



# TEORIA DE JOGOS

**Inteligência Artificial**

Joaquim Filipe

# INTRODUÇÃO

- Primeiros estudos de teoria de jogos datam de 1713, em que Charles Waldegrave sugere uma estratégia minimax para um jogo de cartas entre 2 pessoas.
- Pode haver jogos de 2 adversários ou de muitos adversários.
- Há diversos tipos de jogos e podem definir-se partições do espaço de jogos usando diversas características.

# INTRODUÇÃO (CONT.)

- A área de estudo de “Teoria de Jogos” é estabelecida apenas em 1928 na sequência de um trabalho apresentado por John von Neumann.
- Em 1944 publica o livro *Theory of Games and Economic Behavior* em co-autoria com Oskar Morgenstern.



John von Neumann



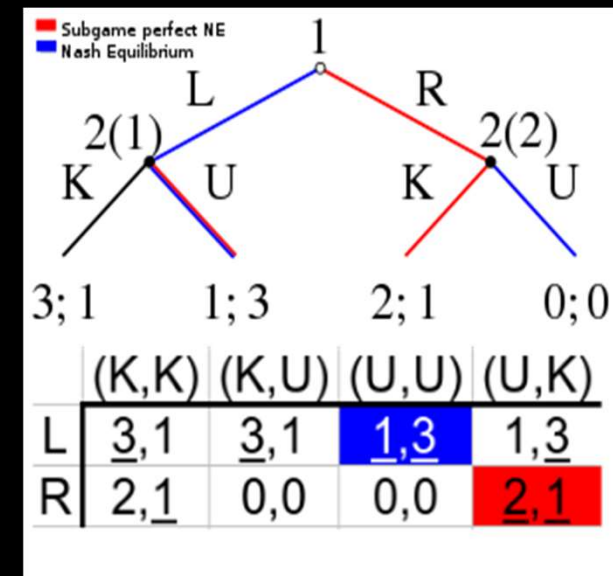
Oskar Morgenstern

# TIPOS DE JOGOS: SEQUENCIAIS VS. SIMULTÂNEOS

- Jogos sequenciais (ou dinâmicos) são jogos em que cada jogador tem conhecimento do lance do seu antecessor.
  - Um subconjunto importante é o dos jogos com informação perfeita: em que cada jogador conhece todas as jogadas feitas por todos os outros jogadores.
    - Exemplos: xadrez, damas, go
- Jogos simultâneos são jogos onde os lances são executados simultaneamente, ou pelo menos os jogadores desconhecem previamente as ações dos seus adversários (tornando-os efetivamente simultâneos).
  - Um jogo simultâneo pode ser um jogo de informação perfeita?

# REPRESENTAÇÃO FORMAL DE JOGOS

- A formalização dos jogos pode fazer-se de várias maneiras:
  - “Extensiva” – Na forma de grafo
  - “Normal” – na forma de matriz de pagamentos
    - A forma normal não permite representar jogos sequenciais
  - “Função característica” – relaciona uma função de utilidade com uma dada população (von Neumann).



# TIPOS DE JOGOS: SIMÉTRICOS VS. ASSIMÉTRICOS

- *Simétrico* se os resultados não dependem do jogador mas apenas da estratégia.
  - Exemplo: Dilema do Prisioneiro

A \ B	N	T
N	1,1	3,0
T	0,3	2,2

Se um dos prisioneiros trair e o outro não, o que trai sai em liberdade e o que não trai apanha 3 anos de prisão;  
Se traírem os dois, apanham 2 ano cada;  
Se nenhum trair, apanham 1 anos cada.

# TIPOS DE JOGOS

## SOMA ZERO VS. SOMA NÃO-ZERO:

- No jogo de soma-zero o valor total dos jogadores, para qualquer combinação de estratégias, é zero, i.e. o que um jogador ganha é perdido pelo(s) outro(s).
  - Exemplos:
    - Poker : o vencedor recebe exatamente a soma das perdas de seus oponentes.
    - A maioria dos jogos clássicos de tabuleiro é de soma zero, incluindo o Go e o Xadrez.



# TIPOS DE JOGOS COOPERATIVOS OU NÃO

- Nos jogos não-cooperativos não existem formas positivas de suportar alianças (e.g. contratos). A coordenação é estabelecida essencialmente através de ameaças credíveis.
  - Exemplo:
    - Dilema do prisioneiro
    - Jogo da galinha
- Nos jogos cooperativos estabelecem-se alianças baseadas em acordos ou contratos.
  - Exemplo:
    - Futebol
    - Tomada de decisão por consenso
    - Revolução



# EQUILIBRIO DE NASH

- Em teoria de jogos, o equilíbrio de Nash é uma solução para um jogo não-cooperativo envolvendo 2 ou mais jogadores.
- Na situação de equilíbrio de Nash
  - Cada jogador toma a melhor decisão possível para si próprio dada a decisão que o outro tomou, desde que o outro não mude.
  - Nenhum jogador poderá melhorar a sua estratégia unilateralmente.
- Exemplo: a situação seguinte tem 2 pontos de equilíbrio de Nash:

A \ B	Conduz à esquerda	Conduz à direita
Conduz à esquerda	10,10	0,0
Conduz à direita	0,0	10,10

Se A conduz à esquerda o B também deve conduzir à esquerda; e vice-versa.

Quantos pontos de equilíbrio de Nash existem no dilema do prisioneiro?



