UC BERKELEY CS188 INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Project1: Search

Q1 (3 pts): Finding a Fixed Food Dot using Depth First Search

In searchAgents.py, you'll find a fully implemented SearchAgent, which plans out a path through Pacman's world and then executes that path step-by-step. The search algorithms for formulating a plan are not implemented – that's your job.

First, test that the SearchAgent is working correctly by running:

python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch

The command above tells the SearchAgent to use tinyMazeSearch as its search algorithm, which is implemented in search.py. Pacman should navigate the maze successfully.

Now it's time to write full-fledged generic search functions to help Pacman plan routes! Pseudocode for the search algorithms you'll write can be found in the lecture slides. Remember that a search node must contain not only a state but also the information necessary to reconstruct the path (plan) which gets to that state.

Implement the depth-first search (DFS) algorithm in the depthFirstSearch function in search.py. To make your algorithm complete, write the graph search version of DFS, which avoids expanding any already visited states.

Your code should quickly find a solution for:

python pacman.py -l tinyMaze -p SearchAgent

python pacman.py -l mediumMaze -p SearchAgent

python pacman.py -l bigMaze -z .5 -p SearchAgent

Solution:

```
# searchAgents.py
# ---------------
# Licensing Information:  You are free to use or extend these projects for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to UC Berkeley, including a link to http://ai.berkeley.edu.


"""
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 fn=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 typing import List, Tuple, Any
from game import Directions
from game import Agent
from game import Actions
import util
import time
import search
import pacman

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

        # 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.__code__.co_varnames:
            print('[SearchAgent] using function ' + fn)
            self.searchFunction = func
        else:
            if heuristic in globals().keys():
                heur = globals()[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 globals().keys() or not prob.endswith('Problem'):
            raise AttributeError(prob + ' is not a search problem type in SearchAgents.py.')
        self.searchType = globals()[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
        if self.actions == None:
            self.actions = []
        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, visualize=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
        self.visualize = visualize
        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 # DO NOT CHANGE

    def getStartState(self):
        return self.startState

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

        # For display purposes only
        if isGoal and self.visualize:
            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 # DO NOT CHANGE
        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, (1, 1),
None, False)

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: pacman.GameState):
    """
    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 # DO NOT CHANGE; Number of search nodes expanded
    # Use tuple to keep track of the corners visited.
    self.visit = (0, 0, 0, 0)

def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    # Return a tuple representation of the state
    return (self.startingPosition, self.visit)

def isGoalState(self, state: Any):
    """
    Returns whether this search state is a goal state of the problem.
    """
    # Check if all corners have been visited.
    return sum(state[1]) == len(self.corners)

def getSuccessors(self, state: Any):
    """
    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:
            # Uncommented and modified the given code
            x,y = state[0]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]
            if not hitsWall:
                nextState = (nextx, nexty)
                visit_lst = list(state[1])
                # As hinted in the writup
                cost = 1
                # Check if a neighbor is the corner
                if nextState in self.corners:
                    # Update the food left to be eaten
                    visit_lst[self.corners.index(nextState)] = 1
                # Add the successor to the list
                successors.append( ( (nextState, tuple(visit_lst)), action, cost) )

        self._expanded += 1 # DO NOT CHANGE
        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: Any, problem: CornersProblem):
    """
```

```
    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)
    # Obtain the current coordinates
    x, y = state[0]
    # Use list comprehension to get corners yet to be visited
    corners = [corners[i] for i in range(len(corners))
        if state[1][i] == 0]
    # Initialize maximum manhattan distance as heuristic.
    # If all corner have been visited, zero will be the heuristic
    max_manhattan = 0
    # Traverse through all unvisited corners and find the maximum
    # Manhattan distance to visit one of these corners.
    for corner in corners:
        manhattan_dist = abs( x - corner[0] ) + abs( y - corner[1] )
        if manhattan_dist > max_manhattan:
            max_manhattan = manhattan_dist
    return max_manhattan

# Minimum Manhattan distance as heuristic, score 2/3
    # corners = problem.corners # These are the corner coordinates
    # walls = problem.walls # These are the walls of the maze, as a Grid (game.py)
    # # Obtain the current coordinates
    # x, y = state[0]
    # # Use list comprehension to get corners yet to be visited
    # corners = [corners[i] for i in range(len(corners))
    #     if state[1][i] == 0]
    # # Initialize maximum manhattan distance as heuristic.
    # # If all corner have been visited, zero will be the heuristic
    # if len(corners) != 0:
    #     min_manhattan = float('inf')
    # else:
    #     min_manhattan = 0
    # # Traverse through all unvisited corners and find the maximum
```

```
    # # Manhattan distance to visit one of these corners.
    # for corner in corners:
    #     manhattan_dist = abs( x - corner[0] ) + abs( y - corner[1] )
    #     if manhattan_dist <= min_manhattan:
    #         min_manhattan = manhattan_dist
    # return min_manhattan

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: pacman.GameState):
        self.start = (startingGameState.getPacmanPosition(), startingGameState.getFood())
        self.walls = startingGameState.getWalls()
        self.startingGameState = startingGameState
        self._expanded = 0 # DO NOT CHANGE
        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 # DO NOT CHANGE
        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: Tuple[Tuple, List[List]], problem: FoodSearchProblem):
    """
    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
        # Get the current position as x,y values
        x, y = position
        # Obtain the grid coordinates of food to be eaten
        dots = foodGrid.asList()
        # Initialize maximum maze distance as heuristic
        max_maze_dist = 0
        # General find of maximum maze distance of all food
        for dot in dots:
            maze_dist = mazeDistance( (x, y), dot, problem.startingGameState)
            if maze_dist > max_maze_dist:
                max_maze_dist = maze_dist
        return max_maze_dist

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: pacman.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)

        # Use USC directly as search algorithm
        return search.uniformCostSearch(problem)

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 # DO NOT
CHANGE

    def isGoalState(self, state: Tuple[int, int]):
```

```
        """
        The state is Pacman's position. Fill this in with a goal test that will
        complete the problem definition.
        """
        x,y = state

        # Find all possible food positions
        food_pos = self.food.asList()
        # Check if we are eating!
        return (x, y) in food_pos

def mazeDistance(point1: Tuple[int, int], point2: Tuple[int, int], gameState: pacman.GameState) -> int:
    """
    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: ' + str(point1)
    assert not walls[x2][y2], 'point2 is a wall: ' + str(point2)
    prob = PositionSearchProblem(gameState, start=point1, goal=point2, warn=False, visualize=False)
    return len(search.bfs(prob))
```

Q2 (3 pts): Breadth First Search
Implement the breadth-first search (BFS) algorithm in the breadthFirstSearch function in search.py. Again, write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for depth-first search for:
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
If you've written your search code generically, your code should work equally well for the eight-puzzle search problem without any changes for:
python eightpuzzle.py

Q3 (3 pts): Varying the Cost Function
While BFS will find a fewest-actions path to the goal, we might want to find paths that are "best" in other senses. Consider mediumDottedMaze and mediumScaryMaze.
By changing the cost function, we can encourage Pacman to find different paths. For example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas, and a rational Pacman agent should adjust its behavior in response.
Implement the uniform-cost graph search algorithm in the uniformCostSearch function in search.py. We encourage you to look through util.py for some data structures that may be useful in your implementation. You should now observe successful behavior in all three of the following layouts, where the agents below are all UCS agents that differ only in the cost function they use (the agents and cost functions are written for you):
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent

Q4 (3 pts): A* search
Implement A* graph search in the empty function aStarSearch in search.py. A* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The nullHeuristic heuristic function in search.py is a trivial example.
You can test your A* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic (implemented already as manhattanHeuristic in searchAgents.py):
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

