```python
# ghostAgents.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

from game import Agent
from game import Actions
from game import Directions
import random
from util import manhattanDistance
import util

class GhostAgent( Agent ):
  def __init__( self, index ):
    self.index = index

  def getAction( self, state ):
    dist = self.getDistribution(state)
    if len(dist) == 0:
      return Directions.STOP
    else:
      return util.chooseFromDistribution( dist )

  def getDistribution(self, state):
    "Returns a Counter encoding a distribution over actions from the provided state."
    util.raiseNotDefined()

class RandomGhost( GhostAgent ):
  "A ghost that chooses a legal action uniformly at random."
  def getDistribution( self, state ):
    dist = util.Counter()
    for a in state.getLegalActions( self.index ): dist[a] = 1.0
    dist.normalize()
    return dist

class DirectionalGhost( GhostAgent ):
  "A ghost that prefers to rush Pacman, or flee when scared."
  def __init__( self, index, prob_attack=0.8, prob_scaredFlee=0.8 ):
    self.index = index
    self.prob_attack = prob_attack
    self.prob_scaredFlee = prob_scaredFlee

  def getDistribution( self, state ):
    # Read variables from state
    ghostState = state.getGhostState( self.index )
    legalActions = state.getLegalActions( self.index )
    pos = state.getGhostPosition( self.index )
    isScared = ghostState.scaredTimer > 0

    speed = 1
    if isScared: speed = 0.5

    actionVectors = [Actions.directionToVector( a, speed ) for a in legalActions]
    newPositions = [( pos[0]+a[0], pos[1]+a[1] ) for a in actionVectors]
```



```python
    pacmanPosition = state.getPacmanPosition()

    # Select best actions given the state
    distancesToPacman = [manhattanDistance( pos, pacmanPosition ) for pos in newPositions]
    if isScared:
      bestScore = max( distancesToPacman )
      bestProb = self.prob_scaredFlee
    else:
      bestScore = min( distancesToPacman )
      bestProb = self.prob_attack
    bestActions = [action for action, distance in zip( legalActions, distancesToPacman ) if
    distance == bestScore]

    # Construct distribution
    dist = util.Counter()
    for a in bestActions: dist[a] = bestProb / len(bestActions)
    for a in legalActions: dist[a] += ( 1-bestProb ) / len(legalActions)
    dist.normalize()
    return dist
```

```python
# keyboardAgents.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

from game import Agent
from game import Directions
import random

class KeyboardAgent(Agent):
  """
  An agent controlled by the keyboard.
  """
  # NOTE: Arrow keys also work.
  WEST_KEY  = 'a'
  EAST_KEY  = 'd'
  NORTH_KEY = 'w'
  SOUTH_KEY = 's'
  STOP_KEY = 'q'

  def __init__( self, index = 0 ):

    self.lastMove = Directions.STOP
    self.index = index
    self.keys = []

  def getAction( self, state):
    from graphicsUtils import keys_waiting
    from graphicsUtils import keys_pressed
    keys = keys_waiting() + keys_pressed()
    if keys != []:
      self.keys = keys

    legal = state.getLegalActions(self.index)
    move = self.getMove(legal)

    if move == Directions.STOP:
      # Try to move in the same direction as before
      if self.lastMove in legal:
        move = self.lastMove

    if (self.STOP_KEY in self.keys) and Directions.STOP in legal: move = Directions.STOP

    if move not in legal:
      move = random.choice(legal)

    self.lastMove = move
    return move

  def getMove(self, legal):
    move = Directions.STOP
    if   (self.WEST_KEY in self.keys or 'Left' in self.keys) and Directions.WEST in legal:
    move = Directions.WEST
    if   (self.EAST_KEY in self.keys or 'Right' in self.keys) and Directions.EAST in legal:
```


d:\Documents and Settings\nomade\Mes documents\program py\pacman\origine\keyboardAgents.py mardi 13 décembre 2011 15:54

```python
    move = Directions.EAST
    if   (self.NORTH_KEY in self.keys or 'Up' in self.keys) and Directions.NORTH in legal:
    move = Directions.NORTH
    if   (self.SOUTH_KEY in self.keys or 'Down' in self.keys) and Directions.SOUTH in legal:
    move = Directions.SOUTH
    return move

class KeyboardAgent2(KeyboardAgent):
  """
  A second agent controlled by the keyboard.
  """
  # NOTE: Arrow keys also work.
  WEST_KEY  = 'j'
  EAST_KEY  = "l"
  NORTH_KEY = 'i'
  SOUTH_KEY = 'k'
  STOP_KEY = 'u'

  def getMove(self, legal):
    move = Directions.STOP
    if   (self.WEST_KEY in self.keys) and Directions.WEST in legal:  move = Directions.WEST
    if   (self.EAST_KEY in self.keys) and Directions.EAST in legal: move = Directions.EAST
    if   (self.NORTH_KEY in self.keys) and Directions.NORTH in legal:   move = Directions.
    NORTH
    if   (self.SOUTH_KEY in self.keys) and Directions.SOUTH in legal: move = Directions.SOUTH
    return move
```

```python
# layout.py
# ---------

from util import manhattanDistance
from game import Grid
import os
import random

VISIBILITY_MATRIX_CACHE = {}

class Layout:
  """  A Layout manages the static information about the game board.  """

  def __init__(self, layoutText):
    self.width = len(layoutText[0])
    self.height= len(layoutText)
    self.walls = Grid(self.width, self.height, False)
    self.food = Grid(self.width, self.height, False)
    self.capsules = []
    self.agentPositions = []
    self.numGhosts = 0
    self.processLayoutText(layoutText)
    self.layoutText = layoutText
    # self.initializeVisibilityMatrix()

  def getNumGhosts(self):
    return self.numGhosts

  def initializeVisibilityMatrix(self):
    global VISIBILITY_MATRIX_CACHE
    if reduce(str.__add__, self.layoutText) not in VISIBILITY_MATRIX_CACHE:
      from game import Directions
      vecs = [(-0.5,0), (0.5,0),(0,-0.5),(0,0.5)]
      dirs = [Directions.NORTH, Directions.SOUTH, Directions.WEST, Directions.EAST]
      vis = Grid(self.width, self.height, {Directions.NORTH:set(), Directions.SOUTH:set(), Directions.EAST:set(),
      Directions.WEST:set(), Directions.STOP:set()})
      for x in range(self.width):
        for y in range(self.height):
          if self.walls[x][y] == False:
            for vec, direction in zip(vecs, dirs):
              dx, dy = vec
              nextx, nexty = x + dx, y + dy
              while (nextx + nexty) != int(nextx) + int(nexty) or not self.walls[int(nextx)][int(nexty)] :
                vis[x][y][direction].add((nextx, nexty))
                nextx, nexty = x + dx, y + dy
      self.visibility = vis
      VISIBILITY_MATRIX_CACHE[reduce(str.__add__, self.layoutText)] = vis
    else:
      self.visibility = VISIBILITY_MATRIX_CACHE[reduce(str.__add__, self.layoutText)]

  def isWall(self, pos):
    x, col = pos
    return self.walls[x][col]

  def getRandomLegalPosition(self):
    x = random.choice(range(self.width))
    y = random.choice(range(self.height))
    while self.isWall( (x, y) ):
      x = random.choice(range(self.width))
      y = random.choice(range(self.height))
    return (x,y)

  def getRandomCorner(self):
    poses = [(1,1), (1, self.height - 2), (self.width - 2, 1), (self.width - 2, self.height - 2)]
    return random.choice(poses)

  def getFurthestCorner(self, pacPos):
    poses = [(1,1), (1, self.height - 2), (self.width - 2, 1), (self.width - 2, self.height - 2)]
    dist, pos = max([(manhattanDistance(p, pacPos), p) for p in poses])
    return pos

  def isVisibleFrom(self, ghostPos, pacPos, pacDirection):
    row, col = [int(x) for x in pacPos]
    return ghostPos in self.visibility[row][col][pacDirection]
```



```python
  def __str__(self):
    return "\n".join(self.layoutText)

  def deepCopy(self):
    return Layout(self.layoutText[:])

  def processLayoutText(self, layoutText):
    """   Coordinates are flipped from the input format to the (x,y) convention here
    The shape of the maze.  Each character
    represents a different type of object.
     % - Wall
     . - Food
     o - Capsule
     G - Ghost
     P - Pacman
    Other characters are ignored.
    """
    maxY = self.height - 1
    for y in range(self.height):
      for x in range(self.width):
        layoutChar = layoutText[maxY - y][x]
        self.processLayoutChar(x, y, layoutChar)
    self.agentPositions.sort()
    self.agentPositions = [ ( i == 0, pos) for i, pos in self.agentPositions]

  def processLayoutChar(self, x, y, layoutChar):
    if layoutChar == '%':
      self.walls[x][y] = True
    elif layoutChar == '.':
      self.food[x][y] = True
    elif layoutChar == 'o':
      self.capsules.append((x, y))
    elif layoutChar == 'P':
      self.agentPositions.append( (0, (x, y) ) )
    elif layoutChar in ['G']:
      self.agentPositions.append( (1, (x, y) ) )
      self.numGhosts += 1
    elif layoutChar in  ['1', '2', '3', '4']:
      self.agentPositions.append( (int(layoutChar), (x,y)))
      self.numGhosts += 1

def getLayout(name, back = 1):
  # Set the layout directory and name to check for different enviroment
  # setups
  layoutDir = ''
  layoutName = name.strip() if name.endswith('.lay') else name.strip() + '.lay'

  if os.path.isdir('layouts/'):
    layoutDir = 'layouts/'
  elif os.path.isdir('src/'):
      layoutDir += 'src/'
      if os.path.isdir(layoutDir+'layouts/'):
          layoutDir += 'layouts/'

  layout = tryToLoad(layoutDir + layoutName)
  if layout == None:
    layout = tryToLoad(layoutName)

  # Try again in super directory
  if layout == None and back > 0:
    curdir = os.path.abspath('.')
    os.chdir('..')
    layout = getLayout(name, back - 1)
    os.chdir(curdir)

  return layout

def tryToLoad(fullname):
  if(not os.path.exists(fullname)): return None
  f = open(fullname)
  try: return Layout([line.strip() for line in f])
  finally: f.close()
```

```python
# graphicsUtils.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 sys
import math
import random
import string
import time
import types
import Tkinter

_Windows = sys.platform == 'win32'  # True if on Win95/98/NT

_root_window = None      # The root window for graphics output
_canvas = None      # The canvas which holds graphics
_canvas_xs = None      # Size of canvas object
_canvas_ys = None
_canvas_x = None      # Current position on canvas
_canvas_y = None
_canvas_col = None      # Current colour (set to black below)
_canvas_tsize = 12
_canvas_tserifs = 0

def formatColor(r, g, b):
  return '#%02x%02x%02x' % (int(r * 255), int(g * 255), int(b * 255))

def colorToVector(color):
  return map(lambda x: int(x, 16) / 256.0, [color[1:3], color[3:5], color[5:7]])

if _Windows:
    _canvas_tfonts = ['times new roman', 'lucida console']
else:
    _canvas_tfonts = ['times', 'lucidasans-24']
    pass # XXX need defaults here

def sleep(secs):
    global _root_window
    if _root_window == None:
        time.sleep(secs)
    else:
        _root_window.update_idletasks()
        _root_window.after(int(1000 * secs), _root_window.quit)
        _root_window.mainloop()

def begin_graphics(width=640, height=480, color=formatColor(0, 0, 0), title=None):

    global _root_window, _canvas, _canvas_x, _canvas_y, _canvas_xs, _canvas_ys, _bg_color

    # Check for duplicate call
    if _root_window is not None:
        # Lose the window.
        _root_window.destroy()

    # Save the canvas size parameters
    _canvas_xs, _canvas_ys = width - 1, height - 1
    _canvas_x, _canvas_y = 0, _canvas_ys
    _bg_color = color

    # Create the root window
    _root_window = Tkinter.Tk()
    _root_window.protocol('WM_DELETE_WINDOW', _destroy_window)
    _root_window.title(title or 'Graphics Window')
    _root_window.resizable(0, 0)
```



```python
    # Create the canvas object
    try:
      _canvas = Tkinter.Canvas(_root_window, width=width, height=height)
      _canvas.pack()
      draw_background()
      _canvas.update()
    except:
      _root_window = None
      raise

    # Bind to key-down and key-up events
    _root_window.bind( "<KeyPress>", _keypress )
    _root_window.bind( "<KeyRelease>", _keyrelease )
    _root_window.bind( "<FocusIn>", _clear_keys )
    _root_window.bind( "<FocusOut>", _clear_keys )
    _root_window.bind( "<Button-1>", _leftclick )
    _root_window.bind( "<Button-2>", _rightclick )
    _root_window.bind( "<Button-3>", _rightclick )
    _root_window.bind( "<Control-Button-1>", _ctrl_leftclick)
    _clear_keys()

_leftclick_loc = None
_rightclick_loc = None
_ctrl_leftclick_loc = None

def _leftclick(event):
  global _leftclick_loc
  _leftclick_loc = (event.x, event.y)

def _rightclick(event):
  global _rightclick_loc
  _rightclick_loc = (event.x, event.y)

def _ctrl_leftclick(event):
  global _ctrl_leftclick_loc
  _ctrl_leftclick_loc = (event.x, event.y)

def wait_for_click():
  while True:
    global _leftclick_loc
    global _rightclick_loc
    global _ctrl_leftclick_loc
    if _leftclick_loc != None:
      val = _leftclick_loc
      _leftclick_loc = None
      return val, 'left'
    if _rightclick_loc != None:
      val = _rightclick_loc
      _rightclick_loc = None
      return val, 'right'
    if _ctrl_leftclick_loc != None:
      val = _ctrl_leftclick_loc
      _ctrl_leftclick_loc = None
      return val, 'ctrl_left'
    sleep(0.05)

def draw_background():
    corners = [(0,0), (0, _canvas_ys), (_canvas_xs, _canvas_ys), (_canvas_xs, 0)]
    polygon(corners, _bg_color, fillColor=_bg_color, filled=True, smoothed=False)

def _destroy_window(event=None):
    sys.exit(0)
#    global _root_window
#    _root_window.destroy()
#    _root_window = None
    #print "DESTROY"
```

```
def end_graphics():
    global _root_window, _canvas, _mouse_enabled
    try:
      try:
        sleep(1)
        if _root_window != None:
          _root_window.destroy()
      except SystemExit, e:
        print 'Ending graphics raised an exception:', e
    finally:
      _root_window = None
      _canvas = None
      _mouse_enabled = 0
      _clear_keys()

def clear_screen(background=None):
    global _canvas_x, _canvas_y
    _canvas.delete('all')
    draw_background()
    _canvas_x, _canvas_y = 0, _canvas_ys

def polygon(coords, outlineColor, fillColor=None, filled=1, smoothed=1, behind=0, width=1):
  c = []
  for coord in coords:
    c.append(coord[0])
    c.append(coord[1])
  if fillColor == None: fillColor = outlineColor
  if filled == 0: fillColor = ""
  poly = _canvas.create_polygon(c, outline=outlineColor, fill=fillColor, smooth=smoothed, width=width)
  if behind > 0:
    _canvas.tag_lower(poly, behind) # Higher should be more visible
  return poly

def square(pos, r, color, filled=1, behind=0):
  x, y = pos
  coords = [(x - r, y - r), (x + r, y - r), (x + r, y + r), (x - r, y + r)]
  return polygon(coords, color, color, filled, 0, behind=behind)

def circle(pos, r, outlineColor, fillColor, endpoints=None, style='pieslice', width=2):
    x, y = pos
    x0, x1 = x - r - 1, x + r
    y0, y1 = y - r - 1, y + r
    if endpoints == None:
      e = [0, 359]
    else:
      e = list(endpoints)
    while e[0] > e[1]: e[1] = e[1] + 360

    return _canvas.create_arc(x0, y0, x1, y1, outline=outlineColor, fill=fillColor,
                              extent=e[1] - e[0], start=e[0], style=style, width=width)

def image(pos, file="../../blueghost.gif"):
    x, y = pos
    # img = PhotoImage(file=file)
    return _canvas.create_image(x, y, image = Tkinter.PhotoImage(file=file), anchor = Tkinter.NW)


def refresh():
      _canvas.update_idletasks()

def moveCircle(id, pos, r, endpoints=None):
    global _canvas_x, _canvas_y

    x, y = pos
#    x0, x1 = x - r, x + r + 1
#    y0, y1 = y - r, y + r + 1
    x0, x1 = x - r - 1, x + r
    y0, y1 = y - r - 1, y + r
```



```
    if endpoints == None:
      e = [0, 359]
    else:
      e = list(endpoints)
    while e[0] > e[1]: e[1] = e[1] + 360

    edit(id, ('start', e[0]), ('extent', e[1] - e[0]))
    move_to(id, x0, y0)

def edit(id, *args):
    _canvas.itemconfigure(id, **dict(args))

def text(pos, color, contents, font='Helvetica', size=12, style='normal', anchor="nw"):
    global _canvas_x, _canvas_y
    x, y = pos
    font = (font, str(size), style)
    return _canvas.create_text(x, y, fill=color, text=contents, font=font, anchor=anchor)

def changeText(id, newText, font=None, size=12, style='normal'):
  _canvas.itemconfigure(id, text=newText)
  if font != None:
    _canvas.itemconfigure(id, font=(font, '-%d' % size, style))

def changeColor(id, newColor):
  _canvas.itemconfigure(id, fill=newColor)

def line(here, there, color=formatColor(0, 0, 0), width=2):
  x0, y0 = here[0], here[1]
  x1, y1 = there[0], there[1]
  return _canvas.create_line(x0, y0, x1, y1, fill=color, width=width)

##############################################################################
### Keypress handling ########################################################
##############################################################################

