Planejamento baseado em Processos de Decisão Markovianos

Silvio do Lago Pereira e Leliane Nunes de Barros

Instituto de Matemática e Estatística
Universidade de São Paulo
{slago,leliane}@ime.usp.br

Resumo Há muito tempo, planejamento tem sido um assunto de grande interesse na área de Inteligência Artificial. Na sua concepção clássica, a suposição epstemológica é de que o agente tem conhecimento completo sobre o mundo, bem como sobre os efeitos de suas ações. Entretanto, embora essa suposição faça sentido em vários contextos, na prática, ela nem sempre é possível. *Planejamento baseado em Processos de Decisão Markovianos* visa, justamente, tratar os casos em que o agente deve planejar levando em conta que os efeitos de suas ações são incertos.

1 Introdução

Planejamento baseado em Processos de Decisão Markovianos [1] é uma importante extensão de planejamento clássico, que nos permite tratar problemas em que as ações do agente têm efeitos incertos. Conforme veremos, modelando esses problemas de planejamento como processos de decisão markovianos, importantes resultados da Pesquisa Operacional podem ser adaptados para resolvê-los.

Essencialmente, um problema de planejamento sob incerteza pode ser visto como um processo de decisão em que, a cada passo, o agente observa o estado corrente de seu ambiente e escolhe uma ação que maximize a probabilidade de que, no futuro, um estado satisfazendo a meta de planejamento seja atingido. Num processo de decisão markoviano, embora os efeitos das ações possam ser incertos, assumimos que o agente sempre tem percepção completa e precisa de seu ambiente.

2 Processos de Decisão Markovianos

Um processo *estocástico* é uma seqüência de variáveis aleatórias $\langle S_t \rangle_{t=0}^{\infty}$, representando estados, que induz uma função de transição probabilística da forma $Pr[S_{t+1} = s_{t+1} \mid S_0 = s_0, \ldots, S_t = s_t]$; ou seja, a probabilidade de transição para um estado futuro s_{t+1} depende do passado do processo $s_0, \ldots, s_t$.

Um processo estocástico é *markoviano* se seu estado corrente resume seu passado de forma compacta, sem descartar informações necessárias para prever seu

estado futuro. Assim, um processo markoviano induz uma função de transição probabilística da forma $Pr[S_{t+1} = s_{t+1} \mid S_t = s_t]$. Em outras palavras, num processo markoviano, a probabilidade de transição para um estado futuro depende apenas do estado corrente desse processo, sendo seu histórico irrelevante.

Um processo de decisão markoviano é um modelo formal para a interação síncrona entre um agente e seu ambiente: a cada instante, o agente observa o estado corrente de seu ambiente s_t e decide executar uma ação a_t; essa ação afeta o estado corrente, produzindo um estado futuro s_{t+1} e um ganho g_{t+1} (figura 1). O objetivo do agente é maximizar seu ganho com o passar do tempo.

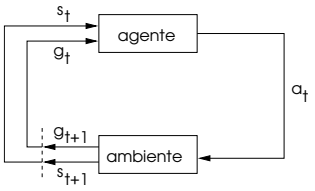

Figura 1. Interação síncrona entre agente e ambiente

2.1 O modelo formal

Um *processo de decisão markoviano* é definido por uma tupla $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, p, r, c \rangle$, onde:

- $\mathcal{S}$ é um conjunto finito de estados possíveis do ambiente;
- $\mathcal{A}$ é um conjunto finito de ações executáveis pelo agente;
- $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1]$ é uma função de transição de estados probabilística;
- $r : \mathcal{S} \mapsto \mathbb{R}_+$ é uma função que associa uma recompensa a cada estado;
- $c : \mathcal{A} \mapsto \mathbb{R}_+$ é uma função que associa um custo a cada ação.

A função de transição p define uma distribuição de probabilidades sobre $\mathcal{S}$; *i.e.*, $\sum_{s' \in \mathcal{S}} p(s, a, s') = 1$, onde $p(s, a, s')$ denota a probabilidade de transição para o estado s', dado que a ação a foi executada no estado s.

