FiMDP Documentation

Release 1.0

Frantisek Blahoudek and Pranay Thangeda

Oct 24, 2020

Contents

CHAPTER 1

Introduction

FiMDP is a Python package for analysis and controller synthesis of Markov decision processes with resource constraints, modeled as Consumption Markov Decision Processes (ConsMDPs). The model of ConsMDPs and associated algorithms were first introduced in our work titled *Qualitative Controller Synthesis for Consumption Markov Decision Processes* presented at CAV2020 *Citation Info*.

This package includes interactive Jupyter notebooks with examples from FiMDPEnv, our related project that provides realistic simulation environments that model real-world ConsMDPs. The package also includes tutorials designed to help you get started with our tool.

CHAPTER 2

Authors

This package is developed and maintained by František Blahoudek, and Pranay Thangeda. Contact information is provided in the section *Support*.

Note: This overview assumes no background in the topic and tries to explains the problem in layman terms. For a detailed explanation of our work, please go through our paper *Citation Info*.

Several real-world systems of interest are resource constrained, i.e., they utilize certain resource from a limited supply that must be replenished regularly. For example, autonomous electric systems such as driverless cars, autonomous drones, planetary rovers, etc, are constrained by design to operate on power drawn from a battery of limited capacity and resource exhaustion could potentially lead to several undesirable consequences including safety hazards. Further, such systems usually operate in uncertain environments with stochastic dynamics that can be effectively modeled as Markov decision process (MDPs).

Traditionally, resource-constrained systems were studied using so called *energy* based models that are known not to admit polynomial-time controller synthesis algorithms. We instead build up-on so called *consumption* models that typically admit more efficient analysis and extend it to probabilistic setting. A consumption MDP (ConsMDP) is then characterized by an MDP that models the stochastic environment, the capacity of the agent, and the initial resource level. Given a ConsMDP with no zero-consumption cycles, and a set of target states, we show that we can prove in polynomial time the existence of a strategy that prevents resource exhaustion and visits some target states infinitely often. If such a strategy exists, we provide its polynomial-size representation in polynomial-time.

The interactive map above visualizes an example ConsMDP and the strategy obtained from our algorithm. **Zoom in** to see the action choices at different states (intersections) for different energy levels. The green nodes indicate reload states, the blue nodes indicate target states, and the red nodes are states where no safe action is prescribed by the strategy at the current energy level.

CHAPTER 3

Installation

FiMDP is written in Python 3 and the examples are presented using interactive Jupyter notebooks. FiMDP can be installed using pip from PyPI:

```
pip install -U fimdp
```

While the baseline package has minimal dependencies, FiMDP depends on several other tools for extended functionality. Some of the recommended dependencies are:

- Graphviz: for visualizations in Jupyter notebooks,
- Storm and Stormpy: for reading PRSIM, JANI, and Storm models,
- Spot: for support of labeled ConsMDPs and specifications given as deterministic Büchi automata or the recurrence fragment of Linear-time Temporal Logic (LTL).

CHAPTER 4

Citation Info

This tool is based on original paper that introduces the notion of ConsMDPs and also presents associated algorithms and guarantees. To cite our work on ConsMDPs please use the following publication:

Blahoudek F., Brázdil T., Novotný P., Ornik M., Thangeda P., Topcu U. (2020) **Qualitative Controller Synthesis for Consumption Markov Decision Processes**, in proceedings of 32nd International Conference on Computer-Aided Verification (CAV 2020), Lecture Notes in Computer Science, vol 12225. Springer. https://doi.org/10.1007/978-3-030-53291-8_22

CHAPTER 5

Support

Detailed documentation of the modules has been provided in the *Documentation* section. If you have any trouble with the installation, or have any questions, raise an issue in GitHub or email František Blahoudek or Pranay Thangeda.

CHAPTER 6

License

This package is released under the highly permissive MIT license which also makes it clear that the authors and the organizations they are a part of cannot be held liable to any damage caused by usage of this package or any topic discussed in it. For a detailed statement, go through the license file - *License*.

CHAPTER 7

Detailed Contents

7.1 Documentation

This section provides the documentation of all the modules in the package.

7.1.1 fimdp.core module

Core module defining basic data structures and classes of the FiMDP package.

