libPRADA – Version 1.1
Learning and Planning with Probabilistic Relational Rules

Tobias Lang
http://userpage.fu-berlin.de/tlang/prada/

June 6, 2012

When using this library, please cite Lang and Toussaint [2010].

Contents

1 Disclaimer

libPRADA is work in progress! I did my best to avoid mistakes and to increase comprehensibility (at least at high-level interfaces). Nonetheless, you will probably encounter bugs and lacks in functionality. **Please contact me in this case!** I'll be happy to fix bugs, hear your complaints, add desired functionality and implement proposals and wishes for the library.

2 Introduction

libPRADA is a C++-library for model-based relational reinforcement learning in stochastic domains.

- libPRADA provides basic **data-structures and methods for symbolic reasoning with relational representations**. In particular, it implements probabilistic relational rules which can be used as a transition model $P(s' \mid s, a)$ in a Markov decision process.
- libPRADA provides algorithms for **planning** with ground relational rules: PRADA [Lang and Toussaint, 2010], sparse sampling trees (SST) [Kearns et al., 2002] and UCT [Kocsis and Szepesvari, 2006].
- libPRADA implements the algorithm for **learning** probabilistic relational rules by Pasula et al. [2007].

3 Installation

In the root-directory `libPRADA/`, please type `make`. This should compile the library `lib/libPRADA.so` and the demos in `test/`. If you have problems getting this working, please contact me.

If compilation succeeded, you can check out the three demos in `test/`.

In `make-config`, you can set the flag `OPTIM` to control the compiler options: `OPTIM = debug` sets debug information, `OPTIM = fast` optimizes the compiled code.

4 Programmer's guide

We will discuss the following headers in `libPRADA/src/`:

- `MT/array.h`: data-structure for lists
- `relational/symbols.h`: definition of the relational vocabulary; symbols are predicates and functions such as $on(\cdot,\cdot)$ and $size(\cdot)$
- `relational/literals.h`: definitions and basic methods for dealing with literals and sets of literals;

literals are instantiated symbosl such as $on(a, b)$ and $size(b) = 2$

- `relational/rules.h`: definitions and basic methods for dealing with rules and argument substitutions; probabilistic relational rules provide a transition model $P(s' \mid s, a)$
- `relational/reason.h`: reasoning with symbols, literals and rules such as calculating which rules cover a state and action
- `relational/plan.h`, `relational/prada.h`: planning with rules
- `relational/learn.h`: learning rules from data

4.1 Lists

File: `MT/array.h`

I use Marc Toussaint's `Array` class to store all kinds of data (primitive data-types, objects, pointers) in lists. `Array` is also used to define mathematical objects such as vectors and matrices. For instance, `Array< double >`, a list of `double`, represents a vector of reals. `Array` provides many basic self-explanatory methods to deal with lists, vectors and matrices.

4.2 Symbols

File: `relational/symbols.h`

Data structure A symbol consists of the following attributes:

```
MT::String name;
uint  arity ;    // e.g. 2 for on(.,.)
SymbolType symbol_type;
RangeType range_type;
uintA range;    // optional
ArgumentTypeL arg_types;  // optional
```

`SymbolType` is an enumeration in `Symbol`:

```
enum SymbolType {action, primitive, conjunction, transclosure,  ...};
```

The most important symbol types are `action` and `primitive`. They define the basic predicates and functions in a domain. Typically, the truth of primitive symbols needs to be provided from outside the logical machinery (e.g., by compiling observations into logical symbols, also known as the physical "grounding" problem). All other symbol types denote symbol which are defined in terms of other symbols. They include:

- `ConjunctionSymbol`: a conjunction of (possibly negated) symbols, allowing for either existentially or universally quantified variables; example: $clear(X) = \forall Y \neg on(Y, X)$
- `TransClosureSymbol`: transitive closure for a symbol; example $above(X, Y) = +on(X, Y)$
- `CountSymbol`: counts how often a symbol holds; example $countClear() = \sharp X : clear(X)$
- `AverageFunction`: computes the average over function values; example $averageHeight() = average_X[height(X)]$
- `SumFunction`: computes the sum over function values; example $sumHeight() = \sum_X height(X)$
- `MaxFunction`: computes the max over function values; example $maxHeight() = \max_X height(X)$

`Symbol.h` provides self-explanatory implementations for the derived symbol types. You can also add your own symbol types with appropriate definitions. Please note that the derived symbols can be used to specify *complex logical reward functions* (see also below and in `test/relational_plan/main.cpp`). For instance, it is possible to thereby define a reward for stacking large towers of blocks in a blocks-world.