John F. Nash

# TIPOS DE JOGOS: COMBINATÓRIOS

- Designam-se por combinatórios os jogos em que o encontrar a estratégia ótima padece do problema da explosão combinatória.
  - Exemplos: xadrez e go
- Há uma sub-área da teoria de jogos dedicadas a este tipo de jogos, decorrentes da teoria da complexidade computacional: *complexidade de jogos*.
  - A complexidade de alguns jogos conhecidos está descrita de forma tabular em:  
[https://en.wikipedia.org/wiki/Game\\_complexity](https://en.wikipedia.org/wiki/Game_complexity)

# TIPOS DE JOGOS QUE SERÃO ANALISADOS

- Vamos centrar a atenção em:
- Jogos de 2 adversários
- Jogos simétricos
- Jogos sequenciais
- Jogos de soma nula, não cooperativos
- Jogos combinatórios

# CONCEPTUALIZAÇÃO

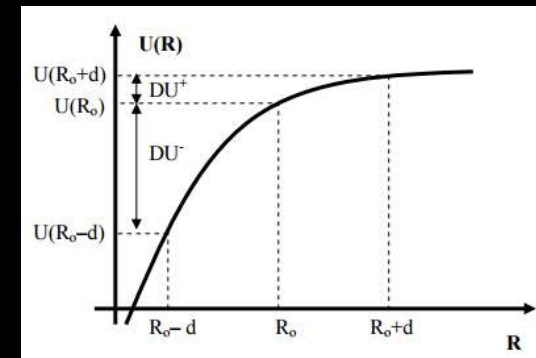
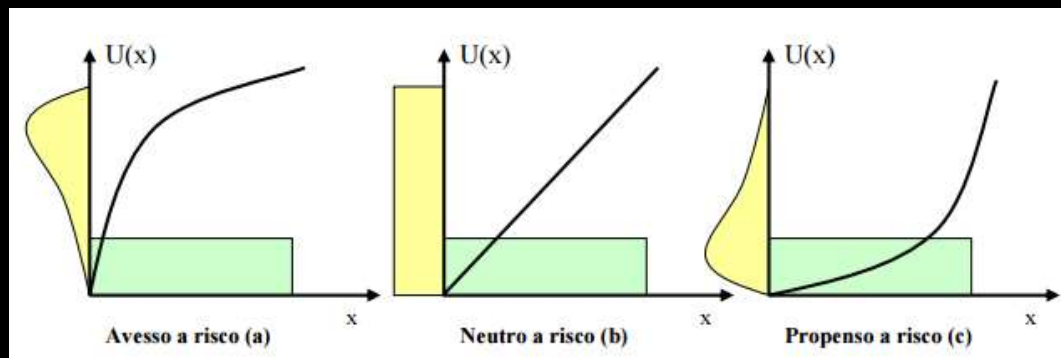
- O jogo do xadrez foi o primeiro a ser abordado formalmente: 1950, por Claude Shannon e Alan Turing.
- Foco de atenção da IA em: jogos sequenciais simétricos com informação perfeita, de 2 adversários e de soma nula, de natureza combinatória.
  - A incerteza nos jogos surge não por falta de informação ou por informação inexata mas por falta de tempo para explorar a informação disponível.
  - Pode conceptualizar-se o jogo como um problema de procura em espaço de estados, em que existem agentes hostis que procuram diminuir o nosso próprio valor.

# HEURISTICAS E FUNÇÕES DE AVALIAÇÃO

- Para facilitar a identificação da melhor jogada usam-se heurísticas.
- As heurísticas auxiliam a exploração das árvores que descrevem o espaço de estados mediante:
  - técnicas de **corte da árvore (pruning)**
  - **funções de avaliação (evaluation functions)**
- Isso permite calcular a utilidade de um estado sem explorar toda a subárvore respectiva.

# FUNÇÕES DE UTILIDADE

- As heurísticas estão geralmente ligadas a funções de utilidade, e são diferentes de jogador para jogador.



- O princípio da “utilidade esperada”, estabelecido por John von Neumann e Oskar Morgenstern, permite valorar a distribuição de probabilidade dos possíveis resultados de uma decisão e assim estabelecer a preferência entre as decisões associadas a estas distribuições de probabilidade.

# JOGOS DE 2 PESSOAS

- Componentes da procura:
  - Estado inicial
  - Conjunto de operadores
  - Teste terminal
  - Função de utilidade
- **MAX** vs. **MIN**
  - O MAX ▲ tenta maximizar o valor da sua função de utilidade; representa o jogador que toma a decisão.
  - O MIN ▼ representa o adversário. Presume-se que o MIN tem uma função de utilidade simétrica do MAX e portanto ao tentar maximizar a sua própria utilidade irá minimizar a do MAX (o que pode ser irrealista).