# We bind to key-down and key-up events.

_keysdown = {}
_keyswaiting = {}
# This holds an unprocessed key release.  We delay key releases by up to
# one call to keys_pressed() to get round a problem with auto repeat.
_got_release = None

def _keypress(event):
    global _got_release
    #remap_arrows(event)
    _keysdown[event.keysym] = 1
    _keyswaiting[event.keysym] = 1
#    print event.char, event.keycode
    _got_release = None

def _keyrelease(event):
    global _got_release
    #remap_arrows(event)
    try:
      del _keysdown[event.keysym]
    except:
      pass
    _got_release = 1

def remap_arrows(event):
    # TURN ARROW PRESSES INTO LETTERS (SHOULD BE IN KEYBOARD AGENT)
    if event.char in ['a', 's', 'd', 'w']:
      return
    if event.keycode in [37, 101]: # LEFT ARROW (win / x)
      event.char = 'a'
    if event.keycode in [38, 99]: # UP ARROW
      event.char = 'w'
```

```
    if event.keycode in [39, 102]: # RIGHT ARROW
      event.char = 'd'
    if event.keycode in [40, 104]: # DOWN ARROW
      event.char = 's'

def _clear_keys(event=None):
    global _keysdown, _got_release, _keyswaiting
    _keysdown = {}
    _keyswaiting = {}
    _got_release = None

def keys_pressed(d_o_e=Tkinter.tkinter.dooneevent,
                 d_w=Tkinter.tkinter.DONT_WAIT):
    d_o_e(d_w)
    if _got_release:
      d_o_e(d_w)
    return _keysdown.keys()

def keys_waiting():
  global _keyswaiting
  keys = _keyswaiting.keys()
  _keyswaiting = {}
  return keys

# Block for a list of keys...

def wait_for_keys():
    keys = []
    while keys == []:
        keys = keys_pressed()
        sleep(0.05)
    return keys

def remove_from_screen(x,
                       d_o_e=Tkinter.tkinter.dooneevent,
                       d_w=Tkinter.tkinter.DONT_WAIT):
    _canvas.delete(x)
    d_o_e(d_w)

def _adjust_coords(coord_list, x, y):
    for i in range(0, len(coord_list), 2):
        coord_list[i] = coord_list[i] + x
        coord_list[i + 1] = coord_list[i + 1] + y
    return coord_list

def move_to(object, x, y=None,
            d_o_e=Tkinter.tkinter.dooneevent,
            d_w=Tkinter.tkinter.DONT_WAIT):
    if y is None:
        try: x, y = x
        except: raise  'incomprehensible coordinates'

    horiz = True
    newCoords = []
    current_x, current_y = _canvas.coords(object)[0:2] # first point
    for coord in  _canvas.coords(object):
      if horiz:
        inc = x - current_x
      else:
        inc = y - current_y
      horiz = not horiz

      newCoords.append(coord + inc)

    _canvas.coords(object, *newCoords)
    d_o_e(d_w)

def move_by(object, x, y=None,
```



```
            d_o_e=Tkinter.tkinter.dooneevent,
            d_w=Tkinter.tkinter.DONT_WAIT):
    if y is None:
        try: x, y = x
        except: raise Exception, 'incomprehensible coordinates'

    horiz = True
    newCoords = []
    for coord in  _canvas.coords(object):
      if horiz:
        inc = x
      else:
        inc = y
      horiz = not horiz

      newCoords.append(coord + inc)

    _canvas.coords(object, *newCoords)
    d_o_e(d_w)

def writePostscript(filename):
  "Writes the current canvas to a postscript file."
  psfile = file(filename, 'w')
  psfile.write(_canvas.postscript(pageanchor='sw',
                   y='0.c',
                   x='0.c'))
  psfile.close()

ghost_shape = [
    (0, - 0.5),
    (0.25, - 0.75),
    (0.5, - 0.5),
    (0.75, - 0.75),
    (0.75, 0.5),
    (0.5, 0.75),
    (- 0.5, 0.75),
    (- 0.75, 0.5),
    (- 0.75, - 0.75),
    (- 0.5, - 0.5),
    (- 0.25, - 0.75)
  ]

if __name__ == '__main__':
  begin_graphics()
  clear_screen()
  ghost_shape = [(x * 10 + 20, y * 10 + 20) for x, y in ghost_shape]
  g = polygon(ghost_shape, formatColor(1, 1, 1))
  move_to(g, (50, 50))
  circle((150, 150), 20, formatColor(0.7, 0.3, 0.0), endpoints=[15, - 15])
  sleep(2)
```

```python
# util.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 sys
import inspect
import heapq, random


"""
 Data structures useful for implementing SearchAgents
"""

class Stack:
  "A container with a last-in-first-out (LIFO) queuing policy."
  def __init__(self):
    self.list = []

  def push(self,item):
    "Push 'item' onto the stack"
    self.list.append(item)

  def pop(self):
    "Pop the most recently pushed item from the stack"
    return self.list.pop()

  def isEmpty(self):
    "Returns true if the stack is empty"
    return len(self.list) == 0

class Queue:
  "A container with a first-in-first-out (FIFO) queuing policy."
  def __init__(self):
    self.list = []

  def push(self,item):
    "Enqueue the 'item' into the queue"
    self.list.insert(0,item)

  def pop(self):
    """
      Dequeue the earliest enqueued item still in the queue. This
      operation removes the item from the queue.
    """
    return self.list.pop()

  def isEmpty(self):
    "Returns true if the queue is empty"
    return len(self.list) == 0

class PriorityQueue:
  """
    Implements a priority queue data structure. Each inserted item
    has a priority associated with it and the client is usually interested
    in quick retrieval of the lowest-priority item in the queue. This
    data structure allows O(1) access to the lowest-priority item.

    Note that this PriorityQueue does not allow you to change the priority
    of an item.  However, you may insert the same item multiple times with
    different priorities.
  """
  def  __init__(self):
    self.heap = []

  def push(self, item, priority):
      pair = (priority,item)
      heapq.heappush(self.heap,pair)
```



```python
  def pop(self):
      (priority,item) = heapq.heappop(self.heap)
      return item

  def isEmpty(self):
    return len(self.heap) == 0

class PriorityQueueWithFunction(PriorityQueue):
  """
  Implements a priority queue with the same push/pop signature of the
  Queue and the Stack classes. This is designed for drop-in replacement for
  those two classes. The caller has to provide a priority function, which
  extracts each item's priority.
  """
  def  __init__(self, priorityFunction):
    "priorityFunction (item) -> priority"
    self.priorityFunction = priorityFunction      # store the priority function
    PriorityQueue.__init__(self)        # super-class initializer

  def push(self, item):
    "Adds an item to the queue with priority from the priority function"
    PriorityQueue.push(self, item, self.priorityFunction(item))


def manhattanDistance( xy1, xy2 ):
  "Returns the Manhattan distance between points xy1 and xy2"
  return abs( xy1[0] - xy2[0] ) + abs( xy1[1] - xy2[1] )

"""
  Data structures and functions useful for various course projects

  The search project should not need anything below this line.
"""

class Counter(dict):
  """
  A counter keeps track of counts for a set of keys.

  The counter class is an extension of the standard python
  dictionary type.  It is specialized to have number values
  (integers or floats), and includes a handful of additional
  functions to ease the task of counting data.  In particular,
  all keys are defaulted to have value 0.  Using a dictionary:

  a = {}
  print a['test']

  would give an error, while the Counter class analogue:

  >>> a = Counter()
  >>> print a['test']
  0

  returns the default 0 value. Note that to reference a key
  that you know is contained in the counter,
  you can still use the dictionary syntax:

  >>> a = Counter()
  >>> a['test'] = 2
  >>> print a['test']
  2

  This is very useful for counting things without initializing their counts,
  see for example:

  >>> a['blah'] += 1
  >>> print a['blah']
  1

  The counter also includes additional functionality useful in implementing
  the classifiers for this assignment.  Two counters can be added,
  subtracted or multiplied together.  See below for details.  They can
```

```
  also be normalized and their total count and arg max can be extracted.
  """
  def __getitem__(self, idx):
    self.setdefault(idx, 0)
    return dict.__getitem__(self, idx)

  def incrementAll(self, keys, count):
    """
    Increments all elements of keys by the same count.

    >>> a = Counter()
    >>> a.incrementAll(['one','two', 'three'], 1)
    >>> a['one']
    1
    >>> a['two']
    1
    """
    for key in keys:
      self[key] += count

  def argMax(self):
    """
    Returns the key with the highest value.
    """
    if len(self.keys()) == 0: return None
    all = self.items()
    values = [x[1] for x in all]
    maxIndex = values.index(max(values))
    return all[maxIndex][0]

  def sortedKeys(self):
    """
    Returns a list of keys sorted by their values.  Keys
    with the highest values will appear first.

    >>> a = Counter()
    >>> a['first'] = -2
    >>> a['second'] = 4
    >>> a['third'] = 1
    >>> a.sortedKeys()
    ['second', 'third', 'first']
    """
    sortedItems = self.items()
    compare = lambda x, y:  sign(y[1] - x[1])
    sortedItems.sort(cmp=compare)
    return [x[0] for x in sortedItems]

  def totalCount(self):
    """
    Returns the sum of counts for all keys.
    """
    return sum(self.values())

  def normalize(self):
    """
    Edits the counter such that the total count of all
    keys sums to 1.  The ratio of counts for all keys
    will remain the same. Note that normalizing an empty
    Counter will result in an error.
    """
    total = float(self.totalCount())
    if total == 0: return
    for key in self.keys():
      self[key] = self[key] / total

  def divideAll(self, divisor):
    """
    Divides all counts by divisor
    """
    divisor = float(divisor)
    for key in self:
      self[key] /= divisor
```



```
  def copy(self):
    """
    Returns a copy of the counter
    """
    return Counter(dict.copy(self))

  def __mul__(self, y ):
    """
    Multiplying two counters gives the dot product of their vectors where
    each unique label is a vector element.

    >>> a = Counter()
    >>> b = Counter()
    >>> a['first'] = -2
    >>> a['second'] = 4
    >>> b['first'] = 3
    >>> b['second'] = 5
    >>> a['third'] = 1.5
    >>> a['fourth'] = 2.5
    >>> a * b
    14
    """
    sum = 0
    x = self
    if len(x) > len(y):
      x,y = y,x
    for key in x:
      if key not in y:
        continue
      sum += x[key] * y[key]
    return sum

  def __radd__(self, y):
    """
    Adding another counter to a counter increments the current counter
    by the values stored in the second counter.

    >>> a = Counter()
    >>> b = Counter()
    >>> a['first'] = -2
    >>> a['second'] = 4
    >>> b['first'] = 3
    >>> b['third'] = 1
    >>> a += b
    >>> a['first']
    1
    """
    for key, value in y.items():
      self[key] += value

  def __add__( self, y ):
    """
    Adding two counters gives a counter with the union of all keys and
    counts of the second added to counts of the first.

    >>> a = Counter()
    >>> b = Counter()
    >>> a['first'] = -2
    >>> a['second'] = 4
    >>> b['first'] = 3
    >>> b['third'] = 1
    >>> (a + b)['first']
    1
    """
    addend = Counter()
    for key in self:
      if key in y:
        addend[key] = self[key] + y[key]
      else:
        addend[key] = self[key]
    for key in y:
```

```python
      if key in self:
        continue
      addend[key] = y[key]
    return addend

  def __sub__( self, y ):
    """
    Subtracting a counter from another gives a counter with the union of all keys and
    counts of the second subtracted from counts of the first.

    >>> a = Counter()
    >>> b = Counter()
    >>> a['first'] = -2
    >>> a['second'] = 4
    >>> b['first'] = 3
    >>> b['third'] = 1
    >>> (a - b)['first']
    -5
    """
    addend = Counter()
    for key in self:
      if key in y:
        addend[key] = self[key] - y[key]
      else:
        addend[key] = self[key]
    for key in y:
      if key in self:
        continue
      addend[key] = -1 * y[key]
    return addend

def raiseNotDefined():
  print "Method not implemented: %s" % inspect.stack()[1][3]
  sys.exit(1)

def normalize(vectorOrCounter):
  """
  normalize a vector or counter by dividing each value by the sum of all values
  """
  normalizedCounter = Counter()
  if type(vectorOrCounter) == type(normalizedCounter):
    counter = vectorOrCounter
    total = float(counter.totalCount())
    if total == 0: return counter
    for key in counter.keys():
      value = counter[key]
      normalizedCounter[key] = value / total
    return normalizedCounter
  else:
    vector = vectorOrCounter
    s = float(sum(vector))
    if s == 0: return vector
    return [el / s for el in vector]

def nSample(distribution, values, n):
  if sum(distribution) != 1:
    distribution = normalize(distribution)
  rand = [random.random() for i in range(n)]
  rand.sort()
  samples = []
  samplePos, distPos, cdf = 0,0, distribution[0]
  while samplePos < n:
    if rand[samplePos] < cdf:
      samplePos += 1
      samples.append(values[distPos])
    else:
      distPos += 1
      cdf += distribution[distPos]
  return samples

def sample(distribution, values = None):
  if type(distribution) == Counter:
```



```python
    items = distribution.items()
    distribution = [i[1] for i in items]
    values = [i[0] for i in items]
  if sum(distribution) != 1:
    distribution = normalize(distribution)
  choice = random.random()
  i, total= 0, distribution[0]
  while choice > total:
    i += 1
    total += distribution[i]
  return values[i]

def sampleFromCounter(ctr):
  items = ctr.items()
  return sample([v for k,v in items], [k for k,v in items])

def getProbability(value, distribution, values):
  """
    Gives the probability of a value under a discrete distribution
    defined by (distributions, values).
  """
  total = 0.0
  for prob, val in zip(distribution, values):
    if val == value:
      total += prob
  return total

def flipCoin( p ):
  r = random.random()
  return r < p

def chooseFromDistribution( distribution ):
  "Takes either a counter or a list of (prob, key) pairs and samples"
  if type(distribution) == dict or type(distribution) == Counter:
    return sample(distribution)
  r = random.random()
  base = 0.0
  for prob, element in distribution:
    base += prob
    if r <= base: return element

def nearestPoint( pos ):
  """
  Finds the nearest grid point to a position (discretizes).
