gridworld.py (original)

```
# gridworld.py
# ------------
# Licensing Information: Please do not distribute or publish solutions to this
# project. You are free to use and extend these projects for educational
# purposes. The Pacman AI projects were developed at UC Berkeley, primarily by
# John DeNero (denero@cs.berkeley.edu) and Dan Klein (klein@cs.berkeley.edu).
# For more info, see http://inst.eecs.berkeley.edu/~cs188/sp09/pacman.html

import random
import sys
import mdp
import environment
import util
import optparse

class Gridworld(mdp.MarkovDecisionProcess):
  """
    Gridworld
  """
  def __init__(self, grid):
    # layout
    if type(grid) == type([]): grid = makeGrid(grid)
    self.grid = grid

    # parameters
    self.livingReward = 0.0
    self.noise = 0.2

  def setLivingReward(self, reward):
    """
    The (negative) reward for exiting "normal" states.

    Note that in the R+N text, this reward is on entering
    a state and therefore is not clearly part of the state's
    future rewards.
    """
    self.livingReward = reward

  def setNoise(self, noise):
    """
    The probability of moving in an unintended direction.
    """
    self.noise = noise


  def getPossibleActions(self, state):
    """
    Returns list of valid actions for 'state'.

    Note that you can request moves into walls and
    that "exit" states transition to the terminal
    state under the special action "done".
    """
    if state == self.grid.terminalState:
      return ()
    x,y = state
    if type(self.grid[x][y]) == int:
      return ('exit',)
    return ('north','west','south','east')

  def getStates(self):
    """
    Return list of all states.
    """
```

```
    # The true terminal state.
    states = [self.grid.terminalState]
    for x in range(self.grid.width):
      for y in range(self.grid.height):
        if self.grid[x][y] != '#':
          state = (x,y)
          states.append(state)
    return states

  def getReward(self, state, action, nextState):
    """
    Get reward for state, action, nextState transition.

    Note that the reward depends only on the state being
    departed (as in the R+N book examples, which more or
    less use this convention).
    """
    if state == self.grid.terminalState:
      return 0.0
    x, y = state
    cell = self.grid[x][y]
    if type(cell) == int or type(cell) == float:
      return cell
    return self.livingReward

  def getStartState(self):
    for x in range(self.grid.width):
      for y in range(self.grid.height):
        if self.grid[x][y] == 'S':
          return (x, y)
    raise 'Grid has no start state'

  def isTerminal(self, state):
    """
    Only the TERMINAL_STATE state is *actually* a terminal state.
    The other "exit" states are technically non-terminals with
    a single action "exit" which leads to the true terminal state.
    This convention is to make the grids line up with the examples
    in the R+N textbook
    """
    return state == self.grid.terminalState


  def getTransitionStatesAndProbs(self, state, action):
    """
    Returns list of (nextState, prob) pairs
    representing the states reachable
    from 'state' by taking 'action' along
    with their transition probabilities.
    """

    if action not in self.getPossibleActions(state):
      raise "Illegal action!"

    if self.isTerminal(state):
      return []

    x, y = state

    if type(self.grid[x][y]) == int or type(self.grid[x][y]) == float:
      termState = self.grid.terminalState
      return [(termState, 1.0)]

    successors = []

    northState = (self.__isAllowed(y+1,x) and (x,y+1)) or state
    westState = (self.__isAllowed(y,x-1) and (x-1,y)) or state
    southState = (self.__isAllowed(y-1,x) and (x,y-1)) or state
    eastState = (self.__isAllowed(y,x+1) and (x+1,y)) or state
```

```
    if action == 'north' or action == 'south':
      if action == 'north':
        successors.append((northState,1-self.noise))
      else:
        successors.append((southState,1-self.noise))

      massLeft = self.noise
      successors.append((westState,massLeft/2.0))
      successors.append((eastState,massLeft/2.0))

    if action == 'west' or action == 'east':
      if action == 'west':
        successors.append((westState,1-self.noise))
      else:
        successors.append((eastState,1-self.noise))

      massLeft = self.noise
      successors.append((northState,massLeft/2.0))
      successors.append((southState,massLeft/2.0))

    successors = self.__aggregate(successors)

    return successors

  def __aggregate(self, statesAndProbs):
    counter = util.Counter()
    for state, prob in statesAndProbs:
      counter[state] += prob
    newStatesAndProbs = []
    for state, prob in counter.items():
      newStatesAndProbs.append((state, prob))
    return newStatesAndProbs

  def __isAllowed(self, y, x):
    if y < 0 or y >= self.grid.height: return False
    if x < 0 or x >= self.grid.width: return False
    return self.grid[x][y] != '#'

class GridworldEnvironment(environment.Environment):

  def __init__(self, gridWorld):
    self.gridWorld = gridWorld
    self.reset()

  def getCurrentState(self):
    return self.state

  def getPossibleActions(self, state):
    return self.gridWorld.getPossibleActions(state)

  def doAction(self, action):
    successors = self.gridWorld.getTransitionStatesAndProbs(self.state, action)
    sum = 0.0
    rand = random.random()
    state = self.getCurrentState()
    for nextState, prob in successors:
      sum += prob
      if sum > 1.0:
        raise 'Total transition probability more than one; sample failure.'
