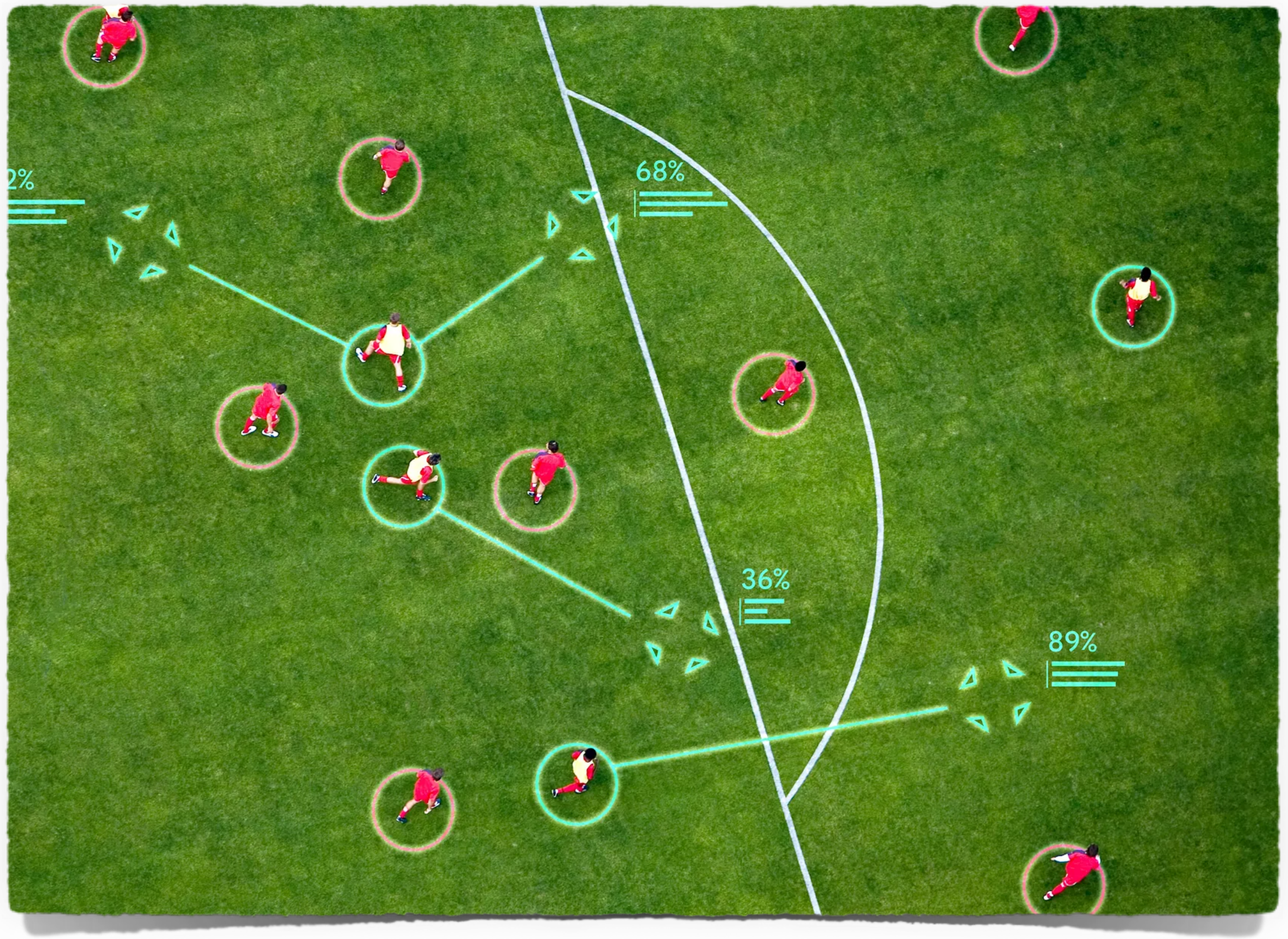
INF623
2024/1

UFV

Inteligência Artificial

A6: Busca competitiva I

IA na mídia



19/03/2024

DeepMind Tactic AI
Modelo desenvolvido em parceria com Liverpool FC para auxiliar na criação de estratégias de jogo

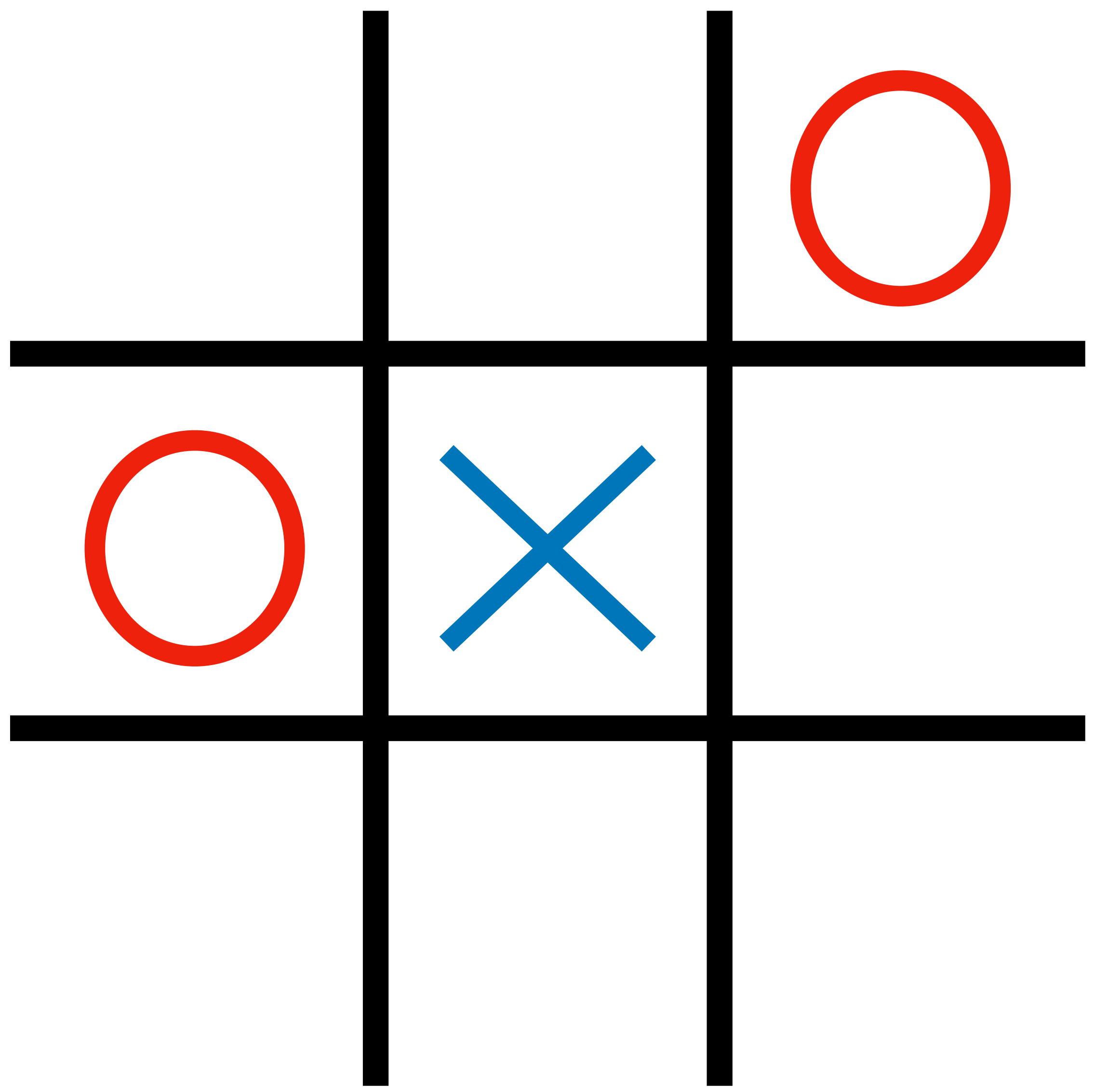
https://deepmind.google/discover/blog/tacticai-ai-assistant-for-football-tactics/

Plano de aula

- Intro. à Teoria dos jogos
- Jogos como problema de busca
- Algoritmo minimax
- Poda alpha-beta
- Funções de avaliação

