Package: markovDP (via r-universe)

November 3, 2026

Title Infrastructure for Discrete-Time Markov Decision Processes (MDP)

Version 0.99.0

Date 2026-xx-xx

Description Provides the infrastructure to work with Markov Decision Processes (MDPs) in R. The focus is on convenience in formulating MDPs, the support of sparse representations (using sparse matrices, lists and data.frames) and visualization of results. Some key components are implemented in C++ to speed up computation. Several popular solvers are implemented.

License GPL (>=3)

URL https://github.com/mhahsler/markovDP, http://michael.hahsler.net/markovDP/

BugReports https://github.com/mhahsler/markovDP/issues

Depends R (>= 3.5.0)

Imports fastmap, foreach, igraph, lpSolve, Matrix, MatrixExtra, methods, progress, Rcpp, stats

Suggests doParallel, gifski, knitr, rmarkdown, testthat (>= 3.0.0), visNetwork

LinkingTo Rcpp

VignetteBuilder knitr

Classification/ACM G.4, G.1.6, I.2.6

Copyright Copyright (C) Michael Hahsler.

Encoding UTF-8

Roxygen list(markdown = TRUE)

Config/testthat/edition 3

SystemRequirements C++17

Collate 'AAA_check_installed.R' 'AAA_colors.R' 'AAA_foreach_helper.R' 'AAA_near_equal.R' 'AAA_nodots.R' 'AAA_package.R' 'AAA_progress.R' 'AAA_sample_sparse.R' 'AAA_shorten.R'

'AAA_which_max_random.R' 'Cliff_walking.R' 'DynaMaze.R' 'MDP.R'
'MDPSample.R' 'Maze.R' 'Q_values.R' 'RcppExports.R'
'Windy_gridworld.R' 'absorbing_states.R'
'accessors_transitions.R' 'accessors_reward.R' 'accessors.R'
'act.R' 'action.R' 'action_state_helpers.R'
'available_actions.R' 'bellman_operator.R'
'check_and_fix_MDP.R' 'convergence_horizon.R'
'find_reachable_states.R' 'greedy.R' 'gridworld.R'
'linear_function_approximation.R' 'policy.R'
'policy_evaluation.R' 'policy_evaluation_LP.R'
'policy_evaluation_MC.R' 'policy_evaluation_bellman.R'
'reachable_states.R' 'regret.R' 'return.R'
'round_stochchastic.R' 'sample_MDP.R' 'sample_MDPSample.R'
'schedule.R' 'solve_MDP.R' 'solve_MDP_APPROX.R'
'solve_MDP_DP.R' 'solve_MDP_LP.R' 'solve_MDP_MC.R'
'solve_MDP_PG.R' 'solve_MDP_SAMP.R' 'solve_MDP_TD.R'
'sparse_helpers.R' 'start.R' 'transformation.R'
'transition_graph.R' 'unreachable_states.R' 'value_function.R'
'visit_probability.R' 'zzz.R'

Config/roxygen2/version 8.1.0

Config/pak/sysreqs libglpk-dev libxml2-dev

Repository https://mhahsler.r-universe.dev

Date/Publication 2026-10-04 17:01:26 UTC

RemoteUrl https://github.com/mhahsler/markovDP

RemoteRef HEAD

RemoteSha b3d6b301cc2535bdb109bdddcfdd2164c44eaf0f

Contents

absorbing_states . . . 3
act . . . 5
action . . . 6
action_state_helpers . . . 7
available_actions . . . 9
bellman_update . . . 10
Cliff_walking . . . 12
colors . . . 13
convergence_horizon . . . 14
DynaMaze . . . 15
expected_return . . . 16
find_reachable_states . . . 17
greedy_action . . . 21
gridworld . . . 22
linear_function_approximation . . . 30
Maze . . . 33
MDP . . . 36

MDPSample 40
policy 43
policy_evaluation 46
Q_values 49
reachable_states 51
regret 56
round_stochastic 59
sample_MDP 60
sample_MDP.MDPSample 62
schedule 64
solve_MDP 66
solve_MDP_APPROX 69
solve_MDP_DP 73
solve_MDP_LP 77
solve_MDP_MC 79
solve_MDP_PG 82
solve_MDP_SAMP 84
solve_MDP_TD 86
start 90
transformation 92
transition_graph 94
transition_matrix 95
unreachable_states 99
value_function 100
visit_probability 102
Windy_gridworld 103

Index **106**

`absorbing_states` *Absorbing States*

Description

Find absorbing states using the transition model.

Usage

```
absorbing_states(model, state = NULL, ...)

## S3 method for class 'MDP'
absorbing_states(
  model,
  state = NULL,
  sparse = "states",
  use_precomputed = TRUE,
  ...
)
```

```
## S3 method for class 'MDPSample'
absorbing_states(
  model,
  state = NULL,
  sparse = "features",
  use_precomputed = TRUE,
  ...
)
```

Arguments

model	a MDP object.
state	a single state to check. This can be much faster if the model contains a transition model implemented as a function. NULL means all states are checked.
...	further arguments are passed on.
sparse	logical; if return a sparse logical vector?
use_precomputed	logical; should precomputed values in the MDP be used?

Details

The function absorbing_states() checks if a state or a set of states are absorbing (terminal states). A state is absorbing if there is for all actions a probability of 1 for staying in the state.

Value

absorbing_states() returns a logical vector indicating if the states are absorbing (terminal).

Author(s)

Michael Hahsler

See Also

Other MDP: MDP(), act(), action_state_helpers, available_actions(), find_reachable_states(), reachable_states(), sample_MDP(), sample_MDP.MDPSample(), start, transition_graph(), transition_matrix(), unreachable_states()

Other MDPSample: MDPSample(), act(), action_state_helpers, reachable_states(), sample_MDP.MDPSample(), solve_MDP_PG(), start

Examples

```
data(Maze)

gw_matrix(Maze)
gw_matrix(Maze, what = "labels")
gw_matrix(Maze, what = "absorbing")
```

```
# -1 and +1 are absorbing states
absorbing_states(Maze)
absorbing_states(Maze, sparse = FALSE)
absorbing_states(Maze, sparse = "states")

# check individual states
absorbing_states(Maze, "s(1,1)")
absorbing_states(Maze, "s(1,4)")
```

`act` *Perform an Action*

Description

Performs an action in a state and returns the new state and reward.

Usage

```
act(model, state, action, fast = FALSE, ...)

## S3 method for class 'MDPModel'
act(model, state, action = NULL, fast = FALSE, ...)

## S3 method for class 'MDPSample'
act(model, state, action, fast = FALSE, ...)
```

Arguments

`model`	an MDP model.
`state`	the current state.
`action`	the chosen action. If the action is not specified (`NULL`) and the MDP model contains a policy, then the action is chosen according to the policy.
`fast`	logical; if `TRUE` then extra state id to label conversions are avoided.
`...`	if action is unspecified, then the additional parameters are passed on to `action()` to determine the action using the model's policy.

Value

a named list with the `reward` and the next state `state_prime`.

Author(s)

Michael Hahsler

See Also

Other MDP: MDP(), absorbing_states(), action_state_helpers, available_actions(), find_reachable_states(), reachable_states(), sample_MDP(), sample_MDP.MDPSample(), start, transition_graph(), transition_matrix(), unreachable_states()

Other MDPSample: MDPSample(), absorbing_states(), action_state_helpers, reachable_states(), sample_MDP.MDPSample(), solve_MDP_PG(), start

Examples

```
data(Maze)

act(Maze, "s(1,3)", "right")

# solve the maze and then ask for actions using the policy
sol <- solve_MDP(Maze)
act(sol, "s(1,3)")

# make the policy in sol epsilon-soft and ask 10 times for the action
replicate(10, act(sol, "s(1,3)", epsilon = .2))
```

---

action *Choose an Action Given a Policy*

Description

Returns an action given a deterministic policy. The policy can be made epsilon-soft.

Usage

```
action(model, state, epsilon = 0, epoch = 1, as = "factor", ...)
```

Arguments

model	a solved MDP.
state	the state.
epsilon	make the policy epsilon soft.
epoch	what epoch of the policy should be used. Use 1 for converged policies.
as	string, format for returning the action (e.g., "factor", "id", "label").
...	further parameters are passed on.

Value

The name of the optimal action as a factor.

Author(s)

Michael Hahsler

See Also

Other policy: expected_return(), greedy_action(), policy(), policy_evaluation(), regret(), visit_probability()

Examples

```
data("Maze")
Maze

sol <- solve_MDP(Maze)
policy(sol)

action(sol, state = "s(1,3)")

## choose from an epsilon-soft policy
table(replicate(100, action(sol, state = "s(1,3)", epsilon = 0.1)))
```

---

action_state_helpers *Conversions for Action and State IDs and Labels*

Description

Several helper functions to convert state and action (integer) IDs to labels and vice versa.

Usage

```
normalize_state(state, model, as = "factor")

normalize_state_id(state, model)

normalize_state_label(state, model)

normalize_state_features(state, model = NULL)

normalize_action(action, model, as = "factor")

normalize_action_id(action, model)

normalize_action_label(action, model)

state2features(state)

features2state(x)

s(...)

get_state_features(model)
```

Arguments

state	a state in any format
model	an MDP model
as	character; specifies the desired output format
action	an action in any format
x	a state feature vector or a matrix of state feature vectors as rows.
...	features that should be converted into a row vector used to describe a state.

Details

`normalize_state()` and `normalize_action()` convert labels or ids into a desired standard representation. If only the label or the integer id (i.e., the index) is needed, the additional functions can be used. These are typically a lot faster.

To support a factored state representation as feature vectors, `state2features()`, `feature2states()`, and `get_state_features()` are provided. `get_state_features()` is only available if the model explicitly stores a finite state space.

Note: A factored state is represented as a **row** vector (matrix with a single row) for a single state (conveniently created via `s()`) or a matrix with row vectors for a set of states are used. State labels are constructed in the form `s(feature1, feature2, ...)`. Factored state representation is used for value function approximation (see `solve_MDP_APPROX()`) and for MDPSample to describe MDP's via a transition function between factored states.

Value

Functions ending in

- `_factor` return a factor,
- `_id` return an integer id,
- `_label` return a character string,
- `_features` return a state feature matrix,
- no ending return the type specified with parameter `as`.

Other functions:

- `state2features()` returns a feature vector/matrix.
- `features2state(x)` returns a state label in the format `s(feature list)`.
- `s()` returns a state features row vector.

See Also

Other MDP: `MDP()`, `absorbing_states()`, `act()`, `available_actions()`, `find_reachable_states()`, `reachable_states()`, `sample_MDP()`, `sample_MDP.MDPSample()`, `start`, `transition_graph()`, `transition_matrix()`, `unreachable_states()`

Other MDPSample: `MDPSample()`, `absorbing_states()`, `act()`, `reachable_states()`, `sample_MDP.MDPSample()`, `solve_MDP_PG()`, `start`

Examples

```
data(Maze)

# states
normalize_state(1, Maze)
normalize_state(1, Maze, as = "id")
normalize_state(1, Maze, as = "label")
normalize_state(1, Maze, as = "features")

get_state_features(Maze)

# actions
normalize_action(1, Maze)
normalize_action(1, Maze, as = "id")
normalize_action(1, Maze, as = "label")

# state label to feature conversion
state2features("s(1,1)")
s(1,1)
```

---

available_actions *Available Actions in a State*

Description

Determine the set of actions available in a state.

Usage

```
available_actions(model, state, neg_inf_reward = TRUE, stay_in_place = FALSE)
```

Arguments

model	a MDP object.
state	a character vector specifying the states.
neg_inf_reward	logical; consider an action that produced -Inf reward to all end states unavailable?
stay_in_place	logical; consider an action that results in the same state with a probability of 1 as unavailable. Note that this means that absorbing states have no available action!

Details

Unavailable actions are modeled as actions that have an immediate reward of -Inf in the reward function. For a maze, also actions that do not change the state can be considered unavailable.

Value

a character vector with the available actions.

a vector with the available actions.

Author(s)

Michael Hahsler

See Also

Other MDP: MDP(), absorbing_states(), act(), action_state_helpers, find_reachable_states(), reachable_states(), sample_MDP(), sample_MDP.MDPSample(), start, transition_graph(), transition_matrix(), unreachable_states()

Examples

```
data(DynaMaze)
gw_plot(DynaMaze)

# The following actions are always available:
DynaMaze$actions

# only right and down is unavailable for s(1,1) because they
#   make the agent stay in place.
available_actions(DynaMaze, state = "s(1,1)", stay_in_place = TRUE)

# An action that leaves the grid currently is allowed but does not do
# anything.
act(DynaMaze, "s(1,1)", "up")
```

---

bellman_update *Bellman Update and Bellman operator*

Description

Update the value function with a Bellman update.

Usage

```
bellman_update(model, V)

bellman_operator(model, pi, V)
```

Arguments

model	an MDP problem specification.
V	a vector representing the value function. A single 0 can be used as a shorthand for a value function with all 0s.
pi	a policy as a data.frame with at least columns for states and action. If NULL, then the policy in model is used.

Details

The Bellman update updates a value function given the model by applying the Bellman equation as an update rule for each state:

$$v_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}(s)} \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v_k(s')]$$

The Bellman update moves the estimated value function V closer to the optimal value function v_*.

The Bellman operator B_π updates a value function given the model, and a policy π:

$$(B_\pi v)(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v(s')]$$

The Bellman error is $\delta = B_\pi v - v$. The Bellman operator reduces the Bellman error and moves the value function closer to the fixed point of the true value function:

$$v_\pi = B_\pi v_\pi.$$

Value

a list with the updated state value vector U and the taken actions pi.

Author(s)

Michael Hahsler

References

Sutton, R. S., Barto, A. G. (2020). Reinforcement Learning: An Introduction. Second edition. The MIT Press.

See Also

Other value_function: `Q_values()`, `value_function()`

Examples

```
data(Maze)
Maze

# single Bellman update from an all-zero value function
bellman_update(Maze, V = 0)

# perform simple value iteration for 10 iterations
V <- 0
for (i in seq(10))
  V <- bellman_update(Maze, V)$V

V
```

`Cliff_walking` *Cliff Walking Gridworld MDP*

Description

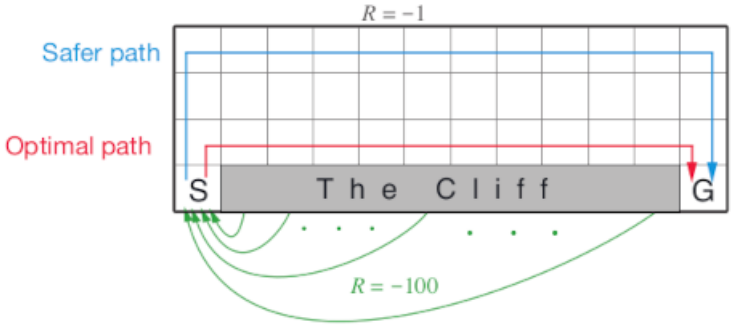
The cliff walking gridworld MDP example from Chapter 6 of the textbook "Reinforcement Learning: An Introduction."

Format

An object of class MDP.

Details

The cliff walking gridworld has the following layout:



The gridworld is represented as a 4 x 12 matrix of states. The states are labeled with their x and y coordinates. The start state is in the bottom left corner. Each action has a reward of -1, falling off the cliff has a reward of -100 and returns the agent back to the start. The episode is finished once the agent reaches the absorbing goal state in the bottom right corner. No discounting is used (i.e., $\gamma = 1$).

References

Richard S. Sutton and Andrew G. Barto (2018). Reinforcement Learning: An Introduction Second Edition, MIT Press, Cambridge, MA.

See Also

Other MDP_examples: `DynaMaze`, `MDP()`, `Maze`, `Windy_gridworld`

Other gridworld: `DynaMaze`, `Maze`, `Windy_gridworld`, `gridworld`

Examples

```
data(Cliff_walking)
Cliff_walking

gw_matrix(Cliff_walking)
gw_matrix(Cliff_walking, what = "labels")

# The Goal is an absorbing state
absorbing_states(Cliff_walking, sparse = "states")

# visualize the transition graph
gw_plot_transition_graph(Cliff_walking)

# solve using different methods
sol <- solve_MDP(Cliff_walking)
sol
policy(sol)
gw_plot(sol)
```

---

colors *Default Colors for Visualization*

Description

Default discrete and continuous colors used in the package markovDP.

Usage

```
colors_discrete(n, col = NULL)

colors_continuous(val, col = NULL)
```

Arguments

n	number of states.
col	custom color palette. colors_discrete() uses the first n colors. colors_continuous() uses the given colors to calculate a palette (see grDevices::colorRamp()). The default is a blue-red color ramp.
val	a vector with values to be translated to colors.

Value

colors_discrete() returns a color palette and colors_continuous() returns the colors associated with the supplied values.

See Also

Other utilities: round_stochastic()

Examples

```
colors_discrete(5)

colors_continuous(runif(10))
```

`convergence_horizon` *Estimate the Convergence Horizon for an Infinite-Horizon MDP*

Description

Many sampling-based methods require a finite horizon. For infinite horizons, discounting leads to convergences during a finite horizon. This function estimates the number of steps till convergence using rules of thumb.

Usage

```
convergence_horizon(model, delta = 0.001, n_updates = 10)
```

Arguments

`model`	an MDP model.
`delta`	maximum update error.
`n_updates`	integer; average number of time each state is updated.

Details

The horizon is estimated differently for the discounted and the undiscounted case.

Discounted Case:

The effect of the largest reward $R_{\max}$ update decreases with t as $\delta_t = \gamma^t R_{\max}$. The convergence horizon is estimated as the smallest t for which $\delta_t < \delta$.

Undiscounted Case:

For the undiscounted case, episodes end when an absorbing state is reached. It cannot be guaranteed that a model will reach an absorbing state. To avoid infinite loops, we set the maximum horizon such that each entry in the Q-table is on average updated `n_updates` times. This is a very rough rule ot thumb.

Value

An estimated convergence horizon.

Author(s)

Michael Hahsler

See Also

Other solver: schedule, solve_MDP(), solve_MDP_APPROX(), solve_MDP_DP(), solve_MDP_LP(), solve_MDP_MC(), solve_MDP_PG(), solve_MDP_SAMP(), solve_MDP_TD()

Examples

```
data(Maze)
Maze

convergence_horizon(Maze)

# make the Maze into a discounted problem where future rewards count less.
Maze_discounted <- Maze
Maze_discounted$discount <- .9
Maze_discounted

convergence_horizon(Maze_discounted)
```

DynaMaze *The Dyna Maze*

Description

The Dyna Maze from Chapter 8 of the textbook "Reinforcement Learning: An Introduction."

Format

An object of class MDP.

Details

The simple 6x9 maze with a few walls.

References

Richard S. Sutton and Andrew G. Barto (2018). Reinforcement Learning: An Introduction Second Edition, MIT Press, Cambridge, MA.

See Also

Other MDP_examples: Cliff_walking, MDP(), Maze, Windy_gridworld

Other gridworld: Cliff_walking, Maze, Windy_gridworld, gridworld

Examples

```
data(DynaMaze)

DynaMaze

gw_matrix(DynaMaze)
gw_matrix(DynaMaze, what = "labels")

gw_plot_transition_graph(DynaMaze)
```

`expected_return` *Calculate the Expected Return of a Policy*

Description

This function calculates the expected total return for an MDP policy given a start state (distribution). The value is calculated using the value function stored in the MDP solution.

Usage

```
expected_return(model, ...)

## S3 method for class 'MDP'
expected_return(model, start = NULL, method = "solution", ...)
```

Arguments

`model`	a solved MDP object.
`...`	further arguments are passed on to `policy_evaluation()` or `sample_MDP()`.
`start`	specification of the current state (see argument start in MDP for details). By default the start state defined in the model as start is used. Multiple states can be specified as rows in a matrix.
`method`	`"solution"` uses the converged value function stored in the solved model, `"policy_evaluation"` estimates the value function, and `"sample"` calculates the average return by sampling episodes from the model.

Details

The return is typically calculated using the value function of the solution. If these are not available, then `sample_MDP()` is used instead with a warning.

Value

`expected_return()` returns a vector of returns, one for each start state if a matrix is specified.

`state`	start state to calculate the return for. If `NULL` then the start state of model is used.

Author(s)

Michael Hahsler

See Also

Other policy: action(), greedy_action(), policy(), policy_evaluation(), regret(), visit_probability()

Examples

```
data("Maze")
Maze
gw_matrix(Maze)

sol <- solve_MDP(Maze)
policy(sol)

# return for the start state s(3,1) specified in the model
expected_return(sol)

# return for starting next to the goal at s(1,3)
expected_return(sol, start = "s(1,3)")

# expected return when we start from a random state as returned from the solver
expected_return(sol, start = "uniform")

# estimate the return using sampling following the policy
expected_return(sol, method = "sample", start = "uniform", n = 10000, horizon = 1000)
```

---

find_reachable_states *Find Reachable State Space from a Transition Model Function*

Description

Finds the reachable state space from a transition model function that takes the arguments `model`, `action`, `start.state` and returns a named vector with the probabilities for the resulting end.states (typically only the ones with a probability greater than 1).