Solution:
```
# search.py
# ---------
# Licensing Information:  You are free to use or extend these projects for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to UC Berkeley, including a link to http://ai.berkeley.edu.
```

```
"""
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: SearchProblem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    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()))
    """
    # Stack to do DFS
    fringe = util.Stack()
    # Push a tuple of start state and it's corresponding path into stack
    fringe.push( (problem.getStartState(), []) )
    # A set to keep track all of visited stated/positions
    explored = set()
    while True:
        # Exit the while loop if Stack is empty
        if fringe.isEmpty():
            break
        # Get the last in tuple from Stack
        state = fringe.pop()
        # Check if goal state is reached
```

```
            if problem.isGoalState(state[0]):
                # Return solution path if goal state is reached
                return state[1]
            if state[0] not in explored:
                # Add the visited state to the set to avoid expanding twice
                explored.add(state[0])
                # Add successors to the Stack for further exploration
                for neighbor in problem.getSuccessors(state[0]):
                    fringe.push( (neighbor[0], state[1] + [neighbor[1]]) )

def breadthFirstSearch(problem: SearchProblem):
    """Search the shallowest nodes in the search tree first."""
    # Queue to do BFS, reuse most of the code from DFS function.
    fringe = util.Queue()
    # Push a tuple of start state and it's corresponding path into Queue
    fringe.push( (problem.getStartState(), []) )
    # A set to keep track all of visited stated/positions
    explored = set()
    while True:
        # Exit the while loop if Stack is empty
        if fringe.isEmpty():
            break
        # Get the first in tuple from Queue
        state = fringe.pop()
        # Check if goal state is reached
        if problem.isGoalState(state[0]):
            # Return solution path if goal state is reached
            return state[1]
        if state[0] not in explored:
            # Add the visited state to the set to avoid expanding twice
            explored.add(state[0])
            # Add successors to the Queue for further exploration
            for neighbor in problem.getSuccessors(state[0]):
                fringe.push( (neighbor[0], state[1] + [neighbor[1]]) )

def uniformCostSearch(problem: SearchProblem):
    """Search the node of least total cost first."""
    # PriorityQueue to do UCS, reuse most of the code from DFS function.
    fringe = util.PriorityQueue()
    # Push a tuple of start state, path, and the cost value to the
    # PriorityQueue, with the priority value
    fringe.push( (problem.getStartState(), [], 0), 0 )
    # A set to keep track all of visited stated/positions
    explored = set()
```

```
    while True:
        # Exit the while loop if Stack is empty
        if fringe.isEmpty():
            break
        # Get the state with highest priority
        state = fringe.pop()
        # Check if goal state is reached
        if problem.isGoalState(state[0]):
            # Return solution path if goal state is reached
            return state[1]
        if state[0] not in explored:
            # Add the visited state to the set to avoid expanding twice
            explored.add(state[0])
            # Add successors to the PriorityQueue for further exploration
            for neighbor in problem.getSuccessors(state[0]):
                fringe.update( (neighbor[0], state[1] + [neighbor[1]],
                    state[2] + neighbor[2]), state[2] + neighbor[2] )

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: SearchProblem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    # PriorityQueue to do A* search, reuse most of the code from DFS function.
    fringe = util.PriorityQueue()
    # Push a tuple of start state, path, and the cost value to the
    # PriorityQueue, with the priority value
    fringe.push( (problem.getStartState(), [], 0),
        heuristic(problem.getStartState(), problem))
    # A set to keep track all of visited stated/positions
    explored = set()
    while True:
        # Exit the while loop if PriorityQueue is empty
        if fringe.isEmpty():
            break
        # Get the state with highest priority
        state = fringe.pop()
        # Check if goal state is reached
        if problem.isGoalState(state[0]):
            # Return solution path if goal state is reached
```

```
            return state[1]
        if state[0] not in explored:
            # Add the visited state to the set to avoid expanding twice
            explored.add(state[0])
            # Add successors to the PriorityQueue for further exploration
            for neighbor in problem.getSuccessors(state[0]):
                fringe.update( (neighbor[0], state[1] + [neighbor[1]],
                    state[2] + neighbor[2]),
                    state[2] + neighbor[2] + heuristic(neighbor[0], problem))


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

Q5 (3 pts): Finding All the Corners

The real power of A* will only be apparent with a more challenging search problem. Now, it's time to formulate a new problem and design a heuristic for it.

In corner mazes, there are four dots, one in each corner. Our new search problem is to find the shortest path through the maze that touches all four corners (whether the maze actually has food there or not). Note that for some mazes like tinyCorners, the shortest path does not always go to the closest food first! Hint: the shortest path through tinyCorners takes 28 steps.

Implement the CornersProblem search problem in searchAgents.py. You will need to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached. Now, your search agent should solve:

python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

Q6 (3 pts): Corners Problem: Heuristic

Admissibility vs. Consistency: Remember, heuristics are just functions that take search states and return numbers that estimate the cost to a nearest goal. More effective heuristics will return values closer to the actual goal costs. To be admissible, the heuristic values must be lower bounds on the actual shortest path cost to the nearest goal (and non-negative). To be consistent, it must additionally hold that if an action has cost c, then taking that action can only cause a drop in heuristic of at most c.

Remember that admissibility isn't enough to guarantee correctness in graph search – you need the stronger condition of consistency. However, admissible heuristics are usually also consistent, especially if they are derived from problem relaxations. Therefore it is usually easiest to start out by brainstorming admissible heuristics. Once you have an admissible heuristic that works well, you can check whether it is indeed consistent, too. The only way to guarantee consistency is with a proof. However, inconsistency can often be detected by verifying that for each node you expand, its successor nodes are equal or higher in in f-value. Moreover, if UCS and A* ever return paths of different lengths, your heuristic is inconsistent. This stuff is tricky!

Non-Trivial Heuristics: The trivial heuristics are the ones that return zero everywhere (UCS) and the heuristic which computes the true completion cost. The former won't save you any time, while the latter will timeout the autograder. You want a heuristic which reduces total compute time, though for this assignment the autograder will only check node counts (aside from enforcing a reasonable time limit).

Implement a non-trivial, consistent heuristic for the CornersProblem in cornersHeuristic in searchAgents.py for:

python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5

Q7 (4 pts): Eating All The Dots

Now we'll solve a hard search problem: eating all the Pacman food in as few steps as possible. For this, we'll need a new search problem definition which formalizes the food-clearing problem: FoodSearchProblem in searchAgents.py (implemented for you). A solution is defined to be a path that collects all of the food in the Pacman world. For

the present project, solutions do not take into account any ghosts or power pellets; solutions only depend on the placement of walls, regular food and Pacman. (Of course ghosts can ruin the execution of a solution! We'll get to that in the next project.) If you have written your general search methods correctly, A* with a null heuristic (equivalent to uniform-cost search) should quickly find an optimal solution to testSearch with no code change on your part (total cost of 7) for:
python pacman.py -l testSearch -p AStarFoodSearchAgent

Q8 (3 pts): Suboptimal Search
Sometimes, even with A* and a good heuristic, finding the optimal path through all the dots is hard. In these cases, we'd still like to find a reasonably good path, quickly. In this section, you'll write an agent that always greedily eats the closest dot. ClosestDotSearchAgent is implemented for you in searchAgents.py, but it's missing a key function that finds a path to the closest dot.
Implement the function findPathToClosestDot in searchAgents.py. Our agent solves this maze (suboptimally!) in under a second with a path cost of 350:
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5

Solution: See Q1

Project2: Multi-Agent Search

Q1 (4 pts): Reflex Agent

Improve the ReflexAgent in multiAgents.py to play respectably. The provided reflex agent code provides some helpful examples of methods that query the GameState for information. A capable reflex agent will have to consider both food locations and ghost locations to perform well. Your agent should easily and reliably clear the testClassic layout:

python pacman.py -p ReflexAgent -l testClassic

Try out your reflex agent on the default mediumClassic layout with one ghost or two (and animation off to speed up the display):

python pacman.py --frameTime 0 -p ReflexAgent -k 1

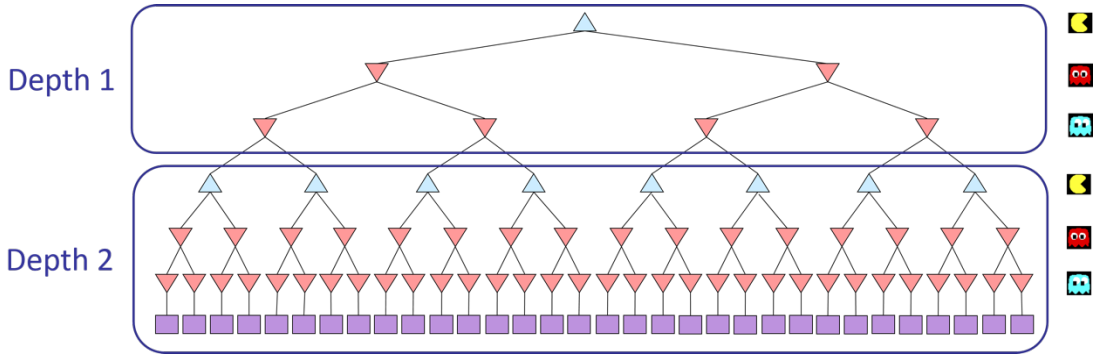
python pacman.py --frameTime 0 -p ReflexAgent -k 2

Q2 (5 pts): Minimax

Now you will write an adversarial search agent in the provided MinimaxAgent class stub in multiAgents.py. Your minimax agent should work with any number of ghosts, so you'll have to write an algorithm that is slightly more general than what you've previously seen in lecture. In particular, your minimax tree will have multiple min layers (one for each ghost) for every max layer.

Your code should also expand the game tree to an arbitrary depth. Score the leaves of your minimax tree with the supplied self.evaluationFunction, which defaults to scoreEvaluationFunction. MinimaxAgent extends MultiAgentSearchAgent, which gives access to self.depth and self.evaluationFunction. Make sure your minimax code makes reference to these two variables where appropriate as these variables are populated in response to command line options.

Important: A single search ply is considered to be one Pacman move and all the ghosts' responses, so depth 2 search will involve Pacman and each ghost moving two times (see diagram below):



Q3 (5 pts): Alpha-Beta Pruning

Make a new agent that uses alpha-beta pruning to more efficiently explore the minimax tree, in AlphaBetaAgent in multiAgents.py. Again, your algorithm will be slightly more general than the pseudocode from lecture, so part of the challenge is to extend the alpha-beta pruning logic appropriately to multiple minimizer agents.

You should see a speed-up (perhaps depth 3 alpha-beta will run as fast as depth 2

minimax). Ideally, depth 3 on smallClassic should run in just a few seconds per move or faster for:
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic

Q4 (5 pts): Expectimax
Minimax and alpha-beta are great, but they both assume that you are playing against an adversary who makes optimal decisions. As anyone who has ever won tic-tac-toe can tell you, this is not always the case. In this question you will implement the ExpectimaxAgent in multiAgents.py, which is useful for modeling probabilistic behavior of agents who may make suboptimal choices.
As with the search and problems yet to be covered in this class, the beauty of these algorithms is their general applicability. To expedite your own development, we've supplied some test cases based on generic trees.
Once your algorithm is working on small trees, you can observe its success in Pacman. Random ghosts are of course not optimal minimax agents, and so modeling them with minimax search may not be appropriate. ExpectimaxAgent will no longer take the min over all ghost actions, but the expectation according to your agent's model of how the ghosts act. To simplify your code, assume you will only be running against an adversary which chooses amongst their getLegalActions uniformly at random.
To see how the ExpectimaxAgent behaves in Pacman, run:
python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3
You should now observe a more cavalier approach in close quarters with ghosts. In particular, if Pacman perceives that he could be trapped but might escape to grab a few more pieces of food, he'll at least try. Investigate the results of these two scenarios:
python pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10
python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10

Q5 (6 pts): Evaluation Function
Write a better evaluation function for Pacman in the provided function betterEvaluationFunction in multiAgents.py. The evaluation function should evaluate states, rather than actions like your reflex agent evaluation function did. With depth 2 search, your evaluation function should clear the smallClassic layout with one random ghost more than half the time and still run at a reasonable rate.
If you win at least once without timing out the autograder, you receive 1 points. Any agent not satisfying these criteria will receive 0 points.
+1 for winning at least 5 times, +2 for winning all 10 times
+1 for an average score of at least 500, +2 for an average score of at least 1000 (including scores on lost games)
+1 if your games take on average less than 30 seconds on the autograder machine, when run with --no-graphics.
The additional points for average score and computation time will only be awarded if you win at least 5 times.

Solution:

```
# multiAgents.py
# --------------
# Licensing Information:  You are free to use or extend these projects for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to UC Berkeley, including a link to http://ai.berkeley.edu.


from util import manhattanDistance
from game import Directions
import random, util

from game import Agent
from pacman import GameState

class ReflexAgent(Agent):
    """
    A reflex agent chooses an action at each choice point by examining
    its alternatives via a state evaluation function.

    The code below is provided as a guide.  You are welcome to change
    it in any way you see fit, so long as you don't touch our method
    headers.
    """


    def getAction(self, gameState: GameState):
        """
        You do not need to change this method, but you're welcome to.

        getAction chooses among the best options according to the evaluation function.

        Just like in the previous project, getAction takes a GameState and returns
        some Directions.X for some X in the set {NORTH, SOUTH, WEST, EAST, STOP}
        """
        # Collect legal moves and successor states
        legalMoves = gameState.getLegalActions()

        # Choose one of the best actions
        scores = [self.evaluationFunction(gameState, action) for action in legalMoves]
        bestScore = max(scores)
```

```
        bestIndices = [index for index in range(len(scores)) if scores[index] ==
bestScore]
        chosenIndex = random.choice(bestIndices) # Pick randomly among the best

        "Add more of your code here if you want to"

        return legalMoves[chosenIndex]

    def evaluationFunction(self, currentGameState: GameState, action):
        """
        Design a better evaluation function here.

        The evaluation function takes in the current and proposed successor
        GameStates (pacman.py) and returns a number, where higher numbers are
better.

        The code below extracts some useful information from the state, like the
        remaining food (newFood) and Pacman position after moving (newPos).
        newScaredTimes holds the number of moves that each ghost will remain
        scared because of Pacman having eaten a power pellet.

        Print out these variables to see what you're getting, then combine them
        to create a masterful evaluation function.
        """
        # Useful information you can extract from a GameState (pacman.py)
        successorGameState = currentGameState.generatePacmanSuccessor(action)
        newPos = successorGameState.getPacmanPosition()
        newFood = successorGameState.getFood()
        newGhostStates = successorGameState.getGhostStates()
        newScaredTimes = [ghostState.scaredTimer for ghostState in
newGhostStates]

        # Sanity check in case no more food on the game board
        if len(newFood.asList()) == 0:
            foodScore = 0
        else:
            # Use manhattan distance to find the closest food distance
            closestFoodDist = min([manhattanDistance(newPos, foodPos)
                for foodPos in newFood.asList()])
            # Take the reciprocal as a part of the evaluation function
            foodScore = 1 / closestFoodDist

        # Use manhattan distance to find the closest ghost distance
        closestGhostDist = min([manhattanDistance(newPos,
```

```
                ghostState.configuration.pos) for ghostState in newGhostStates])
        # Sanity check in case there is no more ghost
        if closestGhostDist == 0:
            ghostScore = 0
        else:
            # Take the reciprocal as a part of the evaluation function
            ghostScore = 2 / closestGhostDist
        return successorGameState.getScore() + foodScore - ghostScore

def scoreEvaluationFunction(currentGameState: GameState):
    """
    This default evaluation function just returns the score of the state.
    The score is the same one displayed in the Pacman GUI.

    This evaluation function is meant for use with adversarial search agents
    (not reflex agents).
    """
    return currentGameState.getScore()

class MultiAgentSearchAgent(Agent):
    """
    This class provides some common elements to all of your
    multi-agent searchers.  Any methods defined here will be available
    to the MinimaxPacmanAgent, AlphaBetaPacmanAgent & ExpectimaxPacmanAgent.

    You *do not* need to make any changes here, but you can if you want to
    add functionality to all your adversarial search agents.  Please do not
    remove anything, however.

    Note: this is an abstract class: one that should not be instantiated.  It's
    only partially specified, and designed to be extended.  Agent (game.py)
    is another abstract class.
    """

    def __init__(self, evalFn = 'scoreEvaluationFunction', depth = '2'):
        self.index = 0 # Pacman is always agent index 0
        self.evaluationFunction = util.lookup(evalFn, globals())
        self.depth = int(depth)

class MinimaxAgent(MultiAgentSearchAgent):
    """
    Your minimax agent (question 2)
    """
```

```
    def getAction(self, gameState: GameState):
        """
        Returns the minimax action from the current gameState using self.depth
        and self.evaluationFunction.