Como as transições são probabilísticas, o estado s', resultante da execução da ação a no estado s, não é *previsível*, mas apenas *observável* (figura 2). Assim, diferentemente do que ocorre no planejamento clássico, a *solução* para um problema de planejamento modelado por um PDM não é uma seqüência de ações, mas sim uma função $\pi : \mathcal{S} \mapsto \mathcal{A}$, que mapeia estados em ações. Essa função π, denominada *política*, descreve o comportamento do agente; especificando, para cada estado s do ambiente, que ação $\pi(s)$ deve ser executada por ele. Quando as ações especificadas pela política são sempre as mesmas, para cada estado, independentemente do instante de tempo, dizemos que a política é *estacionária*; do contrário, dizemos que é *não-estacionária*.

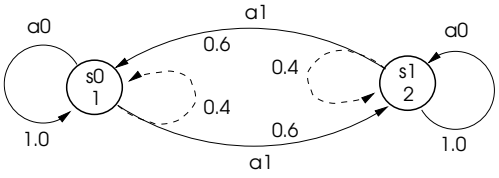

Figura 2. Diagrama de transições de um PDM, onde a ação a_i tem custo i

2.2 Avaliação de políticas

Para avaliarmos uma política, precisamos antes especificar por quanto tempo ela será seguida pelo agente. Há duas possibilidades:

Horizonte finito: o agente segue a política por um número finito de passos. Nesse caso, a maneira como o agente se comporta costuma mudar, à medida em que ele se aproxima de seus últimos passos. Assim, quando o tempo de vida do agente é finito, geralmente, a política é não-estacionária.

Sejam π uma política não-estacionária (que associa o estado s à ação $\pi_n(s)$, quando ainda restam n passos ao agente) e $v_n^\pi(s)$ a soma esperada dos ganhos obtidos, a partir do estado s, seguindo-se a política π por n passos. Definimos indutivamente o valor de π como:

$$v_n^\pi(s) = \begin{cases} r(s) & \text{se } n = 0 \\ r(s) - c(\pi_n(s)) + \sum_{s' \in \mathcal{S}} p(s, \pi_n(s), s') v_{n-1}^\pi(s') & \text{se } n > 0 \end{cases}$$

Horizonte infinito: o agente segue a política por um número infinito de passos. Agora, como o agente tem sempre uma quantidade infinita de passos restantes, não há porque mudar o seu seu modo de agir com o passar do tempo. Então, no caso de horizonte infinito, é mais razoável considerar π uma política estacionária. Além disso, para garantir que seu valor seja finito (apesar de ser dado por uma soma de infinitos termos), usaremos uma *taxa de desconto* $0 < \gamma < 1$.

$$v_t^\pi(s) = r(s) - c(\pi(s)) + \gamma \sum_{s' \in \mathcal{S}} p(s, \pi(s), s') v_{t+1}^\pi(s')$$

2.3 Obtenção de uma política ótima

Uma política π^* é *ótima* se, para toda política π e todo estado s, temos $v^{\pi^*}(s) \geq v^\pi(s)$; ou seja, uma política é ótima se maximiza a função v.

Seja v^* a função valor de uma política ótima. Um *política gulosa* com relação a v^*, denotada por π^*, é uma política ótima. Essa política é definida por:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} \{r(s) - c(a) + \gamma \sum_{s' \in \mathcal{S}} p(s, a, s') v(s')\}$$

Há dois métodos bastante utilizados para obtenção de políticas ótimas:

Iteração de valor: esse método calcula o valor v^* de uma política ótima e, simultaneamente, constrói uma política ótima π^*. Em cada iteração n do processo, consideramos que restam apenas n passos até o final da vida do agente e escolhemos uma ação que maximiza seu ganho esperado nesse ponto. À medida em que essas ações vão sendo escolhidas, uma política ótima vai sendo construída. No caso de horizonte finito ($\gamma = 1$), as ações escolhidas em cada iteração n formam uma política ótima não-estacionária π^*_n. Já no caso de horizonte infinito ($0 < \gamma < 1$), após um número finito de iterações k, o processo converge para a função v^*. Então, as ações escolhidas na k-ésima etapa desse processo constituem uma política ótima estacionária.

