

Inteligência Artificial

Prof. Fabrício Olivetti de França Prof. Denis Fantinato

3º Quadrimestre de 2018



1. Busca Competitiva
2. Formalização de Jogos
3. Árvore de Jogos
4. Busca Minimax
5. Poda Alfa-Beta
6. Exercícios

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**.

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**.

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**.

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.

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

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

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

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

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

- **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

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 .

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

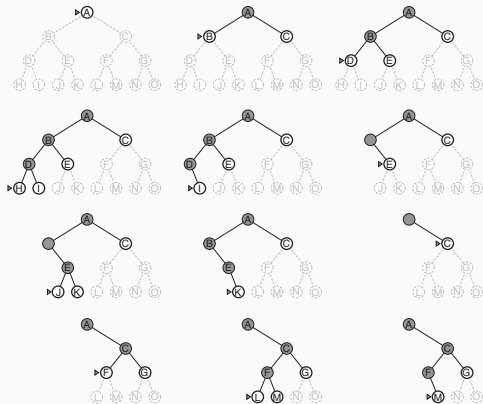
Você consegue imaginar uma função $\pi(s)$ para os jogo da velha ou pedra-papel-tesoura?

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.

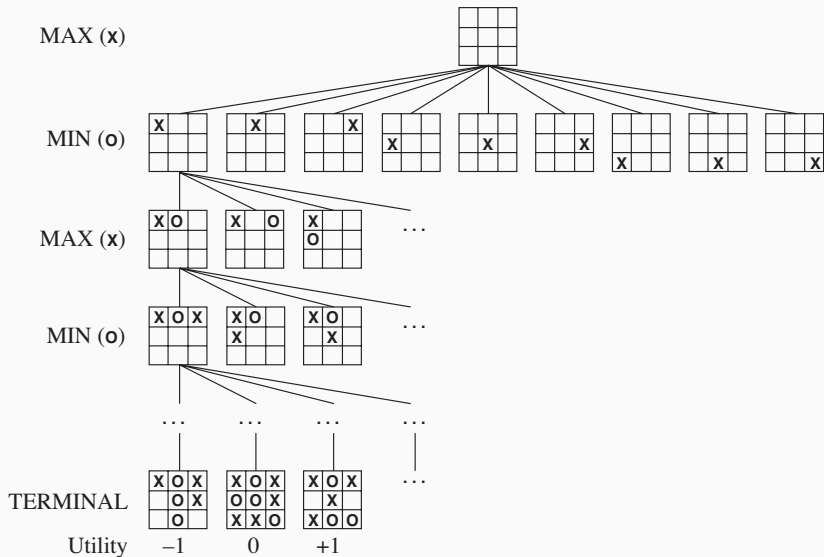
Em jogos competitivos temos a árvore Minimax, em que cada ramo da árvore é intercalado entre um ramo **MAX** e um ramo **MIN**.

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

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

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

Assume um jogo determinístico!

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

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?

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

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

Valor Minimax

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

Valor Minimax

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

Algoritmo Minimax

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

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

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

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: $O(bm)$

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

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

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

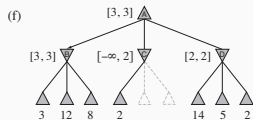
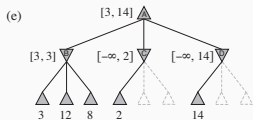
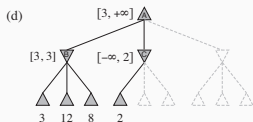
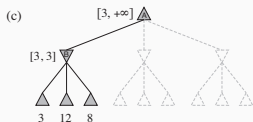
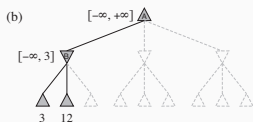
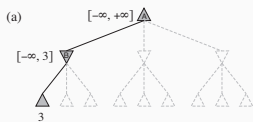
Porém, precisamos de uma heurística para estimar o valor utilidade de um nó intermediário.

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:



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.

O algoritmo agora fica:

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

Poda Alfa-Beta

```
1 def maxVal(s, Alfa, beta):
2     v = -Inf
3     for a in Acoes(s):
4         v = max(v, MiniMaxUtil(Result(s,a), MIN, Alfa, beta))
5         if v >= beta: return v
6         Alfa = max(Alfa, v)
7     return v
```

Poda Alfa-Beta

```
1 def minVal(s, Alfa, beta):
2     v = Inf
3     for a in Acoes(s):
4         v = min(v, MiniMaxUtil(Result(s,a), MAX, Alfa, beta))
5         if v <= Alfa: return v
6         beta = min(beta, v)
7     return v
```

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. . .

Exercícios

Defina em sua linguagem de programação favorita as funções que formalizam um jogo para:

- Pedra-Papel-Tesoura
- Jogo da Velha

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.

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**.