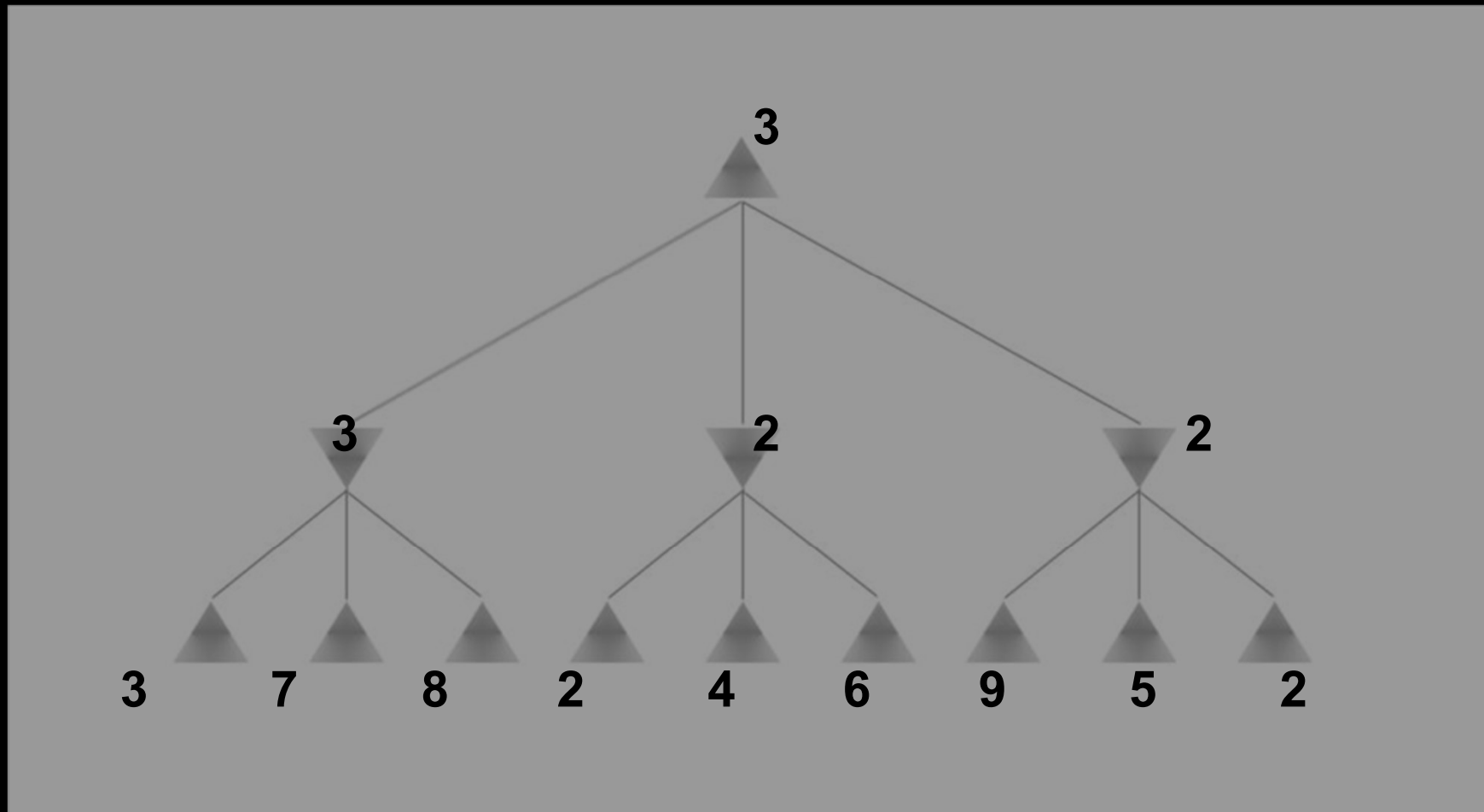


# O ALGORITMO MINIMAX

O algoritmo é desenhado para determinar a estratégia ótima para o MAX.

1. Gerar toda a árvore de procura desde o nó inicial até aos nós terminais.
2. Aplicar a função de utilidade a cada nó terminal para determinar o respectivo valor.
3. Usar a utilidade dos nós terminais para determinar através de um processo de *backup* a utilidade dos nós no nível imediatamente acima na árvore de procura:
  1. Se for um lance do MIN o valor calculado é o mínimo dos nós do nível inferior;
  2. se for o MAX a jogar, o valor calculado é o máximo.
4. Continuar a usar o processo de *backup* um nível de cada vez até atingir o nó inicial.
5. Tendo chegado ao nó inicial, realizar o lance correspondente ao valor determinado para esse nó.

# EXEMPLO DE APLICAÇÃO DO MINIMAX



# PSEUDOCÓDIGO DO MINIMAX

**function** minimax(node, depth, maximizingPlayer) **is**

**if** depth = 0 **or** node is a terminal node **then**

**return** the heuristic value of node

**if** maximizingPlayer **then**

value :=  $-\infty$

**for each** child of node **do**

value := max(value, minimax(child, depth - 1, FALSE))

**return** value

**else** (\* minimizing player \*)

value :=  $+\infty$

**for each** child of node **do**

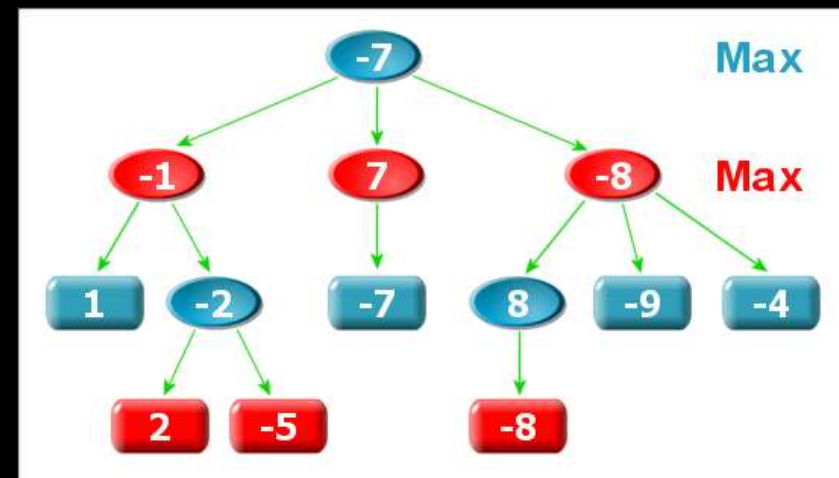
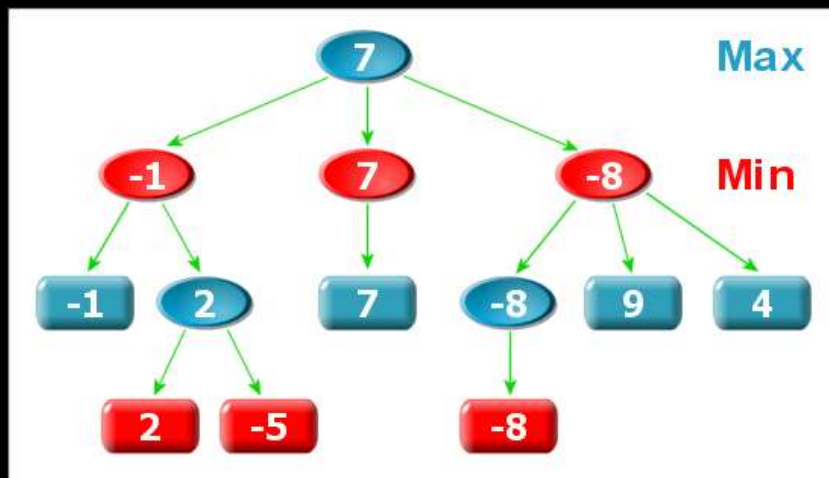
value := min(value, minimax(child, depth - 1, TRUE))

**return** value

depth: Em termos práticos poderá definir-se um limiar de profundidade máximo para o minimax.

