

# Jogos



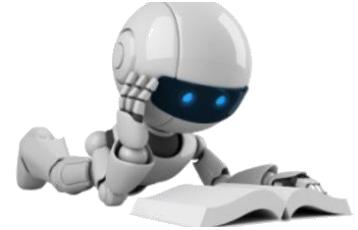
“como planear num mundo onde outros agentes estão a planear contra nós”

# Sumário

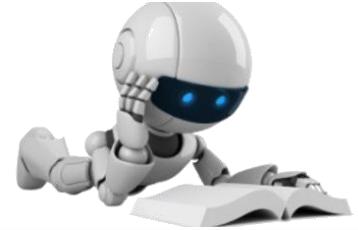


- Jogos: Conceitos Básicos
- Decisões óptimas em jogos
  - Estratégias óptimas e o Minimax
  - Estratégias óptimas com múltiplos jogadores
- Cortes  $\alpha$ - $\beta$
- Decisões imperfeitas em tempo real
- Jogos estocásticos
- Estado da arte em jogos

# Sumário



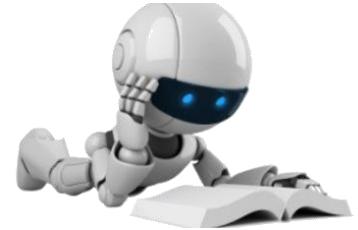
- Jogos: Conceitos Básicos
- Decisões óptimas em jogos
  - Estratégias óptimas e o Minimax
  - Estratégias óptimas com múltiplos jogadores
- Cortes  $\alpha$ - $\beta$
- Decisões imperfeitas em tempo real
- Jogos estocásticos
- Estado da arte em jogos



- Em ambientes multi-agente há que prever e perceber as acções dos outros agentes;
- As acções dos outros e a sua imprevisibilidade levam a restrições na decisão dos nossos agentes;
- Em ambientes competitivos onde os agentes têm objectivos que estão em conflito, a decisão pode ser baseada em procura adversarial: Jogos!

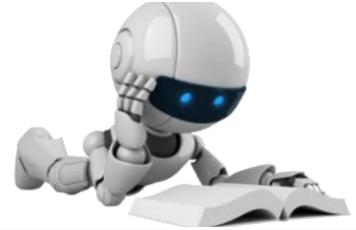


# Jogos vs. Problemas de Procura



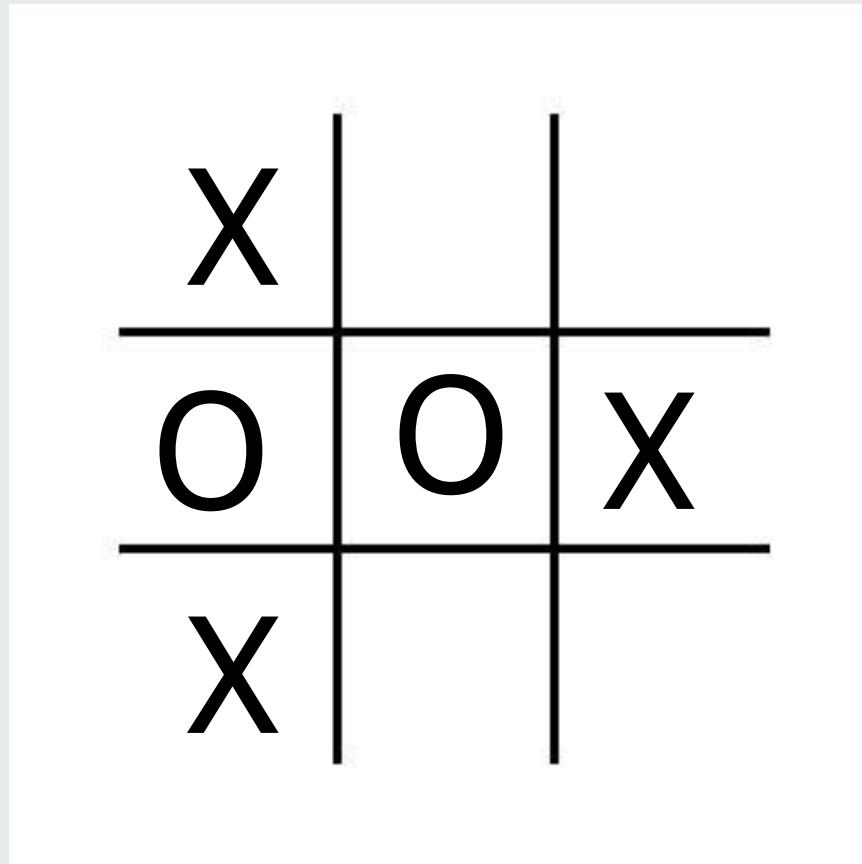
- Problemas de Procura Adversária (ou **Jogos**)
  - Ambientes competitivos onde os objectivos dos vários agentes do mundo estão em conflito
  - Tipicamente
    - 2 jogadores, agindo um de cada vez alternadamente
  - Jogos de soma zero (“zero-sum games”)
    - Pontuação com sinais opostos
    - O que é bom para um jogador (vitória=+1) é mau para o outro (derrota=-1)
  - Adversário “imprevisível”
    - necessidade de tomar em consideração todas os movimentos que podem ser tomados pelo adversário
  - Demasiado complexos
    - tipicamente não é encontrada a solução óptima mas antes uma aproximação

# Ingredientes



- Vamos considerar que:
  - Existem dois jogadores (para já),
  - Vamos chamar a um jogador o MAX e ao outro o MIN
  - Considera-se que o MAX joga primeiro
    - alternam turnos entre eles
  - No fim do jogo o vencedor ganha pontos e o adversário é penalizado (ou há um empate)

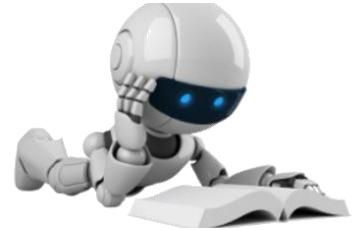
# Exemplo



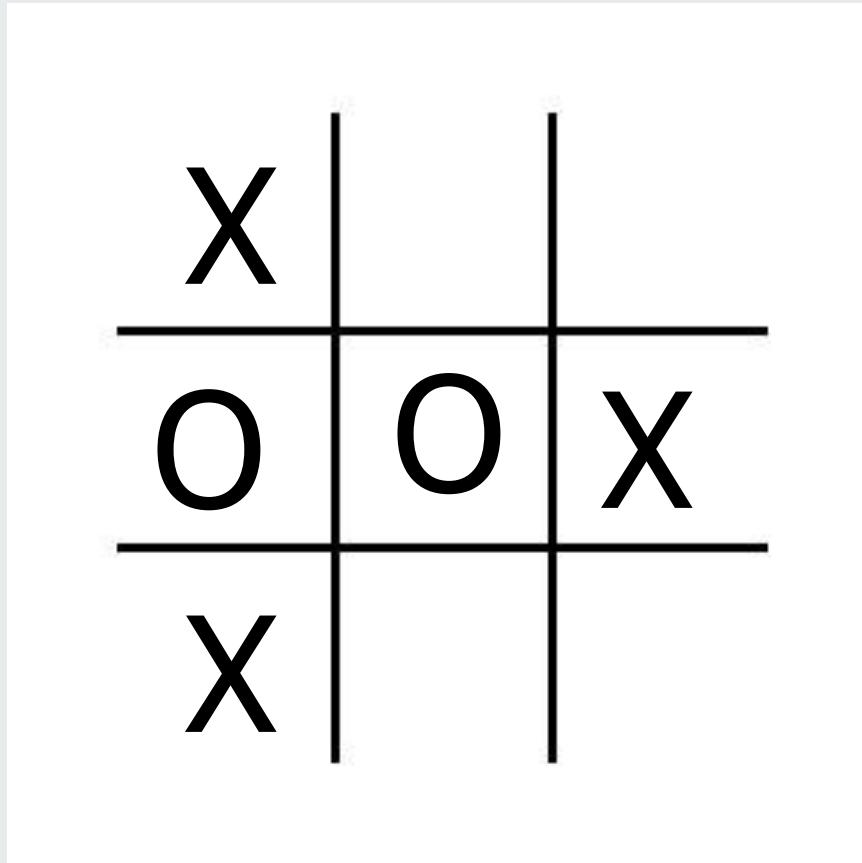


- Jogo pode ser visto como um problema de procura com:
  - Estado inicial
  - Jogador
    - função que dado um estado  $e$  retorna o jogador a jogar nesse estado
  - Acções
    - função que dado um estado  $e$  retorna o conjunto de jogadas válidas que podem ser feitas em  $e$
  - Resultado
    - função que dada um estado  $e$  e uma acção  $a$ , retorna o estado  $e'$  que resulta de executar  $a$  em  $e$
  - Teste-terminal
    - função que recebe um estado  $e$  e verifica se esse estado corresponde a um estado terminal (jogo terminado)
  - Utilidade
    - função que dado um estado terminal  $e$  e um jogador  $j$  retorna a utilidade final do estado para o jogador (valor numérico)

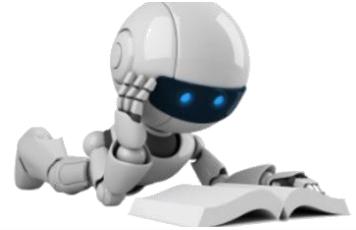
# Exemplo



Estado “e”:



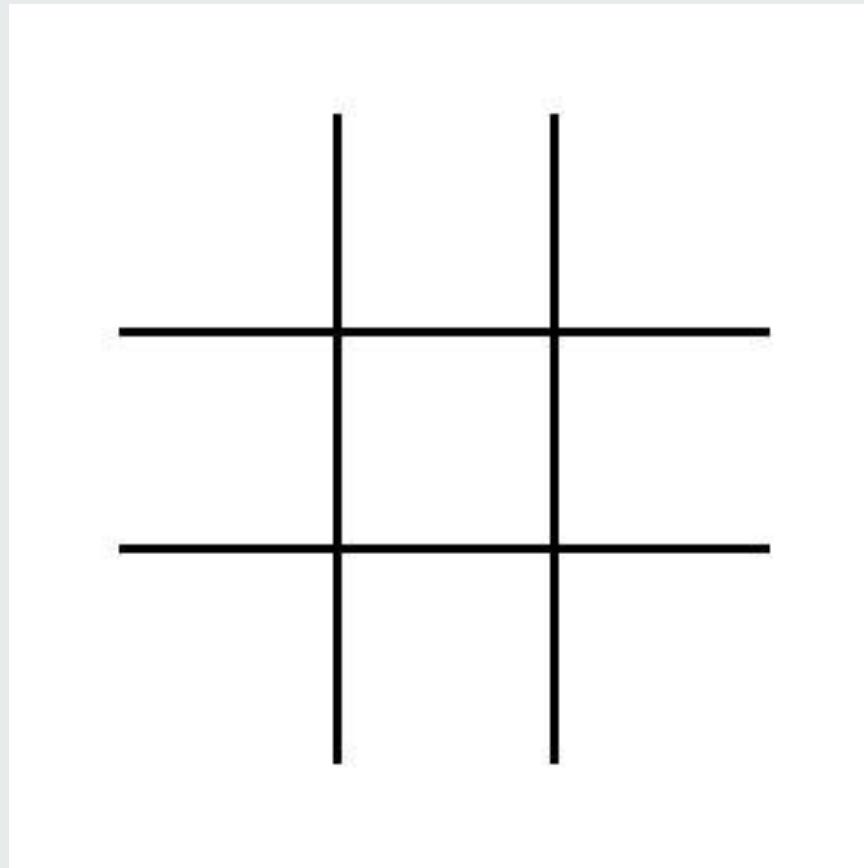
# Árvore do jogo

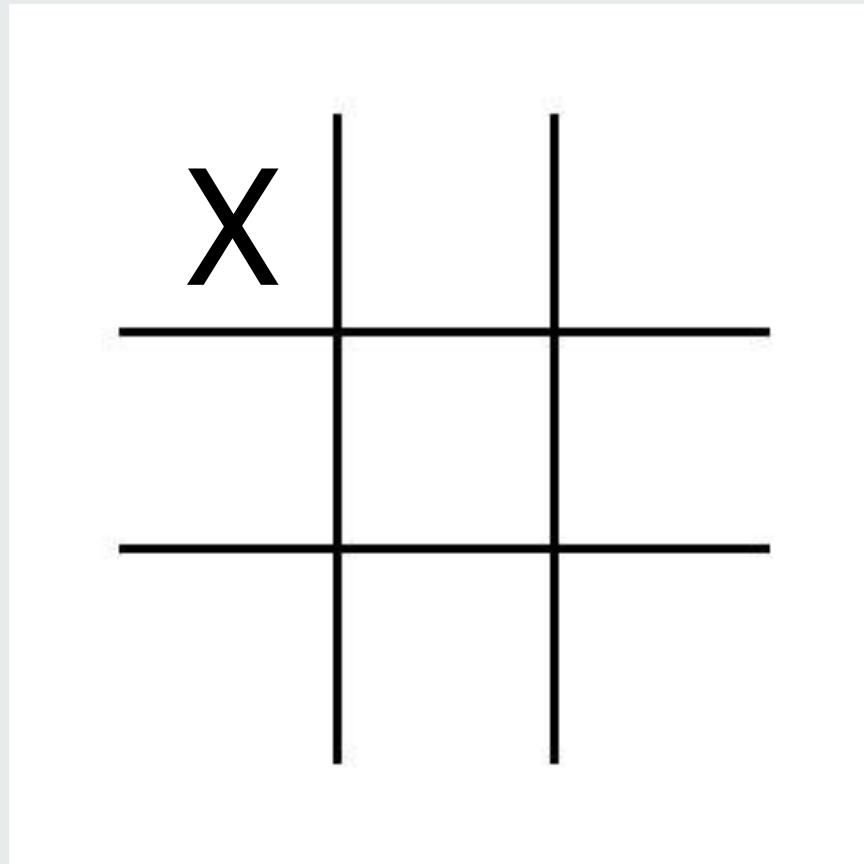


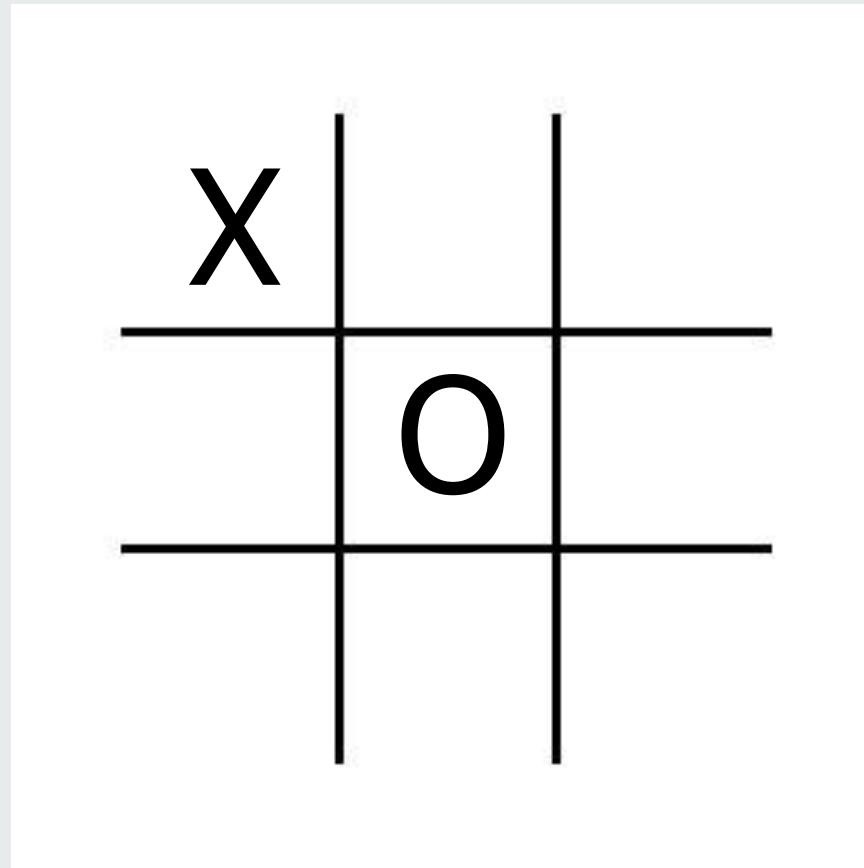
- O estado inicial e as funções Acções e Resultado definem a **árvore do jogo**.
  - Árvore onde os nós são estados do jogo
  - Ramos são jogadas

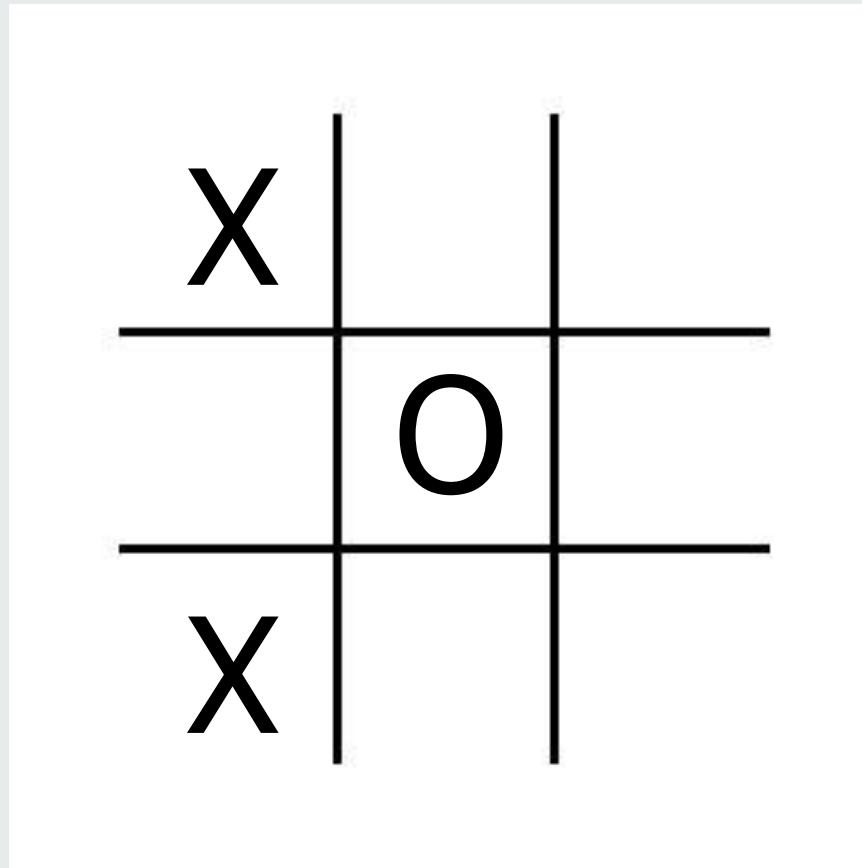


TÉCNICO  
LISBOA



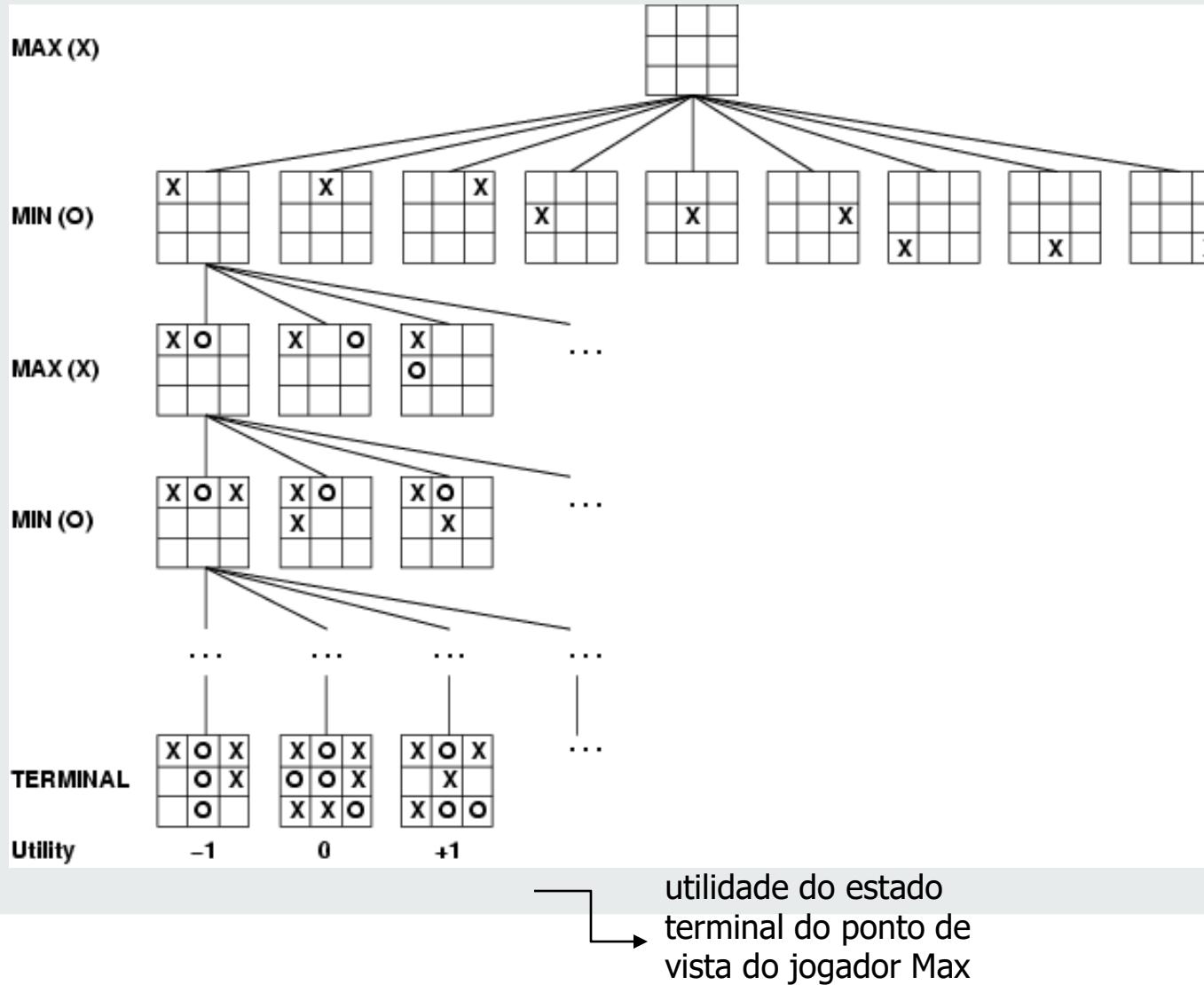




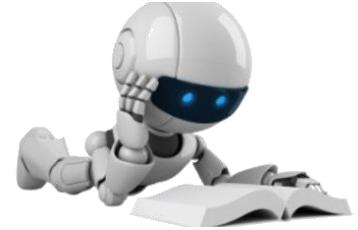




# Exemplo: Árvore para o jogo do galo

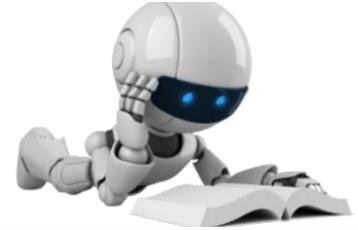


# Sumário

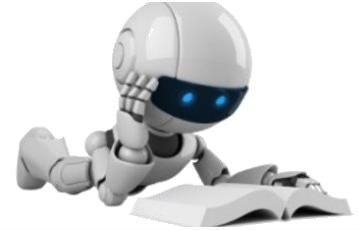


- Jogos: Conceitos Básicos
- Decisões óptimas em jogos
  - Estratégias óptimas e o Minimax
  - Estratégias óptimas com múltiplos jogadores
- Cortes  $\alpha$ - $\beta$
- Decisões imperfeitas em tempo real
- Jogos estocásticos
- Estado da arte em jogos

# Minimax



- Estratégia óptima para uma árvore de jogo
  - escolher jogada para o estado com o maior **valor minimax**
- Valor minimax ( $n$ )
  - **Utilidade** do estado  $n$  para o jogador **Max** assumindo que **ambos jogadores** vão **jogar optimalmente** a partir de  $n$  até ao fim do jogo
    - O jogador Max irá escolher acções de modo a maximizar a utilidade
    - O jogador Min irá escolher acções de modo a minimizar a utilidade