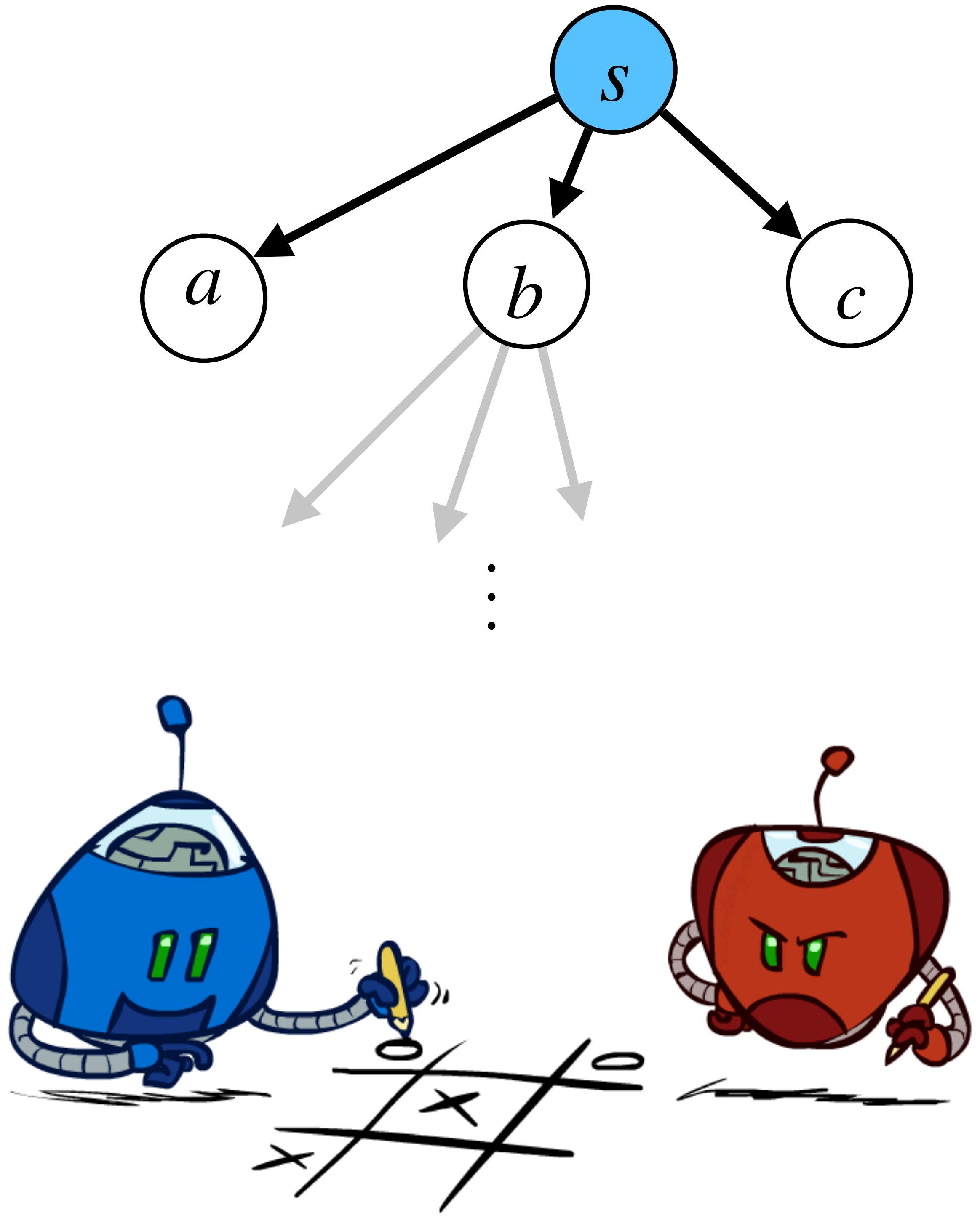
Exemplo 1: jogo da velha

Considere o tabuleiro de jogo da velha abaixo e que você é o jogador **X**. Qual é a melhor jogada que você pode fazer?



Agentes racionais para jogos

Para resolver problemas desse tipo, chamados de **jogos de soma zero**, um agente assume que o jogo é representado por um **espaço de estados** e que objetivo é encontrar uma **estratégia (política)** que recomende **jogadas** (ações) a cada estado que o leve a vitória.

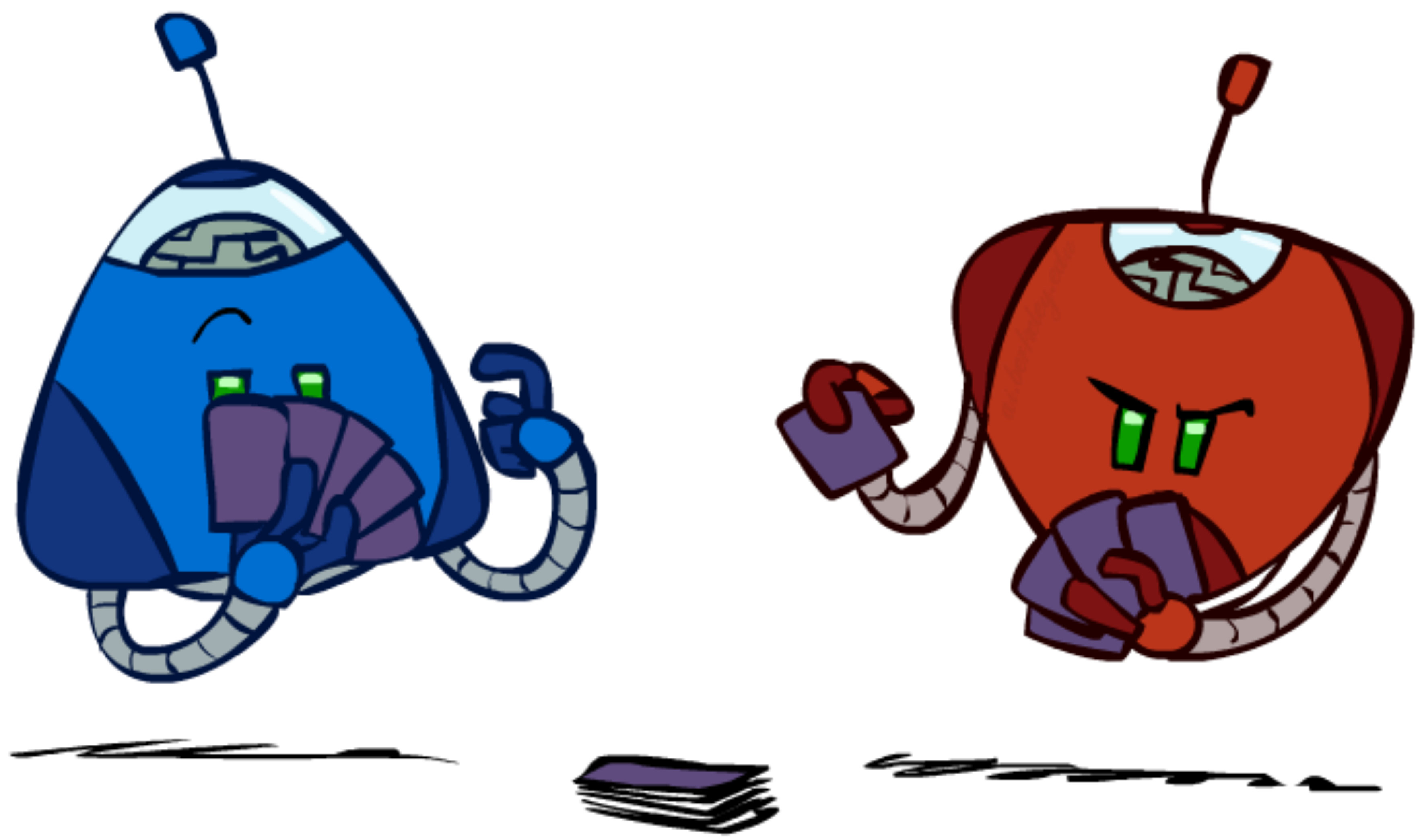


- **Jogadas** (→) modificam o estado corrente, e **não** possuem custos;
- Uma solução é um **política** – uma função P(s) que mapeia um estado s em uma ação a;
- O agente deve considerar o que seu adversário vai fazer para planejar suas ações

