# Inteligência Artificial

Prof. Fabrício Olivetti de França  Prof. Denis Fantinato

3º Quadrimestre de 2019

# Busca Competitiva

# Busca Competitiva

Quando falamos sobre agentes mencionamos alguns cenários em que o ambiente era **competitivo**, ou seja, múltiplos agentes competindo pela melhor solução.

Esse tipo de ambiente é denominado **jogo**, e uma solução para tal cenário é a **busca competitiva**.

# Jogos

A **teoria dos jogos** (um ramo da economia) estuda ambientes multiagentes como um jogo sempre que a ação de um agente afeta outro agente significativamente.

Tais cenários podem ser tanto **cooperativos** como **competitivos**.

# Soma Zero

Na área de Inteligência Artificial, os tipos de jogos mais estudados são determinísticos, de jogadas alternadas, com dois jogadores e de **soma zero** ou **informação perfeita**.

# Soma Zero

Um jogo de soma zero é aquele que o **valor utilidade** ao final do jogo para os dois jogadores é igual e de sinal oposto.

**Valor utilidade** é a pontuação final do agente.

# Soma Zero

No jogo **pedra-papel-tesoura** com dois agentes temos os possíveis resultados finais:

| Jogador 1 | Jogador 2 |
|---|---|
| -1 | 1 |
| 0 | 0 |
| 1 | -1 |

# Jogos em IA

O estudo de jogos em IA despertou interesse dos pesquisadores pois:

- São facilmente modelados como um problema de busca
- Apresentam um conjunto de estados intratável para busca exata
- Busca-se por um tempo de resposta rápido
- Devemos tomar decisões até mesmo quando não temos uma decisão ótima a ser feita

# Jogos em IA

- **Damas:** em 1950 temos o primeiro computador jogando damas, somente em 1994 o computador foi campeão mundial. Em 2007 já temos o agente perfeito.

# Jogos em IA

- **Xadrez:** em 1997 o Deep Blue derrotou Kasparov analisando 200 milhões de posições por segundo.

# Jogos em IA

- **Go:** em 2017 o AlphaGo venceu o atual campeão mundial utilizando Árvore de Busca de Monte Carlo e Deep Learning (redes neurais artificiais).

# Formalização de Jogos

# Jogos

Podemos formalizar um jogo como:

- $S_0$: o estado inicial do jogo (arranjo do tabuleiro de xadrez).
- $Jogador(s)$: indica qual jogador é o próximo a jogar no estado $s$.
- $Acoes(s)$: retorna as ações disponíveis no estado $s$ para o $Jogador(s)$.
- $Result(s, a)$: a transição de estados ao aplicar a ação $a$ no estado $s$.
- $Terminal(s)$: retorna se o jogo atingiu um estado terminal.
- $Utilidade(s, p)$: retorna o valor de utilidade para o jogador $p$ no estado final $s$.

# Exercício

Formalize o jogo Pedra-Papel-Tesoura na sua linguagem de programação favorita.

# Exercício

Formalize o jogo da velha na sua linguagem de programação favorita.

# Agente inteligente em Jogos

Um agente inteligente em jogos é aquele que define uma política $\pi(s)$ que dado um estado $s$ retorna a melhor ação $a$.

Você consegue imaginar uma função $\pi(s)$ para os jogos dos exercícios anteriores?