Usage

```
find_reachable_states(
  transition_model,
  start_state,
  actions,
  model = NULL,
  horizon = Inf,
  progress = TRUE
)
```

Arguments

start_state	labels of the start states.
actions	a vector with the available actions.
model	if needed, the model passed on to the transition model function.
horizon	only return states reachable in the given horizon.
progress	logical; show a progress bar?
transition_model	a transition function (see details for requirements).

Details

The function performs a (depth-limited) depth-first traversal of the search state space and returns a vector with the names of all encountered states. This vector can be used as the states for creating an MDP model.

The transition function needs to be a function with the argument list `model`, `action`, `start.state` which returns named vector only containing the non-zero probabilities named by the corresponding end state. A partial model that contains `actions` and `start.state` will be supplied to the function.

Value

a character vector with all reachable states.

Author(s)

Michael Hahsler

See Also

Other MDP: `MDP()`, `absorbing_states()`, `act()`, `action_state_helpers`, `available_actions()`, `reachable_states()`, `sample_MDP()`, `sample_MDP.MDPSample()`, `start`, `transition_graph()`, `transition_matrix()`, `unreachable_states()`

Examples

```
# define an MDP for Tic-Tac-Toe

# state description: matrix with the characters _, x, and o
#                    can be converted into a label of 9 characters

# set of actions
A <- as.character(1:9)

# helper functions
ttt_empty_board <- function() matrix('_', ncol = 3, nrow = 3)

ttt_state2label <- function(state) paste(state, collapse = '')

ttt_label2state <- function(label) matrix(strsplit(label, "")[[1]],
```

```
                                          nrow = 3, ncol = 3)

ttt_available_actions <- function(state) {
  if (length(state) == 1L) state <- ttt_label2state(state)
  which(state == "_")
}

ttt_result <- function(state, player, action) {
  if (length(state) == 1L) state <- ttt_label2state(state)

  if (state[action] != "_")
    stop("Illegal action.")

  state[action] <- player
  state
}

ttt_terminal <- function(state) {
  if (length(state) == 1L) state <- ttt_label2state(state)

  # Check the board for a win and return one of
  # 'x', 'o', 'd' (draw), or 'n' (for next move)
  win_possibilities <- rbind(state,
                             t(state),
                             diag(state),
                             diag(t(state)))

  wins <- apply(win_possibilities, MARGIN = 1, FUN = function(x) {
    if (x[1] != '_' && length(unique(x)) == 1) x[1]
    else '_'
  })

  if (any(wins == 'x'))
    return('x')

  if (any(wins == 'o'))
    return('o')

  # Check for draw
  if (sum(state == '_') < 1)
    return('d')

  return('n')
}

# define the transition function:
#     * return a probability vector for an action in a start state
#     * we define the special states 'win', 'loss', and 'draw'
P <- function(model, action, start.state) {
  action <- as.integer(action)

  # absorbing states
  if (start.state %in% c('win', 'loss', 'draw', 'illegal')) {
```

```
    return(structure(1, names = start.state))
  }

  # avoid illegal action by going to the very expensive illegal state
  if (!(action %in% ttt_available_actions(start.state))) {
    return(structure(1, names = "illegal"))
  }

  # make x's move
  next_state <- ttt_result(start.state, 'x', action)

  # terminal?
  term <- ttt_terminal(next_state)
  if (term == 'x') {
    return(structure(1, names = "win"))
  } else if (term == 'o') {
    return(structure(1, names = "loss"))
  } else if (term == 'd') {
    return(structure(1, names = "draw"))
  }

  # it is o's turn
  actions_of_o <- ttt_available_actions(next_state)
  possible_end_states <- lapply(
    actions_of_o,
    FUN = function(a)
      ttt_result(next_state, 'o', a)
  )

  # fix terminal states
  term <- sapply(possible_end_states, ttt_terminal)
  possible_end_states <- sapply(possible_end_states, ttt_state2label)
  possible_end_states[term == 'x'] <- 'win'
  possible_end_states[term == 'o'] <- 'loss'
  possible_end_states[term == 'd'] <- 'draw'

  possible_end_states <- unique(possible_end_states)

  return(structure(rep(1 / length(possible_end_states),
                       length(possible_end_states)),
                   names = possible_end_states))
}

# define the reward
R <- rbind(
  R_(                    value = 0),
  R_(end.state = 'win',  value = +1),
  R_(end.state = 'loss', value = -1),
  R_(end.state = 'draw', value = +.5),
  R_(end.state = 'illegal', value = -Inf),
  # Note: there is no more reward once the agent is in a terminal state
  R_(start.state = 'win',  value = 0),
  R_(start.state = 'loss', value = 0),
```

```
  R_(start.state = 'draw', value = 0),
  R_(start.state = 'illegal', value = 0)
)

# start state
start <- ttt_state2label(ttt_empty_board())
start

# find the reachable state space
S <- union(c('win', 'loss', 'draw', 'illegal'),
          reachable_states(P, start_state = start, actions = A))
head(S)

tictactoe <- MDP(S, A, P, R, discount = 1, start = start, name = "TicTacToe")
tictactoe
```

---

greedy_action *Greedy Actions and Policies*

Description

Extract a greedy policy or select a greedy action from a solved model or a Q matrix.

Usage

```
greedy_action(x, s, Q = NULL, epsilon = 0, as = "factor", prob = FALSE)

greedy_policy(x)
```

Arguments

x	a solved MDP model or a Q matrix.
s	a state.
Q	an optional Q-matrix.
epsilon	an epsilon > 0 applies an epsilon-greedy policy.
as	string, format for returning the action (e.g., "factor", "id", "label").
prob	logical; return a probability distribution over the actions.

Value

- greedy_action() returns the action with the highest q-value for state s. If prob = TRUE, then a vector with the probability for each action is returned.
- greedy_policy() returns a data.frame with the policy.

greedy_policy() returns the greedy policy given Q.

Author(s)

Michael Hahsler

References

Sutton, R. S., Barto, A. G. (2020). Reinforcement Learning: An Introduction. Second edition. The MIT Press.

See Also

Other policy: action(), expected_return(), policy(), policy_evaluation(), regret(), visit_probability()

Examples

```
data(Maze)
Maze

# create a random policy and calculate q-values
pi_random <- random_policy(Maze)
pi_random

V <- policy_evaluation(Maze, pi_random)
V

# calculate Q values
Q <- Q_values(Maze, V)
Q

# get the greedy policy form the Q values
pi_greedy <- greedy_policy(Q)
pi_greedy
Maze_with_policy <- add_policy(Maze, pi_greedy)
gw_plot(Maze_with_policy, main = "Maze: Greedy Policy")

# find the greedy/ epsilon-greedy action for the top-left corner state
greedy_action(Maze, "s(1,1)", Q, epsilon = 0, prob = FALSE)
greedy_action(Maze, "s(1,1)", Q, epsilon = 0, prob = TRUE)
greedy_action(Maze, "s(1,1)", Q, epsilon = .1, prob = TRUE)

# we can also specify a model with a policy and use the internal Q-values
greedy_action(Maze_with_policy, "s(1,1)", epsilon = .1, prob = TRUE)
```

---

gridworld *Helper Functions for Gridworld MDPs*

Description

Helper functions for gridworld MDPs to convert between state names and gridworld positions, and for visualizing policies.

Usage

```
gw_init(
  dim,
  actions = c("up", "right", "down", "left"),
  start = NULL,
  goal = NULL,
  absorbing_states = NULL,
  blocked_states = NULL,
  state_labels = list()
)

gw_s2rc(s)

gw_rc2s(rc)

gw_matrix(model, epoch = 1L, what = "states")

gw_plot(
  model,
  epoch = 1L,
  actions = "character",
  states = TRUE,
  index = FALSE,
  labels = TRUE,
  impossible_actions = FALSE,
  main = NULL,
  cex = 1,
  offset = 0.5,
  lines = TRUE,
  contour = FALSE,
  col = hcl.colors(100, "YlOrRd", rev = TRUE),
  blocked_col = "gray20",
  ...
)

gw_plot_transition_graph(
  x,
  remove.loops = TRUE,
  vertex.color = "gray",
  vertex.shape = "square",
  vertex.size = 10,
  vertex.label = NA,
  edge.arrow.size = 0.3,
  margin = 0.2,
  main = NULL,
  ...
)
```

```
gw_animate(model, method, n, zlim = NULL, continue = FALSE, ...)

gw_transition_model(model, action, start.state)

gw_transition_model_sparse(model, action, start.state)

gw_transition_model_named(model, action, start.state)

gw_transition_model_end_state(model, action, start.state, end.state)

gw_maze_MDP(
  dim,
  start,
  goal,
  walls = NULL,
  actions = c("up", "right", "down", "left"),
  goal_reward = 100,
  step_cost = 1,
  restart = FALSE,
  discount = 1,
  horizon = Inf,
  info = NULL,
  normalize = FALSE,
  access = "model",
  name = "A Maze"
)

gw_random_maze(
  dim,
  wall_prob = 0.2,
  start = NULL,
  goal = NULL,
  normalize = FALSE
)

gw_read_maze(file, discount = 1, restart = FALSE, name = "Maze")

gw_path(model, start = NULL, goal = NULL, horizon = NULL)
```

Arguments

dim	vector of length two with the x and y extent of the gridworld.
actions	how to show actions. Options are: simple "character", "unicode" arrows (needs to be supported by the used font), "label" of the action, and "none" to suppress showing the action.
start, goal	start and goal states. If NULL then the states specified in the model are used.
absorbing_states	a vector with state labels for absorbing states.

blocked_states	a vector with state labels for unreachable states. These states will be excluded.
state_labels	a list with labels for states. The element names need to be state names.
s	a state label or a vector of labels.
rc	a vector of length two with the row and column coordinate of a state in the gridworld matrix. A matrix with one state per row can be also supplied.
model, x	a solved gridworld MDP.
epoch	epoch for unconverged finite-horizon solutions.
what	What should be returned in the matrix. Options are: "states", "index", "labels", "values", "actions", "absorbing", and "unreachable".
states	logical; show state names.
index	logical; show the state indices.
labels	logical; show state labels.
impossible_actions	logical; show the value and the action for absorbing states.
main	a main title for the plot. Defaults to the name of the problem.
cex	expansion factor for the action.
offset	move the state labels out of the way (in fractions of a character width).
lines	logical; draw lines to separate states.
contour	logical; add value function contours.
col	a colors for the utility values.
blocked_col	a color used for blocked states. Use NA for no color.
...	further arguments are passed on to igraph::plot.igraph().
remove.loops	logical; do not show transitions from a state back to itself.
vertex.color, vertex.shape, vertex.size, vertex.label, edge.arrow.size	see igraph::igraph.plotting for details. Set vertex.label = NULL to show the state labels on the graph.
margin	a single number specifying the margin of the plot. Can be used if the graph does not fit inside the plotting area.
method	an MDP solution method for solve_MDP().
n	number of iterations to animate.
zlim	limits for visualizing the state value.
continue	logical; continue solving a solution.
action, start.state, end.state	parameters for the transition function.
walls	a vector with state labels for walls. Walls will become unreachable states.
goal_reward	reward to transition to the goal state.
step_cost	cost of each action that does not lead to the goal state.
restart	logical; if TRUE then the problem automatically restarts when the agent reaches the goal state.

`discount`, `horizon`
: MDP discount factor, and horizon.

`info`
: A list with additional information. Has to contain the gridworld dimensions as element `dim` and can be created using `gw_init()`.

`normalize`
: logical; should the description be normalized for faster access using `normalize_MDP()`.

`access`
: string; create an MDP with `"model"` or `"sample"` access.

`name`
: a string to identify the MDP problem.

`wall_prob`
: probability to make a tile a wall.

`file`
: filename for a maze text file.

Details

Gridworlds are implemented with state names `s(row,col)`, where `row` and `col` are locations in the matrix representing the gridworld. The default actions are `"up"`, `"right"`, `"down"`, and `"left"`.

Creating a Gridworld:

`gw_init()` initializes a new gridworld creating a matrix of states with the given dimensions. Other action names can be specified, but they must have the same effects in the same order as above. Blocked states (walls) and absorbing state can be defined. This information can be used to build a custom gridworld MDP. Note that blocked states are removed from the model description using `remove_unreachable_states()`.

Converting Between State Names and Coordinates:

`gw_s2rc()` and `gw_rc2s` help with converting from state names to xy-coordinates and vice versa.

Inspecting Gridworlds:

`gw_matrix()` returns different information (state names, values, actions, etc.) as a matrix. Note that some gridworlds have unreachable states removed. These states will be represented in the matrix as `NA`.

`gw_plot()` plots a gridworld.

`gw_plot_transition_graph()` plots the transition graph using the gridworld matrix as the layout.

`gw_animate()` applies algorithms from `solve_MDP()` iteration by iteration and visualized the state utilities. This helps to understand how the algorithms work.

Gridworld Transition Model:

The transition model is available in several forms:

- `gw_transition_model()` returns a dense vector for the action and start state.
- `gw_transition_model_sparse()` returns a sparse vector for the action and start state. Note: creating sparse vectors is very expensive and should only be used for sparse models with a large state space.
- `gw_transition_model_named()` returns only the non-zero probabilities as a named vector.
- `gw_transition_model_end_state()` returns a single value for a given action, start and end state. Note: Using this function is very slow since it results in excessive function calls.

Mazes:

gw_maze_MDP() helps to easily define maze-like gridworld MDPs. By default, the goal state is absorbing, but with restart = TRUE, the agent restarts the problem at the start state every time it reaches the goal and receives the reward. Note that this implies that the goal state itself becomes unreachable.

gw_read_maze() reads a maze in text format from a file and converts it into a gridworld MDP.

gw_path() checks if a solved gridworld has a policy that leads from the start to the goal. Note this function currently samples only a single path which is an issue with stochastic transitions!

Value

gw_animate() returns the final solution invisibly.

gw_maze_MDP() returns an MDP object.

gw_path() returns a list with the elements "path", "reward" and "solved".

See Also

Other gridworld: Cliff_walking, DynaMaze, Maze, Windy_gridworld

Other visualization: transition_graph()

Examples

```
# Defines states, actions and a transition model for a standard gridworld
gw <- gw_init(
  dim = c(7, 7),
  blocked_states = c("s(2,2)", "s(7,3)", "s(3,6)"),
  absorbing_states = "s(4,4)",
  state_labels = list("s(4,4)" = "Black Hole")
)

str(gw)

# display the state labels in the gridworld (states not represented in the
# model are shown as NA)
gw_matrix(gw)
gw_matrix(gw, what = "label")
gw_matrix(gw, what = "absorbing")
gw_matrix(gw, what = "unreachable") # these are actually missing from the model

# a transition function for regular moves in the gridworld is provided
gw_transition_model(gw, "right", "s(1,1)")
gw_transition_model_end_state(gw, "right", "s(1,1)", "s(1,2)")

# convert between state names and row/column indices
gw_s2rc("s(1,1)")
gw_rc2s(c(1, 1))

# The information in gw can be used to build a custom MDP.
```

```
# We modify the standard transition function so there is a 50% chance that
# you will get sucked into the black hole from the adjacent squares.
trans_black_hole <- function(model,
                             action,
                             start.state,
                             end.state) {
  # states around the black hole
  if (start.state %in% c(
    "s(3,3)", "s(3,4)", "s(3,5)", "s(4,3)", "s(4,5)",
    "s(5,3)", "s(5,4)", "s(5,5)"
  )) {
    if (end.state == "s(4,4)") {
      return(.5 + gw_transition_model_end_state(model, action, start.state,
                                        end.state) * .5)
    } else {
      return(gw_transition_model_end_state(model, action, start.state,
                                        end.state) * .5)
    }
  }

  # use the standard gridworld movement
  gw_transition_model_end_state(model, action, start.state, end.state)
}

black_hole <- MDP(
  states = gw$states,
  actions = gw$actions,
  transition_model = trans_black_hole,
  reward = rbind(R_(                      value = +1),
                 R_(end.state = "s(4,4)", value = -100),
                 R_(start.state = "s(4,4)", value = 0)
                 ),
  info = gw$info,
  name = "Black hole"
)

black_hole
black_hole <- normalize_MDP(black_hole)

gw_plot_transition_graph(black_hole)

# solve the problem
sol <- solve_MDP(black_hole, error = 1)
gw_matrix(sol, what = "values")
gw_plot(sol)
# the optimal policy is to fly around, but avoid the black hole.

# Build a Maze: The Dyna Maze from Chapter 8 in the RL book

DynaMaze <- gw_maze_MDP(
  dim = c(6, 9),
  start = "s(3,1)",
  goal = "s(1,9)",
```

```
  walls = c(
    "s(2,3)", "s(3,3)", "s(4,3)",
    "s(5,6)",
    "s(1,8)", "s(2,8)", "s(3,8)"
  ),
  restart = TRUE,
  discount = 0.95,
  name = "Dyna Maze",
)
DynaMaze

gw_matrix(DynaMaze)
gw_matrix(DynaMaze, what = "labels")

gw_plot_transition_graph(DynaMaze)
# Note that the problems resets if the goal state would be reached.

sol <- solve_MDP(DynaMaze, method = "LP:LP")

gw_matrix(sol, what = "values")
gw_matrix(sol, what = "actions")
gw_plot(sol, states = TRUE)

# check if we found a solution
gw_path(sol)

# Read a maze from a text file
#   (X are walls, S is the start and G is the goal)

# some examples are installed with the package
maze_dir <- system.file("mazes", package = "markovDP")
dir(maze_dir)

file.show(file.path(maze_dir, "small_maze.txt"))

maze <- gw_read_maze(file.path(maze_dir, "small_maze.txt"))
maze
gw_matrix(maze, what = "label")
gw_plot(maze)

# Prioritized sweeping is especially effective for larger mazes.
sol <- solve_MDP(maze, method = "DP:GenPS")
sol

gw_plot(sol)
gw_path(sol, horizon = 1000)

# A maze can also be created directly from a character vector
maze <- gw_read_maze(
    textConnection(c("XXXXXX",
                     "XS  GX",
                     "XXXXXX")))
gw_plot(maze)
```

```
# Create a small random maze
rand_maze <- gw_random_maze(dim = c(5, 5))
gw_plot(rand_maze)
```

`linear_function_approximation` *Linear Function Approximation*

Description

Approximate a Q-function a value function or a policy using linear function approximation.

Usage

```
q_approx_linear(model, transformation = transformation_linear_basis, ...)

v_approx_linear(model, transformation = transformation_linear_basis, ...)

pi_approx_linear(model, transformation = transformation_linear_basis, ...)

approx_value(f, state, action = NULL, w = NULL, model = NULL)
```

Arguments

`model`	a MDP model with defined state features.
`transformation`	a transformation function. See transformation.
`...`	further parameters are passed on to the transformation function.
`f`	a linear approximation object.
`state`	a state or state features.
`action`	an action. If `NULL`, then the value for all available actions will be calculated.
`w`	a weight vector to be used instead of `f$w`.

Details

Linear Approximation:

Approximate Q Values:
The state-action value function is approximated by

$$\hat{q}(s, a) = \boldsymbol{w}^\top \phi(s, a),$$

where $\boldsymbol{w} \in \mathbb{R}^n$ is a weight vector and $\phi : S \times A \to \mathbb{R}^n$ is a feature function that maps each state-action pair to a feature vector. Linear approximation has a single optimum and can be optimized using a simple update rule following the gradient of the state-action function

$$\nabla \hat{q}(s, a, \boldsymbol{w}) = \phi(s, a).$$

Approximate Value Function:
Value function approximation works with state features $\phi(s)$:

$$\hat{v}(s) = \boldsymbol{w}^\top \phi(s)$$

The gradient is

$$\nabla \hat{v}(s, \boldsymbol{w}) = \phi(s).$$

Approximate Policy:
The most popular method implemented here uses a linear preference function

$$h(s, a, \boldsymbol{w}) = \boldsymbol{w}^\top \phi(s, a).$$

The action with the highest preference is the greedy action. To represents a stochastic policy, the preference scores can be converted into probabilities using the softmax function

$$\hat{\pi}(a|s, \boldsymbol{w}) = \frac{e^{h(s,a,\boldsymbol{w})}}{\sum_b e^{h(s,b,\boldsymbol{w})}}$$

The gradient is

$$\nabla \hat{\pi}(a|s, \boldsymbol{w}) = \hat{\pi}(a|s, \boldsymbol{w})[\phi(s, a) - \sum_b \hat{\pi}(b|s, \boldsymbol{w})\phi(s, b)].$$

The last term represents the feature vector reduced by the expected feature vector across all actions under the current policy. This pushes the approximation to make the chosen action a more likely.
We use here the gradient of the log-policy which avoids multiplying by $\hat{\pi}(a|s, \boldsymbol{w})$ and is numerically more stable:

$$\nabla log\ \hat{\pi}(a|s, \boldsymbol{w}) = \phi(s, a) - \sum_b \hat{\pi}(b|s, \boldsymbol{w})\phi(s, b).$$

The gradient of the log-policy points in the same direction as the original gradient.

