Inteligência Artificial - Notas de aula

Raoni F. S. Teixeira

Aula 4 - Busca Competitiva

1 Introdução

Essa aula apresenta o algoritmo de busca *Minmax* utilizado em jogos competitivos. Nesse tipo de ambiente, o agente não pode prever o comportamento do adversário, o que inviabiliza o uso dos algoritmos abordados anteriormente.

2 Jogo de dois jogadores

Jogos como xadrex ou damas ocorrem em ambientes determinísticos e observáveis, onde os jogadores se alternam até o fim da partida. Ao final, o vencedor recebe uma recompensa e o perdedor, uma penalidade. Em caso de empate, ambos recebem a mesma pontuação.

Formalmente, um jogo é descrito por seis componentes:

1. um estado inicial $s_i \in \mathcal{S}$ que especifica como o jogo começa;
2. uma função J, $J(s) \mapsto [\text{jogador}_1, \text{jogador}_2]$, que informa qual jogador deve agir em cada estado s;
3. uma função A, $A(s) \mapsto \mathcal{A}$, que devolve as ações válidas no estado s;
4. um modelo de transição T, $T(s, a) \mapsto s'$, que especifica o próximo estado (s') após aplicar a ação a no estado s;
5. uma função F, $F(s) \mapsto [\text{Verdadeiro}, \text{Falso}]$, que devolve o valor Verdadeiro se o jogo termina em s e Falso caso contrário e
6. Uma função utilidade $U(s) \mapsto [-\infty, \infty]$, que atribui valor numérico a estados terminais, do ponto de vista de um jogador. Se um jogador vence, $U(s) = 1$; se perde, $U(s) = -1$ e $U(s) = 0$ em caso de empate.