      if rand < sum:
        reward = self.gridWorld.getReward(state, action, nextState)
        self.state = nextState
        return (nextState, reward)
    raise 'Total transition probability less than one; sample failure.'

  def reset(self):
    self.state = self.gridWorld.getStartState()
```

```
class Grid:
  """
  A 2-dimensional array of immutables backed by a list of lists.  Data is accessed
  via grid[x][y] where (x,y) are cartesian coordinates with x horizontal,
  y vertical and the origin (0,0) in the bottom left corner.

  The __str__ method constructs an output that is oriented appropriately.
  """
  def __init__(self, width, height, initialValue=' '):
    self.width = width
    self.height = height
    self.data = [[initialValue for y in range(height)] for x in range(width)]
    self.terminalState = 'TERMINAL_STATE'

  def __getitem__(self, i):
    return self.data[i]

  def __setitem__(self, key, item):
    self.data[key] = item

  def __eq__(self, other):
    if other == None: return False
    return self.data == other.data

  def __hash__(self):
    return hash(self.data)

  def copy(self):
    g = Grid(self.width, self.height)
    g.data = [x[:] for x in self.data]
    return g

  def deepCopy(self):
    return self.copy()

  def shallowCopy(self):
    g = Grid(self.width, self.height)
    g.data = self.data
    return g

  def _getLegacyText(self):
    t = [[self.data[x][y] for x in range(self.width)] for y in range(self.height)]
    t.reverse()
    return t

  def __str__(self):
    return str(self._getLegacyText())

def makeGrid(gridString):
  width, height = len(gridString[0]), len(gridString)
  grid = Grid(width, height)
  for ybar, line in enumerate(gridString):
    y = height - ybar - 1
    for x, el in enumerate(line):
      grid[x][y] = el
  return grid

def getCliffGrid():
  grid = [[' ',' ',' ',' ',' ',' '],
          ['S',' ',' ',' ',10],
          [-100,-100, -100, -100, -100]]
  return Gridworld(makeGrid(grid))

def getCliffGrid2():
  grid = [[' ',' ',' ',' ',' ',' '],
          [8,'S',' ',' ',10],
          [-100,-100, -100, -100, -100]]
  return Gridworld(grid)
```

```
def getDiscountGrid():
  grid = [[' ',' ',' ',' ',' '],
          [' ','#',' ',' ',' '],
          [' ','#', 1,'#', 10],
          ['S',' ',' ',' ',' '],
          [-10,-10, -10, -10, -10]]
  return Gridworld(grid)

def getBridgeGrid():
  grid = [[ '#',-100, -100, -100, -100, -100, '#'],
          [   1, 'S',  ' ',  ' ',  ' ',  ' ',  10],
          [ '#',-100, -100, -100, -100, -100, '#']]
  return Gridworld(grid)

def getBookGrid():
  grid = [[' ',' ',' ',+1],
          [' ','#',' ',-1],
          ['S',' ',' ',' ']]
  return Gridworld(grid)

def getMazeGrid():
  grid = [[' ',' ',' ',+1],
          ['#','#',' ','#'],
          [' ','#',' ',' '],
          [' ','#','#',' '],
          ['S',' ',' ',' ']]
  return Gridworld(grid)
```



```
def getUserAction(state, actionFunction):
  """
  Get an action from the user (rather than the agent).
```



```
  Used for debugging and lecture demos.