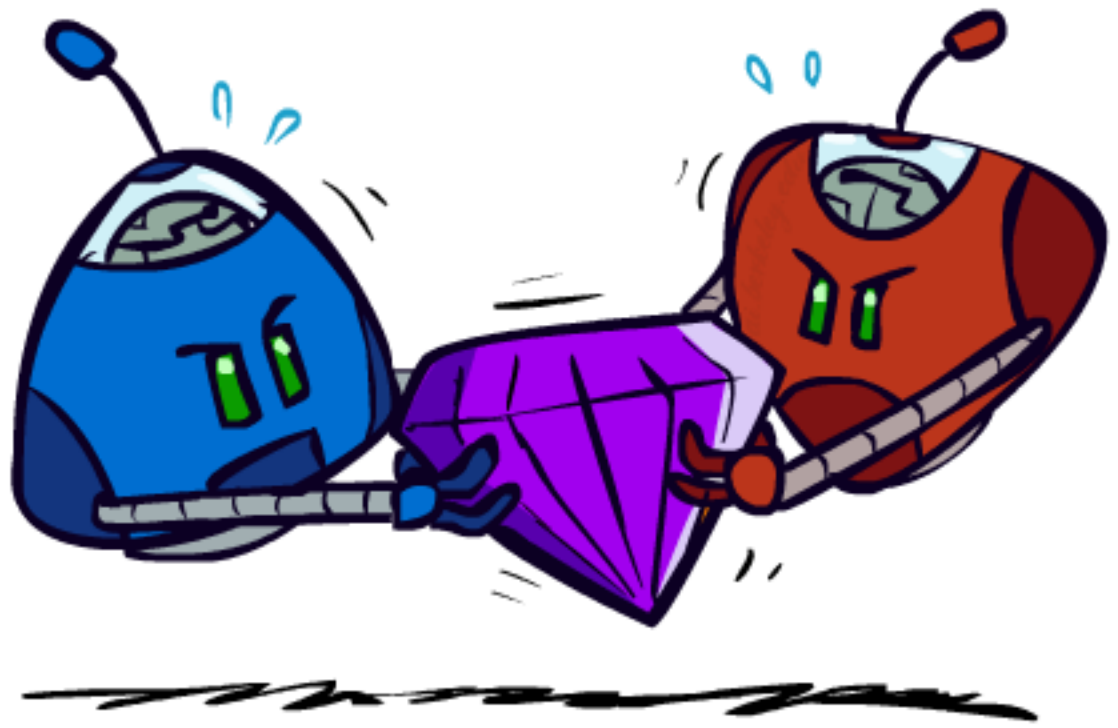
Tipos de jogos

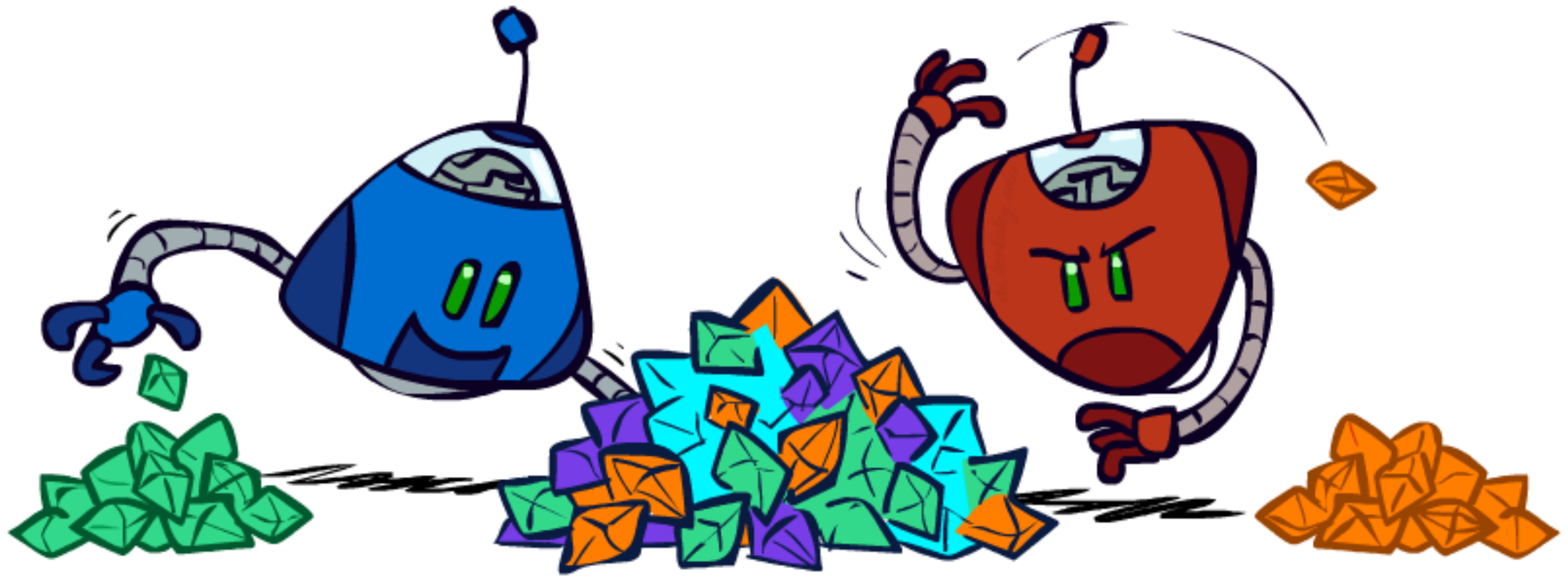
Em Teoria de Jogos, esses problemas são classificados de acordo com as seguintes propriedades:

- Determinístico ou estocástico?
- Um, dois, ou mais jogadores?
- Soma zero?
- Informação perfeita (todas as jogadas são conhecidas por todos os jogadores)?



Jogos de soma zero de dois jogadores





Jogos de soma zero

- Agentes têm utilidades opostas (a soma das utilidades é zero)
- Adversarial, puramente competitivo

Jogos de soma diferente de zero

- Agentes têm utilidades independentes
- Cooperação, indiferença e competição

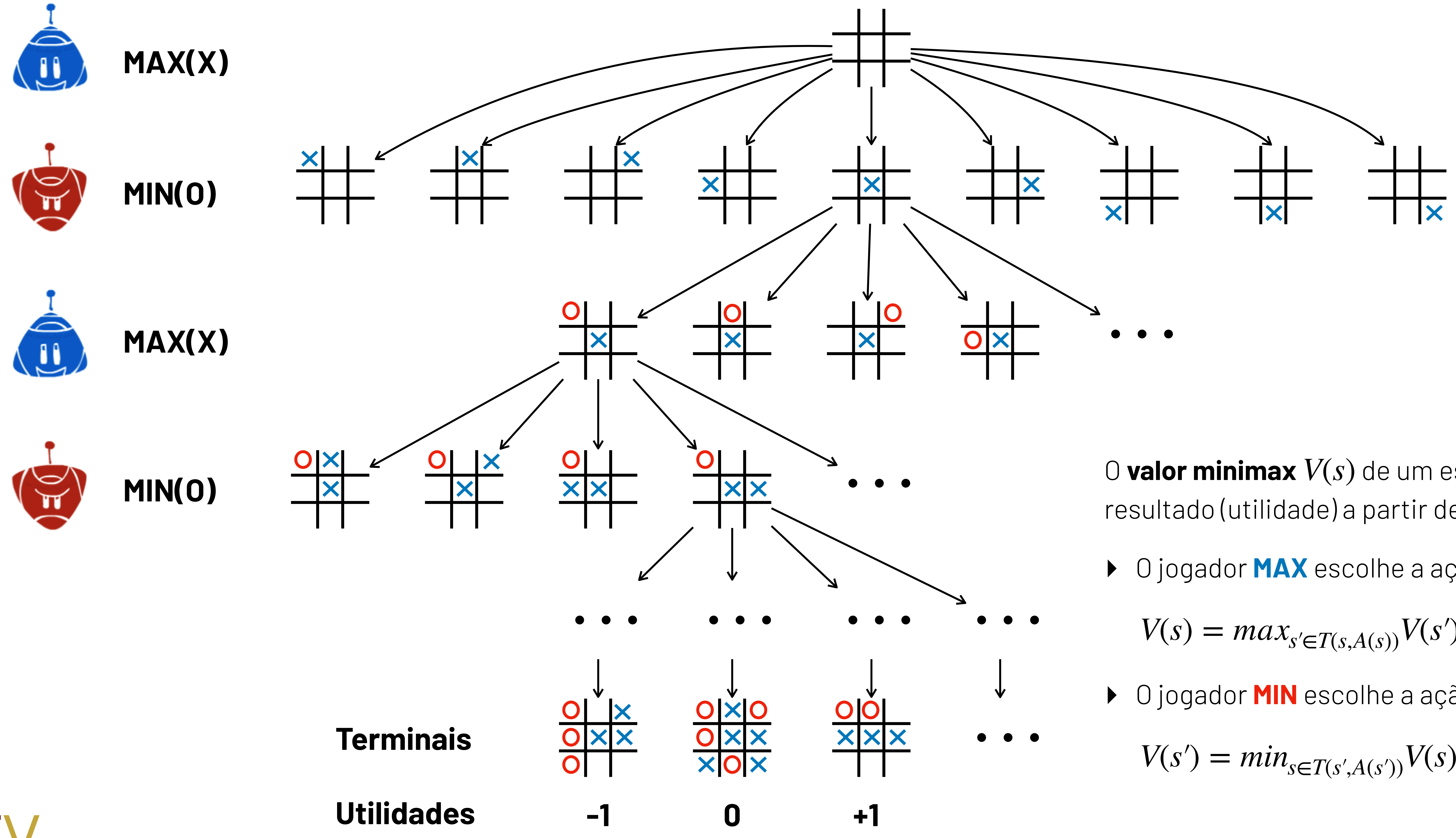
Jogos como problemas de busca

Um jogo de soma zero pode ser definido por:

- Conjunto de estados S, chamado de **espaço de estados**
- **Estado inicial** $s_0 \in S$
- **Teste de jogador** $J(s)$: retorna o jogador p que tem a vez no estado s
- **Função de ações** $A(s)$: retorna o conjunto finito de ações possíveis em s
- **Modelo de transição** $T(s, a)$: retorna o novo estado s' resultado da aplicação da ação a no estado s
- **Teste de fim** $E(s)$**:** retorna VERDADEIRO quando o estado s é terminal e FALSE caso contrário
- **Função de utilidade (*payoff*)** $U(s, p)$**:** retorna o valor de um estado terminal s para o jogador p

UFV

Árvore de busca competitiva



O **valor minimax** $V(s)$ de um estado s é o melhor resultado (utilidade) a partir de s:

- O jogador **MAX** escolhe a ação de maior valor:

$$V(s) = max_{s' \in T(s,A(s))} V(s')$$

- O jogador **MIN** escolhe a ação de menor valor:

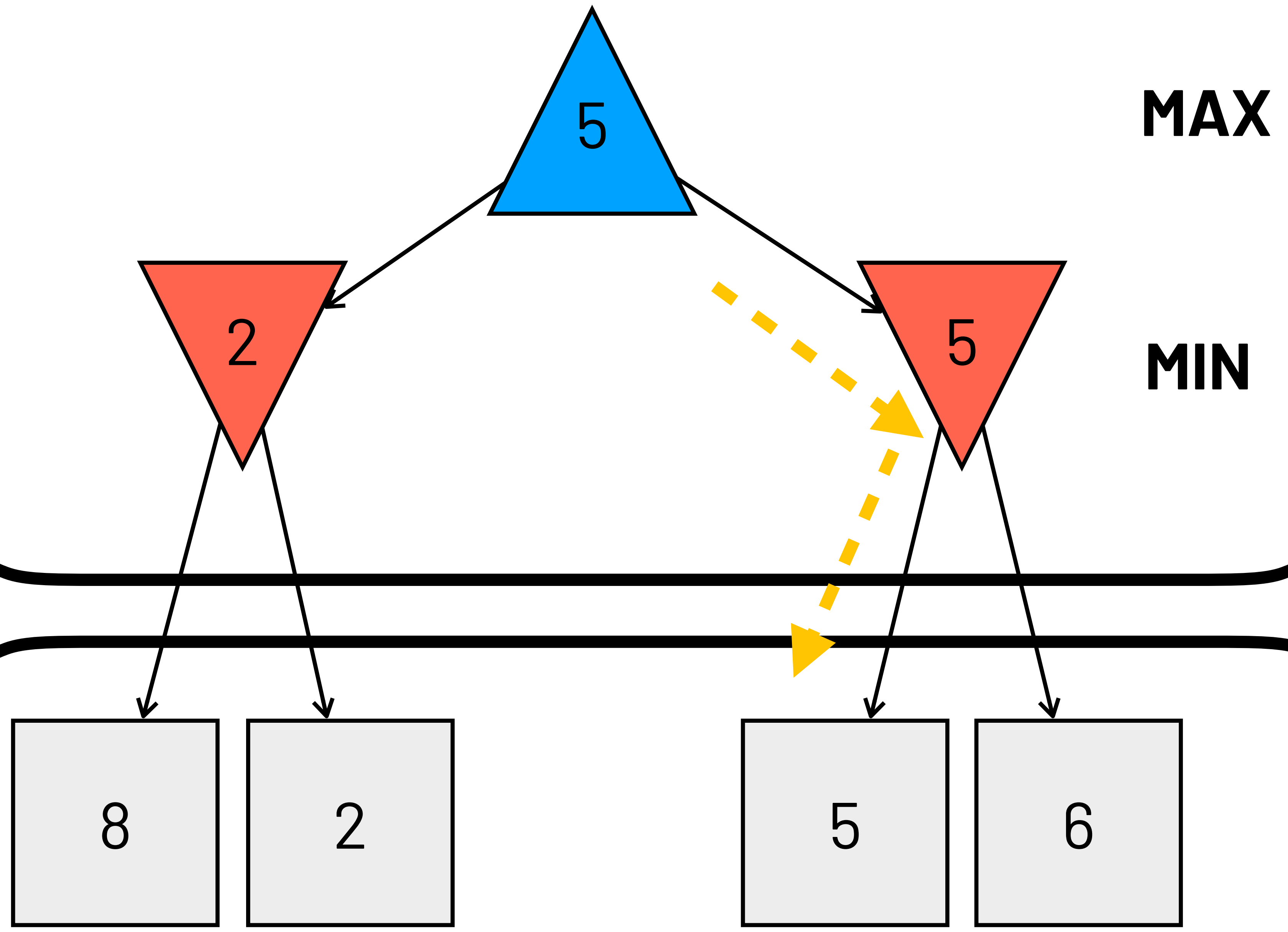
$$V(s') = min_{s \in T(s',A(s'))} V(s)$$

Algoritmo minimax: ideia geral

O algoritmo **minimax** produz uma estratégia ótima para **jogos de soma zero**:

- Jogo da velha, xadrez, damas
- Os jogadores alternam suas jogos em turnos
  - Um jogador maximiza o resultado
  - O outro minimiza o resultado
- O algoritmo calcula o valor minimax para cada estado: o melhor utilidade alcançável contra um **oponente racional**

O **valor minimax** $V(s)$ de um estado s é calculado recursivamente pelo algoritmo **minimax**:

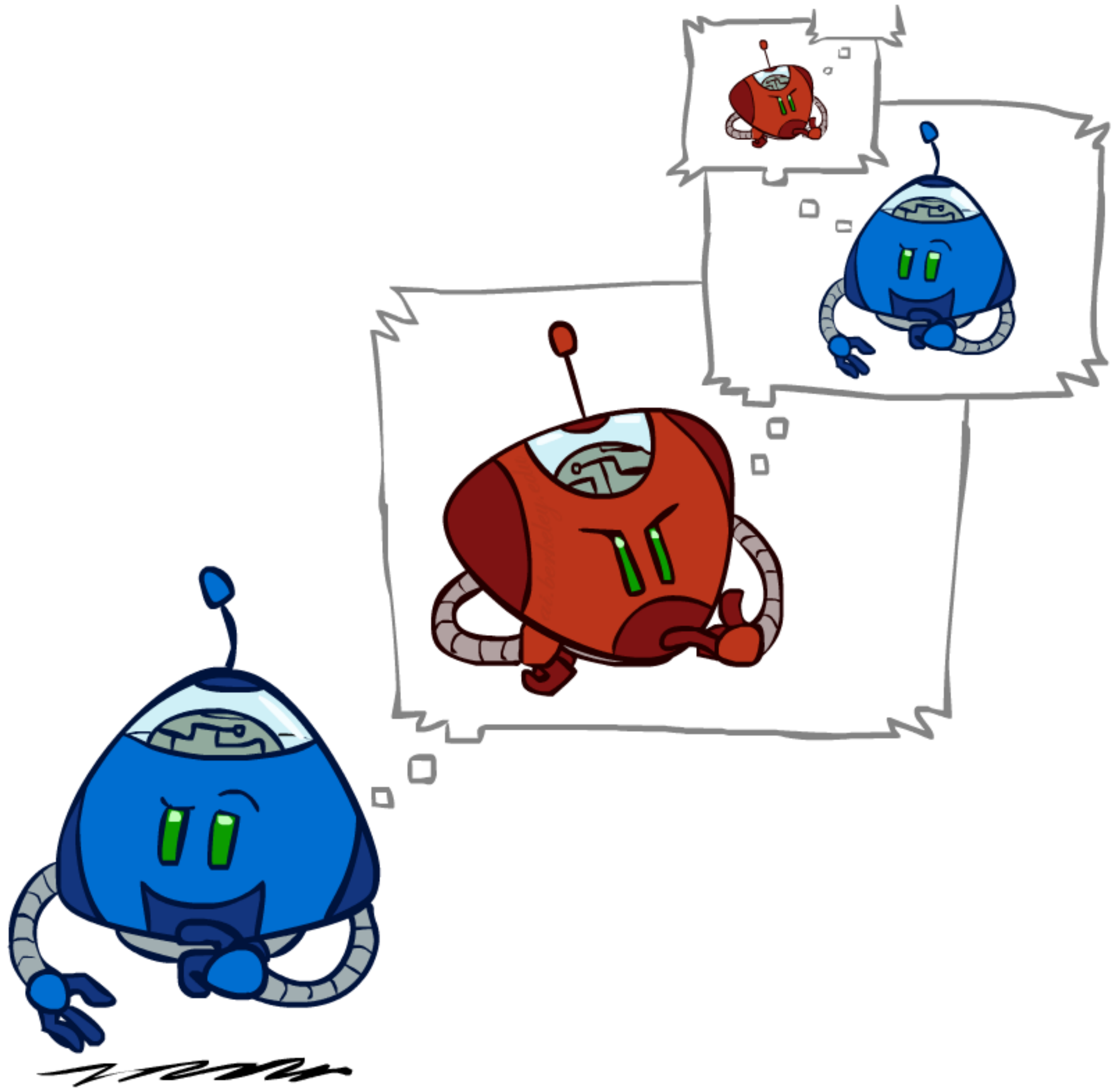


O **valor** $V(s)$ de um estado terminal é s é dado pela função de utilidade do problema