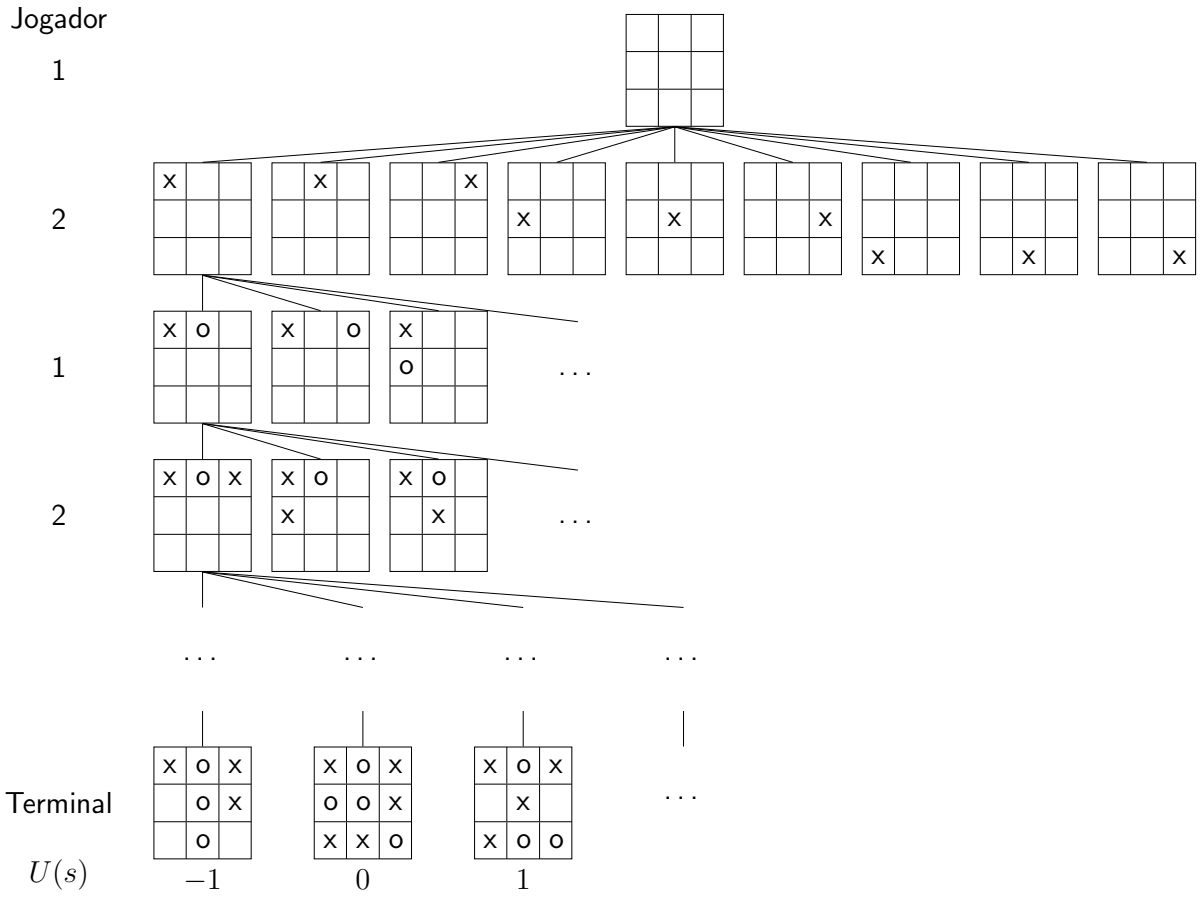


Figura 1: Árvore de partidas do jogo da velha. Os nós são os estados do jogo e cada caminho da raiz à folha é uma partida.

A Figura 1 mostra a árvore de partidas do jogo da velha, com o estado inicial na raiz e os estados terminais nas folhas. Cada caminho da raiz a uma folha é uma sequência completa de jogadas. A função U atribui valores aos nós terminais para indicar vitória (1), derrota (-1) ou empate (0).

3 Minmax

Nesse ambiente competitivo, o agente não pode planejar considerando apenas suas próprias jogadas, pois deve levar em conta que o adversário - sendo racional - sempre escolherá as jogadas que minimizam a utilidade do agente. Diferentemente de problemas de busca convencionais, onde se busca um caminho único até o objetivo, aqui precisamos de uma **estratégia com-**

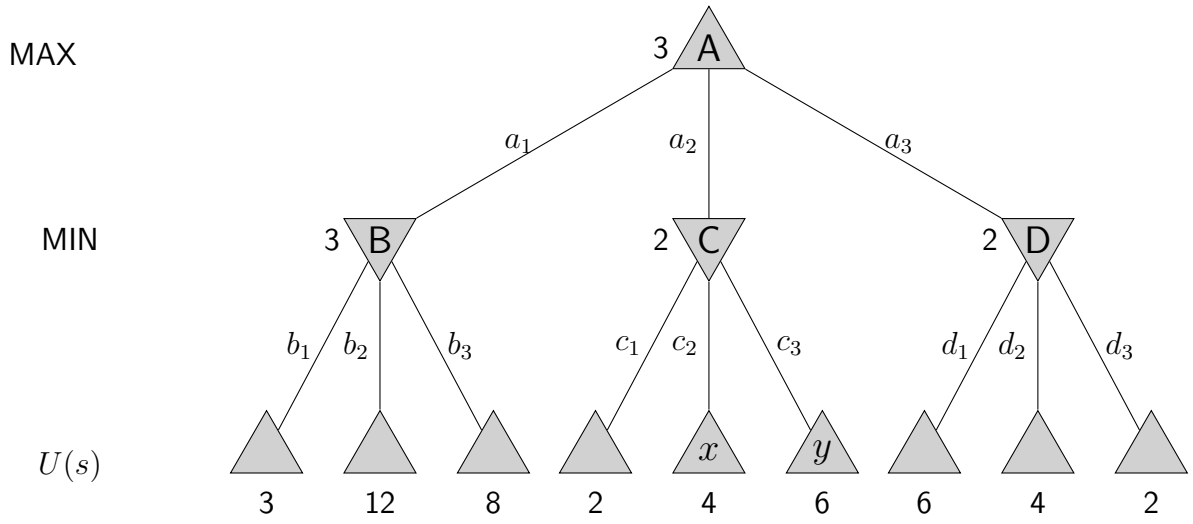


Figura 2: Árvore de um jogo de dois jogadores. $\triangle$ e $\triangledown$ indicam o primeiro jogador (agente) e seu adversário. Os números nas folhas são os valores da função $U(s)$.

pleta que especifique todos os movimentos do agente em resposta a qualquer jogada possível do oponente.

