Package: markovDP (via r-universe)

July 5, 2024

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

Version 0.99.0

Date 2024-xx-xx

Description The package provides the infrastructure to work with 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. It also implements several popular solvers.

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

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

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

Depends R (>= 3.5.0)

Imports stats, methods, Rcpp, Matrix, foreach, igraph, lpSolve

Suggests knitr, rmarkdown, testthat, visNetwork, doParallel, gifski

SystemRequirements C++17

LinkingTo Rcpp

VignetteBuilder knitr

Encoding UTF-8

License GPL (>=3)

Copyright Copyright (C) Michael Hahsler.

RoxygenNote 7.3.1

Roxygen list(markdown = TRUE)

Collate 'AAA_check_installed.R' 'AAA_package.R' 'AAA_shorten.R' 'Cliff_walking.R' 'DynaMaze.R' 'MDP.R' 'Maze.R' 'RcppExports.R' 'Windy_gridworld.R' 'accessors.R' 'accessors_reward.R' 'accessors_transitions.R' 'action.R' 'actions.R' 'add_policy.R' 'check_and_fix_MDP.R' 'colors.R' 'foreach_helper.R' 'gridworld.R' 'policy.R' 'policy_evaluation.R' 'q_values.R' 'reachable_and_absorbing.R' 'regret.R' 'reward.R'

'round_stochchastic.R' 'simulate_MDP.R' 'solve_MDP.R'
'solve_MDP_DP.R' 'solve_MDP_LP.R' 'solve_MDP_TD.R'
'transition_graph.R' 'value_function.R' 'which_max_random.R'

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

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

RemoteRef HEAD

RemoteSha 519b951586e35a2b5c22f39a4b7c4b120fb19609

Contents

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

Description

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

Usage

```
start_vector(x, start = NULL)

normalize_MDP(
  x,
  sparse = TRUE,
  trans_start = FALSE,
  trans_function = TRUE,
  trans_keyword = FALSE
)

reward_matrix(
  x,
  action = NULL,
  start.state = NULL,
  end.state = NULL,
  ...,
  sparse = FALSE
)

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

Arguments

x	A MDP object.
start	a start state description (see MDP). If NULL then the start vector is created using the start stored in the model.
sparse	logical; use sparse matrices when the density is below 50% and keeps data.frame representation for the reward field. NULL returns the representation stored in the problem description which saves the time for conversion.
trans_start	logical; expand the start to a probability vector?
trans_function	logical; convert functions into matrices?
trans_keyword	logical; convert distribution keywords (uniform and identity) in transition_prob matrices?
action	name or index of an action.
start.state, end.state	name or index of the state.
...	further arguments are passed on.

Details

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

Transition Probabilities $T(s'|s,a)$:

`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 density below 50% can be requested in sparse format (as a Matrix::dgCMatrix).

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`). The accessor converts the column vectors automatically into matrices with start states as rows and end states as columns. This conversion can be suppressed by calling `reward_matrix(..., state_matrix = FALSE)` Note that the reward structure cannot be efficiently stored using a standard sparse matrix since there might be a fixed cost for each action resulting in no entries with 0.

Start state:

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

Convert the Complete MDP Description into a consistent form:

`normalize_MDP()` returns a new MDP definition where `transition_prob`, `reward`, and `start` are normalized.

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.

Value

A list or a list of lists of matrices.

Author(s)

Michael Hahsler

See Also

Other MDP: `MDP()`, `actions()`, `add_policy()`, `gridworld`, `policy_evaluation()`, `q_values()`, `reachable_and_absorbing`, `regret()`, `simulate_MDP()`, `solve_MDP()`, `transition_graph()`, `value_function()`

Examples

```
data("Maze")
gridworld_matrix(Maze)

# List of |A| transition matrices. One per action in the from start.states x end.states
Maze$transition_prob
transition_matrix(Maze)
transition_matrix(Maze, action = "up", sparse = TRUE)
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)

# Normalize the whole model using dense representation
Maze_norm <- normalize_MDP(Maze, sparse = FALSE)
Maze_norm$transition_prob
```

---

action *Action Given a Policy*

Description

Returns an action given a policy. If the policy is optimal, then also the action will be optimal.

Usage

```
action(model, ...)

## S3 method for class 'MDP'
action(model, state, epsilon = 0, epoch = 1, ...)
```

Arguments

model	a solved MDP.
...	further parameters are passed on.

state	the state.
epsilon	make the policy epsilon soft.
epoch	what epoch of the policy should be used. Use 1 for converged policies.

Value

The name of the optimal action as a factor.

Author(s)

Michael Hahsler

See Also

Other policy: policy(), policy_evaluation(), q_values(), reward(), value_function()

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)))
```

---

actions *Available Actions in a State*

Description

Determine the set of actions available in a state.