It defines the class for representation of Consumption Decision Processes, the interface for strategies (including exceptions), and Counter-strategies (and the needed building blocks). Further, it provides support for ConsMDPs that represent a product with some input ConsMDP and some other component (explicit energy, automaton, . . .).

## Consumption Markov Decision Processes (CMDPs)

See our [CAV paper] for the theoretical backgrounds.

The classes needed for representation of (CMDPs) are:

- *ConsMDP*: represent an CMDP object
- *ActionData*: represent actions of CMDPs

The *ProductConsMDP* are ConsMDPs with states fromed by 2 components.

## Interface for strategies

A *strategy* should offer *strategy.next_action()* that picks an action based on history, and the function *strategy.update_state(outcome)* which tells the strategy that the last action was resolved to the state outcome — which becomes the new current state. Each call to *next_action()* should be followed by a call to *update_state* and vice versa. Both functions raise *WrongCallOrderError* if the calls do not alternate.

To simplify code, *strategy.next_action(outcome)* is a shorthand for

```
>>> strategy.update_state(outcome)
>>> strategy.next_action()
```

The function *update_state(outcome)* raises a *ValueError* if *outcome* is not a valid successor for the last action returned by *next_action()*. Based on the *outcome*, the strategy can update its memory.

The strategy can be used in a new play using *strategy.reset()* which allows new initialization of initial state and memory.

The class *Strategy* implements the basic interface for strategies, but it neither updates any memory nor picks actions. Its subclasses should override the function *._next_action()* and (when using memory) also *._update(outcome)*.

## Counter-strategies

The main ingredient of a counter strategy is a *counter selector*. A counter selector is a mapping from states to *selection rules*. A selection rule selects actions based on given energy level. See, again, out [CAV paper] for details.

The classes *CounterStrategy*, *CounterSelector*, and *SelectionRule* implement the respective objects as described in the paper.

ProductSelector and *ProductSelectorWrapper* are two selectors that can be used to hide the product construction from the user and maps actions and states of a ProductConsMDP into states and actions of the original ConsMDP.

[CAV paper]: https://link.springer.com/chapter/10.1007/978-3-030-53291-8_22

class `fimdp.core.`**`ActionData`**(*src*, *cons*, *distr*, *label*, *next_succ*)

Bases: `object`

Holds data of an action in ConsMDP.

The defining attributes of an action are:

- source state *src*
- consumption *cons*,
- the successors probability distribution *distr*,
- the action *label*

The attribute *next_succ* is used to keep a nested linked-list of actions with the same *src*.

`get_succs`()

class `fimdp.core.`**`ConsMDP`**(*layout=None*)

Bases: `object`

Represent Markov Decision Process with consumption on actions.

States are represented by integers and actions are represented by *ActionData* objects. To add an action, use *add_action*. To iterate over actions of a state *s* use *actions_for_state(s)*. If you wish to remove actions during the iteration, use *out_iteraser(s)* instead. There is also *remove_action* which requires an action id. See implementation details for further info.

States can have names using the list *names*. Reload states are stored in the set *reload_states*.

Important: Functions that change the structure of the consMDP should always call *self.structure_change()*.

The action objects are stored in a sparse-matrix fashion using two vectors: *succ* and *actions*. The latter is just a list of *ActionData* objects stored in the order in which the actions were created. Using the *ActionData.next_succ* the actions form a linked-list of actions for each state (that is how *actions_for_state(s)* work internally). The vector *succ* serves to locate the first action in this linked-list for given state (*actions[succ[s]]* hold the first action of *s*).

Do not modify the two vectors directly. Always use *ConsMDP.add_action* to add and *ConsMDP.remove_action* or *ConsMDP.out_iteraser(s)* to remove actions.

Parameters **`layout`** (*`str or None (default)`*) – one of the Graphviz engines to compute graph layouts ("dot", "neato", "twopi", "circo"). The engine "dot" is used if *layout* is not specified. The layout can be later changed using the attribute *ConsMDP.dot_layout*.

`actions_for_state`(*s*)
Return iterator of actions available for state *s*.

`add_action`(*src*, *distribution*, *label*, *consumption=0*)
Add action to consMDP.

Returns: index of the new action in the *actions* list. :raises: * * *ValueError* if attempt to use non-existent state

- * *ValueError* if src-label->… exists already.

`get_dot`(*options="*)

`is_reload`(*sid*)
Return the reload status of state *sid*.