Essa dinâmica caracteriza o problema de busca Minimax: o agente maximiza sua utilidade assumindo que o adversário sempre minimizará esse mesmo valor. Formalmente:

$$\mathrm{Minmax}(s) = \begin{cases} U(s), & \text{se } F(s) \text{ é Verdadeiro} \\ \max\limits_{a \in A(s)} \mathrm{Minmax}(T(s,a)), & \text{se } J(s) \text{ é o agente} \\ \min\limits_{a \in A(s)} \mathrm{Minmax}(T(s,a)), & \text{se } J(s) \text{ é o adversário} \end{cases} \tag{1}$$

A função Minmax avalia todas as combinações possíveis e devolve a jogada ótima (maior valor de U), assumindo que escolhas racionais de ambos os jogadores.

A Figura 2 ilustra a árvore de decisões de um jogo entre dois jogadores. Cada estado não terminal (A, B, C e D) possui três ações possíveis. Os valores nas folhas representam os resultados da função $U(s)$: o jogador $\triangle$ busca maximizar esses valores, enquanto seu oponente, $\triangledown$, tenta minimizá-los.

Nesse cenário, $\triangle$ presume que o adversário escolherá as ações b_1, c_1 e d_3, que minimizam $U(s)$. Com base nessa suposição, $\triangle$ escolhe a ação a_1, pois ela leva ao maior valor possível de Minmax.

Como o número de estados cresce exponencialmente com a profundidade da árvore, é necessário aplicar estratégias para reduzir o tempo de execução do algoritmo. As duas mais

comuns são:

- limitar a profundidade da árvore de busca e
- podar partes da árvore que não influenciam o resultado.

O controle de profundidade depende de uma função heurística que avalia estados não terminais. O algoritmo a seguir implementa a Equação 1 com esse controle. A função Eval(s) estima o quão favorável é um estado s para o agente. Dados um estado inicial s e um nível de profundidade p, o algoritmo retorna o valor Minmax correspondente à melhor jogada possível.

```
MinMax(s, p)
1  if F(s) = Verdadeiro ou p ≤ 0 then-
2       return Eval(s)
3  if J(s) = agente then-
4       return MaxValor(s, p)
5  else-
6       return MinValor(s, p)
```

```
MaxValor(s, p)
1  v ← −∞
2  foreach a ∈ A(s) do-
3       v ← max(v, MinMax(T(s, a), p − 1))
4  return v
```

```
MinValor(s, p)
1  v ← +∞
2  foreach a ∈ A(s) do-
3       v ← min(v, MinMax(T(s, a), p − 1))
4  return v
```

Embora o Minimax seja eficaz, a enumeração completa de todas as jogadas é computacionalmente proibitiva devido ao seu custo exponencial. A poda alfa-beta ameniza esse problema permitindo eliminar ramos da árvore que não influenciarão o resultado final, como veremos a seguir.

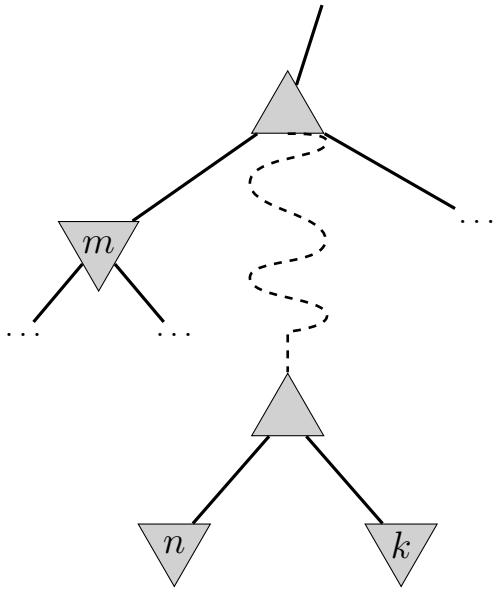


Figura 3: Poda alpha-beta. Se m ou n são melhores que k, então k pode ser podado.

4 Poda alpha-beta

A poda alfa-beta (α-β) **acelera** o Minimax ao eliminar subárvores irrelevantes para o resultado final, reduzindo significativamente o número de nós avaliados.