State-action Feature Vector Construction:

For a small number of actions, we can follow the construction described by Geramifard et al (2013) which uses a state feature function $\phi : S \to \mathbb{R}^m$ to construct the complete state-action feature vector. Here, we also add an intercept term. The state-action feature vector has length $1 + |A| \times m$. It has the intercept and then one component for each action. All these components are set to zero and only the active action component is set to $\phi(s)$, where s is the current state. For example, for the state feature vector $\phi(s) = (3, 4)$ and action $a = 2$ out of three possible actions $A = \{1, 2, 3\}$, the complete state-action feature vector is $\phi(s, a) = (0, 0, 0, 1, 3, 4, 0, 0, 0)$. Each action component has three entries and the 1 represent the intercept for the state feature vector. The zeros represent the components for the two not chosen actions.

The construction of the state-action values is implemented in `add_linear_approx_Q_function()`.

The state feature function $\phi(s)$ starts with raw state feature vector $\mathbf{x} = (x_1, x_2, ..., x_m)$ that are either user-specified or constructed by parsing the state labels of form `s(feature list)`. Then an optional nonlinear transformation can be performed (see transformation).

Internal Representation:

All approximations are lists with the elements:

- `x(s, a)` ... function to construct from state features, action-state features.
- `f(s, a, w)` ... approx. function
- `gradient(s, a, w)` ... gradient of f at w
- `w` ... the weight vector (initially all 0s)
- `transformation` ... a transformation kernel function that is applied to state features in x.

Fitting:

The weight vector are fitted using gradient descent by the MDP solver (e.g., solve_MDP_APPROX).

Prediction:

`approx_value()` calculates approximate value given the weights in the model or a specified weight vector.

References

Alborz Geramifard, Thomas J. Walsh, Stefanie Tellex, Girish Chowdhary, Nicholas Roy, and Jonathan P. How. 2013. A Tutorial on Linear Function Approximators for Dynamic Programming and Reinforcement Learning. Foundations and Trends in Machine Learning 6(4), December 2013, pp. 375-451. doi:10.1561/2200000042

Konidaris, G., Osentoski, S., & Thomas, P. 2011. Value Function Approximation in Reinforcement Learning Using the Fourier Basis. Proceedings of the AAAI Conference on Artificial Intelligence, 25(1), 380-385. doi:10.1609/aaai.v25i1.7903

See Also

Other approximation: `solve_MDP_APPROX()`, `transformation`

Examples

```
data(Maze)

# Approx Q function
f_q <- q_approx_linear(Maze)
f_q

approx_value(f_q, state = "s(3,1)", action = "up", model = Maze)

# update the weights using a learning rate of .1 and a delta of 100
# ([] is used to preserve the names)
f_q$w[] <- .1 * 100 * f_q$gradient(s = s(3,1), a = "up", f_q$w)
f_q

approx_value(f_q, state = "s(3,1)", action = "up", model = Maze)

# find an approximate Q function using a solver
sol <- solve_MDP_APPROX(Maze, horizon = 1000, n = 100)
sol$solution$q_approx_linear
approx_value(sol$solution$q_approx_linear,
             state = "s(3,1)", action = "up", model = Maze)
```

```
# Approx V function
f_v <- v_approx_linear(Maze)
f_v
approx_value(f_v, state = "s(3,1)", model = Maze)

# Approx Policy
f_pi <- pi_approx_linear(Maze)
f_pi

approx_value(f_pi, state = "s(3,1)", model = Maze)

# update the weights using a learning rate of 0.1 and the gradient of
# the log-policy.
f_pi$w <- .1 * f_pi$gradient(s = s(3,1), a = "up", f_pi$w)
approx_value(f_pi, state = "s(3,1)", model = Maze)
```

`Maze` *Steward Russell's 4x3 Maze Gridworld MDP*

Description

The 4x3 maze is described in Chapter 17 of the textbook "Artificial Intelligence: A Modern Approach" (AIMA).

Format

An object of class MDP.

Details

The simple maze has the following layout:

```
   1234           Transition model:
  ######             .8 (action direction)
 1#   +#              ^
 2# # -#              |
 3#S   #          .1 <-|-> .1
  ######
```

We represent the maze states as a gridworld matrix with 3 rows and 4 columns. The states are labeled `s(row, col)` representing the position in the matrix. The # (state `s(2,2)`) in the middle of the maze is an obstruction and not reachable. Rewards are associated with transitions. The default reward (penalty) is -0.04. The start state marked with `S` is `s(3,1)`. Transitioning to + (state `s(1,4)`) gives a reward of +1.0, transitioning to - (state `s_(2,4)`) has a reward of -1.0. Both these states are absorbing (i.e., terminal) states.

Actions are movements (`up`, `right`, `down`, `left`). The actions are unreliable with a .8 chance to move in the correct direction and a 0.1 chance to instead to move in a perpendicular direction leading to a stochastic transition model.

Note that the problem has reachable terminal states which leads to a proper policy (that is guaranteed to reach a terminal state). This means that the solution also converges without discounting (`discount = 1`).

References

Russell,9 S. J. and Norvig, P. (2020). Artificial Intelligence: A modern approach. 4rd ed.

See Also

Other MDP_examples: `Cliff_walking`, `DynaMaze`, `MDP()`, `Windy_gridworld`

Other gridworld: `Cliff_walking`, `DynaMaze`, `Windy_gridworld`, `gridworld`

Examples

```
# The problem can be loaded using data(Maze).

# Here is the complete problem definition.

# We first look at the state layout
gw_matrix(gw_init(dim = c(3, 4)))

# the wall at s(2,2) is unreachable
gw <- gw_init(dim = c(3, 4),
        start = "s(3,1)",
        goal = "s(1,4)",
        absorbing_states = c("s(1,4)", "s(2,4)"),
        blocked_states = "s(2,2)",
        state_labels = list(
            "s(3,1)" = "Start",
            "s(2,4)" = "-1",
            "s(1,4)" = "Goal: +1")
)
gw_matrix(gw)
gw_matrix(gw, what = "index")
gw_matrix(gw, what = "labels")

# gw_init has created the following information
str(gw)

# the transition function is stochastic so we cannot use the standard
# gridworld function provided in gw$transition_model() and we
# have to replace it
P <- function(model, action, start.state) {
  action <- match.arg(action, choices = A(model))

  P <- structure(numeric(length(S(model))), names = S(model))

  # absorbing states
  if (start.state %in% model$info$absorbing_states) {
    P[start.state] <- 1
    return(P)
```

```
  }

  if (action %in% c("up", "down")) {
    error_direction <- c("right", "left")
  } else {
    error_direction <- c("up", "down")
  }

  rc <- gw_s2rc(start.state)
  delta <- list(
    up = c(-1, 0),
    down = c(+1, 0),
    right = c(0, +1),
    left = c(0, -1)
  )

  # there are 3 directions. For blocked directions, stay in place
  # 1) action works .8
  rc_new <- gw_rc2s(rc + delta[[action]])
  if (rc_new %in% S(model))
    P[rc_new] <- .8
  else
    P[start.state] <- .8

  # 2) off to the right .1
  rc_new <- gw_rc2s(rc + delta[[error_direction[1]]])
  if (rc_new %in% S(model))
    P[rc_new] <- .1
  else
    P[start.state] <-  P[start.state] + .1

  # 3) off to the left .1
  rc_new <- gw_rc2s(rc + delta[[error_direction[2]]])
  if (rc_new %in% S(model))
    P[rc_new] <- .1
  else
    P[start.state] <-  P[start.state] + .1

  P
  }

P(gw, "up", "s(3,1)")

R <- rbind(
  R_(                         value = -0.04),
  R_(end.state = "s(2,4)",    value = -1 - 0.04),
  R_(end.state = "s(1,4)",    value = +1 - 0.04),
  R_(start.state = "s(2,4)",  value = 0),
  R_(start.state = "s(1,4)",  value = 0)
)


Maze <- MDP(
```

```
  name = "Stuart Russell's 3x4 Maze",
  discount = 1,
  horizon = Inf,
  states = gw$states,
  actions = gw$actions,
  start = "s(3,1)",
  transition_model = P,
  reward = R,
  info = gw$info
)

Maze

str(Maze)

gw_matrix(Maze)
gw_matrix(Maze, what = "labels")
gw_plot(Maze)

# find absorbing (terminal) states
absorbing_states(Maze)

maze_solved <- solve_MDP(Maze)
policy(maze_solved)

gw_matrix(maze_solved, what = "values")
gw_matrix(maze_solved, what = "actions")

gw_plot(maze_solved)
```

MDP *Define an MDP Problem with Model Access*

Description

Defines all the elements of a discrete-time finite state-space MDP problem.

Usage

```
MDP(
  states,
  actions,
  transition_model,
  reward,
  discount = 0.9,
  horizon = Inf,
  start = "uniform",
  info = NULL,
  name = NA
```

```
)

S(model)

A(model)

is_solved_MDP(model, policy = TRUE, approx = FALSE, stop = FALSE)

is_converged_MDP(model, stop = FALSE)

P_(action = NA, start.state = NA, end.state = NA, probability)

R_(action = NA, start.state = NA, end.state = NA, value)
```

Arguments

states	a character vector specifying the names of the states.
actions	a character vector specifying the names of the available actions.
transition_model	Specifies the transition probabilities between states.
reward	Specifies the rewards, which may depend on the action and state.
discount	numeric; discount rate between 0 and 1.
horizon	numeric; Number of epochs. Inf specifies an infinite horizon.
start	Specifies in which state the MDP starts.
info	A list with additional information.
name	a string to identify the MDP problem.
model	an MDP object.
policy	logical; solution is an explicit policy.
approx	logical; solution is an approximation function.
stop	logical; stop with an error.
action	an action label or integer. The value NA matches any action.
start.state, end.state	state as a state label or an integer. The value NA matches any state.
probability, value	Values used in the helper functions P_() and R_().

Details

Markov decision processes (MDPs) are discrete-time stochastic control processes. This package implements MDPs with a finite state space. MDP() defines all the elements of an MDP problem, including the discount rate, the set of states, the set of actions, the transition model, and the reward model.

We use the following notation. An MDP is a five-tuple:
(S, A, P, R, γ).

S is the set of states; A is the set of actions; P is the conditional transition probability matrix between states; R is the reward function; and γ is the discount factor. We will use lower case letters to represent a member of a set, e.g., s is a specific state. To refer to the size of a set we will use cardinality, e.g., the number of actions is $|A|$.

Names used for mathematical symbols in code:

- S, s, s': `'states'`, `start.state`, `'end.state'`
- A, a: `'actions'`, `'action'`

State names and actions can be specified as strings or index numbers (e.g., `start.state` can be specified as the index of the state in `states`). For the specification as data.frames below, `NA` can be used to mean any `start.state`, `end.state` or `action`.

Specification of transition model: $P(s'|s, a)$:

Transition probability to transition to state s' from given state s and action a. The transition probabilities can be specified in the following ways:

- A data.frame with columns exactly like the arguments of `P_()`. You can use `rbind()` with helper function `P_()` to create this data frame. Probabilities can be specified multiple times and the definition that appears last in the data.frame will take affect.
- A named list of matrices, one for each action. Each matrix is square with rows representing start states s and columns representing end states s'. Instead of a matrix, also the strings `'identity'` or `'uniform'` can be specified.
- A function with the following arguments:
  - A function with the argument list `model`, `action`, `start.state`, `end.state` which returns a single transition probability.
  - A function with the argument list `model`, `action`, `start.state` which returns a transition probability vector for all end states. This vector can be dense, a Matrix::sparseVector or a named vector only containing the non-zero probabilities named by the corresponding end state.

  The arguments `action`, `start.state`, and `end.state` will be always called with the state names as a character vectors of length 1.

Specification of the reward function: $R(a, s, s')$:

The reward function can be specified in the following ways:

- A data frame with columns named exactly like the arguments of `R_()`. You can use `rbind()` with helper function `R_()` to create this data frame. Rewards can be specified multiple times and the definition that appears last in the data.frame will take affect.
- A named list of matrices, one for each action. Each matrix is square with rows representing start states s and columns representing end states s'.
- A function following the same rules as for transition probabilities.

To avoid overflow problems with rewards, reward values should stay well within the range of `[-1e10, +1e10]`. `-Inf` can be used as the reward for unavailable actions and will be translated into a large negative reward for solvers that only support finite reward values.

Specification of the Start State:

The start state of the agent can be a single state or a distribution over the states. The start state definition is used as the default when the return is calculated by `expected_return()` and for sampling with `sample_MDP()`.

Options to specify the start state are:

- A string specifying the name of a single starting state.
- An integer in the range 1 to n to specify the index of a single starting state.
- The string `"uniform"` where the start state is chosen using a uniform distribution over all states.
- A probability distribution over the states. That is, a vector of $|S|$ probabilities, that add up to 1.

By default, the start state is selected uniformly from all states.

Accessing Elements of the MDP:

The convenience functions `S()` and `A()` return the set of states and actions.

See accessors for accessing transition probabilities, rewards, and the start state distribution.

Value

The function returns an object of class MDP which is list with the model specification. `solve_MDP()` reads the object and adds a list element called `'solution'`.

Author(s)

Michael Hahsler

See Also

Other MDP: `absorbing_states()`, `act()`, `action_state_helpers`, `available_actions()`, `find_reachable_states()`, `reachable_states()`, `sample_MDP()`, `sample_MDP.MDPSample()`, `start`, `transition_graph()`, `transition_matrix()`, `unreachable_states()`

Other MDP_examples: `Cliff_walking`, `DynaMaze`, `Maze`, `Windy_gridworld`

Examples

```
# simple MDP example
#
# states:    s1 s2 s3 s4
# transitions: forward moves -> and backward moves <-
# start: s1
# reward: s1, s2, s4 = 0 and s3 = 1

car <- MDP(
  states = c("s1", "s2", "s3", "s4"),
  actions = c("forward", "back", "stop"),
  transition <- list(
    forward = rbind(c(0, 1, 0, 0),
                    c(0, 0, 1, 0),
                    c(0, 0, 0, 1),
                    c(0, 0, 0, 1)),
    back =    rbind(c(1, 0, 0, 0),
                    c(1, 0, 0, 0),
                    c(0, 1, 0, 0),
                    c(0, 0, 1, 0)),
    stop = "identity"
```

```
  ),
  reward = rbind(
    R_(value = 0),
    R_(end.state = "s3", value = 1)
  ),
  discount = 0.9,
  start = "s1",
  name = "Simple Car MDP"
)

car

# internal representation
str(car)

# accessing elements
S(car)
A(car)
start_vector(car, sparse = "states")
transition_matrix(car)
transition_matrix(car, sparse = TRUE)
reward_matrix(car)
reward_matrix(car, sparse = TRUE)

sol <- solve_MDP(car)
sol

policy(sol)
```

`MDPSample` *Define an MDP With Only Sample Access*

Description

Defines a discrete-time Markov decision process with only sample access as a transition function between states using a factored state representation. The state space may be continuous and infinite.

Usage

```
MDPSample(
  actions,
  transition_model,
  start,
  states = NULL,
  absorbing_states = NULL,
  discount = 0.9,
  horizon = Inf,
  info = NULL,
  name = NA
)
```

Arguments

actions	a character vector specifying the names of the available actions.
transition_model	A transition function receiving the current state features and returning the reward and the next state features.
start	Specifies in which state the MDP starts.
states	Optional state labels.
absorbing_states	a single state or a list with absorbing states.
discount	numeric; discount rate between 0 and 1.
horizon	numeric; Number of epochs.
info	A list with additional information.
name	a string to identify the MDP problem.

Details

Defines a Markov decision process as a transition function between state represented as feature vectors (i.e., using a factored state representation). The state space may be

- finite and discrete and specified in the problem,
- finite and discrete but not specified in the problem,
- continuous (implying infinite and not specified in the problem).

Compared to the specification in class MDP, no transition probabilities are available. Instead of the reward function, reward is only returned from calls to the transition function via `act()`. The state space may be specified.

To represent states, a factored state representation as a **row** vector for a single state or a matrix with row vectors for a set of states are used. State labels can be constructed in the form `s(feature1, feature2, ...)`. Conversion between the factored representation and state labels is available in `state2features()` and `features2state()`. Since the state set is not directly represented, **state ids cannot be used!**

Reinforcement learning algorithms with approximation can be used to solve these problems. See: `solve_MDP_APPROX()`.

Value

The function returns an object of class MDPSample which is list with the model specification.

Author(s)

Michael Hahsler

See Also

Other MDPSample: `absorbing_states()`, `act()`, `action_state_helpers`, `reachable_states()`, `sample_MDP.MDPSample()`, `solve_MDP_PG()`, `start`

Examples

```
# Example 1: Define a simple 5x5 maze without walls

transition_model <- function(model, action, state) {
  # make sure action and state are in the correct format
  action <- normalize_action_label(action, model)
  state <- normalize_state_features(state, model)

  if (all(state == s(5, 5)))
    return(list(reward = 0, state_prime = state))

  sp <- state + switch(action,
    up =   c( -1, 0),
    down = c( +1, 0),
    left = c(  0,-1),
    right = c( 0,+1)
  )
  r <- -1

  # check bounds
  if (any(sp < 1) || any(sp > 5)) {
      sp <- state
   }

  # goal
  if (all(sp == s(5, 5)))
    r <- 100

  return(list(reward = r, state_prime = sp))
}

m <- MDPSample(actions = c("up", "right", "down", "left"),
          transition_model,
          start = s(1,1),
          absorbing_states = s(5, 5),
          name = "5x5 Maze")
m

act(m, s(1,1), "down")

# Reach the goal: reward = 100
act(m, s(5,4), "right")

# Illegal action: no movement and reward = -1
act(m, s(1,1), "up")

# Absorbing state: no movement and 0 reward
act(m, s(5,5), "up")


# Example 2: Solve using Linear Feature Approximation
```

```
# Note that we have not specified the state space
m
S(m)

# To use value function approximation, we need to specify the minimum
# and maximum of each state feature for scaling them to [0,1]. The rows
# and column indices used are between 0 and 5.
set.seed(1000)
sol <- solve_MDP_APPROX(m, horizon = 1000, n = 100,
          alpha = 0.01, epsilon = 0.55,
          transformation = transformation_linear_basis,
          min = c(0, 0), max = c(5, 5),
          verbose = FALSE)

# we can calculate the approximate q-values, but have to specify the states.
approx_Q_value(sol, state = rbind(s(1,2), s(4,5)))

# Note: since there is no explicit state space, we also do not have a
#       precalculated policy! We can get approx. greedy actions.

approx_greedy_action(sol, state = s(4,5))

# Example 3: MDPSample with a specified state space

# The same maze as above can be created with this gridworld helper.
# It specifies the finite state space.
m <- gw_maze_MDP(dim = c(5,5), start = "s(1,1)", goal = "s(5,5)", access = "sample")
m
S(m)

# With the defined state space, we do not need to supply min and max
# for the transformation function.
set.seed(1000)
sol <- solve_MDP_APPROX(m, horizon = 1000, n = 100,
          alpha = 0.01, epsilon = 0.55, verbose = FALSE)

# we can use several gridworld helper
gw_plot(sol)
gw_matrix(sol, what = "value")

# and get a policy table
policy(sol)
```

`policy` *Extract, Create Add a Policy to a Model*

Description

Extracts the policy from a solved model or create a policy. All policies are deterministic.

Usage

```
policy(model, epoch = NULL, drop = TRUE)

add_policy(model, policy)

random_policy(
  model,
  prob = NULL,
  estimate_V = FALSE,
  only_available_actions = FALSE,
  ...
)

manual_policy(model, actions, V = NULL, estimate_V = FALSE)

induced_transition_matrix(model, policy = NULL, epoch = 1L, sparse = FALSE)

induced_reward_matrix(model, policy = NULL, epoch = 1L)
```

Arguments

model	A solved MDP object.
epoch	return the policy of the given epoch. NULL returns a list with elements for each epoch.
drop	logical; drop the list for converged, epoch-independent policies.
policy	a policy data.frame.
prob	probability vector for random actions for random_policy(). a logical indicating if action probabilities should be returned for greedy_action().
estimate_V	logical; estimate the value function using policy_evaluation()?
only_available_actions	logical; only sample from available actions? (see available_actions() for details)
...	is passed on to available_actions().
actions	a vector with the action (either the action label or the numeric id) for each state.
V	a vector representing the value function for the policy. If TRUE, then the it is estimated using policy_evaluation().
sparse	logical; should a sparse transition matrix be returned?