# NEGAMAX

- Não é mais do que outra formulação do MINIMAX em que se passa a procurar sempre apenas o máximo, mas se troca o sinal em cada nível, após o backup.
- A ideia é tirar partido de que  $\max(a, b) = -\min(-a, -b)$



# PSEUDO-CÓDIGO DO NEGAMAX

```

;;; argumentos: nó n, profundidade d, cor c
;;; b = ramificação (número de sucessores)
function negamax (n, d, c &aux bestValue)
  if d = 0 ou n é terminal
    return c * valor heurístico de n
  bestValue :=  $-\infty$ 
  para cada sucessor de n ( $n_1, \dots, n_k, \dots, n_b$ )
    bestValue := max(bestValue, -negamax( $n_k$ , d-1, -c) )
  return bestValue

```

```

Valor do nó inicial
rootNegamaxValue := negamax( rootNode, d, -1)

```

# COMENTÁRIOS AO ALGORITMO MINIMAX

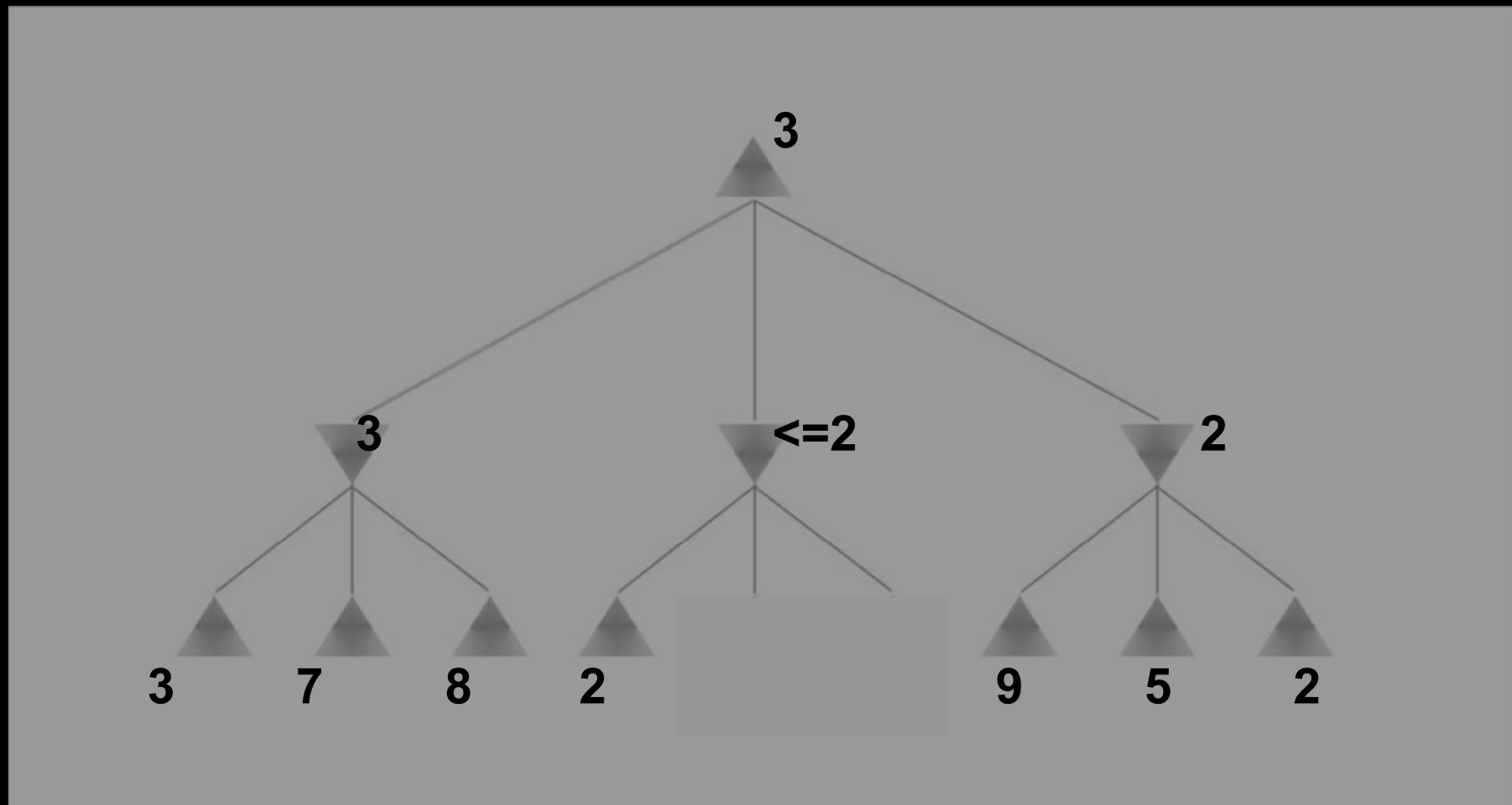
- Se a profundidade máxima da árvore for ***m*** e em cada ponto houver ***b*** lances possíveis (factor de ramificação), então a complexidade (temporal) do minimax é  $O(b^m)$ .
- O algoritmo pode ser implementado essencialmente como *procura em profundidade primeiro* (depth-first) embora usando recursividade em vez de uma fila de nós por isso os requisitos de espaço são lineares em ***b*** e ***m***.
- Para problemas reais o custo de tempo é geralmente inaceitável, mas este algoritmo serve de base a outros métodos mais realistas, bem como de suporte à análise matemática de jogos.

# DECISÕES IMPERFEITAS

- Shannon propôs, em 1950, o corte da árvore de procura num nível acima dos nós terminais, e a utilização de uma função de avaliação para determinar a utilidade das folhas dessa árvore.
- Funções de avaliação:
  - Lineares
    - $F = w_1f_1 + w_2f_2 + \dots + w_nf_n$   
Exemplo: xadrez –  $w$  importância de cada característica;  $f$  – valor dessa característica. Os  $w_i$  poderiam ser o valor dos tipos de peças e  $f_i$  o número de peças desse tipo.
  - Não-Lineares: e.g. Redes neuronais



