State Estimation

Now, let's assume a particular environment for the cat and mouse, and that the cat doesn't know exactly where the mouse is. The cat and mouse live in a world that is a 1 by 10 grid. On each step, the mouse moves to one of the two neighboring (E, W) grid squares uniformly at random, unless it is currently at location 0 or 9, in which case it stays where it is with probability 0.5 and moves to the neighboring square with probability 0.5.

The cat always knows where it is in this world, but not where the mouse is.

The cat's actions are to move east and west, and they always have the intended effect (except when they would cause it to move off the edge of the world, in which case they have no effect). The observation is the result of listening; on each step, the cat will "hear" a distance between 0 and 10. If it is 0, then the mouse is on the same square as the cat. If it is 1, then it is one square away; if 2, two squares, away; etc. If, for example, the cat is on location 1 and it "hears" a distance of 5,

then the mouse *must* be on location 6 (because for it to be 5 squares away in the other direction, it would have to be off the end of the world.)

Question 20: Imagine that cat starts at location 5 and observes that the mouse is 3 squares away. What is the cat's belief state about the location of the mouse?

```
mouseBelief = [0.0, 0.0, 0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.5, 0.0]
```

Question 21: Now, let the cat move east (and the mouse moves as described above). Before making any observation, what is the state of the system? (List both the cat's location and the belief state about the mouse's location).

```
catLocation = 6
mouseBelief = [0.0, 0.25, 0.0, 0.25, 0.0, 0.0, 0.0, 0.25, 0.0, 0.25]
```

Question 22: If the cat now observes that the mouse is 3 squares away, what is the new belief state about the location of the mouse?

```
mouseBelief = [0.0, 0.0, 0.0, 0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.5]
```

Question 23: If, instead, the cat observes that the mouse is 5 squares away (after the transition in question 20), what is the new belief state about the location of the mouse?

```
mouseBelief = [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

Question 24: Now, we'll construct a class that can do belief state estimation for this problem. The class definition and the first line of the initialization method are provided below.

```
class CatMouseState():
    def __init__(self, catLoc):
```

Write the rest of the initialization method, which takes as input the known location of the cat, and defines two instance variables: `self.catLocation`, which contains an integer from 0 to 9 indicating the cat's location; and `self.mouseBelief` which contains a list of 10 probabilities indicating the cat's belief about the mouse's location. The initial value of `self.mouseBelief` should be a representation of the distribution corresponding to having no information about the location of the mouse.

```
    def __init__(self, catLoc, nSpaces = 10):
        self.nSpaces = nSpaces
        self.catLocation = catLoc
        self.mouseBelief = nSpaces*[1.0/nSpaces]
```

Question 25: Provide the bodies of the following methods for the `CatMouseState` class. The procedures should modify the components of the object, and need not return any values. `actionUpdate` should update the state (cat's location and mouse belief state) given the cat's action (but remember that the mouse also moves), and `observationUpdate` should update the belief state based on its observation of the mouse.

```
#  Cat's action can be 'E' or 'W'
def actionUpdate(self, action):
    if action == 'E':
        self.catLocation = min(self.catLocation+1, self.nSpaces-1)
    else:
        self.catLocation = max(self.catLocation-1, 0)
    newBelief = self.nSpaces*[0.0]
    for i in range(self.nSpaces):
        newBelief[min(i+1, self.nSpaces-1)] += self.mouseBelief[i]*0.5
        newBelief[max(i-1, 0)] += self.mouseBelief[i]*0.5
    self.mouseBelief = newBelief

# Observation: distance to mouse, as described above.
def observationUpdate(self, obs):
    newBelief = self.nSpaces*[0.0]
    total = 0.0
    for loc in [self.catLocation + obs, self.catLocation - obs]:
        if  0 <= loc < self.nSpaces:
            newBelief[loc] = self.mouseBelief[loc]
            total += newBelief[loc]
    for i in range(self.nSpaces):
        newBelief[i] /= total
    self.mouseBelief = newBelief
```

5 Friend or Foe (15 points)

Imagine that you are defending a city and there is an aircraft flying toward you. It may be important to know whether that aircraft is a 'friend' or a 'foe' (enemy). Assume you have a radar sensor that can give you noisy information about the type of the aircraft that is approaching.

We will model this situation as an HMM, in which:

- The state space is described by two components `d` and `a`, where `d` is the number of miles (`0, 1, ..., 10`) away the target is and `a` is its *attitude* to you, which is either `'friend'` or `'foe'`. The values of `d` correspond to ranges of distance, so that, in fact, `d == 0` means the aircraft is somewhere between 0 and 1 miles away, etc.
- The observation space is `{'oFriend', 'oFoe'}`, which stands for *observed friend* and *observed foe*.
- There are no actions (or, if you prefer, a single action, which just waits a time step).
- The transition model is specified in the following Python method, which takes a state `s` as input and returns a `DDist` over possible next states:

```
def transitionModel(s, i):
    # note that the i (the input action) is ignored
    (d, a) = s
    if d == 0:
        return DDist({(0, a): 1.0})
    else
        return DDist({(d, a): 0.5, (d-1, a): 0.5})
```

- The observation model is as described in the following Python method, which takes a state `s` as input and returns a `DDist` over possible observations:

```
def observationModel(s):
    (d, a) = s
    if a == 'friend':
        return DDist({'oFriend': 1 - d / 20.0, 'oFoe' : d / 20.0})
    else:
        return DDist({'oFoe': 1 - d / 20.0, 'oFriend' : d / 20.0})
```

Questions:

1. Assume that the aircraft are approaching with a constant velocity. What velocity, in miles per time step, would generate the transition model over distance intervals given in the transition model?

Answer: 0.5 miles/step

2. Provide an alternative transition model in which friendly aircraft move 1.5 miles per time step (and the foes move at the same rate as the given model.)