Algoritmo minimax

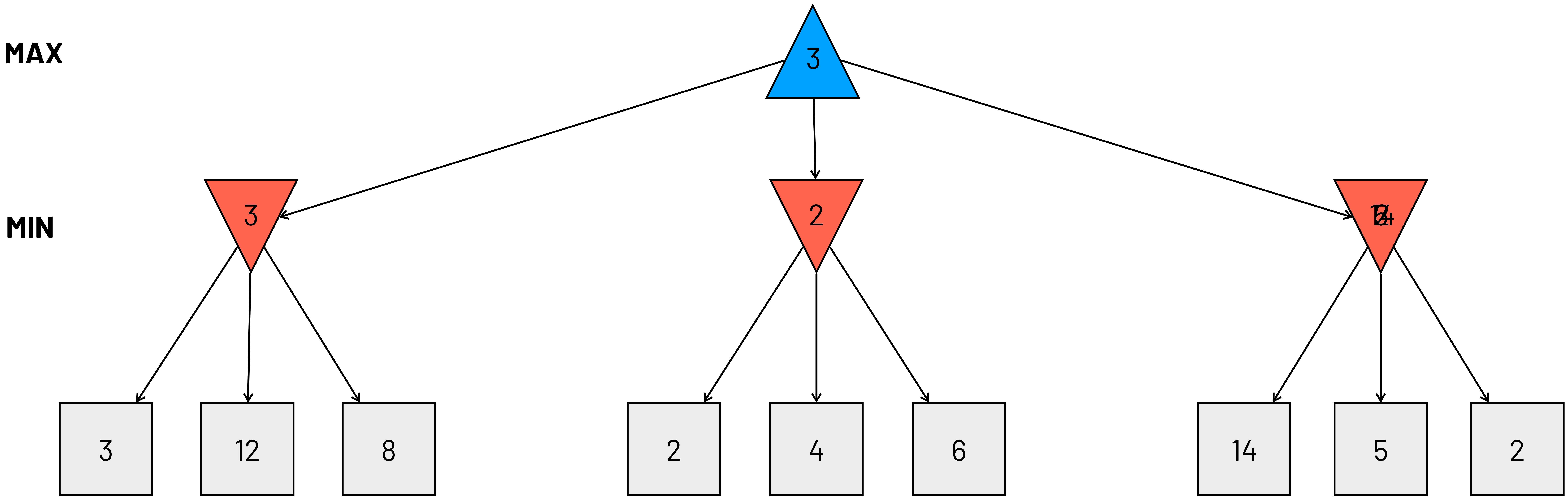
```
def valor-max(s, A, E, U):
if E(s) == True:
    return U(s, ‘max')
v = -∞
for filho in A(s):
    v = max(v, valor-min(filho, A, E, U))
return v

def valor-min(s, A, E, U):
if E(s) == True:
    return U(s, ‘min’)
v = +∞
for filho in A(s):
    v = min(v, valor-max(filho, A, E, U))
return v
```