Usage

```
actions(x, state)
```

Arguments

x	a MDP object.
state	a character vector of length one specifying the state.

Details

Unavailable actions are modeled here a actions that have an immediate reward of -Inf in the reward function.

Value

a character vector with the available actions.

a vector with the available actions.

Author(s)

Michael Hahsler

See Also

Other MDP: MDP(), accessors, add_policy(), gridworld, policy_evaluation(), q_values(), reachable_and_absorbing, regret(), simulate_MDP(), solve_MDP(), transition_graph(), value_function()

Examples

```
data(Maze)
gridworld_matrix(Maze)

# The the following actions are always available:
Maze$actions

# available actions in
actions(Maze, state = "s(3,1)")
```

---

add_policy *Add a Policy to a MDP Problem Description*

Description

Add a policy to a MDP problem description allows the user to test policies on modified problem descriptions or to test manually created policies.

Usage

```
add_policy(model, policy)
```

Arguments

model	a MDP model description.
policy	a policy data.frame.

Details

The new policy needs to be a data.frame with one row for each state in the order the states are defined in the model. The only required column is

- `action`: the action prescribed in the state corresponding to the row.

Optional columns are

- `state`: the state names in the order of the states in the model. The needed names can be obtained by from the `$states` element of the model.
- `U`: with the utility given by the value function for the state.

Value

The model description with the added policy.

Author(s)

Michael Hahsler

See Also

Other POMDP: `reachable_and_absorbing`

Other MDP: `MDP()`, `accessors`, `actions()`, `gridworld`, `policy_evaluation()`, `q_values()`, `reachable_and_absorbing`, `regret()`, `simulate_MDP()`, `solve_MDP()`, `transition_graph()`, `value_function()`

Examples

```
data(Maze)

sol <- solve_MDP(Maze)
sol

policy(sol)
reward(sol)

# Add a random policy
random_pol <- random_policy(Maze)
random_pol
sol_random <- add_policy(Maze, random_pol)
policy(sol_random)
reward(sol_random)
```

Cliff_walking *Cliff Walking Gridworld MDP*

Description

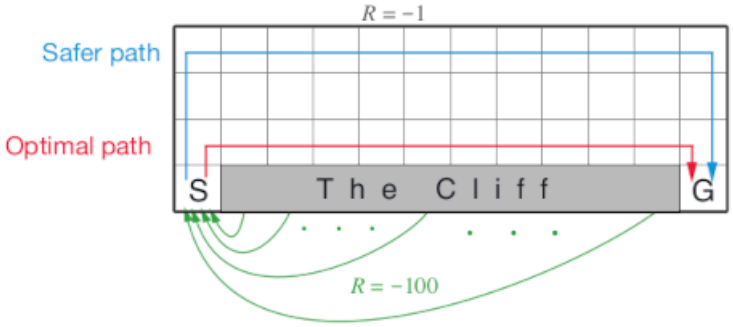
The cliff walking gridworld MDP example from Chapter 6 of the textbook "Reinforcement Learning: An Introduction."

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

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

# The Goal is an absorbing state
which(absorbing_states(Cliff_walking))

# visualize the transition graph
gridworld_plot_transition_graph(Cliff_walking)

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

sol <- solve_MDP(Cliff_walking, method = "q_learning", N = 100)
sol
policy(sol)
gridworld_plot(sol)

sol <- solve_MDP(Cliff_walking, method = "sarsa", N = 100)
sol
policy(sol)
gridworld_plot(sol)

sol <- solve_MDP(Cliff_walking, method = "expected_sarsa", N = 100, alpha = 1)
policy(sol)
gridworld_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.

Examples

```
colors_discrete(5)

colors_continuous(runif(10))
```

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

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

Other gridworld: Cliff_walking, Maze, Windy_gridworld, gridworld

Examples

```
data(DynaMaze)

DynaMaze

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

gridworld_plot_transition_graph(DynaMaze)
```