Este é o caso das jogadas c_2 e c_3 na sub-árvore C da Figura 2 que não alteram o valor de Minmax:

$$\begin{aligned}\mathrm{MinMax}(\mathsf{A}) &= \mathsf{max}(\mathsf{min}(\mathsf{B}), \mathsf{min}(\mathsf{C}), \mathsf{min}(\mathsf{D})) \\ &= \mathsf{max}(\mathsf{min}(3, 12, 8), \mathsf{min}(2, x, y), \mathsf{min}(6, 4, 2)) \\ &= \mathsf{max}(3, \mathsf{min}(2, x, y), 2) \\ &= 3 \qquad (\forall \text{ valores de x, y})\end{aligned}$$

A ideia da poda alpha-beta é otimizar a busca minmax evitando a exploração de subárvores que não influenciam a decisão final. Para isso, o algoritmo mantém dois valores: α, o melhor valor já encontrado para o jogador maximizador, e β, o melhor valor já encontrado para o minimizador. Esses limites permitem interromper precocemente a exploração de certos ramos da árvore de decisão quando se verifica que eles não podem melhorar a jogada já conhecida.

A Figura 3 ilustra esse princípio. Suponha que o jogador tenha a opção de seguir para o nó k. Se existir uma jogada anterior melhor no mesmo nível (por exemplo, n) ou em qualquer ponto acima na árvore (por exemplo, m), então k nunca será selecionado. Assim, ao identificarmos

que k não pode superar as alternativas já conhecidas, podemos interromper a busca por seus descendentes e podar esse ramo da árvore.

A Figura 4 mostra a evolução dos valores α e β durante a execução do algoritmo. No início, $\alpha = -\infty$ e $\beta = \infty$. Cada subfigura (de **a** a **f**) representa um estágio da exploração da árvore, demonstrando como a poda reduz o número de nós visitados.

Na Figura 4a, o valor 3 é encontrado na primeira folha da subárvore B, definindo $\beta = 3$ naquele nó. A Figura 4b mostra que a segunda folha (valor 12) não altera β, pois o jogador $\bigtriangledown$ evitará essa opção. Na Figura 4c, a terceira folha de B tem valor 8. Com todas as jogadas exploradas, o valor final de minmax para B é 3. Esse valor define $\alpha = 3$ em A e será repassado às subárvores C e D.

A Figura 4d exibe a primeira folha de C com valor 2. Como $\alpha = 3$, os nós restantes de C são podados imediatamente.

Na Figura 4e, o valor 6 da primeira folha de D supera $\alpha = 3$, exigindo busca adicional. Com os três filhos da raiz parcialmente avaliados, sabemos que seu valor será no máximo 6.

A Figura 4f completa D com os valores 4 e 2, resultando em valor final 2. Assim, a jogada ótima para $\bigtriangleup$ em A é dirigir-se a B (jogada a_1 na Figura 2).

As Figuras 4d, 4e e 4f revelam como a ordem das jogadas afeta a eficiência da poda. Na Figura 4d, a sequência 2, 4, 6 permite poda precoce. Já nas Figuras 4e e 4f, a ordem 6, 4, 2 impede economia equivalente.

O algoritmo abaixo implementa poda alpha-beta com controle de profundidade. A função Eval representa uma heurística de avaliação de estados. Para executar:

$$\textsc{MinmaxPoda}(s_0, p, -\infty, \infty)$$

onde s_0 é o estado inicial e p a profundidade máxima de busca.

$\textsc{MinMaxPoda}(s, p, \alpha, \beta)$

```
1  if F(s) = Verdadeiro ou p ≤ 0 then-
2        return Eval(s)
3  if J(s) = agente then-
4        return MaxPoda(s, p, α, β)
5  else-
6        return MinPoda(s, p, α, β)
```