  """
  ( current_row, current_col ) = pos

  grid_row = int( current_row + 0.5 )
  grid_col = int( current_col + 0.5 )
  return ( grid_row, grid_col )

def sign( x ):
  """
  Returns 1 or -1 depending on the sign of x
  """
  if( x >= 0 ):
    return 1
  else:
    return -1

def arrayInvert(array):
  """
  Inverts a matrix stored as a list of lists.
  """
  result = [[] for i in array]
  for outer in array:
    for inner in range(len(outer)):
      result[inner].append(outer[inner])
  return result

def matrixAsList( matrix, value = True ):
  """
```

```
  Turns a matrix into a list of coordinates matching the specified value
  """
  rows, cols = len( matrix ), len( matrix[0] )
  cells = []
  for row in range( rows ):
    for col in range( cols ):
      if matrix[row][col] == value:
        cells.append( ( row, col ) )
  return cells

def lookup(name, namespace):
  """
  Get a method or class from any imported module from its name.
  Usage: lookup(functionName, globals())
  """
  dots = name.count('.')
  if dots > 0:
    moduleName, objName = '.'.join(name.split('.')[:-1]), name.split('.')[-1]
    module = __import__(moduleName)
    return getattr(module, objName)
  else:
    modules = [obj for obj in namespace.values() if str(type(obj)) == "<type 'module'>"]
    options = [getattr(module, name) for module in modules if name in dir(module)]
    options += [obj[1] for obj in namespace.items() if obj[0] == name ]
    if len(options) == 1: return options[0]
    if len(options) > 1: raise Exception, 'Name conflict for %s'
    raise Exception, '%s not found as a method or class' % name

def pause():
  """
  Pauses the output stream awaiting user feedback.
  """
  print "<Press enter/return to continue>"
  raw_input()


## code to handle timeouts
import signal
class TimeoutFunctionException(Exception):
    """Exception to raise on a timeout"""
    pass

class TimeoutFunction:

    def __init__(self, function, timeout):
        "timeout must be at least 1 second. WHY??"
        self.timeout = timeout
        self.function = function

    def handle_timeout(self, signum, frame):
        raise TimeoutFunctionException()

    def __call__(self, *args):
        if not 'SIGALRM' in dir(signal):
            return self.function(*args)
        old = signal.signal(signal.SIGALRM, self.handle_timeout)
        signal.alarm(self.timeout)
        try:
            result = self.function(*args)
        finally:
            signal.signal(signal.SIGALRM, old)
        signal.alarm(0)
        return result
```

```python
# search.py
# ---------


""" In search.py, you will implement generic search algorithms which are called
by Pacman agents (in searchAgents.py).
"""

import util

class SearchProblem:
  """
  This class outlines the structure of a search problem, but doesn't implement
  any of the methods (in object-oriented terminology: an abstract class).

  You do not need to change anything in this class, ever.
  """

  def getStartState(self):
     """
     Returns the start state for the search problem
     """
     util.raiseNotDefined()

  def isGoalState(self, state):
     """
       state: Search state

     Returns True if and only if the state is a valid goal state
     """
     util.raiseNotDefined()

  def getSuccessors(self, state):
     """  state: Search state

     For a given state, this should return a list of triples,
     (successor, action, stepCost), where 'successor' is a
     successor to the current state, 'action' is the action
     required to get there, and 'stepCost' is the incremental
     cost of expanding to that successor
     """
     util.raiseNotDefined()

  def getCostOfActions(self, actions):
     """
      actions: A list of actions to take

     This method returns the total cost of a particular sequence of actions.  The sequence must
     be composed of legal moves
     """
     util.raiseNotDefined()


def tinyMazeSearch(problem):
  """
  Returns a sequence of moves that solves tinyMaze.  For any other
  maze, the sequence of moves will be incorrect, so only use this for tinyMaze
  """
  from game import Directions
  s = Directions.SOUTH
  w = Directions.WEST
  return  [s,s,w,s,w,w,s,w]

def depthFirstSearch(problem):
```


d:\Documents and Settings\nomade\Mes documents\program py\pacman\origine\search.py mardi 13 décembre 2011 15:51

```python
  """
  Search the deepest nodes in the search tree first
  [2nd Edition: p 75, 3rd Edition: p 87]

  Your search algorithm needs to return a list of actions that reaches
  the goal.  Make sure to implement a graph search algorithm
  [2nd Edition: Fig. 3.18, 3rd Edition: Fig 3.7].

  To get started, you might want to try some of these simple commands to
  understand the search problem that is being passed in:

  print "Start:", problem.getStartState()
  print "Is the start a goal?", problem.isGoalState(problem.getStartState())
  print "Start's successors:", problem.getSuccessors(problem.getStartState())
  """
  "*** YOUR CODE HERE ***"
  util.raiseNotDefined()

def breadthFirstSearch(problem):
  """
  Search the shallowest nodes in the search tree first.
  [2nd Edition: p 73, 3rd Edition: p 82]
  """
  "*** YOUR CODE HERE ***"
  util.raiseNotDefined()

def uniformCostSearch(problem):
  "Search the node of least total cost first. "
  "*** YOUR CODE HERE ***"
  util.raiseNotDefined()

def nullHeuristic(state, problem=None):
  """
  A heuristic function estimates the cost from the current state to the nearest
  goal in the provided SearchProblem.  This heuristic is trivial.
  """
  return 0

def aStarSearch(problem, heuristic=nullHeuristic):
  "Search the node that has the lowest combined cost and heuristic first."
  "*** YOUR CODE HERE ***"
  util.raiseNotDefined()

# Abbreviations
bfs = breadthFirstSearch
dfs = depthFirstSearch
astar = aStarSearch
ucs = uniformCostSearch
```

```python
# pacmanAgents.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

from pacman import Directions
from game import Agent
import random
import game
import util

class LeftTurnAgent(game.Agent):
  "An agent that turns left at every opportunity"

  def getAction(self, state):
    legal = state.getLegalPacmanActions()
    current = state.getPacmanState().configuration.direction
    if current == Directions.STOP: current = Directions.NORTH
    left = Directions.LEFT[current]
    if left in legal: return left
    if current in legal: return current
    if Directions.RIGHT[current] in legal: return Directions.RIGHT[current]
    if Directions.LEFT[left] in legal: return Directions.LEFT[left]
    return Directions.STOP

class GreedyAgent(Agent):
  def __init__(self, evalFn="scoreEvaluation"):
    self.evaluationFunction = util.lookup(evalFn, globals())
    assert self.evaluationFunction != None

  def getAction(self, state):
    # Generate candidate actions
    legal = state.getLegalPacmanActions()
    if Directions.STOP in legal: legal.remove(Directions.STOP)

    successors = [(state.generateSuccessor(0, action), action) for action in legal]
    scored = [(self.evaluationFunction(state), action) for state, action in successors]
    bestScore = max(scored)[0]
    bestActions = [pair[1] for pair in scored if pair[0] == bestScore]
    return random.choice(bestActions)

def scoreEvaluation(state):
  return state.getScore()
```

```python
# textDisplay.py
# --------------

import pacman, time

DRAW_EVERY = 1
SLEEP_TIME = 0 # This can be overwritten by __init__
DISPLAY_MOVES = False
QUIET = False # Supresses output

class NullGraphics:
  def initialize(self, state, isBlue = False):
    pass

  def update(self, state):
    pass

  def pause(self):
    time.sleep(SLEEP_TIME)

  def draw(self, state):
    print state

  def finish(self):
    pass

class PacmanGraphics:
  def __init__(self, speed=None):
    if speed != None:
      global SLEEP_TIME
      SLEEP_TIME = speed

  def initialize(self, state, isBlue = False):
    self.draw(state)
    self.pause()
    self.turn = 0
    self.agentCounter = 0

  def update(self, state):
    numAgents = len(state.agentStates)
    self.agentCounter = (self.agentCounter + 1) % numAgents
    if self.agentCounter == 0:
      self.turn += 1
      if DISPLAY_MOVES:
        ghosts = [pacman.nearestPoint(state.getGhostPosition(i)) for i in range(1, numAgents)]
        print "%4d) P: %-8s" % (self.turn, str(pacman.nearestPoint(state.getPacmanPosition()))),'| Score:
        %-5d' % state.score,'| Ghosts:', ghosts
      if self.turn % DRAW_EVERY == 0:
        self.draw(state)
        self.pause()
    if state._win or state._lose:
      self.draw(state)

  def pause(self):
    time.sleep(SLEEP_TIME)

  def draw(self, state):
    print state

  def finish(self):
    pass
```

```python
# graphicsDisplay.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

from graphicsUtils import *
import math, time
from game import Directions

###########################
#  GRAPHICS DISPLAY CODE  #
###########################

# Most code by Dan Klein and John Denero written or rewritten for cs188, UC Berkeley.
# Some code from a Pacman implementation by LiveWires, and used / modified with permission.

DEFAULT_GRID_SIZE = 30.0
INFO_PANE_HEIGHT = 35
BACKGROUND_COLOR = formatColor(0,0,0)
WALL_COLOR = formatColor(0.0/255.0, 51.0/255.0, 255.0/255.0)
INFO_PANE_COLOR = formatColor(.4,.4,0)
SCORE_COLOR = formatColor(.9, .9, .9)
PACMAN_OUTLINE_WIDTH = 2
PACMAN_CAPTURE_OUTLINE_WIDTH = 4

GHOST_COLORS = []
GHOST_COLORS.append(formatColor(.9,0,0)) # Red
GHOST_COLORS.append(formatColor(0,.3,.9)) # Blue
GHOST_COLORS.append(formatColor(.98,.41,.07)) # Orange
GHOST_COLORS.append(formatColor(.1,.75,.7)) # Green
GHOST_COLORS.append(formatColor(1.0,0.6,0.0)) # Yellow
GHOST_COLORS.append(formatColor(.4,0.13,0.91)) # Purple

TEAM_COLORS = GHOST_COLORS[:2]

GHOST_SHAPE = [
    ( 0,    0.3 ),
    ( 0.25, 0.75 ),
    ( 0.5,  0.3 ),
    ( 0.75, 0.75 ),
    ( 0.75, -0.5 ),
    ( 0.5,  -0.75 ),
    (-0.5,  -0.75 ),
    (-0.75, -0.5 ),
    (-0.75, 0.75 ),
    (-0.5,  0.3 ),
    (-0.25, 0.75 )
  ]
GHOST_SIZE = 0.65
SCARED_COLOR = formatColor(1,1,1)

GHOST_VEC_COLORS = map(colorToVector, GHOST_COLORS)

PACMAN_COLOR = formatColor(255.0/255.0,255.0/255.0,61.0/255)
PACMAN_SCALE = 0.5
#pacman_speed = 0.25

# Food
FOOD_COLOR = formatColor(1,1,1)
FOOD_SIZE = 0.1

# Laser
LASER_COLOR = formatColor(1,0,0)
LASER_SIZE = 0.02

# Capsule graphics
CAPSULE_COLOR = formatColor(1,1,1)
CAPSULE_SIZE = 0.25
```



```python
# Drawing walls
WALL_RADIUS = 0.15

class InfoPane:
  def __init__(self, layout, gridSize):
    self.gridSize = gridSize
    self.width = (layout.width) * gridSize
    self.base = (layout.height + 1) * gridSize
    self.height = INFO_PANE_HEIGHT
    self.fontSize = 24
    self.textColor = PACMAN_COLOR
    self.drawPane()

  def toScreen(self, pos, y = None):
    """
      Translates a point relative from the bottom left of the info pane.