---

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

```
gridworld_init(
  dim,
  action_labels = c("up", "right", "down", "left"),
  unreachable_states = NULL,
  absorbing_states = NULL,
  labels = NULL
)

gridworld_maze_MDP(
  dim,
  start,
  goal,
  walls = NULL,
  action_labels = c("up", "right", "down", "left"),
  goal_reward = 1,
  step_cost = 0,
  restart = FALSE,
  discount = 0.9,
  horizon = Inf,
  info = NULL,
  name = NA
)

gridworld_s2rc(s)

gridworld_rc2s(rc)
```

```
gridworld_matrix(model, epoch = 1L, what = "states")

gridworld_plot(
  model,
  epoch = 1L,
  actions = "character",
  states = FALSE,
  labels = TRUE,
  impossible_actions = FALSE,
  main = NULL,
  cex = 1,
  offset = 0.5,
  lines = TRUE,
  col = hcl.colors(12, "YlOrRd", rev = TRUE),
  unreachable_col = "black",
  ...
)

gridworld_plot_transition_graph(
  x,
  hide_unreachable_states = TRUE,
  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,
  ...
)

gridworld_animate(x, method, n, zlim = NULL, ...)

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

Arguments

dim	vector of length two with the x and y extent of the gridworld.
action_labels	vector with four action labels that move the agent up, right, down, and left.
unreachable_states	a vector with state labels for unreachable states. These states will be excluded.
absorbing_states	a vector with state labels for absorbing states.
labels	logical; show state labels.
start, goal	labels for the start state and the goal state.
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 gridworld_dim.

name — a string to identify the MDP problem.

s — a state label.

rc — a vector of length two with the row and column coordinate of a state in the gridworld matrix.

model, x — a solved gridworld MDP.

epoch — epoch for unconverged finite-horizon solutions.

what — What should be returned in the matrix. Options are: "states", "labels", "values", "actions", "absorbing", and "reachable".

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.

states — logical; show state names.

impossible_actions — logical; show the value and the action for absorbing or unreachable 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.

col — a colors for the utility values.

unreachable_col — a color used for unreachable states. Use NA for no color.

... — further arguments are passed on to igraph::plot.igraph().

hide_unreachable_states — logical; do not show unreachable states.

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 — a MDP solution method for solve_MDP().

n — number of iterations to animate.

zlim — limits for visualizing the state value.

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 actions are `"up"`, `"right"`, `"down"`, and `"left"`.

`gridworld_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. Unreachable states (walls) and absorbing state can be defined. This information can be used to build a custom gridworld MDP.

Several helper functions are provided to use states, look at the state layout, and plot policies on the gridworld.

`gridworld_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.

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

See Also

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

Other MDP: `MDP()`, `accessors`, `actions()`, `add_policy()`, `policy_evaluation()`, `q_values()`, `reachable_and_absorbing`, `regret()`, `simulate_MDP()`, `solve_MDP()`, `transition_graph()`, `value_function()`

Examples

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

gw$states
gw$actions
gw$info

# display the state labels in the gridworld
gridworld_matrix(gw)
gridworld_matrix(gw, what = "label")
gridworld_matrix(gw, what = "reachable")
gridworld_matrix(gw, what = "absorbing")

# a transition function for regular moves in the gridworld is provided
gw$transition_prob("right", "s(1,1)", "s(1,2)")
gw$transition_prob("right", "s(2,1)", "s(2,2)") ### we cannot move into an unreachable state
gw$transition_prob("right", "s(2,1)", "s(2,1)") ### but the agent stays in place

# convert between state names and row/column indices
```

```
gridworld_s2rc("s(1,1)")
gridworld_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(action = NA, start.state = NA, end.state = NA) {
  # ignore the action next to 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)
    } else {
      return(gw$transition_prob(action, start.state, end.state) * .5)
    }
  }

  # use the standard gridworld movement
  gw$transition_prob(action, start.state, end.state)
}

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

black_hole

gridworld_plot_transition_graph(black_hole)

# solve the problem
sol <- solve_MDP(black_hole)
gridworld_matrix(sol, what = "values")
gridworld_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 <- gridworld_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

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

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

sol <- solve_MDP(DynaMaze)

gridworld_matrix(sol, what = "values")
gridworld_matrix(sol, what = "actions")
gridworld_plot(sol)
gridworld_plot(sol, states = TRUE)

# visualize the first 3 iterations of value iteration
gridworld_animate(DynaMaze, method = "value", n = 3)

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

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

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

maze <- gridworld_read_maze(file.path(maze_dir, "small_maze.txt"))
maze
gridworld_plot(maze)
sol <- solve_MDP(maze, method = "lp", discount = 0.999)
sol

gridworld_plot(sol)
```

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:
gw <- gridworld_init(dim = c(3, 4), unreachable_states = c("s(2,2)"))
gridworld_matrix(gw)

# the transition function is stochastic so we cannot use the standard
# gridworld gw$transition_prob() function and have to replace it
T <- function(action, start.state, end.state) {
  actions <- c("up", "right", "down", "left")
  states <- c(
    "s(1,1)", "s(2,1)", "s(3,1)", "s(1,2)", "s(3,2)", "s(1,3)",
    "s(2,3)", "s(3,3)", "s(1,4)", "s(2,4)", "s(3,4)"
  )
```

```
  action <- match.arg(action, choices = actions)

  # absorbing states
  if (start.state %in% c("s(1,4)", "s(2,4)")) {
    if (start.state == end.state) {
      return(1)
    } else {
      return(0)
    }
  }