Details

policy() extracts the (deterministic) policy from a solved MDP in the form of a a data.frame with columns for:

- state: The state.
- V: The state values if the policy is followed.

- action: The prescribed action.

For unconverged, finite-horizon problems, the solution is a policy for each epoch. This is returned as a list of data.frames.

add_policy() adds a policy to an existing MDP object.

random_policy() and manual_policy() construct new policies.

induced_transition_matrix() returns the single transition matrix which follows the actions specified in a policy.

Value

- policy(), random_policy() and manual_policy() return a data.frame containing the policy. If drop = FALSE then the policy is returned as a list with the policy for each epoch.
- add_policy() returns an MDP object.
- induced_transition_matrix returns a single transition matrix.

The model description with the added policy.

Author(s)

Michael Hahsler

See Also

Other policy: action(), expected_return(), greedy_action(), policy_evaluation(), regret(), visit_probability()

Examples

```
data("Maze")

sol <- solve_MDP(Maze)
sol

## policy with value function and optimal action.
policy(sol)
plot_value_function(sol)
gw_plot(sol)

induced_transition_matrix(sol)

## create a random policy
pi_random <- random_policy(Maze, estimate_V = TRUE)
pi_random

gw_plot(add_policy(Maze, pi_random))

## create a manual policy (go up and in some squares to the right)
acts <- rep("up", times = length(Maze$states))
names(acts) <- Maze$states
acts[c("s(1,1)", "s(1,2)", "s(1,3)")] <- "right"
```

```
acts

pi_manual <- manual_policy(Maze, acts, estimate_V = TRUE)
pi_manual

gw_plot(add_policy(Maze, pi_manual))

# Transition matrix induced by the policy
induced_transition_matrix(Maze, pi_manual, sparse = TRUE)

## Finite horizon (we use incremental pruning because grid does not converge)
sol <- solve_MDP(model = Maze, horizon = 3)
sol

policy(sol)
gw_plot(sol, epoch = 1)
gw_plot(sol, epoch = 2)
gw_plot(sol, epoch = 3)
```

`policy_evaluation` *Policy Evaluation*

Description

Estimate the value function for a policy.

Usage

```
policy_evaluation(
  model,
  pi = NULL,
  method = "bellman",
  ...,
  progress = TRUE,
  verbose = FALSE
)

policy_evaluation_LP(
  model,
  pi = NULL,
  inf = 1000,
  ...,
  progress = FALSE,
  verbose = FALSE
)

policy_evaluation_MC(
  model,
```

```
  pi = NULL,
  n = 1000,
  horizon = NULL,
  first_visit = TRUE,
  ...,
  progress = TRUE,
  verbose = FALSE
)

policy_evaluation_bellman(
  model,
  pi = NULL,
  V = NULL,
  k_backups = 1000L,
  theta = 0.001,
  progress = TRUE,
  verbose = FALSE
)
```

Arguments

model	an MDP problem specification.
pi	a policy as a data.frame with at least columns for states and action. If NULL, then the policy in model is used.
method	used method: "bellman", "LP", or "MC".
...	further parameters are passed on to sample_MDP().
progress	logical; show a progress bar with estimated time for completion.
verbose	logical; should progress and approximation errors be printed.
inf	value used to replace infinity for lpSolve::lp().
n	number of simulated episodes.
horizon	maximum horizon for episodes.
first_visit	if TRUE then only the first visit of a state/action pair in an episode is used to update Q, otherwise, every-visit update is used.
V	a vector with estimated state values representing a value function. If model is a solved model, then the state values are taken from the solution.
k_backups	number of look ahead steps used for approximate policy evaluation used by the policy iteration method. Set k_backups to Inf to only use θ as the stopping criterion.
theta	stop when the largest state Bellman error ($\delta = V_{k+1} - V$) is less than θ.

Details

The LP implementation only works for infinite-horizon problems with a discount factor <1. It solves the following LP:

$$\min \sum_{s \in S} v(s)$$

s.t.

$$v(s) \geq \sum_{s' \in S} p(s'|s, \pi(s))[r(s, \pi(s), s') + \gamma v(s')], \ \forall s \in S$$

Value

a vector with (approximate) state values (U).

Policy Evaluation using Bellman Operator

The value function for a policy can be estimated (called policy evaluation) by repeatedly applying the Bellman operator

$$v \leftarrow B_\pi(v)$$

till convergence.

In each iteration, all state values are updated. In this implementation updating is stopped when the largest state Bellman error is below a threshold.

$$||v_{k+1} - v_k||_\infty < \theta.$$

Or if `k_backups` iterations have been completed.

Author(s)

Michael Hahsler

References

Sutton, R. S., Barto, A. G. (2020). Reinforcement Learning: An Introduction. Second edition. The MIT Press.

See Also

Other policy: `action()`, `expected_return()`, `greedy_action()`, `policy()`, `regret()`, `visit_probability()`

Examples

```
data(Maze)
Maze

# create several policies:
# 1. optimal policy using value iteration
maze_solved <- solve_MDP(Maze, method = "DP:VI")
pi_opt <- policy(maze_solved)
pi_opt

# 2. a manual policy (go up and in some squares to the right)
acts <- rep("up", times = length(Maze$states))
```

```
names(acts) <- Maze$states
acts[c("s(1,1)", "s(1,2)", "s(1,3)")] <- "right"
pi_manual <- manual_policy(Maze, acts)
pi_manual

# 3. a random policy
set.seed(1234)
pi_random <- random_policy(Maze, prob = c(up = .7, right = .1, down = .1, left = 0.1))
pi_random

# 4. an improved policy based on one policy evaluation and
#   policy improvement step.
V <- policy_evaluation(Maze, pi_random)
Q <- Q_values(Maze, V)
pi_greedy <- greedy_policy(Q)
pi_greedy

# compare the approx. value functions for the policies (we restrict
#    the number of backups for the random policy since it may not converge)
rbind(
  random = policy_evaluation(Maze, pi_random, k_backups = 100),
  manual = policy_evaluation(Maze, pi_manual),
  greedy = policy_evaluation(Maze, pi_greedy),
  optimal = policy_evaluation(Maze, pi_opt)
)

# use fist-visit Monte Carlo prediction with 100 episodes
#   and a max horizon of 100
policy_evaluation(Maze, pi_opt, method = "MC", n = 100, horizon = 100)
```

`Q_values` *Q-Values*

Description

Several useful functions to deal with Q-values (action values) which map each state/action pair to a utility value.

Usage

```
Q_values(model, V = NULL, state = NULL)

Q_zero(model, value = 0)

Q_random(model, min = 1e-06, max = 1)
```

Arguments

`model`	an MDP problem specification.

V	the state values. If model is a solved model, then the state values are taken from the solution.
state	specify the state. If NULL then the Q-values for all states is returned as a matrix.
value	value to initialize the Q-value matrix. Default is 0.
min, max	range of the random values

Details

Implemented functions are:

- Q_values() gets the Q-values from the model. If the value function V is specified or if the policy of a solved model contains a value function, then the Q-values are approximated using the Bellman optimality equation:

$$q_*(s,a) = \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v_*(s')]$$

  Exact Q values are calculated if $v = v_*$, the optimal value function, otherwise we get an approximation that might not be consistent with v or the implied policy. Q values can be used as the input for several other functions.

  For solvers, that directly calculate Q-values or an approximate Q-function, then the solver's values are returned.

- Q_zero() and Q_random() create initial Q value matrices for algorithms.

Value

Q_values() returns a state by action matrix specifying the Q-function, i.e., the action value for executing each action in each state. The Q-values are calculated from the value function (U) and the transition model.

Q_zero() and Q_random return a matrix with q-values.

Author(s)

Michael Hahsler

References

Sutton, R. S., Barto, A. G. (2020). Reinforcement Learning: An Introduction. Second edition. The MIT Press.

See Also

Other value_function: bellman_update(), value_function()

Examples

```
data(Maze)
Maze

# create a random policy and calculate q-values
pi_random <- random_policy(Maze)
pi_random

V <- policy_evaluation(Maze, pi_random)
V

# calculate Q values
Q <- Q_values(Maze, V)
Q
```

`reachable_states` *Find Reachable States*

Description

Finds the reachable state space from an MDP or MDPSample.

Usage

```
reachable_states(model, ..., progress = TRUE)

## S3 method for class 'MDPModel'
reachable_states(model, horizon = Inf, ..., progress = TRUE)

## S3 method for class 'MDPSample'
reachable_states(model, n = 100, horizon = NULL, ..., progress = TRUE)

## S3 method for class '`function`'
reachable_states(
  model,
  actions,
  start_state,
  horizon = Inf,
  ...,
  progress = TRUE
)
```

Arguments

`model`	an MDP or an MDP transition function.
`...`	further arguments are passed on (e.g., to `sample_MDP()`)
`progress`	logical; show a progress bar?

horizon	only return states reachable in the given horizon.
n	number if sampled trajectories.
actions	labels of the available actions.
start_state	label of the start state.

Details

There are three application cases for finding reachable states.

- For an MDP that has a state space defined, not all specified states might be reachable. The function performs a (depth-limited) depth-first traversal of the state space and returns a vector with the names of all encountered states. This is used for example for unreachable_states().
- We may only have a transition model function following the specifications of an R transition function for an MDPModel which returns a probability distribution over state transitions. To create a complete MDP, we can use the found reachable states to create a complete MDP object. This search also used depth-first search of the state space.
- To use tabular methods for MDPSample which specify a transition function that returns the reward and the next state, we need to also specify the state space. Since an MDPSample can have a stochastic transition model, trajectory sampling is used. The horizon and the number of trajectories n has to be specified.

  Notes:

  - Not all reachable states may be returned if some states have a very low probability to be in the sample trajectories.
  - A finite horizon is needed for sampling. By default the model horizon is used. Infinite horizon is capped at 1000. Larger values can be specified manually.

Value

a character vector with all reachable states.

Author(s)

Michael Hahsler

See Also

Other MDP: MDP(), absorbing_states(), act(), action_state_helpers, available_actions(), find_reachable_states(), sample_MDP(), sample_MDP.MDPSample(), start, transition_graph(), transition_matrix(), unreachable_states()

Other MDPSample: MDPSample(), absorbing_states(), act(), action_state_helpers, sample_MDP.MDPSample(), solve_MDP_PG(), start

Examples

```
# Example 1: Find the reachable states of a simple MDP with only sample access

line_maze <- MDPSample(actions = c("left", "right"),
      start = s(0),
```

```
      absorbing_states = rbind(s(-10), s(10)),
      transition_model = function(model, action, state) {
            if (state == s(-10) || state == s(+10)) {
              return(list(reward = 0, state_prime = state))
            }

            reward <- 0
            if (action == "left") state <- state - 1
            if (action == "right") state <- state + 1

            if (state == s(-10) || state == s(+10)) {
                reward <- 10
            }

            return(list(reward = reward, state_prime = state))
      },
      name = "line maze [-10,+10]"
      )

# this model has no state specified
line_maze
S(line_maze)

# find the states
states <- reachable_states(line_maze, horizon = 100)
states

# set the states in the model
line_maze$states <- states
line_maze

sol <- solve_MDP(line_maze, method = "TD:q_learning",
  horizon = 100, n = 100, epsilon = .8)

policy(sol)
plot_value_function(sol)

# Example 2: Find the states to define an MDP for Tic-Tac-Toe

# state description: matrix with the characters _, x, and o
#                    can be converted into a label of 9 characters

# set of actions
A <- as.character(1:9)

# helper functions
ttt_empty_board <- function() matrix('_', ncol = 3, nrow = 3)

ttt_state2label <- function(state) paste(state, collapse = '')

ttt_label2state <- function(label) matrix(strsplit(label, "")[[1]],
                                          nrow = 3, ncol = 3)
```

```
ttt_available_actions <- function(state) {
  if (length(state) == 1L) state <- ttt_label2state(state)
  which(state == "_")
}

ttt_result <- function(state, player, action) {
  if (length(state) == 1L) state <- ttt_label2state(state)

  if (state[action] != "_")
    stop("Illegal action.")

  state[action] <- player
  state
}

ttt_terminal <- function(state) {
  if (length(state) == 1L) state <- ttt_label2state(state)

  # Check the board for a win and return one of
  # 'x', 'o', 'd' (draw), or 'n' (for next move)
  win_possibilities <- rbind(state,
                             t(state),
                             diag(state),
                             diag(t(state)))

  wins <- apply(win_possibilities, MARGIN = 1, FUN = function(x) {
    if (x[1] != '_' && length(unique(x)) == 1) x[1]
    else '_'
  })

  if (any(wins == 'x'))
    return('x')

  if (any(wins == 'o'))
    return('o')

  # Check for draw
  if (sum(state == '_') < 1)
    return('d')

  return('n')
}

# define the transition function:
#     * return a probability vector for an action in a start state
#     * we define the special states 'win', 'loss', and 'draw'
P <- function(model, action, start.state) {
  action <- as.integer(action)

  # absorbing states
  if (start.state %in% c('win', 'loss', 'draw', 'illegal')) {
    return(structure(1, names = start.state))
  }
```

```
  # avoid illegal action by going to the very expensive illegal state
  if (!(action %in% ttt_available_actions(start.state))) {
    return(structure(1, names = "illegal"))
  }

  # make x's move
  next_state <- ttt_result(start.state, 'x', action)

  # terminal?
  term <- ttt_terminal(next_state)
  if (term == 'x') {
    return(structure(1, names = "win"))
  } else if (term == 'o') {
    return(structure(1, names = "loss"))
  } else if (term == 'd') {
    return(structure(1, names = "draw"))
  }

  # it is o's turn
  actions_of_o <- ttt_available_actions(next_state)
  possible_end_states <- lapply(
    actions_of_o,
    FUN = function(a)
      ttt_result(next_state, 'o', a)
  )

  # fix terminal states
  term <- sapply(possible_end_states, ttt_terminal)
  possible_end_states <- sapply(possible_end_states, ttt_state2label)
  possible_end_states[term == 'x'] <- 'win'
  possible_end_states[term == 'o'] <- 'loss'
  possible_end_states[term == 'd'] <- 'draw'

  possible_end_states <- unique(possible_end_states)

  return(structure(rep(1 / length(possible_end_states),
                       length(possible_end_states)),
                   names = possible_end_states))
}

# define the reward
R <- rbind(
  R_(                    value = 0),
  R_(end.state = 'win',  value = +1),
  R_(end.state = 'loss', value = -1),
  R_(end.state = 'draw', value = +.5),
  R_(end.state = 'illegal', value = -Inf),
  # Note: there is no more reward once the agent is in a terminal state
  R_(start.state = 'win',  value = 0),
  R_(start.state = 'loss', value = 0),
  R_(start.state = 'draw', value = 0),
  R_(start.state = 'illegal', value = 0)
```

```
)

# start state
start <- ttt_state2label(ttt_empty_board())
start

# find the reachable state space
S <- union(c('win', 'loss', 'draw', 'illegal'),
          reachable_states(P, start_state = start, actions = A))
head(S)

tictactoe <- MDP(S, A, P, R, discount = 1, start = start, name = "TicTacToe")
tictactoe

# this MDP takes about 30 seconds to solve using value iteration
# sol <- solve_MDP(tictactoe)
# policy(sol)[1:10, ]
```

---

regret *Regret of a Policy and Related Measures*

Description

Calculates the regret and related measures for a policy relative to a benchmark policy.

Usage

```
regret(policy, benchmark, start = NULL, relative = FALSE, ...)

action_discrepancy(
  policy,
  benchmark,
  weighted = FALSE,
  proportion = FALSE,
  states = FALSE
)

value_error(policy, benchmark, type = "RMSVE", weighted = FALSE)
```

Arguments

policy	a solved MDP containing the policy to calculate the regret for.
benchmark	a solved MDP with the (optimal) policy. Regret is calculated relative to this policy.
start	start state distribution. If NULL then the start state of the benchmark is used.
relative	logical; should the relative regret (regret divided by the reward of the benchmark) be calculated?

`...`	further arguments are passed on to `expected_return()`.
`weighted`	logical; should mismatched actions or state value errors be weighted by the state visit probability for the benchmark? Rarely or never visited states will have now less influence on the measure.
`proportion`	logical; should the action discrepancy be reported as a proportion of states with a different action.
`states`	logical; return the mismatching state ids.
`type`	type of error root mean square value error (`"RMSVE"`), mean square value error (`"MSVE"`), mean absolute value error (`"MAVE"`), absolute value error vector (`"AVE"`), value error vector (`"VE"`).

Details

Regret:

Regret for a policy π is defined as

$$v_\pi(s_0) - v_*(s_0),$$

where $v_\pi(s_0)$ represents the expected long-term state value for following policy π and the starting in state s_0 (or a start distribution). The relative regret is calculated as

$$\frac{v_\pi(s_0) - v_*(s_0)}{v_*(s_0)}.$$

Note that for regret, usually the optimal policy π^* is used as the benchmark. Since the optimal policy may not be known, regret relative to the best known policy can be used.

Regret is only valid with converged value functions. This means that either the solver has converged, or the value function was estimated for the policy using converged `policy_evaluation()`.

Action Discrepancy:

The action discrepancy measures the difference between two policies as the number of states for which the prescribed action in the policies differs. Often, a policy is compared to the best known policy called the benchmark policy.

Some times two actions are equivalent (have the same q-value) and the algorithm breaks the tie randomly. The implementation accounts for this case.

The action discrepancy can be calculated as a proportion of different actions or be weighted by the state visit probability given the benchmark policy. Both weighted and proportional action discrepancy is scaled in $[0, 1]$.

Root Mean Squared Value Error:

The root mean value error

$$\sqrt{\text{VE}} = \sqrt{||v_\pi - v_*||^2}$$

is the sum of the squared differences of state values between a solution's value function and the optimal value function. For v_*, the value function of the benchmark solution is used. Related measures like MSVE (means squared value error), MAVE (means absolute value error), AVE (absolute value error) and VE (value error) are also provided.

The error can also be weighted by the state visit probability given the benchmark policy. This results in the expected error with respect to the state visit distribution of the benchmark policy. This may be important to evaluate methods that focus only on estimating the value function for states that are actually visited using the policy.

Value

- `regret()` returns the regret as a difference of expected long-term rewards.
- `action_discrepancy()` returns the number or proportion of diverging actions.
- `mean_value_error()` returns the mean squared or absolute difference in the value function.

Author(s)

Michael Hahsler

See Also

Other policy: `action()`, `expected_return()`, `greedy_action()`, `policy()`, `policy_evaluation()`, `visit_probability()`

Examples

```
data(Maze)

sol_optimal <- solve_MDP(Maze)

# a manual policy (go up and in some squares to the right)
acts <- rep("up", times = length(Maze$states))
names(acts) <- Maze$states
acts[c("s(1,1)", "s(1,2)", "s(1,3)")] <- "right"

sol_manual <- add_policy(Maze, manual_policy(Maze, acts, estimate_V = TRUE))

# compare the policies side-by-side
cbind(opt = policy(sol_optimal), manual = policy(sol_manual))

# the regret is very small. It is about 4.8% of the optimal reward
regret(sol_manual, benchmark = sol_optimal)
regret(sol_manual, benchmark = sol_optimal, relative = TRUE)

# The number of different actions (excluding equivalent actions) is 3.
# This about 27% of the actions in the policy.
action_discrepancy(sol_manual, benchmark = sol_optimal)
action_discrepancy(sol_manual, benchmark = sol_optimal, proportion = TRUE)

# Weighted by the probability that a state will be visited shows that
# only 2.3% of the time a different action would be used.
action_discrepancy(sol_manual, benchmark = sol_optimal, weighted = TRUE)

value_error(sol_manual, benchmark = sol_optimal, type = "VE")
value_error(sol_manual, benchmark = sol_optimal, type = "MAVE")
value_error(sol_manual, benchmark = sol_optimal, type = "RMSVE")

# Weighting shows that the expected MAVE (expectation taken over the
# benchmark policy) is rather small.
value_error(sol_manual, benchmark = sol_optimal, type = "MAVE", weighted = TRUE)
```

round_stochastic *Round a stochastic vector or a row-stochastic matrix*

Description

Rounds a vector such that the sum of 1 is preserved. Rounds a matrix such that each row sum up to 1. One entry is adjusted after rounding such that the rounding error is the smallest.

Usage

```
round_stochastic(x, digits = 7)
```

Arguments

x	a stochastic vector or a row-stochastic matrix.
digits	number of digits for rounding.

Value

The rounded vector or matrix.