    """
    if y == None:
      x,y = pos
    else:
      x = pos

    x = self.gridSize + x # Margin
    y = self.base + y
    return x,y

  def drawPane(self):
    self.scoreText = text( self.toScreen(0, 0  ), self.textColor, "SCORE:    0", "Times", self.fontSize, "bold")

  def initializeGhostDistances(self, distances):
    self.ghostDistanceText = []

    size = 20
    if self.width < 240:
      size = 12
    if self.width < 160:
      size = 10

    for i, d in enumerate(distances):
      t = text( self.toScreen(self.width/2 + self.width/8 * i, 0), GHOST_COLORS[i+1], d, "Times", size, "bold")
      self.ghostDistanceText.append(t)

  def updateScore(self, score):
    changeText(self.scoreText, "SCORE: % 4d" % score)

  def setTeam(self, isBlue):
    text = "RED TEAM"
    if isBlue: text = "BLUE TEAM"
    self.teamText = text( self.toScreen(300, 0  ), self.textColor, text, "Times", self.fontSize, "bold")

  def updateGhostDistances(self, distances):
    if len(distances) == 0: return
    if 'ghostDistanceText' not in dir(self): self.initializeGhostDistances(distances)
    else:
      for i, d in enumerate(distances):
        changeText(self.ghostDistanceText[i], d)

  def drawGhost(self):
    pass

  def drawPacman(self):
    pass

  def drawWarning(self):
    pass

  def clearIcon(self):
    pass

  def updateMessage(self, message):
    pass
```

```
  def clearMessage(self):
    pass


class PacmanGraphics:
  def __init__(self, zoom=1.0, frameTime=0.0, capture=False):
    self.have_window = 0
    self.currentGhostImages = {}
    self.pacmanImage = None
    self.zoom = zoom
    self.gridSize = DEFAULT_GRID_SIZE * zoom
    self.capture = capture
    self.frameTime = frameTime

  def initialize(self, state, isBlue = False):
    self.isBlue = isBlue
    self.startGraphics(state)

    # self.drawDistributions(state)
    self.distributionImages = None  # Initialized lazily
    self.drawStaticObjects(state)
    self.drawAgentObjects(state)

    # Information
    self.previousState = state

  def startGraphics(self, state):
    self.layout = state.layout
    layout = self.layout
    self.width = layout.width
    self.height = layout.height
    self.make_window(self.width, self.height)
    self.infoPane = InfoPane(layout, self.gridSize)
    self.currentState = layout

  def drawDistributions(self, state):
    walls = state.layout.walls
    dist = []
    for x in range(walls.width):
      distx = []
      dist.append(distx)
      for y in range(walls.height):
          ( screen_x, screen_y ) = self.to_screen( (x, y) )
          block = square( (screen_x, screen_y),
                          0.5 * self.gridSize,
                          color = BACKGROUND_COLOR,
                          filled = 1, behind=2)
          distx.append(block)
    self.distributionImages = dist

  def drawStaticObjects(self, state):
    layout = self.layout
    self.drawWalls(layout.walls)
    self.food = self.drawFood(layout.food)
    self.capsules = self.drawCapsules(layout.capsules)
    refresh()

  def drawAgentObjects(self, state):
    self.agentImages = [] # (agentState, image)
    for index, agent in enumerate(state.agentStates):
      if agent.isPacman:
        image = self.drawPacman(agent, index)
        self.agentImages.append( (agent, image) )
      else:
        image = self.drawGhost(agent, index)
        self.agentImages.append( (agent, image) )
    refresh()

  def swapImages(self, agentIndex, newState):
    """
      Changes an image from a ghost to a pacman or vis versa (for capture)
    """
```



```
    prevState, prevImage = self.agentImages[agentIndex]
    for item in prevImage: remove_from_screen(item)
    if newState.isPacman:
      image = self.drawPacman(newState, agentIndex)
      self.agentImages[agentIndex] = (newState, image )
    else:
      image = self.drawGhost(newState, agentIndex)
      self.agentImages[agentIndex] = (newState, image )
    refresh()

  def update(self, newState):
    agentIndex = newState._agentMoved
    agentState = newState.agentStates[agentIndex]

    if self.agentImages[agentIndex][0].isPacman != agentState.isPacman: self.swapImages(agentIndex, agentState)
    prevState, prevImage = self.agentImages[agentIndex]
    if agentState.isPacman:
      self.animatePacman(agentState, prevState, prevImage)
    else:
      self.moveGhost(agentState, agentIndex, prevState, prevImage)
    self.agentImages[agentIndex] = (agentState, prevImage)

    if newState._foodEaten != None:
      self.removeFood(newState._foodEaten, self.food)
    if newState._capsuleEaten != None:
      self.removeCapsule(newState._capsuleEaten, self.capsules)
    self.infoPane.updateScore(newState.score)
    if 'ghostDistances' in dir(newState):
      self.infoPane.updateGhostDistances(newState.ghostDistances)

  def make_window(self, width, height):
    grid_width = (width-1) * self.gridSize
    grid_height = (height-1) * self.gridSize
    screen_width = 2*self.gridSize + grid_width
    screen_height = 2*self.gridSize + grid_height + INFO_PANE_HEIGHT

    begin_graphics(screen_width,
                   screen_height,
                   BACKGROUND_COLOR,
                   "CS188 Pacman")

  def drawPacman(self, pacman, index):
    position = self.getPosition(pacman)
    screen_point = self.to_screen(position)
    endpoints = self.getEndpoints(self.getDirection(pacman))

    width = PACMAN_OUTLINE_WIDTH
    outlineColor = PACMAN_COLOR
    fillColor = PACMAN_COLOR

    if self.capture:
      outlineColor = TEAM_COLORS[index % 2]
      fillColor = GHOST_COLORS[index]
      width = PACMAN_CAPTURE_OUTLINE_WIDTH

    return [circle(screen_point, PACMAN_SCALE * self.gridSize,
                   fillColor = fillColor, outlineColor = outlineColor,
                   endpoints = endpoints,
                   width = width)]

  def getEndpoints(self, direction, position=(0,0)):
    x, y = position
    pos = x - int(x) + y - int(y)
    width = 30 + 80 * math.sin(math.pi* pos)

    delta = width / 2
    if (direction == 'West'):
      endpoints = (180+delta, 180-delta)
    elif (direction == 'North'):
      endpoints = (90+delta, 90-delta)
    elif (direction == 'South'):
      endpoints = (270+delta, 270-delta)
```

```
    else:
      endpoints = (0+delta, 0-delta)
    return endpoints

  def movePacman(self, position, direction, image):
    screenPosition = self.to_screen(position)
    endpoints = self.getEndpoints( direction, position )
    r = PACMAN_SCALE * self.gridSize
    moveCircle(image[0], screenPosition, r, endpoints)
    refresh()

  def animatePacman(self, pacman, prevPacman, image):
    if self.frameTime < 0:
      print 'Press any key to step forward, "q" to play'
      keys = wait_for_keys()
      if 'q' in keys:
        self.frameTime = 0.1
    if self.frameTime > 0.01 or self.frameTime < 0:
      start = time.time()
      fx, fy = self.getPosition(prevPacman)
      px, py = self.getPosition(pacman)
      frames = 4.0
      for i in range(1,int(frames) + 1):
        pos = px*i/frames + fx*(frames-i)/frames, py*i/frames + fy*(frames-i)/frames
        self.movePacman(pos, self.getDirection(pacman), image)
        refresh()
        sleep(abs(self.frameTime) / frames)
    else:
      self.movePacman(self.getPosition(pacman), self.getDirection(pacman), image)
    refresh()

  def getGhostColor(self, ghost, ghostIndex):
    if ghost.scaredTimer > 0:
      return SCARED_COLOR
    else:
      return GHOST_COLORS[ghostIndex]

  def drawGhost(self, ghost, agentIndex):
    pos = self.getPosition(ghost)
    dir = self.getDirection(ghost)
    (screen_x, screen_y) = (self.to_screen(pos) )
    coords = []
    for (x, y) in GHOST_SHAPE:
      coords.append((x*self.gridSize*GHOST_SIZE + screen_x, y*self.gridSize*GHOST_SIZE + screen_y))

    colour = self.getGhostColor(ghost, agentIndex)
    body = polygon(coords, colour, filled = 1)
    WHITE = formatColor(1.0, 1.0, 1.0)
    BLACK = formatColor(0.0, 0.0, 0.0)

    dx = 0
    dy = 0
    if dir == 'North':
      dy = -0.2
    if dir == 'South':
      dy = 0.2
    if dir == 'East':
      dx = 0.2
    if dir == 'West':
      dx = -0.2
    leftEye = circle((screen_x+self.gridSize*GHOST_SIZE*(-0.3+dx/1.5), screen_y-self.gridSize*GHOST_SIZE*(0.3-dy/
    1.5)), self.gridSize*GHOST_SIZE*0.2, WHITE, WHITE)
    rightEye = circle((screen_x+self.gridSize*GHOST_SIZE*(0.3+dx/1.5), screen_y-self.gridSize*GHOST_SIZE*(0.3-dy/
    1.5)), self.gridSize*GHOST_SIZE*0.2, WHITE, WHITE)
    leftPupil = circle((screen_x+self.gridSize*GHOST_SIZE*(-0.3+dx), screen_y-self.gridSize*GHOST_SIZE*(0.3-dy)),
    self.gridSize*GHOST_SIZE*0.08, BLACK, BLACK)
    rightPupil = circle((screen_x+self.gridSize*GHOST_SIZE*(0.3+dx), screen_y-self.gridSize*GHOST_SIZE*(0.3-dy)),
    self.gridSize*GHOST_SIZE*0.08, BLACK, BLACK)
    ghostImageParts = []
    ghostImageParts.append(body)
    ghostImageParts.append(leftEye)
    ghostImageParts.append(rightEye)
```



```
    ghostImageParts.append(leftPupil)
    ghostImageParts.append(rightPupil)

    return ghostImageParts

  def moveEyes(self, pos, dir, eyes):
    (screen_x, screen_y) = (self.to_screen(pos) )
    dx = 0
    dy = 0
    if dir == 'North':
      dy = -0.2
    if dir == 'South':
      dy = 0.2
    if dir == 'East':
      dx = 0.2
    if dir == 'West':
      dx = -0.2
    moveCircle(eyes[0],(screen_x+self.gridSize*GHOST_SIZE*(-0.3+dx/1.5), screen_y-self.gridSize*GHOST_SIZE*(0.3-dy/
    1.5)), self.gridSize*GHOST_SIZE*0.2)
    moveCircle(eyes[1],(screen_x+self.gridSize*GHOST_SIZE*(0.3+dx/1.5), screen_y-self.gridSize*GHOST_SIZE*(0.3-dy/
    1.5)), self.gridSize*GHOST_SIZE*0.2)
    moveCircle(eyes[2],(screen_x+self.gridSize*GHOST_SIZE*(-0.3+dx), screen_y-self.gridSize*GHOST_SIZE*(0.3-dy)),
    self.gridSize*GHOST_SIZE*0.08)
    moveCircle(eyes[3],(screen_x+self.gridSize*GHOST_SIZE*(0.3+dx), screen_y-self.gridSize*GHOST_SIZE*(0.3-dy)),
    self.gridSize*GHOST_SIZE*0.08)

  def moveGhost(self, ghost, ghostIndex, prevGhost, ghostImageParts):
    old_x, old_y = self.to_screen(self.getPosition(prevGhost))
    new_x, new_y = self.to_screen(self.getPosition(ghost))
    delta = new_x - old_x, new_y - old_y

    for ghostImagePart in ghostImageParts:
      move_by(ghostImagePart, delta)
    refresh()

    if ghost.scaredTimer > 0:
      color = SCARED_COLOR
    else:
      color = GHOST_COLORS[ghostIndex]
    edit(ghostImageParts[0], ('fill', color), ('outline', color))
    self.moveEyes(self.getPosition(ghost), self.getDirection(ghost), ghostImageParts[-4:])
    refresh()

  def getPosition(self, agentState):
    if agentState.configuration == None: return (-1000, -1000)
    return agentState.getPosition()

  def getDirection(self, agentState):
    if agentState.configuration == None: return Directions.STOP
    return agentState.configuration.getDirection()

  def finish(self):
    end_graphics()

  def to_screen(self, point):
    ( x, y ) = point
    #y = self.height - y
    x = (x + 1)*self.gridSize
    y = (self.height  - y)*self.gridSize
    return ( x, y )

  # Fixes some TK issue with off-center circles
  def to_screen2(self, point):
    ( x, y ) = point
    #y = self.height - y
    x = (x + 1)*self.gridSize
    y = (self.height  - y)*self.gridSize
    return ( x, y )

  def drawWalls(self, wallMatrix):
    wallColor = WALL_COLOR
    for xNum, x in enumerate(wallMatrix):
```

```
      if self.capture and (xNum * 2) < wallMatrix.width: wallColor = TEAM_COLORS[0]
      if self.capture and (xNum * 2) >= wallMatrix.width: wallColor = TEAM_COLORS[1]

      for yNum, cell in enumerate(x):
        if cell: # There's a wall here
          pos = (xNum, yNum)
          screen = self.to_screen(pos)
          screen2 = self.to_screen2(pos)

          # draw each quadrant of the square based on adjacent walls
          wIsWall = self.isWall(xNum-1, yNum, wallMatrix)
          eIsWall = self.isWall(xNum+1, yNum, wallMatrix)
          nIsWall = self.isWall(xNum, yNum+1, wallMatrix)
          sIsWall = self.isWall(xNum, yNum-1, wallMatrix)
          nwIsWall = self.isWall(xNum-1, yNum+1, wallMatrix)
          swIsWall = self.isWall(xNum-1, yNum-1, wallMatrix)
          neIsWall = self.isWall(xNum+1, yNum+1, wallMatrix)
          seIsWall = self.isWall(xNum+1, yNum-1, wallMatrix)

          # NE quadrant
          if (not nIsWall) and (not eIsWall):
            # inner circle
            circle(screen2, WALL_RADIUS * self.gridSize, wallColor, wallColor, (0,91), 'arc')
          if (nIsWall) and (not eIsWall):
            # vertical line
            line(add(screen, (self.gridSize*WALL_RADIUS, 0)), add(screen, (self.gridSize*WALL_RADIUS, self.gridSize
            *(-0.5)-1)), wallColor)
          if (not nIsWall) and (eIsWall):
            # horizontal line
            line(add(screen, (0, self.gridSize*(-1)*WALL_RADIUS)), add(screen, (self.gridSize*0.5+1, self.gridSize
            *(-1)*WALL_RADIUS)), wallColor)
          if (nIsWall) and (eIsWall) and (not neIsWall):
            # outer circle
            circle(add(screen2, (self.gridSize*2*WALL_RADIUS, self.gridSize*(-2)*WALL_RADIUS)), WALL_RADIUS * self.
            gridSize-1, wallColor, wallColor, (180,271), 'arc')
            line(add(screen, (self.gridSize*2*WALL_RADIUS-1, self.gridSize*(-1)*WALL_RADIUS)), add(screen, (self.
            gridSize*0.5+1, self.gridSize*(-1)*WALL_RADIUS)), wallColor)
            line(add(screen, (self.gridSize*WALL_RADIUS, self.gridSize*(-2)*WALL_RADIUS+1)), add(screen, (self.
            gridSize*WALL_RADIUS, self.gridSize*(-0.5))), wallColor)

          # NW quadrant
          if (not nIsWall) and (not wIsWall):
            # inner circle
            circle(screen2, WALL_RADIUS * self.gridSize, wallColor, wallColor, (90,181), 'arc')
          if (nIsWall) and (not wIsWall):
            # vertical line
            line(add(screen, (self.gridSize*(-1)*WALL_RADIUS, 0)), add(screen, (self.gridSize*(-1)*WALL_RADIUS,
            self.gridSize*(-0.5)-1)), wallColor)
          if (not nIsWall) and (wIsWall):
            # horizontal line
            line(add(screen, (0, self.gridSize*(-1)*WALL_RADIUS)), add(screen, (self.gridSize*(-0.5)-1, self.
            gridSize*(-1)*WALL_RADIUS)), wallColor)
          if (nIsWall) and (wIsWall) and (not nwIsWall):
            # outer circle
            circle(add(screen2, (self.gridSize*(-2)*WALL_RADIUS, self.gridSize*(-2)*WALL_RADIUS)), WALL_RADIUS *
            self.gridSize-1, wallColor, wallColor, (270,361), 'arc')
            line(add(screen, (self.gridSize*(-2)*WALL_RADIUS+1, self.gridSize*(-1)*WALL_RADIUS)), add(screen, (self
            .gridSize*(-0.5), self.gridSize*(-1)*WALL_RADIUS)), wallColor)
            line(add(screen, (self.gridSize*(-1)*WALL_RADIUS, self.gridSize*(-2)*WALL_RADIUS+1)), add(screen, (self
            .gridSize*(-1)*WALL_RADIUS, self.gridSize*(-0.5))), wallColor)

          # SE quadrant
          if (not sIsWall) and (not eIsWall):
            # inner circle
            circle(screen2, WALL_RADIUS * self.gridSize, wallColor, wallColor, (270,361), 'arc')
          if (sIsWall) and (not eIsWall):
            # vertical line
            line(add(screen, (self.gridSize*WALL_RADIUS, 0)), add(screen, (self.gridSize*WALL_RADIUS, self.gridSize
            *(0.5)+1)), wallColor)
          if (not sIsWall) and (eIsWall):
            # horizontal line
            line(add(screen, (0, self.gridSize*(1)*WALL_RADIUS)), add(screen, (self.gridSize*0.5+1, self.gridSize*(
```



```
            1)*WALL_RADIUS)), wallColor)
          if (sIsWall) and (eIsWall) and (not seIsWall):
            # outer circle
            circle(add(screen2, (self.gridSize*2*WALL_RADIUS, self.gridSize*(2)*WALL_RADIUS)), WALL_RADIUS * self.
            gridSize-1, wallColor, wallColor, (90,181), 'arc')
            line(add(screen, (self.gridSize*2*WALL_RADIUS-1, self.gridSize*(1)*WALL_RADIUS)), add(screen, (self.
            gridSize*0.5, self.gridSize*(1)*WALL_RADIUS)), wallColor)
            line(add(screen, (self.gridSize*WALL_RADIUS, self.gridSize*(2)*WALL_RADIUS-1)), add(screen, (self.
            gridSize*WALL_RADIUS, self.gridSize*(0.5))), wallColor)

          # SW quadrant
          if (not sIsWall) and (not wIsWall):
            # inner circle
            circle(screen2, WALL_RADIUS * self.gridSize, wallColor, wallColor, (180,271), 'arc')
          if (sIsWall) and (not wIsWall):
            # vertical line
            line(add(screen, (self.gridSize*(-1)*WALL_RADIUS, 0)), add(screen, (self.gridSize*(-1)*WALL_RADIUS,
            self.gridSize*(0.5)+1)), wallColor)
          if (not sIsWall) and (wIsWall):
            # horizontal line
            line(add(screen, (0, self.gridSize*(1)*WALL_RADIUS)), add(screen, (self.gridSize*(-0.5)-1, self.
            gridSize*(1)*WALL_RADIUS)), wallColor)
          if (sIsWall) and (wIsWall) and (not swIsWall):
            # outer circle
            circle(add(screen2, (self.gridSize*(-2)*WALL_RADIUS, self.gridSize*(2)*WALL_RADIUS)), WALL_RADIUS *
            self.gridSize-1, wallColor, wallColor, (0,91), 'arc')
            line(add(screen, (self.gridSize*(-2)*WALL_RADIUS+1, self.gridSize*(1)*WALL_RADIUS)), add(screen, (self.
            gridSize*(-0.5), self.gridSize*(1)*WALL_RADIUS)), wallColor)
            line(add(screen, (self.gridSize*(-1)*WALL_RADIUS, self.gridSize*(2)*WALL_RADIUS-1)), add(screen, (self.
