# COSC 76 Project 6: Hidden Markov Model

**Seungjae Jason Lee**

# Introduction

#### Objective

The objective of the project is to implement hidden markov model for the mazeworld problem. There is only one main task, which is to build filtering method.

#### Game Rule

For the game rule, we are still using the same maze concept as that of project 2. However, in this version, every block has a color on the floor, and a robot, which carries a sensor, reads the color of a floor as it moves along. The algorithm's task is to give a probabilistic distribution of the robot's location at each time step based on the sensor's reading of colors.

# Defining Matrices/Variables.

For the implementation, the hardest part was to figure out how to define states, observation and transition matrices. So, this section of the report will briefly explain how each of these variables are defined and implemented.

1. State: For the state, I made a vector with locations of floors not all locations. Although I could have added "walls" as a part of the state, it just made matrices bigger with bunch of unnecessary zeroes. Thus, I disregarded "walls" from the state. As for the indexing, I made dictionaries that can convert from index to location and vice versa.

2. Observation: For the observation matrix, I made a diagonal matrix where each i th diagonal entry refers to the probability of the robot reading a certain color in i th location (i refers to index, but each index has a corresponding location).

3. Transition Matrix: For building transition matrix, I said that there are equal chances of going to any of the following direction: North, West,South,East. However, if there are walls at any directions, I incremented the probability of the robot staying in the same place by 0.25. For example, If there are walls at northside and southside, then the probability of going east, west is 0.25 while staying in the same place is 0.5.

# HMM Class

HMM class is a hidden markov model built specifically for mazeworld problem. It takes a maze object from project 2, move sequence, and sensor sequence. Move sequence is a list of string where each slot has a direction: "N","S","E",and "W". Sensor sequence is a list of string where each slot has a color reading: "y","r","b", and "g". The following methods are helper methods that builds the problem:

- `__init__(self, maze, move_sequence, sensor_sequence)` : initalizes the class and makes matrices needed for filtering algorithm.
- `build_observation(self)` : builds a list of four sub-list and saves it under self.observation. Each list inside the list represents the observation probability for each color. If the color is same as the floor color, the probability is 0.88. Else, if the probability is different, its 0.04.
- `build_start_state(self)` : builds a starting probablistic distribution of the robot in the maze. The probability is equally divided by the total number of floors on a maze.
- `build_transition_matrix(self)` : builds self.transition, which is a transition matrix for user-given maze. The method goes through every floor and finds all the possible ways to move in a single time step. Then, it computes the probabilty for each way and forms a matrix. So, if there is 0.25 chance of ith floor moving to jth floor, self.transition[i,j] would be 0.25.
- `filtering(self)` : filtering finds distribution probability at each time step and stores it in self.forward. It also returns the last probability distribution. As for the implementation, I used inductive matrix multiplication: P(n) = alpha*Observation*mat*transition*mat*transposed*P(n-1) where P(0) is self.start*state. I used the formula explained in the lecture note for matrix implementation.
- `print_output(self, smooth = False)` : Once the filtering algorithm is over, this method prints the maze and the distribution at each time step and the color read by robot at each time step.

# Extension

**Extension 1:** For extension 1, I implemented `forward_backward` method which makes the distribution smoother. I saved forward distributions by running filtering. As for the backward distribution, I computed them using matrix multplication. For the backward distribution, I had my starting state to be a vector with 1 in every slot. Then, I went backward by multplying matrices in reverse order. Then, I combined forward and backward result by using element-wise multplication and saved the result in self.smoothing. Then, I made print_output method, which can print both filtering result and smoothing result. The picture below shows the output for both filtering and smoothing for maze1.maz for t = 0. Full result can be seen by running the file.

```
rgyb
bbrg
rygy
grby


-------------time:  0  -------------
filtering result:
(0, 0) :  [0.0625]
(0, 1) :  [0.0625]
(0, 2) :  [0.0625]
(0, 3) :  [0.0625]
(1, 0) :  [0.0625]
(1, 1) :  [0.0625]
(1, 2) :  [0.0625]
(1, 3) :  [0.0625]
(2, 0) :  [0.0625]
(2, 1) :  [0.0625]
(2, 2) :  [0.0625]
(2, 3) :  [0.0625]
(3, 0) :  [0.0625]
(3, 1) :  [0.0625]
(3, 2) :  [0.0625]
(3, 3) :  [0.0625]

smoothing result:
(0, 0) :  [0.01064383]
(0, 1) :  [0.08652087]
(0, 2) :  [0.02432921]
(0, 3) :  [0.03269264]
(1, 0) :  [0.08087651]
```

```
(1, 1) :   [0.01437293]
(1, 2) :   [0.08704845]
(1, 3) :   [0.13297753]
(2, 0) :   [0.01828774]
(2, 1) :   [0.082639]
(2, 2) :   [0.12756662]
(2, 3) :   [0.11851817]
(3, 0) :   [0.02723841]
(3, 1) :   [0.02627611]
(3, 2) :   [0.01221386]
(3, 3) :   [0.11779813]
```

**Extension 2:** I implemented `viterbi` method, which uses viterbi algoritm to find the most likely path. To briefly explain the algorithm, for each time step k, the method finds the optimal path for all floor from t = 0 to t = k. It repeats the process until the end of the sensor sequence. Then, the algorithm chooses the floor with highest probability and backtracks the path. Then, it returns the optimal path.

```
----testing viterbi------
rgyb
bbrg
rygy
grby

color sequence:  ['y', 'r', 'g', 'b', 'r']
result of viterbi:  [(3, 3), (2, 3), (2, 2), (2, 1), (2, 0), (1, 0)]
```