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

  rc <- gridworld_s2rc(start.state)
  delta <- list(
    up = c(-1, 0),
    down = c(+1, 0),
    right = c(0, +1),
    left = c(0, -1)
  )
  P <- matrix(0, nrow = 3, ncol = 4)

  add_prob <- function(P, rc, a, value) {
    new_rc <- rc + delta[[a]]
    if (!(gridworld_rc2s(new_rc) %in% states)) {
      new_rc <- rc
    }
    P[new_rc[1], new_rc[2]] <- P[new_rc[1], new_rc[2]] + value
    P
  }

  P <- add_prob(P, rc, action, .8)
  P <- add_prob(P, rc, error_direction[1], .1)
  P <- add_prob(P, rc, error_direction[2], .1)
  P[rbind(gridworld_s2rc(end.state))]
}

T("up", "s(3,1)", "s(2,1)")

R <- rbind(
  R_(end.state = NA, value = -0.04),
  R_(end.state = "s(2,4)", value = -1),
  R_(end.state = "s(1,4)", value = +1),
  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_prob = T,
  reward = R,
  info = list(
    gridworld_dim = c(3, 4),
    gridworld_labels = list(
      "s(3,1)" = "Start",
      "s(2,4)" = "-1",
      "s(1,4)" = "Goal: +1"
    )
  )
)

Maze

str(Maze)

gridworld_matrix(Maze)
gridworld_matrix(Maze, what = "labels")
gridworld_plot(Maze)

# find absorbing (terminal) states
which(absorbing_states(Maze))

maze_solved <- solve_MDP(Maze)
policy(maze_solved)

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

gridworld_plot(maze_solved)
```

MDP *Define an MDP Problem*

Description

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

Usage

```
MDP(
  states,
  actions,
  transition_prob,
```

```
  reward,
  discount = 0.9,
  horizon = Inf,
  start = "uniform",
  info = NULL,
  name = NA,
  normalize = TRUE
)

is_solved_MDP(x, stop = FALSE)

epoch_to_episode(x, epoch)

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

R_(action = NA, start.state = NA, end.state = NA, observation = 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_prob	Specifies the transition probabilities between states.
reward	Specifies the rewards dependent on action and states.
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.
normalize	logical; normalize representation (see normalize_MDP()).
x	a MDP object.
stop	logical; stop with an error.
epoch	integer; an epoch that should be converted to the corresponding episode in a time-dependent MDP.
action	action as a 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 T_() and R_().
observation	unused for MDPs. Must be NA.

Details

Markov decision processes (MDPs) are discrete-time stochastic control process. We implement here MDPs with a finite state space. `MDP()` defines all the element of a MDP problem including the discount rate, the set of states, the set of actions,the transition probabilities, the observation probabilities, and the rewards.

In the following we use the following notation. The MDP is a 5-duple:

(S, A, T, R, γ).

S is the set of states; A is the set of actions; T are the conditional transition probabilities between states; R is the reward function; Ω is the set of observations; 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 probabilities: $T(s'|s, a)$

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 `T_()`. You can use `rbind()` with helper function `T_()` 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 same arguments are `T_()`, but no default values that returns the transition probability.

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 list of state x state matrices. The list elements are for `'action'`. The matrix rows are `start.state` and the columns are `end.state`.
- A function with the same arguments are `R_()`, but no default values that returns the reward.

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.

Note: The code also includes in R_() an argument called observation. Observations are only used POMDPs implemented in package pomdp abs must always be NA for MDPs.

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 reward is calculated by reward() and for simulations with simulate_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.

The default state state is a uniform distribution over all states.

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: accessors, actions(), add_policy(), gridworld, policy_evaluation(), q_values(), reachable_and_absorbing, regret(), simulate_MDP(), solve_MDP(), transition_graph(), value_function()

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

transition_matrix(car)
reward_matrix(car, sparse = TRUE)
reward_matrix(car)

sol <- solve_MDP(car)
policy(sol)
```

`policy` *Extract or Create a Policy*

Description

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

Usage

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

random_policy(x, prob = NULL)

manual_policy(x, actions)
```

Arguments

`x`	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.
`prob`	probability vector for random actions for `random_policy()`. a logical indicating if action probabilities should be returned for `greedy_action()`.
`actions`	a vector with the action (either the action label or the numeric id) for each state.

Details

For an MDP, the deterministic policy is a data.frame with columns for:

- `state`: The state.
- `U`: The state's value (discounted expected utility U) 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.

Value

A data.frame containing the policy. If `drop = FALSE` then the policy is returned as a list with the policy for each epoch.