            gridSize*(-1)*WALL_RADIUS, self.gridSize*(0.5))), wallColor)

  def isWall(self, x, y, walls):
    if x < 0 or y < 0:
      return False
    if x >= walls.width or y >= walls.height:
      return False
    return walls[x][y]

  def drawFood(self, foodMatrix ):
    foodImages = []
    color = FOOD_COLOR
    for xNum, x in enumerate(foodMatrix):
      if self.capture and (xNum * 2) <= foodMatrix.width: color = TEAM_COLORS[0]
      if self.capture and (xNum * 2) > foodMatrix.width: color = TEAM_COLORS[1]
      imageRow = []
      foodImages.append(imageRow)
      for yNum, cell in enumerate(x):
        if cell: # There's food here
          screen = self.to_screen((xNum, yNum ))
          dot = circle( screen,
                        FOOD_SIZE * self.gridSize,
                        outlineColor = color, fillColor = color,
                        width = 1)
          imageRow.append(dot)
        else:
          imageRow.append(None)
    return foodImages

  def drawCapsules(self, capsules ):
    capsuleImages = {}
    for capsule in capsules:
      ( screen_x, screen_y ) = self.to_screen(capsule)
      dot = circle( (screen_x, screen_y),
                        CAPSULE_SIZE * self.gridSize,
                        outlineColor = CAPSULE_COLOR,
                        fillColor = CAPSULE_COLOR,
                        width = 1)
      capsuleImages[capsule] = dot
    return capsuleImages

  def removeFood(self, cell, foodImages ):
    x, y = cell
```

```python
    remove_from_screen(foodImages[x][y])

  def removeCapsule(self, cell, capsuleImages ):
    x, y = cell
    remove_from_screen(capsuleImages[(x, y)])

  def drawExpandedCells(self, cells):
    """
    Draws an overlay of expanded grid positions for search agents
    """
    n = float(len(cells))
    baseColor = [1.0, 0.0, 0.0]
    self.clearExpandedCells()
    self.expandedCells = []
    for k, cell in enumerate(cells):
       screenPos = self.to_screen( cell)
       cellColor = formatColor(*[(n-k) * c * .5 / n + .25 for c in baseColor])
       block = square(screenPos,
                0.5 * self.gridSize,
                color = cellColor,
                filled = 1, behind=2)
       self.expandedCells.append(block)
       if self.frameTime < 0:
         refresh()

  def clearExpandedCells(self):
    if 'expandedCells' in dir(self) and len(self.expandedCells) > 0:
      for cell in self.expandedCells:
        remove_from_screen(cell)


  def updateDistributions(self, distributions):
    "Draws an agent's belief distributions"
    if self.distributionImages == None:
      self.drawDistributions(self.previousState)
    for x in range(len(self.distributionImages)):
      for y in range(len(self.distributionImages[0])):
        image = self.distributionImages[x][y]
        weights = [dist[ (x,y) ] for dist in distributions]

        if sum(weights) != 0:
          pass
        # Fog of war
        color = [0.0,0.0,0.0]
        colors = GHOST_VEC_COLORS[1:] # With Pacman
        if self.capture: colors = GHOST_VEC_COLORS
        for weight, gcolor in zip(weights, colors):
          color = [min(1.0, c + 0.95 * g * weight ** .3) for c,g in zip(color, gcolor)]
        changeColor(image, formatColor(*color))
    refresh()

class FirstPersonPacmanGraphics(PacmanGraphics):
  def __init__(self, zoom = 1.0, showGhosts = True, capture = False, frameTime=0):
    PacmanGraphics.__init__(self, zoom, frameTime=frameTime)
    self.showGhosts = showGhosts
    self.capture = capture

  def initialize(self, state, isBlue = False):

    self.isBlue = isBlue
    PacmanGraphics.startGraphics(self, state)
    # Initialize distribution images
    walls = state.layout.walls
    dist = []
    self.layout = state.layout

    # Draw the rest
    self.distributionImages = None  # initialize lazily
    self.drawStaticObjects(state)
    self.drawAgentObjects(state)

    # Information
```



```python
    self.previousState = state

  def lookAhead(self, config, state):
    if config.getDirection() == 'Stop':
      return
    else:
      pass
      # Draw relevant ghosts
      allGhosts = state.getGhostStates()
      visibleGhosts = state.getVisibleGhosts()
      for i, ghost in enumerate(allGhosts):
        if ghost in visibleGhosts:
          self.drawGhost(ghost, i)
        else:
          self.currentGhostImages[i] = None

  def getGhostColor(self, ghost, ghostIndex):
    return GHOST_COLORS[ghostIndex]

  def getPosition(self, ghostState):
    if not self.showGhosts and not ghostState.isPacman and ghostState.getPosition()[1] > 1:
      return (-1000, -1000)
    else:
      return PacmanGraphics.getPosition(self, ghostState)

def add(x, y):
  return (x[0] + y[0], x[1] + y[1])


# Saving graphical output
# -----------------------
# Note: to make an animated gif from this postscript output, try the command:
# convert -delay 7 -loop 1 -compress lzw -layers optimize frame* out.gif
# convert is part of imagemagick (freeware)

SAVE_POSTSCRIPT = False
POSTSCRIPT_OUTPUT_DIR = 'frames'
FRAME_NUMBER = 0
import os

def saveFrame():
  "Saves the current graphical output as a postscript file"
  global SAVE_POSTSCRIPT, FRAME_NUMBER, POSTSCRIPT_OUTPUT_DIR
  if not SAVE_POSTSCRIPT: return
  if not os.path.exists(POSTSCRIPT_OUTPUT_DIR): os.mkdir(POSTSCRIPT_OUTPUT_DIR)
  name = os.path.join(POSTSCRIPT_OUTPUT_DIR, 'frame_%08d.ps' % FRAME_NUMBER)
  FRAME_NUMBER += 1
  writePostscript(name) # writes the current canvas
```

```python
# pacman.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

"""
Pacman.py holds the logic for the classic pacman game along with the main
code to run a game.  This file is divided into three sections:

  (i)  Your interface to the pacman world:
          Pacman is a complex environment.  You probably don't want to
          read through all of the code we wrote to make the game runs
          correctly.  This section contains the parts of the code
          that you will need to understand in order to complete the
          project.  There is also some code in game.py that you should
          understand.

  (ii)  The hidden secrets of pacman:
          This section contains all of the logic code that the pacman
          environment uses to decide who can move where, who dies when
          things collide, etc.  You shouldn't need to read this section
          of code, but you can if you want.

  (iii) Framework to start a game:
          The final section contains the code for reading the command
          you use to set up the game, then starting up a new game, along with
          linking in all the external parts (agent functions, graphics).
          Check this section out to see all the options available to you.

To play your first game, type 'python pacman.py' from the command line.
The keys are 'a', 's', 'd', and 'w' to move (or arrow keys).  Have fun!
"""
from game import GameStateData
from game import Game
from game import Directions
from game import Actions
from util import nearestPoint
from util import manhattanDistance
import util, layout
import sys, types, time, random, os

###################################################
# YOUR INTERFACE TO THE PACMAN WORLD: A GameState #
###################################################

class GameState:
  """
  A GameState specifies the full game state, including the food, capsules,
  agent configurations and score changes.

  GameStates are used by the Game object to capture the actual state of the game and
  can be used by agents to reason about the game.

  Much of the information in a GameState is stored in a GameStateData object.  We
  strongly suggest that you access that data via the accessor methods below rather
  than referring to the GameStateData object directly.

  Note that in classic Pacman, Pacman is always agent 0.
  """

  ####################################################
  # Accessor methods: use these to access state data #
  ####################################################

  # static variable keeps track of which states have had getLegalActions called
```



```python
  explored = set()
  def getAndResetExplored():
      tmp = GameState.explored.copy()
      GameState.explored = set()
      return tmp
  getAndResetExplored = staticmethod(getAndResetExplored)

  def getLegalActions( self, agentIndex=0 ):
    """
    Returns the legal actions for the agent specified.
    """
    GameState.explored.add(self)
    if self.isWin() or self.isLose(): return []

    if agentIndex == 0:  # Pacman is moving
      return PacmanRules.getLegalActions( self )
    else:
      return GhostRules.getLegalActions( self, agentIndex )

  def generateSuccessor( self, agentIndex, action):
    """
    Returns the successor state after the specified agent takes the action.
    """
    # Check that successors exist
    if self.isWin() or self.isLose(): raise Exception('Can\'t generate a successor of a terminal state.')

    # Copy current state
    state = GameState(self)

    # Let agent's logic deal with its action's effects on the board
    if agentIndex == 0:  # Pacman is moving
      state.data._eaten = [False for i in range(state.getNumAgents())]
      PacmanRules.applyAction( state, action )
    else:                # A ghost is moving
      GhostRules.applyAction( state, action, agentIndex )

    # Time passes
    if agentIndex == 0:
      state.data.scoreChange += -TIME_PENALTY # Penalty for waiting around
    else:
      GhostRules.decrementTimer( state.data.agentStates[agentIndex] )

    # Resolve multi-agent effects
    GhostRules.checkDeath( state, agentIndex )

    # Book keeping
    state.data._agentMoved = agentIndex
    state.data.score += state.data.scoreChange
    return state

  def getLegalPacmanActions( self ):
    return self.getLegalActions( 0 )

  def generatePacmanSuccessor( self, action ):
    """
    Generates the successor state after the specified pacman move
    """
    return self.generateSuccessor( 0, action )

  def getPacmanState( self ):
    """
    Returns an AgentState object for pacman (in game.py)

    state.pos gives the current position
    state.direction gives the travel vector
    """
    return self.data.agentStates[0].copy()
```

```
  def getPacmanPosition( self ):
    return self.data.agentStates[0].getPosition()

  def getGhostStates( self ):
    return self.data.agentStates[1:]

  def getGhostState( self, agentIndex ):
    if agentIndex == 0 or agentIndex >= self.getNumAgents():
      raise Exception("Invalid index passed to getGhostState")
    return self.data.agentStates[agentIndex]

  def getGhostPosition( self, agentIndex ):
    if agentIndex == 0:
      raise Exception("Pacman's index passed to getGhostPosition")
    return self.data.agentStates[agentIndex].getPosition()

  def getGhostPositions(self):
    return [s.getPosition() for s in self.getGhostStates()]

  def getNumAgents( self ):
    return len( self.data.agentStates )

  def getScore( self ):
    return self.data.score

  def getCapsules(self):
    """
    Returns a list of positions (x,y) of the remaining capsules.
    """
    return self.data.capsules

  def getNumFood( self ):
    return self.data.food.count()

  def getFood(self):
    """
    Returns a Grid of boolean food indicator variables.

    Grids can be accessed via list notation, so to check
    if there is food at (x,y), just call

    currentFood = state.getFood()
    if currentFood[x][y] == True: ...
    """
    return self.data.food

  def getWalls(self):
    """
    Returns a Grid of boolean wall indicator variables.

    Grids can be accessed via list notation, so to check
    if there is food at (x,y), just call

    walls = state.getWalls()
    if walls[x][y] == True: ...
    """
    return self.data.layout.walls

  def hasFood(self, x, y):
    return self.data.food[x][y]

  def hasWall(self, x, y):
    return self.data.layout.walls[x][y]

  def isLose( self ):
    return self.data._lose

  def isWin( self ):
```



```
    return self.data._win

  #############################################
  #             Helper methods:               #
  # You shouldn't need to call these directly #
  #############################################

  def __init__( self, prevState = None ):
    """
    Generates a new state by copying information from its predecessor.
    """
    if prevState != None: # Initial state
      self.data = GameStateData(prevState.data)
    else:
      self.data = GameStateData()

  def deepCopy( self ):
    state = GameState( self )
    state.data = self.data.deepCopy()
    return state

  def __eq__( self, other ):
    """
    Allows two states to be compared.
    """
    return self.data == other.data

  def __hash__( self ):
    """
    Allows states to be keys of dictionaries.
    """
    return hash( self.data )

  def __str__( self ):

    return str(self.data)

  def initialize( self, layout, numGhostAgents=1000 ):
    """
    Creates an initial game state from a layout array (see layout.py).
    """
    self.data.initialize(layout, numGhostAgents)

############################################################################
#                      THE HIDDEN SECRETS OF PACMAN                        #
#                                                                          #
# You shouldn't need to look through the code in this section of the file. #
############################################################################

SCARED_TIME = 40    # Moves ghosts are scared
COLLISION_TOLERANCE = 0.7 # How close ghosts must be to Pacman to kill
TIME_PENALTY = 1 # Number of points lost each round

class ClassicGameRules:
  """
  These game rules manage the control flow of a game, deciding when
  and how the game starts and ends.
  """
  def __init__(self, timeout=30):
    self.timeout = timeout

  def newGame( self, layout, pacmanAgent, ghostAgents, display, quiet = False, catchExceptions=False):
    agents = [pacmanAgent] + ghostAgents[:layout.getNumGhosts()]
    initState = GameState()
    initState.initialize( layout, len(ghostAgents) )
    game = Game(agents, display, self, catchExceptions=catchExceptions)
    game.state = initState
    self.initialState = initState.deepCopy()
```

```
    self.quiet = quiet
    return game

  def process(self, state, game):
    """
    Checks to see whether it is time to end the game.
    """
    if state.isWin(): self.win(state, game)
    if state.isLose(): self.lose(state, game)

  def win( self, state, game ):
    if not self.quiet: print "Pacman emerges victorious! Score: %d" % state.data.score
    game.gameOver = True

  def lose( self, state, game ):
    if not self.quiet: print "Pacman died! Score: %d" % state.data.score
    game.gameOver = True

  def getProgress(self, game):
    return float(game.state.getNumFood()) / self.initialState.getNumFood()

  def agentCrash(self, game, agentIndex):
    if agentIndex == 0:
      print "Pacman crashed"
    else:
      print "A ghost crashed"

  def getMaxTotalTime(self, agentIndex):
    return self.timeout

  def getMaxStartupTime(self, agentIndex):
    return self.timeout

  def getMoveWarningTime(self, agentIndex):
    return self.timeout

  def getMoveTimeout(self, agentIndex):
    return self.timeout

  def getMaxTimeWarnings(self, agentIndex):
    return 0

class PacmanRules:
  """
  These functions govern how pacman interacts with his environment under
  the classic game rules.
  """
  PACMAN_SPEED=1

  def getLegalActions( state ):
    """
    Returns a list of possible actions.
    """
    return Actions.getPossibleActions( state.getPacmanState().configuration, state.data.layout.walls )
  getLegalActions = staticmethod( getLegalActions )

  def applyAction( state, action ):
    """
    Edits the state to reflect the results of the action.
    """
    legal = PacmanRules.getLegalActions( state )
    if action not in legal:
      raise Exception("Illegal action " + str(action))

    pacmanState = state.data.agentStates[0]

    # Update Configuration
    vector = Actions.directionToVector( action, PacmanRules.PACMAN_SPEED )
```



```
    pacmanState.configuration = pacmanState.configuration.generateSuccessor( vector )

    # Eat
    next = pacmanState.configuration.getPosition()
    nearest = nearestPoint( next )
    if manhattanDistance( nearest, next ) <= 0.5 :
      # Remove food
      PacmanRules.consume( nearest, state )
  applyAction = staticmethod( applyAction )

  def consume( position, state ):
    x,y = position
    # Eat food
    if state.data.food[x][y]:
      state.data.scoreChange += 10
      state.data.food = state.data.food.copy()
      state.data.food[x][y] = False
      state.data._foodEaten = position
      # TODO: cache numFood?
      numFood = state.getNumFood()
      if numFood == 0 and not state.data._lose:
        state.data.scoreChange += 500
        state.data._win = True
    # Eat capsule
    if( position in state.getCapsules() ):
      state.data.capsules.remove( position )
      state.data._capsuleEaten = position
      # Reset all ghosts' scared timers
      for index in range( 1, len( state.data.agentStates ) ):
        state.data.agentStates[index].scaredTimer = SCARED_TIME
  consume = staticmethod( consume )

class GhostRules:
  """
  These functions dictate how ghosts interact with their environment.
  """
  GHOST_SPEED=1.0
  def getLegalActions( state, ghostIndex ):
    """
    Ghosts cannot stop, and cannot turn around unless they
    reach a dead end, but can turn 90 degrees at intersections.
