Exercício-Programa 3: Vinte e um

Entrega: 14/06/2020

Motivação

O objetivo deste exercício-programa (EP) é praticar a modelagem e resolução de problemas de tomada de decisão sequenciais involvendo incerteza. Para isso, você deverá formular uma versão generalizada do jogo de cartas vinte e um (*blackjack*) como um processo de decisão de Markov (MDP), e implementar algoritmos que encontrem políticas ótimas.

Instruções gerais

Você deverá implementar sua solução no esqueleto de código fornecido no arquivo `ep3.py`, que você deve ter recebido. Você pode criar novas funções, mas não deve modificar o protótipo das funções existentes.

O jogo

Em nossa versão modificada de vinte e um, o baralho incial pode conter uma coleção arbitrária de cartas com diferentes valores, desde que contenha o mesmo número de cartas para cada valor. Chamamos o número de réplicas das cartas de multiplicidade. Por exemplo, um baralho padrão possui 52 cartas com valores de 1 a 13 e multiplicidade 4. Em nosso jogo, permitimos outros tipos de baralho, por exemplo, um baralho contendo apenas os valores 1, 5 e 20, com multiplicidade 10 (portanto contendo 30 cartas). As cartas são sempre embaralhadas antes do início, de forma que qualquer permutação das cartas seja igualmente provável.

Uma vez que o baralho tenha sido definido, o jogo decorre em uma sequência de rodadas. Em cada rodada, o jogador pode:

- pegar a próxima carta do topo do baralho, o que não possui custo nem ganho;
- espiar a próxima carta do topo do baralho, o que *custa* um valor `custo_espiada` e faz com que aquela carta seja mostrada na rodada seguinte; ou
- sair do jogo.

Não é permitido espiar por duas vezes seguidas; se isso acontecer, o método `succAndProbReward()` deve retornar `[]`.

O jogo continua até que uma das seguintes condições se torne verdadeira:

- O jogador pega uma carta e "vai à falência". Isso ocorre quando a soma dos valores das cartas em sua mão excede um limiar estabelecido antes do início do jogo; no jogo de **21**, este valor é, isso mesmo, 21! Mas aqui podemos definir um limiar por jogo. A recompensa imediata para esse caso é 0.

- O jogador executa a ação de sair ou pega a última carta sem ir à falência. A recompensa imediata é a soma dos valores das cartas na sua mão. Note que se o jogador pegar a última carta e for à falência, o jogo termina com recompensa 0.

Você deve representar o estado como uma tupla

```
(total, índiceDaCartaEspiada, baralho)
```

no qual `total` indica a soma dos valores das cartas na mão do jogador, `índiceDaCartaEspiada` é um inteiro entre `0` e `len(baralho)-1` indicando o índice da carta espiada (`None` se a última ação não foi espiar), e `baralho` é uma lista contendo a quantidade de cartas de cada valor. O término do jogo é apresentado atribuindo o valor `None` a `baralho`.

Por exemplo, suponha um baralho com cartas 1, 2 e 3 e multiplicidade 1, e um limiar de 4. O jogador inicia sem cartas, o que corresponde ao estado `(0, None, (1,1,1))`. O jogo progride então dependendo da ação do jogador (pegar, espiar ou sair):

- Se o jogador pegar uma carta, o jogo transita para um dos seguintes estados com probabilidade uniforme 1/3, e o jogador recebe uma recompensa de 0:

  - `(1, None, (0, 1, 1))`
  - `(2, None, (1, 0, 1))`
  - `(3, None, (1, 1, 0))`

  Lembre-se de que a recompensa pelas cartas na mão só é recebida ao término do jogo (e se o jogador não for à falência).

- Se o jogador espiar, os três possíveis estados sucessores são:

  - `(0, 0, (1, 1, 1))`
  - `(0, 1, (1, 1, 1))`
  - `(0, 2, (1, 1, 1))`

  Para qualquer das transições acima, o jogador recebe uma recompensa `-custo_espiada`. Se o jogador executar essa ação então a ação de pegar na rodada seguinte se torna determinística (e a ação de espiar se torna indisponível). Por exemplo, se o jogador estiver no estado `(0, 0, (1, 1, 1))` a ação de pegar leva ao estado `(1, None, (0, 1, 1))` com probabilidade 1. Note que o segundo elemento da tupla representa o índice e não o valor da carta espiada.

- Se o jogador sair, o estado resultante será `(0, None, None)`. Lembre-se que definir o baralho como `None` significa o fim do jogo.

Como outro exemplo, assuma que o estado atual do jogador seja `(3, None, (1, 1, 0))`, e o limiar é 4.

- A ação de sair leva ao estado sucessor `(3, None, None)`.

- A ação pegar leva a um dos seguintes estados (com probabilidade uniforme):
  - `(3 + 1, None, (0, 1, 0))`
  - `(3 + 2, None, None)`

  No segundo estado sucessor, o baralho é definido como `None` para representar que o jogo terminou pois o jogador "foi à falência".

Parte 1: Modelando o MDP

Sua primeira tarefa é formular o jogo como um MDP, implementando o seguinte método (no arquivo `ep3.py`):

```
class BlackjackMDP(util.MDP):
  def succAndProbReward(self, state, action):
```

O argumento `state` é a representação de um estado (tupla de 3 elementos) como descrita anteriormente, e o argumento `action` é uma string representando uma ação aplicável (`'Pegar'`, `'Espiar'` ou `'Sair'`).

