Inteligência Artificial

Prof. Fabrício Olivetti de França Prof. Denis Fantinato

3º Quadrimestre de 2019

Probabilidade

Revisão de Probabilidade

Chamamos de **variável aleatória** uma variável que representa uma realização possível dentre um conjunto de eventos.

A **distribuição de probabilidade** representa a probabilidade de ocorrência para cada resultado possível (isto é, cada valor possível da variável aleatória).

Revisão de Probabilidade

Tráfego na rodovia Imigrantes:

- Variável aleatória: $T = \{leve, medio, pesado\}$
- Distribuição: $P = \{0.25, 0.50, 0.25\}$

Revisão de Probabilidade

As probabilidades são sempre **não negativas** e devem **somar UM** para todos os resultados possíveis.

Revisão de Probabilidade

A **esperança** ou o **valor esperado** de uma variável aleatória está associado ao seu valor central em uma distribuição. Pode ser vista como uma média ponderada de todos os possíveis resultados.

Matematicamente,

$E[X] = \int p_X(x) x dx$ (para variáveis contínuas)
$E[X] = \sum_i p_X(x_i) x_i$ (para variáveis discretas)

Revisão de Probabilidade

Para chegar na baixada santista pela imigrantes levamos 40 mins sem trânsito, 60 mins com trânsito moderado e 120 mins com trânsito pesado. O total de tempo esperado para uma viagem é:

$$0.25 \cdot 40 + 0.50 \cdot 60 + 0.25 \cdot 120 = 70 mins$$

Utilidades

Utilidades

Utilidade é uma função $U : S \mapsto \mathbb{R}$ que mapeia um estado final s para um valor real que indica a preferência do agente.

Utilidades

Em outras palavras, um valor real que indica o quanto é desejável seguir por um determinado caminho.

Utilidades

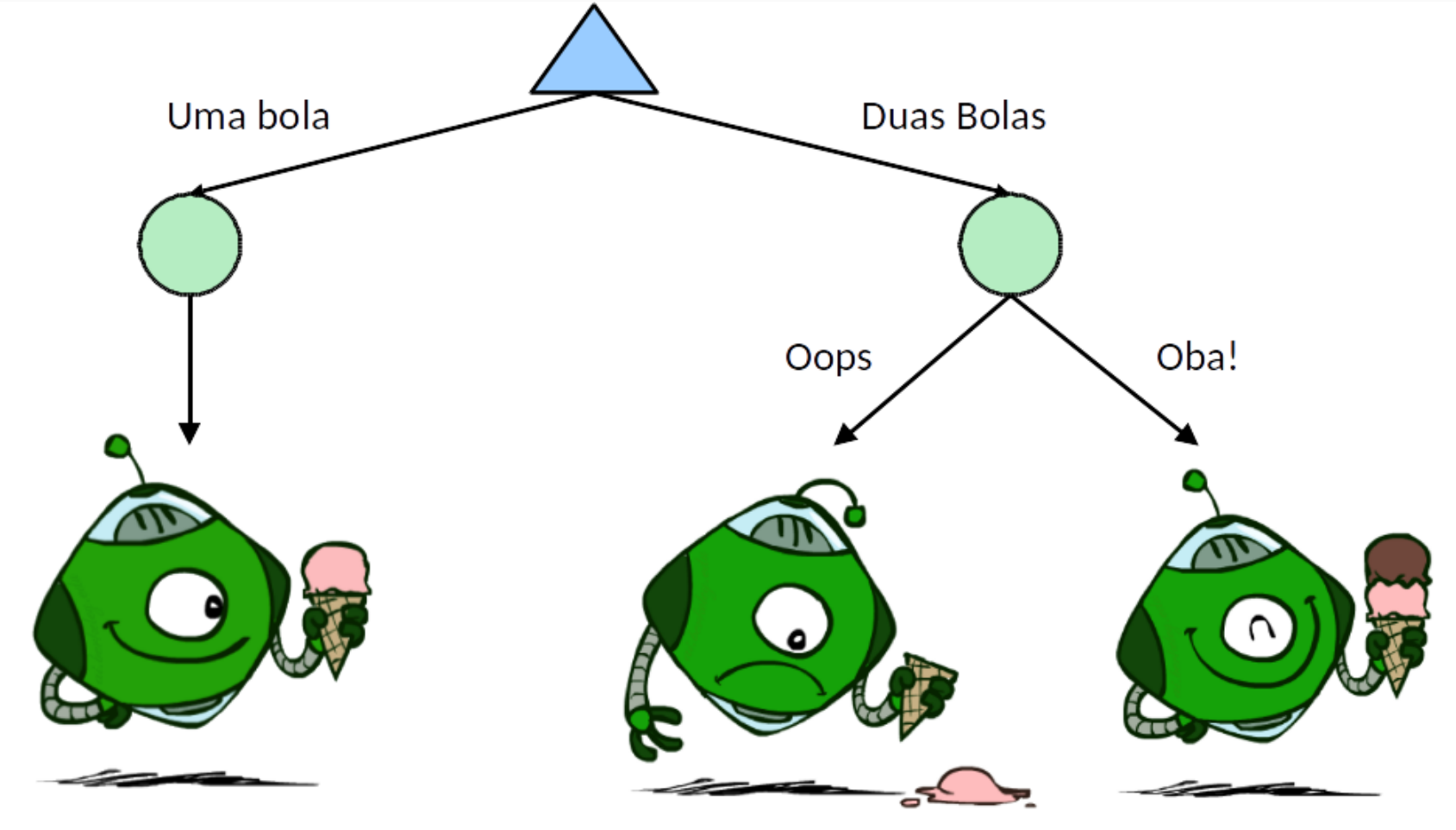
Até o momento modelamos utilidade como $U : S \mapsto \{-1, 0, 1\}$ representando derrota, empate ou vitória.