    """
    conf = state.getGhostState( ghostIndex ).configuration
    possibleActions = Actions.getPossibleActions( conf, state.data.layout.walls )
    reverse = Actions.reverseDirection( conf.direction )
    if Directions.STOP in possibleActions:
      possibleActions.remove( Directions.STOP )
    if reverse in possibleActions and len( possibleActions ) > 1:
      possibleActions.remove( reverse )
    return possibleActions
  getLegalActions = staticmethod( getLegalActions )

  def applyAction( state, action, ghostIndex):

    legal = GhostRules.getLegalActions( state, ghostIndex )
    if action not in legal:
      raise Exception("Illegal ghost action " + str(action))

    ghostState = state.data.agentStates[ghostIndex]
    speed = GhostRules.GHOST_SPEED
    if ghostState.scaredTimer > 0: speed /= 2.0
    vector = Actions.directionToVector( action, speed )
    ghostState.configuration = ghostState.configuration.generateSuccessor( vector )
  applyAction = staticmethod( applyAction )

  def decrementTimer( ghostState):
    timer = ghostState.scaredTimer
    if timer == 1:
```

```python
      ghostState.configuration.pos = nearestPoint( ghostState.configuration.pos )
    ghostState.scaredTimer = max( 0, timer - 1 )
  decrementTimer = staticmethod( decrementTimer )

  def checkDeath( state, agentIndex):
    pacmanPosition = state.getPacmanPosition()
    if agentIndex == 0: # Pacman just moved; Anyone can kill him
      for index in range( 1, len( state.data.agentStates ) ):
        ghostState = state.data.agentStates[index]
        ghostPosition = ghostState.configuration.getPosition()
        if GhostRules.canKill( pacmanPosition, ghostPosition ):
          GhostRules.collide( state, ghostState, index )
    else:
      ghostState = state.data.agentStates[agentIndex]
      ghostPosition = ghostState.configuration.getPosition()
      if GhostRules.canKill( pacmanPosition, ghostPosition ):
        GhostRules.collide( state, ghostState, agentIndex )
  checkDeath = staticmethod( checkDeath )

  def collide( state, ghostState, agentIndex):
    if ghostState.scaredTimer > 0:
      state.data.scoreChange += 200
      GhostRules.placeGhost(state, ghostState)
      ghostState.scaredTimer = 0
      # Added for first-person
      state.data._eaten[agentIndex] = True
    else:
      if not state.data._win:
        state.data.scoreChange -= 500
        state.data._lose = True
  collide = staticmethod( collide )

  def canKill( pacmanPosition, ghostPosition ):
    return manhattanDistance( ghostPosition, pacmanPosition ) <= COLLISION_TOLERANCE
  canKill = staticmethod( canKill )

  def placeGhost(state, ghostState):
    ghostState.configuration = ghostState.start
  placeGhost = staticmethod( placeGhost )

#############################
# FRAMEWORK TO START A GAME #
#############################

def default(str):
  return str + ' [Default: %default]'

def parseAgentArgs(str):
  if str == None: return {}
  pieces = str.split(',')
  opts = {}
  for p in pieces:
    if '=' in p:
      key, val = p.split('=')
    else:
      key,val = p, 1
    opts[key] = val
  return opts

def readCommand( argv ):
  """
  Processes the command used to run pacman from the command line.
  """
  from optparse import OptionParser
  usageStr = """
  USAGE:      python pacman.py <options>
  EXAMPLES:   (1) python pacman.py
                  - starts an interactive game
```



```python
              (2) python pacman.py --layout smallClassic --zoom 2
              OR  python pacman.py -l smallClassic -z 2
                  - starts an interactive game on a smaller board, zoomed in
  """
  parser = OptionParser(usageStr)

  parser.add_option('-n', '--numGames', dest='numGames', type='int',
                    help=default('the number of GAMES to play'), metavar='GAMES', default=1)
  parser.add_option('-l', '--layout', dest='layout',
                    help=default('the LAYOUT_FILE from which to load the map layout'),
                    metavar='LAYOUT_FILE', default='mediumClassic')
  parser.add_option('-p', '--pacman', dest='pacman',
                    help=default('the agent TYPE in the pacmanAgents module to use'),
                    metavar='TYPE', default='KeyboardAgent')
  parser.add_option('-t', '--textGraphics', action='store_true', dest='textGraphics',
                    help='Display output as text only', default=False)
  parser.add_option('-q', '--quietTextGraphics', action='store_true', dest='quietGraphics',
                    help='Generate minimal output and no graphics', default=False)
  parser.add_option('-g', '--ghosts', dest='ghost',
                    help=default('the ghost agent TYPE in the ghostAgents module to use'),
                    metavar = 'TYPE', default='RandomGhost')
  parser.add_option('-k', '--numghosts', type='int', dest='numGhosts',
                    help=default('The maximum number of ghosts to use'), default=4)
  parser.add_option('-z', '--zoom', type='float', dest='zoom',
                    help=default('Zoom the size of the graphics window'), default=1.0)
  parser.add_option('-f', '--fixRandomSeed', action='store_true', dest='fixRandomSeed',
                    help='Fixes the random seed to always play the same game', default=False)
  parser.add_option('-r', '--recordActions', action='store_true', dest='record',
                    help='Writes game histories to a file (named by the time they were played)', default=
                    False)
  parser.add_option('--replay', dest='gameToReplay',
                    help='A recorded game file (pickle) to replay', default=None)
  parser.add_option('-a','--agentArgs',dest='agentArgs',
                    help='Comma separated values sent to agent. e.g. "opt1=val1,opt2,opt3=val3"')
  parser.add_option('-x', '--numTraining', dest='numTraining', type='int',
                    help=default('How many episodes are training (suppresses output)'), default=0)
  parser.add_option('--frameTime', dest='frameTime', type='float',
                    help=default('Time to delay between frames; <0 means keyboard'), default=0.1)
  parser.add_option('-c', '--catchExceptions', action='store_true', dest='catchExceptions',
                    help='Turns on exception handling and timeouts during games', default=False)
  parser.add_option('--timeout', dest='timeout', type='int',
                    help=default('Maximum length of time an agent can spend computing in a single game'),
                    default=30)

  options, otherjunk = parser.parse_args(argv)
  if len(otherjunk) != 0:
    raise Exception('Command line input not understood: ' + str(otherjunk))
  args = dict()

  # Fix the random seed
  if options.fixRandomSeed: random.seed('cs221')

  # Choose a layout
  args['layout'] = layout.getLayout( options.layout )
  if args['layout'] == None: raise Exception("The layout " + options.layout + " cannot be found")

  # Choose a Pacman agent
  noKeyboard = options.gameToReplay == None and (options.textGraphics or options.quietGraphics)
  pacmanType = loadAgent(options.pacman, noKeyboard)
  agentOpts = parseAgentArgs(options.agentArgs)
  if options.numTraining > 0:
    args['numTraining'] = options.numTraining
    if 'numTraining' not in agentOpts: agentOpts['numTraining'] = options.numTraining
  pacman = pacmanType(**agentOpts) # Instantiate Pacman with agentArgs
  args['pacman'] = pacman

  # Don't display training games
  if 'numTrain' in agentOpts:
```

```
    options.numQuiet = int(agentOpts['numTrain'])
    options.numIgnore = int(agentOpts['numTrain'])

  # Choose a ghost agent
  ghostType = loadAgent(options.ghost, noKeyboard)
  args['ghosts'] = [ghostType( i+1 ) for i in range( options.numGhosts )]

  # Choose a display format
  if options.quietGraphics:
      import textDisplay
      args['display'] = textDisplay.NullGraphics()
  elif options.textGraphics:
    import textDisplay
    textDisplay.SLEEP_TIME = options.frameTime
    args['display'] = textDisplay.PacmanGraphics()
  else:
    import graphicsDisplay
    args['display'] = graphicsDisplay.PacmanGraphics(options.zoom, frameTime = options.frameTime)
  args['numGames'] = options.numGames
  args['record'] = options.record
  args['catchExceptions'] = options.catchExceptions
  args['timeout'] = options.timeout

  # Special case: recorded games don't use the runGames method or args structure
  if options.gameToReplay != None:
    print 'Replaying recorded game %s.' % options.gameToReplay
    import cPickle
    f = open(options.gameToReplay)
    try: recorded = cPickle.load(f)
    finally: f.close()
    recorded['display'] = args['display']
    replayGame(**recorded)
    sys.exit(0)

  return args

def loadAgent(pacman, nographics):
  # Looks through all pythonPath Directories for the right module,
  pythonPathStr = os.path.expandvars("$PYTHONPATH")
  if pythonPathStr.find(';') == -1:
    pythonPathDirs = pythonPathStr.split(':')
  else:
    pythonPathDirs = pythonPathStr.split(';')
  pythonPathDirs.append('.')

  for moduleDir in pythonPathDirs:
    if not os.path.isdir(moduleDir): continue
    moduleNames = [f for f in os.listdir(moduleDir) if f.endswith('gents.py')]
    for modulename in moduleNames:
      try:
        module = __import__(modulename[:-3])
      except ImportError:
        continue
      if pacman in dir(module):
        if nographics and modulename == 'keyboardAgents.py':
          raise Exception('Using the keyboard requires graphics (not text display)')
        return getattr(module, pacman)
  raise Exception('The agent ' + pacman + ' is not specified in any *Agents.py.')

def replayGame( layout, actions, display ):
    import pacmanAgents, ghostAgents
    rules = ClassicGameRules()
    agents = [pacmanAgents.GreedyAgent()] + [ghostAgents.RandomGhost(i+1) for i in range(layout.getNumGhosts
    ())]
    game = rules.newGame( layout, agents[0], agents[1:], display )
    state = game.state
    display.initialize(state.data)
```



```
    for action in actions:
      # Execute the action
      state = state.generateSuccessor( *action )
      # Change the display
      display.update( state.data )
      # Allow for game specific conditions (winning, losing, etc.)
      rules.process(state, game)

    display.finish()

def runGames( layout, pacman, ghosts, display, numGames, record, numTraining = 0, catchExceptions=False,
timeout=30 ):
  import __main__
  __main__.__dict__['_display'] = display

  rules = ClassicGameRules(timeout)
  games = []

  for i in range( numGames ):
    beQuiet = i < numTraining
    if beQuiet:
        # Suppress output and graphics
        import textDisplay
        gameDisplay = textDisplay.NullGraphics()
        rules.quiet = True
    else:
        gameDisplay = display
        rules.quiet = False
    game = rules.newGame( layout, pacman, ghosts, gameDisplay, beQuiet, catchExceptions)
    game.run()
    if not beQuiet: games.append(game)

    if record:
      import time, cPickle
      fname = ('recorded-game-%d' % (i + 1)) +  '-'.join([str(t) for t in time.localtime()[1:6]])
      f = file(fname, 'w')
      components = {'layout': layout, 'actions': game.moveHistory}
      cPickle.dump(components, f)
      f.close()

  if (numGames-numTraining) > 0:
    scores = [game.state.getScore() for game in games]
    wins = [game.state.isWin() for game in games]
    winRate = wins.count(True)/ float(len(wins))
    print 'Average Score:', sum(scores) / float(len(scores))
    print 'Scores:       ', ', '.join([str(score) for score in scores])
    print 'Win Rate:      %d/%d (%.2f)' % (wins.count(True), len(wins), winRate)
    print 'Record:       ', ', '.join([ ['Loss', 'Win'][int(w)] for w in wins])

  return games

if __name__ == '__main__':
  """
  The main function called when pacman.py is run
  from the command line:

  > python pacman.py

  See the usage string for more details.

  > python pacman.py --help
  """
  args = readCommand( sys.argv[1:] ) # Get game components based on input
  runGames( **args )
  pass
```

```python
# game.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

from util import *
import time, os
import traceback

#######################
# Parts worth reading #
#######################

class Agent:
  """
  An agent must define a getAction method, but may also define the
  following methods which will be called if they exist:

  def registerInitialState(self, state): # inspects the starting state
  """
  def __init__(self, index=0):
    self.index = index

  def getAction(self, state):
    """
    The Agent will receive a GameState (from either {pacman, capture, sonar}.py) and
    must return an action from Directions.{North, South, East, West, Stop}
    """
    raiseNotDefined()

class Directions:
  NORTH = 'North'
  SOUTH = 'South'
  EAST = 'East'
  WEST = 'West'
  STOP = 'Stop'

  LEFT = {NORTH: WEST,
         SOUTH: EAST,
         EAST:  NORTH,
         WEST:  SOUTH,
         STOP:  STOP}

  RIGHT = dict([(y,x) for x, y in LEFT.items()])

  REVERSE = {NORTH: SOUTH,
             SOUTH: NORTH,
             EAST: WEST,
             WEST: EAST,
             STOP: STOP}

class Configuration:
  """
  A Configuration holds the (x,y) coordinate of a character, along with its
  traveling direction.

  The convention for positions, like a graph, is that (0,0) is the lower left corner, x increases
  horizontally and y increases vertically.  Therefore, north is the direction of increasing y, or (0,1).
  """

  def __init__(self, pos, direction):
    self.pos = pos
    self.direction = direction

  def getPosition(self):
    return (self.pos)

  def getDirection(self):
    return self.direction
```



```python
  def isInteger(self):
    x,y = self.pos
    return x == int(x) and y == int(y)

  def __eq__(self, other):
    if other == None: return False
    return (self.pos == other.pos and self.direction == other.direction)

  def __hash__(self):
    x = hash(self.pos)
    y = hash(self.direction)
    return hash(x + 13 * y)

  def __str__(self):
    return "(x,y)="+str(self.pos)+", "+str(self.direction)

  def generateSuccessor(self, vector):
    """
    Generates a new configuration reached by translating the current
    configuration by the action vector.  This is a low-level call and does
    not attempt to respect the legality of the movement.

    Actions are movement vectors.
    """
    x, y= self.pos
    dx, dy = vector
    direction = Actions.vectorToDirection(vector)
    if direction == Directions.STOP:
      direction = self.direction # There is no stop direction
    return Configuration((x + dx, y+dy), direction)

class AgentState:
  """
  AgentStates hold the state of an agent (configuration, speed, scared, etc).
  """

  def __init__( self, startConfiguration, isPacman ):
    self.start = startConfiguration
    self.configuration = startConfiguration
    self.isPacman = isPacman
    self.scaredTimer = 0

  def __str__( self ):
    if self.isPacman:
      return "Pacman: " + str( self.configuration )
    else:
      return "Ghost: " + str( self.configuration )

  def __eq__( self, other ):
    if other == None:
      return False
    return self.configuration == other.configuration and self.scaredTimer == other.scaredTimer

  def __hash__(self):
    return hash(hash(self.configuration) + 13 * hash(self.scaredTimer))

  def copy( self ):
    state = AgentState( self.start, self.isPacman )
    state.configuration = self.configuration
    state.scaredTimer = self.scaredTimer
    return state

  def getPosition(self):
    if self.configuration == None: return None
    return self.configuration.getPosition()

  def getDirection(self):
    return self.configuration.getDirection()

class Grid:
  """
```

```
  A 2-dimensional array of objects backed by a list of lists.  Data is accessed
  via grid[x][y] where (x,y) are positions on a Pacman map with x horizontal,
  y vertical and the origin (0,0) in the bottom left corner.

  The __str__ method constructs an output that is oriented like a pacman board.
  """
  def __init__(self, width, height, initialValue=False, bitRepresentation=None):
    if initialValue not in [False, True]: raise Exception('Grids can only contain booleans')
    self.CELLS_PER_INT = 30

    self.width = width
    self.height = height
    self.data = [[initialValue for y in range(height)] for x in range(width)]
    if bitRepresentation:
      self._unpackBits(bitRepresentation)

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

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

  def __str__(self):
    out = [[str(self.data[x][y])[0] for x in range(self.width)] for y in range(self.height)]
    out.reverse()
    return '\n'.join([''.join(x) for x in out])

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

  def __hash__(self):
    # return hash(str(self))
    base = 1
    h = 0
    for l in self.data:
      for i in l:
        if i:
          h += base
        base *= 2
    return hash(h)

  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 count(self, item =True ):
    return sum([x.count(item) for x in self.data])

  def asList(self, key = True):
    list = []
    for x in range(self.width):
      for y in range(self.height):
        if self[x][y] == key: list.append( (x,y) )
    return list

  def packBits(self):
    """
    Returns an efficient int list representation

    (width, height, bitPackedInts...)