  """
  import graphicsUtils
  action = None
  while True:
```



```
    keys = graphicsUtils.wait_for_keys()
    if 'Up' in keys: action = 'north'
    if 'Down' in keys: action = 'south'
    if 'Left' in keys: action = 'west'
    if 'Right' in keys: action = 'east'
    if 'q' in keys: sys.exit(0)
    if action == None: continue
    break
  actions = actionFunction(state)
  if action not in actions:
    action = actions[0]
  return action

def printString(x): print x

def runEpisode(agent, environment, discount, decision, display, message, pause,
episode):
  returns = 0
  totalDiscount = 1.0
  environment.reset()
  if 'startEpisode' in dir(agent): agent.startEpisode()
  message("BEGINNING EPISODE: "+str(episode)+"\n")
  while True:

    # DISPLAY CURRENT STATE
    state = environment.getCurrentState()
    display(state)
    pause()

    # END IF IN A TERMINAL STATE
```

```
    actions = environment.getPossibleActions(state)
    if len(actions) == 0:
      message("EPISODE "+str(episode)+" COMPLETE: RETURN WAS "+str(returns)+"\n")
      return returns

    # GET ACTION (USUALLY FROM AGENT)
    action = decision(state)
    if action == None:
      raise 'Error: Agent returned None action'

    # EXECUTE ACTION
    nextState, reward = environment.doAction(action)
    message("Started in state: "+str(state)+
            "\nTook action: "+str(action)+
            "\nEnded in state: "+str(nextState)+
            "\nGot reward: "+str(reward)+"\n")
    # UPDATE LEARNER
    if 'observeTransition' in dir(agent):
        agent.observeTransition(state, action, nextState, reward)

    returns += reward * totalDiscount
    totalDiscount *= discount

  if 'stopEpisode' in dir(agent):
    agent.stopEpisode()

def parseOptions():
    optParser = optparse.OptionParser()
    optParser.add_option('-d', '--discount',action='store',
                         type='float',dest='discount',default=0.9,
                         help='Discount on future (default %default)')
    optParser.add_option('-r', '--livingReward',action='store',
                         type='float',dest='livingReward',default=0.0,
                         metavar="R", help='Reward for living for a time step
(default %default)')
    optParser.add_option('-n', '--noise',action='store',
                         type='float',dest='noise',default=0.2,
                         metavar="P", help='How often action results in ' +
                         'unintended direction (default %default)' )
    optParser.add_option('-e', '--epsilon',action='store',
                         type='float',dest='epsilon',default=0.3,
                         metavar="E", help='Chance of taking a random action in q-
learning (default %default)')
    optParser.add_option('-l', '--learningRate',action='store',
                         type='float',dest='learningRate',default=0.5,
                         metavar="P", help='TD learning rate (default %default)' )
    optParser.add_option('-i', '--iterations',action='store',
                         type='int',dest='iters',default=10,
                         metavar="K", help='Number of rounds of value iteration
(default %default)')
    optParser.add_option('-k', '--episodes',action='store',
                         type='int',dest='episodes',default=1,
                         metavar="K", help='Number of epsiodes of the MDP to run
(default %default)')
    optParser.add_option('-g', '--grid',action='store',
                         metavar="G", type='string',dest='grid',default="BookGrid",
                         help='Grid to use (case sensitive; options are BookGrid,
BridgeGrid, CliffGrid, MazeGrid, default %default)' )
    optParser.add_option('-w', '--windowSize', metavar="X",
type='int',dest='gridSize',default=150,
                         help='Request a window width of X pixels *per grid cell*
(default %default)')
    optParser.add_option('-a', '--agent',action='store', metavar="A",
                         type='string',dest='agent',default="random",
                         help='Agent type (options are \'random\', \'value\'
and \'q\', default %default)')
    optParser.add_option('-t', '--text',action='store_true',
                         dest='textDisplay',default=False,
                         help='Use text-only ASCII display')
```

```
    optParser.add_option('-p', '--pause',action='store_true',
                         dest='pause',default=False,
                         help='Pause GUI after each time step when running the MDP')
    optParser.add_option('-q', '--quiet',action='store_true',
                         dest='quiet',default=False,
                         help='Skip display of any learning episodes')
    optParser.add_option('-s', '--speed',action='store', metavar="S", type=float,
                         dest='speed',default=1.0,
                         help='Speed of animation, S > 1.0 is faster, 0.0 < S < 1.0
is slower (default %default)')
    optParser.add_option('-m', '--manual',action='store_true',
                         dest='manual',default=False,
                         help='Manually control agent')