`new_state`(*reload=False*, *name=None*)
Add a new state into the CMDP.

Returns: the id of the created state Raise *ValueError* if a state with the same name already exists.

`new_states`(*count*, *names=None*)
Create multiple (*count*) states.

The list names must have length *count* if supplied. These will be the names for the states.

Return the list of states ids. Raise *ValueError* if a state with the same name already exists.

`out_iteraser`(*s*)
Return iterator of actions available for state *s* that allows action removal.

`remove_action`(*aid*)
Remove action based on its id.

`set_reload`(*s*, *reload=True*)
Set reload status of given state(s).

If *s* is a list, set all of them as reloading states. By setting *reload=False*, the states will be removed from reloading staes.

`show`(*options="*, *targets=None*, *max_states=None*)

`state_succs`(*s*)
Return successors of *s* over all actions

`state_with_name`(*name*)
Return id of state with name *name* or *None* if not exists.

`structure_change`()

`class` `fimdp.core.`**`CounterSelector`**(*mdp*, *values=None*)
Bases: `list`

CounterSelector selects actions based on given state and energy level.

Counter selector is a list of SelectionRules extended by:

- pointer to the corresponding mdp, and
- **2 functions for updating and accessing actions to be taken:**
  - update(state, energy_level, action)

  - select_action(state, energy)

copy_values_from(*other*, *state_subset=None*)
: Replace values for given *state_subset* by values from *other* counter selector.

  If *state_subset* is not given (or is None), replace values for all states.

select_action(*state*, *energy*)
: Return action selected for *state* and *energy*

update(*state*, *energy_level*, *action*)
: Update the action for given *state* and *energy_level* to *action*.

  energy_level is a lower bound of an interval for which *action* will be selected by *select_action*.

  Raises ValueError if *product_action* is not an action of *product_state*

class fimdp.core.**CounterStrategy**(*mdp*, *selector*, *capacity*, *init_energy*, *init_state=None*, **args*, ***kwargs*)
: Bases: *fimdp.core.Strategy*

  Counter strategy tracks energy consumption in memory and chooses next action based on the current state and the current energy level.

  This class implements the memory and its updates. The selection itself is delegated to *selector*. The attributes *capacity* and *init_energy* are needed to track the energy level correctly.

  The implementation is suited to use CounterSelector as *selector*, but can take anything that implements *select_action(state, energy)*.

exception fimdp.core.**NoFeasibleActionError**
: Bases: Exception

class fimdp.core.**PickFirstStrategy**(*mdp*, *init_state=None*, **args*, ***kwargs*)
: Bases: *fimdp.core.Strategy*

  Class for testing and presentation purposes.

  Always picks the first available action of the CMDP. Does not track energy and does not give any guarantees.

class fimdp.core.**ProductConsMDP**(*orig_mdp*, *other=None*)
: Bases: *fimdp.core.ConsMDP*

  CMDP with states that have two components.

  We call the two components *orig_mdp* and *other*, where *orig_mdp* is some ConsMDP object and other can be arbitrary domain, for example deterministic Büchi automaton, or upper bound of some integer interval. The *orig_mdp* and *other* store pointers to the objects of origin for

  the product mdp (if supplied).

  The function *orig_action* maps actions in this object into actions of the source ConsMDP object. Similarly, *other_action* works for the other object (if makes sense).

add_action(*src*, *distribution*, *label*, *consumption*, *orig_action*, *other_action=None*)
: Create a new action in the product using (src, distribution, label, consumption) and update mappings to *orig_action* and *other_action*.

  Parameters

  - **src** – src in product
  - **distribution** – distribution in product
  - **label** – label of the action
  - **consumption** – consumption in product

- **`orig_action`** – ActionData object from the original mdp
- **`other_action`** – Value to be returned by *other_action* for the new

action. :return: action id in the product

`get_or_create_state`(*orig_s*, *other_s*)
Return state of product based on the two components *(orig_s, other_s)* and create one if it does not exist.

Parameters

- **`orig_s`** – state_id on the original mdp
- **`other_s`** – state of the other component

Returns id of state *(orig_s, other_s)*

`get_state`(*orig_s*, *other_s*)
Return state of product based on the two components *(orig_s, other_s)* if exists and *None* otherwise.

