MAC0425/5739 - Inteligência Artificial

Exercício-Programa 3 - MDPs & RL

Prazo limite de entrega: 23:59 17/06/2017

1 Introdução

Neste EP estudaremos o **problema de tomada de decisões sequenciais** em ambientes incertos. Utilizaremos parte do material/código livremente disponível do curso UC Berkeley CS188 [1].

Os objetivos deste exercício-programa são:

(i) Familiarizar-se com um Processo de Decisão Markoviano (MDP) e comprender o compromisso entre custos/recompensas e a probabilidades de sucesso/falha de um problema de decisão.

(ii) Implementar o algoritmo de Iteração de Valor.

(iii) Implementar o algoritmo *Q-Learning* de Aprendizado por Reforço.

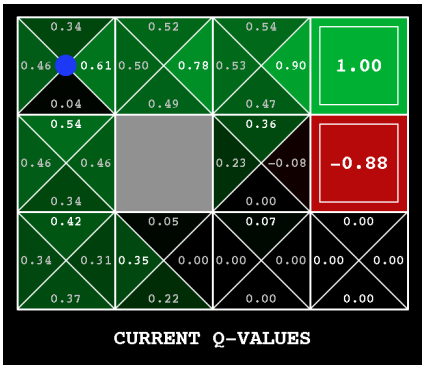

Figura 1: **GridWorld**

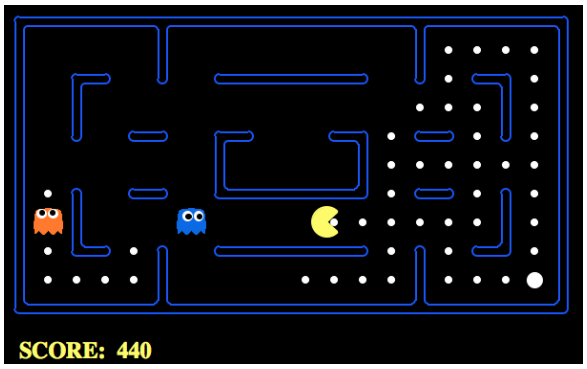

Figura 2: **Pacman**

Utilizaremos como ferramentas didáticas, o problema de navegação no mundo de salas da Figura 3 (*GridWorld*) e o jogo tradicional de *PacMan* da Figura 2.

1.1 Instalação

Para a realização deste EP será necessário ter instalado em sua máquina a versão 2.7 do Python [2]. Faça o download do **arquivo ep3.zip** disponível na página web da disciplina. Descompacte o arquivo zip e rode na raiz do diretório o seguintes comandos para testar a instalação:

```
$ python gridworld.py -m
$ python pacman.py
```

[1] `http://ai.berkeley.edu/reinforcement.html`
[2] `https://www.python.org/downloads/`

2 Processo de Decisão Markoviano (MDP)

Nesse exercício-programa você utilizará a implementação de MDPs definida pela classe base `MarkovDecisionProcess` e seu conjunto de métodos abaixo (disponível no arquivo **mdp.py**):

```
# arquivo mdp.py

class MarkovDecisionProcess:

    def getStates(self):
        """
        Return a list of all states in the MDP.
        """
        abstract

    def getStartState(self):
        """
        Return the start state of the MDP.
        """
        abstract

    def getPossibleActions(self, state):
        """
        Return list of possible actions from 'state'.
        """
        abstract

    def getTransitionStatesAndProbs(self, state, action):
        """
        Returns list of (nextState, prob) pairs representing the states
        reachable from 'state' by taking 'action' along with their transition
        probabilities.
        """
        abstract

    def getReward(self, state, action, nextState):
        """
        Get the reward for the state, action, nextState transition.
        """
        abstract

    def isTerminal(self, state):
        """
        Returns true if the current state is a terminal state.  By convention,
        a terminal state has zero future rewards.  Sometimes the terminal
        state(s) may have no possible actions.  It is also common to think of
        the terminal state as having a self-loop action 'pass' with zero
        reward; the formulations are equivalent.
        """
        abstract
```

3 Implementação

Arquivos que você precisará editar:

- **valueIterationAgents.py** onde o algoritmo de Iteração de Valor será implementado;
- **qlearningAgents.py** onde o algoritmo *Q-Learning* será implementado;
- **analysis.py** onde algumas questões do EP deverão ser repondidas.

Arquivo que você precisará ler e entender:

- **mdp.py** classe que implementa interface para acesso aos componentes do MDP;
- **featureExtractors.py** classes para extração de *features* dos pares *(estado, ação)* a ser utilizado no algoritmo aproximado *Q-Learning*;
- **util.py** estruturas de dados para auxiliar a codificação dos algoritmos.