        Here are some method calls that might be useful when implementing minimax.

        gameState.getLegalActions(agentIndex):
        Returns a list of legal actions for an agent
        agentIndex=0 means Pacman, ghosts are >= 1

        gameState.generateSuccessor(agentIndex, action):
        Returns the successor game state after an agent takes an action

        gameState.getNumAgents():
        Returns the total number of agents in the game

        gameState.isWin():
        Returns whether or not the game state is a winning state

        gameState.isLose():
        Returns whether or not the game state is a losing state
        """
        # Sanity check to see if we reach the end of the game/condition
        if gameState.isWin() or gameState.isLose():
            return self.evaluationFunction(gameState)
        # Initialize the maxValue
        maxValue = -float('inf')
        # Initialize resulting move
        result = None
        # Iterate through all possible actions and compare the value
        for action in gameState.getLegalActions(agentIndex=0):
            # Extract the successor of the current gameState
            successor = gameState.generateSuccessor(agentIndex=0, action=action)
            # Temporary variable used to compare with maxValue
            temp_val = self.min_value(successor, 1, 0)
            # Compare values and update if necessary
            if temp_val > maxValue:
                maxValue = temp_val
                result = action
        return result

    def max_value(self, gameState, agentIndex, depth):
```

```
        """
        Returns the maximum value obtained from iterating through the
        given gameState.
        """
        # Base case when we reach the end of the game/condition
        if gameState.isWin() or gameState.isLose() or depth == self.depth:
            return self.evaluationFunction(gameState)
        # Same as pseudocode in lecture, initialize the maxValue
        maxValue = -float('inf')
        # Get all possible actions from the current gameState
        legalAction = gameState.getLegalActions(agentIndex=agentIndex)
        # Iterate through all possible actions to find the max
        for action in legalAction:
            # Extract the successor of the current gameState
            successor = gameState.generateSuccessor(agentIndex, action)
            # Update maxValue when necessary
            maxValue = max(maxValue, self.min_value(successor, agentIndex + 1,
depth))
        return maxValue

    def min_value(self, gameState, agentIndex, depth):
        """
        Returns the minimum value obtained from iterating through the
        given gameState.
        """
        # Base case when we reach the end of the game/condition
        if gameState.isWin() or gameState.isLose() or depth == self.depth:
            return self.evaluationFunction(gameState)
        # Same as pseudocode in lecture, initialize the maxValue
        minValue = float('inf')
        # Get all possible actions from the current gameState
        legalAction = gameState.getLegalActions(agentIndex=agentIndex)
        # Iterate through all possible actions to find the max
        for action in legalAction:
            # Extract the successor of the current gameState
            successor = gameState.generateSuccessor(agentIndex, action)
            # Update maxValue when necessary; increment depth when we have
            # iterated through all agents; otherwise update agent index
            minValue = min(minValue, self.max_value(successor, 0, depth + 1)
                if agentIndex == gameState.getNumAgents() - 1
                else self.min_value(successor, agentIndex + 1, depth))
        return minValue

class AlphaBetaAgent(MultiAgentSearchAgent):
```

```
    """
    Your minimax agent with alpha-beta pruning (question 3)
    """

    def getAction(self, gameState: GameState):
        """
        Returns the minimax action using self.depth and self.evaluationFunction
        """
        # Initialize variables used for comparison later
        alpha = -float('inf')
        beta = float('inf')
        maxValue = -float('inf')
        # Initialize the resulting move
        result = None
        # Iterate through all possible actions and compare the value
        for action in gameState.getLegalActions(agentIndex=0):
            successor = gameState.generateSuccessor(agentIndex=0, action=action)
            temp_val = self.min_value(successor, agentIndex=1, depth=0,
alpha=alpha, beta=beta)
            if temp_val > maxValue:
                maxValue = temp_val
                result = action
            if maxValue >= beta:
                return action
            alpha = max(alpha, maxValue)
        return result

    def max_value(self, gameState, agentIndex, depth, alpha, beta):
        """
        Returns the maximum value obtained from iterating through the
        given gameState.
        """
        if gameState.isWin() or gameState.isLose() or depth == self.depth:
            return self.evaluationFunction(gameState)

        maxValue = -float('inf')
        legalAction = gameState.getLegalActions(agentIndex=agentIndex)
        for action in legalAction:
            successor = gameState.generateSuccessor(agentIndex, action)
            maxValue = max(maxValue, self.min_value(successor, agentIndex + 1,
depth, alpha, beta))
            if maxValue > beta:
                return maxValue
            alpha = max(alpha, maxValue)
```

```
            return maxValue

        def min_value(self, gameState, agentIndex, depth, alpha, beta):
            """
            Returns the minimum value obtained from iterating through the
            given gameState.
            """
            if gameState.isWin() or gameState.isLose() or depth == self.depth:
                return self.evaluationFunction(gameState)

            minValue = float('inf')
            legalAction = gameState.getLegalActions(agentIndex=agentIndex)
            for action in legalAction:
                successor = gameState.generateSuccessor(agentIndex, action)
                if agentIndex == gameState.getNumAgents() - 1:
                    minValue = min(minValue, self.max_value(successor, 0, depth + 1,
alpha, beta))
                else:
                    minValue = min(minValue, self.min_value(successor, agentIndex +
1, depth, alpha, beta))
                if minValue < alpha:
                    return minValue
                beta = min(beta, minValue)
            return minValue

class ExpectimaxAgent(MultiAgentSearchAgent):
    """
    Your expectimax agent (question 4)
    """

    def getAction(self, gameState: GameState):
        """
        Returns the expectimax action using self.depth and self.evaluationFunction

        All ghosts should be modeled as choosing uniformly at random from their
        legal moves.
        """
        "*** YOUR CODE HERE ***"
        maxValue = -float('inf')
        result = None
        for action in gameState.getLegalActions(agentIndex=0):
            successor = gameState.generateSuccessor(agentIndex=0, action=action)
            temp_val = self.exp_value(successor, 1, 0)
            if temp_val > maxValue:
```

```
                    maxValue = temp_val
                    result = action
        return result

    def max_value(self, gameState, agentIndex, depth):
        """
        Returns the maximum value obtained from iterating through the
        given gameState.
        """
        if gameState.isWin() or gameState.isLose() or depth == self.depth:
            return self.evaluationFunction(gameState)

        maxValue = -float('inf')
        legalAction = gameState.getLegalActions(agentIndex=0)
        for action in legalAction:
            successor = gameState.generateSuccessor(0, action)
            maxValue = max(maxValue, self.exp_value(successor, 1, depth))
        return maxValue

    def exp_value(self, gameState, agentIndex, depth):
        """
        Returns the minimum value obtained from iterating through the
        given gameState.
        """
        if gameState.isWin() or gameState.isLose() or depth == self.depth:
            return self.evaluationFunction(gameState)

        exp = 0
        legalActions = gameState.getLegalActions(agentIndex=agentIndex)
        prob = 1.0 / len(legalActions)
        for action in legalActions:
            successor = gameState.generateSuccessor(agentIndex, action)
            if agentIndex == gameState.getNumAgents() - 1:
                exp_temp = self.max_value(successor, 0, depth + 1)
            else:
                exp_temp = self.exp_value(successor, agentIndex + 1, depth)
            exp += prob * exp_temp
        return exp

def betterEvaluationFunction(currentGameState: GameState):
    """
    Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable
    evaluation function (question 5).
```

```
    DESCRIPTION: <write something here so we know what you did>
    """
    # Useful information you can extract from a GameState (pacman.py)
    currPos = currentGameState.getPacmanPosition()
    currFood = currentGameState.getFood()
    currGhostStates = currentGameState.getGhostStates()
    ScaredTimes = [ghostState.scaredTimer for ghostState in currGhostStates]

    # Check if there are still food remaining
    if len(currFood.asList()) == 0:
        foodScore = 0
    else:
        # Find the closest food using manhattan distance
        closestFoodDist = min([manhattanDistance(currPos, foodPos)
            for foodPos in currFood.asList()])
        # Feature score by taking the reciprocal of closest food distance
        foodScore = 1 / closestFoodDist

    # Use manhattan distance to find the closest ghost distance
    closestGhostDist = min([manhattanDistance(currPos,
        ghostState.configuration.pos) for ghostState in currGhostStates])
    # Sanity check when no ghost exist
    if closestGhostDist == 0:
        ghostScore = 0
    else:
        # Take the reciprocal in the final evaluation function
        ghostScore = 2 / closestGhostDist
    return currentGameState.getScore() + foodScore - ghostScore + sum(ScaredTimes)

# Abbreviation
better = betterEvaluationFunction
```

Project3: Reinforcement Learning

Q1 (5 points): Value Iteration

Write a value iteration agent in ValueIterationAgent, which has been partially specified for you in valueIterationAgents.py. Your value iteration agent is an offline planner, not a reinforcement learning agent, and so the relevant training option is the number of iterations of value iteration it should run (option -i) in its initial planning phase. ValueIterationAgent takes an MDP on construction and runs value iteration for the specified number of iterations before the constructor returns.

Value iteration computes k-step estimates of the optimal values, Vk. In addition to runValueIteration, implement the following methods for ValueIterationAgent using Vk:

- computeActionFromValues(state) computes the best action according to the value function given by self.values.
- computeQValueFromValues(state, action) returns the Q-value of the (state, action) pair given by the value function given by self.values.

These quantities are all displayed in the GUI: values are numbers in squares, Q-values are numbers in square quarters, and policies are arrows out from each square.

Important: Use the "batch" version of value iteration where each vector Vk is computed from a fixed vector Vk-1, not the "online" version where one single weight vector is updated in place. This means that when a state's value is updated in iteration k based on the values of its successor states, the successor state values used in the value update computation should be those from iteration k-1 (even if some of the successor states had already been updated in iteration k).

The following command loads your ValueIterationAgent, which will compute a policy and execute it 10 times. Press a key to cycle through values, Q-values, and the simulation. You should find that the value of the start state (V(start), which you can read off of the GUI) and the empirical resulting average reward (printed after the 10 rounds of execution finish) are quite close.

python gridworld.py -a value -i 100 -k 10

python gridworld.py -a value -i 5

Solution:

```
# valueIterationAgents.py
# -----------------------
# Licensing Information:  You are free to use or extend these projects for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to UC Berkeley, including a link to http://ai.berkeley.edu.
```

```
# valueIterationAgents.py
# -----------------------
# Licensing Information:  You are free to use or extend these projects for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
```

```
# attribution to UC Berkeley, including a link to http://ai.berkeley.edu.


import mdp, util

from learningAgents import ValueEstimationAgent
import collections

class ValueIterationAgent(ValueEstimationAgent):
    """
        * Please read learningAgents.py before reading this.*

        A ValueIterationAgent takes a Markov decision process
        (see mdp.py) on initialization and runs value iteration
        for a given number of iterations using the supplied
        discount factor.
    """
    def __init__(self, mdp: mdp.MarkovDecisionProcess, discount = 0.9, iterations =
100):
        """
          Your value iteration agent should take an mdp on
          construction, run the indicated number of iterations
          and then act according to the resulting policy.

          Some useful mdp methods you will use:
              mdp.getStates()
              mdp.getPossibleActions(state)
              mdp.getTransitionStatesAndProbs(state, action)
              mdp.getReward(state, action, nextState)
              mdp.isTerminal(state)
        """
        self.mdp = mdp
        self.discount = discount
        self.iterations = iterations
        self.values = util.Counter() # A Counter is a dict with default 0
        self.runValueIteration()

    def runValueIteration(self):
        """
          Run the value iteration algorithm. Note that in standard
          value iteration, V_k+1(...) depends on V_k(...)'s.
        """
        "*** YOUR CODE HERE ***"
        for iter in range(self.iterations):
```

```
                values = self.values.copy()
                for state in self.mdp.getStates():
                    if self.mdp.isTerminal(state):
                        continue
                    q_values = []
                    for action in self.mdp.getPossibleActions(state):
                        q_values.append(self.computeQValueFromValues(state,
action))
                    max_value = max(q_values)
                    values[state] = max_value
                self.values = values

    def getValue(self, state):
        """
          Return the value of the state (computed in __init__).
        """
        return self.values[state]

    def computeQValueFromValues(self, state, action):
        """
          Compute the Q-value of action in state from the
          value function stored in self.values.
        """
        "*** YOUR CODE HERE ***"
        q_value = 0
        for next_state, prob in self.mdp.getTransitionStatesAndProbs(state, action):
            reward = self.mdp.getReward(state, action, next_state)
            q_value += prob * (reward + self.discount * self.values[next_state])
        return q_value

    def computeActionFromValues(self, state):
        """
          The policy is the best action in the given state
          according to the values currently stored in self.values.