Utilidades

Em alguns jogos podemos pensar em preferências mais elaboradas.

Utilidades

A utilidade pode nos ajudar a decidir não apenas o caminho do melhor resultado mas aquele que tem a maior expectativa de sucesso.



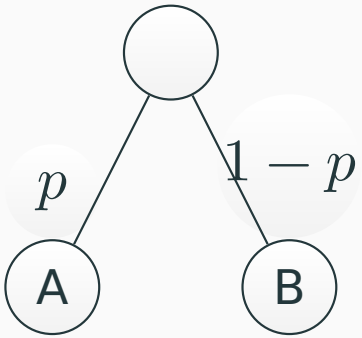
Utilidades

A função utilidade pode também incorporar a **probabilidade** de um evento ocorrer.

Utilidades

Considere um jogo simples em que você escolhe gastar ou não R$ 5,00 em uma raspadinha.

Se ganhar, você consegue um prêmio de R$ 20,00. Podemos descrever a árvore de jogos da seguinte forma para então tomarmos a decisão:



Utilidades

O comportamento do agente costuma ser modelado como a **Maximização da Utilidade Esperada** (Maximization of Expected Utility - MEU).

Utilidades

Dada uma função de utilidade $U(s)$ e probabilidades p_i correspondente a cada estado terminal i, podemos calcular a utilidade esperada de um nó como:

$$E[U](n) = \sum_i p_i \cdot U(s_i)$$

Para todo estado terminal i filho do nó n.

Busca com Incertezas

Jogos com Incertezas

Alguns jogos não são determinísticos: jogos com dados, jogos com sorteios de cartas, etc.

Jogos com Incertezas

Nesses casos não temos certeza do resultado de uma ação:

- As minhas ações possíveis são determinadas pelo lançamento de dados
- As ações de meus adversários também!
- Meu adversário pode se comportar de forma inesperada, até mesmo aleatória
- Falhas mecanicas (agente = robô)

Jogos com Incertezas

Devemos então alterar nosso algoritmo de Minimax para levar em conta essas incertezas.

Solução: substituir o valor do pior caso (Minimax adversarial) pelo caso médio (Expectiminimax).

Expectiminimax

```
def ExpectiminimaxUtil(s, p):
  if Terminal(s): return Utilidade(s, p)
  if p == MAX:    return maxVal(s)
  if p == MIN:    return minVal(s)
  if p == CHANCE: return expectVal(s)
```