Em cada iteração n, o ganho esperado pela execução de uma ação a num estado s, denotado por $q_n(s, a)$, é definido como

$$q_n(s,a) = r(s) - c(a) + \gamma \sum_{s' \in \mathcal{S}} p(s,a,s') v_{n-1}(s'), \quad \text{para } n > 0,$$

a função valor $v(s)$ é definida como

$$v_n(s) = \begin{cases} r(s) & \text{se } n = 0 \\ \max\limits_{a \in \mathcal{A}} \{q_n(s,a)\} & \text{se } n > 0 \end{cases}$$

e, finalmente, a ação $\pi_n(s)$, que maximiza $v_n(s)$, é

$$\pi_n(s) = \arg\max_{a \in \mathcal{A}} \{q_n(s,a)\}, \quad \text{para } n > 0$$

O algoritmo ITERVAL, apresentado a seguir, recebe como entrada um PDM $\mathcal{M}$ e devolve como saída uma política estacionária ótima π^*.

ITERVAL($\mathcal{M}$)
1 **para** $\forall s \in \mathcal{S}$ **faça**
2 $\quad v_0(s) \leftarrow r(s)$
3 $n \leftarrow 0$
4 **repita**
5 $\quad n \leftarrow n + 1$
6 $\quad$ **para** $\forall s \in \mathcal{S}$ **faça**
7 $\quad\quad$ **para** $\forall a \in \mathcal{A}$ **faça**
8 $\quad\quad\quad q_n(s,a) \leftarrow r(s) - c(a) + \gamma \sum_{s' \in \mathcal{S}} p(s,a,s') v_{n-1}(s')$
9 $\quad\quad v_n(s) \leftarrow \max_{a \in \mathcal{A}} q_n(s,a)$
10 $\quad\quad \pi_n(s) \leftarrow \arg\max_{a \in \mathcal{A}} q_n(s,a)$
11 **até** $|v_n(s) - v_{n-1}(s)| < \epsilon, \forall s \in \mathcal{S}$
12 devolva π_n

Exemplo 1. A tabela 3 mostra os cálculos efetuados nas três primeiras iterações do algorimo ITERVAL, para encontrar uma política ótima para o PDM esquematizado na figura 2, supondo $\gamma = 1$. Os cálculos são apresentados com base nas seguintes equações especializadas para esse PDM:

$$
\begin{aligned}
q_n(s_0,a_0) &= r(s_0) - c(a_0) + p(s_0,a_0,s_0)v_{n-1}(s_0) + p(s_0,a_0,s_1)v_{n-1}(s_1)\\
&= 1 - 0 + 1 \times v_{n-1}(s_0) + 0 \times v_{n-1}(s_1)\\
&= 1 + v_{n-1}(s_0)\\
q_n(s_0,a_1) &= r(s_0) - c(a_1) + p(s_0,a_1,s_0)v_{n-1}(s_0) + p(s_0,a_1,s_1)v_{n-1}(s_1)\\
&= 1 - 1 + 0.4 \times v_{n-1}(s_0) + 0.6 \times v_{n-1}(s_1)\\
&= 0.4 \times v_{n-1}(s_0) + 0.6 \times v_{n-1}(s_1)\\
q_n(s_1,a_0) &= r(s_1) - c(a_0) + p(s_1,a_0,s_0)v_{n-1}(s_0) + p(s_1,a_0,s_1)v_{n-1}(s_1)\\
&= 2 - 0 + 0 \times v_{n-1}(s_0) + 1 \times v_{n-1}(s_1)\\
&= 2 + v_{n-1}(s_1)\\
q_n(s_1,a_1) &= r(s_1) - c(a_1) + p(s_1,a_1,s_0)v_{n-1}(s_0) + p(s_1,a_1,s_1)v_{n-1}(s_1)\\
&= 2 - 1 + 0.6 \times v_{n-1}(s_0) + 0.4 \times v_{n-1}(s_1)\\
&= 1 + 0.6 \times v_{n-1}(s_0) + 0.4 \times v_{n-1}(s_1)
\end{aligned}
$$