`RangeType` is an enumeration in `Symbol`:

```
enum RangeType {binary, integer_set, integers, reals};
```

It defines the range of a symbol. Symbols can have a `binary` range and correspond to standard predicates then: they are either true (value 1) or false (value 0). Symbols can be functions which have the `integers` or the `reals` as range. It is also possible to define an `integer_set` for a specific set of range values: this set needs to be defined in `range`.

Finally, libPRADA will also allow for typed arguments by means of the data-structure `ArgumentType`. For instance, one may want to define a type `cube` for constants. However, this is still work in progress. (For now, you can achieve a similar effect in rules by using typing predicates like $cube(\cdot)$.)

File syntax The syntax for a symbols file is simply a list of symbols and (optionally) argument types as follows:

```
<Symbol>+
[                    // optional
<ArgumentType> +  // optional
]                    // optional
```

A single symbol is represented by its spelled out attributes `(name) (arity) (type) (range) [otherstuff]`. `[otherstuff]` is needed to define derived symbols. The exact syntax depends strongly on the type of the derived symbol. Please see the example files in the `test`-directories. Here is an example for a symbols file:

```
grab 1 action binary
on 2 primitive binary
size 1 primitive integers
clear 1 conjunction binary  <-- All Y -on(Y X)
inhandNil 0 conjunction binary  <-- All X -inhand(X)
above 2 transclosure binary  <-- + on
aboveNotable 2 conjunction binary <-- above(X Y) -table(Y)
height 1 count integers  <-- Num Y aboveNotable(X Y)
sum_height 0 sum integers <-- Sum height
```

Object Management To control the number of C++-objects and to exploit pointers in reasoning, C++-objects of symbols cannot be constructed from outside the class. Rather, the following method needs to be called to construct a symbol:

```
static Symbol* get(const MT::String& name, uint arity, SymbolType
symbol_type = primitive, RangeType range_type = binary);
```

Analogous `get`-methods exist for the derived symbols. After having called the previous `get`-method once, symbols can also be accessed by their names:

```
static Symbol* get(const MT::String& name);
```

4.3 Literals

File: `relational/literals.h`

Literals are instantiated symbols (symbols with a list of arguments) together with a value. Examples: $on(a, b) = 1$ and $size(a) = 4$. Actions are literals whose value is ignored, for example $grab(a)$.

4.3.1 Arguments: variables and constants

Arguments of literals are represented by non-negative one-digit and two-digit integers (`uint`) $0, 1, 2 \ldots 99$. For reasoning, we need to distinguish between logic variables and constants. The default is that all `uints` < 10 are variables, all other constants. You can change the default by defining the set of constants with the following methods in `relational/reason.h`:

```
void setConstants(uintA& constants);
bool isConstant(uint id);
```

There are also appropriate methods to fix the `ArgumentType` of constants.

4.3.2 Literals data-structure

A literal consists of the following attributes:

```
Symbol* s;
uintA args;    // uintA = MT::Array < uint >
double value;
ComparisonType comparison_type; // default: comparison_equal
```

`uintA` is short for `MT::Array<uint>` (a list of unsigned integers). For binary symbols (aka predicates), the value 0 represents a negated (/false) symbol and the value 1 a positive (/true) symbol. For instance, for $on(61, 63) = 1$ we have `s = Symbol::get(''on''), args(0)=61, args(1)=62, value=1` and `comparison_type = Literal::comparison_equal`.

`ComparisonType` is an enumeration:

```
enum ComparisonType { comparison_equal, comparison_less,
comparison_lessEqual, comparison_greater, comparison_greaterEqual };
```

The default of `comparison_type` is `comparison_equal`. For example, `comparison_less` is used in $size(a) < 4$.

File syntax Simply use **plain text** descriptions:

```
on(65, 60)   // for predicates: corresponds to value=1
-inhand(71) // for predicates: corresponds to value=0
size(1)<=2
size(66)=2
```

Please note the special notation for the value of binary symbols (= predicates) which is close to standard logic notation: if the value is omitted, it denotes value 1 (= true); with a leading – (for ¬), it denotes value 0 (=false).