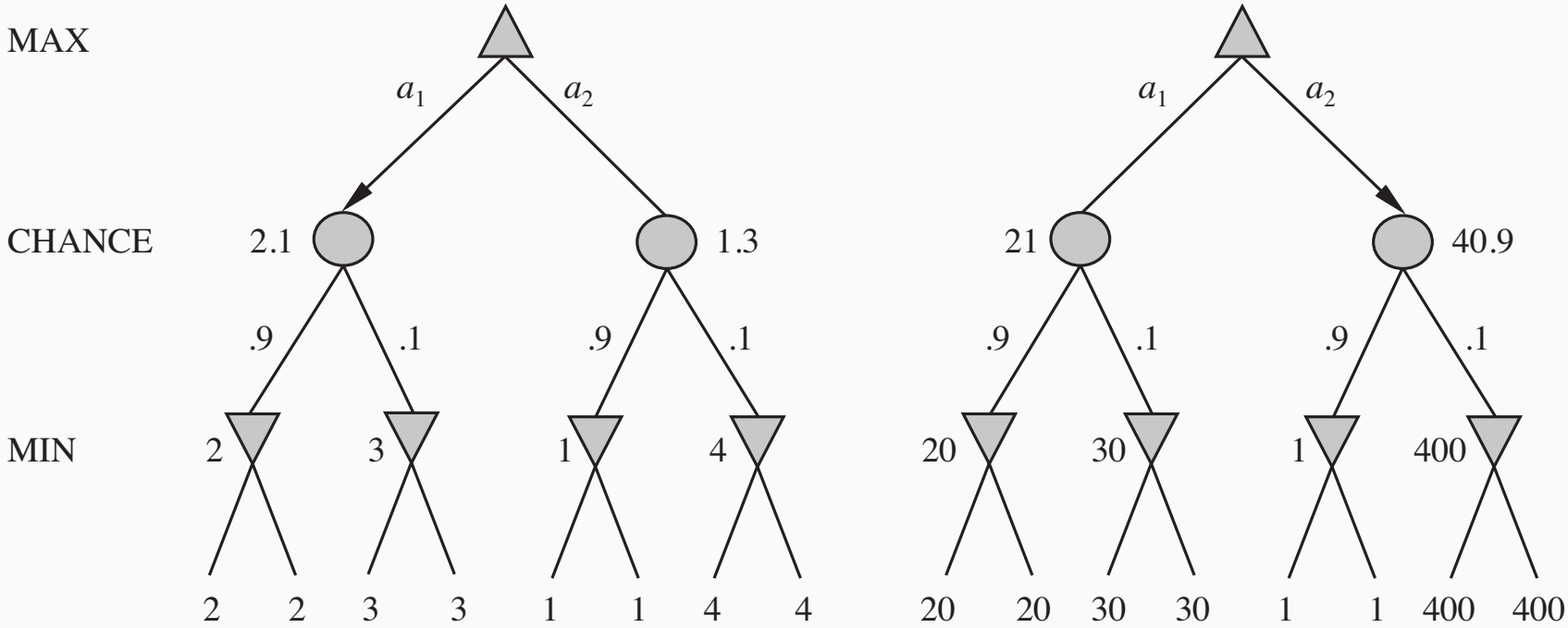
Expectiminimax

```
def expectVal(s):
  return mean([P(a)*ExpectiminimaxUtil(Result(s,a), MIN)
                    for a in Acoes(s)])
```

Expectiminimax

Acrescentamos uma nova ou camada de nós entre MAX e MIN chamado de **CHANCE** (sorte). Estes nós também podem ser chamados de nós probabilísticos.

Expectiminimax



Expectiminimax

Caso as probabilidades não sejam conhecidas, elas podem ser estimadas pelo conhecimento que temos do jogo ou por simulações aleatórias.