See Also

round

Other utilities: colors

Examples

```
# regular rounding would not sum up to 1
x <- c(0.333, 0.334, 0.333)

round_stochastic(x)
round_stochastic(x, digits = 2)
round_stochastic(x, digits = 1)
round_stochastic(x, digits = 0)


# round a stochastic matrix
m <- matrix(runif(15), ncol = 3)
m <- sweep(m, 1, rowSums(m), "/")

m
round_stochastic(m, digits = 2)
round_stochastic(m, digits = 1)
round_stochastic(m, digits = 0)
```

`sample_MDP` *Sample Trajectories from an MDP*

Description

Sample trajectories from an MDP. The start state for each trajectory is chosen using the start definition in the model. Actions are chosen randomly or using an epsilon-greedy policy.

Usage

```
sample_MDP(model, n, ...)

## S3 method for class 'MDP'
sample_MDP(
  model,
  n,
  start = NULL,
  horizon = NULL,
  epsilon = NULL,
  exploring_starts = FALSE,
  delta_horizon = 0.001,
  trajectories = FALSE,
  engine = NULL,
  progress = TRUE,
  verbose = FALSE,
  ...
)
```

Arguments

`model`	an MDP model.
`n`	number of trajectories.
`...`	further arguments are ignored.
`start`	probability distribution over the states for choosing the starting states for the trajectories. Defaults to "uniform".
`horizon`	epochs end once an absorbing state is reached or after the maximal number of epochs specified via `horizon`. If `NULL` then the horizon for the model is used.
`epsilon`	the probability of choosing a random action when using an epsilon-greedy policy. Default for solved models is 0 and for unsolved model 1.
`exploring_starts`	logical; randomly sample a start/action combination to start the episode from.
`delta_horizon`	precision used to determine the horizon for infinite-horizon problems.
`trajectories`	logical; return the complete trajectories.

engine	'cpp' or 'r' to perform simulation using a faster C++ or a native R implementation NULL uses the C++ implementation unless the transition model or the reward are specified as R functions (which are slow in C++).
progress	show a progress bar?
verbose	report used parameters

Details

The default is a faster C++ implementation (engine = 'cpp'). A native R implementation is available (engine = 'r').

Both implementations support parallel execution using the package **foreach**. To enable parallel execution, a parallel backend such as **doParallel** needs to be registered (see doParallel::registerDoParallel()). Note that small samples are slower using parallelization. Therefore, C++ simulations with n * horizon less than 100,000 are always executed using a single worker.

Value

A list with elements:

- avg_return: The average discounted return.
- reward: Reward for each trajectory.
- action_cnt: Action counts.
- state_cnt: State counts.
- trajectories: A data.frame with the trajectories. Each row contains the episode id, the time step, the state s, the chosen action a, the reward r, and the next state s_prime. Trajectories are only returned for trajectories = TRUE.

Author(s)

Michael Hahsler

See Also

Other MDP: MDP(), absorbing_states(), act(), action_state_helpers, available_actions(), find_reachable_states(), reachable_states(), sample_MDP.MDPSample(), start, transition_graph(), transition_matrix(), unreachable_states()

Examples

```
# enable parallel simulation
# doParallel::registerDoParallel()

data(Maze)

# solve the MDP for 5 epochs and no discounting
sol <- solve_MDP(Maze, discount = 1)
sol

# V in the policy is and estimate of the state values when following the optimal policy.
```

```
policy(sol)
gw_matrix(sol, what = "action")

## Example 1: simulate 100 trajectories following the policy,
#             only the final belief state is returned
sim <- sample_MDP(sol, n = 100, horizon = 10, verbose = TRUE)
sim

# Note that all simulations for this model start at s_1 and that the simulated avg. return
# is therefore an estimate to the value function for the start state s_1.
policy(sol)[1, ]

# Calculate proportion of actions taken in the simulation
round_stochastic(sim$action_cnt / sum(sim$action_cnt), 2)

# reward distribution
hist(sim$reward)

## Example 2: simulate trajectories starting from a uniform distribution over all
#             states and return all trajectories
sim <- sample_MDP(sol,
  n = 100, start = "uniform", horizon = 10,
  trajectories = TRUE
)
head(sim$trajectories)

# how often was each state visited?
table(sim$trajectories$s)
```

`sample_MDP.MDPSample` *Sample Trajectories from an MDPSample*

Description

Sample trajectories using an MDPSample.

Usage

```
## S3 method for class 'MDPSample'
sample_MDP(
  model,
  n,
  start = NULL,
  horizon = NULL,
  epsilon = NULL,
  exploring_starts = FALSE,
  trajectories = FALSE,
  progress = TRUE,
  verbose = FALSE,
```

```
  ...
)
```

Arguments

model	an MDPSample model.
n	number of trajectories.
start	start state.
horizon	epochs end once an absorbing state is reached or after the maximal number of epochs specified via horizon. If NULL then the horizon for the model is used.
epsilon	the probability of random actions for using an epsilon-greedy policy. Default for solved models is 0 and for unsolved model 1.
exploring_starts	logical; randomly sample a start/action combination to start the episode from.
trajectories	logical; return the complete trajectories.
progress	show a progress bar?
verbose	report used parameters
...	further arguments are ignored.

Value

A list with elements:

- avg_return: The average discounted return.
- reward: Reward for each trajectory.
- trajectories: A data.frame with the trajectories. Each row contains the episode id, the time step, the state s, the chosen action a, the reward r, and the next state s_prime. Trajectories are only returned for trajectories = TRUE.

Author(s)

Michael Hahsler

See Also

Other MDP: MDP(), absorbing_states(), act(), action_state_helpers, available_actions(), find_reachable_states(), reachable_states(), sample_MDP(), start, transition_graph(), transition_matrix(), unreachable_states()

Other MDPSample: MDPSample(), absorbing_states(), act(), action_state_helpers, reachable_states(), solve_MDP_PG(), start

Examples

```
# enable parallel simulation (useful for sampling with large n)
# doParallel::registerDoParallel()

# Create a simple maze with the layout:
# XXXXXX
# XSX  X
# X    X
# X    X
# X  XGX
# XXXXXX

model <- gw_maze_MDP(
           dim = s(4, 4),
           start = s(1, 1),
           goal = s(4, 4),
           walls = rbind(s(1, 2), s(4, 3)),
           discount = 0.95,
           access = "sample",
           name = "Simple Maze"
       )
model
gw_plot(model)

# sample a random walk (epsilon = 1) from the unsolved MDP
set.seed(1234)
sim <- sample_MDP(model, horizon = 500, n = 1,
                  verbose = TRUE, trajectories = TRUE)
sim

# sample from a solved MDPSample by following the policy
set.seed(1234)
sol <- solve_MDP_APPROX(model, horizon = 500, n = 100,
                  transformation = transformation_fourier_basis, order = 2)
gw_plot(sol)

sim <- sample_MDP(sol, horizon = 500, n = 1,
                  verbose = TRUE, trajectories = TRUE)
sim
```

---

schedule *Schedules to Reduce Alpha, Epsilon and Other Parameters*

Description

Several schedule functions to reduce learning parameters are available using generator functions.

Usage

```
schedule_exp(start, decay)

schedule_exp2(start, basis)

schedule_log(start)

schedule_linear(start, end, n)

schedule_harmonic(start, n)
```

Arguments

start	start value for the schedule.
decay	decay factor for exponential schedules.
basis	basis for exponential schedules.
end	end value for the schedule.
n	number of steps (e.g., epochs) for the schedule.

Details

Several learning parameters need to be reduced during learning to ensure convergence. We provide several schedule function generators that reduce learning parameters after each processed episode or according to the number of times a state-action combination was tried.

Here are the definitions of the available schedules:

- schedule_exp : `function(t) start * exp(-decay * (t - 1))`
- schedule_exp2 : `function(t) start * basis^(t - 1)`
- schedule_log : `function(t) pmin(start/log(1 + t), start)`
- schedule_linear : `step <- (start - end)/n function(t) start - t * step`
- schedule_harmonic : `function(t) pmin(start/t, 1)`

`t` is the time step, episode number or count starting with 1.

See Also

Other solver: `convergence_horizon()`, `solve_MDP()`, `solve_MDP_APPROX()`, `solve_MDP_DP()`, `solve_MDP_LP()`, `solve_MDP_MC()`, `solve_MDP_PG()`, `solve_MDP_SAMP()`, `solve_MDP_TD()`

Examples

```
# create an exponential  schedule function
s_exp <- schedule_exp(1, decay = .1)
s_exp

# plot the schedule for 100 episodes.
episode <- seq_len(100)
```

```
plot(x = episode, s_exp(1:100), type = "l")

# compare some schedule examples
schedules <- cbind(
  `exp decay = 0.1` = schedule_exp(1, decay = .1)(1:100),
  `exp decay = 0.01` = schedule_exp(1, decay = .01)(1:100),
  `exp basis = 0.9` = schedule_exp2(1, basis = .9)(1:100),
  `linear` = schedule_linear(1, end = 0, n = 100)(1:100),
  `log` = schedule_log(1)(1:100),
  `harmonic` = schedule_harmonic(1, n = 100)(1:100),
  `harmonic start = 10` = schedule_harmonic(10, n = 100)(1:100)
  )

matplot(schedules, type = "l",
        col = 1:ncol(schedules),
        lty = 1:ncol(schedules)
        )
legend("topright",
        legend = colnames(schedules),
        col = 1:ncol(schedules),
        lty = 1:ncol(schedules),
        cex = 0.8
        )
```

---

solve_MDP *Solve an MDP Problem*

Description

Implementation of value iteration, modified policy iteration and other methods based on reinforcement learning techniques to solve finite state space MDPs.

Usage

```
solve_MDP(model, ...)

## S3 method for class 'MDP'
solve_MDP(
  model,
  method = "DP:VI",
  horizon = NULL,
  discount = NULL,
  ...,
  matrix = TRUE,
  continue = FALSE,
  verbose = FALSE,
  progress = !verbose
)
```

```
## S3 method for class 'MDPSample'
solve_MDP(
  model,
  method = "APPROX:semi_gradient_sarsa",
  horizon = NULL,
  discount = NULL,
  ...,
  matrix = TRUE,
  continue = FALSE,
  verbose = FALSE,
  progress = !verbose
)
```

Arguments

model	an MDP problem specification.
...	further parameters are passed on to the solver function.
method	string; Composed of the algorithm family abbreviation and the algorithm separated by :. The algorithm families can be found in the See Also section under "Other solvers". The family abbreviation follows solve_MDP_ in the function name.
horizon	an integer with the number of epochs for problems with a finite planning horizon. If set to Inf, the algorithm continues running iterations till it converges to the infinite horizon solution. If NULL, then the horizon specified in model will be used.
discount	discount factor in range $(0, 1]$. If NULL, then the discount factor specified in model will be used.
matrix	logical; if TRUE then matrices for the transition model and the reward function are taken from the model first. This can be slow if functions need to be converted or do not fit into memory if the models are large. If these components are already matrices, then this is very fast. For FALSE, the transition probabilities and the reward is extracted when needed. This is slower, but removes the time and memory requirements needed to calculate the matrices.
continue	logical; show a progress bar with estimated time for completion.
verbose	logical or a numeric verbose level; if set to TRUE or 1, the function displays the used algorithm parameters and progress information. Levels >1 provide more detailed solver output in the R console.
progress	logical; show a progress bar with estimated time for completion.

Details

Several solvers are available. Note that some solvers are only implemented for finite-horizon problems.

Most solvers can be interrupted using Esc/CTRL-C and will return the current solution. Solving can be continued by calling solve_MDP with the partial solution as the model and the parameter continue = TRUE. This method can also be used to reduce parameters like alpha or epsilon (see Q-learning in the Examples section).

A list of available solvers can be found in the See Also section under "Other solvers".

While MDP model contain an explicit specification of the state space, the transition probabilities and the reward structure, MDPSample only contains a transition function. This means that only a small subset of solvers can be used for MDPSamples. This currently includes only includes the solvers in `solve_MDP_APPROX()`.

Value

`solve_MDP()` returns an object of class MDP or MDPSample which is a list with the model specifications (`model`), the solution (`solution`). The solution is a list with the elements that depend on the used method. Common elements are:

- `method` with the name of the used method
- parameters used.
- `converged` did the algorithm converge (`NA`) for finite-horizon problems.
- `policy` a list representing the policy graph. The list only has one element for converged solutions.

Author(s)

Michael Hahsler

References

Russell, Stuart J., and Peter Norvig. 2020. Artificial Intelligence: A Modern Approach (4th Edition). Pearson. http://aima.cs.berkeley.edu/.

Sutton, Richard S., and Andrew G. Barto. 2018. Reinforcement Learning: An Introduction. Second. The MIT Press. http://incompleteideas.net/book/the-book-2nd.html.

See Also

Other solver: `convergence_horizon()`, `schedule`, `solve_MDP_APPROX()`, `solve_MDP_DP()`, `solve_MDP_LP()`, `solve_MDP_MC()`, `solve_MDP_PG()`, `solve_MDP_SAMP()`, `solve_MDP_TD()`

Examples

```
data(Maze)
Maze

# default is value iteration (VI)
maze_solved <- solve_MDP(Maze)
maze_solved
policy(maze_solved)

# plot the value function U
plot_value_function(maze_solved)

# Gridworld solutions can be visualized
gw_plot(maze_solved)
```

solve_MDP_APPROX *Solve MDPs with Temporal Differencing with Function Approximation*

Description

Solve the MDP control problem using state-value approximation by semi-gradient Sarsa (temporal differencing) for episodic problems.

Usage

```
solve_MDP_APPROX(
  model,
  method = "sarsa",
  horizon = NULL,
  discount = NULL,
  alpha = schedule_exp(0.2, 0.1),
  epsilon = schedule_exp(1, 0.1),
  lambda = 0,
  n,
  state_features = NULL,
  transformation = transformation_linear_basis,
  w = NULL,
  ...,
  matrix = TRUE,
  continue = FALSE,
  progress = TRUE,
  verbose = FALSE
)

approx_Q_value(model, state = NULL, action = NULL, w = NULL)

approx_greedy_action(model, state, w = NULL, epsilon = 0, as = "factor")

approx_greedy_policy(model, w = NULL)

approx_V_plot(
  model,
  min = NULL,
  max = NULL,
  w = NULL,
  res = 25,
  col = hcl.colors(res, "YlOrRd", rev = TRUE),
  image = TRUE,
  contour = TRUE,
  main = NULL,
  ...
)
```

Arguments

model	an MDP problem specification.
method	string; one of the following solution methods: 'sarsa'
horizon	an integer with the number of epochs for problems with a finite planning horizon. If set to Inf, the algorithm continues running iterations till it converges to the infinite horizon solution. If NULL, then the horizon specified in model will be used.
discount	discount factor in range $(0, 1]$. If NULL, then the discount factor specified in model will be used.
alpha	step size (learning rate). A scalar value between 0 and 1 or a schedule.
epsilon	used for the ϵ-greedy behavior policies. A scalar value between 0 and 1 or a schedule.
lambda	the trace-decay parameter for the an accumulating trace. If lambda = 0 then 1-step Sarsa is used.
n	number of episodes used for learning.
state_features	a matrix with one row per state with state features to be used. If NULL then get_state_features() will be used to get the state features stored in the model, or to construct state features from state labels.
transformation	a transformation function. See transformation.
w	a weight vector
...	further parameters are passed on to the transformation function.
matrix	logical; if TRUE then matrices for the transition model and the reward function are taken from the model first. This can be slow if functions need to be converted or do not fit into memory if the models are large. If these components are already matrices, then this is very fast. For FALSE, the transition probabilities and the reward is extracted when needed. This is slower, but removes the time and memory requirements needed to calculate the matrices.
continue	logical; show a progress bar with estimated time for completion.
progress	logical; show a progress bar with estimated time for completion.
verbose	logical or a numeric verbose level; if set to TRUE or 1, the function displays the used algorithm parameters and progress information. Levels >1 provide more detailed solver output in the R console.
state	a state (index or name)
action	an action (index or name)
as	character; specifies the desired output format (see normalize_action())
min, max	numeric vectors with minimum/maximum values for each feature in the state feature representation.
res	resolution as the number of values sampled from each feature.
col	colors passed on to graphics::image().
image, contour	logical; include the false color image or the contours in the plot?
main	title for the plot.

Details

Episodic Semi-gradient Sarsa:

The implementation follows the temporal difference algorithm episodic Semi-gradient Sarsa algorithm given in Sutton and Barto (2018).

Schedules:

- epsilon schedule: `t` is increased by each processed episode.
- alpha schedule: `t` is increased by each processed episode.

Value

`solve_MDP()` returns an object of class MDP or MDPSample which is a list with the model specifications (`model`), the solution (`solution`). The solution is a list with the elements that depend on the used method. Common elements are:

- `method` with the name of the used method
- parameters used.
- `converged` did the algorithm converge (`NA`) for finite-horizon problems.
- `policy` a list representing the policy graph. The list only has one element for converged solutions.

References

Sutton, Richard S., and Andrew G. Barto. 2018. Reinforcement Learning: An Introduction. Second. The MIT Press. http://incompleteideas.net/book/the-book-2nd.html.

See Also

Other solver: `convergence_horizon()`, `schedule`, `solve_MDP()`, `solve_MDP_DP()`, `solve_MDP_LP()`, `solve_MDP_MC()`, `solve_MDP_PG()`, `solve_MDP_SAMP()`, `solve_MDP_TD()`

Other approximation: `linear_function_approximation`, `transformation`

Examples

```
# Example 1: A maze without walls. The step cost is 1. The start is top-left and
# the goal (+100 reward) is bottom-right.
# This is the ideal problem for a linear approximation of the Q-function
# using the x/y location as state features.

m <- gw_maze_MDP(c(5, 5), start = "s(1,1)", goal = "s(5,5)")

# gridworlds have state labels of the format "s(row, col)" which can be
# automatically converted into state features used for approximation.
S(m)
get_state_features(m)

# solve using linear state features (no transformation)
set.seed(1000)
sol <- solve_MDP_APPROX(m, horizon = 1000, n = 100)
```

```
# approximation
sol$solution$q_approx_linear

gw_plot(sol)
gw_matrix(sol, what = "value")

# the approximate value function can be visualized for states
# with two features.
approx_V_plot(sol)

# extracting approximate Q-values
approx_greedy_action(sol, "s(4,5)")
approx_Q_value(sol, "s(4,5)", "down")
approx_Q_value(sol)

# extracting a greedy policy using the approximate Q-values
approx_greedy_policy(sol)


# Example 2: Stuart Russell's 3x4 Maze using linear basis approximation
# The wall and the -1 absorbing state make linear approximation
# using just the position directly more difficult.

data(Maze)
gw_plot(Maze)

# if no state features are specified, then they are constructed
# by parsing the state label of the form s(feature list).
set.seed(1000)
sol <- solve_MDP_APPROX(Maze, horizon = 100, n = 100,
                        alpha = schedule_exp(0.3, 0.01),
                        epsilon = schedule_exp(1, 0.1))
gw_plot(sol)
gw_matrix(sol, what = "value")
approx_V_plot(sol, res = 20)


# Example 3: Stuart Russell's 3x4 Maze using
#            order-1 Fourier basis for approximation and
#            1-step Sarsa

set.seed(1000)
sol <- solve_MDP_APPROX(Maze, horizon = 100, n = 100,
                    alpha = schedule_exp(0.3, .01),
                    epsilon = schedule_exp(1, .1),
                    transformation = transformation_fourier_basis,
                    order = 1
                    )

gw_plot(sol)
gw_matrix(sol, what = "value")
approx_V_plot(sol, res = 20)
```

```
# Example 4: Stuart Russell's 3x4 Maze using
#    order-1 Fourier basis for approximation
#    and eligibility traces: Sarsa(lambda)

set.seed(1000)

## TODO: The following example does not converge to 1!
data(Maze)
sol <- solve_MDP_APPROX(Maze, horizon = 100, n = 100,
                    alpha = schedule_exp(0.3, .01),
                    epsilon = schedule_exp(1, .1),
                    lambda = 0.1,
                    transformation = transformation_fourier_basis,
                    order = 1
                    )

gw_plot(sol)
gw_matrix(sol, what = "value")
approx_V_plot(sol, res = 20)
```

`solve_MDP_DP` *Solve MDPs using Dynamic Programming*

Description

Solve MDPs via policy and value iteration.