Author(s)

Michael Hahsler

See Also

Other policy: `action()`, `policy_evaluation()`, `q_values()`, `reward()`, `value_function()`

Examples

```
data("Maze")

sol <- solve_MDP(Maze)
sol

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

## create a random policy
pi_random <- random_policy(Maze)
pi_random

gridworld_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"
pi_manual <- manual_policy(Maze, acts)
pi_manual

gridworld_plot(add_policy(Maze, pi_manual))

## Finite horizon (we use incremental pruning because grid does not converge)
```

```
sol <- solve_MDP(model = Maze, horizon = 3)
sol

policy(sol)
gridworld_plot(sol)
```

`policy_evaluation` *Policy Evaluation*

Description

Evaluate a policy for a model by repeatedly applying the Bellman operator.

Usage

```
policy_evaluation(
  model,
  pi,
  U = NULL,
  k_backups = 1000,
  theta = 0.001,
  verbose = FALSE
)

bellman_operator(model, pi, U)
```

Arguments

`model`	an MDP problem specification.
`pi`	a policy as a data.frame with at least columns for states and action.
`U`	a vector with value function representing the state utilities (expected sum of discounted rewards from that point on). If `model` is a solved model, then the state utilities 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 change in a state value is less than θ.
`verbose`	logical; should progress and approximation errors be printed.

Details

The Bellman operator updates a value function given the model defining T, γ and R, and a policy π by applying the Bellman equation as an update rule for each state:

$$U_{k+1}(s) = \sum_a \pi_{a|s} \sum_{s'} T(s'|s,a)[R(s,a) + \gamma U_k(s')]$$

A policy can be evaluated by applying the Bellman update till convergence. In each iteration, all states are updated. In this implementation updating is stopped afterk_backups iterations or after the largest update

$$||U_{k+1} - U_k||_\infty < \theta.$$

Value

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

Author(s)

Michael Hahsler

References

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

See Also

Other MDP: MDP(), accessors, actions(), add_policy(), gridworld, q_values(), reachable_and_absorbing, regret(), simulate_MDP(), solve_MDP(), transition_graph(), value_function()

Other policy: action(), policy(), q_values(), reward(), value_function()

Examples

```
data(Maze)
Maze

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

# 4. an improved policy based on one policy evaluation and
#   policy improvement step.
u <- policy_evaluation(Maze, pi_random)
```

```
q <- q_values(Maze, U = u)
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)
)
```

`q_values` *Q-Value Functions*

Description

Implementation several functions useful to deal with Q-values for MDPs.

Usage

```
q_values(model, U = NULL)

greedy_action(Q, s, epsilon = 0, prob = FALSE)

greedy_policy(Q)
```

Arguments

`model`	an MDP problem specification.
`U`	a vector with value function representing the state utilities (expected sum of discounted rewards from that point on). If `model` is a solved model, then the state utilities are taken from the solution.
`Q`	an action value function with Q-values as a state by action matrix.
`s`	a state.
`epsilon`	an `epsilon` > `0` applies an epsilon-greedy policy.
`prob`	logical; return a probability distribution over the actions.

Details

Implemented functions are:

- `q_values()` calculates (approximates) Q-values for a given model and value function using the Bellman optimality equation:

$$q(s,a) = \sum_{s'} T(s'|s,a)[R(s,a) + \gamma U(s')]$$

Q-values are calculated if $U = U^*$, the optimal value function otherwise we get an approximation. Q-values can be used as the input for several other functions.

- `greedy_action()` returns the action with the largest Q-value given a state.
- `greedy_policy()` generates a greedy policy using Q-values.

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.

`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 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 MDP: `MDP()`, `accessors`, `actions()`, `add_policy()`, `gridworld`, `policy_evaluation()`, `reachable_and_absorbing`, `regret()`, `simulate_MDP()`, `solve_MDP()`, `transition_graph()`, `value_function()`

Other policy: `action()`, `policy()`, `policy_evaluation()`, `reward()`, `value_function()`

Examples

```
data(Maze)
Maze

# create a random policy and calculate q-values
pi_random <- random_policy(Maze)
u <- policy_evaluation(Maze, pi_random)
q <- q_values(Maze, U = u)

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

greedy_action(q, "s(3,1)", epsilon = 0, prob = FALSE)
greedy_action(q, "s(3,1)", epsilon = 0, prob = TRUE)
greedy_action(q, "s(3,1)", epsilon = .1, prob = TRUE)
```

`reachable_and_absorbing` *Reachable and Absorbing States*

Description

Find reachable and absorbing states in the transition model.

Usage

```
reachable_states(x, states = NULL, ...)

absorbing_states(x, states = NULL, ...)

remove_unreachable_states(x)
```

Arguments

`x`	a MDP object.
`states`	a character vector specifying the names of the states to be checked. `NULL` checks all states.
`...`	further arguments are passed on.

Details

The function `reachable_states()` checks if states are reachable using the transition model and the start probabilities.