Observação: Utilize a estrutura de dado `Counter` disponível no arquivo **util.py** para implementação de distribuição de probabilidades. Essa implementação é necessária para manter a compatibilidade com o aquivo de testes (**autograder.py**). Listas, tuplas, tuplas nomeadas, conjuntos e dicionários do Python podem ser utilizados sem problemas caso necessário.

4 Parte prática

Você deverá implementar algumas funções nos arquivos **valueIterationAgents.py** e **qlearningAgents.py**, além de modificar algumas funções do arquivo **analysis.py**. Não esqueça de remover o código `util.raiseNotDefined()` ao final de cada função.

IMPORTANTE: Antes de implementar cada código, leia atentamente os comentários da função e tenha certeza que entendeu sua(s) entrada(s) e saída(s)!

4.1 Código 1 - Iteração de Valor

Complete a implementação da classe `ValueIterationAgent` no arquivo **valueIterationAgents.py** para implementar o algoritmo (síncrono) de Iteração de Valor. Para rodar os casos de teste, execute o comando:

```
$ python autograder.py -q q1
```

Observações:

- Implemente a versão **síncrona** de Iteração de Valor, isto é, compute todos os valores da função valor $V^{(i+1)}$ em uma variável a parte para então atualizar o atributo `self.values`.
- Para a escolha gulosa das ações, você pode escolher qualquer ação ótima em caso de empate no valor da função Valor.
- Note que o **critério de parada** do algoritmo é o número de iterações dado na instanciação da classe `ValueIterationAgent`.

Rode o seguinte comando para avaliar a saída do algoritmo:

```
$ python gridworld.py -a value -i 5
```

Você deve obter os seguintes resultados:

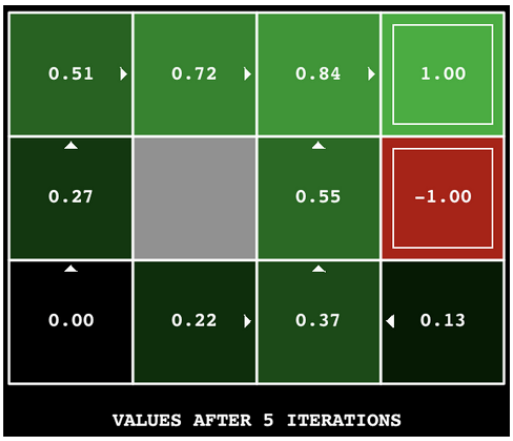

Figura 3: Saída gráfica do problema *GridWorld* após 5 iterações de valor

4.2 Código 2 - Diferentes políticas: influenciando o agente

Considere o *grid* mostrado na Figura 4. Esse *grid* tem 2 estados terminais com recompensa positiva (na linha do meio), uma saída mais próxima de valor +1 e outra saída mais afastada com valor +10. A linha de baixo consiste em estados terminais com recompensa negativa (mostrados em vermelho). O estado inicial do agente é a célula amarela. Nesse exercício distinguiremos **2 tipos de caminho no labirinto**: (1) "**caminhos que arriscam**" passar próximo a linha de baixo; esses caminhos são mais curtos porém apresentam alta probabilidade de um recompensa negativa considerável; (2) "**caminhos que evitam**" a linha de baixo passando pela parte superior do labirinto; esses caminhos são mais longos porém com probabilidades menores de receberem recompensas negativas consideráveis. Esses caminhos estão respectivamente representados na Figura 4 com as setas vermelha e verde.

O objetivo dessa questão será escolher valores para as constantes de fator de desconto, parâmetro de ruído (i.e., probabilidade de falha da ação) e recompensa dos estados não-terminais de forma a influenciar o comportamento do agente, ou seja, de forma a produzir políticas ótimas de diferentes tipos. Sua escolha das constantes e parâmetros do MDP devem ter a propriedade que, se seu agente seguisse a política ótima sem que estivesse sujeito à probabilidade de falha da ação, o caminho realizado seria exatamente o desejado.

Você deverá alterar os valores das variáveis locais nas funções `question3a` até `question3e` no arquivo **analysis.py** para implementar os seguintes tipos de política:

a. Preferir a saída (+1) arriscando passar pela linha de baixo (-10);

b. Preferir a saída (+1) evitando passar pela linha de baixo (-10);

c. Preferir a saída (+10) arriscando passar pela linha de baixo (-10);

d. Preferir a saída (+10) evitando passar pela linha de baixo (-10);

e. Evitar todas as saídas (+1, +10 e -10) de forma que o agente não atinja nenhum estado terminal.

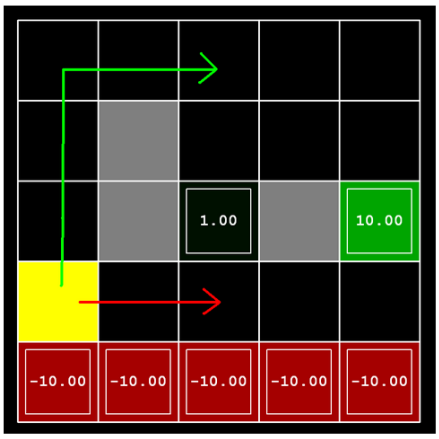

Figura 4: Diferentes políticas: como influenciar o comportamento do agente

Observação: Se você achar que não é possível tal comportamento, a função deverá retornar 'NOT POSSIBLE'.