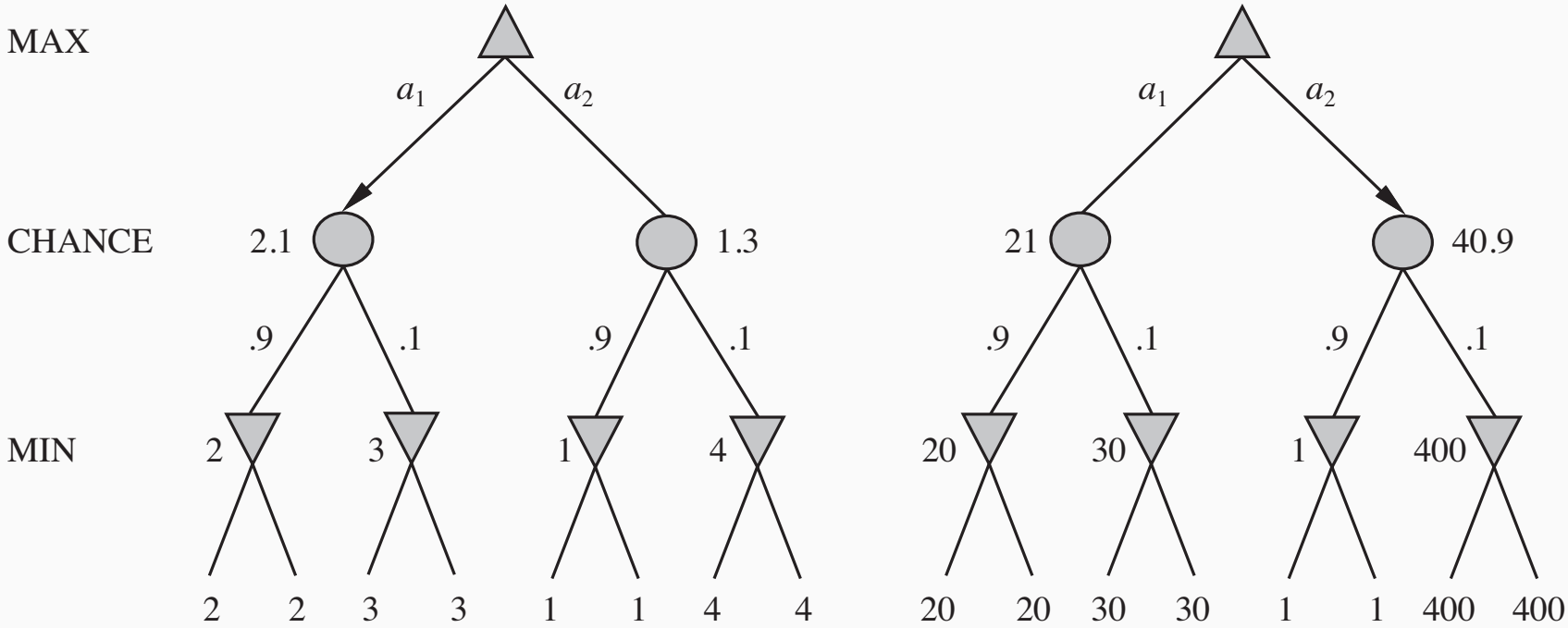
Expectiminimax

Note que a complexidade do problema aumenta muito em relação a jogos determinísticos.

É interessante criarmos heurísticas para avaliar nós internos e interromper a busca em certos nós.

Expectiminimax

Essas heurísticas devem ser escolhidas de tal forma a serem positivas e formarem uma transformação linear das probabilidades de vitória.



Jogos com múltiplos agentes

Quando o jogo não é de soma zero ou possui vários jogadores:

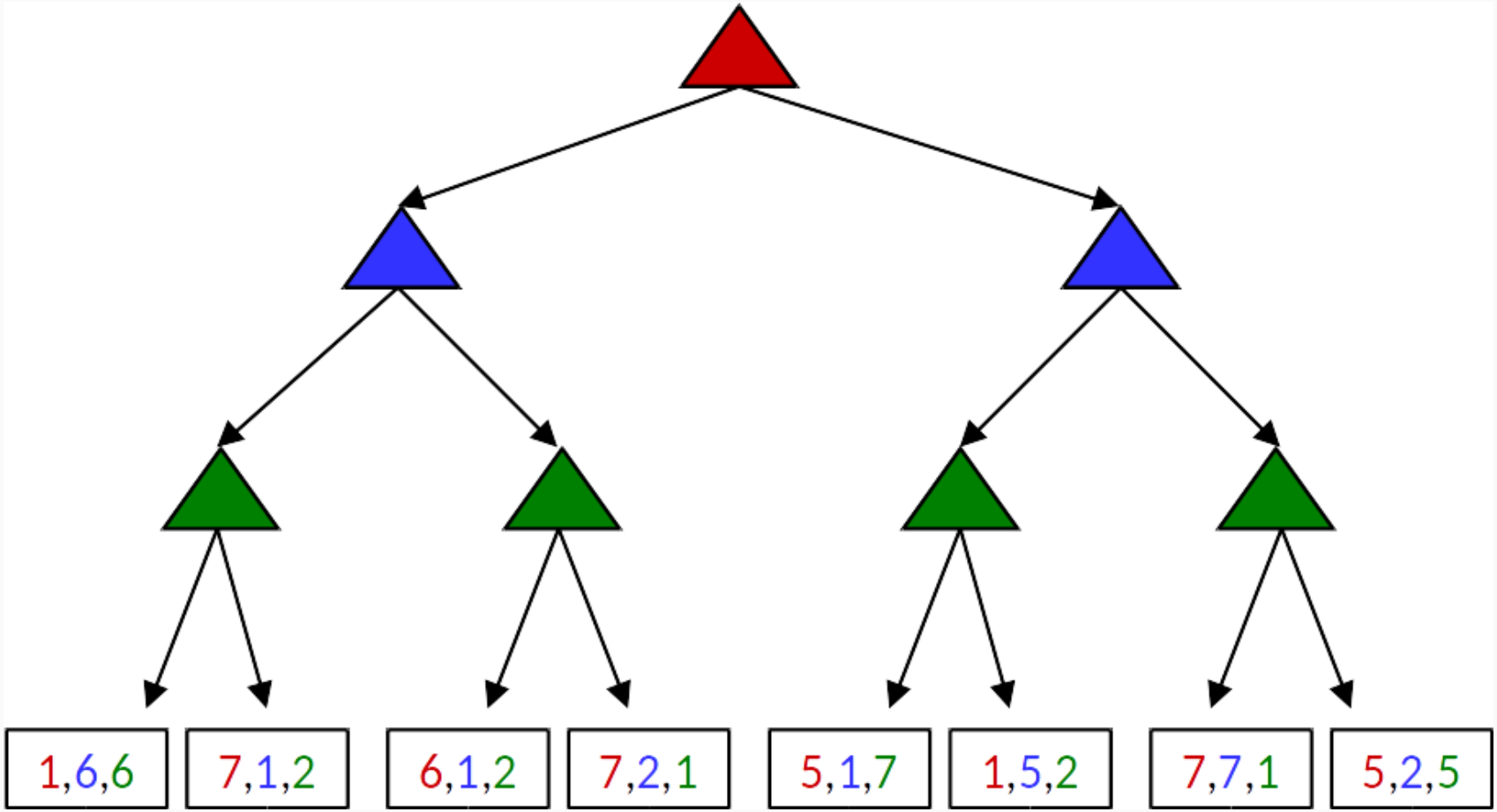
- Banco Imobiliário
- Poker

Jogos com múltiplos agentes

Podemos adaptar o Minimax para esses casos!

- Terminais contém tuplas de utilidades para cada jogador
- Todos os jogadores se tornam MAX