UFV

Exemplo 2: execução do minimax



UFV

Propriedades do minimax

- Complexidade de tempo

  O mesmo que DFS: $O(b^m)$

- Complexidade de espaço

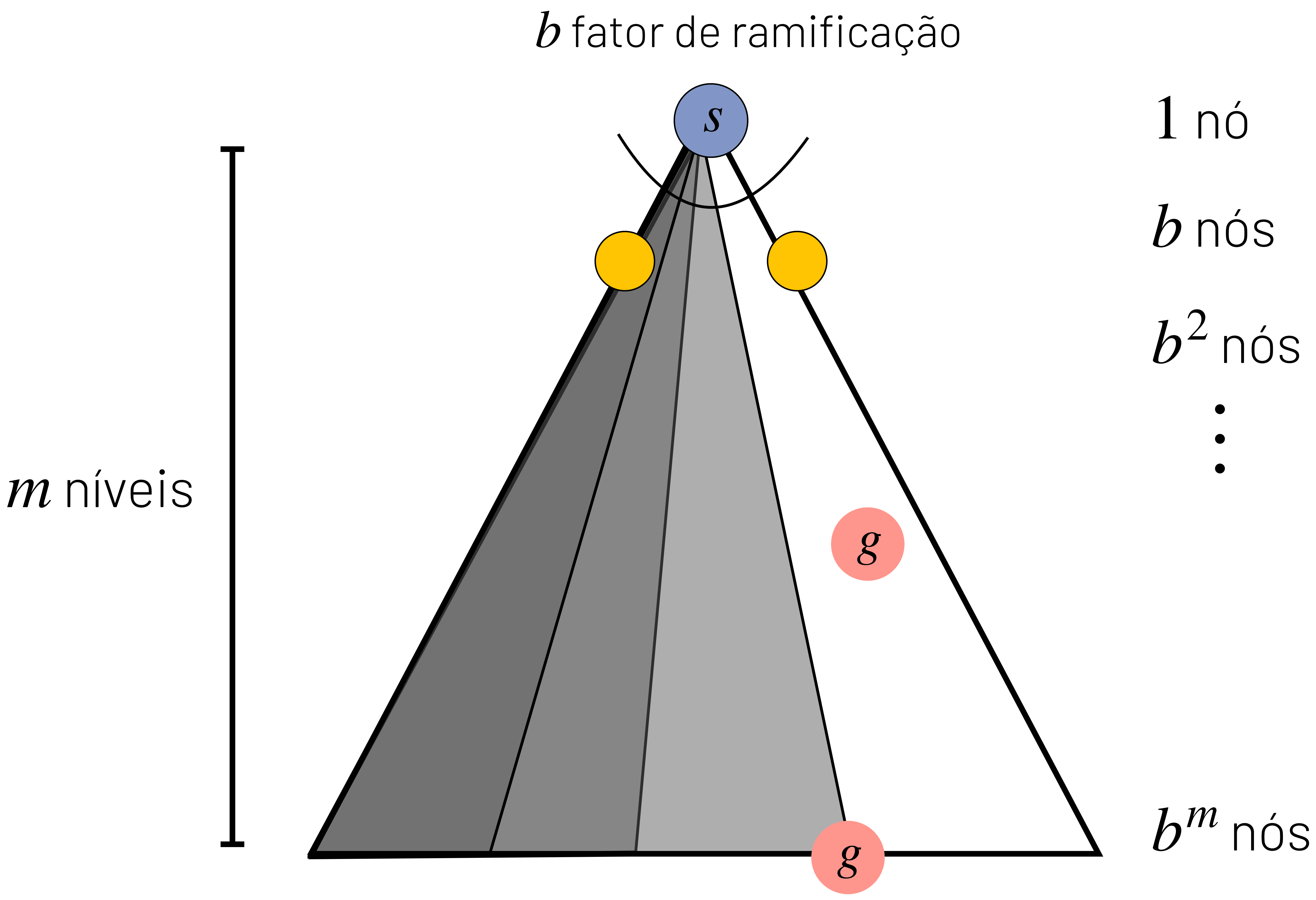
  O mesmo que DFS: $O(bm)$

- Exemplo

  Xadrez – $b \approx 35$, $m \approx 100$

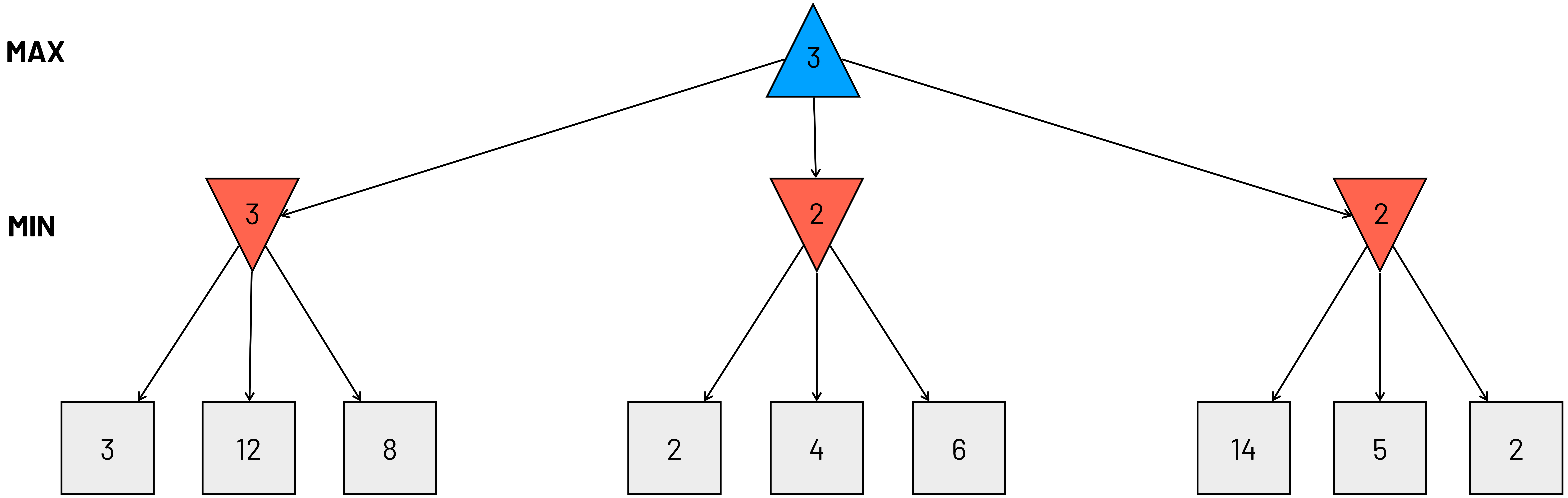
  Encontrar a solução exata é infactível!

  Precisamos mesmo explorar a árvore toda? 🤔



Exemplo 2: execução do minimax

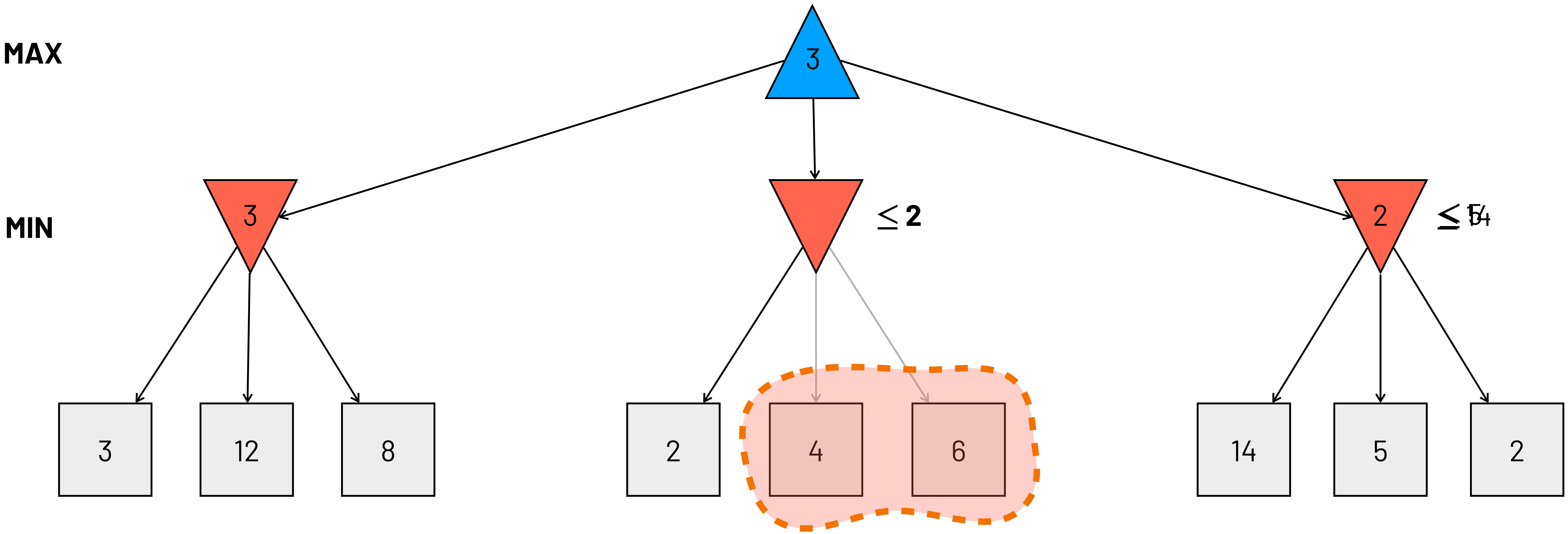
Considerando o exemplo anterior, você pode consegue pensar em uma forma de podar expansões de estados?



UFV

Exemplo 2: execução do minimax

Considerando o exemplo anterior, você pode consegue pensar em uma forma de podar expansões de estados?

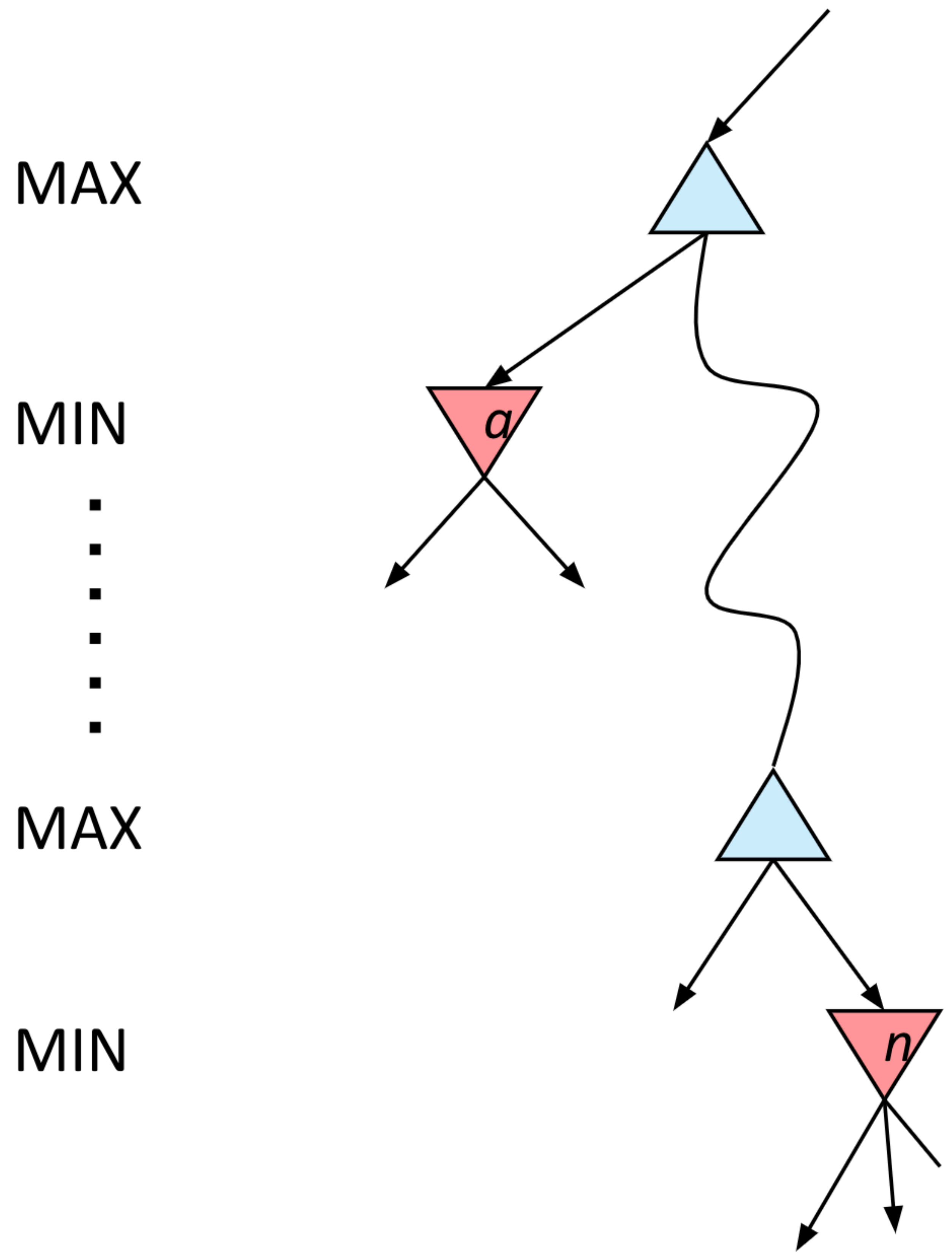


Podemos podar esses estados pois o MIN nunca escolheria um valor ≤ 2 e o MAX já conhece um caminho > 2.