          You may break ties any way you see fit.  Note that if
          there are no legal actions, which is the case at the
          terminal state, you should return None.
        """
        "*** YOUR CODE HERE ***"
        if self.mdp.isTerminal(state):
            return None

        action_values = util.Counter()
```

```
        for action in self.mdp.getPossibleActions(state):
            action_values[action] = self.computeQValueFromValues(state, action)
        return action_values.argMax()


    def getPolicy(self, state):
        return self.computeActionFromValues(state)

    def getAction(self, state):
        "Returns the policy at the state (no exploration)."
        return self.computeActionFromValues(state)

    def getQValue(self, state, action):
        return self.computeQValueFromValues(state, action)
```

Question 2 (1 point): Bridge Crossing Analysis
BridgeGrid is a grid world map with the a low-reward terminal state and a high-reward terminal state separated by a narrow “bridge”, on either side of which is a chasm of high negative reward. The agent starts near the low-reward state. With the default discount of 0.9 and the default noise of 0.2, the optimal policy does not cross the bridge. Change only ONE of the discount and noise parameters so that the optimal policy causes the agent to attempt to cross the bridge. Put your answer in question2() of analysis.py. The default corresponds to:
python gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 --noise 0.2

Solution:

```
# analysis.py
# -----------
# Licensing Information:  You are free to use or extend these projects for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to UC Berkeley, including a link to http://ai.berkeley.edu.


######################
# ANALYSIS QUESTIONS #
######################

# Set the given parameters to obtain the specified policies through
# value iteration.

def question2a():
    """
      Prefer the close exit (+1), risking the cliff (-10).
    """
    answerDiscount = .3
    answerNoise = 0
    answerLivingReward = 0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question2b():
    """
      Prefer the close exit (+1), but avoiding the cliff (-10).
    """
    answerDiscount = .3
    answerNoise = 0.2
    answerLivingReward = 0
    return answerDiscount, answerNoise, answerLivingReward
```

```
    # If not possible, return 'NOT POSSIBLE'

def question2c():
    """
      Prefer the distant exit (+10), risking the cliff (-10).
    """
    answerDiscount = 0.8
    answerNoise = 0
    answerLivingReward = 0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question2d():
    """
      Prefer the distant exit (+10), avoiding the cliff (-10).
    """
    answerDiscount = 0.5
    answerNoise = 0.5
    answerLivingReward = 1
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

def question2e():
    """
      Avoid both exits and the cliff (so an episode should never terminate).
    """
    answerDiscount = 0.2
    answerNoise = 0.2
    answerLivingReward = 10
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

if __name__ == '__main__':
    print('Answers to analysis questions:')
    import analysis
    for q in [q for q in dir(analysis) if q.startswith('question')]:
        response = getattr(analysis, q)()
        print('    Question %s:\t%s' % (q, str(response)))
```

Question 5 (5 points): Q-Learning

Note that your value iteration agent does not actually learn from experience. Rather, it ponders its MDP model to arrive at a complete policy before ever interacting with a real environment. When it does interact with the environment, it simply follows the precomputed policy (e.g. it becomes a reflex agent). This distinction may be subtle in a simulated environment like a Gridword, but it's very important in the real world, where

the real MDP is not available.
You will now write a Q-learning agent, which does very little on construction, but instead learns by trial and error from interactions with the environment through its update(state, action, nextState, reward) method. A stub of a Q-learner is specified in QLearningAgent in qlearningAgents.py, and you can select it with the option '-a q'. For this question, you must implement the update, computeValueFromQValues, getQValue, and computeActionFromQValues methods.
For computeActionFromQValues, you should break ties randomly for better behavior. The random.choice() function will help. In a particular state, actions that your agent hasn't seen before still have a Q-value, specifically a Q-value of zero, and if all of the actions that your agent has seen before have a negative Q-value, an unseen action may be optimal.
Important: Make sure that in your computeValueFromQValues and computeActionFromQValues functions, you only access Q values by calling getQValue. This abstraction will be useful for question 10 when you override getQValue to use features of state-action pairs rather than state-action pairs directly.
With the Q-learning update in place, you can watch your Q-learner learn under manual control, using the keyboard:
python gridworld.py -a q -k 5 -m

Question 6 (2 points): Epsilon Greedy
Complete your Q-learning agent by implementing epsilon-greedy action selection in getAction in qlearningAgents.py, meaning it chooses random actions an epsilon fraction of the time, and follows its current best Q-values otherwise. Note that choosing a random action may result in choosing the best action - that is, you should not choose a random sub-optimal action, but rather any random legal action.
You can choose an element from a list uniformly at random by calling the random.choice function. You can simulate a binary variable with probability p of success by using util.flipCoin(p), which returns True with probability p.
After implementing the getAction method, observe the following behavior of the agent in GridWorld (with epsilon = 0.3) for:
python gridworld.py -a q -k 100
Your final Q-values should resemble those of your value iteration agent, especially along well-traveled paths. However, your average returns will be lower than the Q-values predict because of the random actions and the initial learning phase.
You can also observe the following simulations for different epsilon values for:
python gridworld.py -a q -k 100 --noise 0.0 -e 0.1
python gridworld.py -a q -k 100 --noise 0.0 -e 0.9
Question 9 (4 points): Approximate Q-Learning
Implement an approximate Q-learning agent that learns weights for features of states, where many states might share the same features. Write your implementation in ApproximateQAgent class in qlearningAgents.py, which is a subclass of PacmanQAgent.
By default, ApproximateQAgent uses the IdentityExtractor, which assigns a single

feature to every (state,action) pair. With this feature extractor, your approximate Q-learning agent should work identically to PacmanQAgent. You can test this with the following command:

python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid

Important: ApproximateQAgent is a subclass of QLearningAgent, and it therefore shares several methods like getAction. Make sure that your methods in QLearningAgent call getQValue instead of accessing Q-values directly, so that when you override getQValue in your approximate agent, the new approximate q-values are used to compute actions.

Once you're confident that your approximate learner works correctly with the identity features, run your approximate Q-learning agent with our custom feature extractor, which can learn to win with ease:

python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid

Even much larger layouts should be no problem for your ApproximateQAgent (warning: this may take a few minutes to train):

python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumClassic

Solutions:

```
# qlearningAgents.py
# ------------------
# Licensing Information:  You are free to use or extend these projects for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to UC Berkeley, including a link to http://ai.berkeley.edu.


from game import *
from learningAgents import ReinforcementAgent
from featureExtractors import *

import gridworld

import random,util,math
import copy

class QLearningAgent(ReinforcementAgent):
    """
      Q-Learning Agent
      Functions you should fill in:
        - computeValueFromQValues
        - computeActionFromQValues
        - getQValue
```

```
        - getAction
        - update
    Instance variables you have access to
        - self.epsilon (exploration prob)
        - self.alpha (learning rate)
        - self.discount (discount rate)
    Functions you should use
        - self.getLegalActions(state)
            which returns legal actions for a state
"""
def __init__(self, **args):
        "You can initialize Q-values here..."
        ReinforcementAgent.__init__(self, **args)

        "*** YOUR CODE HERE ***"
        # Same as we did for valueIterationAgents
        self.values = util.Counter()

def getQValue(self, state, action):
        """
            Returns Q(state,action)
            Should return 0.0 if we have never seen a state
            or the Q node value otherwise
        """
        "*** YOUR CODE HERE ***"
        if (state, action) not in self.values:
                return 0.0
        return self.values[(state, action)]

def computeValueFromQValues(self, state):
        """
            Returns max_action Q(state,action)
            where the max is over legal actions.    Note that if
            there are no legal actions, which is the case at the
            terminal state, you should return a value of 0.0.
        """
        "*** YOUR CODE HERE ***"
        actions = self.getLegalActions(state)
        if not actions:
                return 0.0
        q_values = [self.getQValue(state, action) for action in actions]
        return max(q_values)

def computeActionFromQValues(self, state):
```

```
        """
          Compute the best action to take in a state.  Note that if there
          are no legal actions, which is the case at the terminal state,
          you should return None.
        """
        "*** YOUR CODE HERE ***"
        # terminal state
        if not self.getLegalActions(state):
            return None

        # otherwise, compute best action to take at s
        max_value = self.computeValueFromQValues(state)
        max_actions = [action for action in self.getLegalActions(state)
                               if self.getQValue(state, action) == max_value]
        return random.choice(max_actions)

    def getAction(self, state):
        """
          Compute the action to take in the current state.  With
          probability self.epsilon, we should take a random action and
          take the best policy action otherwise.  Note that if there are
          no legal actions, which is the case at the terminal state, you
          should choose None as the action.
          HINT: You might want to use util.flipCoin(prob)
          HINT: To pick randomly from a list, use random.choice(list)
        """
        # Pick Action
        legalActions = self.getLegalActions(state)
        action = None
        "*** YOUR CODE HERE ***"
        if not legalActions:
            return None
        elif util.flipCoin(self.epsilon):
            return random.choice(legalActions)
        else:
            return self.computeActionFromQValues(state)

        return action

    def update(self, state, action, nextState, reward: float):
        """
          The parent class calls this to observe a
          state = action => nextState and reward transition.
          You should do your Q-Value update here
```

```
            NOTE: You should never call this function,
            it will be called on your behalf
        """
        "*** YOUR CODE HERE ***"
        # Based on the formula from lecture
        sample = reward + (self.discount * self.computeValueFromQValues(nextState))
        self.values[(state, action)] = (1 - self.alpha) * self.values[(state, action)] + self.alpha * sample

    def getPolicy(self, state):
        return self.computeActionFromQValues(state)

    def getValue(self, state):
        return self.computeValueFromQValues(state)


class PacmanQAgent(QLearningAgent):
    "Exactly the same as QLearningAgent, but with different default parameters"

    def __init__(self, epsilon=0.05,gamma=0.8,alpha=0.2, numTraining=0, **args):
        """
        These default parameters can be changed from the pacman.py command line.
        For example, to change the exploration rate, try:
            python pacman.py -p PacmanQLearningAgent -a epsilon=0.1
        alpha       - learning rate
        epsilon     - exploration rate
        gamma       - discount factor
        numTraining - number of training episodes, i.e. no learning after these many episodes
        """
        args['epsilon'] = epsilon
        args['gamma'] = gamma
        args['alpha'] = alpha
        args['numTraining'] = numTraining
        self.index = 0  # This is always Pacman
        QLearningAgent.__init__(self, **args)

    def getAction(self, state):
        """
        Simply calls the getAction method of QLearningAgent and then
        informs parent of action for Pacman.  Do not change or remove this
        method.
        """
```

```
        action = QLearningAgent.getAction(self,state)
        self.doAction(state,action)
        return action

class ApproximateQAgent(PacmanQAgent):
    """
       ApproximateQLearningAgent
       You should only have to overwrite getQValue
       and update.  All other QLearningAgent functions
       should work as is.
    """
    def __init__(self, extractor='IdentityExtractor', **args):
        self.featExtractor = util.lookup(extractor, globals())()
        PacmanQAgent.__init__(self, **args)
        self.weights = util.Counter()

    def getWeights(self):
        return self.weights

    def getQValue(self, state, action):
        """
          Should return Q(state,action) = w * featureVector
          where * is the dotProduct operator
        """
        "*** YOUR CODE HERE ***"
        # Obtain all the features for calculation
        features = self.featExtractor.getFeatures(state, action)
        # Initialize the q_value to be 0
        q_value = 0
        # Sum over all features with their corresponding weights
        for feature in features:
            q_value += self.weights[feature] * features[feature]
        return q_value

    def update(self, state, action, nextState, reward: float):
        """
           Should update your weights based on transition
        """
        "*** YOUR CODE HERE ***"
        # Obtain the maximum Q-value for the nextState
        max_q_value = self.computeValueFromQValues(nextState)
        # Calculate the difference that will be used to update weight
        diff = reward + (self.discount * max_q_value) - self.getQValue(state, action)
        # Extract all features in order to update
```

```
            features = self.featExtractor.getFeatures(state, action)
            # Update rule based on the formula shown in lecture
            for feature in features:
                self.weights[feature] = self.weights[feature] + (self.alpha * diff * features[feature])

    def final(self, state):
            """Called at the end of each game."""
            # call the super-class final method
            PacmanQAgent.final(self, state)

            # did we finish training?
            if self.episodesSoFar == self.numTraining:
                # you might want to print your weights here for debugging
                "*** YOUR CODE HERE ***"
                # No print statement here
                Pass
```

Project 4: Ghostbusters

Question 1 (2 points): Bayes Net Structure

Implement the constructBayesNet function in inference.py. It constructs an empty Bayes Net with the structure described below. A Bayes Net is incomplete without the actual probabilities, but factors are defined and assigned by staff code separately; you don't need to worry about it. If you are curious, you can take a look at an example of how it works in printStarterBayesNet in bayesNet.py. Reading this function can also be helpful for doing this question.

Solution:

```
# inference.py
# ------------
# Licensing Information:  You are free to use or extend these projects for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to UC Berkeley, including a link to http://ai.berkeley.edu.


import random
import itertools
from typing import List, Dict, Tuple
import busters
import game
import bayesNet as bn
from bayesNet import normalize
import hunters
from util import manhattanDistance, raiseNotDefined
from factorOperations import joinFactorsByVariableWithCallTracking, joinFactors
from factorOperations import eliminateWithCallTracking

############ ############ ############
############ QUESTION 1    ############
############ ############ ############

def constructBayesNet(gameState: hunters.GameState):
    """
    Construct an empty Bayes net according to the structure given in Figure 1
    of the project description.