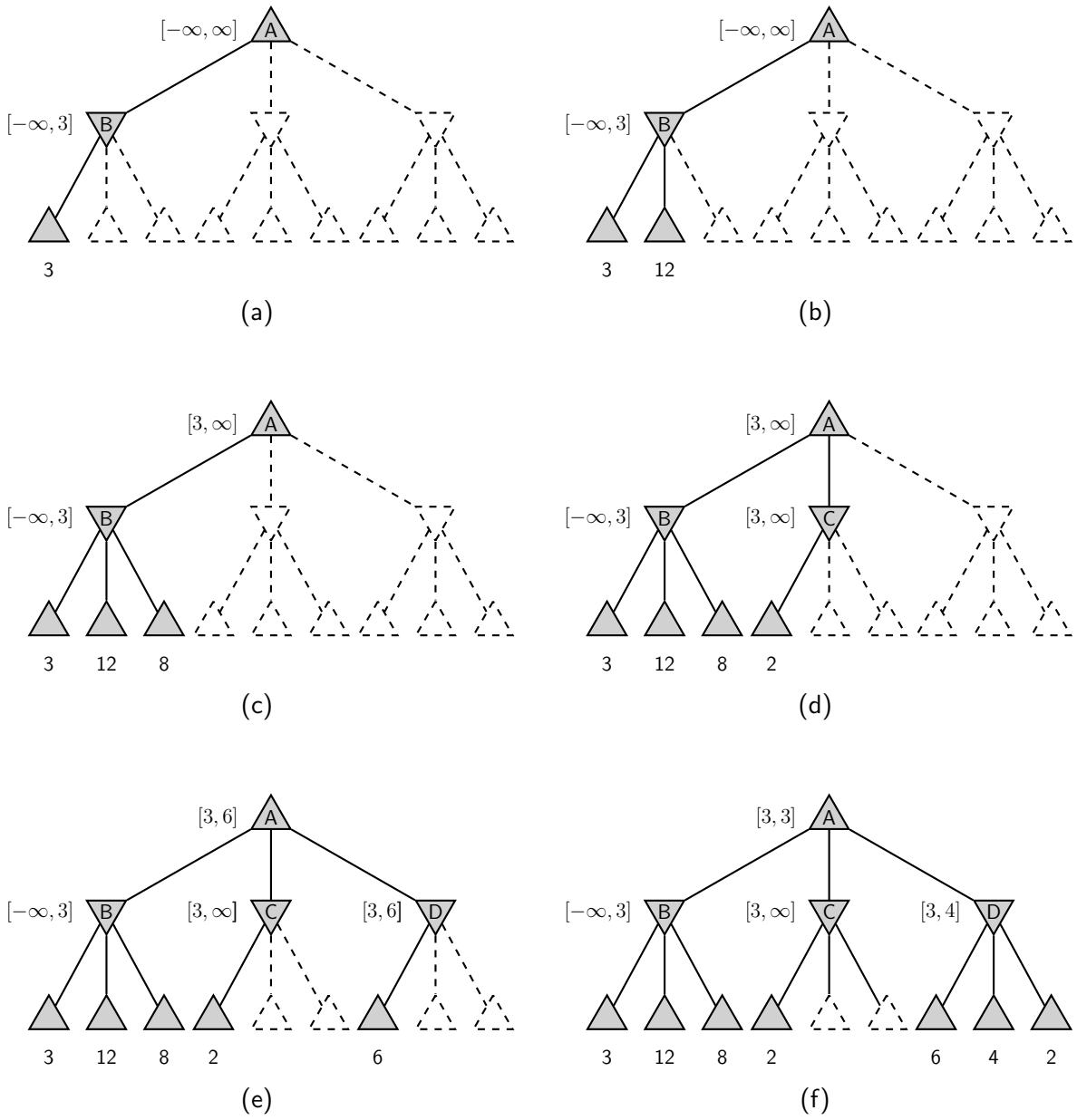


Figura 4: Simulação da poda alpha-beta para árvore da Figura 1. Os valores de alfa e beta estão indicados entre colchetes e são inicialmente $-\infty$ e ∞. A busca é interrompida na sub-árvore C — os estados tracejados na parte (f) foram podados.

```
MaxPoda(s, p, α, β)
1  v ← −∞
2  foreach a ∈ A(s) do-
3      v ← max(v, MinMaxPoda(T(s, a), p − 1, α, β))
4      if v ≥ β then-
5          return v
6      α ← max(α, v)
7  return v
```

```
MinPoda(s, p, α, β)
1  v ← +∞
2  foreach a ∈ A(s) do-
3      v ← min(v, MinMaxPoda(T(s, a), p − 1, α, β))
4      if v ≤ α then-
5          return v
6      β ← min(β, v)
7  return v
```

Prova de Correção da Poda α-β

O algoritmo da poda MinMaxPoda produz o mesmo resultado do MinMax. Em outras palavras, se T é uma árvore de jogo então alpha-beta(T) = minimax(T).

Prova, por indução na altura h de T:

Base da indução: $h = 0$ (nó folha).