Lists of literals are usually written in one line, separated by a space or by a comma.

Object Management To control the number of C++-objects and to exploit pointers in reasoning, C++-objects of literals cannot be constructed from outside the class – this is the same as with symbols. Rather, the following methods need to be called to construct literals:

```
static Literal * get(Symbol* s, const uintA& args, double value,
                     ComparisonType comparison_type = comparison_equal);
static Literal * get(const char* text);
```

The second method is a quick way to read literals from text. Please note that the corresponding symbols need to have been created beforehand (by calling `Symbol::get(...)`).

4.3.3 SymbolicState

A `SymbolicState` consists of:

```
MT::Array<Literal*> lits;
uintA state_constants;
bool derived_lits_are_calculated;
```

The important attribute is the list of literals `lits`. We make the closed world assumption: `lits` contains binary literals only if they are positive (with value 1). Hence, all binary literals which are not explicitly stated are assumed to be false. `state_constants` can be set optionally and contains all constants in the state (note that not necessarily all state constants need to appear as an argument in `lits`). `derived_lits_are_calculated` is a flag which stores whether the derived symbols have already been calculated.

File syntax A state is represented by its list of literals. A trailing list of `state_constants` is optional:

```
[61 62 63 64] // optional
on(61,62) on(65,64) inhand(66) size(64)=3
```

Please note the special syntax for literals of binary symbols (predicates) as described above: no negative binary symbols are allowed and positive binary symbols don't have a value specified (no `=1`).

4.3.4 StateTransition

`StateTransition` is a convenience wrapper for the reinforcement learner's favorite triplet (s, a, s'):

```
SymbolicState pre, post;
Literal * action;
MT::Array< Literal* > changed;
uintA changedConstants;
```

`changed` and `changedConstants` are calculated automatically by the constructor. A list of state transitions can be read using the following method:

```
static StateTransitionL& read(const char* filename);
```

Lists of state transitions provide the data for rule-learning.

4.4 Rules

File: `relational/rules.h`

Probabilistic relational rules are at the heart of libPRADA. They provide a transition model $P(s' \mid s, a)$ for model-based relational reinforcement learning.

4.4.1 Rules

The data-structure `Rule` implements the noisy indeterministic deictic (NID) rules of Pasula et al. [2007]. I follow exactly the semantics described in their paper. Please confer their paper or my Ph.D. thesis [Lang, 2011] for further details, including state-action coverage and the noise outcome. In particular, it is important for you to understand when a rule covers a state-action pair and when not (I'll describe this superficially also below): rule uniqueness and the noisy default rule are important concepts to understand.

`Rule` has the following attributes:

```
Literal * action;
LitL context;
MT::Array< LitL > outcomes;
doubleA probs;
double noise_changes;
arr outcome_rewards; // optional
```

Please note that `LitL` is short for `MT::Array< Literal* >`, a list of `Literal` pointers.

`action` is the rule's action (surprise, surprise). `context` is the list of (abstract) literals which need to be covered by a state so that the rule can apply. `outcomes` contains the different outcomes, `probs` the corresponding probabilities. Please note an important convenience for `outcomes` and `probs`: *the last outcome is the noise outcome*. `noise_changes` is PRADA's noise outcome heuristic: it defines the average number of state properties that change in a noise outcome – however, in practice, this is a negligible parameter. `outcome_rewards` is an optional parameter which specifies a reward for each outcome: planners like PRADA may take them into account in addition to global rewards on states (see the IPPC domains for example domains).

`Rule` provides many convenience methods. It also provides a method to construct the **noisy default rule** which is important when learning and reasoning with NID rules (see Pasula et al. [2007]):

```
static Rule* generateDefaultRule(double noiseProb,
                double minProb, double change);
```

The default rule uses the special action $default()$ and is applied when there is no unique non-default covering rule for a state-action pair.

Sets of rules should be managed by means of the class `RuleSet`. `RuleSet` provides the methods for grounding abstract rule-sets. Furthermore, `RuleSet` controls the deletion of C++-objects of `Rule` so that your working memory does not drown in `Rule` pointers (when grounding abstract rules, many, many ground rules are created).

There is also an additional container structure `RuleSetContainer` for `RuleSet` in `learn.h`: it is used for efficiency reasons in rule-learning and provides auxiliary methods and statistics required for learning and evaluating rules on a given data-set.

