables, which are introduced before the formula.

Syntax

```
add FORMULA WT ;


add [ VARIABLES ] FORMULA WT ;
```

where `WT` is an optional floating point weight. If weight is missing, the clause or rule is considered as a "hard" clause or rule. (This is the same as having infinite weight).

Example

```
add fileHasTask(fi22, ta1) 1.286;
```

adds a weighted ground clause.

```
add [File, Email, Task]
        fileHasTask(File, Task) and attached(File, Email)
      implies emailHasTask(Email, Task);
```

adds a rule with infinite weight. This is essentially asserting the axiom

4.4.7. add_caluse

Similar to `add`, but uses clauses instead of formulas.

Syntax

```
add_clause CLAUSE WT ;
```

```
add ( VARIABLES ) CLAUSE WT ;
```

where `WT` is an optional floating point weight. If weight is missing, the clause or rule is considered as a "hard" clause or rule. (This is the same as having infinite weight).

Example

```
add_clause fileHasTask(fi22, ta1) 1.286;
```

adds a weighted ground clause.

```
add (File, Email, Task)
        ~fileHasTask(File, Task) | ~attached(File, Email)
      | emailHasTask(Email, Task);
```

adds a rule with infinite weight. This is essentially asserting the axiom

4.4.8. ask

Creates instances of the `FORMULA`, runs MCSAT sampling to get marginal probabilities, and prints the results, sorted according to probability (highest to lowest). Only returns results whose marginal probabilities are greater than or equal to the `THRESHOLD`, and at most `MAXRESULTS` are returned - unless it is 0, in which case all instances above the `THRESHOLD` are returned. The results are of the form

```
n results:
[x <- c, ...] prob: clause_instance
...
```

```
Note that the instances of the `FORMULA` are in clausal form, and in
general will not be a syntactic match.
```

Syntax:

```
ask FORMULA THRESHOLD MAXRESULTS ;


ask [ VARIABLES ] FORMULA THRESHOLD MAXRESULTS ;
```

where - `THRESHOLD` is a number between 0.0 and 1.0 inclusive; default 0.0 - `MAXRESULTS` is a nonnegative integer, default 0

Example:

```
ask [e, p] emailfrom(e, p) and hastask(e, ta1) 0.5 2;
```

returns the (at most) 2 instances with probability at least .5, for example

```
2 results:
[e <- em1, p <- pe1] 1.000: (emailfrom(em1, pe1) | hastask(em1, ta1))
[e <- em1, p <- pe2] 0.871: (emailfrom(em1, pe2) | hastask(em1, ta1))
```

4.4.9. mcsat

Runs the MCSAT process, running samples as described in ???.

Syntax

```
mcsat
```

`mcsat_params` is used to set MCSAT parameters.

4.4.10. mcsat_params

Displays or sets the parameters controlling the MCSAT algorithm. Parameters that are set keep their value until set to another value.

Syntax

```
mcsat_params NUMS ;
```

where NUMS is a comma-separated list of numbers, some of which may be omitted. The numbers represent, in order:

Parameter	Type	Default	Description
`max_samples`	nat	100	number of samples generated
`sa_probability`	0.0 .. 1.0	0.5	probability of a simulated annealing step
`samp_temperature`	> 0.0	0.91	temperature for simulated annealing
`rvar_probability`	0.0 .. 1.0	0.2	probability used by a walksat step (see Note)
`max_flips`	nat	1000	bound on the number of flipped variables
`max_extra_flips`	nat	10	number of extra flips performed after a model is found

Note

A walksat step selects an unsat clause and flips one of its variables with probability `rvar_probability`, that variable is chosen randomly with probability (1-rvar_probability), that variable is the one that results in minimal increase of the number of unsat clauses.

Example:

```
mcsat_params , , , .3;
```

sets the `rvar_probability` to `0.3`.

4.4.11. reset

Resets the probabilities and number of samples.

Syntax

```
reset ;
```

4.4.12. dumptables

Displays the current state of the system. This includes the sorts with their constants, the predicates, all known atoms with their current probabilities, clauses, and rules.

Syntax

```
dumptables ;
```

4.4.13. verbosity

Sets the verbosity level, used to control how much is printed. Defaults to 1.