iteração	*cálculos efetuados*
0	$v_0(s_0) = r(s_0) = 1$ $v_0(s_1) = r(s_1) = 2$
1	$q_1(s_0,a_0) = 1 + v_0(s_0) = 1 + 1 = 2$ $q_1(s_0,a_1) = 0.4 \times v_0(s_0) + 0.6 \times v_0(s_1) = 0.4 \times 1 + 0.6 \times 2 = 1.6$ $q_1(s_1,a_0) = 2 + v_0(s_1) = 2 + 2 = 4$ $q_1(s_1,a_1) = 1 + 0.6 \times v_0(s_0) + 0.4 \times v_0(s_1) = 1 + 0.6 \times 1 + 0.4 \times 2 = 2.4$ $v_1(s_0) = \max\{q_1(s_0,a_0), q_1(s_0,a_1)\} = \{2, 1.6\} = 2$ $v_1(s_1) = \max\{q_1(s_1,a_0), q_1(s_1,a_1)\} = \{4, 2.4\} = 4$ $\pi_1(s_0) = \arg\max\{q_1(s_0,a_0), q_1(s_0,a_1)\} = a_0$ $\pi_1(s_1) = \arg\max\{q_1(s_1,a_0), q_1(s_1,a_1)\} = a_0$
2	$q_2(s_0,a_0) = 1 + v_1(s_0) = 1 + 2 = 3$ $q_2(s_0,a_1) = 0.4 \times v_1(s_0) + 0.6 \times v_1(s_1) = 0.4 \times 2 + 0.6 \times 4 = 3.2$ $q_2(s_1,a_0) = 2 + v_1(s_1) = 2 + 4 = 6$ $q_2(s_1,a_1) = 1 + 0.6 \times v_1(s_0) + 0.4 \times v_1(s_1) = 1 + 0.6 \times 2 + 0.4 \times 4 = 3.8$ $v_2(s_0) = \max\{q_2(s_0,a_0), q_2(s_0,a_1)\} = \{3, 3.2\} = 3.2$ $v_2(s_1) = \max\{q_2(s_1,a_0), q_2(s_1,a_1)\} = \{6, 3.8\} = 6$ $\pi_2(s_0) = \arg\max\{q_2(s_0,a_0), q_1(s_0,a_1)\} = a_1$ $\pi_2(s_1) = \arg\max\{q_2(s_1,a_0), q_2(s_1,a_1)\} = a_0$
3	$q_3(s_0,a_0) = 1 + v_2(s_0) = 1 + 3.2 = 4.2$ $q_3(s_0,a_1) = 0.4 \times v_2(s_0) + 0.6 \times v_2(s_1) = 0.4 \times 3.2 + 0.6 \times 6 = 4.88$ $q_3(s_1,a_0) = 2 + v_2(s_1) = 2 + 6 = 8$ $q_3(s_1,a_1) = 1 + 0.6 \times v_2(s_0) + 0.4 \times v_2(s_1) = 1 + 0.6 \times 3.2 + 0.4 \times 6 = 5.32$ $v_3(s_0) = \max\{q_3(s_0,a_0), q_3(s_0,a_1)\} = \{4.2, 4.88\} = 4.88$ $v_3(s_1) = \max\{q_3(s_1,a_0), q_3(s_1,a_1)\} = \{8, 5.32\} = 8$ $\pi_3(s_0) = \arg\max\{q_3(s_0,a_0), q_3(s_0,a_1)\} = a_1$ $\pi_3(s_1) = \arg\max\{q_3(s_1,a_0), q_3(s_1,a_1)\} = a_0$