File Syntax: rules The general syntax is straightforward:

```
ACTION:
<Literal>
CONTEXT:
<Literal>+
OUTCOMES:              // List of outcomes
<double> <Literal>+    // Outcome 1 incl. probability
<double> <Literal>+    // Outcome 2 incl. probability
...
```

Example:

```
ACTION:
  puton(X)
CONTEXT:
  block(X), inhand(Y), size(X)=2
OUTCOMES:
  0.7 on(Y X), upright(Y), -inhand(Y)
  0.2 -inhand(Y)
  0.1 <noise>
```

Some guidelines for rules:

- Outcomes are lists of *primitive* literals with a leading probability. No literals for derived symbols here! Derived literals, however, may appear in the context.
- The last outcome of a rule *must* be the noise outcome.
- The noise outcome can specify how many (random) state properties are expected to change (required by PRADA's heuristic to deal with the noise outcome).

4.4.2 Substitution

A `Substitution` maps integers to integers. The typical use is to ground abstract literals and rules: variables get mapped to constants (please recall that both variables and constants are represented by `uint`). The essential two attributes are:

```
uintA ins;
uintA outs;
```

`ins` and `outs` are lists of `uints`. `ins(i)` maps to `outs(i)`. `Substitution` provides many (hopefully) self-explanatory methods. The most important ones are (i) the various `apply` methods such as applying substitutions to rules and (ii) `void addSubs(uint in, uint out)` for adding a substitution.

Sets of substitutions can be managed by means of the container class `SubstitutionSet`. Similarly as for `RuleSet` the major purpose is to control the deletion of C++-objects of `Substitution`.

4.5 Reasoning

File: `relational/reason.h`

The namespace `reason` provides methods for logical reasoning. These methods are broadly distinguished into **basic** reasoning and **rule reasoning** methods.

4.5.1 Basic reasoning

The basic reasoning methods realize three major functionalities: (i) distinguishing between constants and variables; (ii) deriving literals for derived symbols (symbols

which are defined in terms of other symbols); (iii) basic coverage methods (for example, whether a state covers an abstract literal).

Distinguishing between constants and variables As described in Sec. 4.3.1, arguments of literals (variables and constants) are represented by `uints`. `reason` maintains the information which `uint` refers to a variable and which to a constant. By default, all `uints` $<$ 10 refer to variables, all other to constants. Alternatively, you may provide a set of constants by `void setConstants(uintA& constants)`. The fundamental methods of libPRADA for distinguishing ground and abstract literals are:

```
bool isGround(const Literal* lit );
bool isPurelyAbstract(const Literal* lit );
```

Please note that for a literal to be purely abstract, all arguments need to be variables. For example, $on(a, X)$ with a constant a and a variable X is abstract, but not purely abstract.

Deriving literals There are primitive and derived (/non-primitive) symbols. Derived symbols are defined in terms of other symbols. To describe the world symbolically, the truth of literals for primitive symbols needs to be specified from outside the logical machinery. This is the physical grounding problem: how do symbol relate to the true world? Please note that this is different from logically *grounding* an abstract literal to a ground literal. In contrast to primitive symbols, the literals for derived symbols need to be calculated from other literals using logical reasoning. `reason` provides the required methods. The central method for calculating the derived literals for a state is:

```
void derive(SymbolicState* s);
```

Basic coverage methods There are three types of basic coverage methods which depend on each other. The `holds` methods check for test literals whether they hold in a given list of literals (simple contains check): for example whether literal $on(a, d)$ holds in $\{on(b, a), on(a, d), inhand(e)\}$. The `calcSubstitution` methods try to unify different literals (typically, abstract and ground literals), for example $on(a, X)$ and $on(a, d)$. If successful, the resulting `Substitution` provides the mapping of variables to variables/constants; $X \to d$. The `calcSubstitutions` and `cover` methods try to unify *lists* of literals and return lists of appropriate substitutions.

4.5.2 Rule reasoning

The methods for rule reasoning provide the following functionalities: (i) distinguishing between ground and abstract rules; (ii) calculating successor states and their probabilities for a rule and a given state-action pair; (iii) calculating the coverage of rules; (iv) calculating likelihoods of experiences (s, a, s') for rule-sets.

Coverage of rules The implementation of state-action coverage for NID rules in libPRADA follows directly the specification in Pasula et al. [2007]. This coverage has a specific semantics which might be unexpected. Understanding the semantics is crucial for learning and planning with NID rules. Please see Pasula et al. [2007], Lang and Toussaint [2010] and Lang [2011] for details.