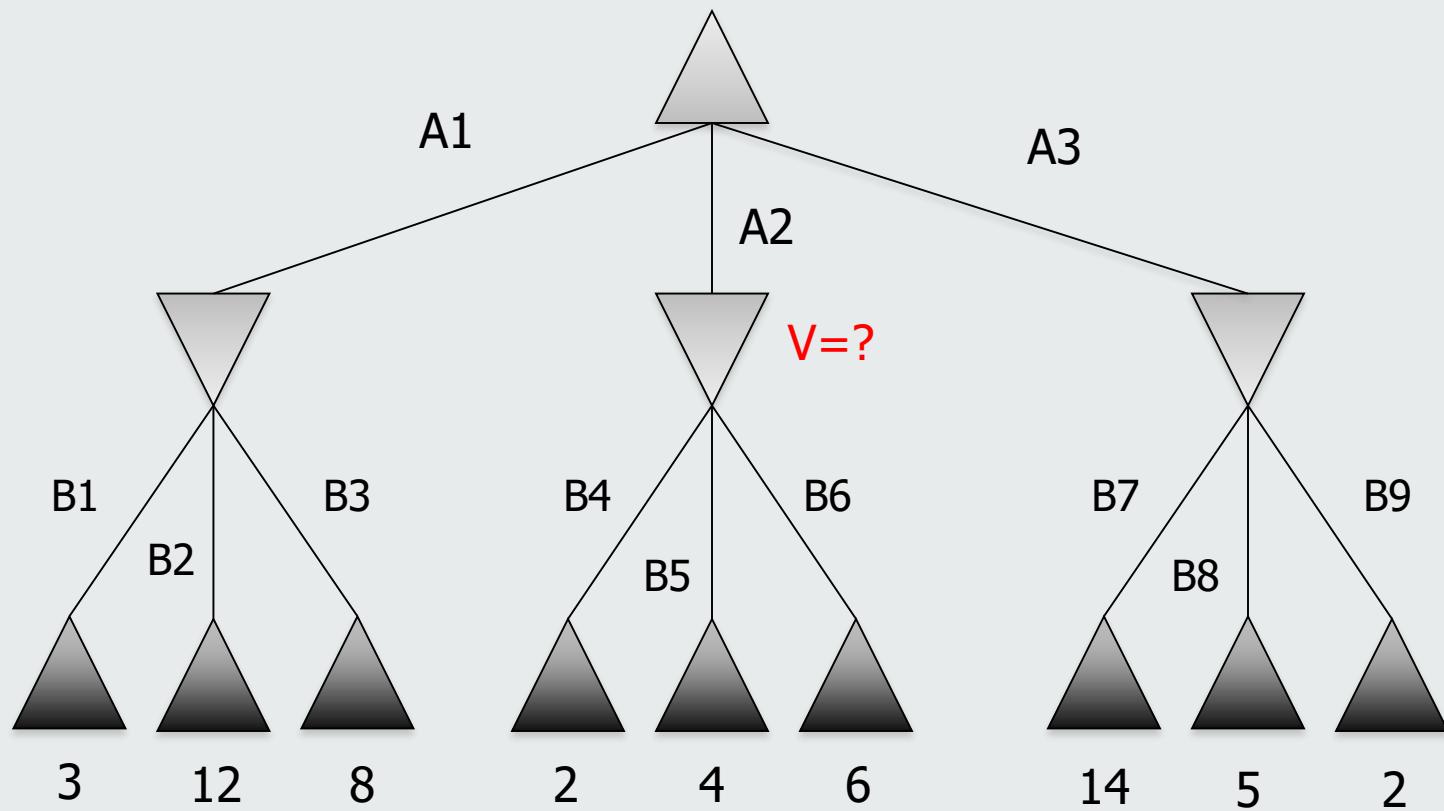
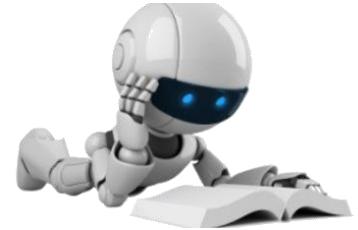
# Minimax

$$\text{Valor-minimax}(n) = \begin{cases} \text{utilidade}(n, \text{Max}) & \text{se } n \text{ é terminal} \\ \max_{s \in \text{sucessores}(n)} \text{Valor-minimax}(s) & \text{se } n \text{ é nó MAX} \\ \min_{s \in \text{sucessores}(n)} \text{Valor-minimax}(s) & \text{se } n \text{ é nó MIN} \end{cases}$$

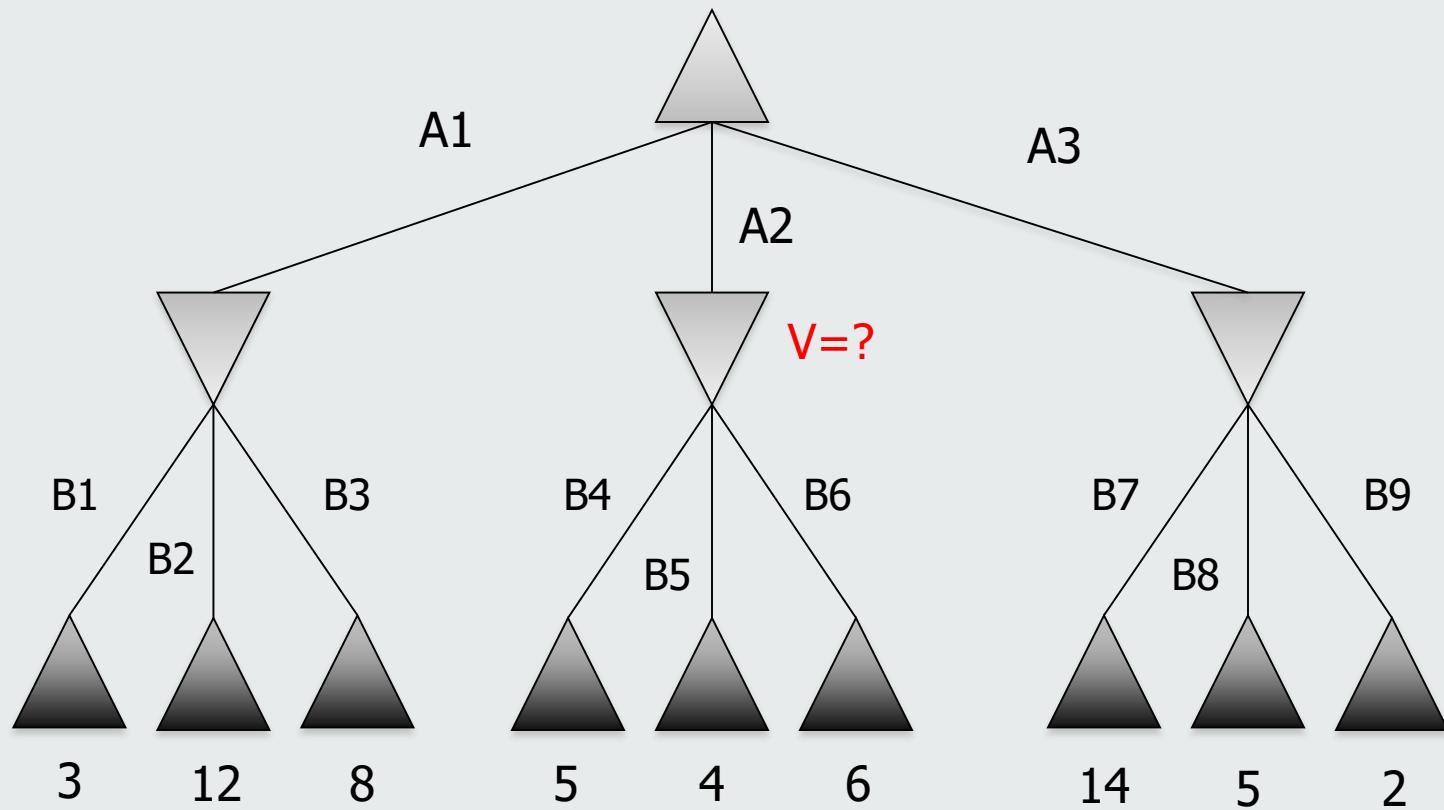
nó Max – nó onde jogador Max escolhe a jogada  
nó Min – nó onde jogador Min escolhe a jogada



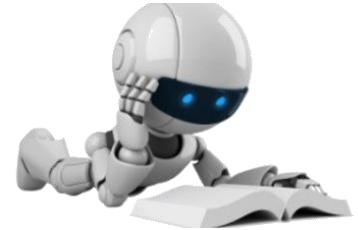
# Minimax: exemplo



# Minimax: exemplo



# Algoritmo Minimax

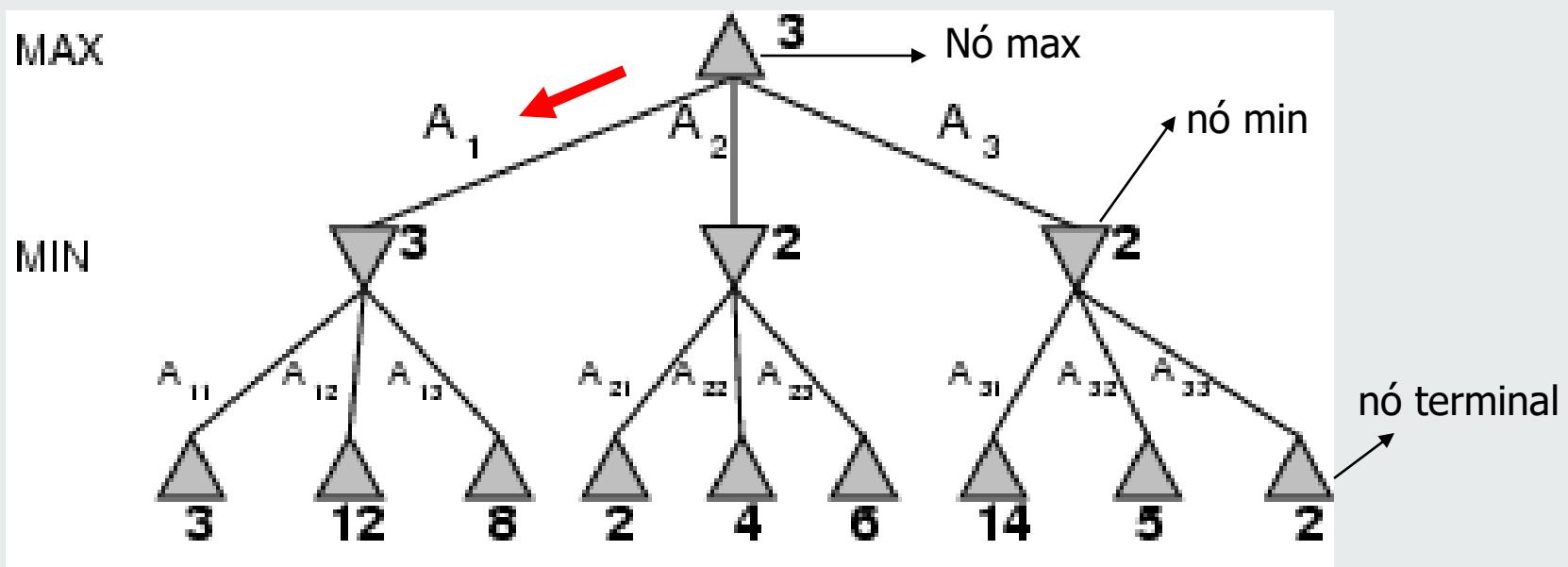


```
function Minimax-Decision(state) returns an action
    return arg maxa ∈ Actions(s) Min-Value(Result(state,a))
```

```
function Max-Value(state) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    v ← -∞
    for each a in Actions(state) do
        v ← Max(v, Min-Value(Result(state,a)))
    return v
```

```
function Min-Value(state) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    v ← +∞
    for each a in Actions(state) do
        v ← Min(v, Max-Value(Result(state,a)))
    return v
```

# Minimax: 2 jogadores

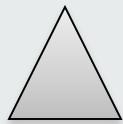


- Observações:
  - Formato dos nós em função do tipo de nó (MIN/MAX)
  - Valores dos estados terminais correspondem à função de utilidade para MAX (quanto vale cada um dos estados terminais)
  - Valores para os restantes estados obtidos a partir dos valores para os nós terminais através do cálculo do valor-minimax
  - Resultado do algoritmo: próxima jogada!



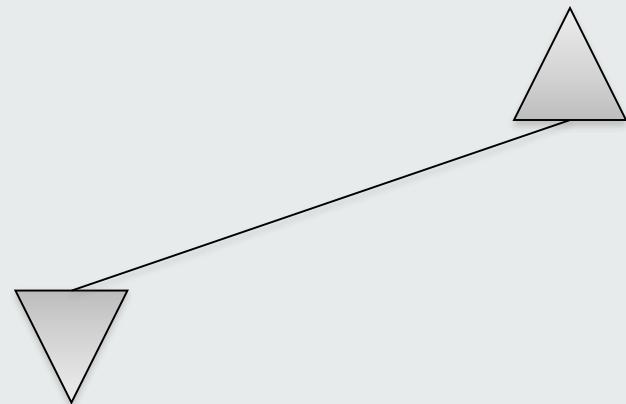
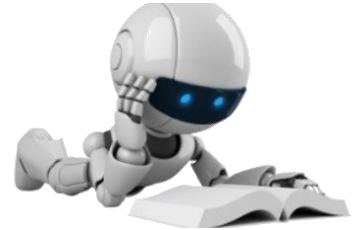
TÉCNICO  
LISBOA

# Minimax: exemplo



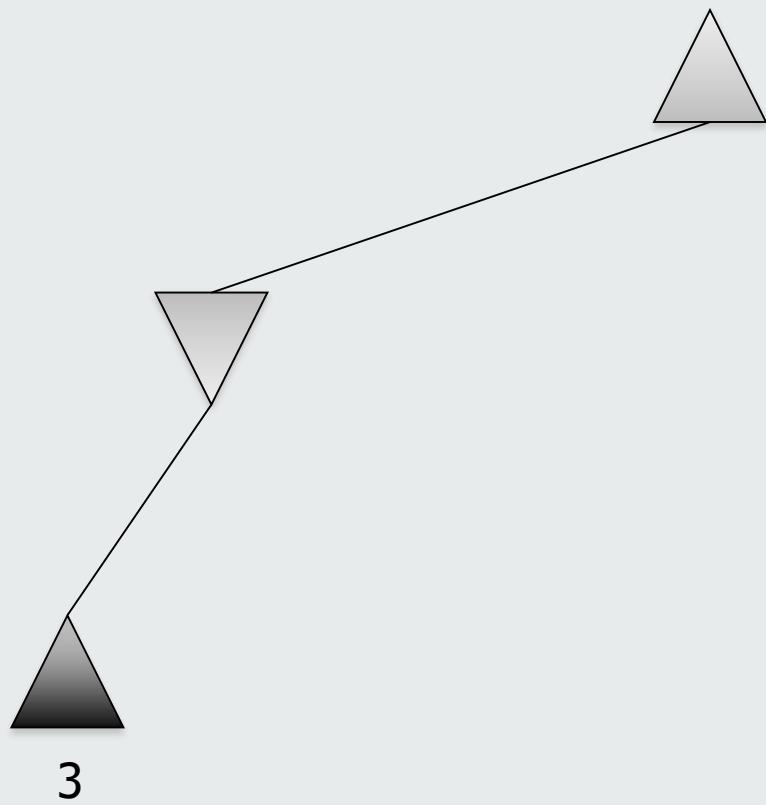
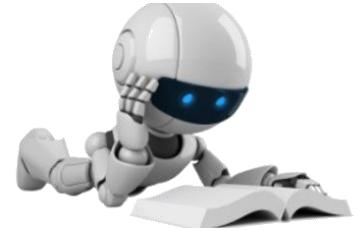


# Minimax: exemplo



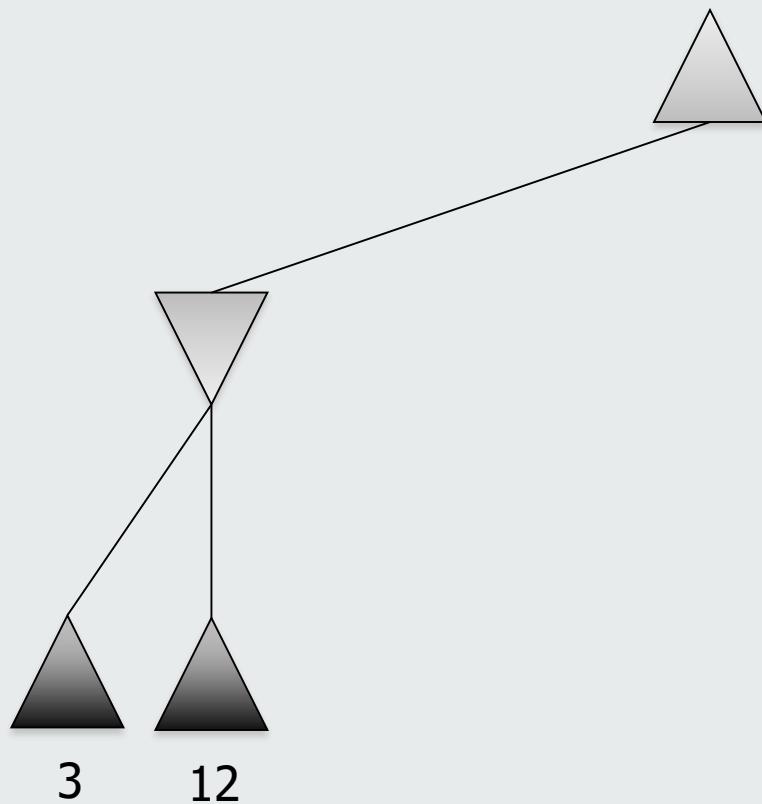


# Minimax: exemplo



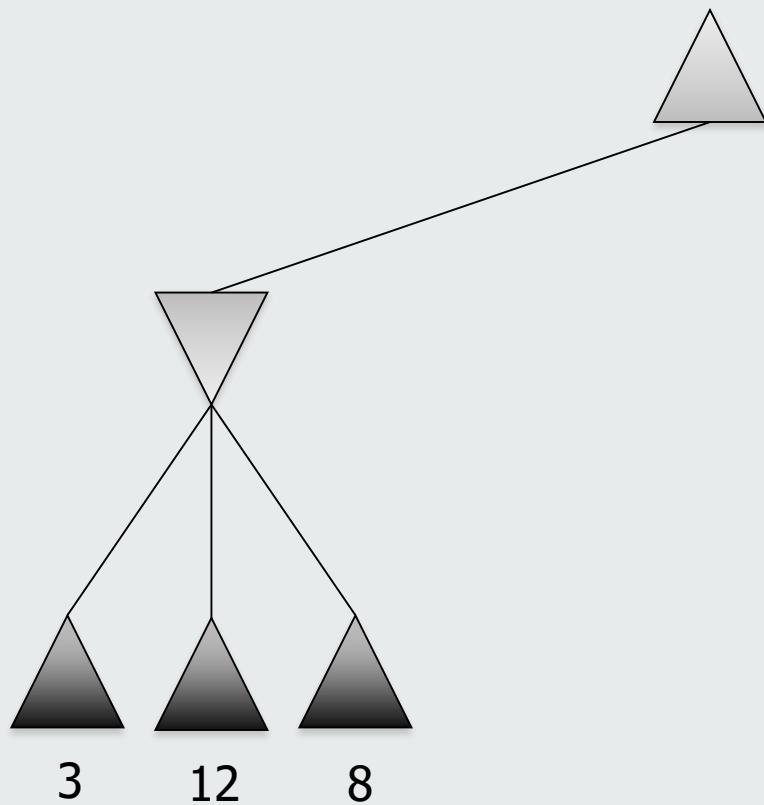
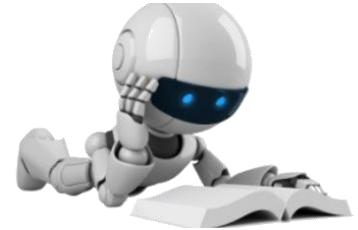


# Minimax: exemplo



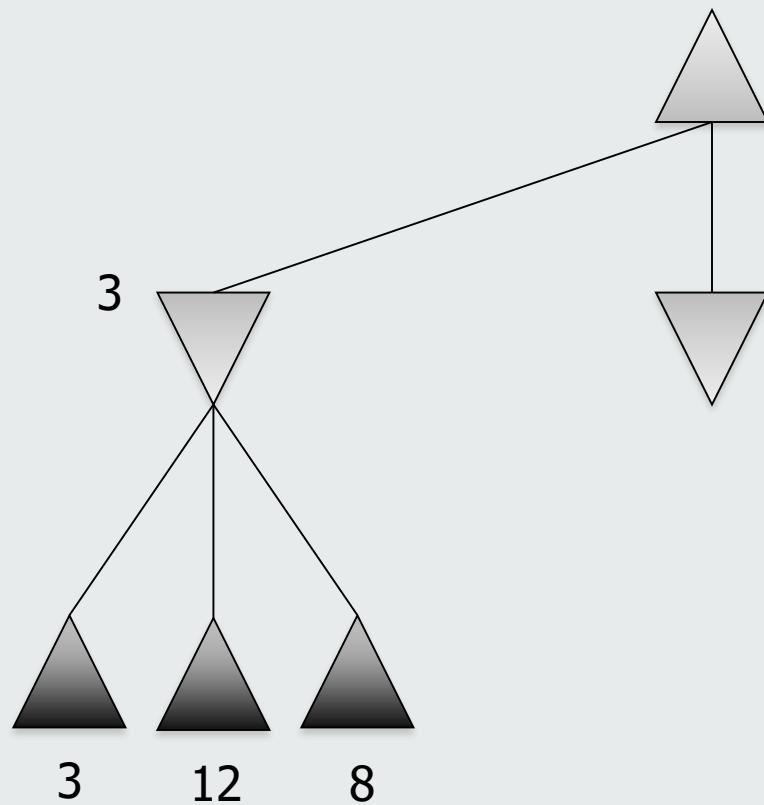
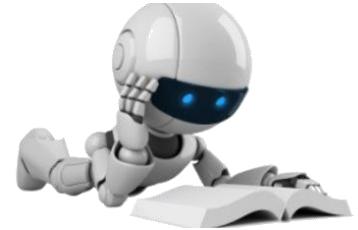


# Minimax: exemplo



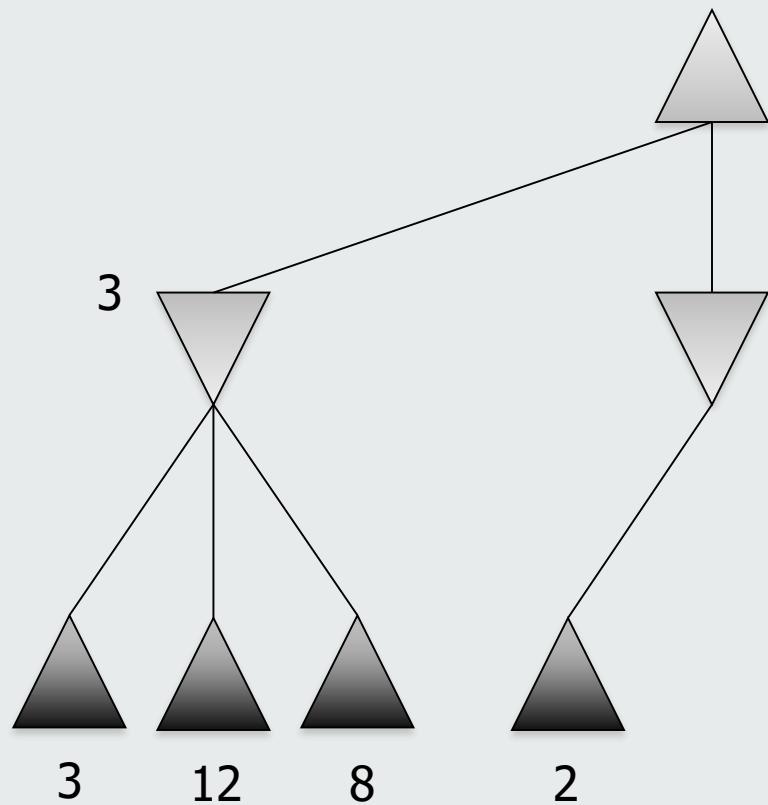
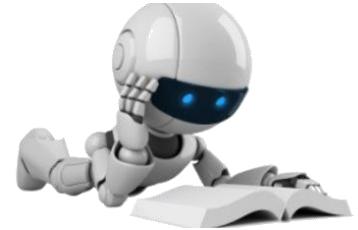