Syntax

```
verbosity NUM ;
```

4.4.14. help

Provides a brief summary of the commands.

Syntax

```
help ;
```

4.4.15. quit

Exits mcsat.

Syntax

```
quit ;
```

5. XPCE

XPCE uses XML-RPC to provide PCE services. The server may be local or accessible over the web, and it supports multiple clients. The data is passed in JSON format, making it easy to read and check for errors.

5.1. Running XPCE

```
xpce PORT
```

This sets up an xpce server on the local host that listens to the given PORT (normally an unused port > 1024).

5.2. XPCE Clients

XPCE clients will connect to the `PORT` using the URL of the server, with `/RPC2` appended. For example, `http://localhost:8080/RPC2` would be used by a client running on the same host, where the server was started with `PORT 8080`. The client may then invoke any of the methods listed below.

5.3. XPCE Methods

XPCE responds to a number of XML-RPC methods. In general, the methods expect JSON input strings and returns JSON strings. For the most part, the methods correspond to the commands of MCSAT. The return is generally a JSON object (i.e., enclosed in *{}*), and if there is a warning or error, it is included in the object as `"warning":` or `"error":` followed by a descriptive string. In the method descriptions below, if **Returns** is missing it defaults to `{}`, and if **Errors** or **Warnings** is missing then the command generates no messages directly, though indirectly a message may still be returned, e.g., for a malformed formula.

5.3.1. `xpce.sort` - add a sort/supersort

Description:

If `"super"` is not provided, introduces new sort with the name `NAME`. Otherwise introduces new sorts as needed, and creates the subsort relation.

Argument
`{"name": NAME, "super": NAME}`

- `"super"` is optional

Errors

- `NAME is invalid`
- `NAME is in use as a sort` - if `"super"` is not provided
- `NAME is already a subsort of NAME`

5.3.2. `xpce.predicate` - add a predicate

Description:
adds the NAME as a predicate, with signature given by the list of sorts in NAMES.

Argument
`{"predicate": NAME, "arguments": NAMES, "observable": BOOL}`

Errors

- `NAME is invalid`
- `NAME is in use as a predicate`

5.3.3. `xpce.const` - add constants of a given sort

Description:
adds the NAMES as new constants of the given sort.

Argument
`{"names": NAMES, "sort": NAME}`

Errors

- `NAME is invalid`
- `NAME is in use as a constant`

5.3.4. `xpce.assert` - assert a fact

Description:
asserts the `FACT` to the internal database. Note that facts are of the form `p(c1,...,cn)`, where `p` is an observable predicate, and the `ci` are all constants. Use `xpce.add` with a high weight for any other formulas.

Argument
`{"fact": FACT}`

5.3.5. `xpce.add` - add a weighted assertion or rule

Description:

asserts the `FORMULA` to the internal database with the given weight. The FORMULA may contain variables, which are instantiated with constants of the corresponding sort.

Argument

`{"formula": FORMULA, "weight": NUM, "source": NAME}`

- `"weight"` is optional, defaults to `DBL_MAX`.
- `"source"` is optional, no default

5.3.6. `xpce.ask` - query for instances

Description:

Creates instances of the given formula, runs MCSAT sampling, and collects the results, sorted according to probability (highest to lowest). Only returns results greater than or equal to the threshold, and at most maxresults are returned - unless it is zero, in which case all instances above the threshold are returned.

Argument

`{"formula": FORMULA, "threshold": NUM, "maxresults": NUM}`

- `"threshold"` is optional between 0.0 and 1.0 - default 0.0
- `"maxresults"` is optional a nonnegative integer, default 0

Returns

a JSON array of the form

```
 [{"subst": SUBST, "formula_instance": FORMULA, "probability": NUM}
  ...
 ]
```

5.3.7. `xpce.command` - run an MCSAT command

Description:

This is the simplest, and the only one that does not expect a JSON string. It simply takes any of the commands as described in [MCSAT] in the form of a string (including the terminating *;*), and returns a string.