The function `absorbing_states()` checks if a state or a set of states are absorbing (terminal states) with a zero reward (or `-Inf` for unavailable actions). If no states are specified (`states = NULL`), then all model states are checked. This information can be used in simulations to end an episode.

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

Value

`reachable_states()` returns a logical vector indicating if the states are reachable.

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

the model with all unreachable states removed

Author(s)

Michael Hahsler

See Also

Other MDP: `MDP()`, `accessors`, `actions()`, `add_policy()`, `gridworld`, `policy_evaluation()`, `q_values()`, `regret()`, `simulate_MDP()`, `solve_MDP()`, `transition_graph()`, `value_function()`

Other POMDP: `add_policy()`

Examples

```
data(Maze)

gridworld_matrix(Maze)
gridworld_matrix(Maze, what = "labels")

# -1 and +1 are absorbing states
absorbing_states(Maze)
which(absorbing_states(Maze))

# all states in the model are reachable
reachable_states(Maze)
which(!reachable_states(Maze))
```

---

regret *Calculate the Regret of a Policy*

Description

Calculates the regret of a policy relative to a benchmark policy.

Usage

```
regret(policy, benchmark, start = NULL)
```

Arguments

policy	a solved POMDP containing the policy to calculate the regret for.
benchmark	a solved POMDP 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.

Details

Regret is defined as $V^{\pi^*}(s_0) - V^{\pi}(s_0)$ with V^{π} representing the expected long-term state value (represented by the value function) given the policy π and the start state 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.

Value

the regret as a difference of expected long-term rewards.

Author(s)

Michael Hahsler

See Also

Other MDP: MDP(), accessors, actions(), add_policy(), gridworld, policy_evaluation(), q_values(), reachable_and_absorbing, simulate_MDP(), solve_MDP(), transition_graph(), value_function()

Examples

```
data(Maze)

sol_optimal <- solve_MDP(Maze)
policy(sol_optimal)

# 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))
policy(sol_manual)

regret(sol_manual, benchmark = sol_optimal)
```

---

reward *Calculate the Expected Reward of a Policy*

Description

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

Usage

```
reward(x, ...)

## S3 method for class 'MDP'
reward(x, start = NULL, epoch = 1L, ...)
```

Arguments

x	a solved MDP object.
...	further arguments are passed on.
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.
epoch	epoch for a finite-horizon solutions.

Details

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

Value

`reward()` returns a vector of reward values, one for each belief if a matrix is specified.

`state`	start state to calculate the reward for. if `NULL` then the start state of model is used.

Author(s)

Michael Hahsler

See Also

Other policy: `action()`, `policy()`, `policy_evaluation()`, `q_values()`, `value_function()`

Examples

```
data("Maze")
Maze
gridworld_matrix(Maze)

sol <- solve_MDP(Maze)
policy(sol)

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

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

# expected reward when we start from a random state
reward(sol, start = "uniform")
```

---

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

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)
```

---

simulate_MDP *Simulate Trajectories in a MDP*

Description

Simulate trajectories through a MDP. The start state for each trajectory is randomly chosen using the specified belief. The belief is used to choose actions from an epsilon-greedy policy and then update the state.

Usage

```
simulate_MDP(
  model,
  n = 100,
  start = NULL,
  horizon = NULL,
```

```
  epsilon = NULL,
  delta_horizon = 0.001,
  return_trajectories = FALSE,
  engine = "cpp",
  verbose = FALSE,
  ...
)
```

Arguments

model	a MDP model.
n	number of trajectories.
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 random actions for using an epsilon-greedy policy. Default for solved models is 0 and for unsolved model 1.
delta_horizon	precision used to determine the horizon for infinite-horizon problems.
return_trajectories	logical; return the complete trajectories.
engine	'cpp' or 'r' to perform simulation using a faster C++ or a native R implementation.
verbose	report used parameters.
...	further arguments are ignored.

Details

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

Both implementations support parallel execution using the package **foreach**. To enable parallel execution, a parallel backend like **doparallel** needs to be available needs to be registered (see doParallel::registerDoParallel()). Note that small simulations 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_reward: The average discounted reward.
- 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 return_trajectories = TRUE.

Author(s)

Michael Hahsler

See Also

Other MDP: MDP(), accessors, actions(), add_policy(), gridworld, policy_evaluation(), q_values(), reachable_and_absorbing, regret(), solve_MDP(), transition_graph(), value_function()

Examples

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

data(Maze)

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

# U in the policy is and estimate of the utility of being in a state when using the optimal policy.
policy(sol)
gridworld_matrix(sol, what = "action")

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

# Note that all simulations start at s_1 and that the simulated avg. reward
# is therefore an estimate to the U value 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 starting following a uniform distribution over all
#             states and return all trajectories
sim <- simulate_MDP(sol,
  n = 100, start = "uniform", horizon = 10,
  return_trajectories = TRUE
)
head(sim$trajectories)

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

`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, method = "value_iteration", ...)

solve_MDP_DP(
  model,
  method = "value_iteration",
  horizon = NULL,
  discount = NULL,
  N_max = 1000,
  error = 0.01,
  k_backups = 10,
  U = NULL,
  verbose = FALSE
)

solve_MDP_LP(
  model,
  method = "lp",
  horizon = NULL,
  discount = NULL,
  verbose = FALSE,
  ...
)

solve_MDP_TD(
  model,
  method = "q_learning",
  horizon = NULL,
  discount = NULL,
  alpha = 0.5,
  epsilon = 0.1,
  N = 100,
  U = NULL,
  verbose = FALSE
)
```

