*"""Markov Decision*

*First we define an MDP, in which*

*states are laid out in a 2-dimensional grid. We also represent a policy*

*as a dictionary of {state:action} pairs, and a Utility function as a*

*dictionary of {state:number} pairs. We then define the value\_iteration*

*and policy\_iteration algorithms."""*

from [utils](http://aima.cs.berkeley.edu/python/utils.html) import \*

**class** **MDP**:

*"""A Markov Decision Process, defined by an initial state, transition model,*

*and reward function. We also keep track of a gamma value, for use by*

*algorithms. The transition model is represented somewhat differently from*

*the text. Instead of T(s, a, s') being probability number for each*

*state/action/state triplet, we instead have T(s, a) return a list of (p, s')*

*pairs. We also keep track of the possible states, terminal states, and*

*actions for each state. """*

**def** **\_\_init\_\_**(self, init, actlist, terminals, gamma=.9):

update(self, init=init, actlist=actlist, terminals=terminals,

gamma=gamma, states=set(), reward={})

**def** **R**(self, state):

*"Return a numeric reward for this state."*

return self.reward[state]

**def** **T**(state, action):

*"""Transition model. From a state and an action, return a list*

*of (result-state, probability) pairs."""*

abstract

**def** **actions**(self, state):

*"""Set of actions that can be performed in this state. By default, a*

*fixed list of actions, except for terminal states. Override this*

*method if you need to specialize by state."""*

if state in self.terminals:

return [None]

else:

return self.actlist

**class** **GridMDP**(MDP):

*"""A two-dimensional grid MDP, as in. All you have to do is*

*specify the grid as a list of lists of rewards; use None for an obstacle*

*(unreachable state). Also, you should specify the terminal states.*

*An action is an (x, y) unit vector; e.g. (1, 0) means move east."""*

**def \_\_init\_\_**(self, grid, terminals, init=(0, 0), gamma=.9):

grid.reverse() ## because we want row 0 on bottom, not on top

MDP.\_\_init\_\_(self, init, actlist=orientations,

terminals=terminals, gamma=gamma)

update(self, grid=grid, rows=len(grid), cols=len(grid[0]))

for x in range(self.cols):

for y in range(self.rows):

self.reward[x, y] = grid[y][x]

if grid[y][x] is not None:

self.states.add((x, y))

**def** **T**(self, state, action):

if action == None:

return [(0.0, state)]

else:

return [(0.8, self.go(state, action)),

(0.1, self.go(state, turn\_right(action))),

(0.1, self.go(state, turn\_left(action)))]

**def** **go**(self, state, direction):

*"Return the state that results from going in this direction."*

state1 = vector\_add(state, direction)

return if\_(state1 in self.states, state1, state)

**def** **to\_grid**(self, mapping):

*"""Convert a mapping from (x, y) to v into a [[..., v, ...]] grid."""*

return list(reversed([[mapping.get((x,y), None)

for x in range(self.cols)]

for y in range(self.rows)]))

**def** **to\_arrows**(self, policy):

chars = {(1, 0):*'>'*, (0, 1):*'^'*, (-1, 0):*'<'*, (0, -1):*'v'*, None: *'.'*}

return self.to\_grid(dict([(s, chars[a]) for (s, a) in policy.items()]))

Fig[17,1] = GridMDP([[-0.04, -0.04, -0.04, +1],

[-0.04, None, -0.04, -1],

[-0.04, -0.04, -0.04, -0.04]],

terminals=[(3, 2), (3, 1)])

**def** **value\_iteration**(mdp, epsilon=0.001):

*"Solving an MDP by value iteration. "*

U1 = dict([(s, 0) for s in mdp.states])

R, T, gamma = mdp.R, mdp.T, mdp.gamma

while True:

U = U1.copy()

delta = 0

for s in mdp.states:

U1[s] = R(s) + gamma \* max([sum([p \* U[s1] for (p, s1) in T(s, a)])

for a in mdp.actions(s)])

delta = max(delta, abs(U1[s] - U[s]))

if delta < epsilon \* (1 - gamma) / gamma:

return U

**def** **best\_policy**(mdp, U):

*"""Given an MDP and a utility function U, determine the best policy,*

*as a mapping from state to action."""*

pi = {}

for s in mdp.states:

pi[s] = argmax(mdp.actions(s), lambda a:expected\_utility(a, s, U, mdp))

return pi

**def** **expected\_utility**(a, s, U, mdp):

*"The expected utility of doing a in state s, according to the MDP and U."*

return sum([p \* U[s1] for (p, s1) in mdp.T(s, a)])

**def** **policy\_iteration**(mdp):

*"Solve an MDP by policy iteration "*

U = dict([(s, 0) for s in mdp.states])

pi = dict([(s, random.choice(mdp.actions(s))) for s in mdp.states])

while True:

U = policy\_evaluation(pi, U, mdp)

unchanged = True

for s in mdp.states:

a = argmax(mdp.actions(s), lambda a: expected\_utility(a,s,U,mdp))

if a != pi[s]:

pi[s] = a

unchanged = False

if unchanged:

return pi

**def** **policy\_evaluation**(pi, U, mdp, k=20):

*"""Return an updated utility mapping U from each state in the MDP to its*

*utility, using an approximation (modified policy iteration)."""*

R, T, gamma = mdp.R, mdp.T, mdp.gamma

for i in range(k):

for s in mdp.states:

U[s] = R(s) + gamma \* sum([p \* U[s] for (p, s1) in T(s, pi[s])])

return U