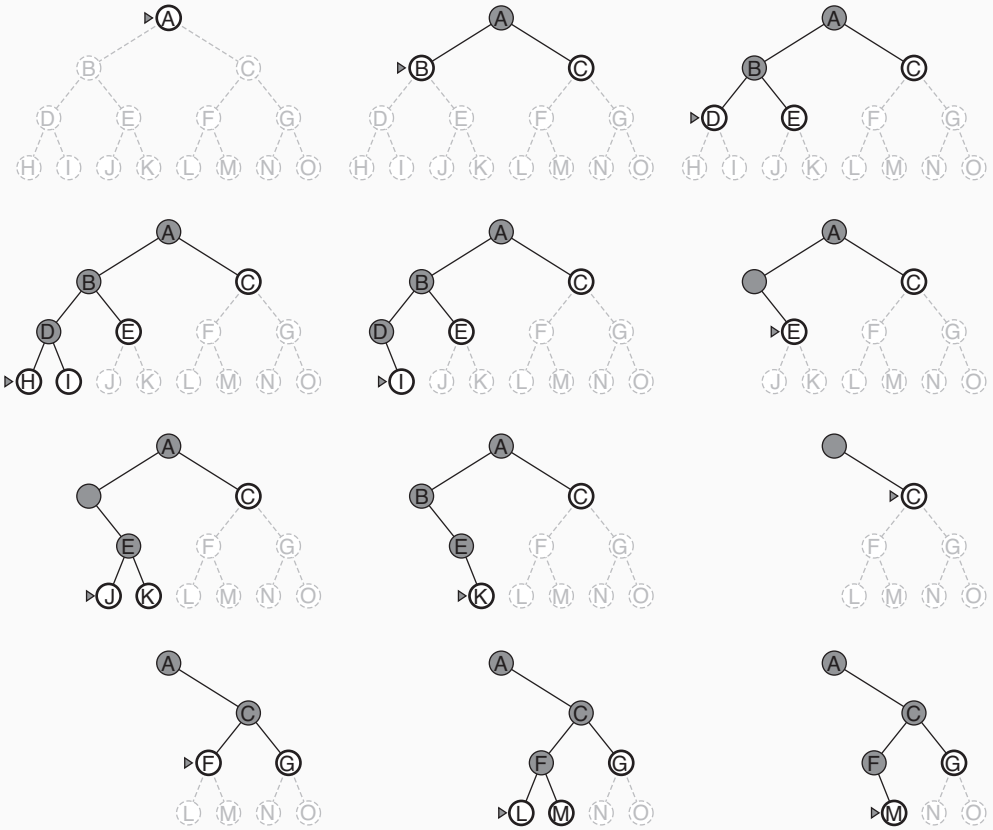
# Árvore de Jogos

# Árvore de Jogos

Uma vez formalizado, podemos encontrar a solução de um jogo utilizando árvores de busca.

# Árvore de Jogos

Em um jogo com um único agente, temos a árvore de busca tradicional:




Os nós terminais são complementados pela utilidade do estado.

# Árvore de Jogos

Em jogos competitivos temos a árvore Minimax, em que cada ramo da árvore é intercalado entre um ramo **MAX** e um ramo **MIN**.

# Árvore de Jogos

O ramo **MAX** representa o Jogador 1, ele tem o objetivo de executar as ações que **maximizam** sua utilidade.