    You *must* name all variables using the constants in this function.

    In this method, you should:
    - populate `variables` with the Bayes Net nodes
    - populate `edges` with every edge in the Bayes Net. we will represent each
```

```
    edge as a tuple `(from, to)`.
- set each `variableDomainsDict[var] = values`, where `values` is a list
    of the possible assignments to `var`.
        - each agent position is a tuple (x, y) where x and y are 0-indexed
        - each observed distance is a noisy Manhattan distance:
            it's non-negative and |obs - true| <= MAX_NOISE
- this uses slightly simplified mechanics vs the ones used later for simplicity
"""
# constants to use
PAC = "Pacman"
GHOST0 = "Ghost0"
GHOST1 = "Ghost1"
OBS0 = "Observation0"
OBS1 = "Observation1"
X_RANGE = gameState.getWalls().width
Y_RANGE = gameState.getWalls().height
MAX_NOISE = 7

variables = []
edges = []
variableDomainsDict = {}

"*** YOUR CODE HERE ***"
# Constructing Bayes' nets: variables list
variables += [PAC, GHOST0, GHOST1, OBS0, OBS1]

# Constructing Bayes' nets, edge list: (x, y) means edge from x to y
edges.append((GHOST0, OBS0))
edges.append((PAC, OBS0))
edges.append((PAC, OBS1))
edges.append((GHOST1, OBS1))

# Construct the domain for each variable (a list like)
agent_domain = [(x,y) for x in range(X_RANGE) for y in range(Y_RANGE)]
obj_domain = [x for x in range(X_RANGE + Y_RANGE + MAX_NOISE - 1)]
variableDomainsDict[PAC]    = agent_domain.copy()
variableDomainsDict[GHOST0] = agent_domain.copy()
variableDomainsDict[GHOST1]    = agent_domain.copy()
variableDomainsDict[OBS0]    = obj_domain.copy()
variableDomainsDict[OBS1]    = obj_domain.copy()
"*** END YOUR CODE HERE ***"

net = bn.constructEmptyBayesNet(variables, edges, variableDomainsDict)
return net
```

```
def inferenceByEnumeration(bayesNet: bn, queryVariables: List[str], evidenceDict: Dict):
    """
    An inference by enumeration implementation provided as reference.
    This function performs a probabilistic inference query that
    returns the factor:

    P(queryVariables | evidenceDict)

    bayesNet:         The Bayes Net on which we are making a query.
    queryVariables: A list of the variables which are unconditioned in
                      the inference query.
    evidenceDict:   An assignment dict {variable : value} for the
                      variables which are presented as evidence
                      (conditioned) in the inference query.
    """
    callTrackingList = []
    joinFactorsByVariable = joinFactorsByVariableWithCallTracking(callTrackingList)
    eliminate = eliminateWithCallTracking(callTrackingList)

    # initialize return variables and the variables to eliminate
    evidenceVariablesSet = set(evidenceDict.keys())
    queryVariablesSet = set(queryVariables)
    eliminationVariables = (bayesNet.variablesSet() - evidenceVariablesSet) - queryVariablesSet

    # grab all factors where we know the evidence variables (to reduce the size of the tables)
    currentFactorsList = bayesNet.getAllCPTsWithEvidence(evidenceDict)

    # join all factors by variable
    for joinVariable in bayesNet.variablesSet():
        currentFactorsList, joinedFactor = joinFactorsByVariable(currentFactorsList, joinVariable)
        currentFactorsList.append(joinedFactor)

    # currentFactorsList should contain the connected components of the graph now as factors, must join the connected components
    fullJoint = joinFactors(currentFactorsList)

    # marginalize all variables that aren't query or evidence
```

```
    incrementallyMarginalizedJoint = fullJoint
    for eliminationVariable in eliminationVariables:
        incrementallyMarginalizedJoint = eliminate(incrementallyMarginalizedJoint,
eliminationVariable)

    fullJointOverQueryAndEvidence = incrementallyMarginalizedJoint

    # normalize so that the probability sums to one
    # the input factor contains only the query variables and the evidence variables,
    # both as unconditioned variables
    queryConditionedOnEvidence = normalize(fullJointOverQueryAndEvidence)
    # now the factor is conditioned on the evidence variables

    # the order is join on all variables, then eliminate on all elimination variables
    return queryConditionedOnEvidence

########## ########## ##########
########## QUESTION 4    ##########
########## ########## ##########

def inferenceByVariableEliminationWithCallTracking(callTrackingList=None):

    def inferenceByVariableElimination(bayesNet: bn, queryVariables: List[str],
evidenceDict: Dict, eliminationOrder: List[str]):
        """
        This function should perform a probabilistic inference query that
        returns the factor:

        P(queryVariables | evidenceDict)

        It should perform inference by interleaving joining on a variable
        and eliminating that variable, in the order of variables according
        to eliminationOrder.    See inferenceByEnumeration for an example on
        how to use these functions.

        You need to use joinFactorsByVariable to join all of the factors
        that contain a variable in order for the autograder to
        recognize that you performed the correct interleaving of
        joins and eliminates.

        If a factor that you are about to eliminate a variable from has
        only one unconditioned variable, you should not eliminate it
        and instead just discard the factor.    This is since the
        result of the eliminate would be 1 (you marginalize
```

```
        all of the unconditioned variables), but it is not a
        valid factor.    So this simplifies using the result of eliminate.

        The sum of the probabilities should sum to one (so that it is a true
        conditional probability, conditioned on the evidence).

        bayesNet:               The Bayes Net on which we are making a query.
        queryVariables:      A list of the variables which are unconditioned
                                in the inference query.
        evidenceDict:         An assignment dict {variable : value} for the
                                variables which are presented as evidence
                                (conditioned) in the inference query.
        eliminationOrder: The order to eliminate the variables in.

        Hint: BayesNet.getAllCPTsWithEvidence will return all the Conditional
        Probability Tables even if an empty dict (or None) is passed in for
        evidenceDict. In this case it will not specialize any variable domains
        in the CPTs.

        Useful functions:
        BayesNet.getAllCPTsWithEvidence
        normalize
        eliminate
        joinFactorsByVariable
        joinFactors
        """

        # this is for autograding -- don't modify
        joinFactorsByVariable                                                    =
joinFactorsByVariableWithCallTracking(callTrackingList)
        eliminate                    = eliminateWithCallTracking(callTrackingList)
        if eliminationOrder is None: # set an arbitrary elimination order if None given
            eliminationVariables = bayesNet.variablesSet() - set(queryVariables) -\
                                            set(evidenceDict.keys())
            eliminationOrder = sorted(list(eliminationVariables))

        "*** YOUR CODE HERE ***"
        factors = bayesNet.getAllCPTsWithEvidence(evidenceDict)
        for eli_var in eliminationOrder:
            # joinFactorsByVariable Returns a tuple of (factors not joined, resulting
factor from joinFactors)
            factors, result_factor = joinFactorsByVariable(factors, eli_var)
            # discard the case when only one unconditioned variable
            if len(result_factor.unconditionedVariables()) == 1:
```

```
                    continue
                # Get the factor after elimination
                new_factor = eliminate(result_factor, eli_var)
                # Append the new factor to the list
                factors.append(new_factor)
            # Join variables not eliminated
            result = joinFactors(factors)
            # Normalize so that the probabilities add up to 1
            return normalize(result)
            "*** END YOUR CODE HERE ***"


    return inferenceByVariableElimination

inferenceByVariableElimination = inferenceByVariableEliminationWithCallTracking()

def sampleFromFactorRandomSource(randomSource=None):
    if randomSource is None:
        randomSource = random.Random()

    def sampleFromFactor(factor, conditionedAssignments=None):
        """
        Sample an assignment for unconditioned variables in factor with
        probability equal to the probability in the row of factor
        corresponding to that assignment.

        factor:                    The factor to sample from.
        conditionedAssignments: A dict of assignments for all conditioned
                                   variables in the factor.  Can only be None
                                   if there are no conditioned variables in
                                   factor, otherwise must be nonzero.

        Useful for inferenceByLikelihoodWeightingSampling

        Returns an assignmentDict that contains the conditionedAssignments but
        also a random assignment of the unconditioned variables given their
        probability.
        """
        if conditionedAssignments is None and len(factor.conditionedVariables()) >
0:
            raise ValueError("Conditioned assignments must be provided since \n"
+
                             "this factor has conditionedVariables: " + "\n" +
                             str(factor.conditionedVariables()))
```

```
        elif conditionedAssignments is not None:
            conditionedVariables = set([var for var in
conditionedAssignments.keys()])

            if not
conditionedVariables.issuperset(set(factor.conditionedVariables())):
                raise ValueError("Factor's conditioned variables need to be a subset
of the \n"
                                 + "conditioned assignments passed in. \n"
+ \
                                 "conditionedVariables: " +
str(conditionedVariables) + "\n" +
                                 "factor.conditionedVariables: " +
str(set(factor.conditionedVariables())))

            # Reduce the domains of the variables that have been
            # conditioned upon for this factor
            newVariableDomainsDict = factor.variableDomainsDict()
            for (var, assignment) in conditionedAssignments.items():
                newVariableDomainsDict[var] = [assignment]

            # Get the (hopefully) smaller conditional probability table
            # for this variable
            CPT = factor.specializeVariableDomains(newVariableDomainsDict)
        else:
            CPT = factor

        # Get the probability of each row of the table (along with the
        # assignmentDict that it corresponds to)
        assignmentDicts = sorted([assignmentDict for assignmentDict in
CPT.getAllPossibleAssignmentDicts()])
        assignmentDictProbabilities = [CPT.getProbability(assignmentDict) for
assignmentDict in assignmentDicts]

        # calculate total probability in the factor and index each row by the
        # cumulative sum of probability up to and including that row
        currentProbability = 0.0
        probabilityRange = []
        for i in range(len(assignmentDicts)):
            currentProbability += assignmentDictProbabilities[i]
            probabilityRange.append(currentProbability)

        totalProbability = probabilityRange[-1]
```

```
            # sample an assignment with probability equal to the probability in the row
            # for that assignment in the factor
            pick = randomSource.uniform(0.0, totalProbability)
            for i in range(len(assignmentDicts)):
                if pick <= probabilityRange[i]:
                    return assignmentDicts[i]

        return sampleFromFactor

sampleFromFactor = sampleFromFactorRandomSource()

class DiscreteDistribution(dict):
    """
    A DiscreteDistribution models belief distributions and weight distributions
    over a finite set of discrete keys.
    """
    def __getitem__(self, key):
        self.setdefault(key, 0)
        return dict.__getitem__(self, key)

    def copy(self):
        """
        Return a copy of the distribution.
        """
        return DiscreteDistribution(dict.copy(self))

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

    def total(self):
        """
        Return the sum of values for all keys.
        """
        return float(sum(self.values()))
```

```
class InferenceModule:
    """
    An inference module tracks a belief distribution over a ghost's location.
    """
    ################################################
    # Useful methods for all inference modules #
    ################################################

    def __init__(self, ghostAgent):
        """
        Set the ghost agent for later access.
        """
        self.ghostAgent = ghostAgent
        self.index = ghostAgent.index
        self.obs = []    # most recent observation position

    def getJailPosition(self):
        return (2 * self.ghostAgent.index - 1, 1)

    def getPositionDistributionHelper(self, gameState, pos, index, agent):
        try:
            jail = self.getJailPosition()
            gameState = self.setGhostPosition(gameState, pos, index + 1)
        except TypeError:
            jail = self.getJailPosition(index)
            gameState = self.setGhostPositions(gameState, pos)
        pacmanPosition = gameState.getPacmanPosition()
        ghostPosition = gameState.getGhostPosition(index + 1)    # The position you
set
        dist = DiscreteDistribution()
        if pacmanPosition == ghostPosition:    # The ghost has been caught!
            dist[jail] = 1.0
            return dist
        pacmanSuccessorStates = game.Actions.getLegalNeighbors(pacmanPosition,
\
                    gameState.getWalls())    # Positions Pacman can move to
        if ghostPosition in pacmanSuccessorStates:    # Ghost could get caught
            mult = 1.0 / float(len(pacmanSuccessorStates))
            dist[jail] = mult
        else:
            mult = 0.0
        actionDist = agent.getDistribution(gameState)
        for action, prob in actionDist.items():
```

```
                successorPosition = game.Actions.getSuccessor(ghostPosition, action)
                if successorPosition in pacmanSuccessorStates:    # Ghost could get
caught
                    denom = float(len(actionDist))
                    dist[jail] += prob * (1.0 / denom) * (1.0 - mult)
                    dist[successorPosition] = prob * ((denom - 1.0) / denom) * (1.0 -
mult)
                else:
                    dist[successorPosition] = prob * (1.0 - mult)
            return dist

        def getPositionDistribution(self, gameState, pos, index=None, agent=None):
            """
            Return a distribution over successor positions of the ghost from the
            given gameState. You must first place the ghost in the gameState, using
            setGhostPosition below.
            """
            if index == None:
                index = self.index - 1
            if agent == None:
                agent = self.ghostAgent
            return self.getPositionDistributionHelper(gameState, pos, index, agent)

        def setGhostPosition(self, gameState, ghostPosition, index):
            """
            Set the position of the ghost for this inference module to the specified
            position in the supplied gameState.