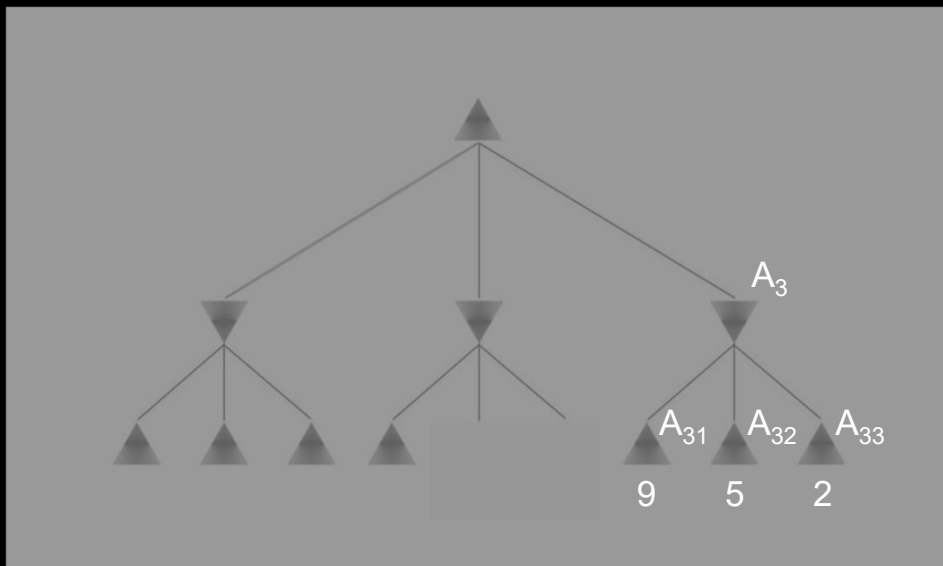
# EXEMPLO DE CORTE ALFA-BETA



# ORDEM DE ANÁLISE DOS NÓS E CORTES ALFA-BETA

- A eficácia do Alfa-Beta depende da ordem com que os sucessores são examinados. Na figura anterior verifica-se que não foi possível eliminar  $A_3$  porque os nós  $A_{31}$  e  $A_{32}$  foram gerados antes do  $A_{33}$ .

Isto sugere que seria melhor examinar primeiro os sucessores que tiverem maior probabilidade de ser melhores.



# CORTES ALFA-BETA

- Objectivo: Calcular o minimax correto sem analisar todos os nós da árvore.
- Princípio geral: Considere-se um nó ***n*** tal que MAX tem a opção de jogar para esse nó. Se MAX detetar a possibilidade de escolher um nó ***m*** melhor que ***n***, ou no nível do nó pai de ***n*** ou em algum nível acima desse, então ***n*** nunca será atingido durante o jogo. Por consequência, assim que se souber o suficiente acerca de ***n*** (explorando alguns dos seus descendentes) para validar esta conclusão, pode cortar-se o nó ***n***.

# TERMINOLOGIA

- O limite inferior do valor de um nó designa-se por alfa ( $\alpha$ ).
- O limite superior do valor de um nó designa-se por beta ( $\beta$ ).
- O valor real do nó está no intervalo  $[\alpha, \beta]$ .
- Os valores  $\alpha$  dos nós MAX nunca podem decrescer.
- Os valores  $\beta$  dos nós MIN nunca podem crescer.
- Estes considerandos permitem enunciar regras de descontinuação da procura (corte) – no próximo slide.

# REGRAS DE CORTE

- A procura pode ser descontinuada abaixo de qualquer nó MIN com um valor  $\beta \leq \alpha$  de qualquer dos seus antecessores MAX (corte  $\alpha$ ).
  - ✓ Nesse caso, o valor retornado por esse nó MIN pode ser o seu valor  $\beta$  corrente, apesar de este valor poder não ser o mesmo que se obtém com o minimax.
- A procura pode ser descontinuada abaixo de qualquer nó MAX com um valor  $\alpha \geq \beta$  de qualquer dos seus antecessores MIN (corte  $\beta$ ).
  - ✓ Nesse caso, o valor retornado por esse nó MAX pode ser o seu valor  $\alpha$  corrente.

# ALGORITMO MINIMAX COM CORTES ALFA-BETA

- Notar que a procura alfabetada é do tipo *depth-first*, pelo que em cada instante apenas é necessário considerar os nós ao longo de um ramo da árvore de procura.
- Seja  $\alpha$  o valor da melhor escolha encontrada até ao momento, ao longo do ramo corrente, para MAX.
- Seja  $\beta$  o valor da melhor escolha encontrada até ao momento, ao longo do ramo corrente, para MIN.
- O Alfa-Beta atualiza o valor de  $\alpha$  e de  $\beta$  ao longo da procura e corta uma subárvore (terminando uma chamada recursiva) logo que se sabe ser pior que os valores correntes de  $\alpha$  e de  $\beta$ .

# FAIL-SOFT VS. FAIL-HARD

- A versão **Fail-soft** do alfabeta permite que sejam devolvidos valores fora do intervalo  $[a, \beta]$
- A versão **Fail-hard** é uma variante do pseudo-código anterior em que o valor devolvido é limitado ao intervalo  $[a, \beta]$  definido em parâmetros.
- Isso corresponde a, no caso de haver um sucessor que dê origem a um corte (valor fora do intervalo  $[a, \beta]$ ) devolver o valor limite do intervalo.



# PSEUDOCÓDIGO DO ALFABETA (FAIL-SOFT)

## AlfaBeta( $n$ ; $\alpha$ ; $\beta$ )

1. Se  $n$  no limite de profundidade  $d$ , devolve AlfaBeta( $n$ )= $f(n)$ , caso contrário calcula os sucessores  $n_1, \dots, n_k, \dots, n_b$  (por ordem), faz  $k=1$  e, se  $n$  é um nó MAX, vai p/2 c.c. vai p/ ii.
2.  $v = -\infty$
3.  $v \leftarrow \max[v, \text{AlfaBeta}(n_k; \alpha; \beta)]$  ←
4.  $\alpha \leftarrow \max[v, \alpha]$
5. Se  $\alpha \geq \beta$  devolve  $v$  (corte)
6. Se  $k=b$  devolve  $v$ ; c.c. vai para  $n_{k+1}$  i.e.  $k \leftarrow k+1$  e vai p/3
- ii.  $v = +\infty$
- iii.  $v \leftarrow \min[v, \text{AlfaBeta}(n_k; \alpha; \beta)]$  ←
- iv.  $\beta \leftarrow \min[v, \beta]$
- v. Se  $\beta \leq \alpha$  devolve  $v$  (corte)
- vi. Se  $k=b$  devolve  $v$ ; c.c. vai para  $n_{k+1}$  i.e.  $k \leftarrow k+1$  e vai p/iii