    optParser.add_option('-v', '--valueSteps',action='store_true' ,default=False,
                         help='Display each step of value iteration')

    opts, args = optParser.parse_args()

    if opts.manual and opts.agent != 'q':
      print '## Disabling Agents in Manual Mode (-m) ##'
      opts.agent = None

    # MANAGE CONFLICTS
    if opts.textDisplay or opts.quiet:
    # if opts.quiet:
      opts.pause = False
      # opts.manual = False

    if opts.manual:
      opts.pause = True

    return opts


if __name__ == '__main__':

  opts = parseOptions()

  ###########################
  # GET THE GRIDWORLD
  ###########################

  import gridworld
  mdpFunction = getattr(gridworld, "get"+opts.grid)
  mdp = mdpFunction()
  mdp.setLivingReward(opts.livingReward)
  mdp.setNoise(opts.noise)
  env = gridworld.GridworldEnvironment(mdp)


  ###########################
  # GET THE DISPLAY ADAPTER
  ###########################

  import textGridworldDisplay
  display = textGridworldDisplay.TextGridworldDisplay(mdp)
  if not opts.textDisplay:
    import graphicsGridworldDisplay
    display = graphicsGridworldDisplay.GraphicsGridworldDisplay(mdp, opts.gridSize,
opts.speed)
  display.start()

  ###########################
  # GET THE AGENT
  ###########################

  import valueIterationAgents, qlearningAgents
  a = None
  if opts.agent == 'value':
```

```
    a = valueIterationAgents.ValueIterationAgent(mdp, opts.discount, opts.iters)
  elif opts.agent == 'q':
    #env.getPossibleActions, opts.discount, opts.learningRate, opts.epsilon
    #simulationFn = lambda agent, state:
simulation.GridworldSimulation(agent,state,mdp)
    gridWorldEnv = GridworldEnvironment(mdp)
    actionFn = lambda state: mdp.getPossibleActions(state)
    qLearnOpts = {'gamma': opts.discount,
                  'alpha': opts.learningRate,
                  'epsilon': opts.epsilon,
                  'actionFn': actionFn}
    a = qlearningAgents.QLearningAgent(**qLearnOpts)
  elif opts.agent == 'random':
    # # No reason to use the random agent without episodes
    if opts.episodes == 0:
      opts.episodes = 10
    class RandomAgent:
      def getAction(self, state):
        return random.choice(mdp.getPossibleActions(state))
      def getValue(self, state):
        return 0.0
      def getQValue(self, state, action):
        return 0.0
      def getPolicy(self, state):
        "NOTE: 'random' is a special policy value; don't use it in your code."
        return 'random'
      def update(self, state, action, nextState, reward):
        pass
    a = RandomAgent()
  else:
    if not opts.manual: raise 'Unknown agent type: '+opts.agent


  ###########################
  # RUN EPISODES
  ###########################
  # DISPLAY Q/V VALUES BEFORE SIMULATION OF EPISODES
  if not opts.manual and opts.agent == 'value':
    if opts.valueSteps:
      for i in range(opts.iters):
        tempAgent = valueIterationAgents.ValueIterationAgent(mdp, opts.discount, i)
        display.displayValues(tempAgent, message = "VALUES AFTER "+str(i)+"
ITERATIONS")
        display.pause()

    display.displayValues(a, message = "VALUES AFTER "+str(opts.iters)+" ITERATIONS")
    display.pause()
    display.displayQValues(a, message = "Q-VALUES AFTER "+str(opts.iters)+"
ITERATIONS")
    display.pause()



  # FIGURE OUT WHAT TO DISPLAY EACH TIME STEP (IF ANYTHING)
  displayCallback = lambda x: None
  if not opts.quiet:
    if opts.manual and opts.agent == None:
      displayCallback = lambda state: display.displayNullValues(state)
    else:
      if opts.agent == 'random': displayCallback = lambda state:
display.displayValues(a, state, "CURRENT VALUES")
      if opts.agent == 'value': displayCallback = lambda state:
display.displayValues(a, state, "CURRENT VALUES")
      if opts.agent == 'q': displayCallback = lambda state: display.displayQValues(a,
state, "CURRENT Q-VALUES")

  messageCallback = lambda x: printString(x)
  if opts.quiet:
    messageCallback = lambda x: None
```

```
  # FIGURE OUT WHETHER TO WAIT FOR A KEY PRESS AFTER EACH TIME STEP
  pauseCallback = lambda : None
  if opts.pause:
    pauseCallback = lambda : display.pause()

  # FIGURE OUT WHETHER THE USER WANTS MANUAL CONTROL (FOR DEBUGGING AND DEMOS)
  if opts.manual:
    decisionCallback = lambda state : getUserAction(state, mdp.getPossibleActions)
  else:
    decisionCallback = a.getAction

  # RUN EPISODES
  if opts.episodes > 0:
    print
    print "RUNNING", opts.episodes, "EPISODES"
    print
  returns = 0
  for episode in range(1, opts.episodes+1):
    returns += runEpisode(a, env, opts.discount, decisionCallback, displayCallback,
messageCallback, pauseCallback, episode)
  if opts.episodes > 0:
    print
    print "AVERAGE RETURNS FROM START STATE: "+str((returns+0.0) / opts.episodes)
    print
    print

  # DISPLAY POST-LEARNING VALUES / Q-VALUES
  if opts.agent == 'q' and not opts.manual:
    display.displayQValues(a, message = "Q-VALUES AFTER "+str(opts.episodes)+"
EPISODES")
    display.pause()
    display.displayValues(a, message = "VALUES AFTER "+str(opts.episodes)+"
EPISODES")
    display.pause()
```