            Note that calling setGhostPosition does not change the position of the
            ghost in the GameState object used for tracking the true progression of
            the game.    The code in inference.py only ever receives a deep copy of
            the GameState object which is responsible for maintaining game state,
            not a reference to the original object.    Note also that the ghost
            distance observations are stored at the time the GameState object is
            created, so changing the position of the ghost will not affect the
            functioning of observe.
            """
            conf = game.Configuration(ghostPosition, game.Directions.STOP)
            gameState.data.agentStates[index] = game.AgentState(conf, False)
            return gameState

        def setGhostPositions(self, gameState, ghostPositions):
            """
            Sets the position of all ghosts to the values in ghostPositions.
```

```
        """
        for index, pos in enumerate(ghostPositions):
            conf = game.Configuration(pos, game.Directions.STOP)
            gameState.data.agentStates[index + 1] = game.AgentState(conf, False)
        return gameState

    def observe(self, gameState):
        """
        Collect the relevant noisy distance observation and pass it along.
        """
        distances = gameState.getNoisyGhostDistances()
        if len(distances) >= self.index:    # Check for missing observations
            obs = distances[self.index - 1]
            self.obs = obs
            self.observeUpdate(obs, gameState)

    def initialize(self, gameState):
        """
        Initialize beliefs to a uniform distribution over all legal positions.
        """
        self.legalPositions = [p for p in gameState.getWalls().asList(False) if p[1] >
1]
        self.allPositions = self.legalPositions + [self.getJailPosition()]
        self.initializeUniformly(gameState)

    ######################################
    # Methods that need to be overridden #
    ######################################

    def initializeUniformly(self, gameState):
        """
        Set the belief state to a uniform prior belief over all positions.
        """
        raise NotImplementedError

    def observeUpdate(self, observation, gameState):
        """
        Update beliefs based on the given distance observation and gameState.
        """
        raise NotImplementedError

    def elapseTime(self, gameState):
        """
        Predict beliefs for the next time step from a gameState.
```

```
        """
        raise NotImplementedError

    def getBeliefDistribution(self):
        """
        Return the agent's current belief state, a distribution over ghost
        locations conditioned on all evidence so far.
        """
        raise NotImplementedError


class ExactInference(InferenceModule):
    """
    The exact dynamic inference module should use forward algorithm updates to
    compute the exact belief function at each time step.
    """
    def initializeUniformly(self, gameState):
        """
        Begin with a uniform distribution over legal ghost positions (i.e., not
        including the jail position).
        """
        self.beliefs = DiscreteDistribution()
        for p in self.legalPositions:
            self.beliefs[p] = 1.0
        self.beliefs.normalize()

    def getBeliefDistribution(self):
        return self.beliefs


class ParticleFilter(InferenceModule):
    """
    A particle filter for approximately tracking a single ghost.
    """
    def __init__(self, ghostAgent, numParticles=300):
        InferenceModule.__init__(self, ghostAgent)
        self.setNumParticles(numParticles)

    def setNumParticles(self, numParticles):
        self.numParticles = numParticles
```

Question 2 (3 points): Join Factors
Implement the joinFactors function in factorOperations.py. It takes in a list of Factors and returns a new Factor whose probability entries are the product of the corresponding rows of the input Factors.
joinFactors can be used as the product rule, for example, if we have a factor of the form P(X|Y) and another factor of the form P(Y), then joining these factors will yield P(X,Y). So, joinFactors allows us to incorporate probabilities for conditioned variables (in this case, Y). However, you should not assume that joinFactors is called on probability tables – it is possible to call joinFactors on Factors whose rows do not sum to 1.

Question 3 (2 points): Eliminate (not ghosts yet)
Implement the eliminate function in factorOperations.py. It takes a Factor and a variable to eliminate and returns a new Factor that does not contain that variable. This corresponds to summing all of the entries in the Factor which only differ in the value of the variable being eliminated.

Solution:

```
# factorOperations.py
# -------------------
# Licensing Information:  You are free to use or extend these projects for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to UC Berkeley, including a link to http://ai.berkeley.edu.


from typing import List
from bayesNet import Factor
import functools
from util import raiseNotDefined

def joinFactorsByVariableWithCallTracking(callTrackingList=None):


    def joinFactorsByVariable(factors: List[Factor], joinVariable: str):
        """
        Input factors is a list of factors.
        Input joinVariable is the variable to join on.

        This function performs a check that the variable that is being joined on
        appears as an unconditioned variable in only one of the input factors.

        Then, it calls your joinFactors on all of the factors in factors that
        contain that variable.
```

```
            Returns a tuple of
            (factors not joined, resulting factor from joinFactors)
            """

            if not (callTrackingList is None):
                callTrackingList.append(('join', joinVariable))

            currentFactorsToJoin =    [factor for factor in factors if joinVariable in factor.variablesSet()]
            currentFactorsNotToJoin = [factor for factor in factors if joinVariable not in factor.variablesSet()]

            # typecheck portion
            numVariableOnLeft = len([factor for factor in currentFactorsToJoin if joinVariable in factor.unconditionedVariables()])
            if numVariableOnLeft > 1:
                print("Factor failed joinFactorsByVariable typecheck: ", factor)
                raise ValueError("The joinBy variable can only appear in one factor as an \nunconditioned variable. \n" +
                                 "joinVariable: " + str(joinVariable) + "\n" +
                                 ",                                 ".join(map(str, [factor.unconditionedVariables() for factor in currentFactorsToJoin])))

            joinedFactor = joinFactors(currentFactorsToJoin)
            return currentFactorsNotToJoin, joinedFactor

        return joinFactorsByVariable

joinFactorsByVariable = joinFactorsByVariableWithCallTracking()

########### ########### ###########
########### QUESTION 2    ###########
########### ########### ###########

def joinFactors(factors: List[Factor]):
    """
    Input factors is a list of factors.

    You should calculate the set of unconditioned variables and conditioned
    variables for the join of those factors.

    Return a new factor that has those variables and whose probability entries
    are product of the corresponding rows of the input factors.
```

```
    You may assume that the variableDomainsDict for all the input
    factors are the same, since they come from the same BayesNet.

    joinFactors will only allow unconditionedVariables to appear in
    one input factor (so their join is well defined).

    Hint: Factor methods that take an assignmentDict as input
    (such as getProbability and setProbability) can handle
    assignmentDicts that assign more variables than are in that factor.

    Useful functions:
    Factor.getAllPossibleAssignmentDicts
    Factor.getProbability
    Factor.setProbability
    Factor.unconditionedVariables
    Factor.conditionedVariables
    Factor.variableDomainsDict
    """

    # typecheck portion
    setsOfUnconditioned = [set(factor.unconditionedVariables()) for factor in factors]
    if len(factors) > 1:
        intersect = functools.reduce(lambda x, y: x & y, setsOfUnconditioned)
        if len(intersect) > 0:
            print("Factor failed joinFactors typecheck: ", factor)
            raise ValueError("unconditionedVariables can only appear in one factor.
\n"
                    + "unconditionedVariables: " + str(intersect) +
                    "\nappear in more than one input factor.\n" +
                    "Input factors: \n" +
                    "\n".join(map(str, factors)))


    "*** YOUR CODE HERE ***"
    # Initialize variables to store the result
    unconditioned_var = []
    conditioned_var = []
    factors = list(factors)
    variableDomainsDict = factors[0].variableDomainsDict()

    # Extract conditioned/unconditioned variables
    for factor in factors:
        # Add new unconditioned variables
        unconditioned_var += factor.unconditionedVariables()
```

```
            # Add new conditioned variables
            condition = factor.conditionedVariables()
            for var in condition:
                # Append new variables only
                # Possibly use set here to reduce one line of code.
                if var not in conditioned_var:
                    conditioned_var.append(var)
        # Eliminate redundant conditioned variables
        conditioned_var = [x for x in conditioned_var if x not in unconditioned_var]
        # Initialize the new factor returned
        result = Factor(unconditioned_var, conditioned_var, variableDomainsDict)
        # Get all possible assignments
        assignment_dicts = result.getAllPossibleAssignmentDicts()
        # Compute the new probabilities
        for assignment in assignment_dicts:
            prob = 1
            for factor in factors:
                # product of the corresponding rows of the input Factors.
                prob *= factor.getProbability(assignment)
            result.setProbability(assignment, prob)
        return result
        "*** END YOUR CODE HERE ***"

########### ########### ###########
########### QUESTION 3    ###########
########### ########### ###########

def eliminateWithCallTracking(callTrackingList=None):

    def eliminate(factor: Factor, eliminationVariable: str):
        """
        Input factor is a single factor.
        Input eliminationVariable is the variable to eliminate from factor.
        eliminationVariable must be an unconditioned variable in factor.

        You should calculate the set of unconditioned variables and conditioned
        variables for the factor obtained by eliminating the variable
        eliminationVariable.

        Return a new factor where all of the rows mentioning
        eliminationVariable are summed with rows that match
        assignments on the other variables.

        Useful functions:
```

```
        Factor.getAllPossibleAssignmentDicts
        Factor.getProbability
        Factor.setProbability
        Factor.unconditionedVariables
        Factor.conditionedVariables
        Factor.variableDomainsDict
        """
        # autograder tracking -- don't remove
        if not (callTrackingList is None):
            callTrackingList.append(('eliminate', eliminationVariable))

        # typecheck portion
        if eliminationVariable not in factor.unconditionedVariables():
            print("Factor failed eliminate typecheck: ", factor)
            raise ValueError("Elimination variable is not an unconditioned variable
" \
                             + "in this factor\n" +
                             "eliminationVariable: " + str(eliminationVariable) +
\
                             "\nunconditionedVariables:"                    +
str(factor.unconditionedVariables()))

        if len(factor.unconditionedVariables()) == 1:
            print("Factor failed eliminate typecheck: ", factor)
            raise ValueError("Factor has only one unconditioned variable, so you "
\
                    + "can't eliminate \nthat variable.\n" + \
                    "eliminationVariable:" + str(eliminationVariable) + "\n" +\
                    "unconditionedVariables:                "                    +
str(factor.unconditionedVariables()))

        "*** YOUR CODE HERE ***"
        # Get a copy of all unconditioned variables except for the one we want
        # to eliminate
        unconditioned_var = [x for x in factor.unconditionedVariables()
            if x != eliminationVariable]
        # Get all conditioned variables
        conditioned_var = factor.conditionedVariables()
        # Get all possible variable domains
        variableDomainsDict = factor.variableDomainsDict()
        # Find the region of elimination variable we want to sum over
        sum_region = variableDomainsDict[eliminationVariable]
        # Initialize the new factor returned
        result = Factor(unconditioned_var, conditioned_var, variableDomainsDict)
```