Parameters

- **`orig_s`** – state_id on the original mdp
- **`other_s`** – state of the other component

Returns id of state *(orig_s, other_s)* or None

`new_state`(*orig_s*, *other_s*, *reload=False*, *name=None*)
Create a new product state (orig_s, other_s).

Parameters

- **`orig_s`** – state_id in the original mdp
- **`other_s`** – state of the other component
- **`reload`** – is state reloading? (Bool)
- **`name`** – a custom name of the state, *orig_s,other_s* by default.

Returns id of the new state

`orig_action`(*action*)
Decompose the action from the product to the action in the original mdp.

Parameters **`action`** – ActionData from product (as used in for loops)

Returns ActionData from the original mdp

`other_action`(*action*)
Decompose the action from the product onto the second component, if defined.

Parameters **`action`** – ActionData from product (as used in for loops)

Returns value supplied on creation of *action* (if any), or None

class `fimdp.core.`**`ProductSelector`**(*product_mdp: fimdp.core.ProductConsMDP*)
Bases: `dict`

Selector suited for ConsMDPs whose analysis requires a product ConsMDP.

It combines the approach of CounterSelector with decomposition of the product states and actions into the original components and works for selection even after destruction of the product MDP. The intended use is as follows.

For a MDP called *orig* and some *other* object, we build *product* MDP. The analysis of *product* calls *ProductSelector.update()* with states and actions belonging to the product MDP. For selection of the next action, *ProductSelector.select_action* should be called with *orig_state* and *other_state* that belong to *orig* and *other* and no translation from/to product states is needed. Indeed, the translation happens directly on update.

In short, based on information what action should be picked in a product (supplied using *update), ProductSelector selects actions of the original mdp (at the time 'select_action* is called).

It is implemented as 2-dimensional dict (other × orig) to SelectionRules. The reason for dicts instead of lists is that the product can be sparse.

`copy_values_from`(*other*, *product_state_subset=None*)
Replace values by values from *other* ProductSelector.

If *product_state_subset* is not given (or is None), replace values for all states. Otherwise, replaces only those values that correspond to the given states from the product.

`select_action`(*orig_state*, *other_state*, *energy*)
Return action selected for *orig_state*×*other_state* and *energy*.

`update`(*product_state*, *energy_level*, *product_action*)
For given state of product with components *(orig, other)* update the selection rule for *selector[other][orig]* with *rule[energy]=action* where *action* belongs to *orig* and corresponds to *product_action*.

energy_level' is a lower bound of an interval for which *action* will be selected by *select_action*.

Raises ValueError if *product_action* is not an action of *product_state*

class `fimdp.core.`**`ProductSelectorWrapper`**(*mdp: fimdp.core.ProductConsMDP*, *product_selector=None*)
Bases: `fimdp.core.CounterSelector`

Selector suited for ConsMDPs whose analysis requires a product ConsMDP.

The *ProductSelectorWrapper* is a wrapper around *CounterSelector* built for the product and the *ProductSelectorWrapper* translates the states of the product into their two components *(state, other_state)* and back. The same applies to actions.

The main purpose of the selector is to provide interface that is accessible without the knowledge of the product. Therefore, it selects actions based on:

- state of the original mdp (before product),
- state of the other component, and
- energy level.

The actions returned by *select_action* are actions of the original mdp.

`select_action`(*state*, *other_state*, *energy*)
Return action selected for *state* and *energy*

class `fimdp.core.`**`SelectionRule`**
Bases: `dict`

Selection rule is a partial function: $\rightarrow$ Actions.

Intuitively, a selection according to rule ϕ selects the action that corresponds to the largest value from dom(ϕ) that is not larger than the given energy level.

For dom(ϕ) = $n_1 < n_2 < \ldots$ < n_k and energy level e the selection returns ϕ(n_i) where i is largest integer such that n_i <= e.

`copy`() $\rightarrow$ a shallow copy of D

`select_action`(*energy*)
Select action for given energy level.

Parameters **`energy`** – energy level

Returns action selected by the rule for *energy*

Raise *NoFeasibleActionError* if no action can be selected for the given *energy*

class `fimdp.core.`**`Simulator`**(*strategy*, *num_steps=0*)

Bases: `object`

Class for simulating a strategy object on a ConsMDP.