UFV

Poda alpha-beta: ideia geral

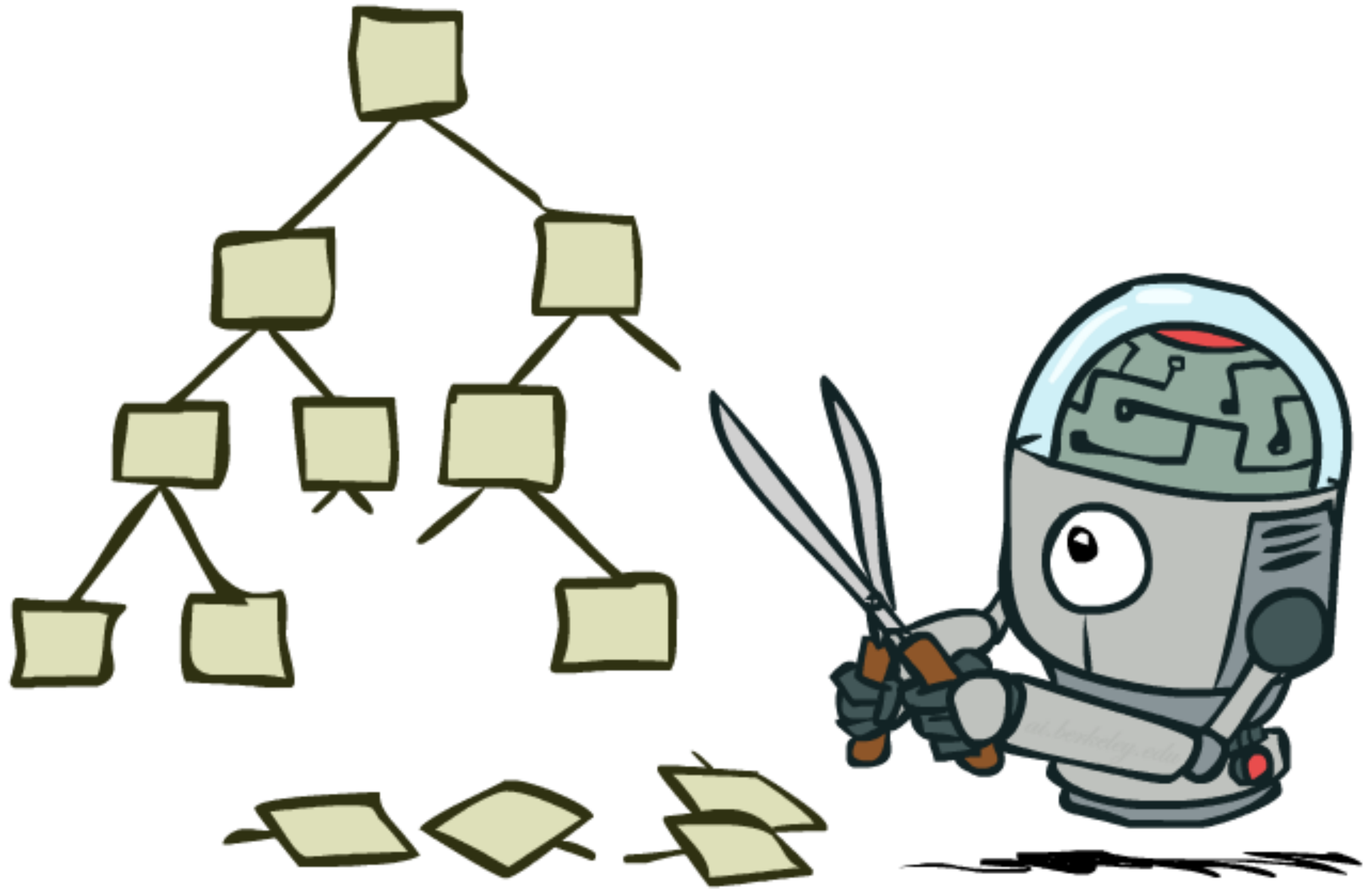
- Ideia geral para o jogador **MIN**:
  - Na função **valor-min(n)**, nós iteramos nos filhos de n
    - A estimativa do valor de n sempre diminui
    - Qual jogador quer saber o valor de n? **MAX**!
  - Seja a o melhor valor que **MAX** pode alcançar até agora no caminho para a raiz
  - Se a estimativa de n ficar pior que a, **MAX** irá evitar n, então paramos de considerar os outros filhos de n
- A versão do **MAX** é simétrica.



Poda alpha-beta

```
def valor-max(s, alpha, beta, A, E, U):
if E(s) == True:
    return U(s, 'max')
v = -∞
for filho in A(s):
    v = max(v, valor-min(filho, alpha, beta, A, E, U))
    if v >= beta return v
    alpha = max(alpha, v)
return v

def valor-min(s, alpha, beta, A, E, U):
if E(s) == True:
    return U(s, 'min')
v = +∞
for filho in A(s):
    v = min(v, valor-max(filho, alpha, beta, A, E, U))
    if v <= alpha return v
    beta = min(beta, v)
return v
```