```
            # get all possible assignments
            assignment_dicts = result.getAllPossibleAssignmentDicts()
            # Compute the new probabilities
            for assignment in assignment_dicts:
                # Initialize the resulting probability
                prob = 0
                for eli_var_val in sum_region:
                    # sum over the variable to be eliminated
                    assignment[eliminationVariable] = eli_var_val
                    # add over the probability
                    prob += factor.getProbability(assignment)
                result.setProbability(assignment, prob)
            return result
            "*** END YOUR CODE HERE ***"

        return eliminate

eliminate = eliminateWithCallTracking()
```

Question 4 (2 points): Variable Elimination

Implement the inferenceByVariableElimination function in inference.py. It answers a probabilistic query, which is represented using a BayesNet, a list of query variables, and the evidence.

Question 5a: DiscreteDistribution Class

Unfortunately, having timesteps will grow our graph far too much for variable elimination to be viable. Instead, we will use the Forward Algorithm for HMM's for exact inference, and Particle Filtering for even faster but approximate inference.

For the rest of the project, we will be using the DiscreteDistribution class defined in inference.py to model belief distributions and weight distributions. This class is an extension of the built-in Python dictionary class, where the keys are the different discrete elements of our distribution, and the corresponding values are proportional to the belief or weight that the distribution assigns that element. This question asks you to fill in the missing parts of this class, which will be crucial for later questions (even though this question itself is worth no points).

First, fill in the normalize method, which normalizes the values in the distribution to sum to one, but keeps the proportions of the values the same. Use the total method to find the sum of the values in the distribution. For an empty distribution or a distribution where all of the values are zero, do nothing. Note that this method modifies the distribution directly, rather than returning a new distribution.

Second, fill in the sample method, which draws a sample from the distribution, where the probability that a key is sampled is proportional to its corresponding value. Assume that the distribution is not empty, and not all of the values are zero. Note that the distribution does not necessarily have to be normalized prior to calling this method. You may find Python's built-in random.random() function useful for this question.

Question 5b (1 point): Observation Probability

In this question, you will implement the getObservationProb method in the InferenceModule base class in inference.py. This method takes in an observation (which is a noisy reading of the distance to the ghost), Pacman's position, the ghost's position, and the position of the ghost's jail, and returns the probability of the noisy distance reading given Pacman's position and the ghost's position. In other words, we want to return P(noisyDistance|pacmanPosition,ghostPosition).

The distance sensor has a probability distribution over distance readings given the true distance from Pacman to the ghost. This distribution is modeled by the function busters.getObservationProbability(noisyDistance, trueDistance), which returns P(noisyDistance|trueDistance) and is provided for you. You should use this function to help you solve the problem, and use the provided manhattanDistance function to find the distance between Pacman's location and the ghost's location.

However, there is the special case of jail that we have to handle as well. Specifically, when we capture a ghost and send it to the jail location, our distance sensor deterministically returns None, and nothing else (observation = None if and only if ghost is in jail). One consequence of this is that if the ghost's position is the jail position,

then the observation is None with probability 1, and everything else with probability 0. Make sure you handle this special case in your implementation; we effectively have a different set of rules for whenever ghost is in jail, as well as whenever observation is None.

Question 6 (2 points): Exact Inference Observation

In this question, you will implement the observeUpdate method in ExactInference class of inference.py to correctly update the agent's belief distribution over ghost positions given an observation from Pacman's sensors. You are implementing the online belief update for observing new evidence. The observeUpdate method should, for this problem, update the belief at every position on the map after receiving a sensor reading. You should iterate your updates over the variable self.allPositions which includes all legal positions plus the special jail position. Beliefs represent the probability that the ghost is at a particular location, and are stored as a DiscreteDistribution object in a field called self.beliefs, which you should update.

Before typing any code, write down the equation of the inference problem you are trying to solve. You should use the function self.getObservationProb that you wrote in the last question, which returns the probability of an observation given Pacman's position, a potential ghost position, and the jail position. You can obtain Pacman's position using gameState.getPacmanPosition(), and the jail position using self.getJailPosition().

In the Pacman display, high posterior beliefs are represented by bright colors, while low beliefs are represented by dim colors. You should start with a large cloud of belief that shrinks over time as more evidence accumulates. As you watch the test cases, be sure that you understand how the squares converge to their final coloring.

Question 7 (2 points): Exact Inference with Time Elapse

In the previous question you implemented belief updates for Pacman based on his observations. Fortunately, Pacman's observations are not his only source of knowledge about where a ghost may be. Pacman also has knowledge about the ways that a ghost may move; namely that the ghost can not move through a wall or more than one space in one time step.

To understand why this is useful to Pacman, consider the following scenario in which there is Pacman and one Ghost. Pacman receives many observations which indicate the ghost is very near, but then one which indicates the ghost is very far. The reading indicating the ghost is very far is likely to be the result of a buggy sensor. Pacman's prior knowledge of how the ghost may move will decrease the impact of this reading since Pacman knows the ghost could not move so far in only one move.

In this question, you will implement the elapseTime method in ExactInference in inference.py. The elapseTime step should, for this problem, update the belief at every position on the map after one time step elapsing. Your agent has access to the action distribution for the ghost through self.getPositionDistribution. In order to obtain the distribution over new positions for the ghost, given its previous position, use this line of code:

newPosDist = self.getPositionDistribution(gameState, oldPos)

Where oldPos refers to the previous ghost position. newPosDist is a DiscreteDistribution object, where for each position p in self.allPositions, newPosDist[p] is the probability that the ghost is at position p at time t + 1, given that the ghost is at position oldPos at time t. Note that this call can be fairly expensive, so if your code is timing out, one thing to think about is whether or not you can reduce the number of calls to self.getPositionDistribution.

Before typing any code, write down the equation of the inference problem you are trying to solve. In order to test your predict implementation separately from your update implementation in the previous question, this question will not make use of your update implementation.

Since Pacman is not observing the ghost's actions, these actions will not impact Pacman's beliefs. Over time, Pacman's beliefs will come to reflect places on the board where he believes ghosts are most likely to be given the geometry of the board and ghosts' possible legal moves, which Pacman already knows.

For the tests in this question we will sometimes use a ghost with random movements and other times we will use the GoSouthGhost. This ghost tends to move south so over time, and without any observations, Pacman's belief distribution should begin to focus around the bottom of the board. To see which ghost is used for each test case you can look in the .test files.

Question 9 (1 points): Approximate Inference Initialization and Beliefs

Approximate inference is very trendy among ghost hunters this season. For the next few questions, you will implement a particle filtering algorithm for tracking a single ghost.

First, implement the functions initializeUniformly and getBeliefDistribution in the ParticleFilter class in inference.py. A particle (sample) is a ghost position in this inference problem. Note that, for initialization, particles should be evenly (not randomly) distributed across legal positions in order to ensure a uniform prior. We recommend thinking about how the mod operator is useful for initializeUniformly.

Note that the variable you store your particles in must be a list. A list is simply a collection of unweighted variables (positions in this case). Storing your particles as any other data type, such as a dictionary, is incorrect and will produce errors. The getBeliefDistribution method then takes the list of particles and converts it into a DiscreteDistribution object.

Question 10 (2 points): Approximate Inference Observation

Next, we will implement the observeUpdate method in the ParticleFilter class in inference.py. This method constructs a weight distribution over self.particles where the weight of a particle is the probability of the observation given Pacman's position and that particle location. Then, we resample from this weighted distribution to construct our new list of particles.

You should again use the function self.getObservationProb to find the probability of an observation given Pacman's position, a potential ghost position, and the jail position.

The sample method of the DiscreteDistribution class will also be useful. As a reminder, you can obtain Pacman's position using gameState.getPacmanPosition(), and the jail position using self.getJailPosition().
There is one special case that a correct implementation must handle. When all particles receive zero weight, the list of particles should be reinitialized by calling initializeUniformly. The total method of the DiscreteDistribution may be useful.

Question 11 (2 points): Approximate Inference with Time Elapse
Implement the elapseTime function in the ParticleFilter class in inference.py. This function should construct a new list of particles that corresponds to each existing particle in self.particles advancing a time step, and then assign this new list back to self.particles. When complete, you should be able to track ghosts nearly as effectively as with exact inference.
Note that in this question, we will test both the elapseTime function in isolation, as well as the full implementation of the particle filter combining elapseTime and observe.
As in the elapseTime method of the ExactInference class, you should use:
newPosDist = self.getPositionDistribution(gameState, oldPos)
This line of code obtains the distribution over new positions for the ghost, given its previous position (oldPos). The sample method of the DiscreteDistribution class will also be useful.

Solution: See Q1

Project 5: Machine Learning

Question 1 (6 points): Perceptron

Before starting this part, be sure you have numpy and matplotlib installed!

In this part, you will implement a binary perceptron. Your task will be to complete the implementation of the PerceptronModel class in models.py.

For the perceptron, the output labels will be either 1 or −1, meaning that data points (x, y) from the dataset will have y be a nn.Constant node that contains either 1 or −1 as its entries.

We have already initialized the perceptron weights self.w to be a 1 by dimensions parameter node. The provided code will include a bias feature inside x when needed, so you will not need a separate parameter for the bias.

Your tasks are to:

- Implement the run(self, x) method. This should compute the dot product of the stored weight vector and the given input, returning an nn.DotProduct object.
- Implement get_prediction(self, x), which should return 1 if the dot product is non-negative or −1 otherwise. You should use nn.as_scalar to convert a scalar Node into a Python floating-point number.
- Write the train(self) method. This should repeatedly loop over the data set and make updates on examples that are misclassified. Use the update method of the nn.Parameter class to update the weights. When an entire pass over the data set is completed without making any mistakes, 100% training accuracy has been achieved, and training can terminate.
- In this project, the only way to change the value of a parameter is by calling parameter.update(direction, multiplier), which will perform the update to the weights: weights←weights+direction·multiplier

The direction argument is a Node with the same shape as the parameter, and the multiplier argument is a Python scalar. Additionally, use iterate_once to loop over the dataset; see Provided Code (Part I) for usage.

Question 2 (6 points): Non-linear Regression

For this question, you will train a neural network to approximate sin(x) over $[-2\pi, 2\pi]$. You will need to complete the implementation of the RegressionModel class in models.py. For this problem, a relatively simple architecture should suffice (see Neural Network Tips for architecture tips). Use nn.SquareLoss as your loss.

Your tasks are to:

- Implement RegressionModel.__init__ with any needed initialization.
- Implement RegressionModel.run to return a batch_size by 1 node that represents your model's prediction.
- Implement RegressionModel.get_loss to return a loss for given inputs and target outputs.
- Implement RegressionModel.train, which should train your model using gradient-based updates.

There is only a single dataset split for this task (i.e., there is only training data and no validation data or test set). Your implementation will receive full points if it gets a loss

of 0.02 or better, averaged across all examples in the dataset. You may use the training loss to determine when to stop training (use nn.as_scalar to convert a loss node to a Python number). Note that it should take the model a few minutes to train.
Suggested network architecture (added 2022/11/28): Normally, you would need to use trial-and-error to find working hyperparameters. Below is a set of hyperparameters that worked for us (took less than a minute to train on the hive machines), but feel free to experiment and use your own.
Hidden layer size 512
Batch size 200
Learning rate 0.05
One hidden layer (2 linear layers in total)

Question 3 (6 points): Digit Classification
For this question, you will train a network to classify handwritten digits from the MNIST dataset.
Each digit is of size 28 by 28 pixels, the values of which are stored in a 784-dimensional vector of floating point numbers. Each output we provide is a 10-dimensional vector which has zeros in all positions, except for a one in the position corresponding to the correct class of the digit.
Complete the implementation of the DigitClassificationModel class in models.py. The return value from DigitClassificationModel.run() should be a batch_size by 10 node containing scores, where higher scores indicate a higher probability of a digit belonging to a particular class (0-9). You should use nn.SoftmaxLoss as your loss. Do not put a ReLU activation in the last linear layer of the network.
For both this question and Q4, in addition to training data, there is also validation data and a test set. You can use dataset.get_validation_accuracy() to compute validation accuracy for your model, which can be useful when deciding whether to stop training.
Suggested network architecture (added 2022/11/28): Normally, you would need to use trial-and-error to find working hyperparameters. Below is a set of hyperparameters that worked for us (took less than a minute to train on the hive machines), but feel free to experiment and use your own.
Hidden layer size 200
Batch size 100
Learning rate 0.5
One hidden layer (2 linear layers in total)

Question 4 (7 points): Language Identification
Language identification is the task of figuring out, given a piece of text, what language the text is written in. For example, your browser might be able to detect if you've visited a page in a foreign language and offer to translate it for you.
In this project, we're going to build a smaller neural network model that identifies language for one word at a time. Our dataset consists of words in five languages, such as the table below:
Word Language

discussed English
eternidad Spanish
itseänne Finnish
paleis Dutch
mieszkać Polish

Your task is to complete the implementation of the LanguageIDModel in models.py.

Solution:

```
import nn

class PerceptronModel(object):
    def __init__(self, dimensions):
        """
        Initialize a new Perceptron instance.