Picks actions based on given strategy for *num_steps* of simulation steps and stores the state and action history for further analysis. Interface allows for extending simulation and resetting given instance using *simulate* and *reset* methods.

`reset`(*init_state=None*, **args*, ***kwargs*)

Prepare a new simulation with the same strategy.

The arguments are passed to *strategy.reset* functions. We can thus change the initial state or the initial energy in the case of Counter strategies.

If no init_state is given, the previous initial state is reused

`simulate`(*num_steps*)

Continue the simulation for additional *num_steps* steps.

class `fimdp.core.`**`Strategy`**(*mdp*, *init_state=None*, **args*, ***kwargs*)

Bases: `object`

Abstract class that implements the interface for strategies (see the docstring for the *strategy.py* module). It handles the checks for outcomes and alternation of calls to *.next_action* and *.update_state*.

Calls to *.next_action()* and *.update_state(outcome)* should alternate unless *next_action(outcome)* are used exclusively.

`next_action`(*outcome=None*)

Pick the next action to play

Parameters **`outcome`** – sid (state id) or *None* (default *None*) *outcome* must be a successor of the action picked by the last call to *.next_action()*. If defined, update the current state to *outcome*.

Returns action to play

`reset`(*init_state=None*, **args*, ***kwargs*)

Reset the memory and initial state for a new play.

`update_state`(*outcome*)

Tells the strategy that the last action picked by *next_action* was resolved to *outcome*.

Parameters **`outcome`** – sid (state id) *outcome* must be a successor of the action picked by the last call to *.next_action()*.

exception `fimdp.core.`**`WrongCallOrderError`**

Bases: `Exception`

7.1.2 fimdp.distribution module

Module that defines probability distributions and distributions-related functions.

A distribution is a mapping from states (integers) to probability values where the values sum up to 1.

`fimdp.distribution.`**`is_distribution`**(*distribution*)

Checks if the given mapping is a probability distribution (sums up to 1).

Parameters **`distribution`** (*`a mapping from integers to probabilities`*) –

Returns

Return type True if values in *distribution* sum up to 1.

`fimdp.distribution.`**`uniform`**(*destinations*)

Create a uniform distribution for given destinations.

destinations: iterable of states

7.1.3 fimdp.dot module

Core module defining the functions for converting a consumption Markov Decision Process from consMDP model to dot representation and present it.

`class` `fimdp.dot.`**`consMDP2dot`**(*mdp*, *solver=None*, *options=”*)

Bases: `object`

Convert consMDP to dot

`add_incomplete`(*s*)

Adds a dashed line from s to a dummy … node for the given state s.

`add_legend`()

`finish`()

`get_dot`()

`get_state_name`(*s*)

`process_action`(*a*)

`process_state`(*s*)

`start`()

`fimdp.dot.`**`dot_to_svg`**(*dot_str*, *mdp=None*)

Send some text to dot for conversion to SVG.

7.1.4 fimdp.energy_solvers module

Module with energy-aware qualitative solvers for Consumption MDPs

Currently, the supported objectives are:

- minInitCons: reaching a reload state within >0 steps
- safe : survive from *s* forever
- **positiveReachability(T)** [survive and the probability of reaching] some target from T is positive (>0)
- **almostSureReachability(T): survive and the probability of reaching** some target from T is 1
- Büchi(T) : survive and keep visiting T forever (with prob. 1).

The results of a solver for an objective *o* are twofolds:

1. For each state *s* we provide value *o[s]* which is the minimal initial load of energy needed to satisfy the objective *o* from *s*.
2. Corresponding strategy that, given at least *o[s]* in *s* guarantees that *o* is satisfied.

The computed values *o[s]* from 1. can be visualized in the *mdp* object by setting *mdp.EL=solver* and then calling *mdp.show()*.

class `fimdp.energy_solvers.`**`BasicES`**(*mdp*, *cap*, *targets*)

Bases: `object`

Solve qualitative objectives for Consumption MDPs.