# Minimax: exemplo



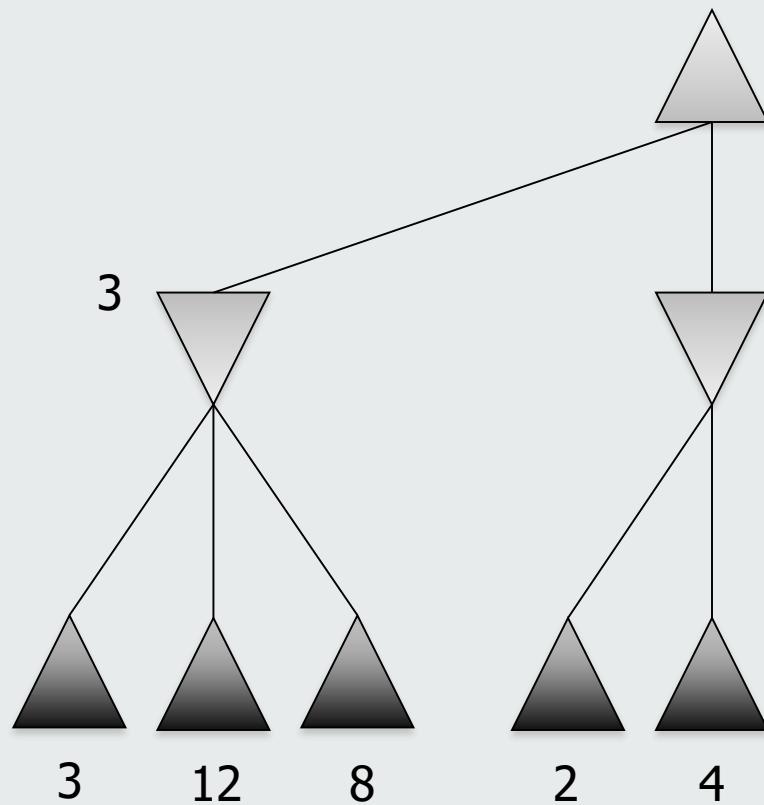


# Minimax: exemplo



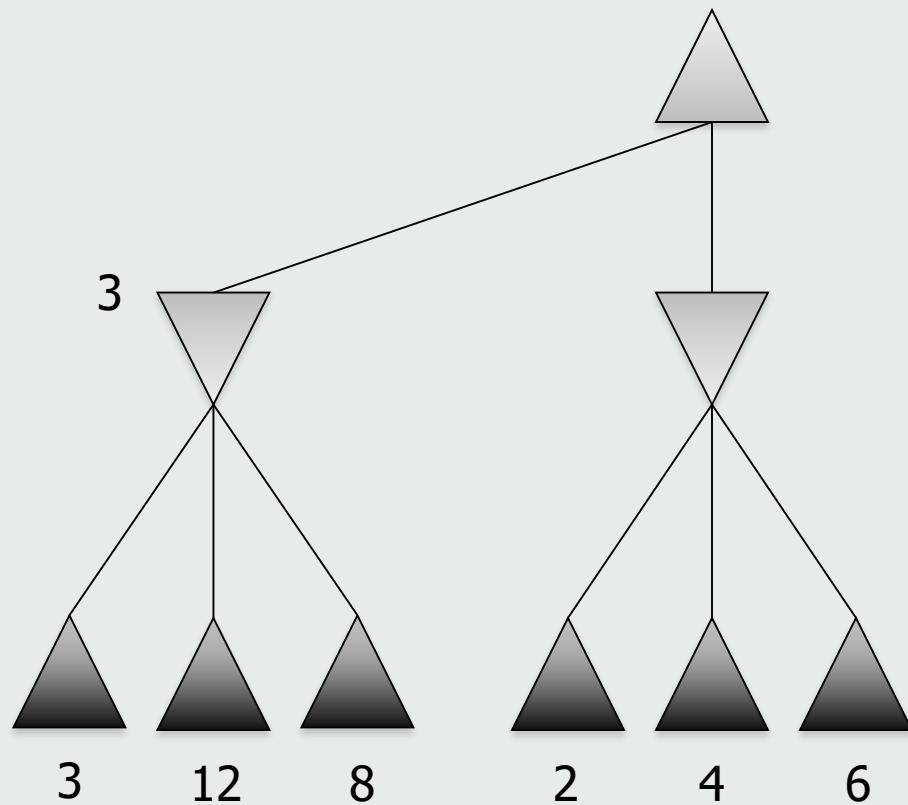
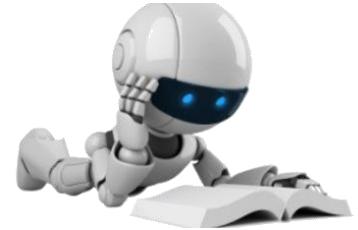


# Minimax: exemplo

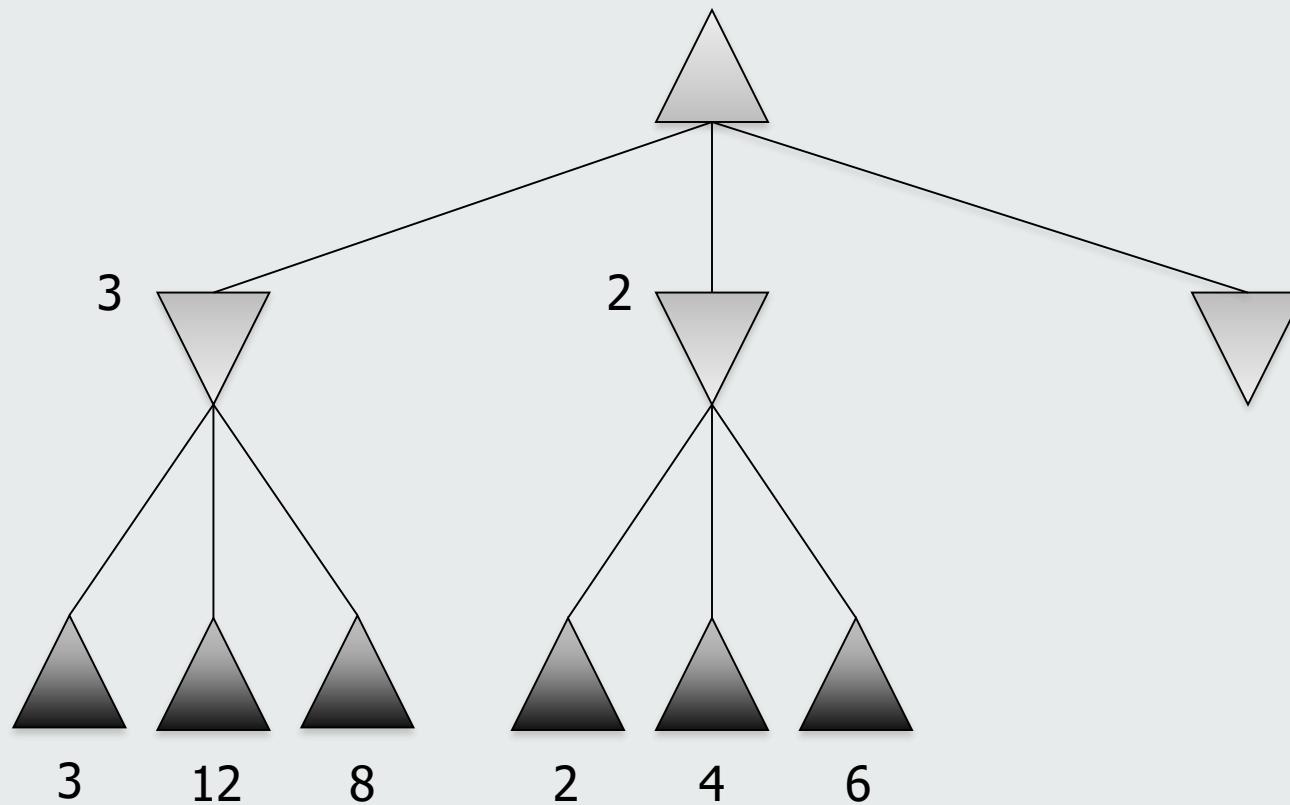




# Minimax: exemplo

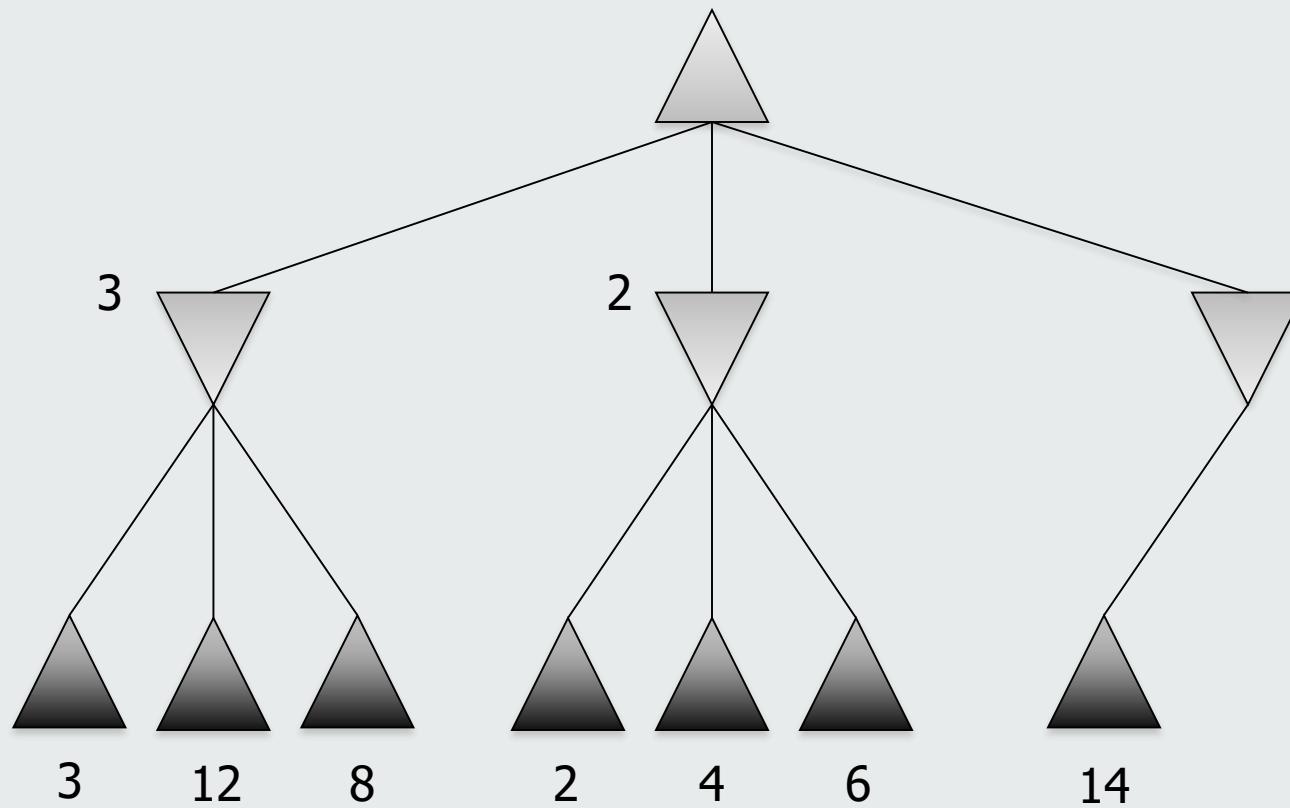


# Minimax: exemplo



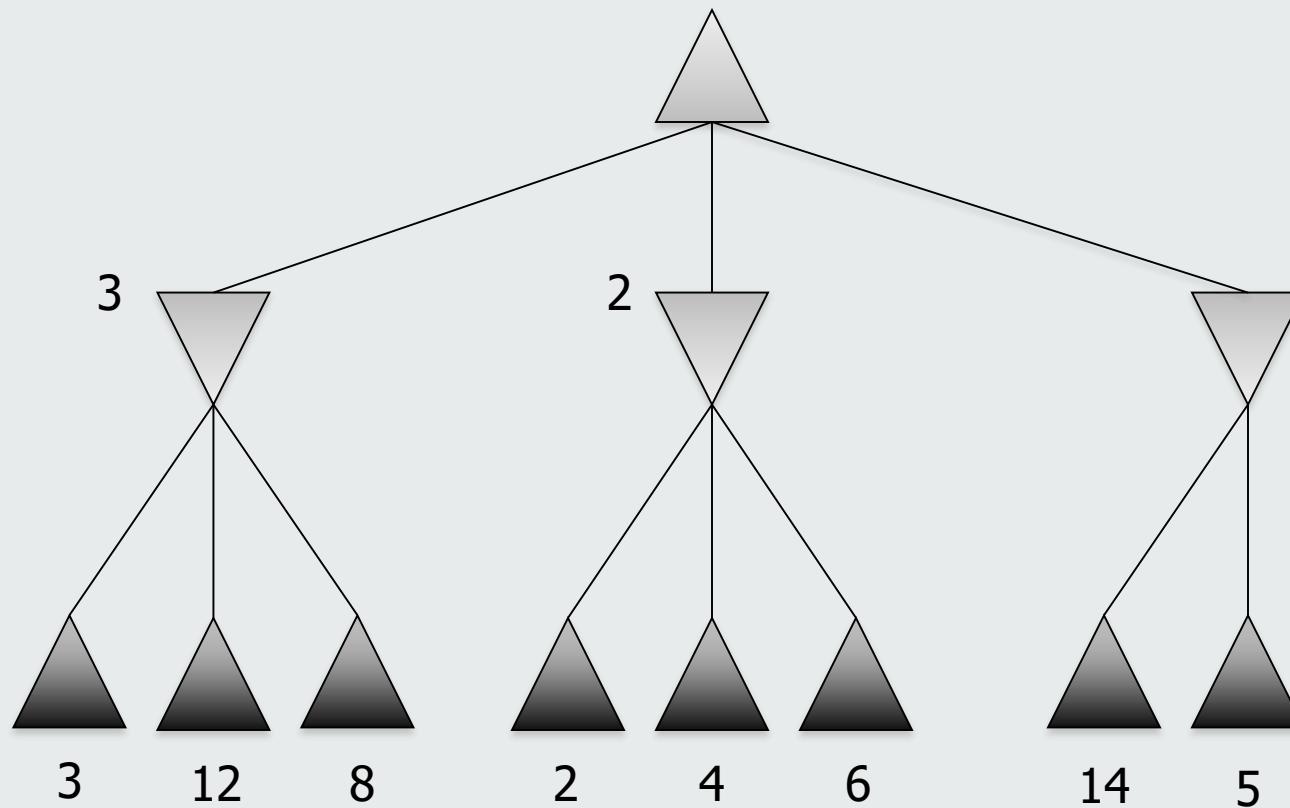


# Minimax: exemplo



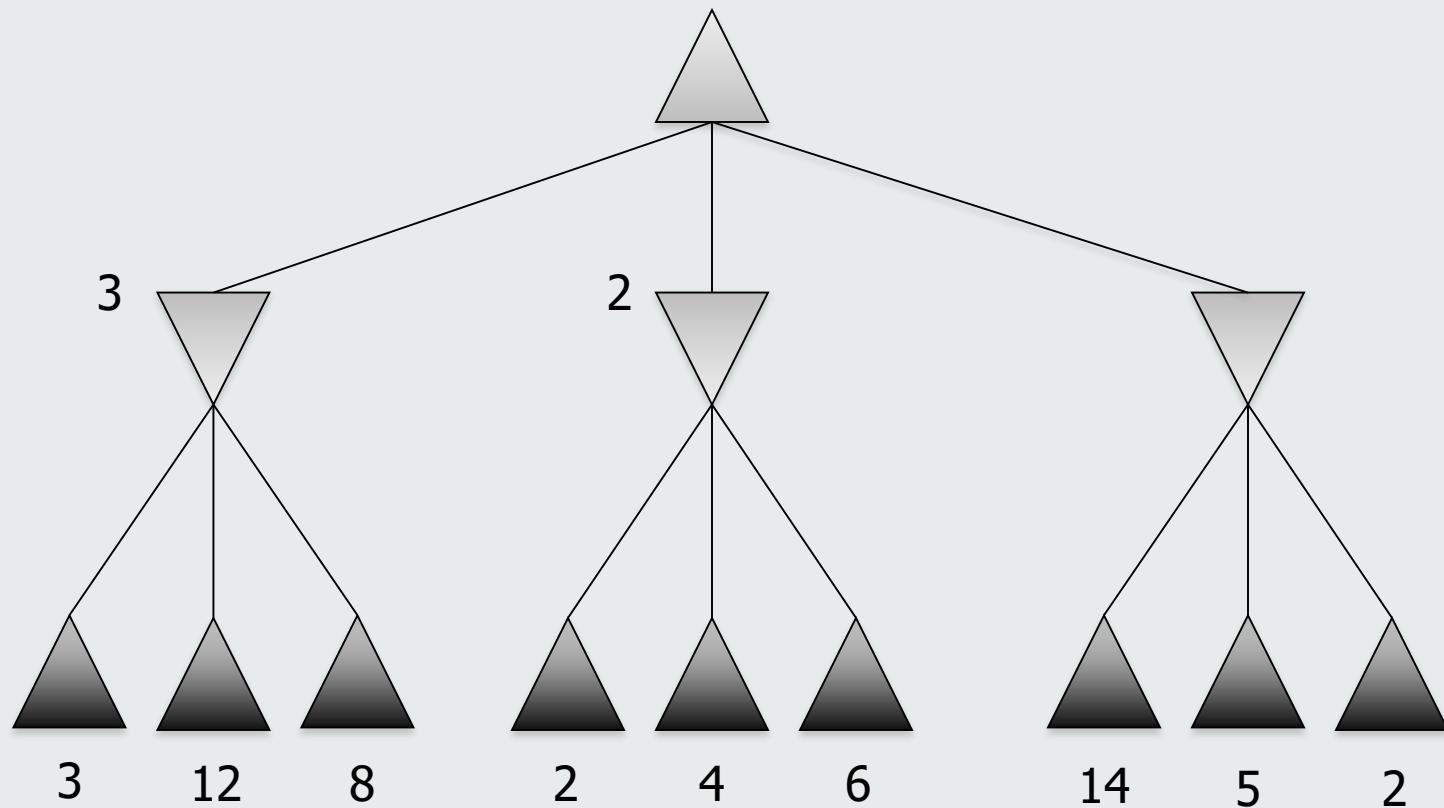


# Minimax: exemplo



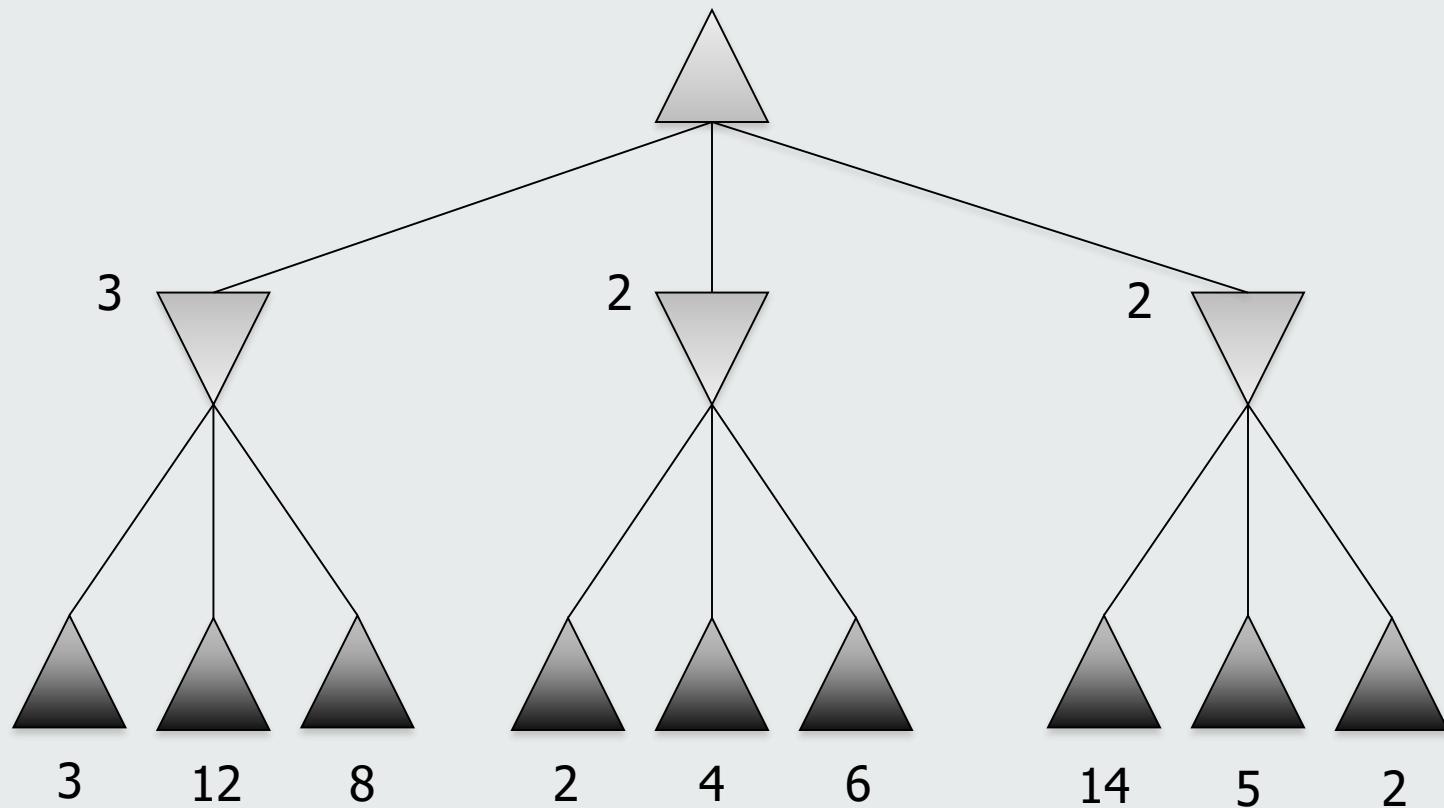


# Minimax: exemplo



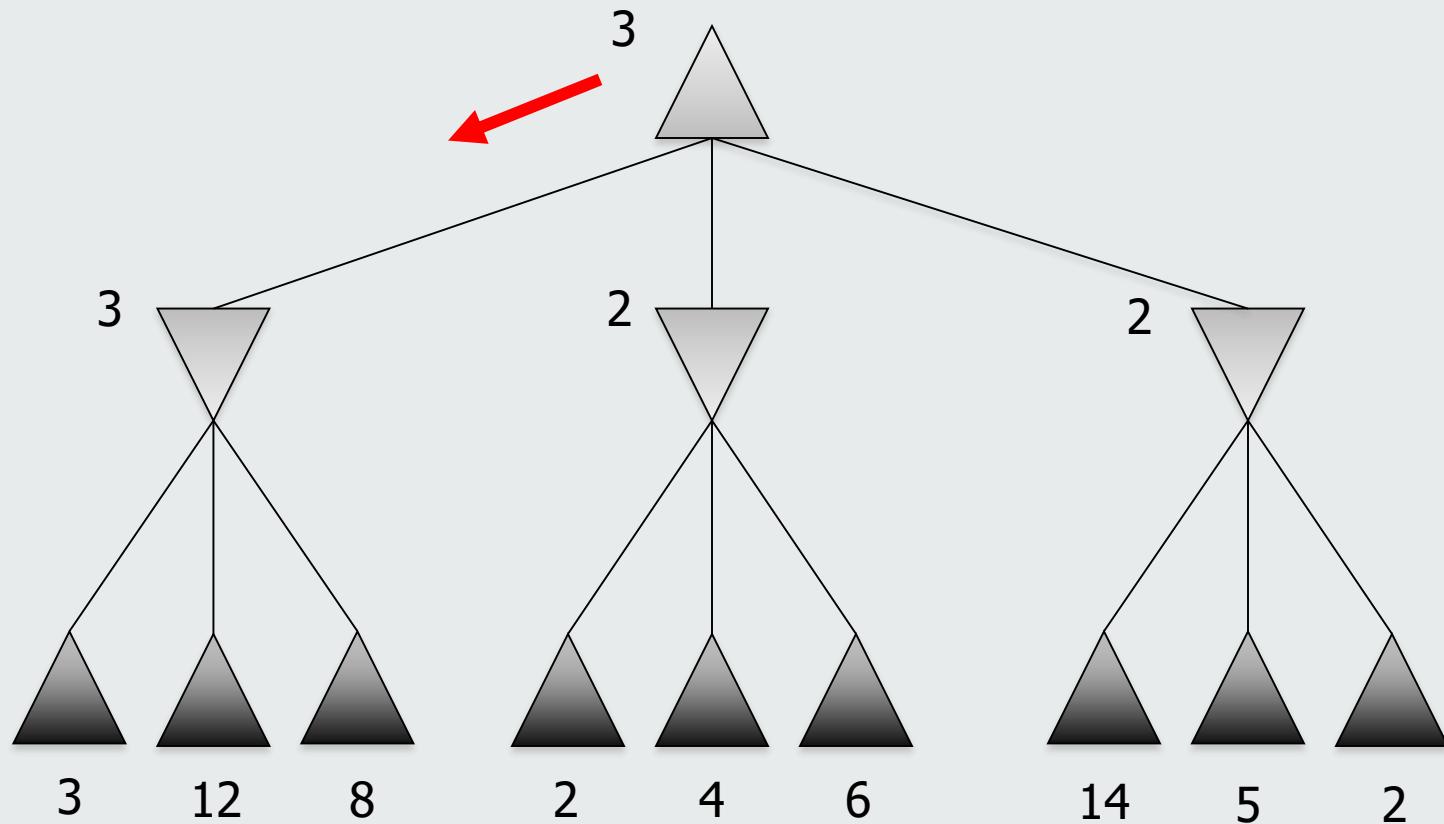


# Minimax: exemplo





# Minimax: exemplo





# Propriedades do algoritmo Minimax

O algoritmo faz uma procura em profundidade,  
**explorando toda a árvore de jogo.**

- É completo? Sim (se a árvore de procura é finita)
- É óptimo? Sim (contra um adversário óptimo)

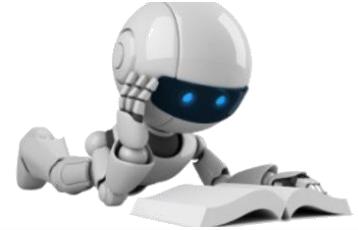
Considerando:

$m$  = profundidade máxima da árvore

$b$  = nº de movimentos legais em cada ponto

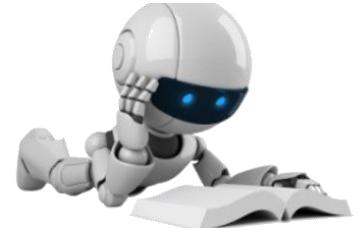
- Complexidade temporal?
  - $O(b^m)$
- Complexidade espacial?
  - $O(bm)$  para um algoritmo que gera todos os sucessores de uma vez
  - $O(m)$  para um algoritmo que gera os sucessores um a um