Jogos com múltiplos agentes

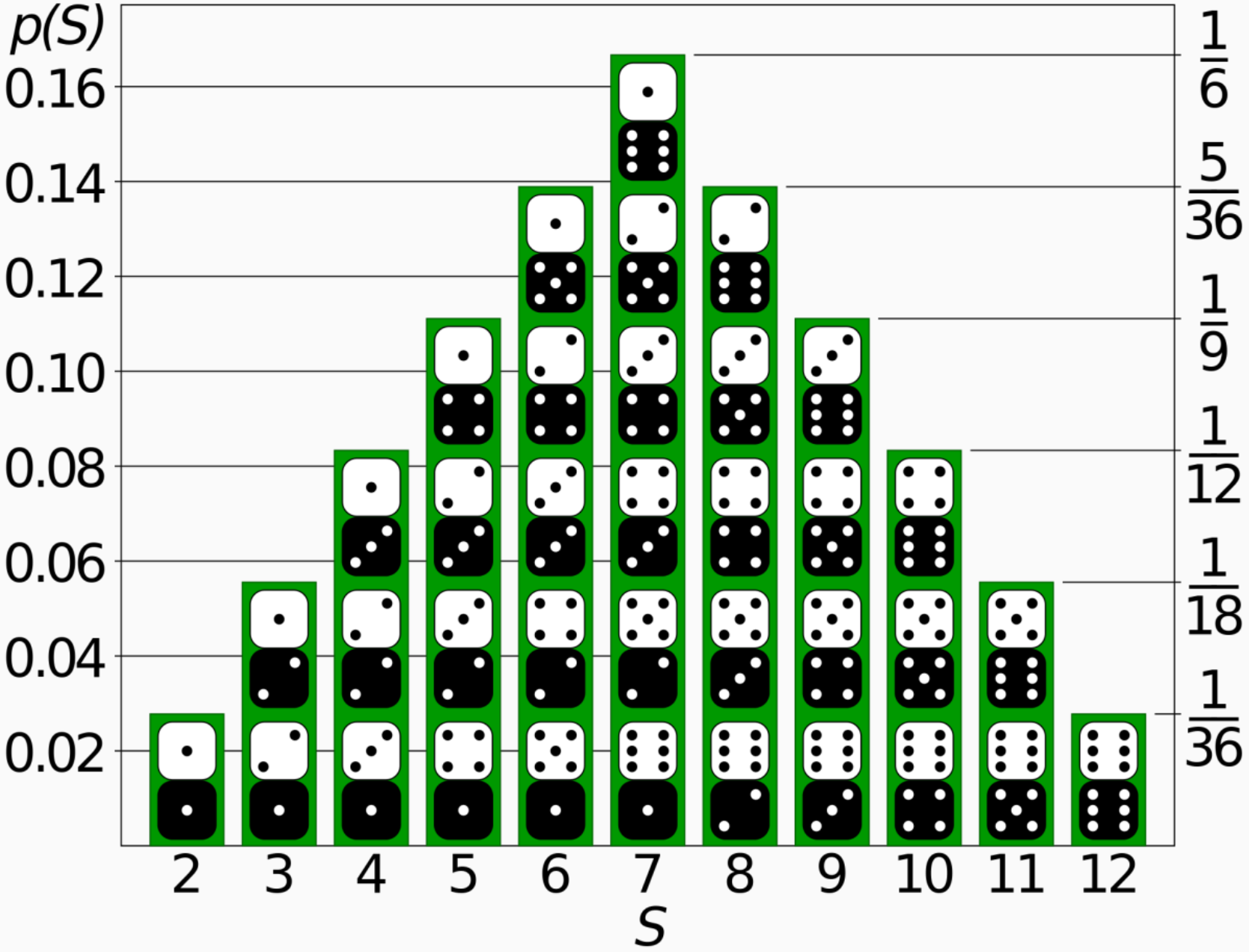


Exercício

Considere o jogo: anuncie um número entre 7 e 9. Então, role dois dados. Se o número for igual ou maior que o número anunciado, você ganha a quantia do número anunciado em reais. Se for menor, você recebe apenas 2 reais.

Monte os ramos da árvore Expectiminimax para esse problema. Qual é a melhor opção para maximizar o ganho?

Exercício


p(S)
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
2
3
4
5
6
7
8
9
10
11
12
S
1/6
5/36
1/9
1/12
1/18
1/36

Monte Carlo Tree Search

Monte Carlo Tree Search

Os algoritmos de busca em árvore de jogos vistos até então sofrem com problema de alta dimensionalidade.

Todos eles são impraticáveis para jogos com um fator de ramificação muito alto.

Monte Carlo Tree Search

Ideia: Usar simulações aleatórias para estimar a probabilidade de um nó ter mais ou menos chances de sucesso.

Como é um processo de simulações, pode ser repetido durante o tempo computacional disponível, quanto mais simulações melhor será o conhecimento adquirido.

Monte Carlo Tree Search

Imagine uma fileira de máquinas caça-níqueis em um Cassino.

Cada uma delas está ajustada com probabilidades de vitória distitntas e quantidade de prêmio também distintos entre si.

Monte Carlo Tree Search

Como maximizar seu ganho?

Você não pode observar outros jogando para ganhar informação...

A estratégia UCB1 constrói intervalo de confiança para cada máquina:

$\bar{x}_i \pm \sqrt{\frac{2 \ln n}{n_i}}$ ou $\left[\bar{x}_i - \sqrt{\frac{2 \ln n}{n_i}}, \bar{x}_i + \sqrt{\frac{2 \ln n}{n_i}}\right]$

$\bar{x}_i$ é a média dos ganhos da máquina i.

n_i é o número de vezes que testamos a máquina i.

n é a soma de todos os n_i.

Isso te retorna uma tupla de valores (baixo, alto), limitando sua confiança no ganho daquela máquina dentro dessa faixa.

Monte Carlo Tree Search

Sua estratégia é apostar sempre na máquina com maior valor de **alto**, imaginando que é a maior chance de ganhos.

Conforme você joga, a média daquela máquina é ajustada e a faixa de confiança se torna mais estreita.

Monte Carlo Tree Search

Enquanto isso, a faixa de confiança das demais máquinas tornam-se maiores.

Eventualmente outra máquina se torna mais interessante...

Monte Carlo Tree Search

O processo de Monte Carlo compreende um número grande de simulações aleatórias para ganhar informação de um sistema.

Podemos pensar em cada nó da árvore de jogos como um problema de caça-níqueis!

Monte Carlo Tree Search

Imagine uma árvore de jogos inicial e incompleta com cada nó contendo informação sobre número de vitórias do jogador e total de partidas que passaram por aquele nó.

Monte Carlo Tree Search

O primeiro passo é utilizar a heurística UCB1 para percorrer a árvore até um nó folha.

Se esse nó for terminal, atualize as estatísticas da árvore.