I highlight the most important feature here: For a given state-action pair (s, a) and a set Γ of abstract rules, we check which rules in Γ cover (s, a). That is, we substitute the arguments of the action in an abstract rule with the constants in a. Then, we try to find a unique substitution for the remaining variables in the rule. These remaining variables which do not appear in the action's arguments are called deictic references. If there is exactly one non-default rule in Γ covering (s, a) we call it the unique covering rule and use it to model $P(s' \mid s, a)$. If there is no such unique covering rule (there is no covering rule or there are at least two covering rules), we use the default rule, basically saying that we do not know what will happen.

There is an important subtlety concerning the deictic references: for a rule to cover (s, a) there needs to be a unique substitution for these references. If a deictic reference has several groundings such that the rule's context holds then the rule does *not* cover (s, a) (since such a variable is not a well-defined deictic reference in s).

Since this is so important, I give a small example here. Please consider the rule:

```
ACTION:
  puton(X)
CONTEXT:
  block(X), inhand(Y)
OUTCOMES:
  0.9 on(Y X), −inhand(Y)
  0.1  <noise>
```

Y is a deictic reference here. Consider the symbolic states $s_1 = \{block(a), inhand(b)\}$, $s_2 = \{block(a)\}$, $s_3 = \{block(a), inhand(b), inhand(c)\}$ and the action $a = puton(a)$. The rule covers only s_1: there is the unique grounding $\{X \to a, Y \to b\}$. However, it does not cover s_2: Y cannot be resolved. Most importantly, the rule does not cover s_3, either: there is no unique grounding. The reason is that the deictic reference Y cannot be resolved uniquely: there are two possible groundings $\{X \to a, Y \to b\}$ and $\{X \to a, Y \to c\}$.

4.6 Planning

Files: `relational/plan.h`, `relational/prada.h`

This is a core part of libPRADA. It provides four algorithms (PRADA, A-PRADA, SST, UCT) for planning with *ground* NID rules. The important methods for any planner are:

```
Literal * plan_action(const SymbolicState& current_state);
void setReward(Reward*);
void setGroundRules(RuleSet& ground_rules);
```

The first method prompts the planner to plan an action for a given state. Thus, this method defines a policy $\pi : s \to a$. The planner tries to find the approximately

best action leading to high rewards. PRADA and A-PRADA also provide a method to generate a complete plan. In contrast, SST and UCT cannot provide a complete plan due to their outcome sampling. The second method sets the reward function of the planner (see below). The third method sets the *ground* rules which provide the transition model $P(s' \mid s, a)$ used by the planner. Hence, a set of abstract NID rules needs to be grounded first with respect to the domain constants; the ground rules are then provided to the planner. The methods for grounding abstract rule-sets are provided by the class `RuleSet` in `relational/rules.h`.

All planning algorithms share the following parameters:

```
double discount;
uint  horizon;
double noise_scaling_factor;
```

`discount` is a discount factor $\gamma \in (0, 1]$ (a future reward at time t is discounted by γ^t). `horizon` is the planning horizon $h > 0$. Specific parameters for the individual planners are discussed below. `noise_scaling_factor` is a noise scaling factor η used by SST and UCT: it scales down the future values when sampling the noise outcome.

4.6.1 Individual planners

PRADA PRADA stands for "probabilistic relational action-sampling in dynamic Bayesian networks planning algorithm" [Lang and Toussaint, 2010]. PRADA samples action-sequences and evaluates their rewards using a dynamic Bayesian network (DBN) `PRADA_DBN*` `dbn` for all *ground* symbols. Its most important parameter is `num_samples` which controls the number of samples: the more samples, the higher the probability to find good plans, but also the higher the computational demand.

Furthermore, you may specify `threshold_reward` $\in (0, 1]$ to define what a "good" plan is: it sets the threshold on the probability to achieve the reward. It should not be set too high since PRADA's approximate inference may underestimate the true probability. Another parameter is `noise_softener` $\in (0, 1]$: there is no clear way how to deal with the noise outcome of rules; PRADA's heuristic to do so can be harmful if the noise outcome has too high probability; this parameter reduces the effect of the noise outcome in planning.