# Sumário

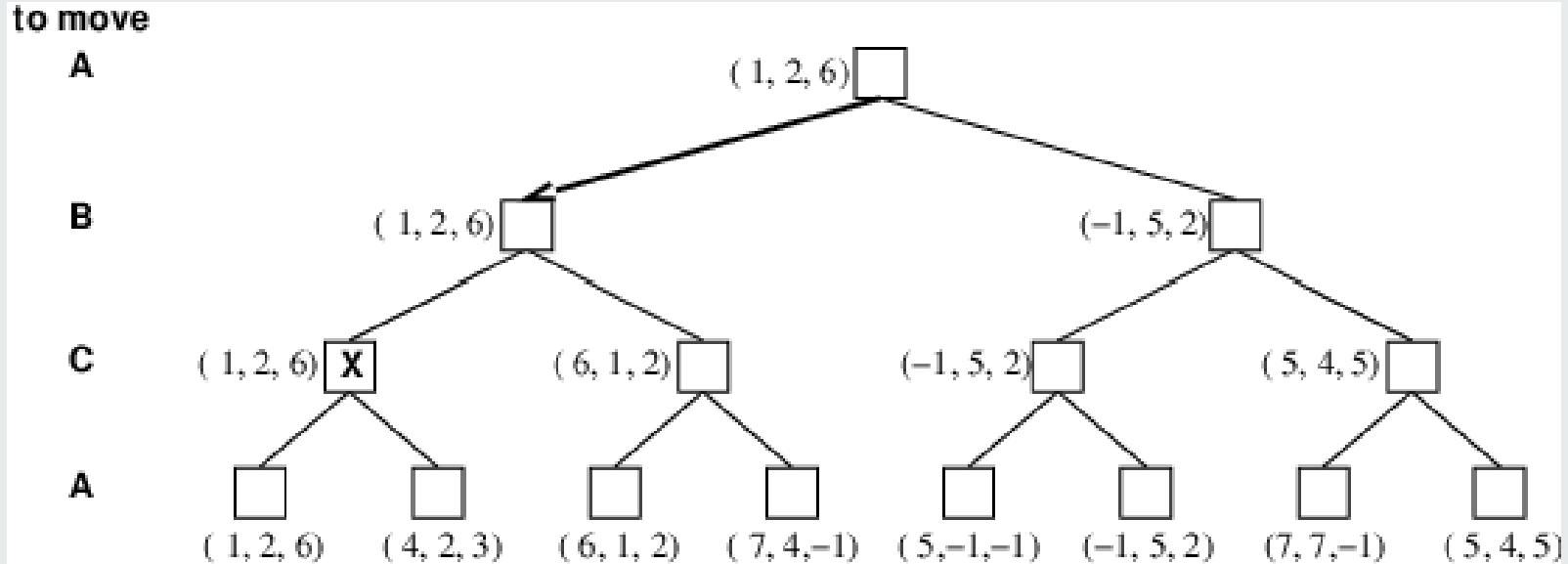


- Jogos: Conceitos Básicos
- Decisões óptimas em jogos
  - Estratégias óptimas e o Minimax
  - Estratégias óptimas com múltiplos jogadores
- Cortes  $\alpha$ - $\beta$
- Decisões imperfeitas em tempo real
- Jogos estocásticos
- Estado da arte em jogos

# Jogos com mais do que 2 jogadores



# Minimax: mais de 2 jogadores



- Função de utilidade devolve vector de valores com utilidade do estado do ponto de vista de cada jogador
- Cada jogador procura maximizar a sua utilidade (ex: C em X, escolhe a jogada que lhe dá mais pontos - 6)

# Minimax: mais de 2 jogadores

- Tipicamente levam a alianças, formais ou informais, entre os jogadores (também podem existir alianças em jogos com 2 jogadores).
- Estas alianças podem ser uma consequência natural de estratégias óptimas para cada jogador num jogo multi-jogadores.

# Sumário



- Jogos: Conceitos Básicos
- Decisões óptimas em jogos
  - Estratégias óptimas e o Minimax
  - Estratégias óptimas com múltiplos jogadores
- Cortes  $\alpha$ - $\beta$
- Decisões imperfeitas em tempo real
- Jogos estocásticos
- Estado da arte em jogos

# Problema do Minimax



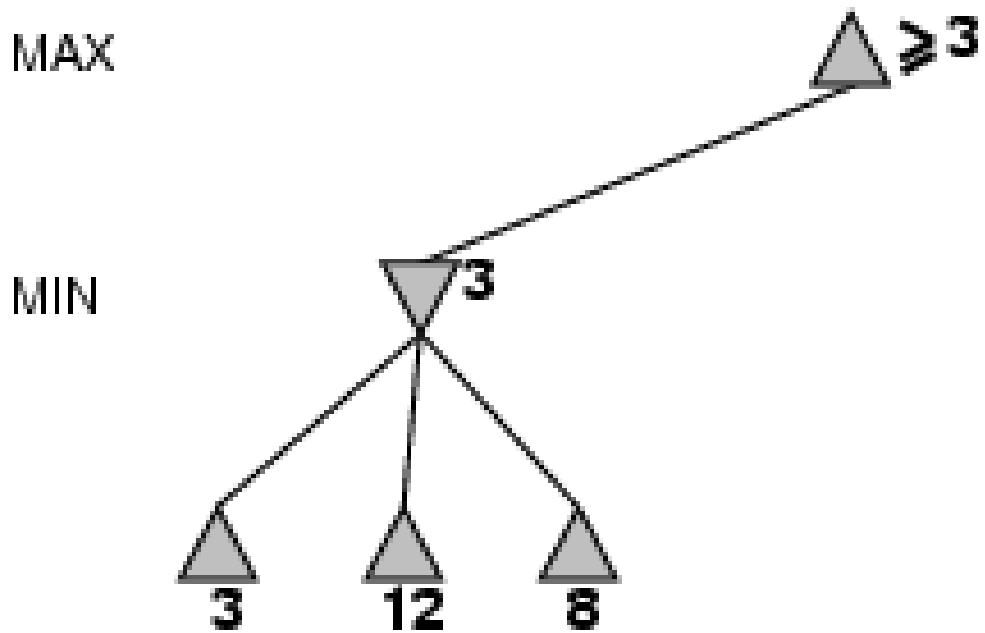
- Complexidade temporal muito elevada  $O(b^m)$ 
  - Não se consegue aplicar em jogos reais
  - Jogos são muito mais difíceis do que os problemas de procura
  - Factor de ramificação muito elevado - e.g. xadrez
    - factor de ramificação  $\approx 35$
    - 50 jogadas/jogador  $\rightarrow 35^{100}$  nós (destes “apenas”  $10^{40}$  são nós distintos)
    - impossível determinar a solução exacta



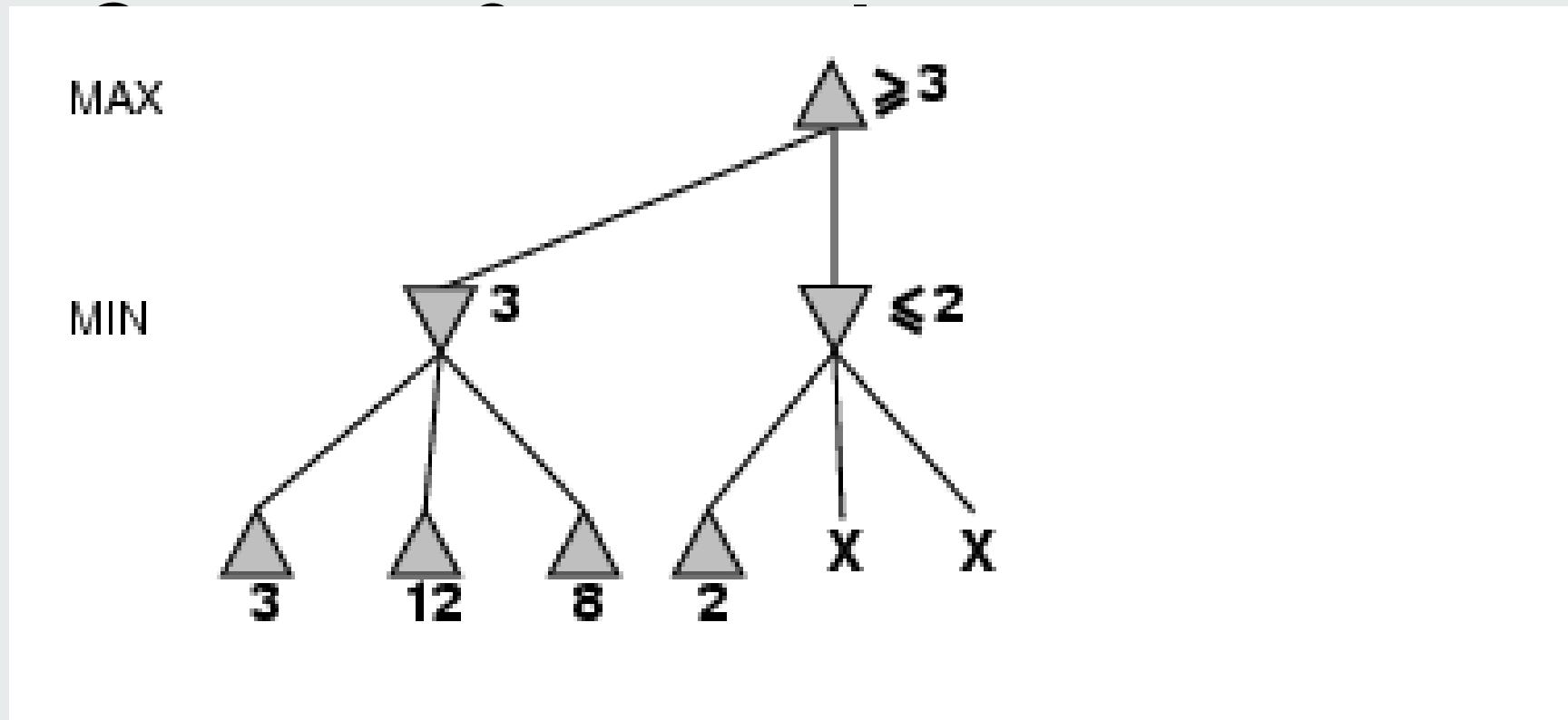
- Motivação:
  - Minimax: número de estados examinados é exponencial em função do número de jogadas
  - Não é possível eliminar o factor exponencial, mas podemos reduzir o número de estados analisados para metade!
  - É possível calcular **exactamente a mesma decisão** resultante do algoritmo Minimax **sem ter de analisar todos os estados**
  - Cortes permitem eliminar partes da árvore de procura que são irrelevantes para o resultado final.



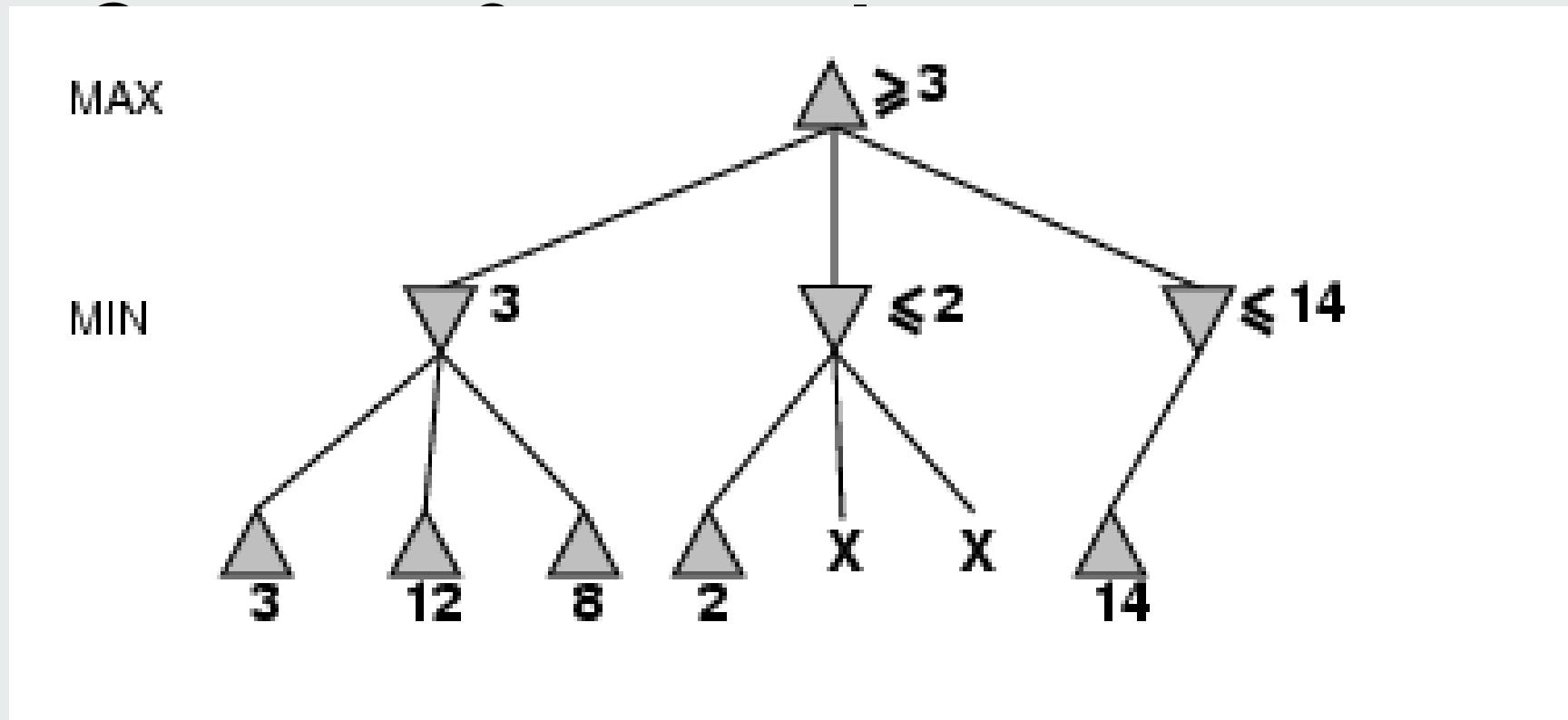
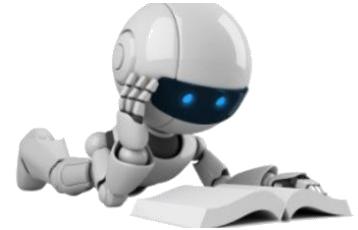
# Cortes $\alpha$ - $\beta$ : exemplo



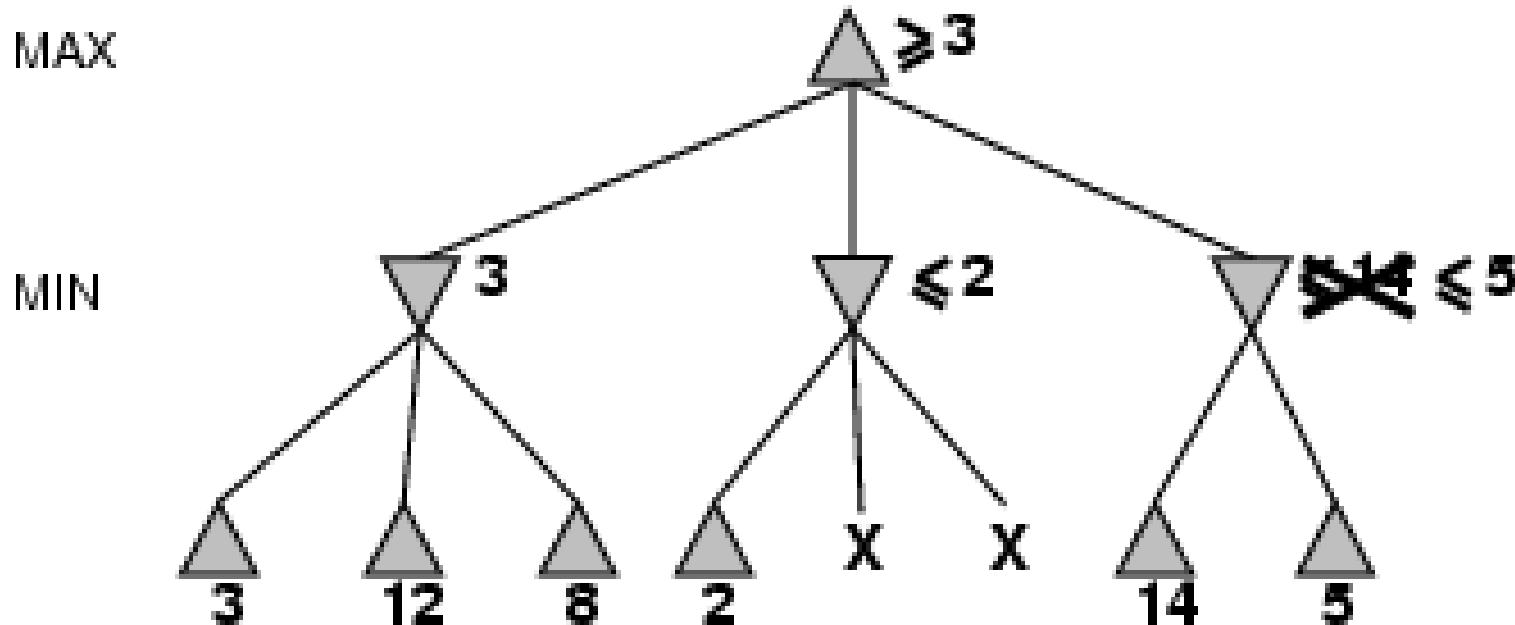
# Cortes $\alpha$ - $\beta$ : exemplo



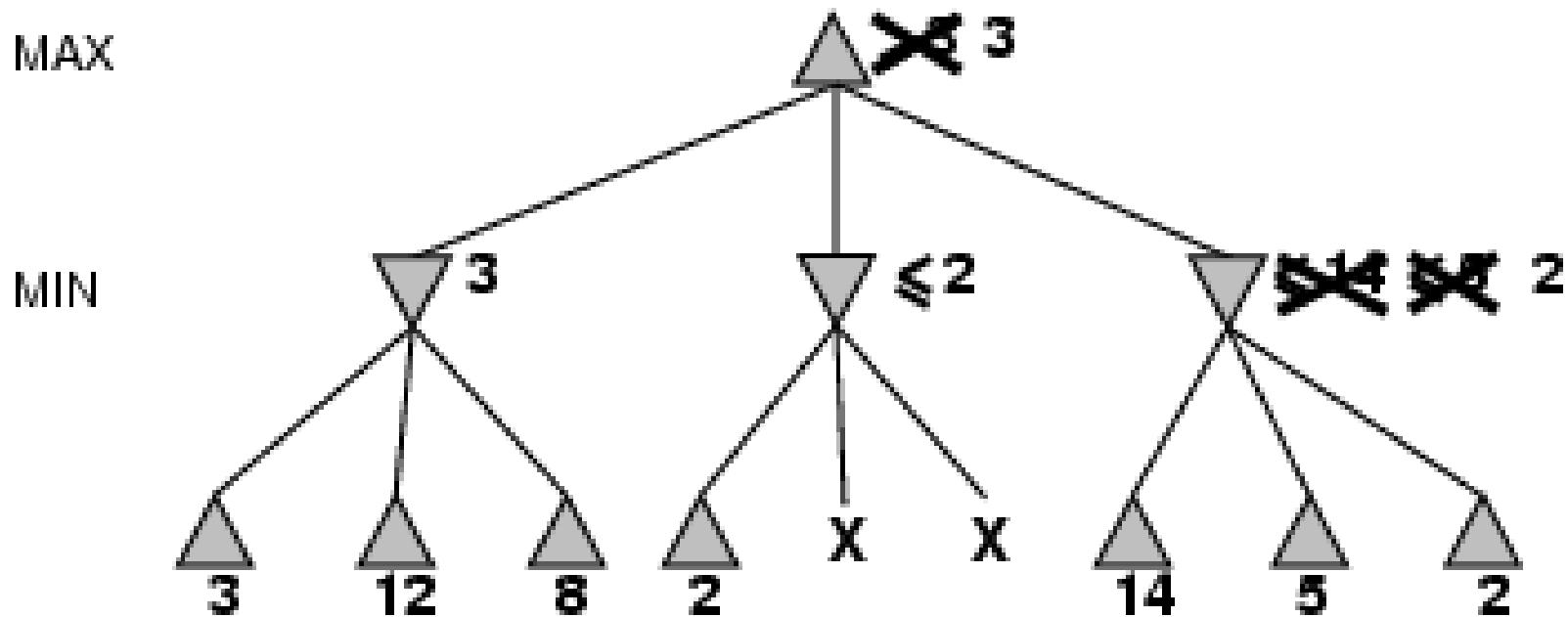
# Cortes $\alpha$ - $\beta$ : exemplo



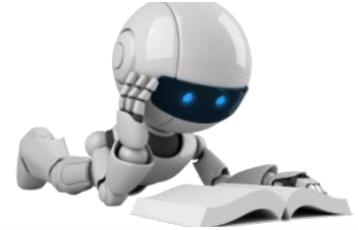
# Cortes $\alpha$ - $\beta$ : exemplo



# Cortes $\alpha$ - $\beta$ : exemplo

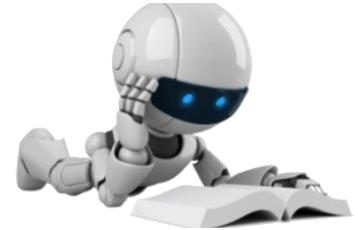


- Os nós sucessores do primeiro nó a ser expandido em cada nível de profundidade nunca podem ser cortados



- $\alpha$  de um nó  $n$ 
  - valor da **melhor escolha para o jogador Max** (i.e., valor mais elevado), encontrada em qualquer ponto de decisão **ao longo do caminho para  $n$**
- $\beta$  de um nó  $n$ 
  - valor da **melhor escolha para o jogador Min** (i.e., valor mais baixo), encontrada em qualquer ponto de decisão **ao longo do caminho para  $n$**
- Nó  $n$  Max:
  - Se valor nó  $n \geq \beta$ , então o nó nunca vai ser escolhido
- Nó  $n$  Min:
  - Se valor nó  $n \leq \alpha$ , então o nó nunca vai ser escolhido
- Nestes casos não precisamos de visitar todos os sucessores do nó  $n$

# Algoritmo $\alpha$ - $\beta$



**function** Alfa-Beta-Search(*state*) **returns** an *action*

$v \leftarrow \text{Max-Value}(\text{state}, -\infty, +\infty)$

**return** the *action* in Actions(*state*) with value  $v$

**function** Max-Value(*state*,  $\alpha$ ,  $\beta$ ) **returns** a *utility* value

**if** Terminal-Test (*state*) **then return** Utility(*state*)

$V \leftarrow -\infty$

**for each** *a* in Actions(*state*) **do**

$v \leftarrow \text{Max}(v, \text{Min-Value}(\text{Result}(\text{state}, a), \alpha, \beta))$

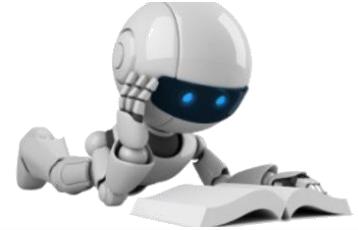
**if**  $v \geq \beta$  **then return**  $v$