    """
    bits = [self.width, self.height]
```



```
    currentInt = 0
    for i in range(self.height * self.width):
      bit = self.CELLS_PER_INT - (i % self.CELLS_PER_INT) - 1
      x, y = self._cellIndexToPosition(i)
      if self[x][y]:
        currentInt += 2 ** bit
      if (i + 1) % self.CELLS_PER_INT == 0:
        bits.append(currentInt)
        currentInt = 0
    bits.append(currentInt)
    return tuple(bits)

  def _cellIndexToPosition(self, index):
    x = index / self.height
    y = index % self.height
    return x, y

  def _unpackBits(self, bits):
    """
    Fills in data from a bit-level representation
    """
    cell = 0
    for packed in bits:
      for bit in self._unpackInt(packed, self.CELLS_PER_INT):
        if cell == self.width * self.height: break
        x, y = self._cellIndexToPosition(cell)
        self[x][y] = bit
        cell += 1

  def _unpackInt(self, packed, size):
    bools = []
    if packed < 0: raise ValueError, "must be a positive integer"
    for i in range(size):
      n = 2 ** (self.CELLS_PER_INT - i - 1)
      if packed >= n:
        bools.append(True)
        packed -= n
      else:
        bools.append(False)
    return bools

def reconstituteGrid(bitRep):
  if type(bitRep) is not type((1,2)):
    return bitRep
  width, height = bitRep[:2]
  return Grid(width, height, bitRepresentation= bitRep[2:])

####################################
# Parts you shouldn't have to read #
####################################

class Actions:
  """
  A collection of static methods for manipulating move actions.
  """
  # Directions
  _directions = {Directions.NORTH: (0, 1),
                 Directions.SOUTH: (0, -1),
                 Directions.EAST:  (1, 0),
                 Directions.WEST:  (-1, 0),
                 Directions.STOP:  (0, 0)}

  _directionsAsList = _directions.items()

  TOLERANCE = .001

  def reverseDirection(action):
    if action == Directions.NORTH:
      return Directions.SOUTH
    if action == Directions.SOUTH:
      return Directions.NORTH
    if action == Directions.EAST:
```

```python
            return Directions.WEST
        if action == Directions.WEST:
            return Directions.EAST
        return action
    reverseDirection = staticmethod(reverseDirection)

    def vectorToDirection(vector):
        dx, dy = vector
        if dy > 0:
            return Directions.NORTH
        if dy < 0:
            return Directions.SOUTH
        if dx < 0:
            return Directions.WEST
        if dx > 0:
            return Directions.EAST
        return Directions.STOP
    vectorToDirection = staticmethod(vectorToDirection)

    def directionToVector(direction, speed = 1.0):
        dx, dy =  Actions._directions[direction]
        return (dx * speed, dy * speed)
    directionToVector = staticmethod(directionToVector)

    def getPossibleActions(config, walls):
        possible = []
        x, y = config.pos
        x_int, y_int = int(x + 0.5), int(y + 0.5)

        # In between grid points, all agents must continue straight
        if (abs(x - x_int) + abs(y - y_int)  > Actions.TOLERANCE):
            return [config.getDirection()]

        for dir, vec in Actions._directionsAsList:
            dx, dy = vec
            next_y = y_int + dy
            next_x = x_int + dx
            if not walls[next_x][next_y]: possible.append(dir)

        return possible

    getPossibleActions = staticmethod(getPossibleActions)

    def getLegalNeighbors(position, walls):
        x,y = position
        x_int, y_int = int(x + 0.5), int(y + 0.5)
        neighbors = []
        for dir, vec in Actions._directionsAsList:
            dx, dy = vec
            next_x = x_int + dx
            if next_x < 0 or next_x == walls.width: continue
            next_y = y_int + dy
            if next_y < 0 or next_y == walls.height: continue
            if not walls[next_x][next_y]: neighbors.append((next_x, next_y))
        return neighbors
    getLegalNeighbors = staticmethod(getLegalNeighbors)

    def getSuccessor(position, action):
        dx, dy = Actions.directionToVector(action)
        x, y = position
        return (x + dx, y + dy)
    getSuccessor = staticmethod(getSuccessor)

class GameStateData:
    """

    """
    def __init__( self, prevState = None ):
        """
        Generates a new data packet by copying information from its predecessor.
        """
        if prevState != None:
```



```python
            self.food = prevState.food.shallowCopy()
            self.capsules = prevState.capsules[:]
            self.agentStates = self.copyAgentStates( prevState.agentStates )
            self.layout = prevState.layout
            self._eaten = prevState._eaten
            self.score = prevState.score
        self._foodEaten = None
        self._capsuleEaten = None
        self._agentMoved = None
        self._lose = False
        self._win = False
        self.scoreChange = 0

    def deepCopy( self ):
        state = GameStateData( self )
        state.food = self.food.deepCopy()
        state.layout = self.layout.deepCopy()
        state._agentMoved = self._agentMoved
        state._foodEaten = self._foodEaten
        state._capsuleEaten = self._capsuleEaten
        return state

    def copyAgentStates( self, agentStates ):
        copiedStates = []
        for agentState in agentStates:
            copiedStates.append( agentState.copy() )
        return copiedStates

    def __eq__( self, other ):
        """
        Allows two states to be compared.
        """
        if other == None: return False
        # TODO Check for type of other
        if not self.agentStates == other.agentStates: return False
        if not self.food == other.food: return False
        if not self.capsules == other.capsules: return False
        if not self.score == other.score: return False
        return True

    def __hash__( self ):
        """
        Allows states to be keys of dictionaries.
        """
        for i, state in enumerate( self.agentStates ):
            try:
                int(hash(state))
            except TypeError, e:
                print e
                #hash(state)
        return int((hash(tuple(self.agentStates)) + 13*hash(self.food) + 113* hash(tuple(self.capsules)) + 7 * hash(
        self.score)) % 1048575 )

    def __str__( self ):
        width, height = self.layout.width, self.layout.height
        map = Grid(width, height)
        if type(self.food) == type((1,2)):
            self.food = reconstituteGrid(self.food)
        for x in range(width):
            for y in range(height):
                food, walls = self.food, self.layout.walls
                map[x][y] = self._foodWallStr(food[x][y], walls[x][y])

        for agentState in self.agentStates:
            if agentState == None: continue
            if agentState.configuration == None: continue
            x,y = [int( i ) for i in nearestPoint( agentState.configuration.pos )]
            agent_dir = agentState.configuration.direction
            if agentState.isPacman:
                map[x][y] = self._pacStr( agent_dir )
            else:
                map[x][y] = self._ghostStr( agent_dir )
```

```
    for x, y in self.capsules:
      map[x][y] = 'o'

    return str(map) + ("\nScore: %d\n" % self.score)

  def _foodWallStr( self, hasFood, hasWall ):
    if hasFood:
      return '.'
    elif hasWall:
      return '%'
    else:
      return ' '

  def _pacStr( self, dir ):
    if dir == Directions.NORTH:
      return 'v'
    if dir == Directions.SOUTH:
      return '^'
    if dir == Directions.WEST:
      return '>'
    return '<'

  def _ghostStr( self, dir ):
    return 'G'
    if dir == Directions.NORTH:
      return 'M'
    if dir == Directions.SOUTH:
      return 'W'
    if dir == Directions.WEST:
      return '3'
    return 'E'

  def initialize( self, layout, numGhostAgents ):
    """
    Creates an initial game state from a layout array (see layout.py).
    """
    self.food = layout.food.copy()
    self.capsules = layout.capsules[:]
    self.layout = layout
    self.score = 0
    self.scoreChange = 0

    self.agentStates = []
    numGhosts = 0
    for isPacman, pos in layout.agentPositions:
      if not isPacman:
        if numGhosts == numGhostAgents: continue # Max ghosts reached already
        else: numGhosts += 1
      self.agentStates.append( AgentState( Configuration( pos, Directions.STOP), isPacman) )
    self._eaten = [False for a in self.agentStates]

class Game:
  """
  The Game manages the control flow, soliciting actions from agents.
  """

  def __init__( self, agents, display, rules, startingIndex=0, muteAgents=False, catchExceptions=False ):
    self.agentCrashed = False
    self.agents = agents
    self.display = display
    self.rules = rules
    self.startingIndex = startingIndex
    self.gameOver = False
    self.muteAgents = muteAgents
    self.catchExceptions = catchExceptions
    self.moveHistory = []
    self.totalAgentTimes = [0 for agent in agents]
    self.totalAgentTimeWarnings = [0 for agent in agents]
    self.agentTimeout = False
    import cStringIO
    self.agentOutput = [cStringIO.StringIO() for agent in agents]
```



```
  def getProgress(self):
    if self.gameOver:
      return 1.0
    else:
      return self.rules.getProgress(self)

  def _agentCrash( self, agentIndex, quiet=False):
    "Helper method for handling agent crashes"
    if not quiet: traceback.print_exc()
    self.gameOver = True
    self.agentCrashed = True
    self.rules.agentCrash(self, agentIndex)

  OLD_STDOUT = None
  OLD_STDERR = None

  def mute(self, agentIndex):
    if not self.muteAgents: return
    global OLD_STDOUT, OLD_STDERR
    import cStringIO
    OLD_STDOUT = sys.stdout
    OLD_STDERR = sys.stderr
    sys.stdout = self.agentOutput[agentIndex]
    sys.stderr = self.agentOutput[agentIndex]

  def unmute(self):
    if not self.muteAgents: return
    global OLD_STDOUT, OLD_STDERR
    # Revert stdout/stderr to originals
    sys.stdout = OLD_STDOUT
    sys.stderr = OLD_STDERR


  def run( self ):
    """
    Main control loop for game play.
    """
    self.display.initialize(self.state.data)
    self.numMoves = 0

    ###self.display.initialize(self.state.makeObservation(1).data)
    # inform learning agents of the game start
    for i in range(len(self.agents)):
      agent = self.agents[i]
      if not agent:
        self.mute(i)
        # this is a null agent, meaning it failed to load
        # the other team wins
        print "Agent %d failed to load" % i
        self.unmute()
        self._agentCrash(i, quiet=True)
        return
      if ("registerInitialState" in dir(agent)):
        self.mute(i)
        if self.catchExceptions:
          try:
            timed_func = TimeoutFunction(agent.registerInitialState, int(self.rules.getMaxStartupTime(i)))
            try:
              start_time = time.time()
              timed_func(self.state.deepCopy())
              time_taken = time.time() - start_time
              self.totalAgentTimes[i] += time_taken
            except TimeoutFunctionException:
              print "Agent %d ran out of time on startup!" % i
              self.unmute()
              self.agentTimeout = True
              self._agentCrash(i, quiet=True)
              return
          except Exception,data:
            self._agentCrash(i, quiet=False)
            self.unmute()
```

```
          return
        else:
          agent.registerInitialState(self.state.deepCopy())
        ## TODO: could this exceed the total time
        self.unmute()

    agentIndex = self.startingIndex
    numAgents = len( self.agents )

    while not self.gameOver:
      # Fetch the next agent
      agent = self.agents[agentIndex]
      move_time = 0
      skip_action = False
      # Generate an observation of the state
      if 'observationFunction' in dir( agent ):
        self.mute(agentIndex)
        if self.catchExceptions:
          try:
            timed_func = TimeoutFunction(agent.observationFunction, int(self.rules.getMoveTimeout(agentIndex)))
            try:
              start_time = time.time()
              observation = timed_func(self.state.deepCopy())
            except TimeoutFunctionException:
              skip_action = True
            move_time += time.time() - start_time
            self.unmute()
          except Exception,data:
            self._agentCrash(agentIndex, quiet=False)
            self.unmute()
            return
        else:
          observation = agent.observationFunction(self.state.deepCopy())
        self.unmute()
      else:
        observation = self.state.deepCopy()

      # Solicit an action
      action = None
      self.mute(agentIndex)
      if self.catchExceptions:
        try:
          timed_func = TimeoutFunction(agent.getAction, int(self.rules.getMoveTimeout(agentIndex)) - int(move_time))
          try:
            start_time = time.time()
            if skip_action:
              raise TimeoutFunctionException()
            action = timed_func( observation )
          except TimeoutFunctionException:
            print "Agent %d timed out on a single move!" % agentIndex
            self.agentTimeout = True
            self._agentCrash(agentIndex, quiet=True)
            self.unmute()
            return

          move_time += time.time() - start_time

          if move_time > self.rules.getMoveWarningTime(agentIndex):
            self.totalAgentTimeWarnings[agentIndex] += 1
            print "Agent %d took too long to make a move! This is warning %d" % (agentIndex, self.
            totalAgentTimeWarnings[agentIndex])
            if self.totalAgentTimeWarnings[agentIndex] > self.rules.getMaxTimeWarnings(agentIndex):
              print "Agent %d exceeded the maximum number of warnings: %d" % (agentIndex, self.
              totalAgentTimeWarnings[agentIndex])
              self.agentTimeout = True
              self._agentCrash(agentIndex, quiet=True)
              self.unmute()

          self.totalAgentTimes[agentIndex] += move_time
          #print "Agent: %d, time: %f, total: %f" % (agentIndex, move_time, self.totalAgentTimes[agentIndex])
          if self.totalAgentTimes[agentIndex] > self.rules.getMaxTotalTime(agentIndex):
            print "Agent %d ran out of time! (time: %1.2f)" % (agentIndex, self.totalAgentTimes[agentIndex])
```



```
              self.agentTimeout = True
              self._agentCrash(agentIndex, quiet=True)
              self.unmute()
              return
            self.unmute()
          except Exception,data:
            self._agentCrash(agentIndex)
            self.unmute()
            return
        else:
          action = agent.getAction(observation)
        self.unmute()

        # Execute the action
        self.moveHistory.append( (agentIndex, action) )
        if self.catchExceptions:
          try:
            self.state = self.state.generateSuccessor( agentIndex, action )
          except Exception,data:
            self.mute(agentIndex)
            self._agentCrash(agentIndex)
            self.unmute()
            return
        else:
          self.state = self.state.generateSuccessor( agentIndex, action )

        # Change the display
        self.display.update( self.state.data )
        ###idx = agentIndex - agentIndex % 2 + 1
        ###self.display.update( self.state.makeObservation(idx).data )

        # Allow for game specific conditions (winning, losing, etc.)
        self.rules.process(self.state, self)
        # Track progress
        if agentIndex == numAgents + 1: self.numMoves += 1
        # Next agent
        agentIndex = ( agentIndex + 1 ) % numAgents

      # inform a learning agent of the game result
      for agentIndex, agent in enumerate(self.agents):
        if "final" in dir( agent ) :
          try:
            self.mute(agentIndex)
            agent.final( self.state )
            self.unmute()
          except Exception,data:
            if not self.catchExceptions: raise
            self._agentCrash(agentIndex)
            self.unmute()
            return
      self.display.finish()
```

```python
# searchAgents.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

"""
This file contains all of the agents that can be selected to
control Pacman.  To select an agent, use the '-p' option
when running pacman.py.  Arguments can be passed to your agent
using '-a'.  For example, to load a SearchAgent that uses
depth first search (dfs), run the following command:

> python pacman.py -p SearchAgent -a searchFunction=depthFirstSearch

Commands to invoke other search strategies can be found in the
project description.

Please only change the parts of the file you are asked to.
Look for the lines that say

"*** YOUR CODE HERE ***"

The parts you fill in start about 3/4 of the way down.  Follow the
project description for details.

Good luck and happy searching!
"""
from game import Directions
from game import Agent
from game import Actions
import util
import time
import search
# import self to use dir()
import searchAgents

class GoWestAgent(Agent):
  "An agent that goes West until it can't."

  def getAction(self, state):
    "The agent receives a GameState (defined in pacman.py)."
    if Directions.WEST in state.getLegalPacmanActions():
      return Directions.WEST
    else:
      return Directions.STOP

#######################################################
# This portion is written for you, but will only work #
#       after you fill in parts of search.py          #
#######################################################

class SearchAgent(Agent):
  """
  This very general search agent finds a path using a supplied search algorithm for a
  supplied search problem, then returns actions to follow that path.