This implements the algorithms as described in the paper Qualitative Controller Synthesis for Consumption Markov Decision Processes

Parameters

- **`mdp`** (*) –
- **`cap`** (*) –
- **`targets`** (*) –

`compute`(*objective*)

`get_dot`(*options=''*)

`get_min_levels`(*objective*, *recompute=False*)

Return minimal levels required to satisfy *objective*

Parameters

- **`objective`** (*one of MIN_INIT_CONS, SAFE, POS_REACH, AS_REACH, BUCHI*) –
- **`recompute`** (if *True* forces all computations to be done again) –

`get_selector`(*objective*, *recompute=False*)

Return (and compute) strategy such that it ensures it can handle the minimal levels of energy required to satisfy given objective from each state (if $< \infty$).

objective : one of MIN_INIT_CONS, SAFE, POS_REACH, AS_REACH, BUCHI *recompute* : if *True* forces all computations to be done again

`show`(*options=''*, *max_states=None*)

class `fimdp.energy_solvers.`**`GoalLeaningES`**(*mdp*, *cap*, *targets=None*, *threshold=0*)

Bases: *`fimdp.energy_solvers.BasicES`*

Solver that prefers actions leading to target with higher probability.

This class extends *BasicES* (implementation of CAV'2020 algorithms) by a heuristic that make the strategies more useful for control. The main goal of this class is to create strategies that go to targets quickly.

The solver modifies only the computation of positive reachability computation.

Among action that achieves the minimal _action_value_T, choose the one with the highest probability of hitting the picked successor. The modification is twofold:

1. redefine _action_value_T
2. instead of classical argmin, use pick_best_action that works on tuples (value, probability of hitting good successor).

See more technical description in docstring for _action_value_T.

If threshold is set to value > 0, then we also modify how fixpoint works:

3. Use 2-shot fixpoint computations for positive reachability; the first run ignores successors that can be reached with probability < threshold. The second fixpoint is run with threshold=0 to cover the cases where the below-threshold outcomes only would lead to higher initial loads.

Parameters

- **mdp** (*) –
- **cap** (*) –
- **targets** (*) –
- **threshold** (*) – Successor less likely then *treshold* will be ignored in the first fixpoint.

double_fixpoint (**args*, ***kwargs*)

class fimdp.energy_solvers.**LeastFixpointES** (*mdp*, *cap*, *targets*)

Bases: *fimdp.energy_solvers.BasicES*

Solver that uses (almost) least fixpoint to compute Safe values.

The worst case number of iterations is c_max * `|S|` and thus the worst case complexity is `c_max * |S|^2` steps. The worst case complexity of the largest fixpoint version is ``|S|``^2 iterations and thus ``|S|``^3 steps.

fimdp.energy_solvers.**argmin** (*items*, *func*)

Compute argmin of func on iterable *items*.

Returns (i, v) such that v=func(i) is smallest in *items*.

fimdp.energy_solvers.**largest_fixpoint** (*solver*, *values*, *action_value*, *value_adj=<function <lambda>>*, *skip_state=<function <lambda>>*, *on_update=<function <lambda>>*, *argmin=<function argmin>*)

Largest fixpoint on list of values indexed by states.

Most of the computations of energy levels are, in the end, using this function in one way or another.

The value of a state *s* is a minimum over *action_value(a)* among all possible actions *a* of *s*. Values should be properly initialized (to ∞ or some other value) before calling.

Parameters

- **mdp** (*) –
- **values** (*) –
- **action_value** (*) –

  based on current values in *values*. Takes 2 paramers:

  – action : *ActionData* action of MDP to evaluate

  – values : *list of ints* current values

- **functions that alter the computation** (*) –

  – **value_adj** [*state* × *v* -> *v'* (default *labmda x, v: v*)] Change the value *v* for *s* to *v'* in each iteration (based on the candidate value). For example use for *v* > *capacity* -> ∞ Allows to handle various types of states in a different way.

  – **skip_state** [*state* -> *Bool* (default *lambda x: False*)] If True, stave will be skipped and its value not changed.

  – argmin : function that chooses which action to pick

- **on_upadate** (*) – Arguments are: state × value × action The meaning is for *s* we found new value *v* using action *a*. By default only None is returned.

We have 2 options that help us debug the code using this function. These should be turned on in the respective solver:

- *debug* : print *values* at start of each iteration
- *debug_vis* : display *mdp* using the IPython *display*

`fimdp.energy_solvers.`**`least_fixpoint`**(*solver*, *values*, *action_value*, *value_adj=<function <lambda>>*, *skip_state=None*)

Least fixpoint on list of values indexed by states.