# PSEUDOCÓDIGO DO ALFABETA (FAIL-HARD)

## AlfaBeta( $n$ ; $\alpha$ ; $\beta$ )

1. Se  $n$  no limite de profundidade  $d$ , devolve AlfaBeta( $n$ )= $f(n)$ , caso contrário calcula os sucessores  $n_1, \dots, n_k, \dots, n_b$  (por ordem), faz  $k=1$  e, se  $n$  é um nó MAX, vai p/2 c.c. vai p/ ii.
2.  $v = -\infty$
3.  $v \leftarrow \max[v, \text{AlfaBeta}(n_k; \alpha; \beta)]$  ←
4.  $\alpha \leftarrow \max[v, \alpha]$
5. Se  $\alpha \geq \beta$  devolve  $\beta$  (corte)
6. Se  $k=b$  devolve  $v$ ; c.c. vai para  $n_{k+1}$  i.e.  $k \leftarrow k+1$  e vai p/3
- ii.  $v = +\infty$
- iii.  $v \leftarrow \min[v, \text{AlfaBeta}(n_k; \alpha; \beta)]$  ←
- iv.  $\beta \leftarrow \min[v, \beta]$
- v. Se  $\beta \leq \alpha$  devolve  $\alpha$  (corte)
- vi. Se  $k=b$  devolve  $v$ ; c.c. vai para  $n_{k+1}$  i.e.  $k \leftarrow k+1$  e vai p/iii

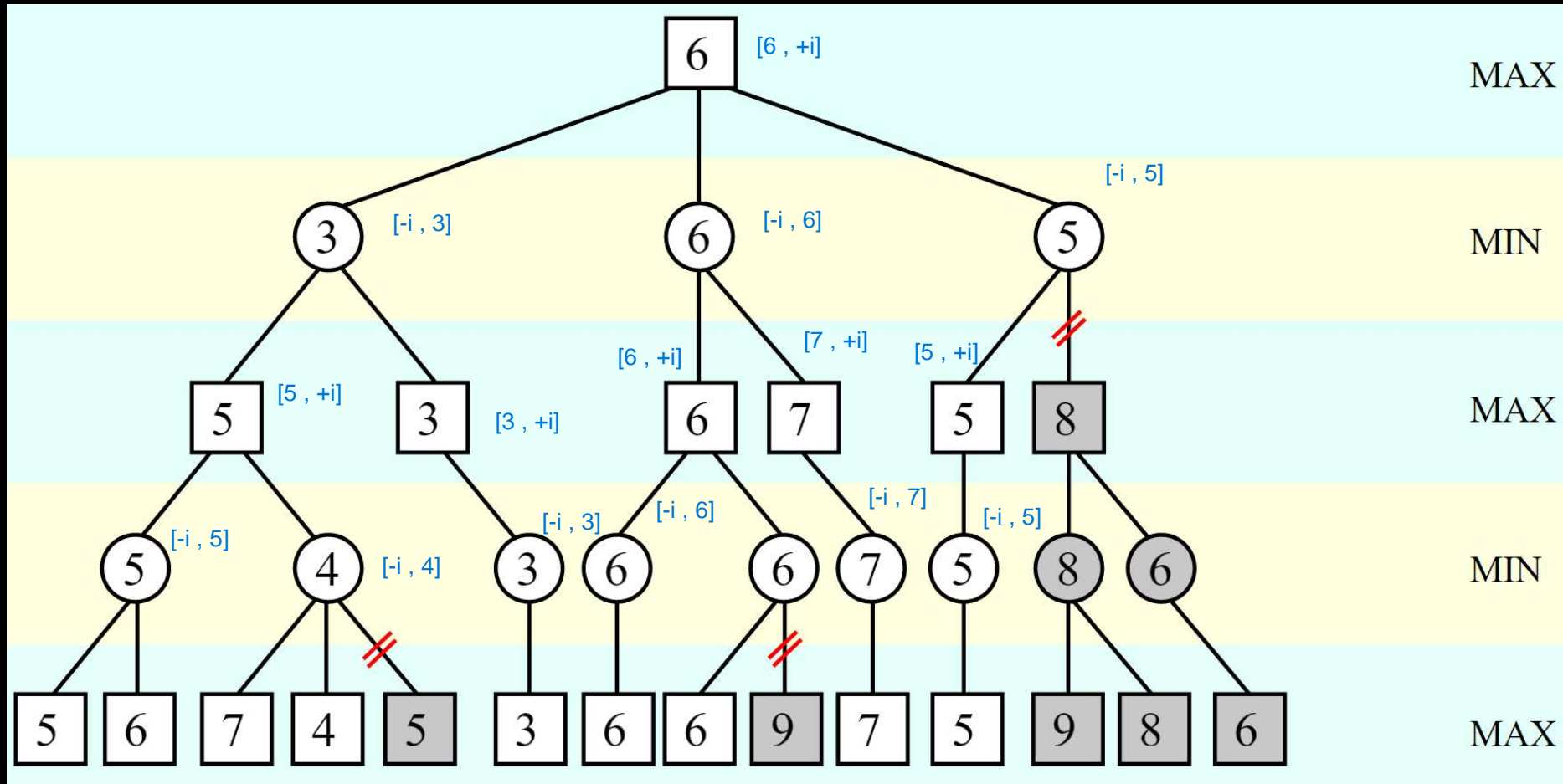
# PSEUDOCÓDIGO (FAIL-HARD) SIMPLIFICADO

## AlfaBeta( $n$ ; $\alpha$ ; $\beta$ )

1. Se  $n$  no limite de profundidade  $d$ , devolve AlfaBeta( $n$ )= $f(n)$ , caso contrário calcula os sucessores  $n_1, \dots, n_k, \dots, n_b$  (por ordem), faz  $k=1$  e, se  $n$  é um nó MAX, vai p/2 c.c. vai p/ ii.
2.  $\alpha \leftarrow \max[\alpha, \text{AlfaBeta}(n_k; \alpha; \beta)]$
3. Se  $\alpha \geq \beta$  devolve  $\beta$ ; c.c. continua
4. Se  $k=b$  devolve  $\alpha$ ; c.c. vai para  $n_{k+1}$  i.e.  $k \leftarrow k+1$  e vai p/2
- ii.  $\beta \leftarrow \min[\beta, \text{AlfaBeta}(n_k; \alpha; \beta)]$
- iii. Se  $\beta \leq \alpha$  devolve  $\alpha$ ; c.c. continua
- iv. Se  $k=b$  devolve  $\beta$ ; c.c. vai para  $n_{k+1}$  i.e.  $k \leftarrow k+1$  e vai p/ii

# UM EXEMPLO DE ALFA-BETA (FAIL-SOFT)

Árvore percorrida da esquerda para a direita; nós a cinzento não são explorados.