  As a default, this agent runs DFS on a PositionSearchProblem to find location (1,1)

  Options for fn include:
    depthFirstSearch or dfs
    breadthFirstSearch or bfs


  Note: You should NOT change any code in SearchAgent
  """

  def __init__(self, fn='depthFirstSearch', prob='PositionSearchProblem', heuristic='nullHeuristic'):
    # Warning: some advanced Python magic is employed below to find the right functions and problems
```



```python
    # Get the search function from the name and heuristic
    if fn not in dir(search):
      raise AttributeError, fn + ' is not a search function in search.py.'
    func = getattr(search, fn)
    if 'heuristic' not in func.func_code.co_varnames:
      print('[SearchAgent] using function ' + fn)
      self.searchFunction = func
    else:
      if heuristic in dir(searchAgents):
        heur = getattr(searchAgents, heuristic)
      elif heuristic in dir(search):
        heur = getattr(search, heuristic)
      else:
        raise AttributeError, heuristic + ' is not a function in searchAgents.py or search.py.'
      print('[SearchAgent] using function %s and heuristic %s' % (fn, heuristic))
      # Note: this bit of Python trickery combines the search algorithm and the heuristic
      self.searchFunction = lambda x: func(x, heuristic=heur)

    # Get the search problem type from the name
    if prob not in dir(searchAgents) or not prob.endswith('Problem'):
      raise AttributeError, prob + ' is not a search problem type in SearchAgents.py.'
    self.searchType = getattr(searchAgents, prob)
    print('[SearchAgent] using problem type ' + prob)

  def registerInitialState(self, state):
    """
    This is the first time that the agent sees the layout of the game board. Here, we
    choose a path to the goal.  In this phase, the agent should compute the path to the
    goal and store it in a local variable.  All of the work is done in this method!

    state: a GameState object (pacman.py)
    """
    if self.searchFunction == None: raise Exception, "No search function provided for SearchAgent"
    starttime = time.time()
    problem = self.searchType(state) # Makes a new search problem
    self.actions  = self.searchFunction(problem) # Find a path
    totalCost = problem.getCostOfActions(self.actions)
    print('Path found with total cost of %d in %.1f seconds' % (totalCost, time.time() - starttime))
    if '_expanded' in dir(problem): print('Search nodes expanded: %d' % problem._expanded)

  def getAction(self, state):
    """
    Returns the next action in the path chosen earlier (in registerInitialState).  Return
    Directions.STOP if there is no further action to take.

    state: a GameState object (pacman.py)
    """
    if 'actionIndex' not in dir(self): self.actionIndex = 0
    i = self.actionIndex
    self.actionIndex += 1
    if i < len(self.actions):
      return self.actions[i]
    else:
      return Directions.STOP

class PositionSearchProblem(search.SearchProblem):
  """
  A search problem defines the state space, start state, goal test,
  successor function and cost function.  This search problem can be
  used to find paths to a particular point on the pacman board.

  The state space consists of (x,y) positions in a pacman game.

  Note: this search problem is fully specified; you should NOT change it.
  """

  def __init__(self, gameState, costFn = lambda x: 1, goal=(1,1), start=None, warn=True):
    """
    Stores the start and goal.

    gameState: A GameState object (pacman.py)
    costFn: A function from a search state (tuple) to a non-negative number
```

```
    goal: A position in the gameState
    """
    self.walls = gameState.getWalls()
    self.startState = gameState.getPacmanPosition()
    if start != None: self.startState = start
    self.goal = goal
    self.costFn = costFn
    if warn and (gameState.getNumFood() != 1 or not gameState.hasFood(*goal)):
      print 'Warning: this does not look like a regular search maze'

    # For display purposes
    self._visited, self._visitedlist, self._expanded = {}, [], 0

  def getStartState(self):
    return self.startState

  def isGoalState(self, state):
     isGoal = state == self.goal

     # For display purposes only
     if isGoal:
       self._visitedlist.append(state)
       import __main__
       if '_display' in dir(__main__):
         if 'drawExpandedCells' in dir(__main__._display): #@UndefinedVariable
           __main__._display.drawExpandedCells(self._visitedlist) #@UndefinedVariable

     return isGoal

  def getSuccessors(self, state):
    """
    Returns successor states, the actions they require, and a cost of 1.

     As noted in search.py:
         For a given state, this should return a list of triples,
     (successor, action, stepCost), where 'successor' is a
     successor to the current state, 'action' is the action
     required to get there, and 'stepCost' is the incremental
     cost of expanding to that successor
    """

    successors = []
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
      x,y = state
      dx, dy = Actions.directionToVector(action)
      nextx, nexty = int(x + dx), int(y + dy)
      if not self.walls[nextx][nexty]:
        nextState = (nextx, nexty)
        cost = self.costFn(nextState)
        successors.append( ( nextState, action, cost) )

    # Bookkeeping for display purposes
    self._expanded += 1
    if state not in self._visited:
      self._visited[state] = True
      self._visitedlist.append(state)

    return successors

  def getCostOfActions(self, actions):
    """
    Returns the cost of a particular sequence of actions.  If those actions
    include an illegal move, return 999999
    """
    if actions == None: return 999999
    x,y= self.getStartState()
    cost = 0
    for action in actions:
      # Check figure out the next state and see whether its' legal
      dx, dy = Actions.directionToVector(action)
      x, y = int(x + dx), int(y + dy)
      if self.walls[x][y]: return 999999
```



```
      cost += self.costFn((x,y))
    return cost

class StayEastSearchAgent(SearchAgent):
  """
  An agent for position search with a cost function that penalizes being in
  positions on the West side of the board.

  The cost function for stepping into a position (x,y) is 1/2^x.
  """
  def __init__(self):
      self.searchFunction = search.uniformCostSearch
      costFn = lambda pos: .5 ** pos[0]
      self.searchType = lambda state: PositionSearchProblem(state, costFn)

class StayWestSearchAgent(SearchAgent):
  """
  An agent for position search with a cost function that penalizes being in
  positions on the East side of the board.

  The cost function for stepping into a position (x,y) is 2^x.
  """
  def __init__(self):
      self.searchFunction = search.uniformCostSearch
      costFn = lambda pos: 2 ** pos[0]
      self.searchType = lambda state: PositionSearchProblem(state, costFn)

def manhattanHeuristic(position, problem, info={}):
  "The Manhattan distance heuristic for a PositionSearchProblem"
  xy1 = position
  xy2 = problem.goal
  return abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])

def euclideanHeuristic(position, problem, info={}):
  "The Euclidean distance heuristic for a PositionSearchProblem"
  xy1 = position
  xy2 = problem.goal
  return ( (xy1[0] - xy2[0]) ** 2 + (xy1[1] - xy2[1]) ** 2 ) ** 0.5

#####################################################
# This portion is incomplete.  Time to write code!  #
#####################################################

class CornersProblem(search.SearchProblem):
  """
  This search problem finds paths through all four corners of a layout.

  You must select a suitable state space and successor function
  """

  def __init__(self, startingGameState):
    """
    Stores the walls, pacman's starting position and corners.
    """
    self.walls = startingGameState.getWalls()
    self.startingPosition = startingGameState.getPacmanPosition()
    top, right = self.walls.height-2, self.walls.width-2
    self.corners = ((1,1), (1,top), (right, 1), (right, top))
    for corner in self.corners:
      if not startingGameState.hasFood(*corner):
        print 'Warning: no food in corner ' + str(corner)
    self._expanded = 0 # Number of search nodes expanded

    "*** YOUR CODE HERE ***"

  def getStartState(self):
    "Returns the start state (in your state space, not the full Pacman state space)"
    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()

  def isGoalState(self, state):
    "Returns whether this search state is a goal state of the problem"
```

```
    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()

  def getSuccessors(self, state):
    """
    Returns successor states, the actions they require, and a cost of 1.

     As noted in search.py:
         For a given state, this should return a list of triples,
     (successor, action, stepCost), where 'successor' is a
     successor to the current state, 'action' is the action
     required to get there, and 'stepCost' is the incremental
     cost of expanding to that successor
    """

    successors = []
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
      # Add a successor state to the successor list if the action is legal
      # Here's a code snippet for figuring out whether a new position hits a wall:
      #   x,y = currentPosition
      #   dx, dy = Actions.directionToVector(action)
      #   nextx, nexty = int(x + dx), int(y + dy)
      #   hitsWall = self.walls[nextx][nexty]

      "*** YOUR CODE HERE ***"

    self._expanded += 1
    return successors

  def getCostOfActions(self, actions):
    """
    Returns the cost of a particular sequence of actions.  If those actions
    include an illegal move, return 999999.  This is implemented for you.
    """
    if actions == None: return 999999
    x,y= self.startingPosition
    for action in actions:
      dx, dy = Actions.directionToVector(action)
      x, y = int(x + dx), int(y + dy)
      if self.walls[x][y]: return 999999
    return len(actions)


def cornersHeuristic(state, problem):
  """
  A heuristic for the CornersProblem that you defined.

    state:   The current search state
             (a data structure you chose in your search problem)

    problem: The CornersProblem instance for this layout.

  This function should always return a number that is a lower bound
  on the shortest path from the state to a goal of the problem; i.e.
  it should be admissible (as well as consistent).
  """
  corners = problem.corners # These are the corner coordinates
  walls = problem.walls # These are the walls of the maze, as a Grid (game.py)

  "*** YOUR CODE HERE ***"
  return 0 # Default to trivial solution

class AStarCornersAgent(SearchAgent):
  "A SearchAgent for FoodSearchProblem using A* and your foodHeuristic"
  def __init__(self):
    self.searchFunction = lambda prob: search.aStarSearch(prob, cornersHeuristic)
    self.searchType = CornersProblem

class FoodSearchProblem:
  """
  A search problem associated with finding the a path that collects all of the
  food (dots) in a Pacman game.
```



```
  A search state in this problem is a tuple ( pacmanPosition, foodGrid ) where
    pacmanPosition: a tuple (x,y) of integers specifying Pacman's position
    foodGrid:       a Grid (see game.py) of either True or False, specifying remaining food
  """
  def __init__(self, startingGameState):
    self.start = (startingGameState.getPacmanPosition(), startingGameState.getFood())
    self.walls = startingGameState.getWalls()
    self.startingGameState = startingGameState
    self._expanded = 0
    self.heuristicInfo = {} # A dictionary for the heuristic to store information

  def getStartState(self):
    return self.start

  def isGoalState(self, state):
    return state[1].count() == 0

  def getSuccessors(self, state):
    "Returns successor states, the actions they require, and a cost of 1."
    successors = []
    self._expanded += 1
    for direction in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
      x,y = state[0]
      dx, dy = Actions.directionToVector(direction)
      nextx, nexty = int(x + dx), int(y + dy)
      if not self.walls[nextx][nexty]:
        nextFood = state[1].copy()
        nextFood[nextx][nexty] = False
        successors.append( ( ((nextx, nexty), nextFood), direction, 1) )
    return successors

  def getCostOfActions(self, actions):
    """Returns the cost of a particular sequence of actions.  If those actions
    include an illegal move, return 999999"""
    x,y= self.getStartState()[0]
    cost = 0
    for action in actions:
      # figure out the next state and see whether it's legal
      dx, dy = Actions.directionToVector(action)
      x, y = int(x + dx), int(y + dy)
      if self.walls[x][y]:
        return 999999
      cost += 1
    return cost

class AStarFoodSearchAgent(SearchAgent):
  "A SearchAgent for FoodSearchProblem using A* and your foodHeuristic"
  def __init__(self):
    self.searchFunction = lambda prob: search.aStarSearch(prob, foodHeuristic)
    self.searchType = FoodSearchProblem

def foodHeuristic(state, problem):
  """
  Your heuristic for the FoodSearchProblem goes here.

  This heuristic must be consistent to ensure correctness.  First, try to come up
  with an admissible heuristic; almost all admissible heuristics will be consistent
  as well.

  If using A* ever finds a solution that is worse uniform cost search finds,
  your heuristic is *not* consistent, and probably not admissible!  On the other hand,
  inadmissible or inconsistent heuristics may find optimal solutions, so be careful.

  The state is a tuple ( pacmanPosition, foodGrid ) where foodGrid is a
  Grid (see game.py) of either True or False. You can call foodGrid.asList()
  to get a list of food coordinates instead.

  If you want access to info like walls, capsules, etc., you can query the problem.
  For example, problem.walls gives you a Grid of where the walls are.

  If you want to *store* information to be reused in other calls to the heuristic,
```

```
  there is a dictionary called problem.heuristicInfo that you can use. For example,
  if you only want to count the walls once and store that value, try:
    problem.heuristicInfo['wallCount'] = problem.walls.count()
  Subsequent calls to this heuristic can access problem.heuristicInfo['wallCount']
  """
  position, foodGrid = state
  "*** YOUR CODE HERE ***"
  return 0

class ClosestDotSearchAgent(SearchAgent):
  "Search for all food using a sequence of searches"
  def registerInitialState(self, state):
    self.actions = []
    currentState = state
    while(currentState.getFood().count() > 0):
      nextPathSegment = self.findPathToClosestDot(currentState) # The missing piece
      self.actions += nextPathSegment
      for action in nextPathSegment:
        legal = currentState.getLegalActions()
        if action not in legal:
          t = (str(action), str(currentState))
          raise Exception, 'findPathToClosestDot returned an illegal move: %s!\n%s' % t
        currentState = currentState.generateSuccessor(0, action)
    self.actionIndex = 0
    print 'Path found with cost %d.' % len(self.actions)

  def findPathToClosestDot(self, gameState):
    "Returns a path (a list of actions) to the closest dot, starting from gameState"
    # Here are some useful elements of the startState
    startPosition = gameState.getPacmanPosition()
    food = gameState.getFood()
    walls = gameState.getWalls()
    problem = AnyFoodSearchProblem(gameState)

    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()

class AnyFoodSearchProblem(PositionSearchProblem):
  """
    A search problem for finding a path to any food.

    This search problem is just like the PositionSearchProblem, but
    has a different goal test, which you need to fill in below.  The
    state space and successor function do not need to be changed.

    The class definition above, AnyFoodSearchProblem(PositionSearchProblem),
    inherits the methods of the PositionSearchProblem.

    You can use this search problem to help you fill in
    the findPathToClosestDot method.
  """

  def __init__(self, gameState):
    "Stores information from the gameState.  You don't need to change this."
    # Store the food for later reference
    self.food = gameState.getFood()

    # Store info for the PositionSearchProblem (no need to change this)
    self.walls = gameState.getWalls()
    self.startState = gameState.getPacmanPosition()
    self.costFn = lambda x: 1
    self._visited, self._visitedlist, self._expanded = {}, [], 0

  def isGoalState(self, state):
    """
    The state is Pacman's position. Fill this in with a goal test
    that will complete the problem definition.
    """
    x,y = state

    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()
```



```
##################
# Mini-contest 1 #
##################

class ApproximateSearchAgent(Agent):
  "Implement your contest entry here.  Change anything but the class name."

  def registerInitialState(self, state):
    "This method is called before any moves are made."
    "*** YOUR CODE HERE ***"

  def getAction(self, state):
    """
    From game.py:
    The Agent will receive a GameState and must return an action from
    Directions.{North, South, East, West, Stop}
    """
    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()

def mazeDistance(point1, point2, gameState):
  """
  Returns the maze distance between any two points, using the search functions
  you have already built.  The gameState can be any game state -- Pacman's position
  in that state is ignored.

  Example usage: mazeDistance( (2,4), (5,6), gameState)

  This might be a useful helper function for your ApproximateSearchAgent.
  """
  x1, y1 = point1
  x2, y2 = point2
  walls = gameState.getWalls()
  assert not walls[x1][y1], 'point1 is a wall: ' + point1
  assert not walls[x2][y2], 'point2 is a wall: ' + str(point2)
  prob = PositionSearchProblem(gameState, start=point1, goal=point2, warn=False)
  return len(search.bfs(prob))
```