Arguments

`model`	an MDP problem specification.

method	string; one of the following solution methods: 'value_iteration', 'policy_iteration', 'lp', 'q_learning', 'sarsa', or 'expected_sarsa'.
...	further parameters are passed on to the solver function.
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_max	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 error allowed in the utility of any state (i.e., the maximum policy loss) used as the termination criterion.
k_backups	policy iteration: number of look ahead steps used for approximate policy evaluation used by the policy iteration method.
U	a vector with initial utilities used for each state. If NULL, then the default of a vector of all 0s is used.
verbose	logical, if set to TRUE, the function provides the output of the solver in the R console.
alpha	step size in (0, 1].
epsilon	used for ϵ-greedy policies.
N	number of episodes used for learning.

Details

Several solvers are available.

Dynamic Programming:

Implemented are the following dynamic programming methods (following 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 (estimate the value function for the current policy using k_backups and function policy_evaluation(), and
  2. policy improvement (calculate a greedy policy given the value function). The algorithm stops when it converges to a stable policy (i.e., no changes between two iterations).
- **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 equation. The iterations are terminated either after N_max iterations or when the solution converges. Approximate convergence is achieved for discounted problems (with $\gamma < 1$) when the maximal value function change for any state δ is $\delta \leq error(1-\gamma)/\gamma$. 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 $\delta \leq error$.
  The greedy policy is calculated from the final value function. Value iteration can be seen as policy iteration with truncated policy evaluation.

- **Prioritized Sweeping** (Moore and Atkeson, 1993) approximate the optimal value function by iteratively adjusting always only the state value of the state with the largest Bellman error. This leads to faster convergence compared to value iteration which always updates the value function for all states. This implementation stops iteration when the sum of the priority values for all states is less than the specified `error`.

Note that the policy converges earlier than the value function.

Linear Programming:

The following linear programming formulation (Manne 1960) is implemented. For the optimal value function, the Bellman equation holds:

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \; \forall a \in A, s \in S$$

We can find the optimal value function by solving the following linear program:

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

subject to

$$V(s) \geq \sum_{s' \in S} T(s,a,s')[R(s,a,s') + \gamma V(s')], \; \forall a \in A, s \in S$$

Note:

- The discounting factor has to be strictly less than 1.
- Additional parameters to to `solve_MDP` are passed on to `lpSolve::lp()`.
- We use the solver in `lpSolve::lp()` which requires all decision variables (state values) to be non-negative. To ensure this, for negative rewards, all rewards as shifted so the smallest reward is 0. This does not change the optimal policy.

Temporal Difference Control:

Implemented are the following temporal difference control methods described in Sutton and Barto (2020). Note that the MDP transition and reward models are only used to simulate the environment for these reinforcement learning methods. The algorithms use a step size parameter α (learning rate) for the updates and the exploration parameter ϵ for the ϵ-greedy policy.

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 10,000 actions with a warning. For models without absorbing states, a episode length has to be specified via `horizon`.

- **Q-Learning** (Watkins and Dayan 1992) is an off-policy temporal difference method that uses an ϵ-greedy behavior policy and learns a greedy target policy.
- **Sarsa** (Rummery and Niranjan 1994) is an on-policy method that follows and learns an ϵ-greedy policy. The final ϵ-greedy policy is converted into a greedy policy.
- **Expected Sarsa** (R. S. Sutton and Barto 2018). We implement an on-policy version that uses the expected value under the current policy for the update. It moves deterministically in the same direction as Sarsa moves in expectation. Because it uses the expectation, we can set the step size α to large values and even 1.

Value

`solve_MDP()` returns an object of class POMDP which is a list with the model specifications (`model`), the solution (`solution`). The solution is a list with the elements:

- `policy` a list representing the policy graph. The list only has one element for converged solutions.
- `converged` did the algorithm converge (`NA`) for finite-horizon problems.
- `delta` final δ (value iteration and infinite-horizon only)
- `iterations` number of iterations to convergence (infinite-horizon only)

Author(s)

Michael Hahsler

References

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.