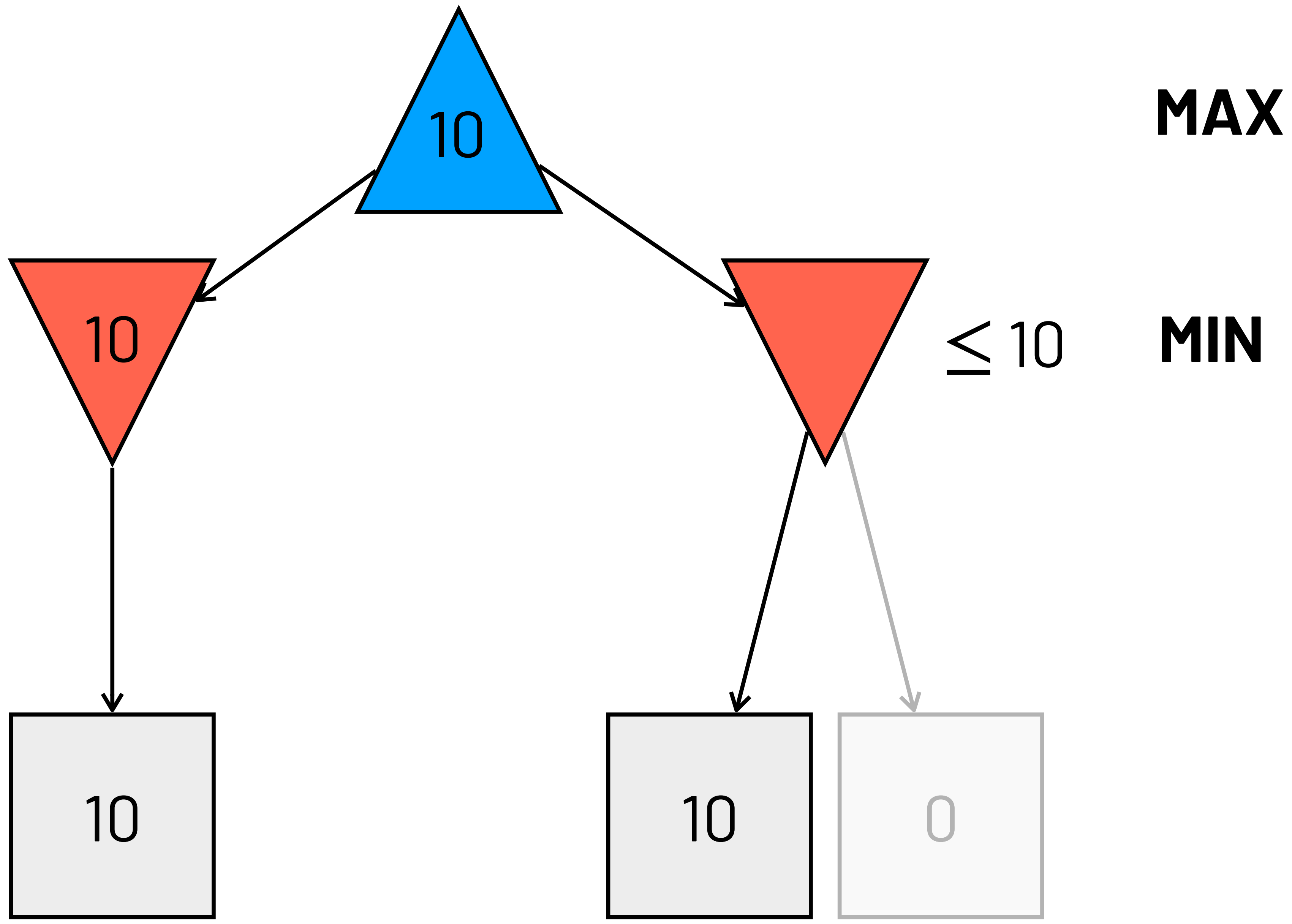


alpha: melhor opção do **MAX** no caminho para a raiz

beta: melhor opção do **MIN** no caminho para a raiz

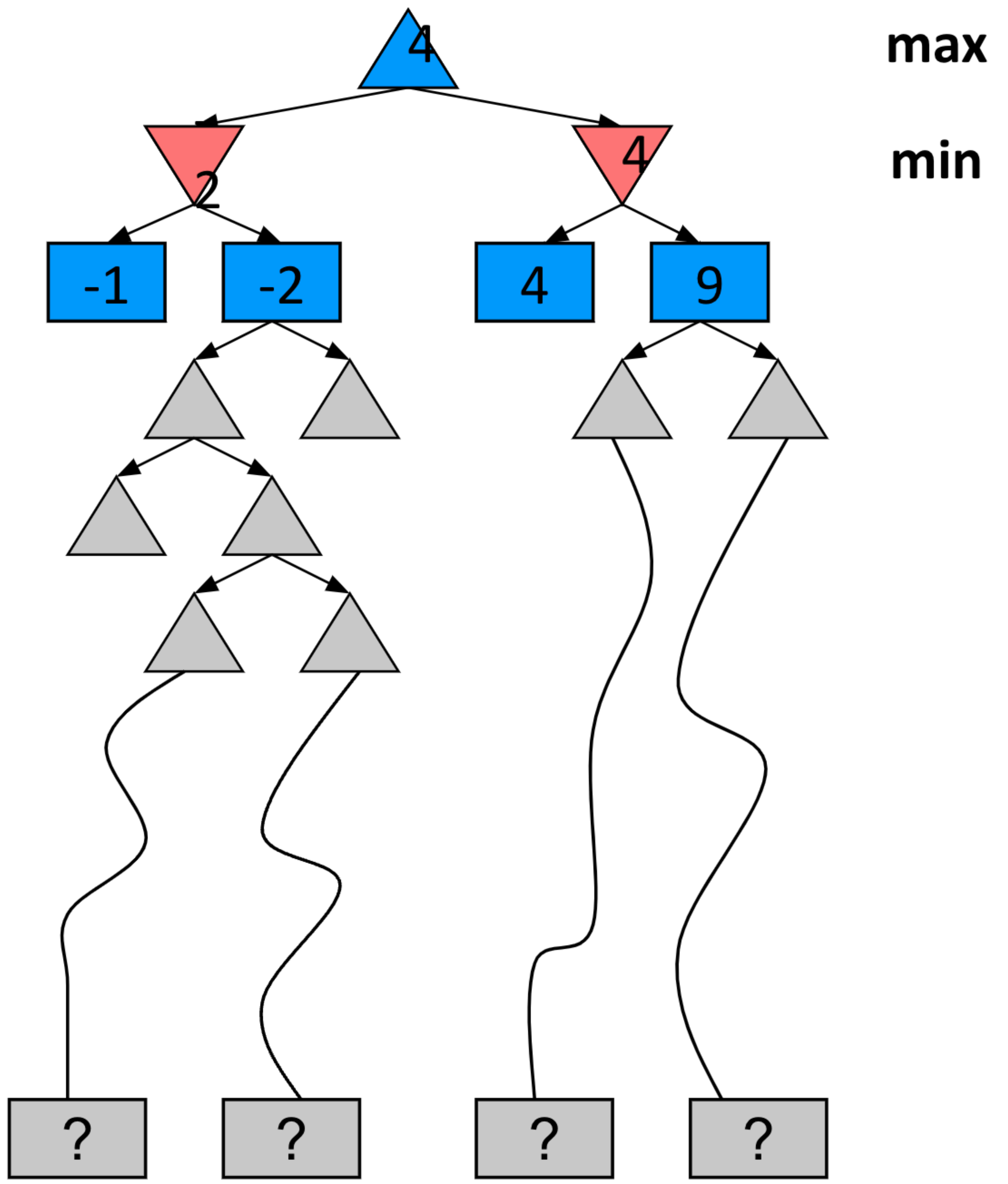
Propriedades da poda alpha-beta

- A poda alpha-beta **não altera** o valor minimax calculado para a raiz!
- Os valores de nós intermediários podem estar errados:
  - Os filhos da raiz podem ter um valor errado.
  - É importante armazenar a ação associada ao melhor valor até agora (a primeira)
- A ordem de exploração dos filhos aumenta a eficácia de poda:
  - **Explorar boas opções primeiro!**
  - Com ordem perfeita:
    - Complexidade de tempo reduz para $O(b^{\frac{m}{2}})$
    - Duplica a profundidade que pode ser considerada
    - Essa melhora ainda não é suficiente para problemas do tamanho do xadrez



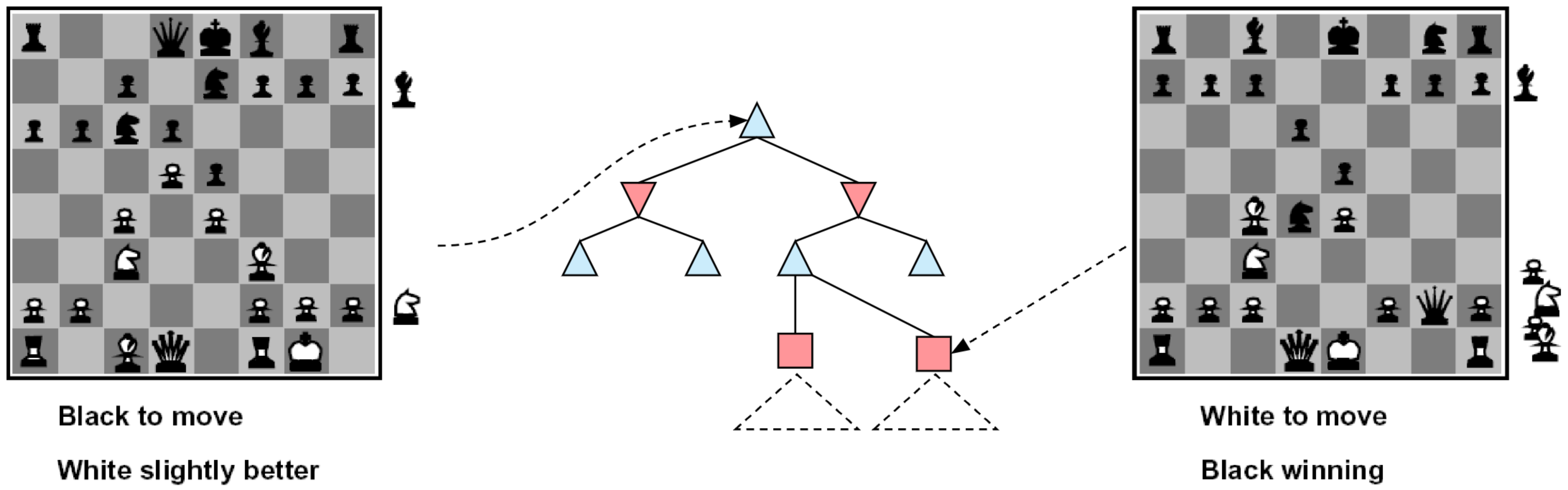
Limite de recursos

- **Problema**: em jogos reais, não conseguimos executar a busca até os estados terminais
- **Solução**: Busca com profundidade limitada
  - Buscar até uma dada profundidade d
  - Avaliar estados não-terminais com uma **função de avaliação (heurística)**
- Exemplo
  - Assuma que temos 100s e podemos explorar 10k nós/s
  - Podemos explorar 1M nós por jogada
  - Alpha-beta alcança a profundidade 8 em bons programas de xadrez
- **A garantia de jogada ótima é perdida!**



Funções de avaliação

- Funções de avaliação avaliam estados não-terminais na busca com profundidade limitada



- Função ideal: retorna o **valor minimax** exato naquela posição não-terminal
- Na prática: geralmente uma soma ponderada de características:

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

Por exemplo: $f_1(s)$ – número de rainhas brancos - número de rainhas pretas, etc...

Próxima aula

A7: Busca competitiva II

Jogos estocásticos e parcialmente observáveis