```
def transitionModel(s, i):
    (d, a) = s
    if d == 0 or (d == 1 and a == 'friend'):
        return DDist({(0, a): 1.0})
    else
        if a == 'foe':
           return DDist({(d, a): 0.5, (d-1, a): 0.5})
        else:
           return DDist({(d-1, a): 0.5, (d-2, a): 0.5})
```

3. At what distance is our sensor most useful?

Answer: 0 miles

How does it behave at that distance?

Answer: perfectly

4. Using the original transition and observation models, if the initial belief state, b_0, is

`DDist({(10, 'friend') : 0.5, (10, 'foe'): 0.5})`

then what would b_0' be, after receiving observation $o_0 =$`'oFriend'`.

Answer: DDist((10, 'friend') : 0.5, (10, 'foe'): 0.5)

5. After that, what would the belief state b_1 be?

Answer: DDist((9, 'friend') : 0.25, (9, 'foe'): 0.25, (10, 'friend') : 0.25, (10, 'foe'): 0.25)

6. How many non-zero entries are there in b_4 (the answer will be the same for any sequence of observations)?

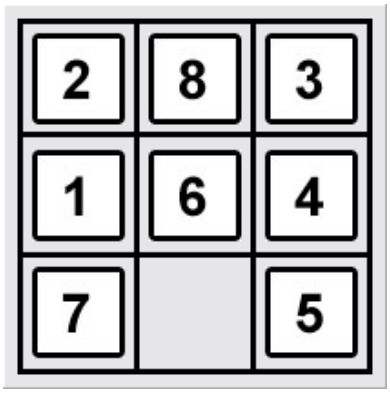
Answer: 10

7. Starting again from the initial belief state, and imagining the following observation sequence: `['oFriend', 'oFoe', 'oFriend', 'oFoe']`, is it more likely that the aircraft is a friend or a foe?

Answer: foe

7 A Puzzle (20 points)

Consider the standard *Eight Puzzle*. It has 8 tiles arranged on a 3 by 3 grid. Here is one possible arrangement of the tiles:



The tiles neighboring an empty space can be slid into the space. We can think of the operations on the puzzle as *moving the space* **up**, **down**, **left**, or **right**. It obviously cannot be moved beyond the bounds of the puzzle. In the example above, if we were to slide the space **up**, then the **6** tile would be in the bottom row and the space would be in the middle.

We can solve this problem using search. A state of the search would be an array of numbers (and None for the space) showing the location of each tile on grid. A goal state would be some specified arrangement of the tiles.

Assuming we apply pruning rule 1, but not the others, how many descendants are there for the state shown above:

- at level 1 (immediate children)?

  Answer: 3

- at level 2 (children of level 1)?

  Answer: 5

7.1 Search

We would like to identify the advantages and disadvantages of each of the following search methods **for this problem**. We assume, as always, that we use pruning rules 1 and 2.

Enter T or F in the boxes provided if the following statement is true or false, respectively.

- **depth-first, no dynamic programming**:
  - [T] This method is guaranteed to find a path.
  - [F] The path that is found is guaranteed to be short.
  - [T] The same board state may be visited multiple times.
  - [T] The agenda is likely to remain short (relative to the other methods).

- **depth-first, with dynamic programming**:
  - [T] This method is guaranteed to find a path.
  - [F] The path that is found is guaranteed to be short.
  - [F] The same board state may be visited multiple times.
  - [T] The agenda is likely to remain short (relative to the other methods).

- **breadth-first, no dynamic programming**:
  - [T] This method is guaranteed to find a path.
  - [T] The path that is found is guaranteed to be short.
  - [T] The same board state may be visited multiple times.
  - [F] The agenda is likely to remain short (relative to the other methods).

- **breadth-first, with dynamic programming**:
  - [T] This method is guaranteed to find a path.
  - [T] The path that is found is guaranteed to be short.
  - [F] The same board state may be visited multiple times.
  - [F] The agenda is likely to remain short (relative to the other methods).

7.2 State machine

We can describe the puzzle and its evolution using a state machine, in much the same way we described the wolf-goat-cabbage problem. We will specify the state with two components:

- A pair of *row*, *column* indices, indicating where the space is; and
- A list of three lists of items; each item is a digit between 1 and 8, or `None`. This describes the state of the board.

So, for example, we could represent the puzzle state above as:

```
((2, 1), [[2, 8, 3], [1, 6, 4], [7, None, 5]])
```

Technically speaking, we don't need the first component of the state (it could always be computed from the list of three lists), but it will simplify our coding if we maintain both representations.

A skeleton of the state machine defining the eight puzzle is shown on the next page. Fill in the definition of `getNextValues`. If a move would produce and illegal state, then the new state should be the same as the current state. The output of the machine should just be the next state.

```
class EightPuzzle(SM):
    startState = ((0, 2), [[0, 1, None], [2, 3, 4], [5, 6, 7]])
    size = 3
    offsets = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}
    legalInputs = ['up', 'down', 'left', 'right']

    def getNextValues(self, state, inp):
        ((p1, p2), board) = state
        bcopy = copy.deepcopy(board)

        (off1, off2) = self.offsets[inp]
        if 0 <= p1+off1 < self.size and 0 <= p2+off2 < self.size:
            c = board[p1][p2]
            np1 = clip(p1+off1,0,self.size-1)
            np2 = clip(p2+off2,0,self.size-1)
            nc = board[np1][np2]
            bcopy[p1][p2] = nc
            bcopy[np1][np2] = c
            newState = ((np1, np2), bcopy)
            return (newState, newState)
        else:
            return (state, state)
```

6.01 Introduction to Electrical Engineering and Computer Science I
Fall 2009