Usage

```
solve_MDP_DP(
  model,
  method = "VI",
  horizon = NULL,
  discount = NULL,
  n = 1000L,
  error = 0.001,
  k_backups = 10L,
  V = NULL,
  ...,
  matrix = TRUE,
  continue = FALSE,
  verbose = FALSE,
  progress = TRUE
)
```

Arguments

model	an MDP problem specification.
method	string; one of the following solution methods: • 'VI' - value iteration • 'PI' - policy iteration • 'GenPS', 'PS_error', 'PS_random' - prioritized sweeping
horizon	an integer with the number of epochs for problems with a finite planning horizon. If set to Inf, the algorithm continues running iterations till it converges to the infinite horizon solution. If NULL, then the horizon specified in model will be used.
discount	discount factor in range $(0, 1]$. If NULL, then the discount factor specified in model will be used.
n	maximum number of iterations allowed to converge. If the maximum is reached then the non-converged solution is returned with a warning.
error	value iteration: maximum Bellman error allowed for the convergence criterion.
k_backups	policy iteration: maximum number of Bellman backups used in the iterative policy evaluation step. Policy evaluation typically converges earlier with a maximum Bellman error less than error.
V	a vector with initial state values. If NULL, then the default of a vector of all 0s (V_zero()) is used.
...	further parameters are passed on to the solver function.
matrix	logical; if TRUE then matrices for the transition model and the reward function are taken from the model first. This can be slow if functions need to be converted or do not fit into memory if the models are large. If these components are already matrices, then this is very fast. For FALSE, the transition probabilities and the reward is extracted when needed. This is slower, but removes the time and memory requirements needed to calculate the matrices.
continue	logical; show a progress bar with estimated time for completion.
verbose	logical or a numeric verbose level; if set to TRUE or 1, the function displays the used algorithm parameters and progress information. Levels >1 provide more detailed solver output in the R console.
progress	logical; show a progress bar with estimated time for completion.

Details

The following dynamic programming methods are implemented using the algorithms presented in Russell and Norvig (2010).

- **(Modified) Policy Iteration** (Howard 1960; Puterman and Shin 1978) starts with a random policy and iteratively performs a sequence of
  1. (Approximate) policy evaluation to estimate the value function for the current policy. Iterative policy evaluation can be approximated by stopping early after k_backups iterations (see policy_evaluation(). In this case the algorithm is called *modified* policy iteration.

2. Policy improvement is performed by updating the policy to be greedy (see greedy_policy()) with respect to the new value function. The algorithm stops when it converges to a stable policy (i.e., no changes between two iterations). Note that the policy typically stabilizes before the value function converges.

- **Value Iteration** (Bellman 1957) starts with an arbitrary value function (by default all 0s) and iteratively updates the value function for each state using the Bellman update equation (see bellman_update()).

$$v(s) \leftarrow \max_{a \in \mathcal{A}(s)} \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v(s')]$$

The iteration is terminated when the solution converges or the maximum of n iterations has been reached. Approximate convergence is achieved for discounted problems (with $\gamma < 1$) when the maximal value function change for any state δ is

$$\delta \leq \frac{error(1-\gamma)}{\gamma}.$$

It can be shown that this means that no state value is more than $error$ from the value in the optimal value function. For undiscounted problems, we use $\delta \leq error$.

A greedy policy is extracted from the final value function. Value iteration can be seen as policy iteration with policy evaluation truncated to one step.

- **Prioritized Sweeping** (Moore and Atkeson, 1993; Andre et al., 1997; Li and Littman, 2008) approximate the optimal value function by iteratively adjusting one state at a time. While value and policy iteration sweep in every iteration through all states, prioritized sweeping updates states in the order given by their priority. The priority reflects how much a state value may change given the most recently updated other states that can be directly reached via an action. This update order often lead to faster convergence compared to sweeping the whole state space in regular value iteration.

We implement the two priority update strategies described as **PS** and **GenPS** by Li and Littman (2008).

  - **PS** (Moore and Atkeson, 1993) updates the priority of a state $H(s)$ using:

$$\forall s \in \mathcal{S} : H_{t+1}(s) \leftarrow \begin{cases} \max(H_t(s), \Delta_t \max_{a \in \mathcal{A}}(p(s_t|s,a)) & \text{for } s \neq s_{t+1} \\ \Delta_t \max_{a \in A}(p(s_t|s,a) & \text{for } s = s_{t+1} \end{cases}$$

where $\Delta_t = |V_{t+1}(s_t) - V_t(s_t)| = |E(s_t; V_{t+1})|$, i.e., the Bellman error for the updated state.

  - **GenPS** (Andre et al., 1997) updates all state priorities using their current Bellman error:

$$\forall s \in \mathcal{S} : H_{t+1}(s) \leftarrow |E(s; V_{t+1})|$$

where $E(s; V_{t+1}) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s \in S} p(s'|s,a) V(s')\right] - V(s)$ is a state's Bellman error.

The update method can be chosen using the additional parameter H_update as the character string "PS_random", "PS_error" or "GenPS". The default is H_update = "GenPS". For PS, random means that the priority vector is initialized with random values (larger than 0), and error means they are initialized with the Bellman error as in GenPS. However, this requires one complete sweep over all states.

This implementation stops updating when the largest priority values over all states is less than the specified `error`.

Since the algorithm does not sweep through the whole state space for each iteration, `n` is converted into an equivalent number of state updates $n = n\,|S|$.

Value

`solve_MDP()` returns an object of class MDP or MDPSample which is a list with the model specifications (`model`), the solution (`solution`). The solution is a list with the elements that depend on the used method. Common elements are:

- `method` with the name of the used method
- parameters used.
- `converged` did the algorithm converge (`NA`) for finite-horizon problems.
- `policy` a list representing the policy graph. The list only has one element for converged solutions.

References

Andre, D., Friedman, N., and Parr, R. 1997. "Generalized prioritized sweeping." In Advances in Neural Information Processing Systems 10, pp. 1001-1007. NeurIPS Proceedings

Bellman, Richard. 1957. "A Markovian Decision Process." Indiana University Mathematics Journal 6: 679-84. https://www.jstor.org/stable/24900506.

Howard, R. A. 1960. "Dynamic Programming and Markov Processes." Cambridge, MA: MIT Press.

Li, Lihong, and Michael Littman. 2008. "Prioritized Sweeping Converges to the Optimal Value Function." DCS-TR-631. Rutgers University. doi:10.7282/T3TX3JSX

Moore, Andrew, and C. G. Atkeson. 1993. "Prioritized Sweeping: Reinforcement Learning with Less Data and Less Real Time." Machine Learning 13 (1): 103–30. doi:10.1007/BF00993104.

Puterman, Martin L., and Moon Chirl Shin. 1978. "Modified Policy Iteration Algorithms for Discounted Markov Decision Problems." Management Science 24: 1127-37. doi:10.1287/mnsc.24.11.1127.

Russell, Stuart J., and Peter Norvig. 2020. Artificial Intelligence: A Modern Approach (4th Edition). Pearson. http://aima.cs.berkeley.edu/.

See Also

Other solver: `convergence_horizon()`, `schedule`, `solve_MDP()`, `solve_MDP_APPROX()`, `solve_MDP_LP()`, `solve_MDP_MC()`, `solve_MDP_PG()`, `solve_MDP_SAMP()`, `solve_MDP_TD()`

Examples

```
data(Maze)

maze_solved <- solve_MDP(Maze, method = "DP:VI", verbose = TRUE)
policy(maze_solved)

# use prioritized sweeping (which is known to be fast for mazes)
maze_solved <- solve_MDP(Maze, method = "DP:GenPS", verbose = TRUE)
```

```
policy(maze_solved)

# finite horizon
maze_solved <- solve_MDP(Maze, method = "DP:VI", horizon = 3)
policy(maze_solved)
gw_plot(maze_solved, epoch = 1)
gw_plot(maze_solved, epoch = 2)
gw_plot(maze_solved, epoch = 3)
```

`solve_MDP_LP` *Solve MDPs using Linear Programming*

Description

Solve discounted, infinite horizon MDPs via linear programming.

Usage

```
solve_MDP_LP(
  model,
  method = "LP",
  horizon = NULL,
  discount = NULL,
  inf = 1000,
  lpSolve_args = list(),
  ...,
  matrix = NULL,
  continue = FALSE,
  verbose = FALSE,
  progress = NULL
)
```

Arguments

`model`	an MDP problem specification.
`method`	string; one of the following solution methods: `'LP'`
`horizon`	Only infinite-horizon MDPs with `horizon = Inf` are supported.
`discount`	only undiscounted MDPs with `discount = 1` are supported.
`inf`	value used for infinity when calling `lpSolve::lp()`. This should me much larger than the largest absolute reward in the model.
`lpSolve_args`	a list with additional arguments passed on to `lpSolve::lp()`.
`...`	further parameters are passed on to the solver function.

matrix	logical; if TRUE then matrices for the transition model and the reward function are taken from the model first. This can be slow if functions need to be converted or do not fit into memory if the models are large. If these components are already matrices, then this is very fast. For FALSE, the transition probabilities and the reward is extracted when needed. This is slower, but removes the time and memory requirements needed to calculate the matrices.
continue	logical; show a progress bar with estimated time for completion.
verbose	logical or a numeric verbose level; if set to TRUE or 1, the function displays the used algorithm parameters and progress information. Levels >1 provide more detailed solver output in the R console.
progress	not supported by this solver.

Details

A linear programming formulation was developed by Manne (1960) and further described by Puterman (1996). For the optimal value function, the Bellman equation holds:

$$v^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s, a, s')[r(s, a, s') + \gamma v^*(s')] \; \forall a \in \mathcal{A}, s \in \mathcal{S}$$

The maximization problem can reformulate as a minimization with a linear constraint for each state action pair. The optimal value function can be found by solving the following linear program:

$$\min \sum_{s \in S} v(s)$$

subject to

$$v(s) \geq \sum_{s' \in \mathcal{S}} p(s, a, s')[r(s, a, s') + \gamma v(s')], \; \forall a \in \mathcal{A}, s \in \mathcal{S}$$

Note:

- The discounting factor has to be strictly less than 1.
- The used solver does not support infinity and a sufficiently large value needs to be used instead (see parameter inf).
- Additional parameters to to solve_MDP are passed on to lpSolve::lp().

Value

solve_MDP() returns an object of class MDP or MDPSample which is a list with the model specifications (model), the solution (solution). The solution is a list with the elements that depend on the used method. Common elements are:

- method with the name of the used method
- parameters used.
- converged did the algorithm converge (NA) for finite-horizon problems.
- policy a list representing the policy graph. The list only has one element for converged solutions.

References

Manne, Alan. 1960. "On the Job-Shop Scheduling Problem." Operations Research 8 (2): 219-23. doi:10.1287/opre.8.2.219.

Puterman, Martin L. 1996. Markov decision processes: discrete stochastic dynamic programming. John Wiley & Sons.

See Also

Other solver: convergence_horizon(), schedule, solve_MDP(), solve_MDP_APPROX(), solve_MDP_DP(), solve_MDP_MC(), solve_MDP_PG(), solve_MDP_SAMP(), solve_MDP_TD()

Examples

```
data(Maze)

# we change the discount to 0.9 since LP is only implemented for discounted MDPs.
maze_solved <- solve_MDP(Maze, discount = 0.9, method = "LP:LP", verbose = TRUE)
maze_solved
policy(maze_solved)
```

---

solve_MDP_MC *Solve MDPs using Monte Carlo Control*

Description

Solve MDPs using Monte Carlo control.

Usage

```
solve_MDP_MC(
  model,
  method = "exploring_starts",
  horizon = NULL,
  discount = NULL,
  n = 100,
  Q = NULL,
  epsilon = NULL,
  alpha = NULL,
  first_visit = TRUE,
  ...,
  matrix = TRUE,
  continue = FALSE,
  progress = TRUE,
  verbose = FALSE
)
```

Arguments

model	an MDP problem specification.
method	string; one of the following solution methods: • 'exploring_starts' - on-policy MC control with exploring starts. • 'on_policy' - on-policy MC control with an ϵ-greedy policy. • 'off_policy"' - off-policy MC control using an ϵ-greedy behavior policy.
horizon	an integer with the number of epochs for problems with a finite planning horizon. If set to Inf, the algorithm continues running iterations till it converges to the infinite horizon solution. If NULL, then the horizon specified in model will be used.
discount	discount factor in range $(0, 1]$. If NULL, then the discount factor specified in model will be used.
n	number of episodes used for learning.
Q	an initial state-action value matrix. By default an all 0 matrix is used.
epsilon	used for the ϵ-greedy behavior policies. A scalar value between 0 and 1 or a schedule.
alpha	step size (learning rate). A scalar value between 0 and 1 or a schedule.
first_visit	if TRUE then only the first visit of a state/action pair in an episode is used to update Q, otherwise, every-visit update is used.
...	further parameters are passed on to the solver function.
matrix	logical; if TRUE then matrices for the transition model and the reward function are taken from the model first. This can be slow if functions need to be converted or do not fit into memory if the models are large. If these components are already matrices, then this is very fast. For FALSE, the transition probabilities and the reward is extracted when needed. This is slower, but removes the time and memory requirements needed to calculate the matrices.
continue	logical; show a progress bar with estimated time for completion.
progress	logical; show a progress bar with estimated time for completion.
verbose	logical or a numeric verbose level; if set to TRUE or 1, the function displays the used algorithm parameters and progress information. Levels >1 provide more detailed solver output in the R console.

Details

The idea is to estimate the action value function for a policy as the average of sampled returns.

$$q_\pi(s, a) = \mathbb{E}_\pi[R_i | S_0 = s, A_0 = a] \approx \frac{1}{n} \sum_{i=1}^{n} R_i$$

Monte Carlo control simulates a whole episode using the current behavior policy and uses the sampled reward to update the Q values. For on-policy methods, the behavior policy is updated to be greedy (i.e., optimal) with respect to the new Q values. Then the next episode is simulated till the predefined number of episodes is completed.

Implemented methods:

Implemented are the following temporal difference control methods described in Sutton and Barto (2018).

- **Monte Carlo Control with exploring Starts** learns the optimal greedy policy. It uses the same greedy policy for behavior and target (on-policy learning). After each episode, the policy is updated to be greedy with respect to the current Q values. To make sure all states/action pairs are explored, it uses exploring starts meaning that new episodes are started at a randomly chosen state using a randomly chooses action.
- **On-policy Monte Carlo Control** learns an epsilon-greedy policy which it uses for behavior and as the target policy (on-policy learning). An epsilon-greedy policy is used to provide exploration. For calculating running averages, an update with $\alpha = 1/n$ is used by default. A different update factor can be set using the parameter `alpha` as either a fixed value or a function with the signature `function(t, n)` which returns the factor in the range $[0, 1]$.
- **Off-policy Monte Carlo Control** uses for behavior an arbitrary soft policy (a soft policy has in each state a probability greater than 0 for all possible actions). We use an epsilon-greedy policy and the method learns a greedy policy using importance sampling. Note: This method can only learn from the tail of the sampled runs where greedy actions are chosen. This means that it is very inefficient in learning the beginning portion of long episodes. This problem is especially problematic when larger values for ϵ are used.

Schedules:

- epsilon schedule: `t` is increased by each processed episode.
- alpha schedule: `t` is set to the number of times the a Q-value for state-action combination was updated.

Value

`solve_MDP()` returns an object of class MDP or MDPSample which is a list with the model specifications (`model`), the solution (`solution`). The solution is a list with the elements that depend on the used method. Common elements are:

- `method` with the name of the used method
- parameters used.
- `converged` did the algorithm converge (`NA`) for finite-horizon problems.
- `policy` a list representing the policy graph. The list only has one element for converged solutions.

References

Sutton, Richard S., and Andrew G. Barto. 2018. Reinforcement Learning: An Introduction. Second. The MIT Press. http://incompleteideas.net/book/the-book-2nd.html.

See Also

Other solver: `convergence_horizon()`, `schedule`, `solve_MDP()`, `solve_MDP_APPROX()`, `solve_MDP_DP()`, `solve_MDP_LP()`, `solve_MDP_PG()`, `solve_MDP_SAMP()`, `solve_MDP_TD()`

`solve_MDP_PG` *Solve MDPs with Policy Gradient Methods*

Description

Not implemented yet: Solve the MDP control problem using a parameterized policy and policy gradient methods. The implemented method one-step actor-critic control and actor-critic control with eligibility traces.

Usage

```
solve_MDP_PG(
  model,
  method = "actor-critic",
  horizon = NULL,
  discount = NULL,
  alpha_actor = schedule_exp(0.2, 0.1),
  alpha_critic = schedule_exp(0.2, 0.1),
  epsilon = schedule_exp(1, 0.1),
  lambda = 0,
  n,
  w = NULL,
  theta = NULL,
  ...,
  matrix = TRUE,
  continue = FALSE,
  progress = TRUE,
  verbose = FALSE
)
```

Arguments

`model`	an MDP problem specification.
`method`	string; one of the following solution methods: `'sarsa'`
`horizon`	an integer with the number of epochs for problems with a finite planning horizon. If set to `Inf`, the algorithm continues running iterations till it converges to the infinite horizon solution. If `NULL`, then the horizon specified in `model` will be used.
`discount`	discount factor in range $(0, 1]$. If `NULL`, then the discount factor specified in `model` will be used.
`alpha_actor, alpha_critic`	alpha schedules
`epsilon`	used for the ϵ-greedy behavior policies. A scalar value between 0 and 1 or a schedule.
`lambda`	the trace-decay parameter for the an accumulating trace. If `lambda = 0` then 1-step Sarsa is used.

n	number of episodes used for learning.
w	an initial weight vector. By default a vector with 0s is used.
theta	parameter...
...	further parameters are passed on to the solver function.
matrix	logical; if TRUE then matrices for the transition model and the reward function are taken from the model first. This can be slow if functions need to be converted or do not fit into memory if the models are large. If these components are already matrices, then this is very fast. For FALSE, the transition probabilities and the reward is extracted when needed. This is slower, but removes the time and memory requirements needed to calculate the matrices.
continue	logical; show a progress bar with estimated time for completion.
progress	logical; show a progress bar with estimated time for completion.
verbose	logical or a numeric verbose level; if set to TRUE or 1, the function displays the used algorithm parameters and progress information. Levels >1 provide more detailed solver output in the R console.

Value

solve_MDP() returns an object of class MDP or MDPSample which is a list with the model specifications (model), the solution (solution). The solution is a list with the elements that depend on the used method. Common elements are:

- method with the name of the used method
- parameters used.
- converged did the algorithm converge (NA) for finite-horizon problems.
- policy a list representing the policy graph. The list only has one element for converged solutions.

References

Sutton, Richard S., and Andrew G. Barto. 2018. Reinforcement Learning: An Introduction. Second. The MIT Press. http://incompleteideas.net/book/the-book-2nd.html.

See Also

Other solver: convergence_horizon(), schedule, solve_MDP(), solve_MDP_APPROX(), solve_MDP_DP(), solve_MDP_LP(), solve_MDP_MC(), solve_MDP_SAMP(), solve_MDP_TD()

Other MDPSample: MDPSample(), absorbing_states(), act(), action_state_helpers, reachable_states(), sample_MDP.MDPSample(), start

Examples

```
# EXAMPLE
```

`solve_MDP_SAMP` *Solve MDPs using Random-Sampling*

Description

Solve MDPs via random-sampling. Implemented is Sutton and Barto's one-step random-sampling one-step tabular Q-planning.

Usage

```
solve_MDP_SAMP(
  model,
  method = "q_planning",
  horizon = NULL,
  discount = NULL,
  alpha = schedule_exp(0.2, 0.1),
  n = 1000,
  Q = NULL,
  ...,
  matrix = TRUE,
  continue = FALSE,
  progress = TRUE,
  verbose = FALSE
)
```

Arguments

`model`	an MDP problem specification.
`method`	string; one of the following solution methods: `'q_planning'`
`horizon`	an integer with the number of epochs for problems with a finite planning horizon. If set to `Inf`, the algorithm continues running iterations till it converges to the infinite horizon solution. If `NULL`, then the horizon specified in `model` will be used.
`discount`	discount factor in range $(0, 1]$. If `NULL`, then the discount factor specified in `model` will be used.
`alpha`	step size (learning rate). A scalar value between 0 and 1 or a schedule.
`n`	number of episodes used for learning.
`Q`	an initial state-action value matrix. By default an all 0 matrix is used.
`...`	further parameters are passed on to the solver function.
`matrix`	logical; if `TRUE` then matrices for the transition model and the reward function are taken from the model first. This can be slow if functions need to be converted or do not fit into memory if the models are large. If these components are already matrices, then this is very fast. For `FALSE`, the transition probabilities and the reward is extracted when needed. This is slower, but removes the time and memory requirements needed to calculate the matrices.

continue	logical; show a progress bar with estimated time for completion.
progress	logical; show a progress bar with estimated time for completion.
verbose	logical or a numeric verbose level; if set to TRUE or 1, the function displays the used algorithm parameters and progress information. Levels >1 provide more detailed solver output in the R console.

