# Aula 3 — Modelos Ocultos de Markov

Inteligência Artificial

Universidade Federal do ABC

22 de novembro de 2017

# Introdução

- Hoje vamos implementar um modelo visto nas aulas de Redes Bayesianas
  - Modelo Oculto de Markov — Hidden Markov Model (HMM)
- Hoje as regras serão diferentes
  - Pacman não fugirá dos fantasmas, mas tentará pegá-los!

# Problema

- Não sabemos exatamente onde está o fantasma
  - Mas temos um sensor que nos dá uma informação ruidosa sobre a posição!
  - No nosso caso, a informação é a distância de manhattan (ruidosa)
- $P(x)$ = distribuição inicial sobre os estados (uniforme)
- $P(x_t|x_{t-1})$ = distribuição de transição entre estados, $x_t$ corresponde ao estado no instante $t$
- $P(e_t|x_i)$ = distribuição de emissão em que $e_t$ é uma evidência no instante $t$
  - No nosso caso temos P(distanciaRuidosa|distanciaReal)

# Arquivos

- Baixe o arquivo aula4_tracking.zip e descompacte
- Você modificará os seguintes arquivos nos exercícios:
  - bustersAgents.py
  - inference.py
- Para verificar a corretude das suas soluções você executará o arquivo:
  - autograder.py
- Os seguintes arquivos contém informações relevantes para os exercícios:
  - busters.py: Substituto do pacman.py
  - ghostAgents.py, bustersGhostAgents: Classes com os novos agentes para os fantasmas
  - distanceCalculator.py: Computa distância do labirinto

# Explorando

- `python2.7 busters.py -t 0.5`
- Blocos de cor indicam possíveis localizações dos fantasmas
  - Considerando as leituras ruidosas dos sensores
  - Os valores no canto direito correspondem as distâncias e estão sempre a no máximo 7 unidades do valor real
  - A probabilidade do sensor ter uma leitura decresce exponencialmente com a diferença para a distância real
- Vamos implementar o HMM para esse problema

# Exercício 1: *Inferência baseada em observação*

- Nossa evidência é uma leitura de *distância*
- Precisamos atualizar nossas crenças de acordo com a leitura
- Finalizar implementação do método `observe` da classe `ExactInference` em `inference.py`
- `emissionModel` armazena a probabilidade da evidência condicionada a uma distância real
  - P(distanciaRuidosa | distanciaReal)
  - É um dicionário, sendo a chave a distância real
- Os fantasmas só podem estar em posições na lista `self.legalPositions`
  - `util.manhattanDistance` computa a distância entre duas posições

# Exercício 1: *Inferência baseada em observação*

- Caso particular quando um fantasma é "capturado"
  - `noisyDistance` será `None` (único caso em que isso ocorre!)
  - Ele deve ir para a cela que fica em `self.getJailPosition()`
  - Atualizar crenças de acordo!
- `self.beliefs` armazena a crença em uma posição
  - Variável que deverá ser atualizada
- Lembrando que:
  - $P(x_1|e_1) \propto P(x_1)P(e_1|x_1)$
  - `Counter.normalize()` realiza a normalização
- Para avaliar:
  - `python2.7 autograder.py -q q1`

# Exercício 2: *Passagem do tempo*

- Se em duas leituras consecutivas temos os valores 1 e 8 de distância, existe algo errado com o sensor
  - Podemos incorporar o nosso conhecimento sobre a transição de estados dos fantasmas
- Implementar método `elapseTime` em `ExactInference` (`inference.py`)

# Exercício 2: *Passagem do tempo*

- `self.setGhostPosition(gameState, ghostPosition)`
  - Modifica a posição do fantasma em um `gameState`
- `self.getPositionDistribution(gameState)`
  - Retorna a probabilidade do fantasma estar em cada uma das posições dado o gameState contendo a posição em que ele estava
- Ver nos comentários do arquivo como juntar ambos
- Lembrando que:
  - $P(x_{t+1}|e_{1:t}) = \sum_{x_t} P(x_{t+1}|x_t)P(x_t|e_{1:t})$
  - `self.beliefs` armazena a crença atual em uma posição (qual o termo correspondente?)
- Para avaliar:
  - `python2.7 autograder.py -q q2`
  - Alguns dos casos de teste usam um fantasma que prefere um dos extremos, qual?

# Exercício 3: *Juntando tudo*

- Temos todos os componentes prontos
  - As chamadas as suas funções já estão implementadas
  - Basta implementar a estratégia do Pacman
- Implementar uma estratégia gulosa
  - Mover o Pacman na direção em que temos maior crença de que o fantasma está
- Finalizar implementação do método `chooseAction` da classe `GreedyBustersAgent` em `bustersAgents.py`

# Exercício 3: *Juntando tudo*

- Encontrar a posição mais provável de cada fantasma que ainda não foi capturado
- Escolher a ação que leva o Pacman a ficar mais próximo desse fantasma
  - `self.distancer.getDistance(pos1, pos2)` computa mazeDistance
  - a posição resultante de uma ação é obtida via `Actions.getSuccessor(position, action)`
- A variável `livingGhostPositionDistributions` contém as probabilidades de cada posição para cada um dos fantasmas
  - lista de `Counter`
  - ver implementação no código, índice de `getLivingGhosts` é 1 a mais que índice de `ghostBeliefs` (Pacman)
- `python2.7 autograder.py -q q3`

# Exercício 4: *Filtro de Partículas — Observação*

- Vamos implementar agora a inferência utilizando filtro de partículas!
  - útil quando o tamanho do espaço é muito grande
- Uma partícula (amostra) é uma posição do fantasma
- Implementar funções `initializeUniformly, getBeliefDistribution, observe` da classe `ParticleFilter` em `inference.py`
- Quando terminar:
  - `python2.7 autograder.py -q q4`

# Exercício 4: *Filtro de Partículas — Observação*

- `initializeUniformly`
  - Distribuir partículas de forma uniforme em uma *lista*
  - Número de partículas = `self.numParticles`
  - `util.nSample` gera $n$ amostras de uma distribuição
  - Posições válidas podem ser encontradas em `self.legalPositions`
- `getBeliefDistribution`
  - Converter lista de partículas em crenças
  - Armazenar crenças em um `Counter` de posições

# Exercício 4: *Filtro de Partículas — Observação*

- `observe`
  - $B(x) \propto P(e|x)B'(x)$
  - Mesmo `emissionModel` para devolver $P(e|x)$
  - Nossa crença anterior equivale à soma dos pesos das partículas no estado
  - Após a observação, reamostrar partículas
    - `util.sample` gera uma amostra de uma distribuição
- Cuidado com dois casos especiais:
  - todas as partículas com peso zero (reamostrar todas novamente)
    - `initializeUniformly`
  - se um fantasma é pego, todas as partículas devem colocá-lo na sua prisão
    - `self.getJailPosition`

# Exercício 5: *Filtro de Partículas — Passagem do tempo*

- Implementar funções `elapseTime` da classe `ParticleFilter` em `inference.py`
  - `python2.7 autograder.py -q q4`
- Para cada partícula é necessário:
  - recuperar a distribuição de transição de estados para o estado atual
    - `self.getPositionDistribution` e `self.setGhostPosition`
  - amostrar dessa distribuição
    - `util.sample`