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

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.

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

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

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 MDP: `MDP()`, `accessors`, `actions()`, `add_policy()`, `gridworld`, `policy_evaluation()`, `q_values()`, `reachable_and_absorbing`, `regret()`, `simulate_MDP()`, `transition_graph()`, `value_function()`

Examples

```
data(Maze)
Maze

# use value iteration
maze_solved <- solve_MDP(Maze, method = "value_iteration")
maze_solved
policy(maze_solved)

# plot the value function U
plot_value_function(maze_solved)

# Gridworld solutions can be visualized
gridworld_plot(maze_solved)

# Use linear programming
maze_solved <- solve_MDP(Maze, method = "lp")
maze_solved
policy(maze_solved)

# use modified policy iteration
maze_solved <- solve_MDP(Maze, method = "policy_iteration")
policy(maze_solved)

# finite horizon
maze_solved <- solve_MDP(Maze, method = "value_iteration", horizon = 3)
policy(maze_solved)
gridworld_plot(maze_solved, epoch = 1)
gridworld_plot(maze_solved, epoch = 2)
gridworld_plot(maze_solved, epoch = 3)

# create a random policy where action n is very likely and approximate
#  the value function. We change the discount factor to .9 for this.
Maze_discounted <- Maze
Maze_discounted$discount <- .9
pi <- random_policy(Maze_discounted,
  prob = c(n = .7, e = .1, s = .1, w = 0.1)
)
pi

# compare the utility function for the random policy with the function for the optimal
#  policy found by the solver.
maze_solved <- solve_MDP(Maze)

policy_evaluation(Maze, pi, k_backup = 100)
policy_evaluation(Maze, policy(maze_solved), k_backup = 100)

# Note that the solver already calculates the utility function and returns it with the policy
policy(maze_solved)

# Learn a Policy using Q-Learning
maze_learned <- solve_MDP(Maze, method = "q_learning", N = 100)
```

```
maze_learned

maze_learned$solution
policy(maze_learned)
plot_value_function(maze_learned)
gridworld_plot(maze_learned)
```

`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 a POMDP is a Markov Chain. This function extracts the transition model as an igraph object.

Value

returns the transition model as an igraph object.

See Also

Other MDP: `MDP()`, `accessors`, `actions()`, `add_policy()`, `gridworld`, `policy_evaluation()`, `q_values()`, `reachable_and_absorbing`, `regret()`, `simulate_MDP()`, `solve_MDP()`, `value_function()`

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")
}
```

value_function *Value Function*

Description

Extracts the value function from a solved MDP.

Usage

```
value_function(model, drop = TRUE)

plot_value_function(
  model,
  epoch = 1,
  legend = TRUE,
  col = NULL,
  ylab = "Value",
  las = 3,
  main = NULL,
  ...
)
```

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

the function as a numeric vector with one value for each state.

Author(s)

Michael Hahsler

See Also

Other policy: action(), policy(), policy_evaluation(), q_values(), reward()

Other MDP: MDP(), accessors, actions(), add_policy(), gridworld, policy_evaluation(), q_values(), reachable_and_absorbing, regret(), simulate_MDP(), solve_MDP(), transition_graph()

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
gridworld_plot(sol, epoch = 1)
gridworld_plot(sol, epoch = 2)
gridworld_plot(sol, epoch = 3)
```

Windy_gridworld *Windy Gridworld MDP Windy Gridworld MDP*

Description

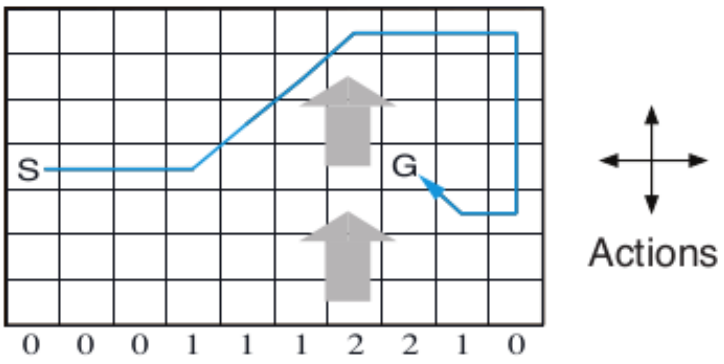
The Windy gridworld MDP example from Chapter 6 of the textbook "Reinforcement Learning: An Introduction."

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

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

gridworld_plot(Windy_gridworld)

# The Goal is an absorbing state
which(absorbing_states(Windy_gridworld))

# visualize the transition graph
gridworld_plot_transition_graph(Windy_gridworld)

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

Index