A-PRADA Adaptive-PRADA extends PRADA as described in Lang and Toussaint [2010]. The important method called during planning is

```
double shorten_plan(LitL& seq_best, const LitL& seq, double value_old)
```

It takes a plan and examines whether this plan can be improved by deleting some actions.

SST SST stands for "sparse sampling tree" (SST) algorithm [Kearns et al., 2002]. SST is used for planning with NID rules in the work by Pasula et al. [2007]. It has the following additional parameter:

- `branch` (b) determines the number of samples from the successor state distribution for a given action (= branching factor of the sampling tree).

UCT UCT stands for "upper confidence bounds applied to trees" [Kocsis and Szepesvari, 2006]. It has the following two additional parameters:

- A bias `c` for less often explored actions.
- `numEpisodes` (e) determines the number of episodes (or rollouts).

4.6.2 Reward functions

Reward functions $R : S \to A$ are modelled by the class `Reward`. Please note that R only depends on the state. There is also the possibility to associate reward with actions by setting rewards to individual rule outcomes (see Sec. 4.4).

`Reward` provides the following methods to evaluate states:

```
double evaluate(State& s);
bool satisfied(State& s);
bool possible(State& s);
```

Methods two and three may have trivial implementations by always returning true.

The following pre-defined reward types are provided by libPRADA:

- LiteralReward: the reward is given for achieving a single literal, e.g. $inhand(a)$ or $inhand(X)$
- LiteralListReward: the reward is given for achieving a conjunction of literals, e.g. $on(a, b), on(b, c)$
- MaximizeReward: the value of an atom shall be maximized, e.g. $sumHeight()$ (this defines the stacking task in good, old blocksworld)

You can define arbitrarily **complex logical reward functions** by means of derived symbols: derived symbol (like $sumHeight()$) capture the complex logical structure (like "stacking"); then, these derived symbols simply need to be achieved or maximized.

PRADA reasons on beliefs over states (rather than on states directly). For this purpose, PRADA maintains its own class `PRADA_Reward`. For the three reward functions discussed above, automated conversion routines are used when setting the standard `Reward` for PRADA. If you come up with your own reward function type, however, you must also define a `PRADA_Reward` type that implements evaluating this reward function over beliefs.

File syntax: rewards A flag of the reward type plus the literal(s) as described above:

LiteralReward:

```
1
<Literal>
```

LiteralListReward:

```
2
<Literal>+
```

MaximizeReward:

```
3
<Literal>
```

4.7 Rule learning

File: `relational/learn.h`

libPRADA provides a direct implementation of the learning algorithm for probabilistic relational rules by Pasula et al. [2007].

To learn rules, you need to come up with a set $\{(s, a, s')\}$ of state transitions (s, a, s') using the data-structure `StateTransition` (see 4.3.4). The central method is:

```
void learn_rules(RuleSetContainer& rulesC,
        StateTransitionL& experiences,
        const char* logfile);
```

For efficient rule-learning the data-structure `RuleSetContainer` is used: `RuleSetContainer` is a wrapper for `RuleSet` which stores information on covered state-transitions.

The learning algorithm is very sensitive to two parameters: the regularization penalty α and the lower bound p_{min} on the probability of states under the noise outcome (see Lang and Toussaint [2010] and Pasula et al. [2007] for details). You can set them with these methods:

```
void set_penalty(double alpha_PEN);
void set_p_min(double p_min);
```

You need to get a feeling for these parameters to be able to learn rules which you consider "good". The parameters cannot be set automatically: what "good" is depends on your prior knowledge (for instance, you must choose whether you want an almost perfect, complex model or a compact model only for typical state transitions). I discuss this briefly in my thesis [Lang, 2011].

The learning algorithm defines a heuristic search through the space of rule-sets based on search operators. You may adapt the algorithm to your needs by defining your own search operator, deriving from `SearchOperator`. Which operator is tried at each time-step is determined by the method

```
void set_ChoiceTypeSearchOperators(uint choice_type);
```

The random `choice_type` considers sampling weights for the invididual search operators in its random choice. Hence, you may change the algorithm by modifying the weights of the search operators.