Wikipedia: [https://en.wikipedia.org/wiki/Alpha%E2%80%93beta\\_pruning](https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning)

# NEGAMAX CONJUGADO COM CORTES ALFA-BETA E PROFUNDIDADE LIMITADA

```

;;; argumentos: nó n, profundidade d, cor c
;;; b = ramificação (número de sucessores)
function negamax (n, d,  $\alpha$ ,  $\beta$ , c)
  se d = 0 ou n é terminal
    return c * valor heurístico de n
  sucessores := OrderMoves(GenerateMoves(n))
  bestValue :=  $-\infty$ 
  para cada sucessor  $n_k$  em sucessores
    bestValue := max (bestValue, -negamax ( $n_k$ , d-1,  $-\beta$ ,  $-\alpha$ , -c))
     $\alpha$  := max ( $\alpha$ , bestValue)
    se  $\alpha \geq \beta$ 
      break
  return bestValue

```

Valor do nó inicial

```
rootNegamaxValue := negamax( rootNode, depth,  $-\infty$ ,  $+\infty$ , 1)
```

# EFICIÊNCIA DO MINIMAX COM CORTES ALFA-BETA

- Admita-se que uma árvore tem profundidade  $d$  e um fator de ramificação média  $b$ .
- Assumindo que seria possível obter uma estimativa da ordenação do valor dos sucessores, o alfa-beta apenas necessitaria de examinar  $O(b^{d/2})$  **nós-folha** para escolher o melhor lance, em vez de  $O(b^d)$  com o minimax.
- Isto significa que o factor de ramificação efectivo é  $\sqrt{b}$  em vez de  $b$ . Assim, o alfa-beta consegue examinar mais níveis do que o minimax, no mesmo tempo.
- Não é possível obter a ordenação numa situação real, mas demonstra-se que em média, o alfa-beta avaliaria um número de nós na ordem de  $O(b^{3d/4})$  o que permite aumentar a profundidade de pesquisa em cerca de  $4/3$ .
- A ordem pode ser estimada com base na função de avaliação estática. A eficácia depende da heurística.

# TABELAS DE TRANSPosição

- Em certos jogos de informação perfeita, em que é possível chegar a um estado de mais do que uma maneira (transposições), é viável acelerar a procura usando *hash tables*, que são consultadas de cada vez que se gera um novo estado, constituindo uma espécie de cache.
- Quando há um *hit*, não se poupa apenas a avaliação da posição mas sim a de toda a subárvore abaixo da mesma.
- Quanto maior o *hit rate* maior o ganho de eficiência.



# HASH TABLES

- Criar uma hash table:  
**(make-hash-table &key test size rehash-size rehash-threshold)**  
Valores por default:  
test: eql  
size, rehash-size rehash-threshold: dependente da implementação
- Acesso:  
**(gethash <entrada> <hashtable>)**
- Alteração/modificação:  
**(setf (gethash <entrada> <hashtable>) <valor>)**

# EXEMPLO DE UTILIZAÇÃO DE HASH TABLES

```
(defparameter *my-hash* (make-hash-table))
```

```
*MY-HASH*
```

```
(setf (gethash 'one-entry *my-hash*) "one")
```

```
"one"
```

```
(setf (gethash 'another-entry *my-hash*) 2/4)
```

```
1/2
```

```
(gethash 'one-entry *my-hash*)
```

```
"one"
```

```
T
```

```
(gethash 'another-entry *my-hash*)
```

```
1/2
```

```
T
```

# PROGRAMAÇÃO DINÂMICA E MEMOIZAÇÃO

- A **programação dinâmica** é um método para resolver problemas complexos em que se divide um problema em **problemas mais simples que ocorrem várias vezes** e se guardam as soluções destes para que a resolução só seja efetuada 1 vez.
- A abordagem de guardar em memória as soluções dos subproblemas designa-se por **memoização**. Não confundir com “memorização”.  
<https://en.wikipedia.org/wiki/Memoization>
- Memoização foi um termo criado por Donald Michie em 1968.

# MEMOIZAÇÃO APLICADA À SÉRIE DE FIBONNACI

```
(defun fib (n)
  (cond ((<= n 1) 1)
        (t (+ (fib (1- n)) (fib (- n 2))))))
```

```
(let ((tab (make-hash-table)))
  (defun fib-memo (n)
    (or (gethash n tab)
        (let ((val (funcall #'fib n)))
          (setf (gethash n tab) val)
          val)))))
```

Implementação em  
LISP: **Closure**, i.e.  
“let over lambda”

# EXEMPLO DE FUNCIONAMENTO: 1º RUN

> (trace fib)	3 FIB > ...	2 FIB > ...
	>> N : 0	>> N : 1
> (fib-memo 4)	3 FIB < ...	2 FIB < ...
	<< VALUE-0 : 1	<< VALUE-0 : 1
0 FIB > ...	2 FIB < ...	2 FIB > ...
>> N : 4	<< VALUE-0 : 2	>> N : 0
1 FIB > ...	2 FIB > ...	2 FIB < ...
>> N : 3	>> N : 1	<< VALUE-0 : 1
2 FIB > ...	2 FIB < ...	1 FIB < ...
>> N : 2	<< VALUE-0 : 1	<< VALUE-0 : 2
3 FIB > ...	1 FIB < ...	0 FIB < ...
>> N : 1	<< VALUE-0 : 3	<< VALUE-0 : 5
3 FIB < ...	1 FIB > ...	5
<< VALUE-0 : 1	>> N : 2	

## CONT. EXEMPLO FUNCIONAMENTO: 2º RUN

> (fib-memo 4)

5

Qual é o ganho de eficiência?

Inconveniente: a função (closure) **fib-memo** só serve para memoização da função **fib**

Terá de se fazer uma função de memoização para cada função diferente?

# GENERALIZAÇÃO

```
(defun memo (fn)
  (let ((table (make-hash-table)))
    (lambda (x)
      (or (gethash x table)
          (let ((val (funcall fn x)))
            (setf (gethash x table) val)
            val))))))
```

Implementação em  
LISP: **Closure generator**,  
i.e. “lambda over let  
over lambda”

```
(defun fib (n)
  (cond ((<= n 1) 1)
        (t (+ (fib (1- n)) (fib (- n 2)))
           )))
```

---

```
➤ (setf memo-fib (memo 'fib))
#<CLOSURE -67820031>
```

```
(funcall memo-fib 4) ;; symbol value / function?
➤ 5
```

<https://letoverlambda.com/textmode.cl/guest/chap2.html>

# NOTA FINAL SOBRE MEMOIZAÇÃO

- Todos os programas que jogam jogos de informação perfeita, caracterizados por explosão combinatória, como o xadrez, damas, etc., usam **tabelas de transposição**, i.e. Memoização.