$\alpha \leftarrow \text{Max}(\alpha, v)$

**return**  $v$

→ corte

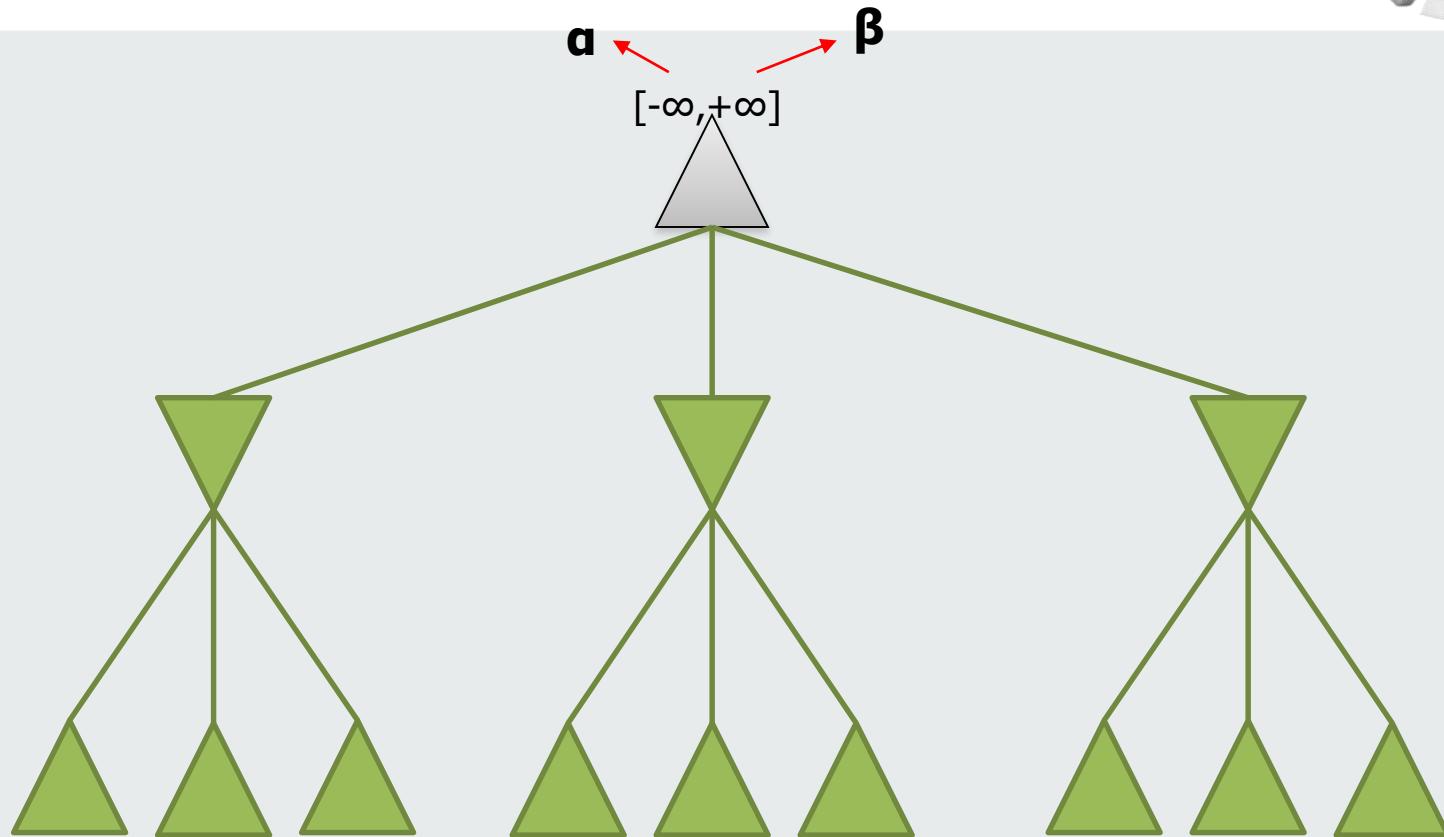
# Algoritmo $\alpha$ - $\beta$



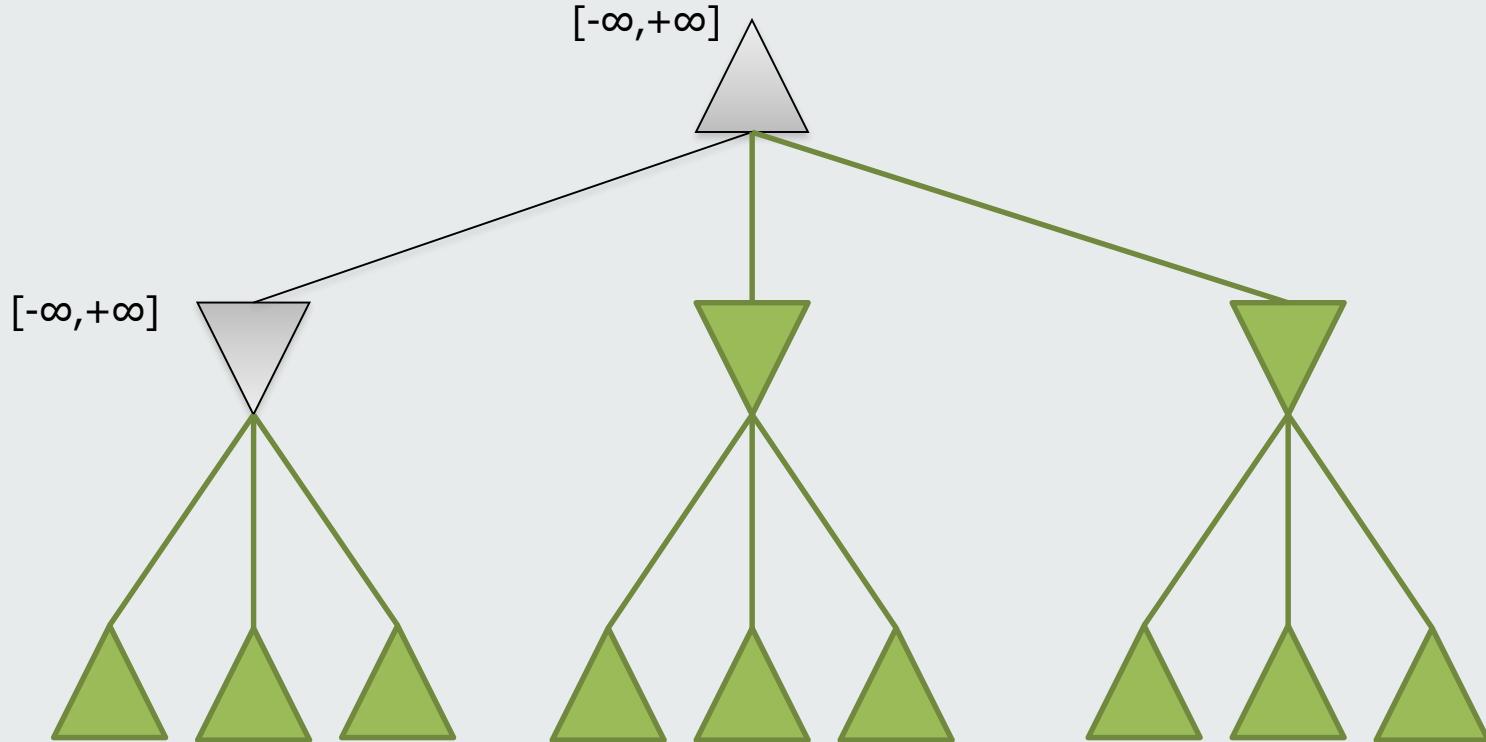
```
function Min-Value(state, $\alpha$ , $\beta$ ) returns a utility value
  if Terminal-Test (state) then return Utility(state)
   $v \leftarrow +\infty$ 
  for each  $a$  in Actions(state) do
     $v \leftarrow \text{Min}(v, \text{Max-Value}(\text{Result}(state, a), \alpha, \beta))$ 
    if  $v \leq \alpha$  then return  $v$ 
     $\beta \leftarrow \text{Min}(\beta, v)$ 
  return  $v$ 
```