5 User's guide

To use libPRADA successfully, you have to understand:

1. how libPRADA represents symbols, literals, states and rules
   Sources: `relational/symbols.h`, `relational/literals.h`, `relational/rules.h`
   Demo: `test/relational_basics`

2. how to set up a planning scenario
   Sources: `relational/plan.h`, `relational/prada.h`
   Demo: `test/relational_plan`

3. how to set up a rule learning run
   Sources: `relational/learn.h`
   Demo: `test/relational_learn`

Please take a look at the `main.cpp` of the corresponding test. The tests are carried out in the robot manipulation domain as first presented in Lang and Toussaint [2010].

Basic steps you always need to perform when using libPRADA:

1. Set up **symbols**
2. Optional: define **constants** (/objects); this tells libPRADA which `uint`s refer to constants, which to variables

Additional steps for **planning**:

1. Set up a **start state** for planning
2. Define a **reward function**
3. Set up (abstract) **rules**
4. **Ground** all rules
5. Set up the **planner**
6. Provide ground rules, reward and all necessary **parameters** to your planner
7. Finally, **plan** :-) !

Additional steps for **learning**:

1. Set up **data** in the form of state transitions (s, a, s') using `StateTransition`
2. Set **regularization** parameter α (`alpha_pen`)
3. Set **lower bound for noise outcome** p_{min} (`p_min`)
4. Finally, **learn** :-) !

6 Trouble shooting

6.1 Planning

- Double-check the abstract rules. Do they really allow to find a correct plan? Are there *unique* covering rules (with *unique* deictic references) for the required actions ?
  The human intuition that "these rules are sufficent for that reward" is often wrong: after detailed inspection of the planner, it often turns out that the rules are actually "wrong" and the planner is "right": the given rules do *not* enable a correct plan.

- Please be sure to understand the concept of *unique covering rule* as described in Pasula et al. [2007] and Lang and Toussaint [2010].

- Is the horizon h set correctly? Too short is obviously bad. However, also too long horizons can confuse PRADA: the approximation in form of the factored frontier gets more and more inexact over time.

- Use a sufficiently large number of samples `numSamples` for PRADA. Likewise, a large number of episodes for UCT and a high branching factor for SST.

- The probabilities of the required rule outcomes must not be too small. If they are too small, the sampling-based planners SST and UCT can fail to sample them. PRADA can drastically underestimate true probabilities: this is due to its factored frontier approximation which multiplies probabilites over time and also within a time-step (when calculating the probability of a rule context based on several individual variables) – in case of small outcome probabilities, this approximation leads quickly to values of almost zero.

- You may want to use the DEBUG information of the individual methods in `relational/prada.cpp` and `relational/plan.cpp`.

- Concerning the speed of planning: Using many complex derived symbols makes planning slow. For each sampled action/state the planners need to compute the derived symbols (or the beliefs over these).

6.2 Learning

- Play with different settings of α and p_{min}. The learning procedure is extremely sensitive to these parameters.

- Do you have sufficient data? Are the data sufficient evidence for the rules you have in mind?

- The learning algorithm is a heuristic search. It does not guarantee an optimal solution. Also, different rule-sets may achieve the same performance (in terms of the loss function): the found rule-set might explain the data as well as the rule-set you actually have in mind.

- You may want to use the DEBUG information of the individual methods in `relational/learn*.cpp`.

7 Future research

Model-based relational reinforcement learning is a promising framework to advance our understanding of intelligent agents acting in the real world. I hope libPRADA helps you to take part in this fascinating research area.

Clearly, NID rules and PRADA have strong limitations. There are plenty of relevant directions for future research. These include: online learning of rules; extending the rule framework to work with continuous symbolic functions; improving PRADA by using a better sampling function of actions; accounting for the relevance of objects; finding better planning strategies which exploit redundancies in the ground DBNs. And many, many more!

Good luck with your research!

8 Acknowledgements

Thanks to Andreas Henne who has provided code for learning and reasoning with non-binary symbols.

References

Michael J. Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine Learning*, 49(2-3):193–208, 2002.

Levente Kocsis and Csaba Szepesvari. Bandit based monte-carlo planning. In *Proc. of the European Conf. on Machine Learning (ECML)*, 2006.

Tobias Lang. *Planning and Exploration in Stochastic Relational Worlds*. PhD thesis, Fachbereich Mathematik und Informatik, Freie Universität Berlin, 2011.

Tobias Lang and Marc Toussaint. Planning with noisy probabilistic relational rules. *Journal of Artificial Intelligence Research*, 39:1–49, 2010.

Hanna M. Pasula, Luke S. Zettlemoyer, and Leslie Pack Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352, 2007.