Figura 3. Três primeiras iterações do algoritmo IterVal para o PDM da figura 2

□

Iteração de política: esse método "chuta" uma política aleatória e, a cada iteração, tenta alterar essa política de modo que seu valor seja aumentado. Após um número finito de iterações, o valor da política converge para o valor ótimo. A política obtida na última iteração é, portanto, uma política ótima. Note que, ao contrário do método de iteração de valor, que pode ser usado para obtenção de políticas ótimas não-estacionárias, o método de iteração de política só permite a obtenção de políticas ótimas estacionárias.

```
IterPol(M)
 1 para ∀s ∈ S faça
 2     π(s) ←ElementoAleatório(A)
 3 avalie π obtendo v(s), ∀s ∈ S
 4 repetir
 5     pontofixo ← verdade
 6     para ∀s : s ∈ S faça
 7         para ∀a : a ∈ A faça
 8           q(s,a) ← r(s) − c(a) + γ Σ_{s'∈S} p(s,a,s')v(s')
 9           se q(s,a) > v(s) então
10               v(s) ← q(s,a)
11               π(s) ← a
12               pontofixo ← falso
13 até pontofixo
14 devolva π
```

3 Um exemplo de planejamento baseado em PDM

Considere um cenário onde um robô vigilante deve permanecer num corredor de um prédio, sem atrapalhar a passagem das pessoas. Como do lado esquerdo desse corredor existem portas, o robô atrapalha menos quando fica do lado direito. Ademais, quando o robô se desloca, a posição desejada é alcançada com probabilidade 0.7 (com probabilidade 0.3, ele continua na mesma posição). Em cada instante, o robô deve decidir qual a melhor forma de agir: permanecer onde está ou se deslocar até uma outra posição (figura 4).

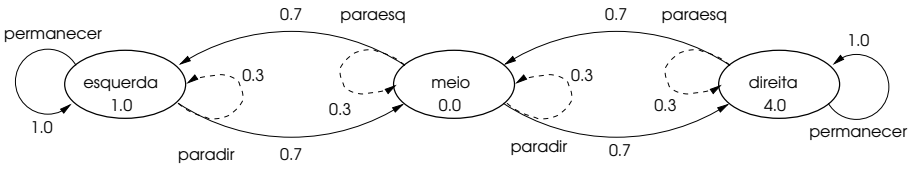

Figura 4. Diagrama de transição de estados para o robô vigilante

3.1 Modelagem usando PDM

Esse problema pode ser modelado por meio de um MDP $\langle \mathcal{S}, \mathcal{A}, p, r, c \rangle$, em que o conjunto $\mathcal{S} = \{esquerda, meio, direita\}$ especifica as possíveis posições do robô; o conjunto $\mathcal{A} = \{permanecer, paraesq, paradir\}$ especifica as possíveis ações do robô; p é a função de transição de estados probabilística:

$p(esquerda, permanecer, esquerda) = 1$
$p(esquerda, paradir, meio) = 0.7$
$p(esquerda, paradir, esquerda) = 0.3$
$p(meio, paraesq, esquerda) = 0.7$
$p(meio, paraesq, meio) = 0.3$
$p(meio, paradir, direita) = 0.7$
$p(meio, paradir, meio) = 0.3$
$p(direita, permanecer, direita) = 1$
$p(direita, paraesq, meio) = 0.7$
$p(direita, paraesq, direita) = 0.3$

r é a função que associa recompensas a estados (de modo que posições preferidas sejam melhor recompensadas):

$r(meio) = 0$
$r(esquerda) = 1$
$r(direita) = 4$

e, finalmente, c é a função que atribui custos às ações do robô (de modo que ações preferidas tenham custo menor):

$c(permanecer) = 0$
$p(paraesq) = 1$
$p(paradir) = 1$ □

Para esse problema, obtemos a seguinte política ótima estacionária ($\gamma = 0.9$):