5.3.8. XPCE JSON Formula Syntax

```
FORMULA := ATOM
         | {"not": FORMULA}
         | {"and": [FORMULA, FORMULA]}
         | {"or": [FORMULA, FORMULA]}
         | {"implies": [FORMULA, FORMULA]}
         | {"iff": [FORMULA, FORMULA]}

ATOM := {"atom": {"predicate": NAME, "arguments": ARGUMENTS}}
```

```
NAMES := [NAME++',']
ARGUMENTS := [ARGUMENT++',']
CONSTANTS := [CONSTANT++',']
NUM := ['+'|'-'] simple floating point number
NAME := chars except whitespace parens ':' ',' ';'
ARGUMENT := CONSTANT | {"var":  NAME}
CONSTANT := NAME
```

A. OAAPCE

OAAPCE is specific to the CALO system, and is unlikely to be useful independently. It requires the Query Manager in order to get at the hard facts and type information, and waits for requests from the PCE harness and various learners. Historically, this was the only version of PCE, and the executable is still called `pce`.

The CALO system was built with several independent agents, all managed through OAA. OAAPCE is an agent that provides a set of OAA solvables as described below. The output from OAAPCE is a table listing the marginal probabilities of the indirect atoms. This table can be queried for specific patterns of atoms to obtain the marginal probabilities of instance atoms matching the pattern.

At startup, OAAPCE generates a log file in the same directory it was started from and includes a timestamp, e.g., `pce_2008-11-05T20-34-56.log`. It then initializes OAA connections, setting up the solvables. After that the internal tables are initialized, and the `pce.init` file is loaded, if it exists. Then the `pce.persist` file is loaded if it exists. Finally, the OAA main loop is invoked, at which point OAAPCE waits for OAA events.

The `pce.init` and `pce.persist` files are both text files, in the format expected by MCSAT (see the MCSAT user guide). The `pce.init` file is intended for relatively fixed information, e.g., the sort, subsort relation, and predicate declarations. Other MCSAT commands may be included, but these are the most important. Note that without predicate declarations OAAPCE will not generate any atoms, and nothing will be inferred. The `pce.persist` file is usually not generated by hand, but reflects the processing that occurred during an earlier session. Of course, it is just a text file and may be edited if desired. It will generally simply grow monotonically with each new session.

OAAPCE keeps track of all ground atoms, collecting the number of samples for which they are true; dividing this by the total number of samples taken gives the probability for the atom. To provide probabilities for more complex formulas, OAAPCE must be told the formula of interest, which then will be sampled along with the atoms.

The following sections describe the installation of OAAPCE, invoking OAAPCE, the OAAPCE solvables, the other OAA interactions, and an Appendix that gives examples of the various files and output generated by OAAPCE.

A.1. Running OAAPCE

Usage:

```
pce [OPTION]...
```

Option	Effect
`-h -? --help`	prints a help summary

`--lazy=BOOL`	whether to use lazy version (default `true`)
`-p --persistmode`	mode for the pce.persist file: `rw` - reads at startup, appends new events `ro` - reads only - no writing `wo` - does not read - but appends new events `none` - no reading or writing
`-v NUM --verbosity=NUM`	sets the verbosity level
`-V --version`	prints the version number and exits

A.2. OAAPCE Solvables

`pce_fact(Source,Atom)`
: States that `Atom` is a fact. Note that the `Atom` is simply a predicate applied to constants - negations are not allowed. The predicate must be **direct**. The `Source` is just a symbol indicating the source of the fact; it is not currently used by OAAPCE.

`pce_learner_assert(Lid,Formula,Weight)`
: This is used to make weighted assertions. `Lid` is a symbol indicating the learner generating the assertion - not currently used by OAAPCE. The `Formula` is an ICL formula. The `Weight` is a positive or negative double.

`pce_learner_assert_list(Lid,List)`
: This is simply a convenience, so that a learner may pass in a list of assertions in one shot. Each list element is a structure with two arguments: a `Formula` and a `Weight` (as in `pce_learner_assert`). The ICL structure can be a list or a function with two arguments - the functor name is ignored.

`pce_queryp(Q,P)`
: For a given query `Q`, OAAPCE will run a series of samples and return the probability `P`.

`pce_subscribe(Lid,Formula,Id)`
: This is a means for getting OAAPCE to "push" the information to an agent, instead of it running `pce_queryp`. The Formula is stored in internal tables, and a subscriber `Id` is returned. When `pce_process_subscriptions` is invoked, all instances of the `Formula` and associated probabilities are sent via `oaa_Solve`.