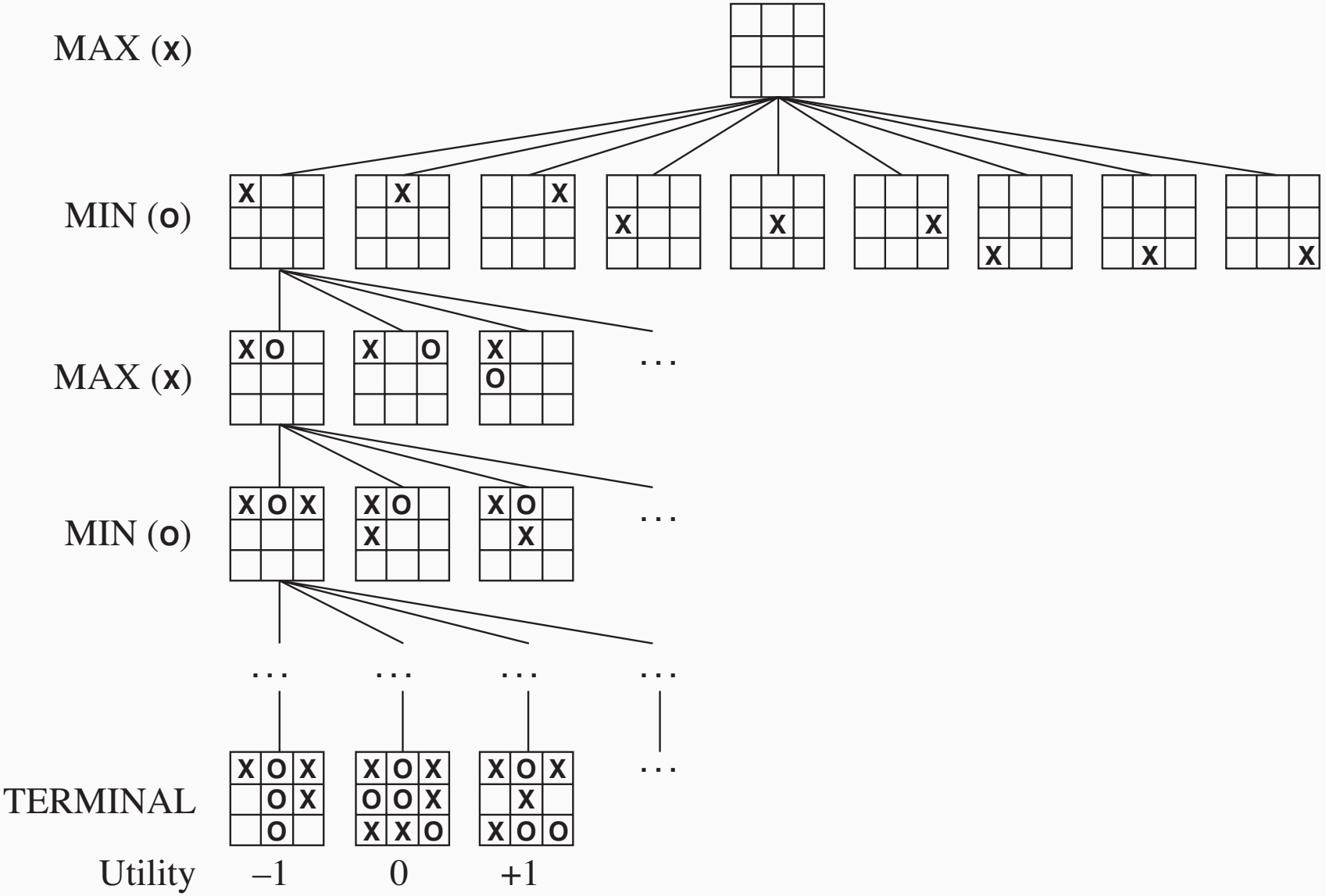
# Árvore de Jogos

O ramo **MIN** representa o Jogador 2, ele tem o objetivo de executar as ações que **minimizam** a utilidade do jogador 1.

# Árvore de Jogos

A ideia geral é que em um jogo competitivo, dada uma ação ótima do Jogador 1, o Jogador 2 irá executar uma ação que minimize a utilidade dele, pois sendo um jogo de soma zero, maximizará sua própria utilidade.

# Árvore de Jogos



# Busca Minimax

# Busca Minimax

Assume um jogo determinístico!

O Jogador 1 maximiza a utilidade enquanto o jogador 2 minimiza a utilidade.

# Busca Minimax

Ideia: construir a árvore Minimax e a política $\pi(s)$ será seguir pelo caminho de $s$ que leva a maior utilidade assumindo que seu adversário também é ótimo.

E se o adversário não for ótimo?

# Valor Minimax

O **valor Minimax** de um nó é a maior (menor) utilidade que pode ser obtida partindo do estado $s$ se você for o jogador MAX (MIN).

# Valor Minimax

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

# Valor Minimax

```
def maxVal(s):
  v = -Inf
  for a in Acoes(s):
    v = max(v, MiniMaxUtil(Result(s,a), MIN))
  return v
```

# Valor Minimax

```
def minVal(s):
  v = Inf
  for a in Acoes(s):
    v = min(v, MiniMaxUtil(Result(s,a), MAX))
  return v
```

# Algoritmo Minimax

Com isso, o algoritmo Minimax é definido para o Jogador 1 como:

```
def Minimax(s):
 return argmax([(a, MiniMaxUtil(Result(s,a), MIN))
                        for a in Acoes(s)])
```

Note que $s$ deve ser um estado em que $Jogador(s) = MAX$.

# Eficiência do Minimax

A complexidade de tempo e espaço da Busca Minimax é proporcional ao fator de ramificação $b$ e a profundidade da árvore de busca $m$:

- Tempo: $O(b^m)$
- Espaço: $O(bm)$

# Eficiência do Minimax

O xadrez possui $b \approx 35, m \approx 100$.

Então o custo de tempo é $\approx 35^{100} \approx 2.5 \cdot 10^{154}$

# Eficiência do Minimax

Solução: limitar a profundidade da busca ao invés de chegar até o nó folha.

Porém, precisamos de uma heurística para estimar o valor utilidade de um nó intermediário.

# Exercício

Imagine a seguinte função de avaliação para um nó intermediário do jogo da velha:

$f(n) =$ quantidade de possíveis jogadas vencedoras.

Aplique o algoritmo Minimax expandindo apenas os $2$ nós com maior avaliação por essa heurística.