$\pi(esquerda) = paradir$
$\pi(meio) = paradir$
$\pi(direita) = permanecer$

3.2 Programa para resolver PDM's

```
// Modificado em 31/08/2005 (por Silvio Lago)

import java.util.*;
import java.io.*;
```

```
public class Mdp {

    private String  filename;
    private int     states = 0;
    private int     actions= 0;
    private String  stateName[] = null;
    private String  actionName[] = null;
    private double  gamma = 0.5;
    private double  transition[][][];
    private double  reward[];
    private double  cost[];
    private double  value[];
    private int     policy[];
    private int     lineNum = 1;
    private double  epsilon = 1e-3;

    // programa principal

    static public void main(String args[]) {

        if( args.length < 2 ) {
            System.out.println("\nUse: Mdp <description> <method> <debug>\n");
            System.out.println("<method> := -v (value iteration) or");
            System.out.println("            -p (policy iteration)");
            System.out.println("<debug>  := -d (optional)\n");
            System.exit(1);
        }

        try {
            Mdp mdp = new Mdp(args[0]);
            if( args[1].equals("-v") ) mdp.valueIteration();
            else if( args[1].equals("-p") ) mdp.policyIteration();
            else { System.out.println("\nInvalid option\n"); System.exit(1); }
            mdp.savePolicy();
        } catch(IOException e) {
            System.out.println("\nFile not found\n");
        }
    }


    // cria a representacao do PDM descrito no arquivo de entrada

    public Mdp(String filename) throws IOException {

        this.filename = filename;

        BufferedReader inBuffer= new BufferedReader(new FileReader(new File(filename)));
        String line;


        while( ((line=inBuffer.readLine())!=null) && (states==0 || actions==0 || gamma==0) ) {
            lineNum++;
            if(line.length() == 0 || line.charAt(0) == '%') continue;
            StringTokenizer stToken = new StringTokenizer(line,"(){},= ");
            if( !stToken.hasMoreTokens() ) continue;
            String token = stToken.nextToken();
            if( token.equals("p") || token.equals("p") || token.equals("p") ) {
                System.out.println("Error (line "+lineNum+"): states and actions must be declared first");
                System.exit(1);
            }
            if( token.equals("states") ) {
                states  = stToken.countTokens();
                stateName = new String[states];
                for(int i=0; i<states; i++ )
                    stateName[i] = stToken.nextToken();
            }
            else if( token.equals("actions") ) {
                actions = stToken.countTokens();
```

```
            actionName = new String[actions];
            for(int i=0; i<actions; i++ )
                actionName[i] = stToken.nextToken();
        }
        else if( token.equals("gamma") ) {
            gamma = (new Double(stToken.nextToken())).doubleValue();
            if( gamma>=1 ) {
                System.out.println("Error: invalid gamma");
                System.exit(1);
            }
        }
    }

    transition = new double[states][actions][states];
    reward     = new double[states];
    cost       = new double[actions];
    value      = new double[states];
    policy     = new int[states];

    for(int i=0; i<states; i++)
        for(int j=0; j<actions; j++)
            for(int k=0; k<states; k++)
                transition[i][j][k] = 0;

    for(int i=0; i<states; i++)  reward[i] = 0;
    for(int i=0; i<actions; i++) cost[i] = 0;

    while( (line=inBuffer.readLine()) != null ) {
        lineNum++;
        if(line.length() == 0 || line.charAt(0) == '%') continue;
        StringTokenizer stToken = new StringTokenizer(line,"(){},= ");
        if( !stToken.hasMoreTokens() ) continue;
        String token = stToken.nextToken();
        if( token.equals("p") ) {
            int    s1 = stateNum(stToken.nextToken());
            int    a  = actionNum(stToken.nextToken());
            int    s2 = stateNum(stToken.nextToken());
            double p  = (new Double(stToken.nextToken())).doubleValue();
            transition[s1][a][s2] = p;
        }
        else if( token.equals("r") ) {
            int    s = stateNum(stToken.nextToken());
            double r = (new Double(stToken.nextToken())).doubleValue();
            reward[s] = r;
        }