Antes de começar a programar, procure pensar como deve ser sua formulação do problema. Por exemplo, reflita como é a função de transição.

Parte 2: Tomando decisões ótimas

Agora que você especificou o jogo, chegou a hora de vencer! Implemente um algoritmo de iteração de valor assíncrono no método `solve` da classe `ValueIteration` em `ep3.py`. Use sua implmentação para resolver (isto é, encontrar a política ótima) para os MDPs `MDP1` e `MDP2` definidos em `ep3.py`.

Suponha agora que você seja o administrador de um cassino e deseja projetar um jogo que faça os jogadores espiarem bastante. Assumindo um limiar fixo de 20, um valor de `custo_espiada` de 1, especifique um baralho para o qual o MDP correspondente possui uma política ótima que prescreve a ação de espiar para pelo menos 10% dos estados. Implemente a função `geraMDPxereta()` para retornar uma instância de `BlackjackMDP` com essa característica. Antes de atribuir valores aleatoriamente, procure refletir sobre as quais casos levariam o jogador a espiar.

Parte 3: Aprendendo a jogar

Suponha agora que você entra em um cassino, e ninguém lhe diz as recompensas nem as probabilidades das transições. Seu objetivo nessa parte é construir uma política utilizando aprendizado por reforço.

Sua tarefa nessa parte é implementar o algoritmo de Q-learning aproximado na classe `QLearningAlgorithm` do arquivo `ep3.py`. Para sua conveniência, parte dos métodos já estão implemenados. Não altere essas partes. O seu algoritmo será usado em uma simulação de um MDP, alternando entre o método `getAction`, que deve retornar uma ação a ser executada, e o método `incorporateFeedback`, que informa sobre o resultado daquela ação, para que a política seja melhorada. Inspecione a função `simulate` no arquivo `util.py` para entender melhor como a simulação funciona.

Seu algoritmo deve aproximar a função Q por uma combinação linear de atributos (features):

$$Q(s,a) = \sum_i w_i \cdot f_i(s,a)\,.$$

No código, os pesos w_i são representado por `self.weights`, os atributos f_i pelo método `featureExtractor` e $Q(s,a)$ por `self.getQ`.

Você deve implementar o método `incorporateFeedback` da classe `QLearningAlgorithm`, o qual deve atualizar os pesos (`self.weights`) de acordo com a atualização Q-learning aproximado. Note que o método `getAction` implementa uma política ϵ-gulosa (e portanto já alterna entre exploração e exploitação).

Execute seu algoritmo utilizando os atributos computados pela função `identityFeatureExtractor()`. Teste seu algoritmo nos MDPs `MDP1` e `MDP2`. Compare a política obtida com a política ótima encontrada com iteração de valor (veja para quantos estados as ações coincidem).

Generalizando o conhecimento

Use sua implementação de Q-Learning aproximado para aprender uma política no MDP `MDP1`, usando 30000 episódios na simulação. Agora ajuste a taxa de aprendizado e a taxa de exploração para zero (para que a política não se altere) e teste a mesma política no MDP `largeMDP`. Simule 30000 episódios e compare as ações encontradas com a política ótima. Você deve notar que a política retornada é muito inferior a ótima. Isso ocorre porque nossa função de características `identityFeatureExtractor()` simplesmente retorna o próprio estado, fazendo com que o algoritmo seja de fato um Q-Learning tabular, e portanto não generalize.

Sua última tarefa nesse EP é utilizar seu conhecimento sobre o domínio para aumentar o poder de generalização de Q-Learning, de forma que o aprendizado sobre um estado se propague para estados semelhantes. Você deve implemente o método `blackjackFeatureExtractor` no arquivo `ep3.py` que recebe um par estado-ação e retorna um conjunto de atributos na forma de uma lista de pares chave do artributo, valor do atributo. O seu algoritmo deve gerar políticas que se aproximam do desempenho da política ótima em `largeMDP`.

Alguns exemplos de características úteis são:

- Indicadores dos pares ação tomada e total atual.
- Indicador da presença/ausência de cada valor de carta no baralho. Exemplo: se o baralho é [3,4,0,2], as características são $(1, 1, 0, 1)$.
- Indicadores do número de cartas restantes no baralho para cada valor e ação.

Instruções para entrega

Dica: Um arquivo de testes `autograder.py` é fornecido. A cada execução, dê uma ohada no arquivo `final_result.txt` gerado; note que este é acrescido a cada execução, ou seja, os resultados se acumulam neste arquivo em vez de se sobrescreverem.

Você deve submeter via eDisciplinas apenas o arquivo `ep3.py` contendo a sua solução até às 23:59 do dia 14/06/2020. Para evitar que seu EP seja zerado, certifique-se que o arquivo foi submetido sem problemas (baixando e executando o arquivo do site, rodando os testes) e que ele consiste em um script executável escrito em Python 3.

Avaliação

Embora seja muito importante que seu código passe em todos os testes, isso não é garantia de que seu código receberá nota máxima. É possível que os testes não verifiquem alguns casos peculiares onde seu programa pode vir a falhar.