`pce_subscribe(Lid,Formula,Who,Condition,Id)`
: Currently the same as above, `Who` and `Condition` are ignored.

`pce_process_subscriptions(X)`
: For each subscription, this sends a list of formula instances and their associated probabilities to the subscriber.

`pce_unsubscribe(Lid,F)`
: Removes a given formula `F` from the subscriptions of the given `Lid`.

`pce_unsubscribe(Lid,F,Who,Condition)`
: Similar to the above, `Who` and `Condition` are ignored.

`pce_unsubscribe(SubscriptionId)`
: Unsubscribes based on subscription `Id`, rather than learner id and formula.

`pce_unsubscribe_learner(Lid)`
Unsubscribes all formulas associated with the given learner id `Lid`.

`pce_full_model(M)`
This returns a list of all ground atoms whose probability is greater than .51. This is not actually a model, but is called that for historical reasons.

`pce_add_user_defined_rule(Username,Text,Rule)`
This is used to provide support for the natural language interface of the Calo system. Users may provide their own rules, using natural language syntax. The rule the user `Username` typed in is given in `Text`, which is not currently used by OAAPCE. It is translated by the natural language interface into the form `implies(Strength,Antecedent,Consequent)`, where `Strength` is a weight, `Antecedent` and `Consequent` are the hypothesis and conclusion of the rule, respectively. This is similar to the learner assertions, but in addition allows abbreviations, e.g., `clib:` abbreviates `http://calo.sri.com/core-plus-office#`.

A.3. Other OAA interactions

These are interactions involving OAA and external agents that are not OAAPCE solvables. `app_idle` is generated by the OAA facilitator, while the others are solvables generated by external agents (e.g., learners, the Query Manager, etc.)

`app_idle`
This is the idle loop. It is called by the OAA facilitator often, and every minute it checks for new facts and constants from the query manager, runs a series of samples, and calls the `pce_update_model_callback` (see below).

`oaa_Solve(query(query_pattern('(rdf:type \"Constant\"" ?x))'),[],Result)`
When a new constant is provided to OAAPCE, its sort must be determined by invoking the Query Manager with this solvable.

`oaa_Solve(pce_subscription_callback(pce_query_p(Inst, Prob), ...)`
This is the solvable generated by `pce_process_subscriptions` (see above).

`oaa_Solve(pce_update_model_callback(Retract, BecameTrueWeights, Flag))`
This is invoked by the idle callback. The first time, `Retract` is the empty list, and `BecameTrueWeights` is a list of pairs of ground atoms and their probabilities, for those with probability greater than .51, and the `Flag` is "`full`". After that, only ground atoms whose value has "flipped" are given. Those that go from true (prob > .51) to false are added to the `Retract` list, while those that go from false (prob <` .51) to true are added to the `BecameTrueWeights` list, along with their associated probabilities. In this case the `Flag` is "`incremental`".

`oaa_Solve(agent_data(..., 'QueryManager', Info)`
This is used to determine if the query manager is available. Note that without the query manager, OAAPCE will not be able to determine the sort of new constants, and will ignore them.

`oaa_Solve(query(query_pattern(PredPat),AnswerPat,Results)`
This is used to determine if any new facts are available. `PredPat` is of the form "`'(Pred ?x01,?x02)'`", and `AnswerPat` of the form "`[answer_pattern('[{?x01},{?x02}]')]`". Again, if the query manager is not available, no new facts can be determined.

Bibliography

[MarkovLogic] Matthew Richardson and Pedro Domingos. *Markov Logic Networks*. *Machine Learning*. 2006.

[MCSAT] Hoifung Poon and Pedro Domingos. *Sound and Efficient Inference with Probabilistic and Deterministic Dependencies*. Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-06). 458—463. 2006. Boston, MA. AAAI Press.

[LazyMCSAT] Parag Singla and Pedro Domingos. *Memory-Efficient Inference in Relational Domains*. 'Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-06). 488—493. 2006. Boston, MA. AAAI Press.

[CALO] *The CALO Project*. SRI International. http://caloproject.sri.com/

[OAA] *Open Agent Architecture*. SRI International. http://www.ai.sri.com/~oaa/

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