Monte Carlo Tree Search

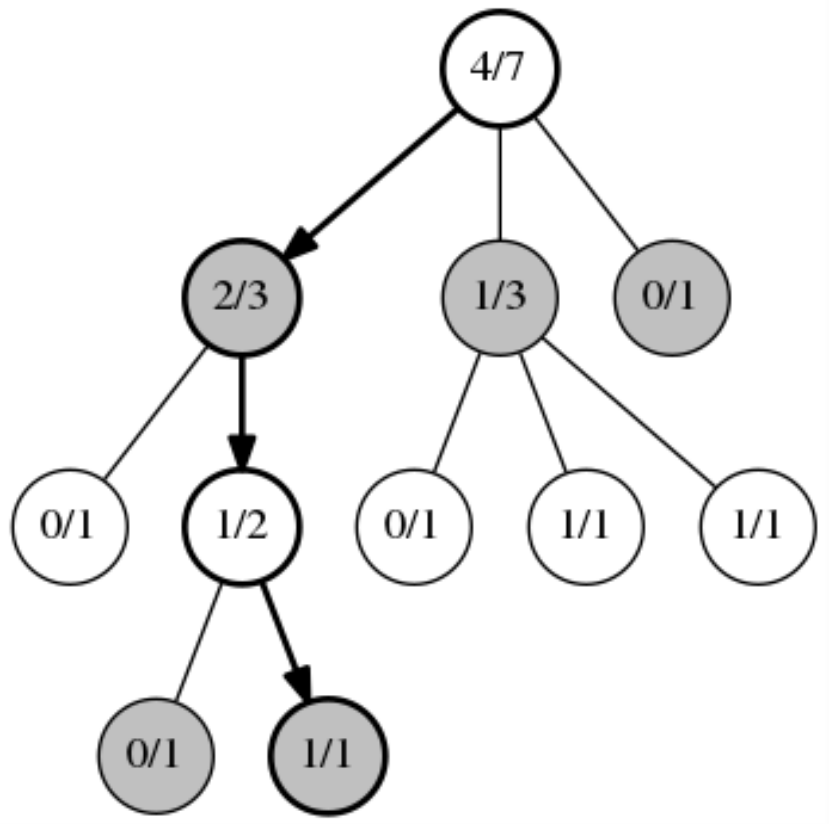


Figure 1:

https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/

Monte Carlo Tree Search

Não sendo terminal, escolha um movimento aleatório e adicione um nó folha (filho do nó escolhido) com estatísca inicial de 0 vitórias em 0 jogadas.

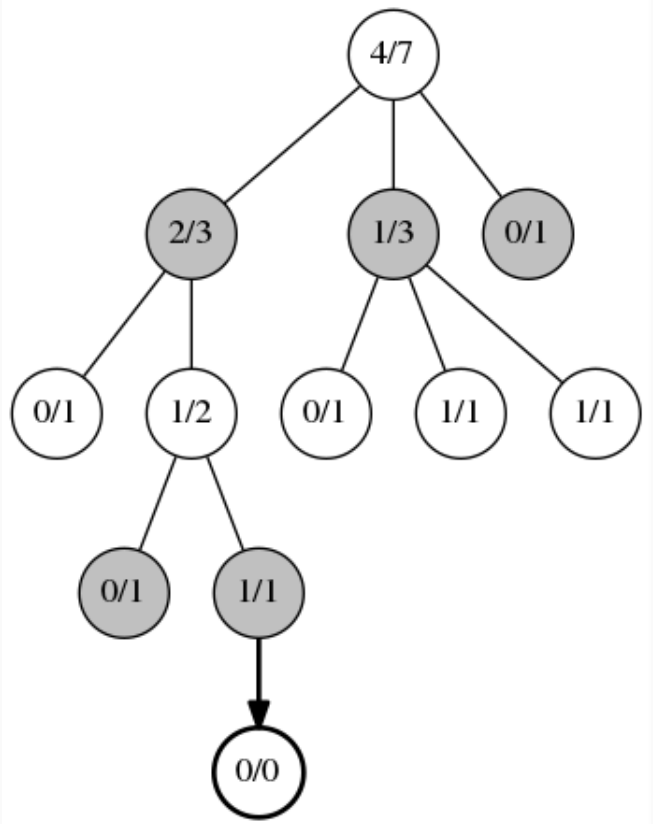


Figure 2:

https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/

Monte Carlo Tree Search

Em seguida, jogue a partida até um nó terminal utilizando jogadas aleatórias ou algum dos métodos ensinados anteriormente.