The value of a state *s* is a minimum over *action_value(a)* among all posible actions *a* of *s*. Values should be properly initialized (to ∞ or some other value) before calling.

For safe values the values should be initialized to minInitCons.

Parameters

- **`solver`** (*) –
- **`values`** (*) –
- **`action_value`** (*) –

  based on current values in *values*. Takes 2 paramers:

  – action : *ActionData* action of MDP to evaluate

  – values : *list of ints* current values

- **`functions that alter the computation`** (*) –

  – **value_adj** [*state* × *v* -> *v'* (default *labmda x, v: v*)] Change the value *v* for *s* to *v'* in each iteration (based on the candidate value). For example use for *v* > *capacity* -> ∞ Allows to handle various types of states in a different way.

  – **skip_state** [*state* -> *Bool*] (default *lambda x: values[x] == inf*) If True, stave will be skipped and its value not changed.

We have 2 options that help us debug the code using this function:

- *debug* : print *values* at start of each iteration
- *debug_vis* : display *mdp* using the IPython *display*

`fimdp.energy_solvers.`**`pick_best_action`**(*actions*, *func*)

Compositional argmin and argmax.

Given *func* of type *action* $\rightarrow$ *value* × *prob*, choose action that achieves the lowest *value* with the highest probability over actions with the same value. Which is, choose action with the lowest d=(*value*, 1-*prob*) using lexicographic order.

7.1.5 fimdp.explicit module

`fimdp.explicit.`**`get_MECs`**(*mdp*)

Given an MDP (not necessarly consMDP), compute its maximal-end-components decomposition.

Returns list of mecs (lists).

`fimdp.explicit.`**`product_energy`**(*cmdp*, *capacity*, *targets=[]*)

Explicit encoding of energy into state-space

The state-space of the newly created MDP consists of tuples *(s, e)*, where *s* is the state of the input CMDP and *e* is the energy level. For a tuple-state *(s,e)* and an action $a$ with consumption (in the input CMDP) *c*, all successors of the action *a* in the new MDP are of the form *(s', e-c)* for non-reload states and *(r, capacity)* for reload states.

7.1.6 fimdp.io module

fimdp.io.**consmdp_to_storm_consmdp**(*cons_mdp*, *targets=None*)

Convert a ConsMDP object from FiMDP into a Storm's SparseMDP representation.

The conversion works in reversible way. In particular, it does not encode the energy levels into state-space. Instead, it uses the encoding using rewards.

The reloading and target states (if given) are encoded using state-labels in the similar fashion.

Parameters

- **cons_mdp** – ConsMDP object to be converted
- **targets** – A list of targets (default None). If specified, each state

in this list is labeled with the label *target*. :return: SparseMDP representation from Stormpy of the cons_mdp.

fimdp.io.**encode_to_stormpy**(*cons_mdp*, *capacity*, *targets=None*)

Convert a ConsMDP object from FiMDP into a Storm's SparseMDP representation that is semantically equivalent.

Running analysis on this object should yield the same results as FiMDP. The energy is encoded explicitly into the state space of the resulting MDP.

The target states (if given) are encoded using state-label "target".

Parameters

- **cons_mdp** – ConsMDP object to be converted
- **capacity** – capacity
- **targets** – A list of targets (default None). If specified, each state

in this list is labeled with the label *target*. :return: SparseMDP representation from Stormpy of the cons_mdp.

fimdp.io.**get_state_name**(*model*, *state*)

fimdp.io.**parse_cap_from_prism**(*filename*)

fimdp.io.**prism_to_consmdp**(*filename*, *constants=None*, *state_valuations=True*, *action_labels=True*, *return_targets=False*)

Build a ConsMDP from a PRISM symbolic description using Stormpy.

The model must specify *consumption* reward on each action (choice) and it needs to contain *reload* label.

The following code sets the consumption of each action to *1* and marks each state where the variable *rel* is equal to *1* as a reloading state.

```
>>> rewards "consumption"
>>>   [] true: 1;
>>> endrewards
>>> label "reload" = (rel=1);
```

The dict *constants* must be given if a parametric prism model is to be read. It must defined all unused constants of the parametric model that affect the model's state space. On the other hand, it must not be defined if the model is not parametric. The format of the dictionary is *{ "constant_name" : constant_value }* where constant value is either an integer or a string that contains a name of other constant.