        A perceptron classifies data points as either belonging to a particular
        class (+1) or not (-1). `dimensions` is the dimensionality of the data.
        For example, dimensions=2 would mean that the perceptron must classify
        2D points.
        """
        self.w = nn.Parameter(1, dimensions)

    def get_weights(self):
        """
        Return a Parameter instance with the current weights of the perceptron.
        """
        return self.w

    def run(self, x):
        """
        Calculates the score assigned by the perceptron to a data point x.

        Inputs:
            x: a node with shape (1 x dimensions)
        Returns: a node containing a single number (the score)
        """
        "*** YOUR CODE HERE ***"
        return nn.DotProduct(self.w, x)

    def get_prediction(self, x):
        """
        Calculates the predicted class for a single data point `x`.

        Returns: 1 or -1
```

```
        """
        "*** YOUR CODE HERE ***"
        return 1 if nn.as_scalar(self.run(x)) >= 0 else -1

    def train(self, dataset):
        """
        Train the perceptron until convergence.
        """
        "*** YOUR CODE HERE ***"
        # Initialize batch size
        batch_size = 1
        while True:
            accurate_classification = True
            # Iterate through all data points and update our model if needed
            for x, y in dataset.iterate_once(batch_size):
                # Obtain the classification for one data point
                prediction = self.get_prediction(x)
                # Update parameter if wrong prediction
                if prediction != nn.as_scalar(y):
                    self.w.update(x, nn.as_scalar(y))
                    accurate_classification = False
            # End iteration if clear pass
            if accurate_classification:
                break


class RegressionModel(object):
    """
    A neural network model for approximating a function that maps from real
    numbers to real numbers. The network should be sufficiently large to be able
    to approximate sin(x) on the interval [-2pi, 2pi] to reasonable precision.
    """
    def __init__(self):
        # Initialize your model parameters here
        "*** YOUR CODE HERE ***"
        # Define a two-layer network with recommended hyperparameters
        self.w0 = nn.Parameter(1, 512)
        self.w1 = nn.Parameter(512, 1)
        self.b0 = nn.Parameter(1, 512)
        self.b1 = nn.Parameter(1, 1)

    def run(self, x):
        """
        Runs the model for a batch of examples.
```

```
        Inputs:
            x: a node with shape (batch_size x 1)
        Returns:
            A node with shape (batch_size x 1) containing predicted y-values
        """
        "*** YOUR CODE HERE ***"
        # Run layer by layer as instructed in the project writeup.
        xw0 = nn.Linear(x, self.w0)
        h1 = nn.ReLU(nn.AddBias(xw0, self.b0))
        xw1 = nn.Linear(h1, self.w1)
        result = nn.AddBias(xw1, self.b1)
        return result

    def get_loss(self, x, y):
        """
        Computes the loss for a batch of examples.

        Inputs:
            x: a node with shape (batch_size x 1)
            y: a node with shape (batch_size x 1), containing the true y-values
                to be used for training
        Returns: a loss node
        """
        "*** YOUR CODE HERE ***"
        # Squared loss used as required
        return nn.SquareLoss(self.run(x), y)

    def train(self, dataset):
        """
        Trains the model.
        """
        "*** YOUR CODE HERE ***"
        # Define recommended hyperparameters.
        batch_size = 200
        alpha = 0.05
        # Run until loss is less than 0.02 as instructed
        while True:
            for x, y in dataset.iterate_once(batch_size):
                # Obtain the gradients as each iteration for parameter updates
                loss = self.get_loss(x, y)
                grad_wrt_w0, grad_wrt_w1, grad_wrt_b0, grad_wrt_b1 =
nn.gradients(loss,
                        [self.w0, self.w1, self.b0, self.b1])
```

```
                # Update parameters as needed
                self.w0.update(grad_wrt_w0, -alpha)
                self.w1.update(grad_wrt_w1, -alpha)
                self.b0.update(grad_wrt_b0, -alpha)
                self.b1.update(grad_wrt_b1, -alpha)
            # Compute loss again to see if stop threshold is met
            loss = nn.as_scalar(self.get_loss(nn.Constant(dataset.x),
nn.Constant(dataset.y)))
            # print(loss)
            if loss <= 0.02:
                break

class DigitClassificationModel(object):
    """
    A model for handwritten digit classification using the MNIST dataset.

    Each handwritten digit is a 28x28 pixel grayscale image, which is flattened
    into a 784-dimensional vector for the purposes of this model. Each entry in
    the vector is a floating point number between 0 and 1.

    The goal is to sort each digit into one of 10 classes (number 0 through 9).

    (See RegressionModel for more information about the APIs of different
    methods here. We recommend that you implement the RegressionModel before
    working on this part of the project.)
    """
    def __init__(self):
        # Initialize your model parameters here
        "*** YOUR CODE HERE ***"
        # Define a two-layer network with recommended hyperparameters
        self.w0 = nn.Parameter(784, 200)
        self.w1 = nn.Parameter(200, 10)
        self.b0 = nn.Parameter(1, 200)
        self.b1 = nn.Parameter(1, 10)

    def run(self, x):
        """
        Runs the model for a batch of examples.

        Your model should predict a node with shape (batch_size x 10),
        containing scores. Higher scores correspond to greater probability of
        the image belonging to a particular class.

        Inputs:
```

```
            x: a node with shape (batch_size x 784)
        Output:
            A node with shape (batch_size x 10) containing predicted scores
                (also called logits)
        """
        "*** YOUR CODE HERE ***"
        # Run layer by layer as instructed in the project writeup.
        xw0 = nn.Linear(x, self.w0)
        h1 = nn.ReLU(nn.AddBias(xw0, self.b0))
        xw1 = nn.Linear(h1, self.w1)
        result = nn.AddBias(xw1, self.b1)
        return result


    def get_loss(self, x, y):
        """
        Computes the loss for a batch of examples.

        The correct labels `y` are represented as a node with shape
        (batch_size x 10). Each row is a one-hot vector encoding the correct
        digit class (0-9).

        Inputs:
            x: a node with shape (batch_size x 784)
            y: a node with shape (batch_size x 10)
        Returns: a loss node
        """
        "*** YOUR CODE HERE ***"
        # Use softmaxLoss as required
        return nn.SoftmaxLoss(self.run(x), y)

    def train(self, dataset):
        """
        Trains the model.
        """
        "*** YOUR CODE HERE ***"
        # Define recommended hyperparameters.
        batch_size = 200
        alpha = 0.5
        # Run until accuracy is greater than 98.5% as instructed (added some buffer
here)
        while True:
            for x, y in dataset.iterate_once(batch_size):
                # Obtain the gradients as each iteration for parameter updates
```

```
                    loss = self.get_loss(x, y)
                    grad_wrt_w0, grad_wrt_w1, grad_wrt_b0, grad_wrt_b1 =
nn.gradients(loss,
                        [self.w0, self.w1, self.b0, self.b1])
                    # Update parameters as needed
                    self.w0.update(grad_wrt_w0, -alpha)
                    self.w1.update(grad_wrt_w1, -alpha)
                    self.b0.update(grad_wrt_b0, -alpha)
                    self.b1.update(grad_wrt_b1, -alpha)
                # Compute accuracy to see if stop threshold is met
                accuracy = dataset.get_validation_accuracy()
                # print(accuracy)
                if accuracy >= 0.98:
                    break

class LanguageIDModel(object):
    """
    A model for language identification at a single-word granularity.

    (See RegressionModel for more information about the APIs of different
    methods here. We recommend that you implement the RegressionModel before
    working on this part of the project.)
    """
    def __init__(self):
        # Our dataset contains words from five different languages, and the
        # combined alphabets of the five languages contain a total of 47 unique
        # characters.
        # You can refer to self.num_chars or len(self.languages) in your code
        self.num_chars = 47
        self.languages = ["English", "Spanish", "Finnish", "Dutch", "Polish"]

        # Initialize your model parameters here
        "*** YOUR CODE HERE ***"
        # Set the size of hidden layers to be 200 (sufficiently large maybe)
        # The output should have dimensionality of length of languages
        self.w0 = nn.Parameter(self.num_chars, 200)
        self.w1 = nn.Parameter(200, 200)
        self.w2 = nn.Parameter(200, len(self.languages))
        self.b0 = nn.Parameter(1, 200)
        self.b1 = nn.Parameter(1, 200)
        self.b2 = nn.Parameter(1, len(self.languages))

    def run(self, xs):
        """
```

```
Runs the model for a batch of examples.

Although words have different lengths, our data processing guarantees
that within a single batch, all words will be of the same length (L).

Here `xs` will be a list of length L. Each element of `xs` will be a
node with shape (batch_size x self.num_chars), where every row in the
array is a one-hot vector encoding of a character. For example, if we
have a batch of 8 three-letter words where the last word is "cat", then
xs[1] will be a node that contains a 1 at position (7, 0). Here the
index 7 reflects the fact that "cat" is the last word in the batch, and
the index 0 reflects the fact that the letter "a" is the inital (0th)
letter of our combined alphabet for this task.

Your model should use a Recurrent Neural Network to summarize the list
`xs` into a single node of shape (batch_size x hidden_size), for your
choice of hidden_size. It should then calculate a node of shape
(batch_size x 5) containing scores, where higher scores correspond to
greater probability of the word originating from a particular language.

Inputs:
    xs: a list with L elements (one per character), where each element
        is a node with shape (batch_size x self.num_chars)
Returns:
    A node with shape (batch_size x 5) containing predicted scores
        (also called logits)
"""
"*** YOUR CODE HERE ***"
# Run the first input to initialize f and h respectively
f_init = nn.Linear(xs[0], self.w0)
h = nn.ReLU(nn.AddBias(f_init, self.b0))
z = h
# Iterate through the rest of inputs and compute the output of each layer
for char in xs[1:]:
    # First sub-network
    f = nn.Linear(z, self.w1)
    h = nn.ReLU(nn.AddBias(f, self.b1))
    # Second sub-network
    f_x = nn.Linear(char, self.w0)
    h_x = nn.ReLU(nn.AddBias(f_x, self.b0))
    z = nn.Add(h, h_x)
# Final layer computation
f_final = nn.Linear(z, self.w2)
return nn.AddBias(f_final, self.b2)
```

```
    def get_loss(self, xs, y):
        """
        Computes the loss for a batch of examples.

        The correct labels `y` are represented as a node with shape
        (batch_size x 5). Each row is a one-hot vector encoding the correct
        language.

        Inputs:
            xs: a list with L elements (one per character), where each element
                is a node with shape (batch_size x self.num_chars)
            y: a node with shape (batch_size x 5)
        Returns: a loss node
        """
        "*** YOUR CODE HERE ***"
        # Use softmax loss in this case
        return nn.SoftmaxLoss(self.run(xs), y)

    def train(self, dataset):
        """
        Trains the model.
        """
        "*** YOUR CODE HERE ***"
        # Define hyperparameters.
        batch_size = 200
        alpha = 0.1
        # Run until accuracy is greater than 87% as instructed (add some buffer here
to pass autograder)
        while True:
            for x, y in dataset.iterate_once(batch_size):
                # Obtain the gradients as each iteration for parameter updates
                loss = self.get_loss(x, y)
                grad_wrt_w0,  grad_wrt_w1,  grad_wrt_w2,  grad_wrt_b0,
grad_wrt_b1, grad_wrt_b2 = nn.gradients(loss,
                        [self.w0, self.w1, self.w2, self.b0, self.b1, self.b2])
                # Update parameters as needed
                self.w0.update(grad_wrt_w0, -alpha)
                self.w1.update(grad_wrt_w1, -alpha)
                self.w2.update(grad_wrt_w2, -alpha)
                self.b0.update(grad_wrt_b0, -alpha)
                self.b1.update(grad_wrt_b1, -alpha)
                self.b2.update(grad_wrt_b2, -alpha)
            # Compute accuracy to see if stop threshold is met
```

```
accuracy = dataset.get_validation_accuracy()
# print(accuracy)
if accuracy >= 0.87:
        break
```