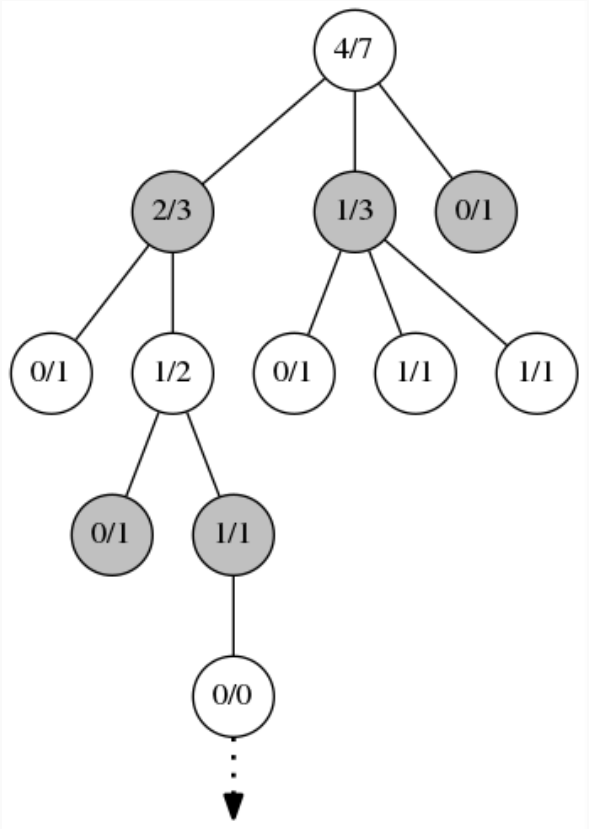


Figure 3:

https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/

Monte Carlo Tree Search

Atualize as informações do nó propagando a informação de vitória/derrota e partidas jogadas para cima até a raíz.

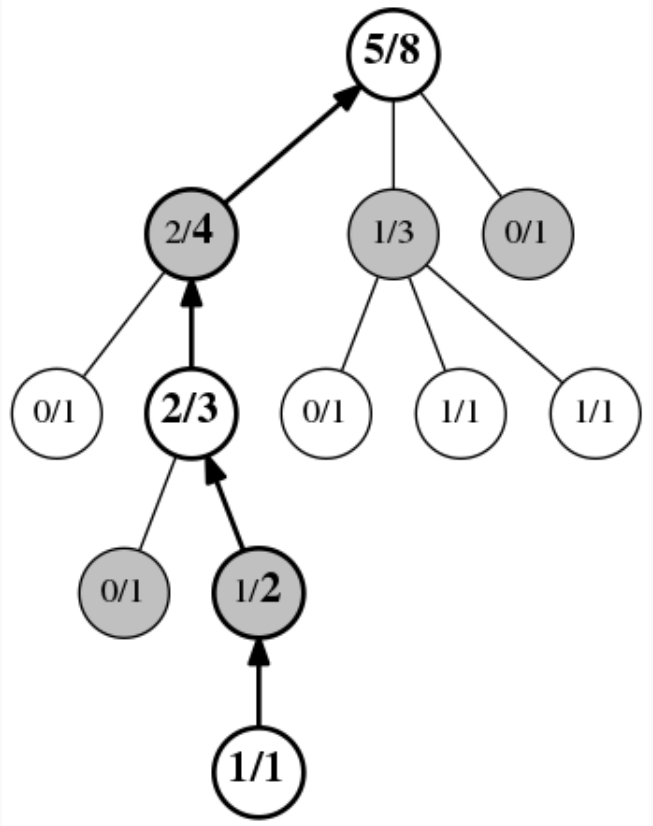


Figure 4:

https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/

Monte Carlo Tree Search

Processo simples:

```
def MCTS(arvore):
  node = seleciona(arvore)
  if Terminal(node):
      resultado = simula(arvore, node)
      return retropropaga(node, resultado)
  node, arvore = expande(arvore, node)
  resultado    = simula(arvore, node)
  arvore       = retropropaga(node, resultado)
  return arvore
```

Monte Carlo Tree Search

A chamada do MCTS deve ser:

```
def geraArvore(s0):
  arvore = s0
  while insuficiente(arvore):
    arvore = MCTS(arvore)
  return arvore
```

Monte Carlo Tree Search

Note que esse algoritmo pode ser utilizado com jogos determinísticos, com incerteza, multi-agente, etc. sem muitas adaptações.

Exercicio

Imagine um jogo em que a cada jogada você deve escolher uma dentre duas moedas para jogar. Toda vez que a moeda der *cara* você ganha 5 pontos, sempre que der *coroa* você perde 10. Uma dessas moedas tem 60% de dar cara enquanto a outra é uma moeda honesta.

Simule algumas possíveis iterações do algoritmo UCB1 para determinar a melhor jogada.