Parameters

- **`filename`** – Path to the PRISM model. Must be an mdp.
- **`constants`** – Dictionary for uninitialized constant initialization.
- **`state_valuations`** – If True (default), set the valuation of states as

Type constants: dict[str(constant_names) -> int/str(constant_names)]

names in the resulting ConsMDP. :param action_labels: If True (default), copies the choice labels in the PRISM model into the ConsMDP as action labels.

Parameters **`return_targets`** – If True (default False), return also the list of

states labeled by the label *target*.

Returns ConsMDP object for the given model, or *ConsMDP, targets* if *return_targets*

`fimdp.io.`**`storm_sparsemdp_to_consmdp`**(*sparse_mdp*, *state_valuations=True*, *action_labels=True*, *return_targets=False*)

Convert Storm's sparsemdp model to ConsMDP.

Parameters **`sparse_mdp`** – Stormpy sparse representation of MDPs. The model must

represent an MDP and it needs to contain action-based reward called *consumption* (needs to be defined for each action) and some states need to be labeled by *reload* label. In particular, *reload* must be a valid state label. :type sparse_mdp: stormpy.storage.storage.SparseMdp

Parameters **`state_valuations`** – if True (default), record the state valuations

(for models built from symbolic description) into state names of the ConsMDP. It is ignored if the *sparse_mdp* does not contain the state valuations. :type state_valuations: Bool

Parameters **`action_labels`** – If True (default), actions are labeled by labels

stored in *sparse_mdp.choice_labeling* (if it is present in the model). Otherwise, the actions are labeled by thier id. :type action_labels: Bool

Parameters **`return_targets`** – If True (default False), parse also target states

(from labels).

Returns *ConsMDP* object or *ConsMDP, list of targets* if *return_targets*

7.1.7 fimdp.labeled module

7.1.8 fimdp.mincap_solvers module

Find minimal capacity needed for given starting location and target location.

`fimdp.mincap_solvers.`**`bin_search`**(*mdp*, *init_loc*, *target_locs*, *starting_capacity=100*, *objective=4*, *max_starting_load=None*)

Search for min. capacity by brute-force using binary search.

For given starting location (*init_loc*) and a set (iterable) of goal states (*target_locs*) in CMDP *mdp*, compute minimal capacity needed to fulfill the objective (Büchi by default) from the the starting location. Please not that giving more targets in target_locs means that we can choose 1 of them only and not visit the rest.

The search starts from *capacity=100* by default. This can be changed by setting *starting_capacity*.

If *max_starting_load* is given, don't consider capacities for which we need more than the given value from the starting location.

The target_locs can be either an integer ID of a state or an *iterable* of those.

Objective can be either *energy_solver.BUCHI* or *energy_solver.AS_REACH*. Default is *BUCHI*.

7.1.9 fimdp.objectives module

Objectives that can be used in FiMDP.

- **MIN_INIT_CONS** stands for _minimal initial **consumption_**. It is the minimal energy needed to surely reach some reloading state from each state.
- **SAFE** stands for _survival_. A SAFE strategy σ guarantees that all plays according to σ will never deplete energy with given capacity. *

POS_REACH stands for _positive reachability_. This subsumes survival and moreover, there the probability to reach the specified target set is larger than 0.

AS_REACH stands for _almost-sure reachability_. Similar to positive reachability, but here the probability is equal to 1.

BUCHI stands for _almost-sure Büchi_. The probability that the target set will be visited infinitely often is equal to 1.

7.1.10 fimdp.utils module

class fimdp.utils.**Duplicator**(*mdp: fimdp.core.ConsMDP*, *init_state=0*, *max_states=inf*, *preserve_names=True*, *solver=None*)

Bases: `object`

Makes an independent (deep) copy of the given consMDP.

The *max_states* parameter is used to limit the number of states that will be used in the new mdp. If set and used, the resulting ConsMDP will have the *.incomplete* attribute which stores the set of states whose successors could not be fully built.

The function starts building the copy from the init_state and builds only the part reachable from this state

run()

fimdp.utils.**copy_consmdp**(*mdp*, *init_state=0*, *max_states=inf*, *preserve_names=True*, *solver=None*)

7.2 License

7.2.1 The MIT License

Copyright (c) 2019, 2020 UT Austin.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER 8

Indices and Search

- genindex
- search

Python Module Index

Index