        else if( token.equals("c") ) {
            int    a = actionNum(stToken.nextToken());
            double c = (new Double(stToken.nextToken())).doubleValue();
            cost[a] = c;
        }
    }
    inBuffer.close();
}


// converte nomes de estados em numeros correspondentes

private int stateNum(String stat) {
    for(int i=0; i<states; i++)
        if( stateName[i].equals(stat) )
            return i;
    System.out.println("\nError (line "+lineNum+"): undeclared state name ("+stat+")\n");
    System.exit(1);
    return -1;
}


// converte nomes de acoes em numeros correspondentes
```

```
private int actionNum(String act) {
    for(int i=0; i<actions; i++)
        if( actionName[i].equals(act) )
            return i;
    System.out.println("\nError (line "+lineNum+"): undeclared action name ("+act+")\n");
    System.exit(1);
    return -1;
}


// implementa o metodo de iteracao de valor

public void valueIteration() {
    double maxError  = -1;
    double sum;
    double w[] = new double[states];
    long   iter = 0;

    for(int i=0; i<states; i++)  value[i] = reward[i];

    do {
        maxError = -1;

        for(int s1=0; s1<states; s1++) {
            double maxValue = Double.NEGATIVE_INFINITY;
            int    maxAction= -1;

            for(int a=0; a<actions; a++) {
                sum = 0;
                for(int s2=0; s2<states; s2++) sum += transition[s1][a][s2]*value[s2];
                double Q = reward[s1]-cost[a] + gamma*sum;
                if( maxValue<Q ) {
                    maxValue = Q;
                    maxAction = a;
                }
            }

            double currentError = Math.abs(maxValue - value[s1]);
            if( currentError>maxError ) maxError = currentError;
            w[s1] = maxValue;
            policy[s1] = maxAction;
        }

        for(int i=0; i<states; i++) value[i] = w[i];
        iter++;
    } while( maxError> epsilon);
}


// implementa o metodo de iteracao de politica

public void policyIteration() {
    boolean fixedPoint = true;
    int iter=0;
    randomPolicy();
    policyEvaluation();

    do {
        for(int s=0; s<states; s++) {
            fixedPoint = true;
            for(int a=0; a<actions; a++) {
                double sum = 0;
                for(int t=0; t<states; t++)
                    sum += transition[s][a][t]*value[t];
                double Q = reward[s]-cost[a] + gamma*sum;
                if( Q>value[s]  ) {
                    value[s] = Q;
```

```
                    policy[s] = a;
                    fixedPoint = false;
                }
            }
        }
        iter++;
    } while( !fixedPoint );
}


// cria uma politica aleatoria

private void randomPolicy() {
    for(int s=0; s<states; s++) {
        policy[s] = (int) Math.round(Math.random()*(actions-1));
    }
}


// avalia uma politica especifica

private void policyEvaluation() {
    double maxError;
    double w[] = new double[states];
    int iter=0;

    for(int s=0; s<states; s++)  value[s] = reward[s];

    do {
        maxError = -1;

        for(int s=0; s<states; s++) {
            double sum = 0;
            for(int t=0; t<states; t++) sum += transition[s][policy[s]][t]*value[t];
            double Q = reward[s]-cost[policy[s]] + gamma*sum;
            if( value[s]<Q ) {
                w[s] = Q;
                double currentError = Math.abs(Q-value[s]);
                if( currentError>maxError ) maxError = currentError;
            }
        }

        for(int s=0; s<states; s++) value[s] = w[s];
        iter++;
    } while( maxError>epsilon );
}


// salva politica otima em arquivo

public void savePolicy() throws IOException {
    BufferedWriter outBuffer = new BufferedWriter(new FileWriter(new File(filename+".out")));
    outBuffer.write("\nPolicy (gamma="+gamma+")\n\n");

    for(int s=0; s<states; s++) {
        String state = (stateName==null) ? s+" " : stateName[s];
        String action = (actionName==null) ? policy[s]+" " : actionName[policy[s]];
        outBuffer.write("if "+state+" then "+action+"\n");
    }
    outBuffer.write("\n");
    outBuffer.close();
    System.out.println("\nOptimal policy written in "+filename+".out\n");
}
}
```

Referências

1. Boutilier, C., Dean, T. and Hanks, S. *Decision-Theoretic Planning: Structural Assumptions and Computational Leverage*, in Journal of Artificial Intelligence Research, v.11, p. 1-94, 1999.