Nesse caso, $\text{MinMaxPoda}(T) = \text{Eval}(T)$ e $\text{MinMax}(T) = \text{Eval}(T)$, portanto $\text{MinMaxPoda}(T) = \text{MinMax}(T)$.

Passo da indução: $h > 0$.

Suponha que a igualdade vale para toda subárvore própria de T (esta é a hipótese de indução).

Caso 1: T é nó maximizador.

Seja $C = \{c_1, c_2, \ldots, c_k\}$ os filhos de T.

Por hipótese de indução, $\text{MinMaxPoda}(c_i) = \text{MinMax}(c_i)$ para todo i.

O algoritmo minimax calcula:

$$\text{MinMax}(T) = \max\{\text{MinMax}(c_1), \text{MinMax}(c_2), \ldots, \text{MinMax}(c_k)\}$$

O algoritmo $\textsc{MinMaxPoda}$ com $\alpha = -\infty$, $\beta = +\infty$ calcula sequencialmente:

$$\begin{aligned} v_1 &= \textsc{MinMaxPoda}(c_1) \\ v_2 &= \textsc{MinMaxPoda}(c_2) \\ &\vdots \\ v_k &= \textsc{MinMaxPoda}(c_k) \end{aligned}$$

e retorna $\max\{v_1, v_2, \ldots, v_k\}$.

Pela hipótese de indução, $v_i = \textsc{MinMax}(c_i)$ para todo i.

Portanto, $\textsc{MinMaxPoda}(T) = \max\{v_1, \ldots, v_k\} = \max\{\textsc{MinMax}(c_1), \ldots, \textsc{MinMax}(c_k)\} = \textsc{MinMax}(T)$.

Falta agora entender quando a poda ocorre e por que é segura:

- A poda acontece quando algum $v_i \geq \beta$
- O valor β vem de um nó ancestral minimizador e representa um **limite superior** que o maximizador atual não pode exceder
- Se $v_i \geq \beta$, então $\textsc{MinMax}(T) \geq v_i \geq \beta$
- Porém, algum ancestral minimizador já encontrou uma opção com valor $\leq \beta$
- Portanto, explorar mais filhos não alterará a decisão final do ancestral minimizador
- Os filhos podados são **irrelevantes** para o valor final da raiz

Como $\beta = +\infty$ na raiz, nenhuma poda acontece no primeiro nível. Em níveis inferiores, a poda elimina apenas ramos que não afetam o valor final.

Portanto, $\textsc{MinMaxPoda}(T) = \max\{v_1, \ldots, v_k\} = \max\{\textsc{MinMax}(c_1), \ldots, \textsc{MinMax}(c_k)\} = \textsc{MinMax}(T)$.

Caso 2: T é nó minimizador.

Análogo ao Caso 1, trocando $\max$ por $\min$, α por β, e ajustando o argumento da poda: a poda ocorre quando $v_i \leq \alpha$, onde α representa um limite inferior vindo de um ancestral maximizador.

Portanto, $\textsc{MinMaxPoda}(T) = \textsc{MinMax}(T)$, como queríamos demonstrar.

5 Discussão Final

O algoritmo Minmax tem origem na teoria dos jogos desenvolvida por John von Neumann e Oskar Morgenstern, que formalizaram a ideia de estratégias ótimas em jogos de soma zero no clássico *Theory of Games and Economic Behavior* [vNM44]. Em Inteligência Artificial, o Minmax foi popularizado com o trabalho pioneiro de Arthur Samuel [Sam59], em seu programa de damas, e posteriormente por Allen Newell e Herbert Simon, no sistema *Chess Program* [1].

O algoritmo permanece até hoje como base para sistemas de decisão em jogos de dois jogadores com turnos alternados, como xadrez, damas, jogo da velha, go e diversos jogos modernos. Combinado com heurísticas de avaliação e técnicas como poda alfa-beta, o Minmax é capaz de realizar buscas profundas de forma eficiente.

Uma das aplicações mais notáveis é o motor de xadrez *Stockfish*[2], considerado um dos mais fortes do mundo. Ele implementa variantes otimizadas do Minmax com extensões de busca, transposition tables e heurísticas sofisticadas para avaliar bilhões de posições. O algoritmo também está presente em IA de jogos clássicos como *Othello*, *Connect Four*, e diversos jogos de tabuleiro em ambientes acadêmicos e comerciais.