 corte

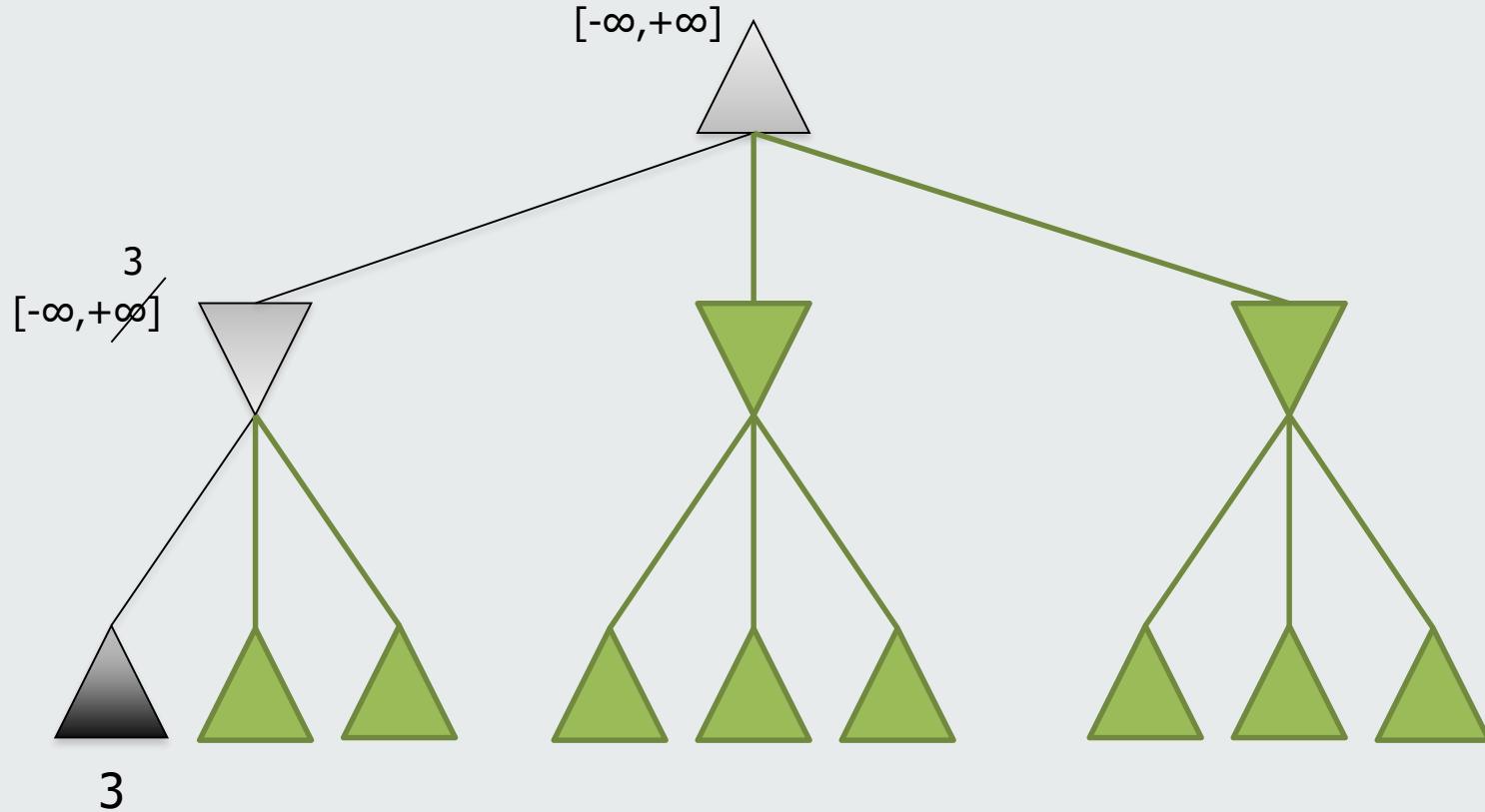
# Cortes $\alpha$ - $\beta$ : exemplo



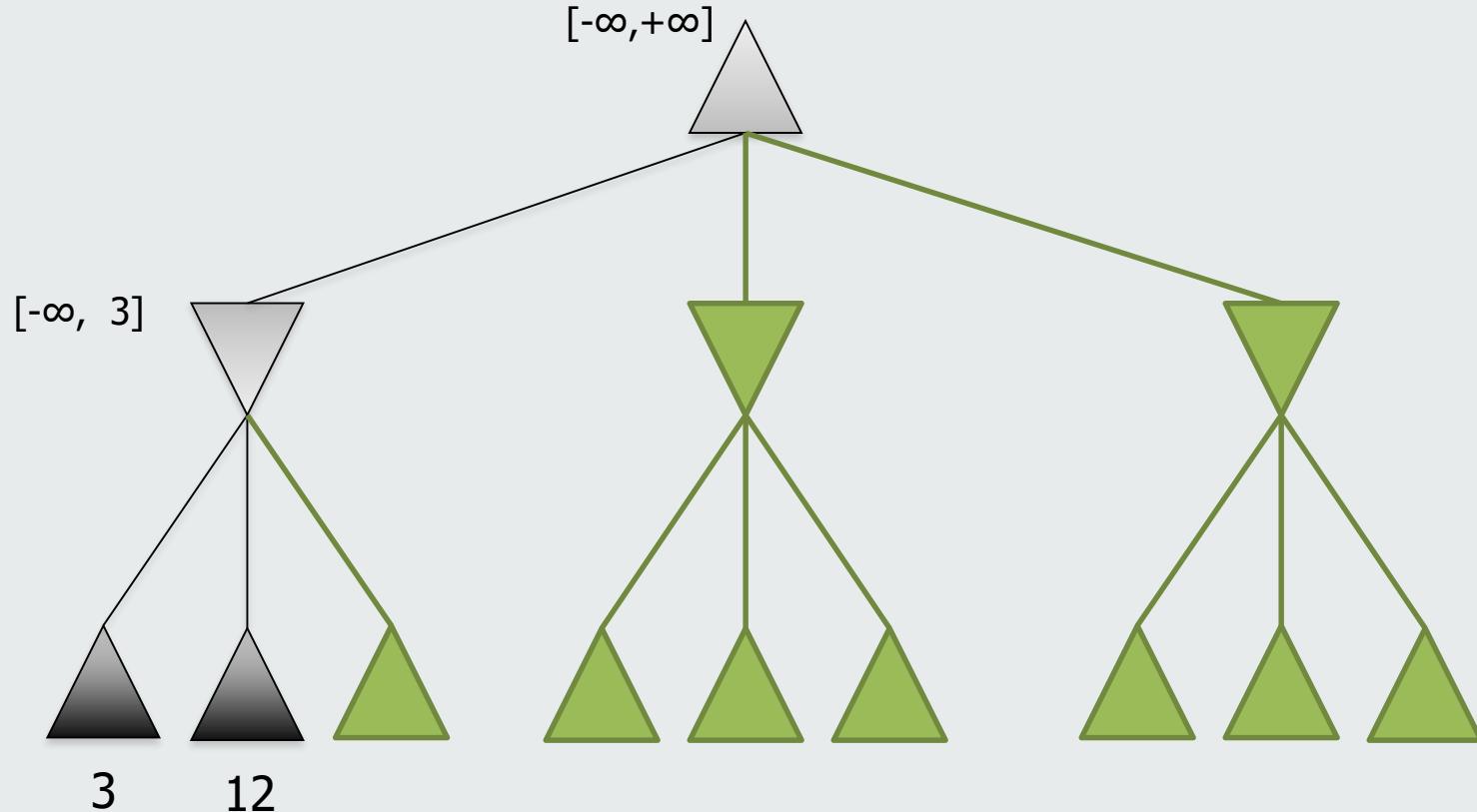
# Cortes $\alpha$ - $\beta$ : exemplo



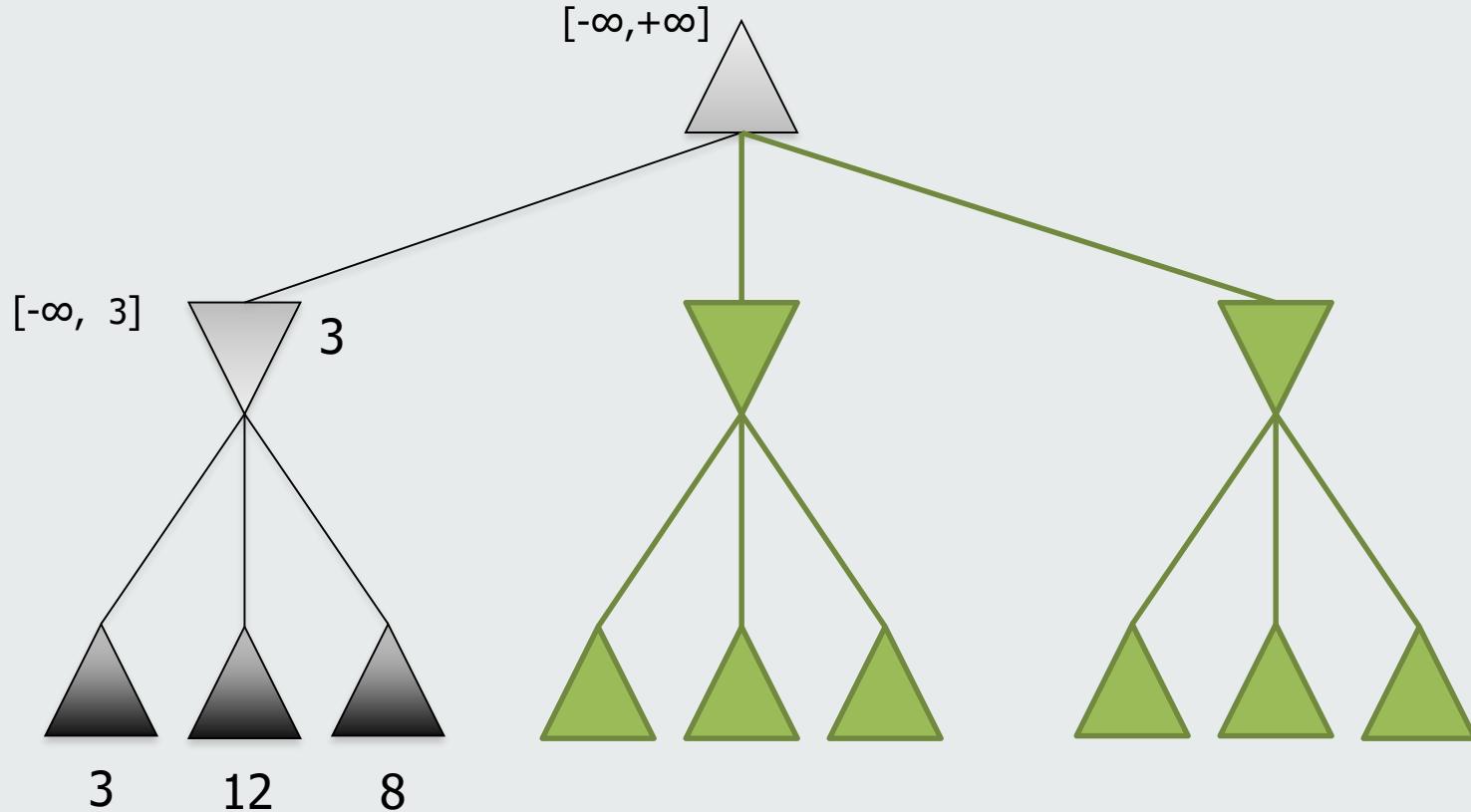
# Cortes $\alpha$ - $\beta$ : exemplo



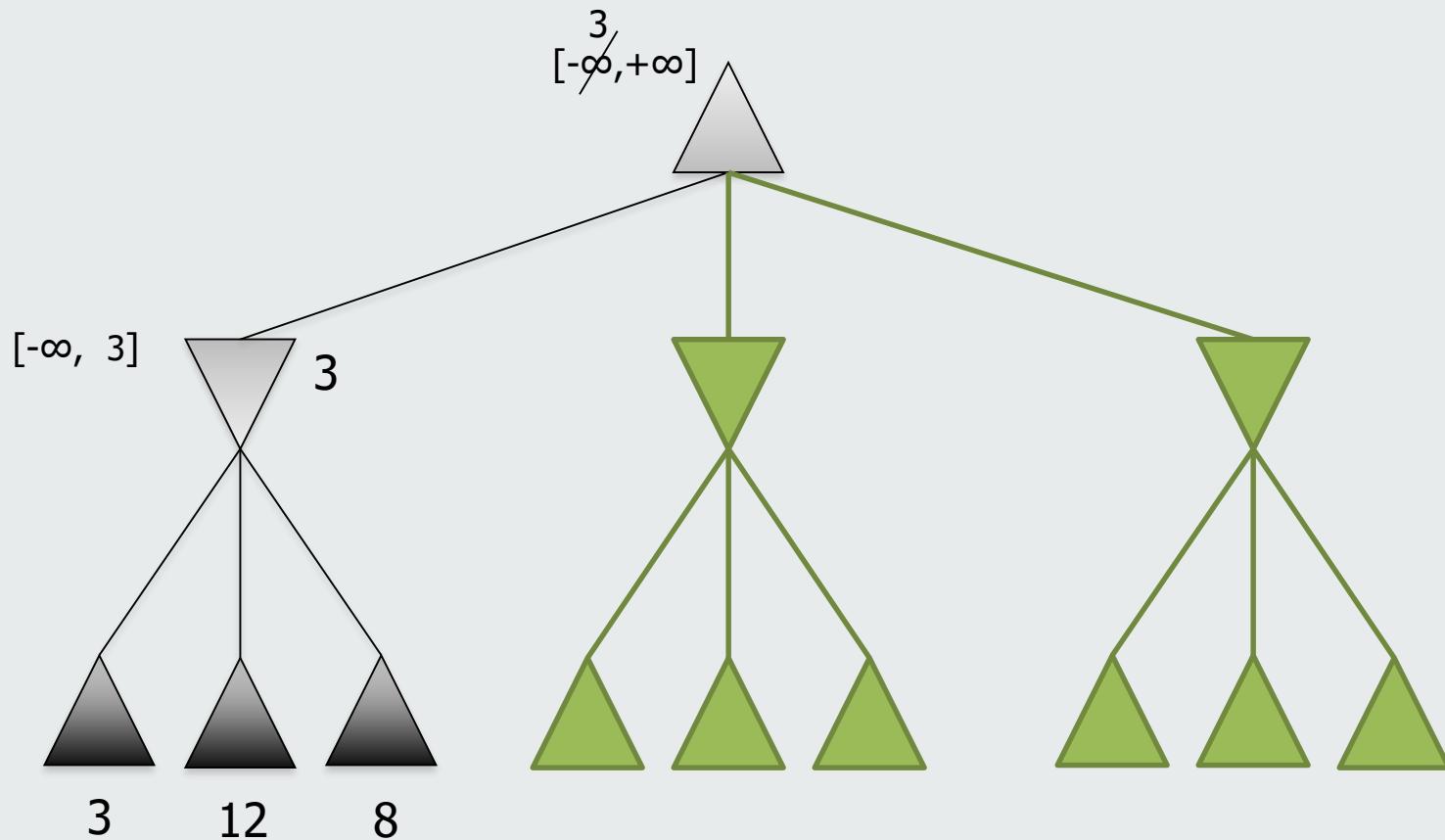
# Cortes $\alpha$ - $\beta$ : exemplo



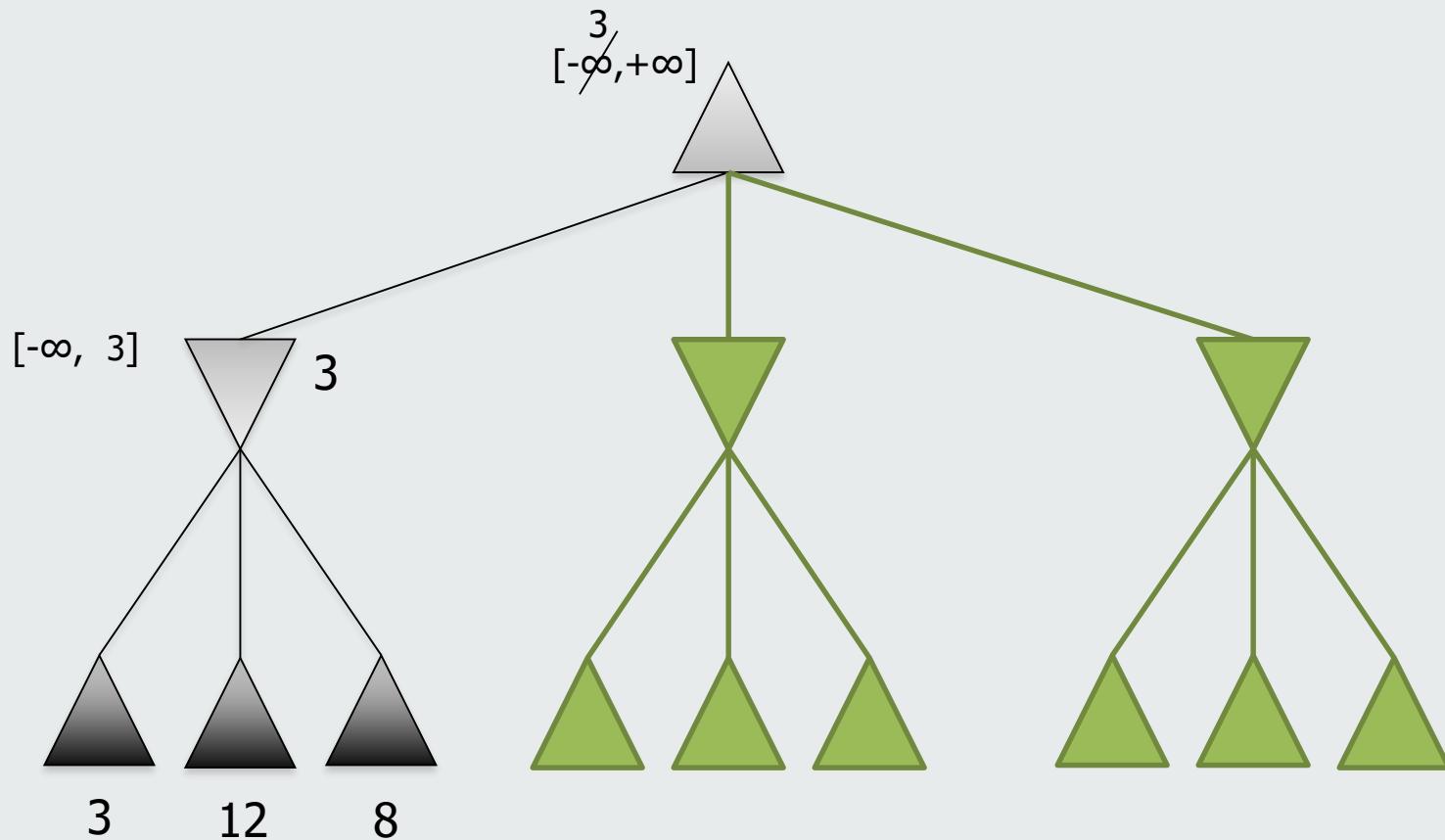
# Cortes $\alpha$ - $\beta$ : exemplo



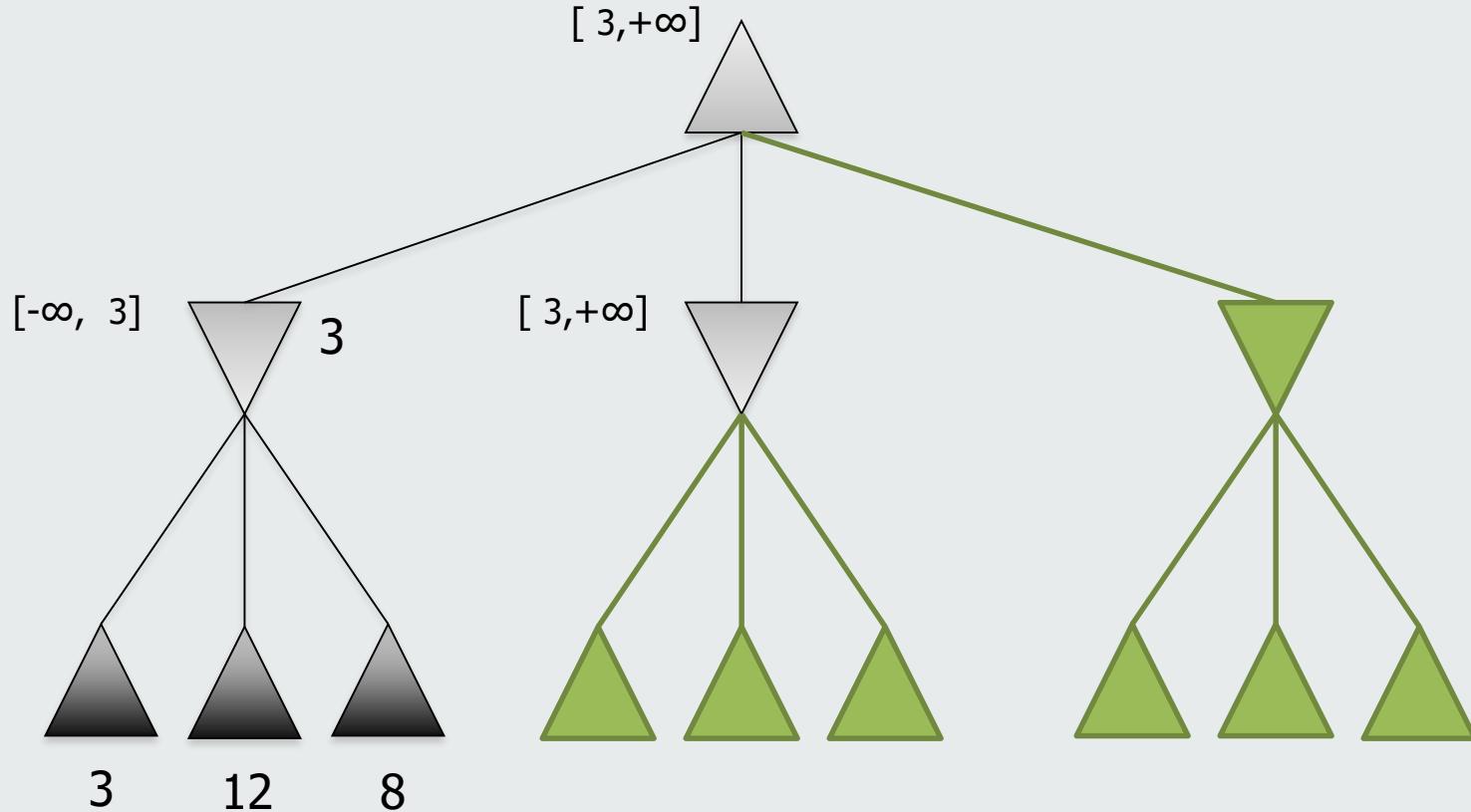
# Cortes $\alpha$ - $\beta$ : exemplo



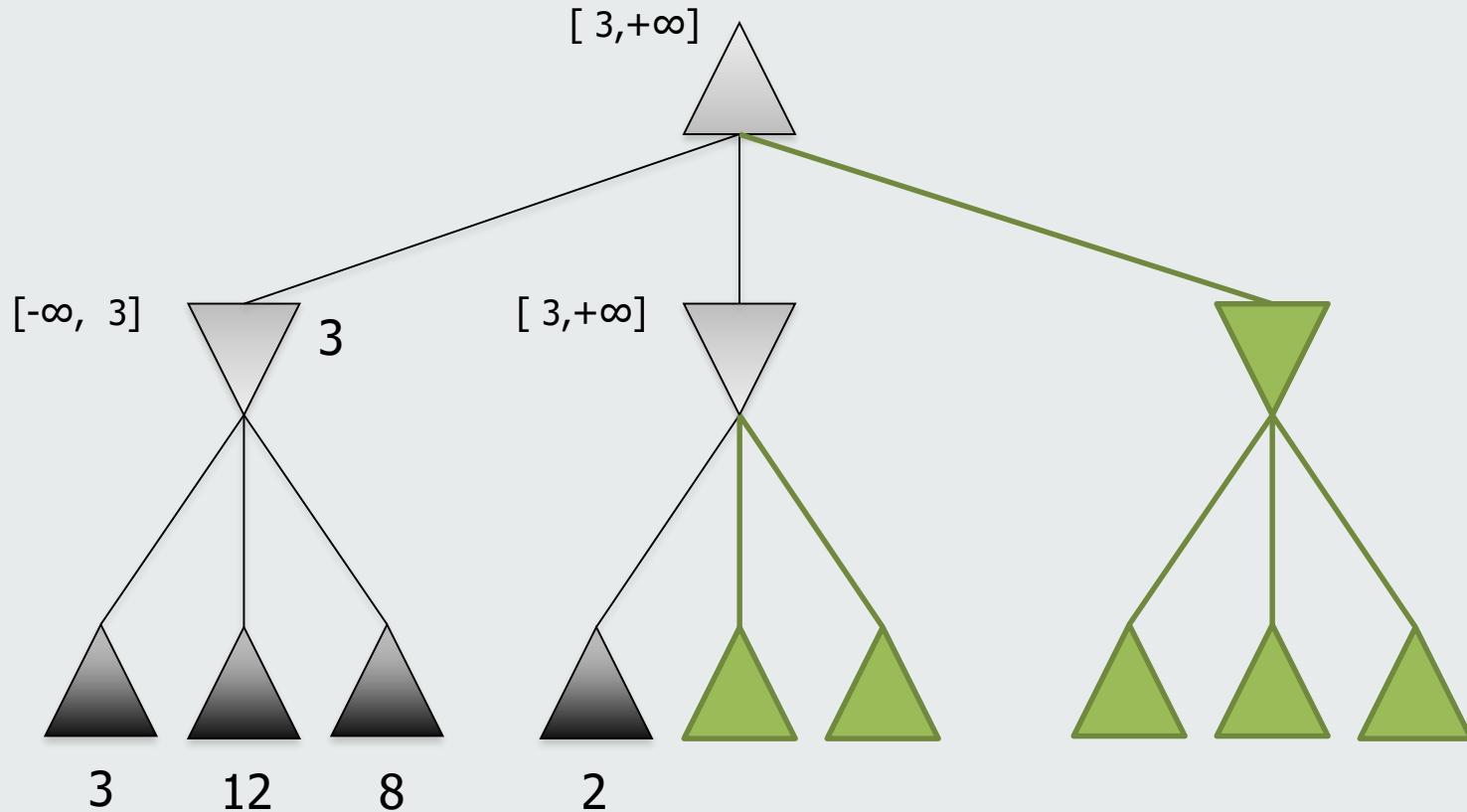
# Cortes $\alpha$ - $\beta$ : exemplo



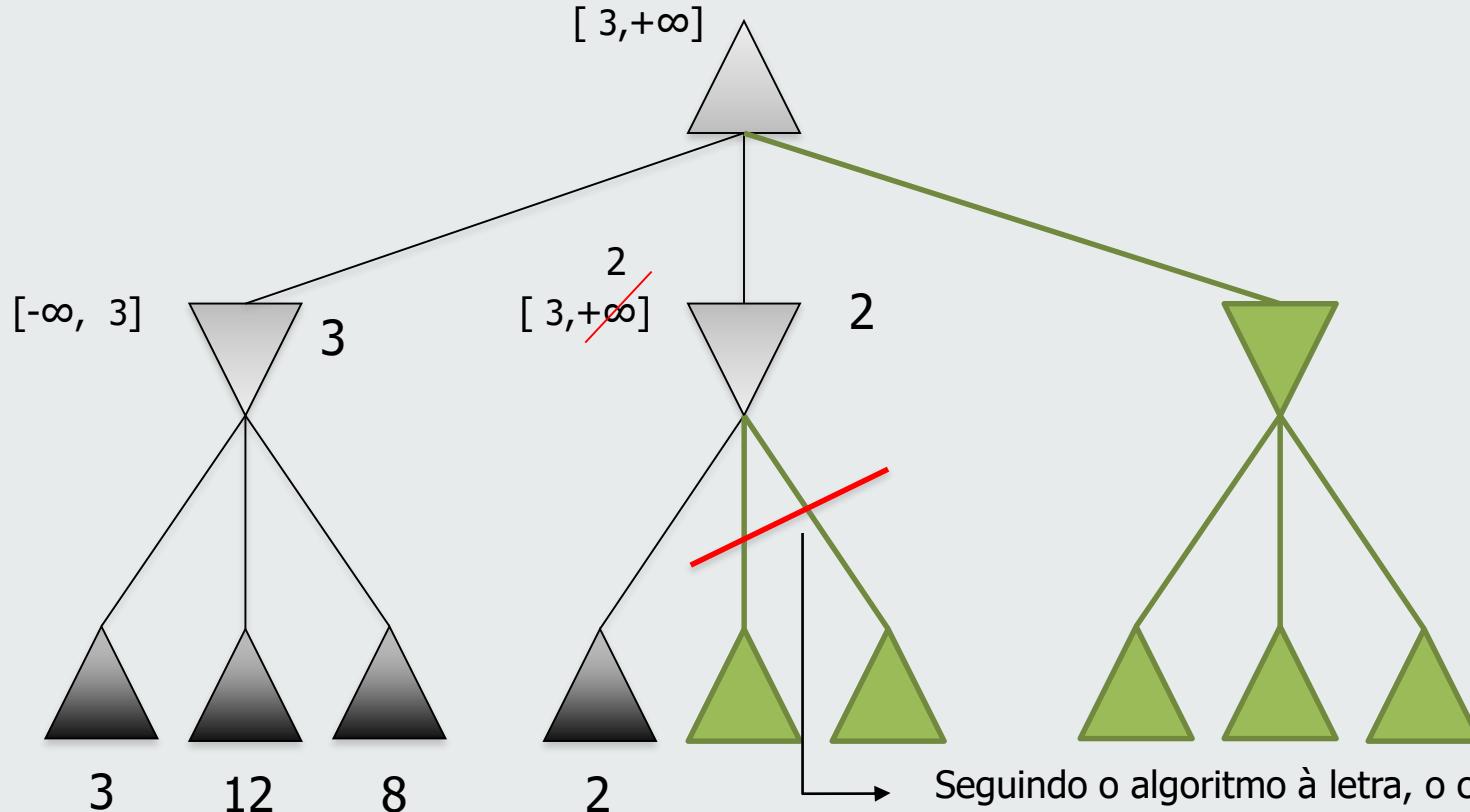
# Cortes $\alpha$ - $\beta$ : exemplo



# Cortes $\alpha$ - $\beta$ : exemplo



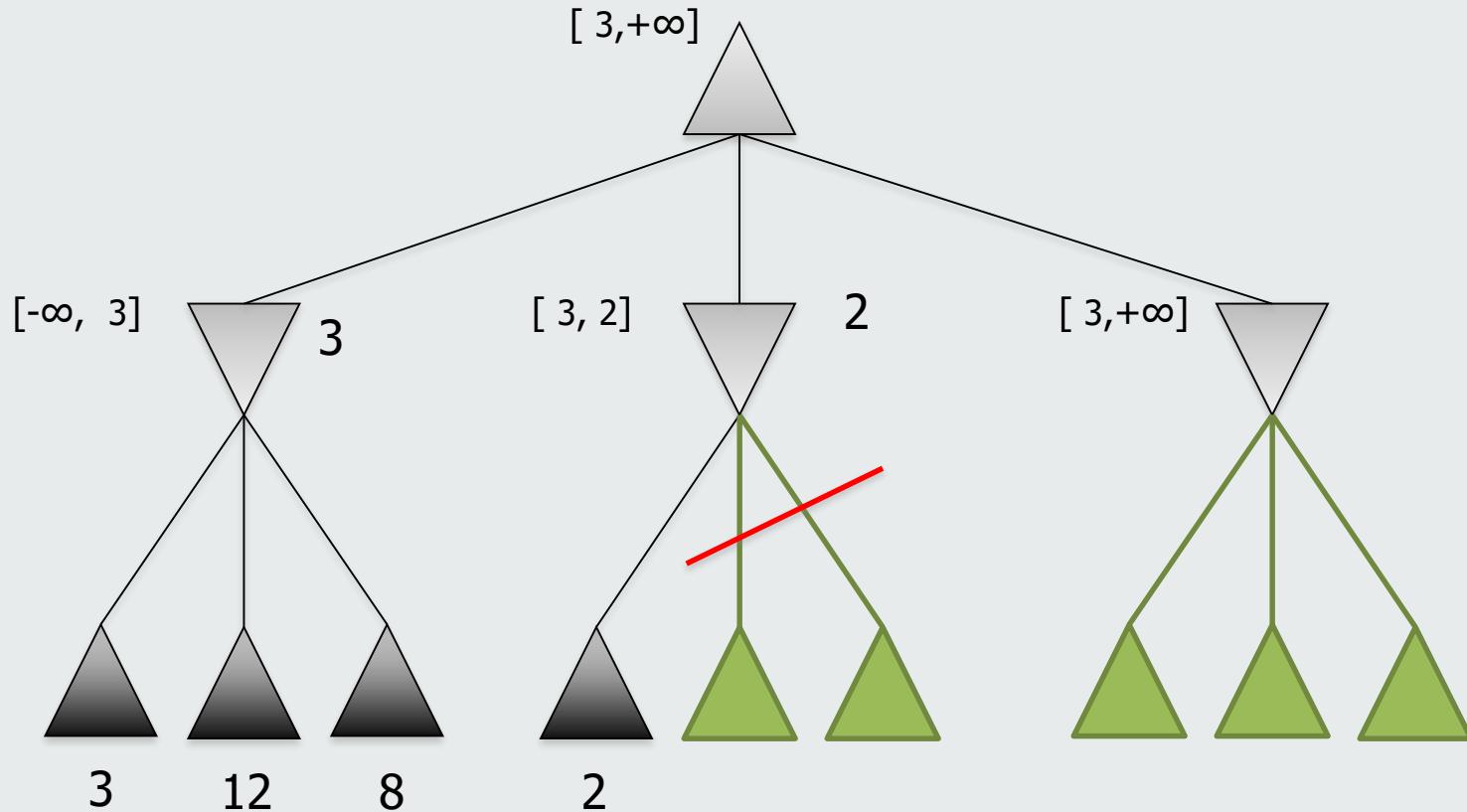
# Cortes $\alpha$ - $\beta$ : exemplo



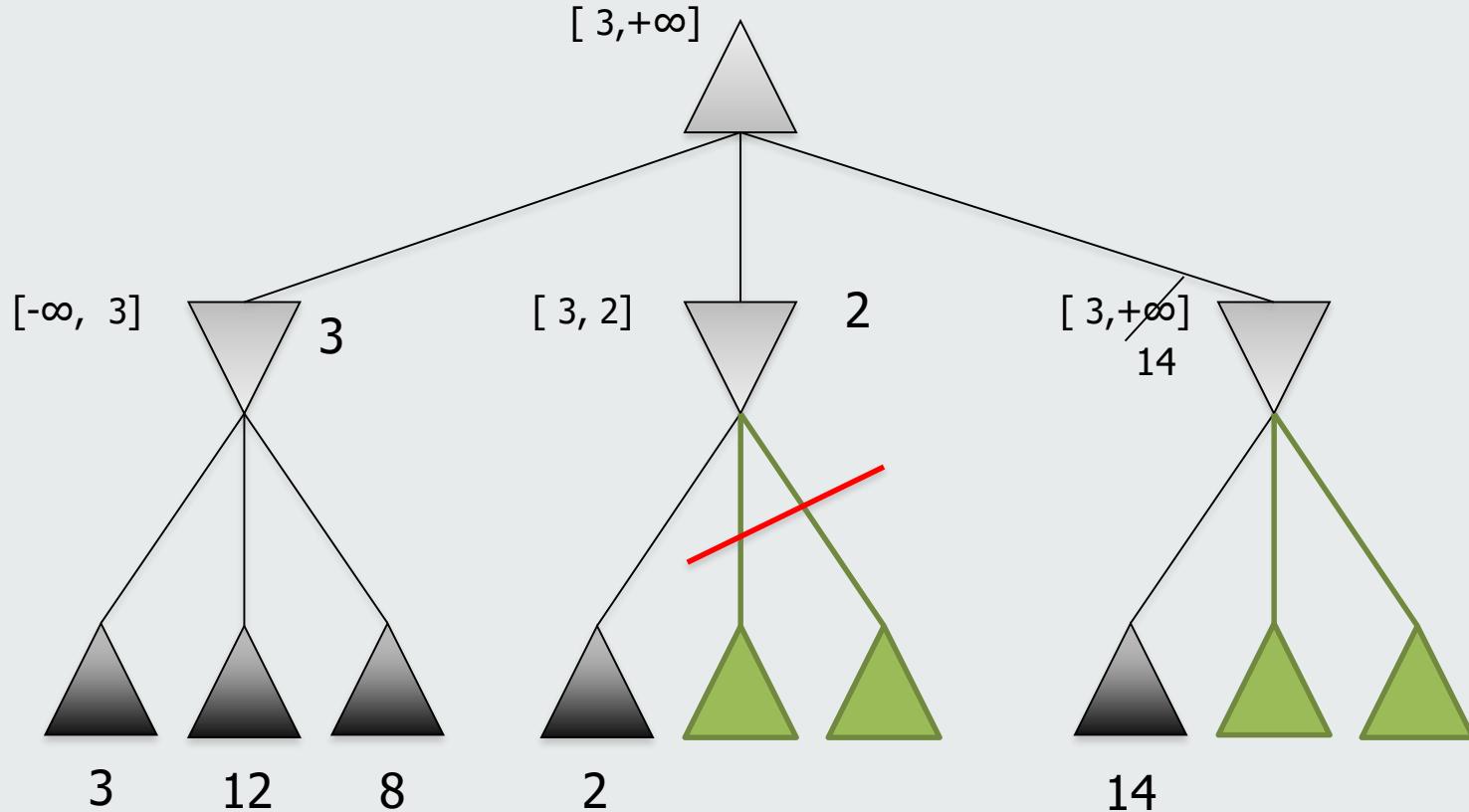
No entanto é equivalente (e mais fácil de compreender) actualizar primeiro  $\beta$ , e cortar sempre que:

- $\alpha \geq \beta$

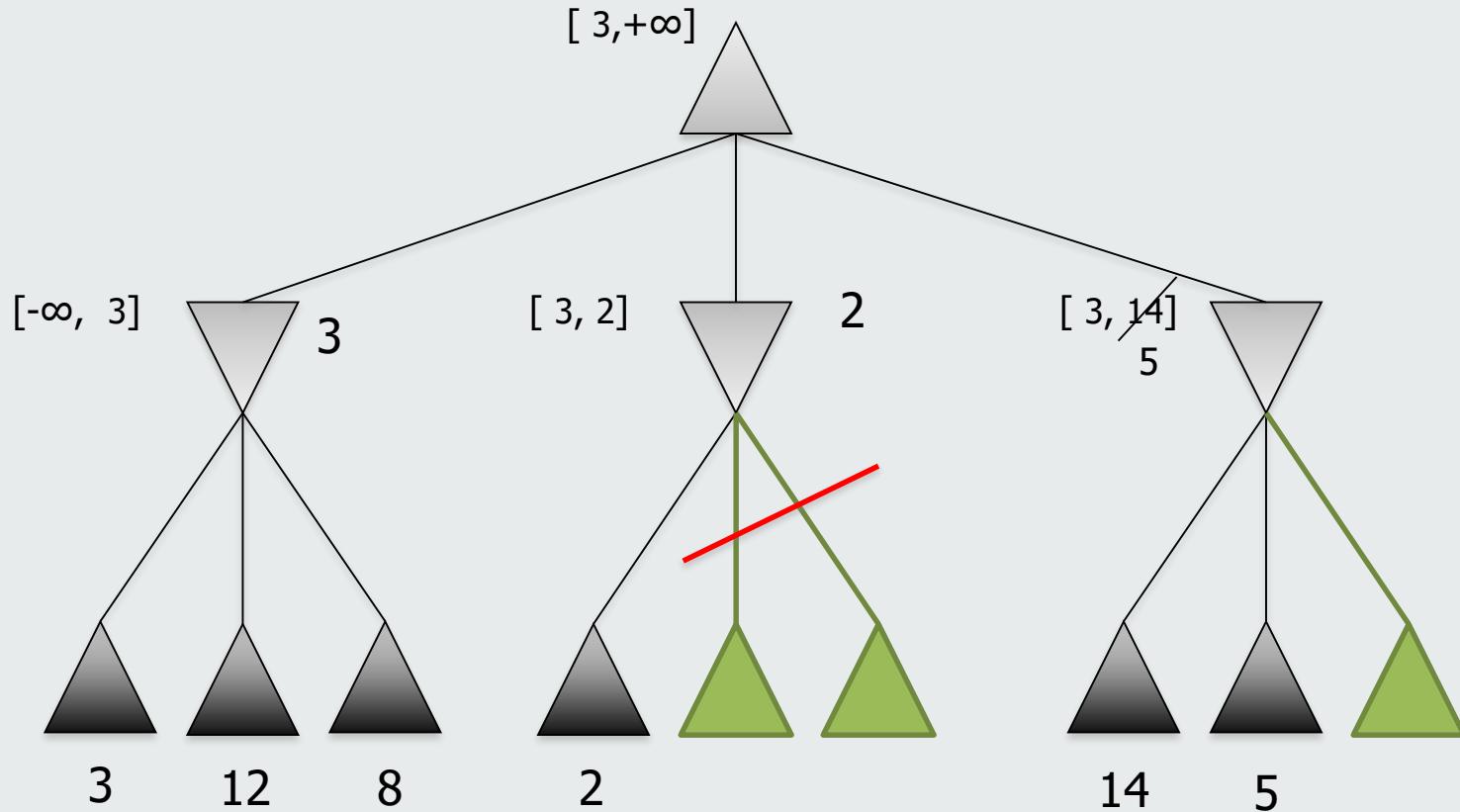
# Cortes $\alpha$ - $\beta$ : exemplo



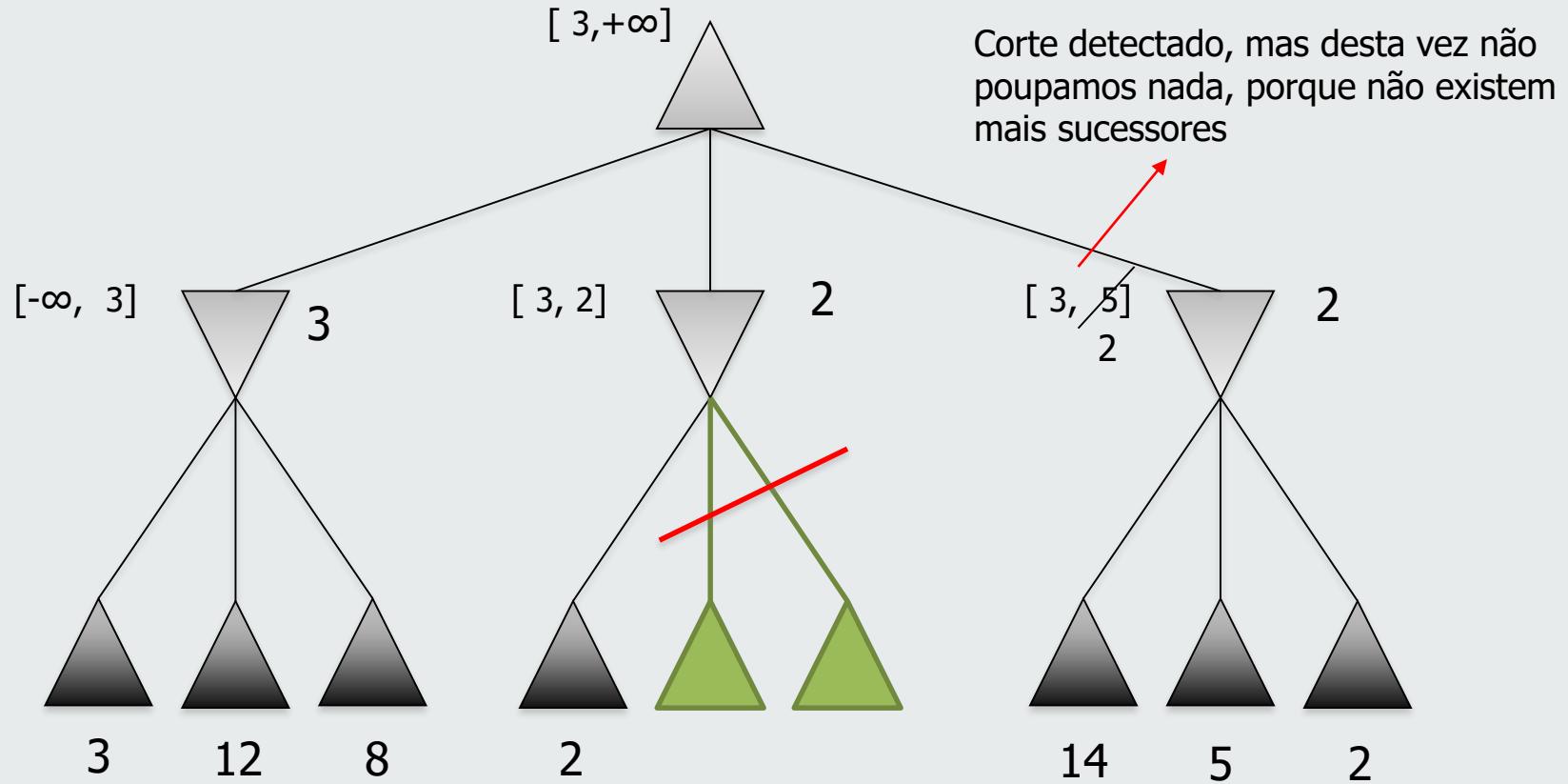
# Cortes $\alpha$ - $\beta$ : exemplo



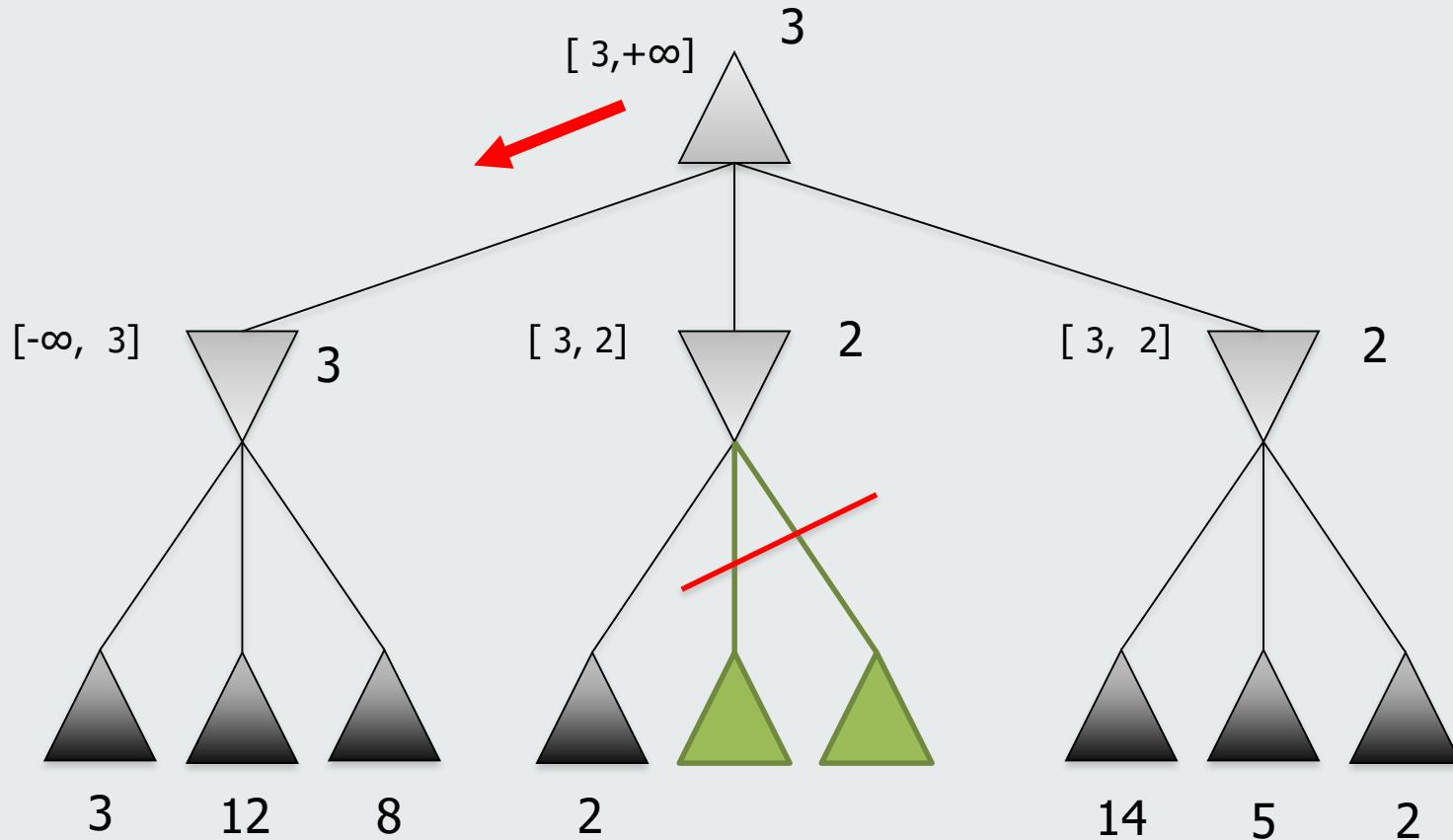
# Cortes $\alpha$ - $\beta$ : exemplo



# Cortes $\alpha$ - $\beta$ : exemplo



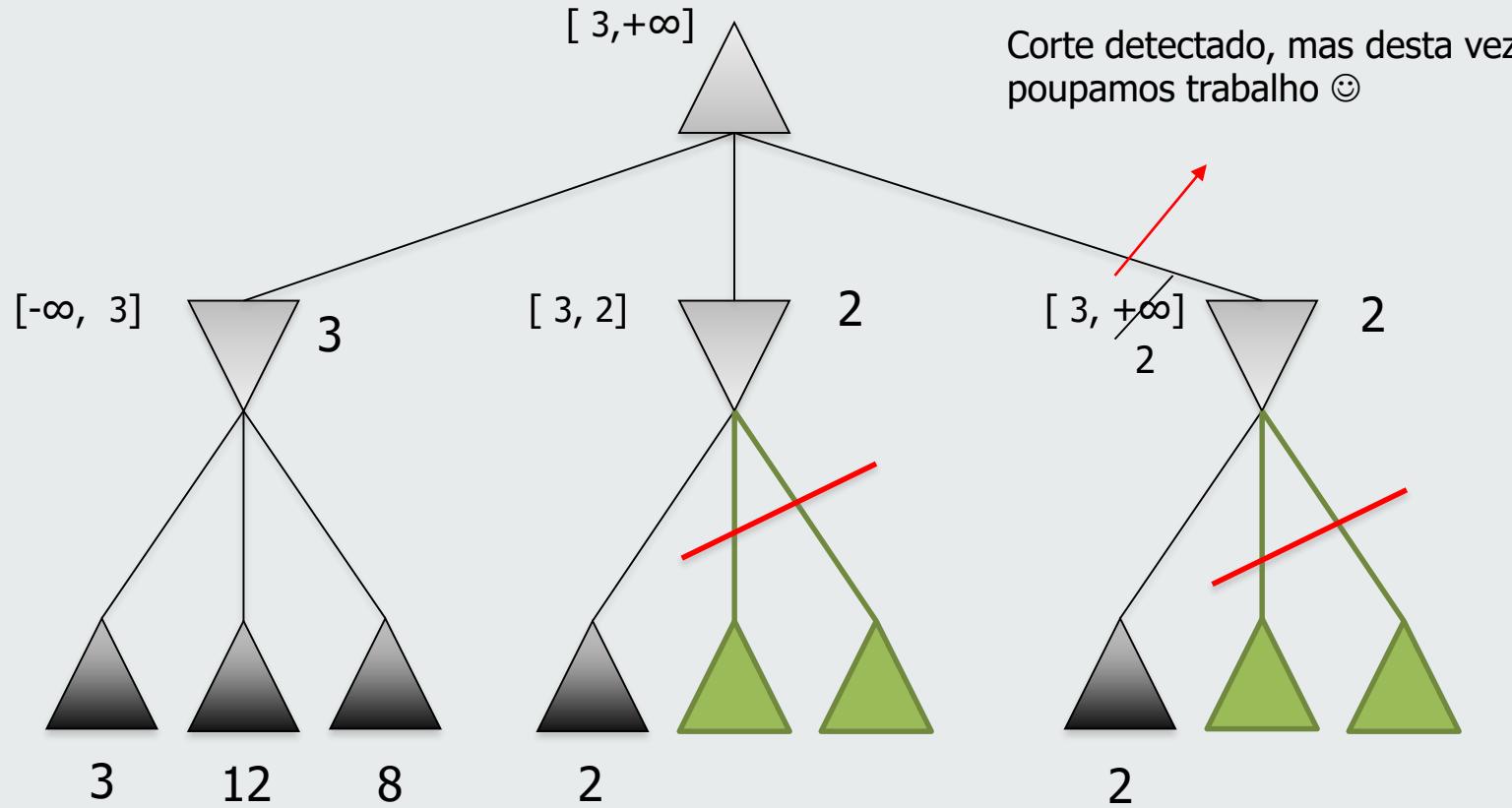
# Cortes $\alpha$ - $\beta$ : exemplo





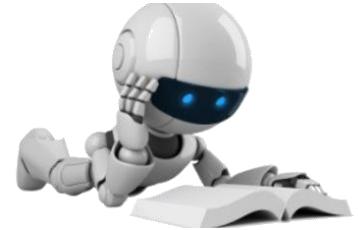
- Cortes **não** afectam resultado final (= MINIMAX)
- Eficiência dos cortes depende da **ordenação** dos sucessores
  - Por exemplo, no caso anterior, se em vez do nó com valor 14 tivesse aparecido o nó com valor 2, não havia necessidade de gerar os outros nós

# Cortes $\alpha$ - $\beta$ : exemplo

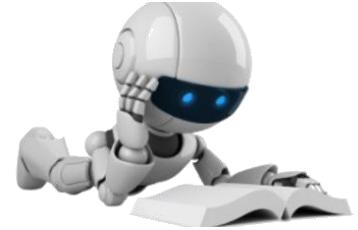




- Se os sucessores forem ordenados aleatoriamente
  - Conseguimos  $\sim O(b^{3/4m})$  em vez de  $O(b^m)$
- Com uma “ordenação perfeita” a complexidade temporal fica reduzida a  $O(b^{m/2})$ 
  - Nesta situação a procura  $\alpha\text{-}\beta$  consegue atingir o **dobro da profundidade** da procura minimax no mesmo período de tempo
- Mas qual a ordenação perfeita que maximiza os cortes?
  - Ideias?



- Qual a melhor ordenação de modo a optimizar os cortes  $\alpha$ - $\beta$ ?
  - Cortes dependem do valor de  $\alpha$  e  $\beta$
  - Portanto vamos escolher primeiro os sucessores que actualizem  $\alpha$  e  $\beta$  da maior maneira possível
  - Nó Max: visitar primeiro o sucessor com maior valor minimax
  - Nó Min: visitar primeiro o sucessor com menor valor minimax
- Na realidade só interessa o primeiro sucessor visitado
  - Ou existe corte logo no primeiro teste
    - Todos os restantes sucessores são cortados
  - Ou então não é possível haver corte



- Infelizmente, não sabemos o valor minimax de um nó antes de o visitar
- Então como escolher qual o melhor para visitar primeiro? (sem o visitar)
  - Estimar o seu valor com uma função de avaliação/heurística
  - Ou utilizar o seu valor minimax de iterações anteriores
    - Iterative Deepening :D
- Killer Move heuristic
  - Estratégia de visitar a melhor jogada primeiro
  - melhor jogada – designada de killer move

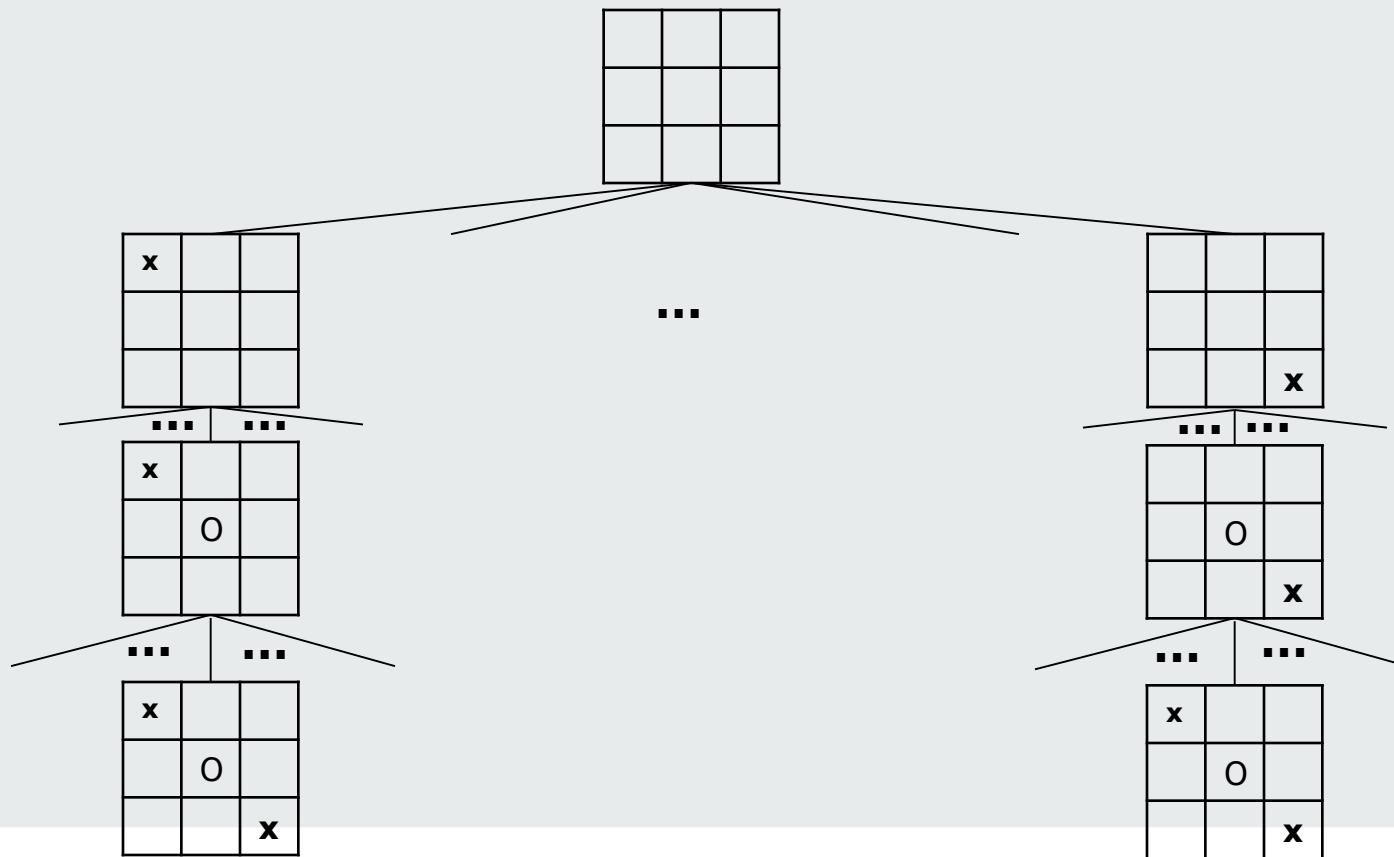


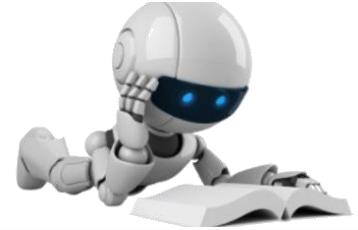
- Dica sobre cortes
  - Se quiserem poupar algum tempo, devem fazer os cortes antes de gerar todos os sucessores
  - Ou seja, gerar todos os sucessores e ordená-los não é necessariamente boa ideia
  - O ideal seria saber logo qual a melhor jogada (estimativa) sem ter de gerar nenhum sucessor
  - É possível fazê-lo
    - Deixo-vos isto como algo para pensar

# Transposições

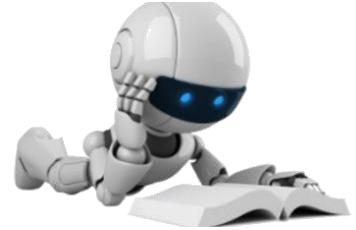


- Em jogos, ocorrem muitas vezes estados repetidos ou equivalentes (devido a permutações)



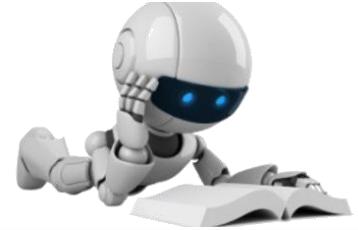


- Pode valer a pena guardar o estado e o seu valor minimax calculado numa hash table
  - Guardado na 1.<sup>a</sup> vez que é encontrado
  - Da próxima vez que o estado for encontrado em vez de calcular o seu valor
    - Usamos o valor guardado na hash table
- A esta tabela chamamos **tabela de Transposição**



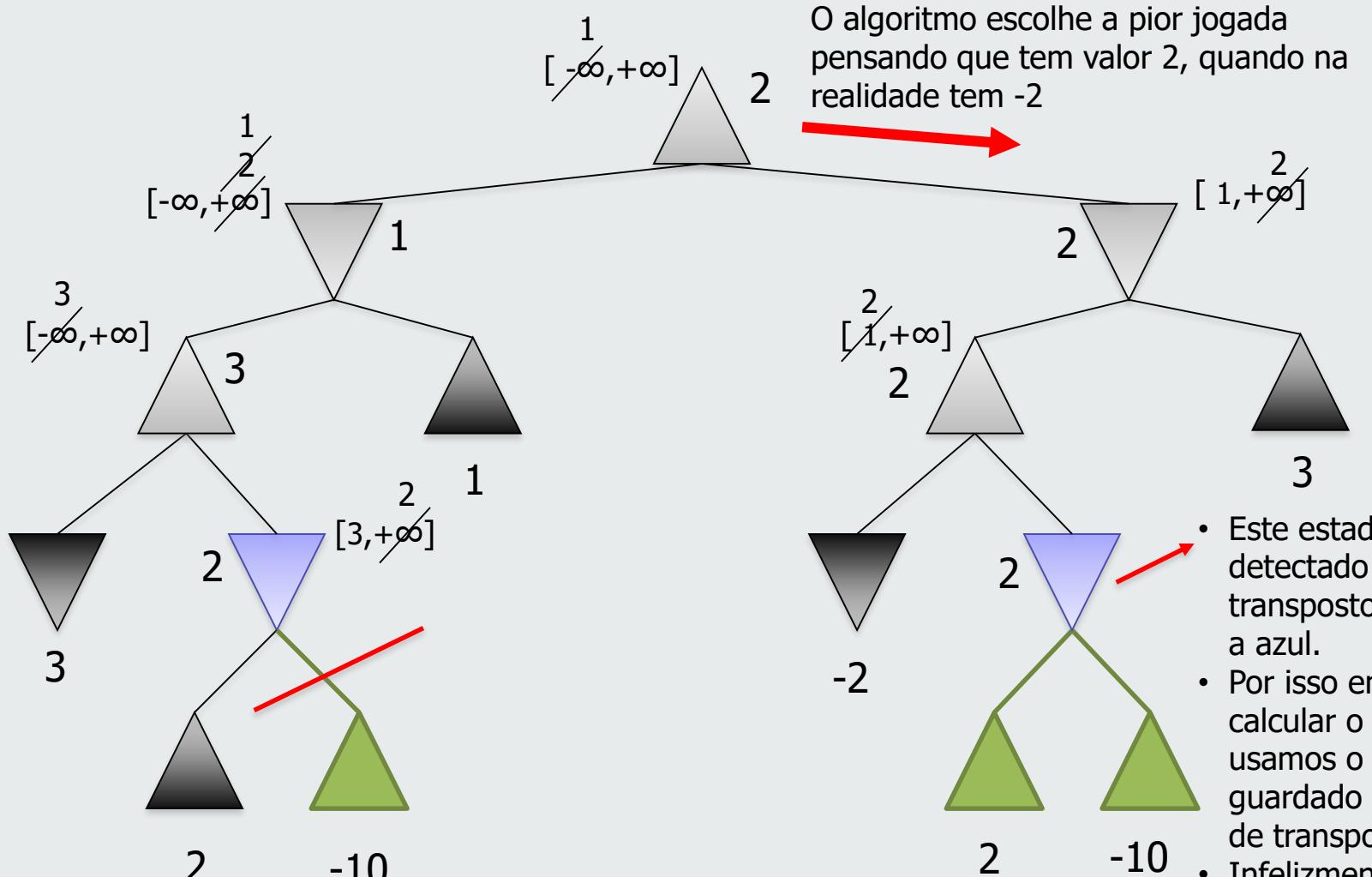
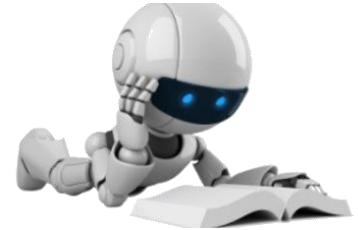
- Tabela de Transposição
  - Pode duplicar a profundidade alcançada no mesmo período de tempo!
  - No entanto, se são avaliados milhões de nós por segundo não é viável guardar tantos nós numa tabela
    - existem várias estratégias para determinar quais os estados a guardar

# Transposições



- Isto não está no livro, mas...
- Usar tabela de transposição é uma técnica muito interessante para alguns problemas
  - No entanto, é preciso **muito cuidado** quando usamos **tabela de transposição** juntamente com **cortes  $\alpha$ - $\beta$ !!!!!!**
  - Quando existe um nó cujos sucessores foram cortados, o valor minimax desse nó pode não ser o mais correcto
  - Isto pode levar a que se escolha uma má jogada sem nos apercebermos

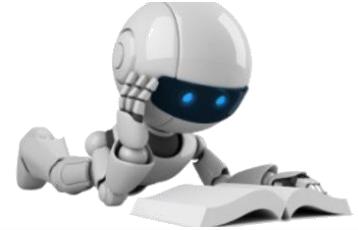
# Exemplo





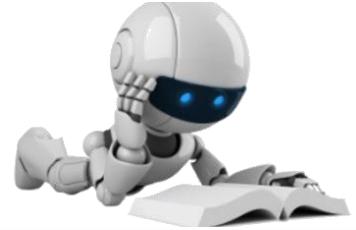
- Como resolver o problema?
  - Se um nó  $n$  teve algum dos seus sucessores **cortado**
    - **Não registrar** o nó e o seu valor minimax na tabela de transposição
- Isto diminui a eficácia da tabela de transposição quando se usam cortes  $\alpha$ - $\beta$
- Então é vale a pena usar transposição com cortes  $\alpha$ - $\beta$ ?
  - Ou apenas cortes  $\alpha$ - $\beta$ ?
  - Ou apenas tabela de transposição?
  - Deixo-vos isto para descobrirem

# Sumário



- Jogos: Conceitos Básicos
- Decisões óptimas em jogos
  - Estratégias óptimas e o Minimax
  - Estratégias óptimas com múltiplos jogadores
- Cortes  $\alpha$ - $\beta$
- Decisões imperfeitas em tempo real
- Jogos estocásticos
- Estado da arte em jogos

# Decisões imperfeitas em tempo real



Mesmo com cortes, a procura  $\alpha$ - $\beta$  tem de percorrer a árvore de jogo até à profundidade máxima

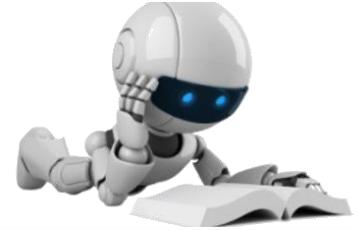
# Decisões imperfeitas em tempo real



Mesmo com cortes, a procura  $\alpha$ - $\beta$  tem de percorrer a árvore de jogo até à profundidade máxima

Decisões têm que ser tomadas em tempo real

- não é possível analisar toda a árvore
- é necessário cortar a árvore e acabar a procura mais cedo
- Mas como saber qual o valor de utilidade de um nó sem chegar ao fim da árvore?



# Função heurística de avaliação (**Eval**) !!!

# Decisões imperfeitas em tempo real



- Decisões têm que ser tomadas em tempo real
  - não é possível analisar toda a árvore
  - é necessário cortar a árvore e acabar a procura mais cedo
  - Mas como saber qual o valor de utilidade de um nó sem chegar ao fim da árvore?
- Função heurística de avaliação (**Eval**) devolve uma estimativa da utilidade do estado
  - Idealmente a ordenação resultante da função de avaliação é igual à da função de utilidade



- O desempenho de um jogo depende da qualidade da função de avaliação!
- E como “escolher” uma boa função de avaliação?
  - Para nós terminais
    - deve ser capaz de ordenar os estados terminais do mesmo modo que a função de utilidade
  - Para nós não terminais
    - deve estar fortemente correlacionada com as hipóteses reais de ganhar
  - O seu cálculo não pode ser demorado!!!!