Para rodar os casos de teste, execute o comando:

```
$ python autograder.py -q q3
```

4.3 Código 3 - Q-Learning

Complete a implementação da classe `QLearningAgent` no arquivo **qlearningAgents.py** para implementar o algoritmo *Q-Learning* de Aprendizado por Reforço. Para rodar os casos de teste, execute os seguintes comandos:

```
$ python autograder.py -q q4
$ python autograder.py -q q5
$ python autograder.py -q q7
```

Observações:

- Para o método `computeActionFromQValues`, você deverá quebrar os empates de forma aleatória usando a função `random.choice()`.
- Garanta que nos métodos `computeValueFromQValues` e `computeActionFromQValues`, você acesse os valores da função Q-Values através do método `getQValue`. Isso será fundamental para garantir a compatibilidade com o próximo código.
- A escolha de ação no método `getAction` deve garantir que em uma fração do tempo proporcional ao valor do parâmetro de exploração **epsilon** o agente escolhe uma ação qualquer aleatoriamente; o restante do tempo deve seguir a política gulosa induzida pelos valores de Q-values.
- Você pode simular uma variável binária de probabilidade de sucesso p por meio de `util.flipCoin(p)`, que retorna `True` com probabilidade p e `False` com probabilidade $1 - p$.

Para visualizar a execução do algoritmo *Q-Learning* rode os comandos:

```
$ python gridworld.py -a q -k 100
$ python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid
```

Observação: Note que o PacMan deve começar a obter um recompensa média positiva entre 1000 e 1400 jogos de treinamento.

4.4 Código 4 - Q-Learning aproximado utilizando *features*

OBSERVAÇÃO: esse código é obrigatório apenas para alunos da pós-graduação.

Complete a implementação da classe `ApproximateQAgent` no arquivo **qlearningAgents.py** para desenvolver um agente baseado em aprendizado por reforço que aprende os pesos de *features* dos estados, isto é, generalizando o valor de um estado para um conjunto de estados que compartilham as mesmas características. Para aproximar a função Q usando as *features* do problema do PacMan considere a Equação 1:

$$Q(s,a) = \sum_{i=1}^{n} f_i(s,a) \cdot w_i \tag{1}$$

onde cada peso w_i está associado com uma *feature* $f_i(s,a)$. Você deverá atualizar os vetores de peso através das Equações 2 e 3:

$$\delta \leftarrow (r + \gamma \cdot max_{a' \in \mathcal{A}(s)} Q(s',a')) - Q(s,a) \tag{2}$$

$$w_i \leftarrow w_i + \alpha \cdot \delta \cdot f_i(s,a) \tag{3}$$

onde s é o estado corrente, s' é o próximo estado, $\mathcal{A}(s)$ é o conjunto de ações válidas para o estado s, r é a recompensa recebida na transição e γ é o fator de desconto.

Para rodar os casos de teste, execute o comando:

```
$ python autograder.py -q q8
```

Observações:

- No seu código, você deverá implementar o vetor de pesos w_i como um dicionário do Python que mapeia cada *feature* com seu peso. Utilize um objeto `util.Counter`.
- O algoritmo aproximado *Q-Learning* assume a existência de uma função de extração de *features* $f_i(s,a)$. No arquivo **featureExtractors.py** já se encontram disponíveis todas as *features* que você deverá usar para implementar o jogador de PacMan. Note que os vetores de *features* são implementados com objetos `util.Counter`.
- Use o método `getQValue` para acessar os valores da função Q.

Para visualizar seu jogador de PacMan jogando, rode os seguintes comandos:

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

5 Entrega

Você deve entregar um arquivo `ep3-SeuNúmeroUSP.zip` contendo **APENAS** os arquivos:

(1) **valueIterationAgents.py**, **qlearningAgents.py** e **analysis.py**

(2) Não haverá confecção de relatório para esse EP.

Não esqueça de identificar cada arquivo com seu nome e número USP! No código coloque um cabeçalho em forma de comentário.

6 Critério de avaliação

O critério de avaliação dependerá dos resultados dos testes automatizados do **autograder.py**. Dessa forma você terá como avaliar por si só a nota que receberá para a parte prática.

Parte prática (graduação: 20 pontos / pós-graduação: 26 pontos)

- Código 1: autograder (6 pontos)
- Código 2: autograder (5 pontos)
- Código 3: autograder (9 pontos)
- Código 4: autograder (6 pontos) (**APENAS** para pós-graduação)