Details

Random-sample one-step tabular Q-planning is a simple, not very effective, planning method shown as an illustration in Chapter 8 of Sutton and Barto (2018). It randomly selects a state/action pair and samples the following state s' to perform a single one-step update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha * (R + \gamma \max_{a}'(Q(s',a')) - Q(s,a))$$

Schedules:

- alpha schedule: t is set to the number of times the a Q-value for state s was updated.

Value

solve_MDP() returns an object of class MDP or MDPSample which is a list with the model specifications (model), the solution (solution). The solution is a list with the elements that depend on the used method. Common elements are:

- method with the name of the used method
- parameters used.
- converged did the algorithm converge (NA) for finite-horizon problems.
- policy a list representing the policy graph. The list only has one element for converged solutions.

References

Sutton, Richard S., and Andrew G. Barto. 2018. Reinforcement Learning: An Introduction. Second. The MIT Press. http://incompleteideas.net/book/the-book-2nd.html.

See Also

Other solver: convergence_horizon(), schedule, solve_MDP(), solve_MDP_APPROX(), solve_MDP_DP(), solve_MDP_LP(), solve_MDP_MC(), solve_MDP_PG(), solve_MDP_TD()

`solve_MDP_TD` *Solve MDPs using Tabular Temporal Differencing*

Description

Solve MDPs using tabular temporal difference control methods like q-learning and 1-step, n-step Sarsa, Sarsa(λ), and Q(λ).

Usage

```
solve_MDP_TD(
  model,
  method = "sarsa",
  horizon = NULL,
  discount = NULL,
  alpha = schedule_exp(0.2, 0.01),
  epsilon = schedule_exp(1, 0.1),
  n_step = 1,
  lambda = 0,
  on_policy = TRUE,
  n,
  Q = 0,
  ...,
  matrix = TRUE,
  continue = FALSE,
  progress = TRUE,
  verbose = FALSE
)
```

Arguments

`model`	an MDP problem specification.
`method`	string; one of the following solution methods: `"sarsa"`, `"q"`, `"q_learning"`, or `"expected_sarsa"`. Not all methods are available for all setting s of `n_step` and `lambda` (see Details).
`horizon`	an integer with the number of epochs for problems with a finite planning horizon. If set to `Inf`, the algorithm continues running iterations till it converges to the infinite horizon solution. If `NULL`, then the horizon specified in `model` will be used.
`discount`	discount factor in range $(0, 1]$. If `NULL`, then the discount factor specified in `model` will be used.
`alpha`	step size (learning rate). A scalar value between 0 and 1 or a schedule.
`epsilon`	used for the ϵ-greedy behavior policies. A scalar value between 0 and 1 or a schedule.
`n_step`	number of steps to look ahead for n-step Sarsa.

lambda	eligibility trace decay factor for Sarsa(lambda).
on_policy	logical; should we learn on-policy (the behavior policy) vs. off-policy (using importance sampling)? Only used for method "sarsa".
n	number of episodes used for learning.
Q	an initial state-action value matrix. By default an all 0 matrix is used.
...	further parameters are passed on to the solver function.
matrix	logical; if TRUE then matrices for the transition model and the reward function are taken from the model first. This can be slow if functions need to be converted or do not fit into memory if the models are large. If these components are already matrices, then this is very fast. For FALSE, the transition probabilities and the reward is extracted when needed. This is slower, but removes the time and memory requirements needed to calculate the matrices.
continue	logical; show a progress bar with estimated time for completion.
progress	logical; show a progress bar with estimated time for completion.
verbose	logical or a numeric verbose level; if set to TRUE or 1, the function displays the used algorithm parameters and progress information. Levels >1 provide more detailed solver output in the R console.

Details

Implemented are several tabular temporal difference control methods described in Sutton and Barto (2018). Note that the MDP transition and reward models are used for these reinforcement learning methods only to sample from the environment.

The implementation uses an ϵ-greedy behavior policy, where the parameter epsilon controls the degree of exploration. The algorithms use a step size parameter α (learning rate). The epsilon and the learning rate alpha can be specified as fixed numbers between 0 and 1 or as a schedule function where the value is gradually reduced (see Schedule section below).

All temporal differencing methods us the following update:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[G_t - Q(S_t, A_t)],$$

where G_t is the target estimate for the given q-value. The different methods below estimate the target value differently (see 1 and n-step methods below).

If the model has absorbing states to terminate episodes, then no maximal episode length (horizon) needs to be specified. To make sure that the algorithm does finish in a reasonable amount of time, episodes are stopped after 1000 actions (with a warning). For models without absorbing states, the episode length has to be specified via horizon.

1-step Methods:

1-step methods estimate the return by just looking one state ahead.

- q_learning: **Q-Learning** (Watkins and Dayan 1992) is an off-policy (learns a greedy target policy) temporal difference method. We use here an ϵ-greedy behavior policy. The update target value is estimated by one-step bootstrapping using the reward and the value of the state following the current greedy target policy:

$$G_t = R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$$

- `sarsa`: **Sarsa** (Rummery and Niranjan 1994) is an on-policy method (behavior and target policy are the same). We use an ϵ-greedy policy and the final ϵ-greedy policy is converted into a greedy policy. ϵ can be lowered over time (see schedule and parameter `continue`) to learn an approximately greedy policy. The target is estimated as the one-step bootstrap estimate following the current behavior policy:

$$G_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$$

- `expected_sarsa` **Expected Sarsa** (Sutton and Barto 2018) learns the behavior policy (on-policy learning). We use Sarsa with an ϵ-greedy policy which uses the the expected value under the current policy for the update:

$$G_t = R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a)$$

  Expected Sarsa moves in the same direction as Sarsa would move in expectation. Because it uses the expectation, we can set the step size α to large values and 1 is common.
  The Q-learning algorithm can be seen as a simplification of the off-policy version of expected Sarsa.

n-step Methods:

n-step methods use a longer look ahead. The return is estimated by looking n time steps ahead (`n_step` in the code) and using the rewards and then the value of the reached state:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}, A_{t+n})$$

While n-step methods conceptually look ahead, the implementation has to wait for the values to become available. This means that updates are n steps delayed, i.e., the update for step t is performed at $t + n$.

Methods:

- `sarsa`: **n-step Sarsa** (Sutton and Barto 2018). The estimated return is used as the update target for Sarsa.
  `n_step` = 1 is regular 1-step Sarsa. Using `n_step` = `Inf` is equivalent to Monte Carlo Control, however, `solve_MDP_MC()` is more memory efficient.
  Sarsa learns on-policy (i.e., the behavioral policy which is often an exploring ϵ-greedy policy). For the off-policy case, when the optimal policy is learned, the update are corrected using the importance sampling ratio.

Eligibility Traces:

Eligibility traces also look ahead but the update is performed in a backward looking manner by using a memory parameter called the trace that remember what states or Q-values lead to the current event. We focus here on tabular Sarsa(λ)

From the forward-view perspective, the λ-reward can be seen as an average of infinite n-step backups:

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)},$$

where λ is the eligibility trace decay factor. The implementation uses the backward view with the TD-error

$$\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$$

and the update

$$Q(s,a) = Q(s,a) + \alpha\delta_t z_t(s,a) \qquad \forall s,a.$$

The trace is updated at each time step as

$$z_t(s,a) = \gamma z_{t-1}(s,a) \qquad \forall s,a$$

and $z_t(S_t,A_t) = z_t(S_t,A_t)+1$. This means, the trace is set to one for the current state and action and then decays with the parameter λ.

Methods:

- `sarsa`: **Sarsa(lambda)** (Sutton and Barto 2018).
- `q`: **Watkins's Q(lambda)** (Sutton and Barto 2018). An off-policy learning algorithm similar to Q-learning which looks only ahead as far as the next exploratory action (i.e., the eligibility vector $z_t(\cdot,\cdot)$ is set to zero whenever an exploratory non-greedy exploratory action is taken).

Schedules:

- epsilon schedule: `t` is increased by each processed episode.
- alpha schedule: `t` is set to the number of times the a Q-value for state `s` was updated.

Value

`solve_MDP()` returns an object of class MDP or MDPSample which is a list with the model specifications (`model`), the solution (`solution`). The solution is a list with the elements that depend on the used method. Common elements are:

- `method` with the name of the used method
- parameters used.
- `converged` did the algorithm converge (`NA`) for finite-horizon problems.
- `policy` a list representing the policy graph. The list only has one element for converged solutions.

References

Rummery, G., and Mahesan Niranjan. 1994. "On-Line Q-Learning Using Connectionist Systems." Techreport CUED/F-INFENG/TR 166. Cambridge University Engineering Department.

Sutton, R. 1988. "Learning to Predict by the Method of Temporal Differences." Machine Learning 3: 9-44. https://link.springer.com/article/10.1007/BF00115009.

Sutton, Richard S., and Andrew G. Barto. 2018. Reinforcement Learning: An Introduction. Second. The MIT Press. http://incompleteideas.net/book/the-book-2nd.html.

Watkins, Christopher J. C. H., and Peter Dayan. 1992. "Q-Learning." Machine Learning 8 (3): 279-92. doi:10.1007/BF00992698.

See Also

Other solver: `convergence_horizon()`, `schedule`, `solve_MDP()`, `solve_MDP_APPROX()`, `solve_MDP_DP()`, `solve_MDP_LP()`, `solve_MDP_MC()`, `solve_MDP_PG()`, `solve_MDP_SAMP()`

Examples

```
data(Maze)

# Example 1: Learn a Policy using Q-Learning
maze_learned <- solve_MDP(Maze, method = "TD:q_learning",
    n = 200, horizon = 100)
maze_learned

policy(maze_learned)
gw_plot(maze_learned)

# Example 2: Learn a Policy using 1-step Sarsa
maze_learned <- solve_MDP(Maze, method = "TD:sarsa",
    n = 200, horizon = 100)
maze_learned

policy(maze_learned)
gw_plot(maze_learned)

# Example 3: Perform one episode for 3-step Sarsa

# run one episode in verbose mode.
maze_learned <- solve_MDP(Maze, method = "TD:sarsa",
    n_step = 5, n = 1, horizon = 100, verbose = 2)

# verbose output:
#  * tau ... time step updated (laggs n_step)
#  * -> ... update of the Q-value
#  * N ... number of updates for this state
#  * G ... reward estimate for the n_steps
#  * alpha ... learning rate (schedule may depend on N)
#  * rho ... importance sampling ratio (1 for on-policy learning)

# run more episode
maze_learned <- solve_MDP(Maze, method = "TD:sarsa",
    n_step = 5, n = 100, horizon = 100)
maze_learned
policy(maze_learned)
gw_plot(maze_learned)

# Example: Tabular Sarsa(lambda)
maze_learned <- solve_MDP(Maze, method = "TD:sarsa",
    lambda = .1, n = 100, horizon = 100)
maze_learned
policy(maze_learned)
gw_plot(maze_learned)
```

`start` *Sample a Start State*

Description

Samples a start state which can be used for simulation.

Usage

```
## S3 method for class 'MDPModel'
start(x, as = "factor", ...)

## S3 method for class 'MDPSample'
start(x, as = "features", ...)
```

Arguments

x	a model.
as	a returned format. See parameter as in normalize_state().
...	additional parameters are ignored.

Value

a start state.

Author(s)

Michael Hahsler

See Also

Other MDP: MDP(), absorbing_states(), act(), action_state_helpers, available_actions(), find_reachable_states(), reachable_states(), sample_MDP(), sample_MDP.MDPSample(), transition_graph(), transition_matrix(), unreachable_states()

Other MDPSample: MDPSample(), absorbing_states(), act(), action_state_helpers, reachable_states(), sample_MDP.MDPSample(), solve_MDP_PG()

Examples

```
data(Maze)

start(Maze)
```

`transformation` *Transformation Functions for Linear Function Approximation*

Description

Several popular transformation functions applied to state features used in linear function approximation for `solve_MDP_APPROX()`.

Usage

```
transformation_linear_basis(model, min = NULL, max = NULL, intercept = TRUE)

transformation_polynomial_basis(
  model,
  min = NULL,
  max = NULL,
  order,
  coefs = NULL
)

transformation_RBF_basis(model, min = NULL, max = NULL, centers, var = NULL)

transformation_fourier_basis(
  model,
  min = NULL,
  max = NULL,
  order,
  coefs = NULL
)

create_basis_coefs(dim, order)
```

Arguments

`model`	the MDP model.
`min, max`	vectors with the minimum and maximum values for each feature. This is used to scale the feature to the $[0, 1]$ interval for the Fourier basis.
`intercept`	logical; add an intercept term to the linear basis?
`order`	order for the Fourier basis.
`coefs`	an optional matrix or data frame to specify the set of coefficient values for the Fourier basis (overrides `order`).
`centers`	a scalar with the number of centers to create a a regular grid with that many steps per feature dimension. Alternatively, a matrix with the centers for the RBF can be supplied.
`var`	a scalar with the variance used for the RBF.
`dim`	number of features to describe a state.

Details

The state feature function $\phi()$ uses the raw state feature vectors $\mathbf{x} = (x_1, x_2, ..., x_m)$ which is either user-specified or constructed by parsing the state labels of form `s(feature list)` and then applies a transformation functions called basis functions. Implemented basis functions are:

- Linear: no additional transformation is applied giving $\phi_0(s) = 1$ for the intercept and $\phi_i(s) = x_i$ for $i = \{1, 2, ..., m\}$.
- Polynomial basis:

$$\phi_i(s) = \prod_{j=1}^{m} x_j^{c_{i,j}},$$

  where $c_{1,j}$ is an integer between 0 and n for and order n polynomial basis.
- Radial Basis: RBF.
- Fourier basis:

$$\phi_i(s) = \cos(\pi \mathbf{c}^i \cdot \mathbf{x}),$$

  where $\mathbf{c}^i = [c_1, c_2, ..., c_m]$ with $c_j = [0, ..., n]$, where n is the order of the basis. The components of the feature vector x are assumed to be scaled to the interval $[0, 1]$. The fourier basis transformation is implemented in `transformation_fourier_basis()`. `min` and `max` are the minimums and maximums for each feature vector component used to resale them to $[0, 1]$ using $\frac{x_i - min_i}{max_i - min_i}$
  Details of this transformation are described in Konidaris et al (2011).

Value

A transformation function

References

Sutton, Richard S., and Andrew G. Barto. 2018. Reinforcement Learning: An Introduction. Second. The MIT Press. http://incompleteideas.net/book/the-book-2nd.html.

Alborz Geramifard, Thomas J. Walsh, Stefanie Tellex, Girish Chowdhary, Nicholas Roy, and Jonathan P. How. 2013. A Tutorial on Linear Function Approximators for Dynamic Programming and Reinforcement Learning. Foundations and Trends in Machine Learning 6(4), December 2013, pp. 375-451. doi:10.1561/2200000042

Konidaris, G., Osentoski, S., & Thomas, P. 2011. Value Function Approximation in Reinforcement Learning Using the Fourier Basis. Proceedings of the AAAI Conference on Artificial Intelligence, 25(1), 380-385. doi:10.1609/aaai.v25i1.7903

See Also

`solve_MDP_APPROX()`

Other approximation: `linear_function_approximation`, `solve_MDP_APPROX()`

`transition_graph` *Transition Graph*

Description

Returns the transition model as an **igraph** object.

Usage

```
transition_graph(
  x,
  action = NULL,
  state_col = NULL,
  simplify_transitions = TRUE,
  remove_unavailable_actions = TRUE
)

plot_transition_graph(
  x,
  action = NULL,
  state_col = NULL,
  simplify_transitions = TRUE,
  main = NULL,
  ...
)

curve_multiple_directed(graph, start = 0.3)
```

Arguments

`x`	object of class MDP.
`action`	the name or id of an action or a set of actions. By default the transition model for all actions is returned.
`state_col`	colors used to represent the states.
`simplify_transitions`	logical; combine parallel transition arcs into a single arc.
`remove_unavailable_actions`	logical; don't show arrows for unavailable actions.
`main`	a main title for the plot.
`...`	further arguments are passed on to `igraph::plot.igraph()`.
`graph`	The input graph.
`start`	The curvature at the two extreme edges.

Details

The transition model of an MDP is a Markov chain. This function extracts it as an **igraph** object.

Value

returns the transition model as an igraph object.

See Also

Other MDP: MDP(), absorbing_states(), act(), action_state_helpers, available_actions(), find_reachable_states(), reachable_states(), sample_MDP(), sample_MDP.MDPSample(), start, transition_matrix(), unreachable_states()

Other visualization: gridworld

Examples

```
data("Maze")

g <- transition_graph(Maze)
g

plot_transition_graph(Maze)
plot_transition_graph(Maze,
  vertex.size = 20,
  edge.label.cex = .1, edge.arrow.size = .5, margin = .5
)

## Plot using the igraph library
library(igraph)
plot(g)

# plot with a different layout
plot(g,
  layout = igraph::layout_with_sugiyama,
  vertex.size = 20,
  edge.label.cex = .6
)

## Use visNetwork (if installed)
if (require(visNetwork)) {
  g_vn <- toVisNetworkData(g)
  nodes <- g_vn$nodes
  edges <- g_vn$edges

  visNetwork(nodes, edges) %>%
    visNodes(physics = FALSE) %>%
    visEdges(smooth = list(type = "curvedCW", roundness = .6), arrows = "to")
}
```

`transition_matrix` *Access to Parts of the Model Description*

Description

Functions to provide uniform access to different parts of the MDP problem description.

Usage

```
transition_matrix(
  model,
  action = NULL,
  start.state = NULL,
  end.state = NULL,
  ...,
  sparse = NULL,
  drop = TRUE,
  simplify = FALSE,
  trans_keyword = TRUE
)

reward_matrix(
  model,
  action = NULL,
  start.state = NULL,
  end.state = NULL,
  ...,
  sparse = NULL,
  drop = TRUE,
  simplify = FALSE
)

start_vector(model, start = NULL, sparse = NULL)

normalize_MDP(
  model,
  transition_model = TRUE,
  reward = TRUE,
  start = FALSE,
  sparse = NULL,
  precompute_absorbing = TRUE,
  check_and_fix = FALSE,
  progress = TRUE
)
```

Arguments

model	A MDP object.
action	name or index of an action.
start.state, end.state	name or index of the state.
...	further arguments are passed on.

sparse	logical; use sparse matrix representation? NULL decides the representation based on the memory it would take to store the faster dense representation.
drop	logical; drop matrices to vectors when one row/one column is selected.
simplify	logical; try to simplify action lists into a vector or matrix?
trans_keyword	logical; translate keywords like "uniform" into matrices.
start	logical; convert the start probability distribution into a vector.
transition_model	logical; convert the transition probabilities into a list of matrices.
reward	logical; convert the reward model into a list of matrices.
precompute_absorbing	logical; should absorbing states be precalculated?
check_and_fix	logical; checks the structure of the problem description.
progress	logical; show a progress bar with estimated time for completion.

Details

Several parts of the MDP description can be defined in different ways. In particular, the fields transition_model, reward, and start can be defined using matrices, data frames, keywords, or functions. See MDP for details. The functions provided here give unified access to the data in these fields and make writing code easier.

Transition Probabilities $p(s'|s,a)$:

transition_matrix() accesses the transition model. The complete model is a list with one element for each action. Each element contains a states x states matrix with s (start.state) as rows and s' (end.state) as columns. Matrices with a low density can be requested in sparse format (as a Matrix::dgRMatrix). It is recommended to load package MatrixExtra to work with sparse matrices.

Reward $r(s,s',a)$:

reward_matrix() accesses the reward model. The preferred representation is a data.frame with the columns action, start.state, end.state, and value. This is a sparse representation.

The dense representation is a list of lists of matrices. The list levels are a (action) and s (start.state). The matrices are column vectors with rows representing s' (end.state).

To represent rewards as sparse matrices, **rewards that correspond to a transition with probability zero are set to zero if the transition model** is stored as a list of matrices. This makes the reward matrices as sparse as the transition matrices. The function normalize_MDP() with sparse = TRUE will perform this representation.

Start state:

start_vector() translates the start state description into a probability vector.

Sparse Matrices and Normalizing MDPs:

Different components can be specified in various ways. It is often necessary to convert each component into a specific form (e.g., a dense matrix) to save time when accessing it. Convert the Complete MDP Description into a consistent form normalize_MDP() converts all components of the MDP description into a consistent form and returns a new MDP definition where

transition_model, reward, and start are normalized. This includes the internal representation (dense, sparse, as a data.frame) and also, states, and actions are ordered as given in the problem definition to make safe access using numerical indices possible. Normalized MDP descriptions can be used in custom code that expects consistently a certain format.

The default behavior of sparse = NULL uses parse matrices for large models where the dense transition model would need more than options("MDP_SPARSE_LIMIT") (the default is about 100 MB which can be changed using options()). Smaller models use faster dense matrices.

Value

A list or a list of lists of matrices.