Outras aplicações incluem:

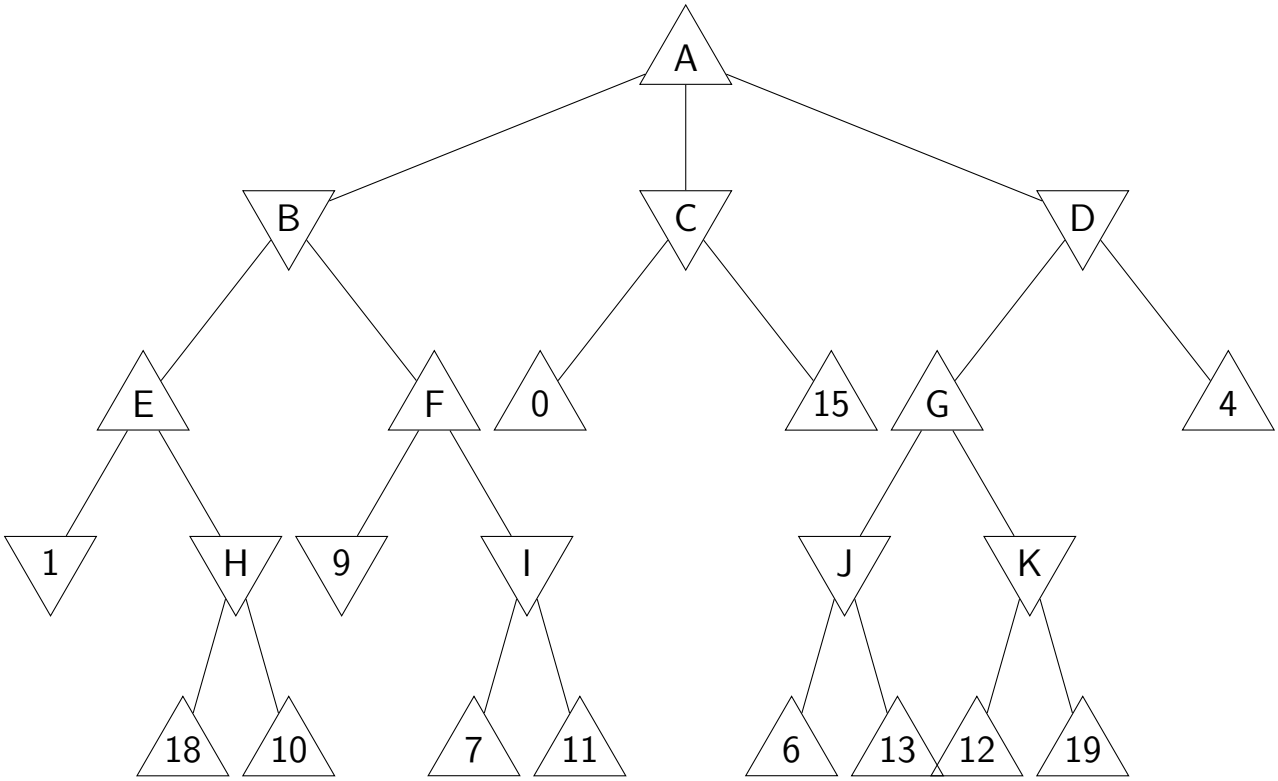
- planejamento adversarial em ambientes simulados;
- tomada de decisão em agentes multiagente;
- robótica competitiva (por exemplo, futebol de robôs);
- jogos eletrônicos com agentes NPC que antecipam ações do jogador [YT18].

Para jogos com espaços de estados muito grandes, como go ou StarCraft, variantes mais avançadas como *Monte Carlo Tree Search* (MCTS) e redes neurais profundas substituem ou complementam o Minmax tradicional.

6 Exercícios

1. Simule o mimax com poda alpha-beta para a Figura 2.
2. Como a busca minimax modela a competição em jogos de dois jogadores?
3. Calcule o valor de Minmax de cada nó árvore a seguir e indique os nós podados pela poda alfa-beta.
4. De que forma a poda alfa-beta melhora a eficiência da busca minimax?

[1] https://www.chessprogramming.org/NSS
[2] https://github.com/official-stockfish/Stockfish



Referências

[Sam59] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[vNM44] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[YT18] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Springer, 2018.