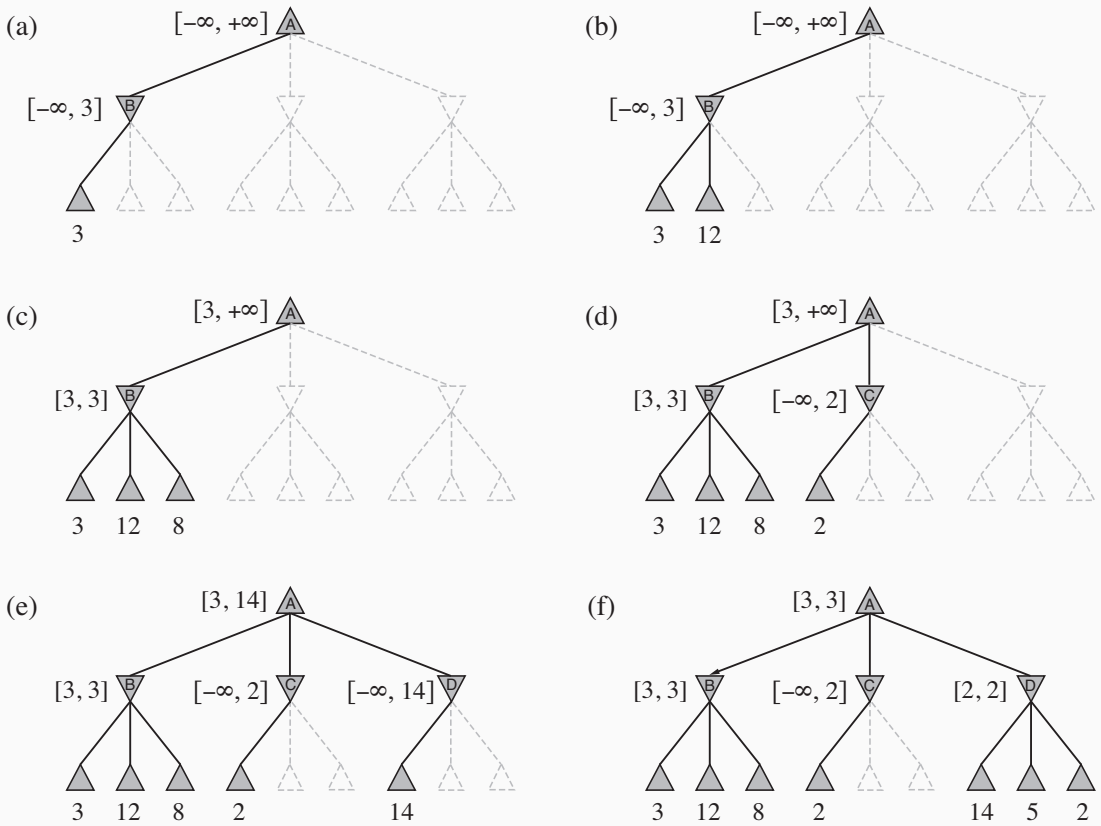
# Poda Alfa-Beta

# Poda Alfa-Beta

Uma outra alternativa é utilizar a informação obtida em uma busca em profundidade para deixar de explorar caminhos que sabemos que não serão utilizados.

# Poda Alfa-Beta

Considere a seguinte árvore de Jogo:



# Poda Alfa-Beta

Note que após expandir o primeiro ramo, o jogador 1 sabe que consegue pelo menos 3 pontos seguindo por ele.

Durante a expansão do segundo ramo, é encontrado uma solução de 2 pontos, que o jogador 2 irá preferir, logo esse segundo ramo deixa de ser interessante.

# Poda Alfa-Beta

O algoritmo agora fica:

```
def MinimaxUtil(s, p, Alfa, beta):
  if Terminal(s): return Utilidade(s, p)
  if p == MAX:    return maxVal(s, Alfa, beta)
  if p == MIN:    return minVal(s, Alfa, beta)
```

# Poda Alfa-Beta

```
def maxVal(s, Alfa, beta):
  v = -Inf
  for a in Acoes(s):
    v = max(v, MiniMaxUtil(Result(s,a), MIN, Alfa, beta))
    if v >= beta: return v
    Alfa = max(Alfa, v)
  return v
```

# Poda Alfa-Beta

```
def minVal(s, Alfa, beta):
  v = Inf
  for a in Acoes(s):
    v = min(v, MiniMaxUtil(Result(s,a), MAX, Alfa, beta))
    if v <= Alfa: return v
    beta = min(beta, v)
  return v
```

# Poda Alfa-Beta

A poda não modifica o valor de Minimax do nó raiz!

Ordenando os nós de tal forma a reduzir a necessidade de exploração aumenta o desempenho.

Mas ainda é complexo demais para ser a solução de jogos como o xadrez...

# Exercício

Construa a árvore Minimax com poda Alfa-Beta para o jogo da velha partindo da solução:

| | | |
|---|---|---|
| X | O | |
| X | X | |
| O | | |

Sendo a próxima jogada do **O**.