Author(s)

Michael Hahsler

See Also

Other MDP: MDP(), absorbing_states(), act(), action_state_helpers, available_actions(), find_reachable_states(), reachable_states(), sample_MDP(), sample_MDP.MDPSample(), start, transition_graph(), unreachable_states()

Examples

```
data("Maze")
gw_matrix(Maze)

# here is the internal structure of the Maze object
str(Maze)

# List of |A| transition matrices. One per action in the from start.states x end.states
Maze$transition_model
transition_matrix(Maze)
transition_matrix(Maze, action = "up", sparse = FALSE)
transition_matrix(Maze,
  action = "up",
  start.state = "s(3,1)", end.state = "s(2,1)"
)

# List of list of reward matrices. 1st level is action and second level is the
#  start state in the form of a column vector with elements for end states.
Maze$reward
reward_matrix(Maze)
reward_matrix(Maze, sparse = TRUE)
reward_matrix(Maze,
  action = "up",
  start.state = "s(3,1)", end.state = "s(2,1)"
)

# Translate the initial start probability vector
Maze$start
start_vector(Maze, sparse = FALSE)
```

```
start_vector(Maze, sparse = "states")
start_vector(Maze, sparse = "index")

# Normalize the whole model using sparse representation
Maze_norm <- normalize_MDP(Maze, sparse = TRUE)
str(Maze_norm)

# Note to make the reward matrix sparse, all rewards
# for transitions with probability of 0 are zeroed out.
reward_matrix(Maze_norm)
```

`unreachable_states` *Unreachable States*

Description

Find or removes unreachable states using the transition model.

Usage

```
unreachable_states(
  model,
  horizon = Inf,
  sparse = "states",
  progress = TRUE,
  ...
)

remove_unreachable_states(model, ...)
```

Arguments

`model`	a MDP object.
`horizon`	only states that can be reached within the horizon are reachable.
`sparse`	logical; return a sparse logical vector?
`progress`	logical; show a progress bar?
`...`	further arguments are passed on.

Details

The function `unreachable_states()` checks if states cannot be reached from any other state. It performs a depth-first search which can be slow. The search breaks cycles to avoid an infinite loop. The search depth can be restricted using `horizon`.

The function `remove_unreachable_states()` simplifies a model by removing unreachable states from the model description.

Value

`unreachable_states()` returns a logical vector indicating the unreachable states.

`remove_unreachable_states()` returns a model with all unreachable states removed.

Author(s)

Michael Hahsler

See Also

Other MDP: `MDP()`, `absorbing_states()`, `act()`, `action_state_helpers`, `available_actions()`, `find_reachable_states()`, `reachable_states()`, `sample_MDP()`, `sample_MDP.MDPSample()`, `start`, `transition_graph()`, `transition_matrix()`

Examples

```
# create a Maze with an unreachable state

maze_unreach <- gw_read_maze(
    textConnection(c("XXXXXX",
                     "XS X X",
                     "X  XXX",
                     "X   GX",
                     "XXXXXX")))
gw_plot(maze_unreach)

unreachable_states(maze_unreach)
unreachable_states(maze_unreach, sparse = FALSE)

maze <- remove_unreachable_states(maze_unreach)
unreachable_states(maze)
gw_plot(maze)
```

`value_function` *Value Function*

Description

Extracts the value function from a solved MDP.

Usage

```
value_function(model, drop = TRUE)

plot_value_function(
  model,
  epoch = 1L,
  legend = TRUE,
```

```
  col = NULL,
  ylab = "Value",
  las = 3,
  main = NULL,
  ...
)

V_zero(model, value = 0)

V_random(model, min = 1e-06, max = 1)
```

Arguments

model	a solved MDP.
drop	logical; drop the list for converged, epoch-independent value functions.
epoch	epoch for finite time horizon solutions.
legend	logical; show legend.
col, ylab, las	are passed on to graphics::barplot().
main	a main title for the plot. Defaults to the name of the problem.
...	further arguments are passed on to graphics::barplot()'.
value	value to initialize the value function with. Default is 0.
min, max	minimum and maximum for the uniformly distributed random state values.

Value

Returns the value function as a numeric vector with one value for each state or as a matrix with rows for states and columns for epochs.

Author(s)

Michael Hahsler

See Also

Other value_function: Q_values(), bellman_update()

Examples

```
data("Maze")
sol <- solve_MDP(Maze)
sol

value_function(sol)
plot_value_function(sol)

## finite-horizon problem
sol <- solve_MDP(Maze, horizon = 3)
policy(sol)
```

```
value_function(sol)
plot_value_function(sol, epoch = 1)
plot_value_function(sol, epoch = 2)
plot_value_function(sol, epoch = 3)

# For a gridworld we can also plot is like this
gw_plot(sol, epoch = 1)
gw_plot(sol, epoch = 2)
gw_plot(sol, epoch = 3)
```

`visit_probability` *State Visit Probability*

Description

Calculates the state visit probability (the modified stationary distribution) when following a policy from the start state.

Usage

```
visit_probability(model, pi = NULL, start = NULL, method = "power", ...)
```

Arguments

`model`	a solved MDP object.
`pi`	the used policy. If missing the policy in `model` is used.
`start`	specification of the start distribution. If missing the specification in `model` is used.
`method`	calculate the modified stationary distribution using `"power"` (power iteration) or `"sample"` (trajectory sampling).
`...`	further arguments are passed on to the internal implementations (see Details section).

Details

The visit probability is the stationary distribution for the transition matrix induced by the policy. To account for absorbing states, we modify the transition matrix by setting all outgoing probabilities from absorbing states to 0.

Power iteration:

The stationary distribution can be estimated as the sum of multiplying the start distribution repeatedly with the modified transition matrix induced by the policy. We stop multiplying when the largest difference between entries in the two consecutive vectors is less then the extra parameter:

- `min_err` stop criterion for the power iteration (default: `1e-6`).
- `sparse` logical; should a sparse transition matrix be used for the power iteration?

The resulting vector is normalized to probabilities.

Sample method:

The stationary distribution is calculated using `n` random walks. The state visit counts are normalized to a probabilities. Additional parameters are:

- `n` number of random walks (default `1000`).
- `horizon` maximal horizon used to stop a random walk if it has not reached an absorbing state.

Value

a visit probability vector over all states.

Author(s)

Michael Hahsler

See Also

Other policy: `action()`, `expected_return()`, `greedy_action()`, `policy()`, `policy_evaluation()`, `regret()`

Examples

```
data("Maze")
Maze

sol <- solve_MDP(Maze)
visit_probability(sol)

# gw_matrix also can calculate the visit_probability.
gw_matrix(sol, what = "visit_probability")
```

Windy_gridworld *Windy Gridworld MDP Windy Gridworld MDP*

Description

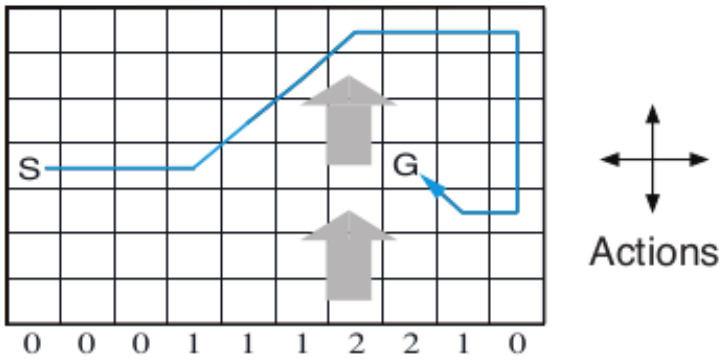
The Windy gridworld MDP example from Chapter 6 of the textbook "Reinforcement Learning: An Introduction."

Format

An object of class MDP.

Details

The gridworld has the following layout:



The grid world is represented as a 7 x 10 matrix of states. In the middle region the next states are shifted upward by wind (the strength in number of squares is given below each column). For example, if the agent is one cell to the right of the goal, then the action left takes the agent to the cell just above the goal.

No discounting is used (i.e., $\gamma = 1$).

References

Richard S. Sutton and Andrew G. Barto (2018). Reinforcement Learning: An Introduction Second Edition, MIT Press, Cambridge, MA.

See Also

Other MDP_examples: Cliff_walking, DynaMaze, MDP(), Maze

Other gridworld: Cliff_walking, DynaMaze, Maze, gridworld

Examples

```
data(Windy_gridworld)
Windy_gridworld

gw_matrix(Windy_gridworld)
gw_matrix(Windy_gridworld, what = "labels")

gw_plot(Windy_gridworld)

# The Goal is an absorbing state
absorbing_states(Windy_gridworld, sparse = "states")

# visualize the transition graph
gw_plot_transition_graph(Windy_gridworld)

# solve using value iteration
sol <- solve_MDP(Windy_gridworld)
```

```
sol
policy(sol)
gw_plot(sol)
```

Index

∗ **MDPSample**
absorbing_states, 3
act, 5
action_state_helpers, 7
MDPSample, 40
reachable_states, 51
sample_MDP.MDPSample, 62
solve_MDP_PG, 82
start, 90
∗ **MDP_examples**
Cliff_walking, 12
DynaMaze, 15
Maze, 33
MDP, 36
Windy_gridworld, 103
∗ **MDP**
absorbing_states, 3
act, 5
action_state_helpers, 7
available_actions, 9
find_reachable_states, 17
MDP, 36
reachable_states, 51
sample_MDP, 60
sample_MDP.MDPSample, 62
start, 90
transition_graph, 94
transition_matrix, 95
unreachable_states, 99
∗ **approximation**
linear_function_approximation, 30
solve_MDP_APPROX, 69
transformation, 92
∗ **datasets**
Cliff_walking, 12
DynaMaze, 15
Maze, 33
Windy_gridworld, 103
∗ **graphs**
policy, 43
∗ **gridworld**
Cliff_walking, 12
DynaMaze, 15
gridworld, 22
Maze, 33
Windy_gridworld, 103
∗ **hplot**
value_function, 100
∗ **policy**
action, 6
expected_return, 16
greedy_action, 21
policy, 43
policy_evaluation, 46
regret, 56
visit_probability, 102
∗ **solver**
convergence_horizon, 14
schedule, 64
solve_MDP, 66
solve_MDP_APPROX, 69
solve_MDP_DP, 73
solve_MDP_LP, 77
solve_MDP_MC, 79
solve_MDP_PG, 82
solve_MDP_SAMP, 84
solve_MDP_TD, 86
∗ **utilities**
colors, 13
round_stochastic, 59
∗ **value_function**
bellman_update, 10
Q_values, 49
value_function, 100
∗ **visualization**
gridworld, 22
transition_graph, 94

A *(*MDP*)*, 36

absorbing_states, 3
absorbing_states(), *6*, *8*, *10*, *18*, *39*, *41*, *52*, *61*, *63*, *83*, *91*, *95*, *98*, *100*
accessors, *39*
accessors *(*transition_matrix*)*, 95
act, 5
act(), *4*, *8*, *10*, *18*, *39*, *41*, *52*, *61*, *63*, *83*, *91*, *95*, *98*, *100*
action, 6
action(), *5*, *17*, *22*, *45*, *48*, *58*, *103*
action_discrepancy *(*regret*)*, 56
action_state_helpers, *4*, *6*, 7, *10*, *18*, *39*, *41*, *52*, *61*, *63*, *83*, *91*, *95*, *98*, *100*
add_policy *(*policy*)*, 43
approx_greedy_action *(*solve_MDP_APPROX*)*, 69
approx_greedy_policy *(*solve_MDP_APPROX*)*, 69
approx_Q_value *(*solve_MDP_APPROX*)*, 69
approx_V_plot *(*solve_MDP_APPROX*)*, 69
approx_value *(*linear_function_approximation*)*, 30
available_actions, 9
available_actions(), *4*, *6*, *8*, *18*, *39*, *44*, *52*, *61*, *63*, *91*, *95*, *98*, *100*

bellman_operator *(*bellman_update*)*, 10
bellman_update, 10
bellman_update(), *50*, *75*, *101*

Cliff_walking, 12, *15*, *27*, *34*, *39*, *104*
cliff_walking *(*Cliff_walking*)*, 12
colors, 13, *59*
colors_continuous *(*colors*)*, 13
colors_discrete *(*colors*)*, 13
convergence_horizon, 14
convergence_horizon(), *65*, *68*, *71*, *76*, *79*, *81*, *83*, *85*, *89*
create_basis_coefs *(*transformation*)*, 92
curve_multiple_directed *(*transition_graph*)*, 94

doParallel::registerDoParallel(), *61*
DynaMaze, *12*, 15, *27*, *34*, *39*, *104*
dynamaze *(*DynaMaze*)*, 15

expected_return, 16
expected_return(), *7*, *22*, *38*, *45*, *48*, *57, 58*, *103*

features2state *(*action_state_helpers*)*, 7
features2state(), *41*
find_reachable_states, 17
find_reachable_states(), *4*, *6*, *8*, *10*, *39*, *52*, *61*, *63*, *91*, *95*, *98*, *100*

get_state_features *(*action_state_helpers*)*, 7
get_state_features(), *70*
graphics::barplot(), *101*
graphics::image(), *70*
grDevices::colorRamp(), *13*
greedy_action, 21
greedy_action(), *7*, *17*, *45*, *48*, *58*, *103*
greedy_policy *(*greedy_action*)*, 21
greedy_policy(), *75*
gridworld, *12*, *15*, 22, *34*, *95*, *104*
gw *(*gridworld*)*, 22
gw_animate *(*gridworld*)*, 22
gw_init *(*gridworld*)*, 22
gw_matrix *(*gridworld*)*, 22
gw_maze_MDP *(*gridworld*)*, 22
gw_path *(*gridworld*)*, 22
gw_plot *(*gridworld*)*, 22
gw_plot_transition_graph *(*gridworld*)*, 22
gw_random_maze *(*gridworld*)*, 22
gw_rc2s *(*gridworld*)*, 22
gw_read_maze *(*gridworld*)*, 22
gw_s2rc *(*gridworld*)*, 22
gw_transition_model *(*gridworld*)*, 22
gw_transition_model_end_state *(*gridworld*)*, 22
gw_transition_model_named *(*gridworld*)*, 22
gw_transition_model_sparse *(*gridworld*)*, 22

induced_reward_matrix *(*policy*)*, 43
induced_transition_matrix *(*policy*)*, 43
is_converged_MDP *(*MDP*)*, 36
is_solved_MDP *(*MDP*)*, 36

linear_function_approximation, 30, *71*, *93*
lpSolve::lp(), *78*

manual_policy *(*policy*)*, 43
Matrix::dgRMatrix, *97*
Matrix::sparseVector, *38*

Maze, *12*, *15*, *27*, 33, *39*, *104*
maze *(*Maze*)*, 33
MDP, *4*, *6*, *9*, *12*, *15*, *16*, *30*, *33*, 36, *41*, *44*, *68*, *92*, *94*, *96*, *97*, *99*, *101–103*
MDP(), *4*, *6*, *8*, *10*, *12*, *15*, *18*, *34*, *52*, *61*, *63*, *91*, *95*, *98*, *100*, *104*
MDPModel, *52*
MDPModel *(*MDP*)*, 36
MDPSample, *8*, 40, *52*, *68*
MDPSample(), *4*, *6*, *8*, *52*, *63*, *83*, *91*

normalize_action *(*action_state_helpers*)*, 7
normalize_action(), *70*
normalize_action_id *(*action_state_helpers*)*, 7
normalize_action_label *(*action_state_helpers*)*, 7
normalize_MDP *(*transition_matrix*)*, 95
normalize_MDP(), *26*
normalize_state *(*action_state_helpers*)*, 7
normalize_state(), *91*
normalize_state_features *(*action_state_helpers*)*, 7
normalize_state_id *(*action_state_helpers*)*, 7
normalize_state_label *(*action_state_helpers*)*, 7

options(), *98*

P_ *(*MDP*)*, 36
pi_approx_linear *(*linear_function_approximation*)*, 30
plot_transition_graph *(*transition_graph*)*, 94
plot_value_function *(*value_function*)*, 100
policy, 43
policy(), *7*, *17*, *22*, *48*, *58*, *103*
policy_evaluation, 46
policy_evaluation(), *7*, *16*, *17*, *22*, *44*, *45*, *57*, *58*, *74*, *103*
policy_evaluation_bellman *(*policy_evaluation*)*, 46
policy_evaluation_LP *(*policy_evaluation*)*, 46
policy_evaluation_MC *(*policy_evaluation*)*, 46

q_approx_linear *(*linear_function_approximation*)*, 30
Q_random *(*Q_values*)*, 49
Q_values, 49
Q_values(), *11*, *101*
Q_zero *(*Q_values*)*, 49

R_ *(*MDP*)*, 36
random_policy *(*policy*)*, 43
reachable_states, 51
reachable_states(), *4*, *6*, *8*, *10*, *18*, *39*, *41*, *61*, *63*, *83*, *91*, *95*, *98*, *100*
regret, 56
regret(), *7*, *17*, *22*, *45*, *48*, *103*
remove_unreachable_states *(*unreachable_states*)*, 99
remove_unreachable_states(), *26*
reward_matrix *(*transition_matrix*)*, 95
round, *59*
round_stochastic, 59
round_stochastic(), *13*

S *(*MDP*)*, 36
s *(*action_state_helpers*)*, 7
sample_MDP, 60
sample_MDP(), *4*, *6*, *8*, *10*, *16*, *18*, *38*, *39*, *47*, *51*, *52*, *63*, *91*, *95*, *98*, *100*
sample_MDP.MDPSample, 62
sample_MDP.MDPSample(), *4*, *6*, *8*, *10*, *18*, *39*, *41*, *52*, *61*, *83*, *91*, *95*, *98*, *100*
schedule, *15*, 64, *68*, *70*, *71*, *76*, *79–86*, *88*, *89*
schedule_exp *(*schedule*)*, 64
schedule_exp2 *(*schedule*)*, 64
schedule_harmonic *(*schedule*)*, 64
schedule_linear *(*schedule*)*, 64
schedule_log *(*schedule*)*, 64
solve_MDP, 66
solve_MDP(), *15*, *25*, *26*, *39*, *65*, *71*, *76*, *79*, *81*, *83*, *85*, *89*
solve_MDP_APPROX, *32*, 69
solve_MDP_APPROX(), *8*, *15*, *32*, *41*, *65*, *68*, *76*, *79*, *81*, *83*, *85*, *89*, *92*, *93*
solve_MDP_DP, 73
solve_MDP_DP(), *15*, *65*, *68*, *71*, *79*, *81*, *83*, *85*, *89*

solve_MDP_LP, 77
solve_MDP_LP(), *15*, *65*, *68*, *71*, *76*, *81*, *83*, *85*, *89*
solve_MDP_MC, 79
solve_MDP_MC(), *15*, *65*, *68*, *71*, *76*, *79*, *83*, *85*, *88, 89*
solve_MDP_PG, 82
solve_MDP_PG(), *4*, *6*, *8*, *15*, *41*, *52*, *63*, *65*, *68*, *71*, *76*, *79*, *81*, *85*, *89*, *91*
solve_MDP_SAMP, 84
solve_MDP_SAMP(), *15*, *65*, *68*, *71*, *76*, *79*, *81*, *83*, *89*
solve_MDP_TD, 86
solve_MDP_TD(), *15*, *65*, *68*, *71*, *76*, *79*, *81*, *83*, *85*
start, *4*, *6*, *8*, *10*, *18*, *39*, *41*, *52*, *61*, *63*, *83*, 90, *95*, *98*, *100*
start_vector *(*transition_matrix*)*, 95
state2features *(*action_state_helpers*)*, 7
state2features(), *41*

transformation, *30–32*, *70, 71*, 92
transformation_fourier_basis *(*transformation*)*, 92
transformation_linear_basis *(*transformation*)*, 92
transformation_polynomial_basis *(*transformation*)*, 92
transformation_RBF_basis *(*transformation*)*, 92
transition_graph, 94
transition_graph(), *4*, *6*, *8*, *10*, *18*, *27*, *39*, *52*, *61*, *63*, *91*, *98*, *100*
transition_matrix, 95
transition_matrix(), *4*, *6*, *8*, *10*, *18*, *39*, *52*, *61*, *63*, *91*, *95*, *100*

unreachable_states, 99
unreachable_states(), *4*, *6*, *8*, *10*, *18*, *39*, *52*, *61*, *63*, *91*, *95*, *98*

v_approx_linear *(*linear_function_approximation*)*, 30
V_random *(*value_function*)*, 100
V_zero *(*value_function*)*, 100
V_zero(), *74*
value_error *(*regret*)*, 56
value_function, 100
value_function(), *11*, *50*
visit_probability, 102
visit_probability(), *7*, *17*, *22*, *45*, *48*, *58*

Windy_gridworld, *12*, *15*, *27*, *34*, *39*, 103
windy_gridworld *(*Windy_gridworld*)*, 103