- Tipicamente são uma soma linear de **características** do jogo ( $f$ ), associadas a diferentes pesos ( $w$ )

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

- Ex: Xadrez
  - rainha vale 9, bispo e cavaleiro 3, etc..
  - $w_1 = 9$  e  $f_1(s) = \text{nº de rainhas brancas}$
  - $w_2 = 3$  e  $f_2(s) = \text{nº de bispos}$
  - ...

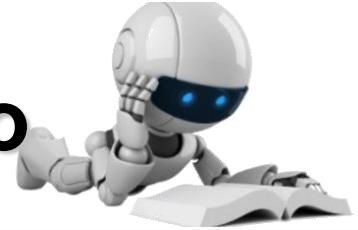
# Funções de Avaliação



- A soma de valores de diferentes características (features) é razoável embora seja uma assumpção demasiado forte pois assume que as características são independentes umas das outras;
- Actualmente a maioria dos programas usam combinações não lineares das “features”...



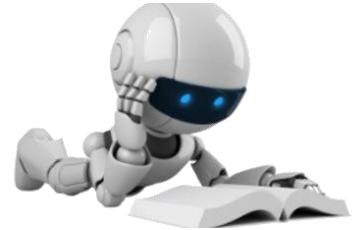
- Mas como cortar a procura?
- Função **teste-limite (cutoff-test)**
  - Recebe um estado e a profundidade desse estado
  - Decide se o estado é considerado final (mesmo não o sendo)
  - Deve retornar T se o estado recebido for um estado terminal



- Jogo pode ser visto como um problema de procura com:
  - Estado inicial
  - Jogador
  - Ações
  - Resultado
  - **Teste-limite (ou teste-corte)**
    - função que recebe um estado  $e$  e a profundidade  $d$  desse estado e determina se a procura deve parar nesse estado ou não
  - **Função-avaliação**
    - função que dado um estado terminal  $e$  e um jogador  $j$  retorna a **estimativa da utilidade esperada** a partir desse estado para o jogador (valor numérico)



# Algoritmo Minimax: revisitado



```
function Minimax-Decision(state) returns an action
    return arg maxa ∈ Actions(s) Min-Value(Result(state,a))
```

```
function Max-Value(state) returns a utility value
    if Terminal-Test(state) then return Utility(state)
```

```
    v ← -∞
    for each a in Actions(state) do
        v ← Max(v, Min-Value(Result(state,a)))
    return v
```

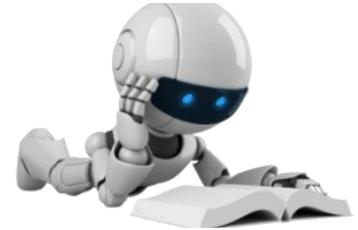
```
function Min-Value(state) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    v ← +∞
    for each a in Actions(state) do
        v ← Min(v, Max-Value(Result(state,a)))
    return v
```

# Algoritmo $\alpha$ - $\beta$ com teste-límite



```
function H-Alfa-Beta-Search(state) returns an action
    v  $\leftarrow$  Max-Value(state, 0,  $-\infty$ ,  $+\infty$ )
    return the action in Actions(state) with value v
```

```
function Max-Value(state, depth,  $\alpha$ ,  $\beta$ ) returns a utility value
    if Cutoff-test(state, depth) then return Eval(state)
    v  $\leftarrow$   $-\infty$ 
    for each a in Actions(state) do
        v  $\leftarrow$  Max(v, Min-Value(Result(state, a), depth+1,  $\alpha$ ,  $\beta$ ))
        if v  $\geq \beta$  then return v
         $\alpha \leftarrow \text{Max}(\alpha, v)$ 
    return v
```



- Problema da **aquiescência (estados inactivos, parados...)**
  - Estados não aquiescentes no limite devem ser expandidos até que sejam gerados estados aquiescentes
    - ex: (por exemplo, um estado em que não foram feitas capturas)
  - A esta procura adicional, chama-se **procura aquiescente**
  - Por vezes é aplicada apenas a algum tipo de jogadas
    - Como capturas
    - rapidamente se pode perceber se são boas ou más

# Teste Limite



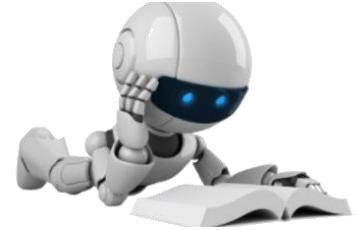
(a) White to move



(b) White to move

**Figure 5.8** Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

# Teste Limite



- Problema do efeito de **horizonte**
  - Procura com limite coloca eventuais problemas futuros para além do horizonte
  - ex: um movimento adversário que vai ter consequências desastrosas, mas que o jogador consegue adiar
    - O jogador consegue adiar, mas não evitar
    - A consequência é adiada para além do horizonte
    - Não se consegue perceber que esta é uma má jogada, por causa do limite

# Teste Limite



The horizon effect. With Black to move, the black bishop is surely doomed. But Black can forestall that event by checking the white king with its pawns, forcing the king to capture the pawns. This pushes the inevitable loss of the bishop over the horizon, and thus the pawn sacrifices are seen by the search algorithm as good moves rather than bad ones.



- Extensões singulares permitem mitigar o problema do **efeito de horizonte**.
  - Quando é encontrada uma jogada considerada claramente melhor que as jogadas restantes para a mesma posição
  - Essa jogada é registada como jogada singular
  - **Aumenta-se o limite de procura** para os sucessores da jogada singular.
- A profundidade de árvore aumenta
  - Mas como existem poucas extensões singulares
  - Não são adicionados muitos nós à árvore

# Cortes Progressivos



- Até agora vimos 2 tipos de corte
  - Cortes alfa-beta (não influenciam o resultado)
  - Cortes limite
- Existe um 3.<sup>º</sup> tipo de corte
  - Cortes progressivos
    - Cortar imediatamente algumas jogadas por cada posição analisada



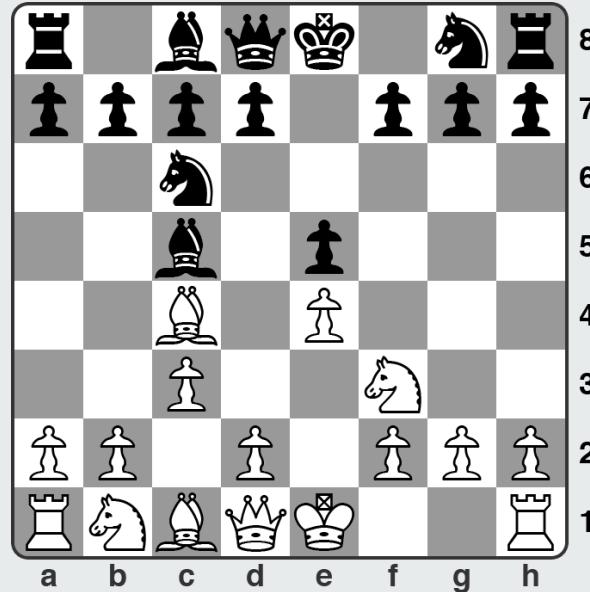
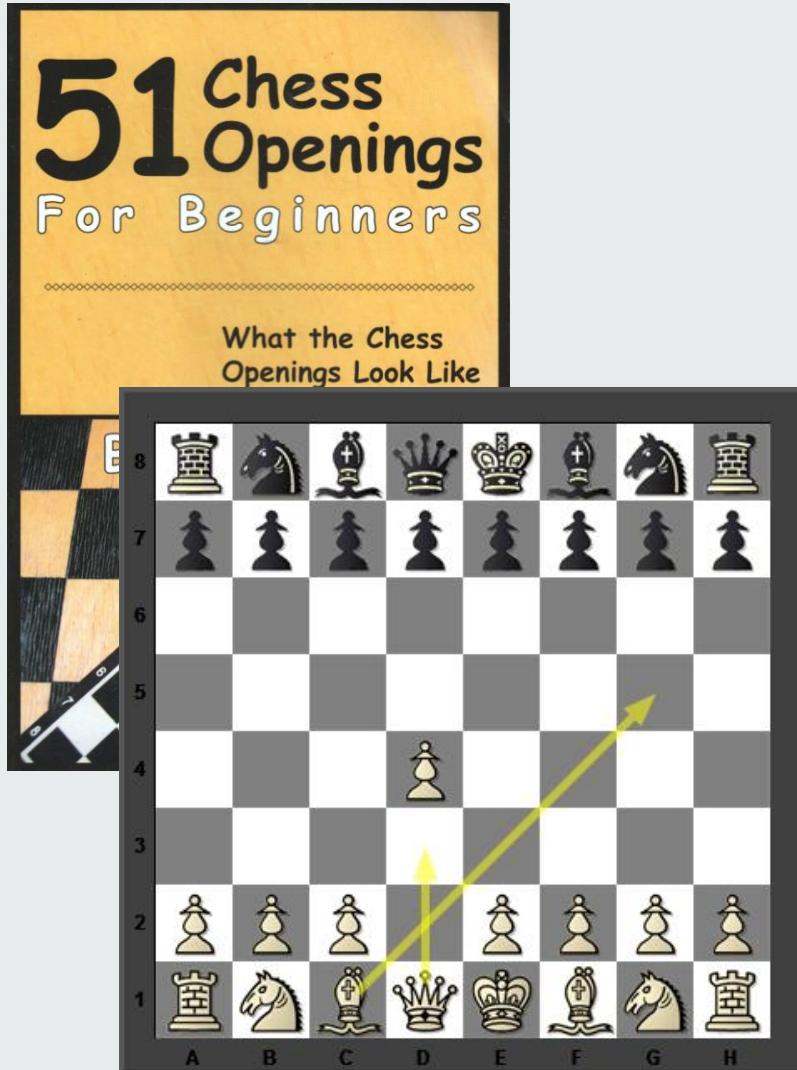
- Procura em banda (**beam search**)
  - Para cada posição considerar as  $n$  melhores jogadas (como os humanos fazem!)
    - Em vez de considerar todas as jogadas
    - Qualidade da jogada estimada com a função de avaliação
    - Perigoso:
      - Nada nos garante que a melhor jogada real não seja imediatamente cortada



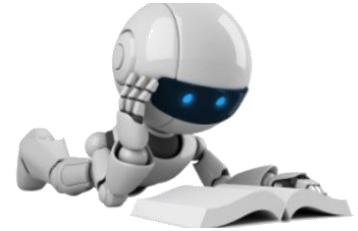
- **ProbCut (probabilistic cut)**
  - Tenta diminuir a probabilidade da melhor jogada real ser cortada
  - A procura alfa-beta corta nós que temos a certeza de ter valor fora da janela ( $\alpha$ - $\beta$ )
  - ProbCut tb corta nós que provavelmente têm valor fora da janela ( $\alpha$ - $\beta$ )
    - Probabilidade calculada fazendo uma procura pouco profunda para calcular o valor minimax  $v$  de um nó
    - Usa-se experiência de procura anteriores para determinar a probabilidade de um valor  $v$  à profundidade  $p$  estar fora da janela ( $\alpha$ - $\beta$ )
    - Se a probabilidade for maior que um threshold, o nó é cortado



# Mas... aberturas e finais



# Tabelas de aberturas e finais



- Muitos programas para jogar jogos usam tabelas de aberturas e finais
- Em vez de usar procura para determinar melhor jogada
  - Usa-se tabela com melhores jogadas iniciais
    - Obtidas por conhecimento perito
    - Ou por estatísticas
      - Quais as jogadas iniciais que levam mais vezes a vitórias?
  - O mesmo para o fim do jogo
    - Tabelas de estados perto do fim, a partir dos quais a jogada óptima está memorizada

# Sumário

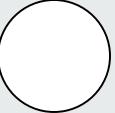


- Jogos: Conceitos Básicos
- Decisões óptimas em jogos
  - Estratégias óptimas e o Minimax
  - Estratégias óptimas com múltiplos jogadores
- Cortes  $\alpha$ - $\beta$
- Decisões imperfeitas em tempo real
- **Jogos estocásticos**
- Estado da arte em jogos



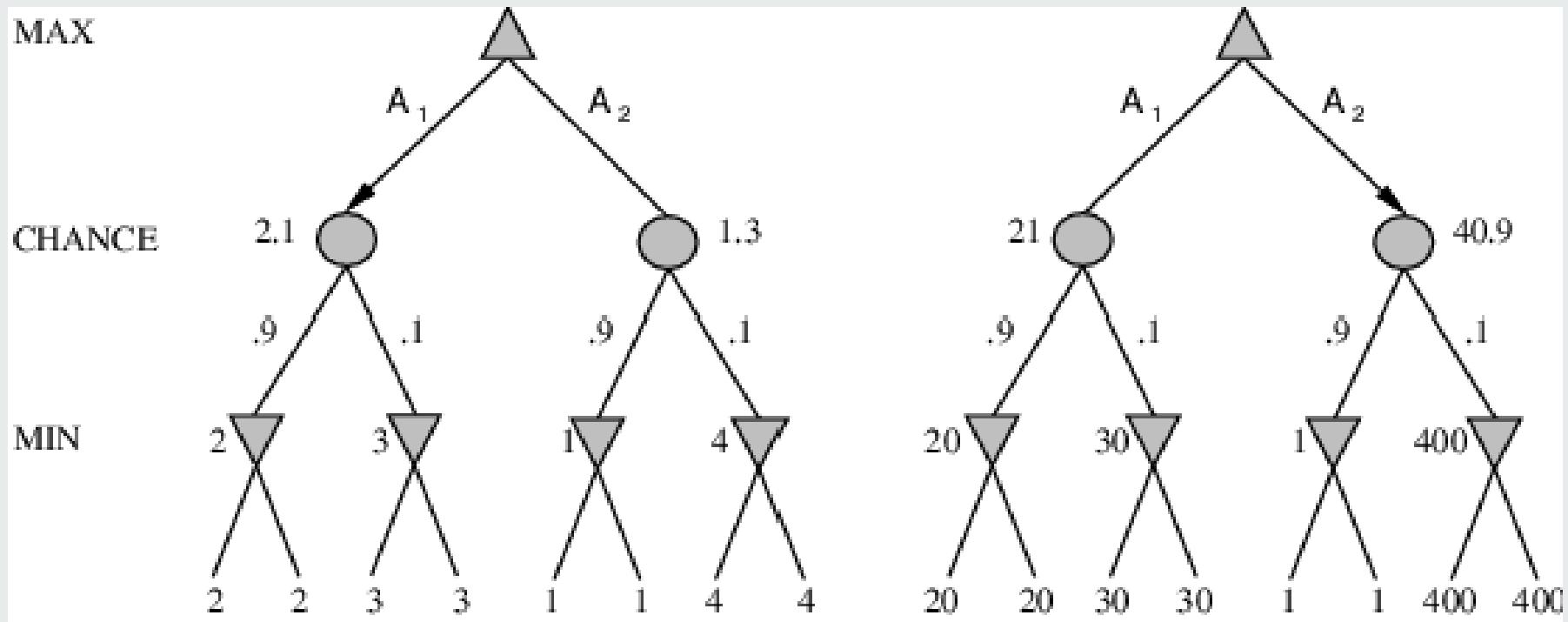
- Jogos com elemento sorte
  - Ex: Backgammon
    - No inicio do turno de um jogador são lançados 2 dados para decidir que acções podem ser feitas.
    - Nunca sabemos ao certo quais as jogadas que um jogador adversário vai fazer
      - Porque dependem do lançamento dos dados
  - Como representar a árvore de jogo?

# Jogos estocásticos

- Árvore de jogo com nós sorte  para além dos nós MIN e MAX
- Cada ramo de um nó sorte representa um resultado diferente nos dados
  - está associado uma probabilidade

Expectiminimax( $n$ ) =

$$\left\{ \begin{array}{ll} \text{Função-utilidade}(n,\text{max}) & \text{se } n \text{ é terminal} \\ \max_{s \in \text{sucessores}(n)} \text{Expectiminimax}(s) & \text{se } n \text{ é nó MAX} \\ \min_{s \in \text{sucessores}(n)} \text{Expectiminimax}(s) & \text{se } n \text{ é nó MIN} \\ \sum_r P(r) \text{Expectiminimax}(\text{Result}(n,r)) & \text{se } n \text{ é nó SORTE} \end{array} \right.$$

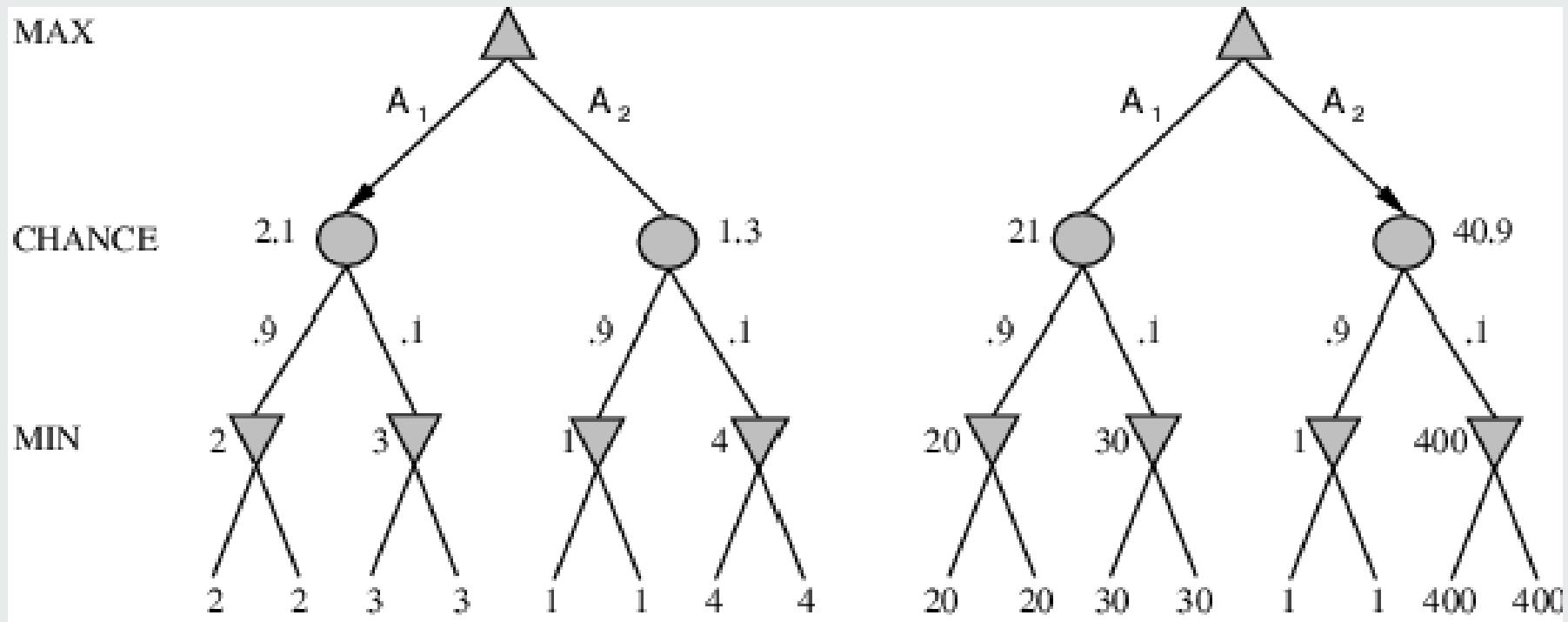


$$2.1 = 0.9 \cdot 2 + 0.1 \cdot 3$$

$$1.3 = 0.9 \cdot 1 + 0.1 \cdot 4$$



- Funções de avaliação para jogos estocásticos
  - É preciso ter um cuidado adicional
  - Em jogos sem sorte
    - ordenação resultante da função de avaliação deve ser igual à da função de utilidade
  - Em jogos estocásticos
    - isto não é suficiente



- Atenção: alteração de valores das folhas mantendo a mesma ordem relativa de valores resulta em decisões diferentes.



- Para lidar com o problema
  - Função de avaliação deve ser:
    - Transformação linear positiva:
      - Da probabilidade de vencer a partir de uma posição
      - Ou da utilidade esperada da posição

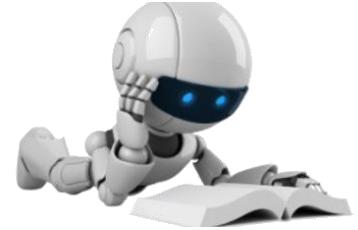


- Complexidade Temporal
  - Se não tivéssemos lançamento de dados
    - $O(b^m)$
  - Mas se considerarmos lançamento de dados
    - $O(b^m n^m)$
    - $n$  – número de lançamentos distintos de dados
  - Ou seja, é um problema mt mais difícil
    - Em jogos estocásticos não se consegue normalmente atingir uma profundidade elevada



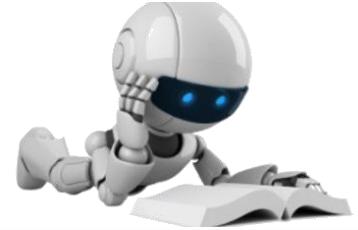
- É possível usar cortes alfa-beta
  - Cortes nós max e min processam-se da mesma maneira
  - Cortes nó sorte
    - Calcular limites (máximo e mínimo) para valor do nó
    - Usar limites para decidir corte
  - Alternativamente
    - Avaliar uma posição usando uma simulação de Monte Carlo
      - Correr o algoritmo para jogar milhares de jogos a partir do estado inicial contra si próprio utilizando valores aleatórios para lançamentos de dados
      - A percentagem de vitória de uma posição é uma boa estimativa do valor da posição
      - Usar valor estimado para decidir cortes

# Sumário

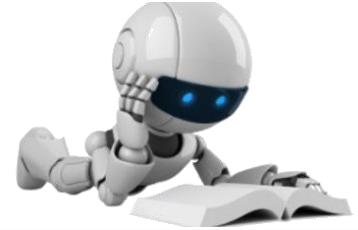


- Jogos: Conceitos Básicos
- Decisões óptimas em jogos
  - Estratégias óptimas e o Minimax
  - Estratégias óptimas com múltiplos jogadores
- Cortes  $\alpha$ - $\beta$
- Decisões imperfeitas em tempo real
- Jogos estocásticos
- Estado da arte em jogos

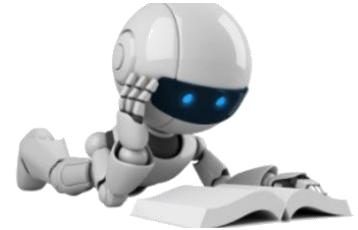
# Estado da Arte



- **Xadrez:**
  - Deep Blue derrotou campeão do mundo Garry Kasparov em 1997.
    - Computador paralelo com 30 processadores a fazer uma procura alfa-beta.
    - 480 processadores específicos para xadrez que fazem geração de jogadas, ordenação de jogadas, e avaliação de jogadas para os últimos níveis da árvore.
    - Procura cerca de 200 milhões de nós por segundo.
    - Consegue chegar a uma profundidade 14. (vê 14 jogadas à frente).



- **Xadrez:**
  - Deep Blue
    - Utiliza extensões singulares para caminhos interessantes de jogadas forçadas/forçantes (i.e. obrigam o oponente a fazer uma jogada particular).
      - Em alguns casos de jogadas singulares, o algoritmo chega à profundidade 40.
    - Utiliza uma função de avaliação muito sofisticada.
      - Baseada em cerca de “8000 features”
      - Muitas das quais descrevem padrões muito específicos
    - Tabela de aberturas com 4000 posições
    - Tabela de finais
      - Todos os estados possíveis com 5 peças
      - Muitos estados com 6 peças

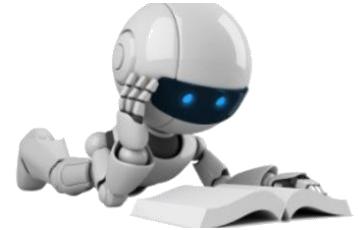


- **Xadrez:**
  - Hydra – sucessor do Deep Blue
    - Hardware com performance semelhante ao Deep Blue
      - 200 milhões de nó por segundo
    - No entanto consegue chegar a 18 níveis de profundidade
    - Utilização agressiva da heurística de jogada nula e cortes progressivos
  - Heurística de jogada nula
    - É feita uma procura de baixa profundidade assumindo que o oponente joga duas vezes seguidas no inicio
    - O valor obtido serve como limite inferior do valor da posição
    - Pode ser usado para decidir cortes alfa-beta sem termos de percorrer a árvore toda
  - Tb é usado “futility pruning”
    - Permite descobrir mais cedo quando irá existir um corte do tipo beta

# Estado da Arte



- **Damas:** Chinook derrotou o campeão do mundo (durante 40 anos) Marion Tinsley in 1994.
- Uso de uma base de dados pré-processada que define uma jogada perfeita para todas as posições envolvendo no máximo 8 peças, num total de 444 biliões de posições.



- **Othello:** campeões humanos recusam-se a competir com computadores, que são muito bons!!!!
- **Go:** campeões humanos recusam-se a competir com computadores, que são muito fracos.
  - Neste jogo,  $b > 300$ .
  - Tb é difícil construir funções de avaliação
    - Só no fim é que se consegue perceber bem a real utilidade de um estado
  - Por isso a aplicação de procura alfa-beta não é muito bem sucedida.
  - A maioria dos programas existentes usa padrões de conhecimento para sugerir jogadas hipotéticas.
  - Um dos melhores programas utiliza simulações de Monte Carlo.