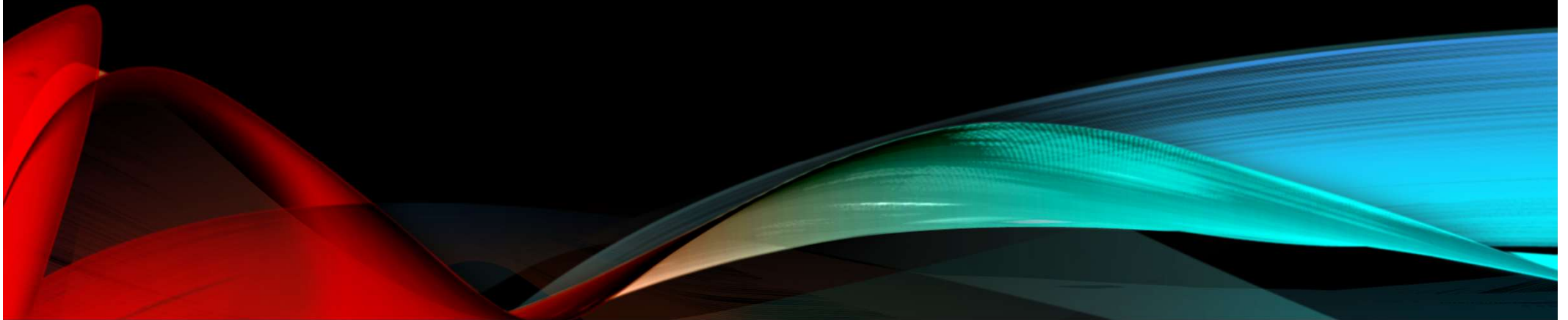


# LIMITAÇÃO DA ÁRVORE DE PROCURA


- **Método 1:** Usar um limite de profundidade fixo  $d$ .
- **Método 2:** Usar o *iterative deepening*.
- Estes métodos podem ter resultados desastrosos, pois pode haver uma variação brusca no valor da função de avaliação no nível seguinte a  $d$ .
  - **Efeito de horizonte (*Horizon Effect*)**
    - Procura quiescente – a função de avaliação só é aplicada a posições com baixa probabilidade de terem variações bruscas no valor da função de avaliação. As posições não-quiescentes podem ser expandidas até se atingirem posições quiescentes.
    - O problema do efeito de horizonte é um dos mais difíceis de eliminar. O valor de uma posição pode aparentar ser estável durante muitas jogadas, não o sendo de facto.
    - [https://en.wikipedia.org/wiki/Horizon\\_effect](https://en.wikipedia.org/wiki/Horizon_effect)

# EXEMPLO DE EXECUÇÃO DE UM ALGORITMO ALFABETA

Cédric Grueau  
(c) 2017

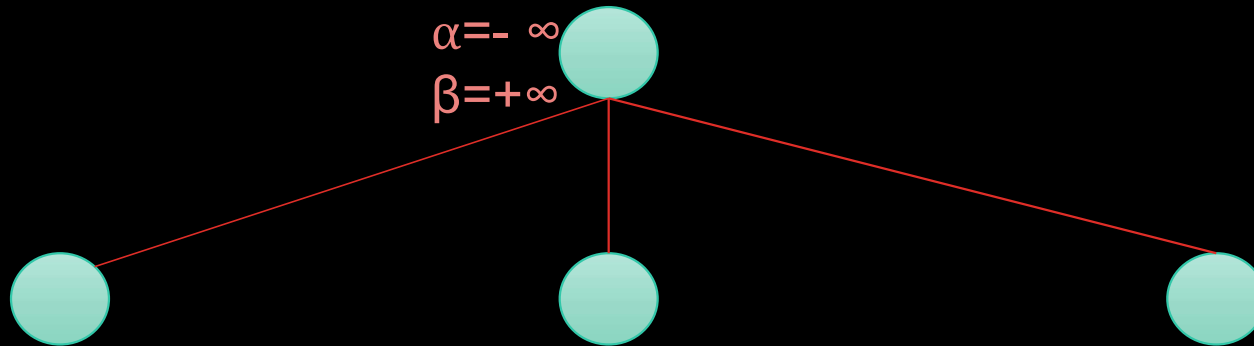


# EXEMPLO ALFABETA (1)

$$\begin{array}{l} \alpha = -\infty \\ \beta = +\infty \end{array}$$


Max

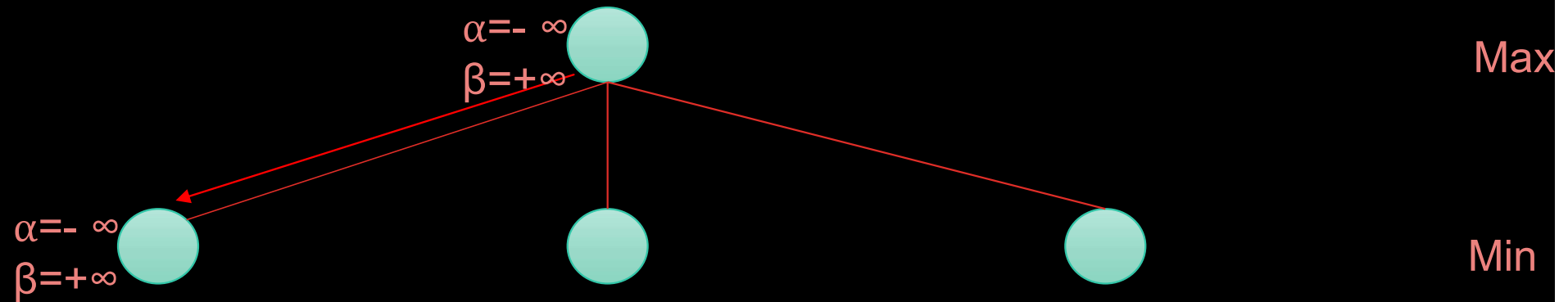
# EXEMPLO ALFABETA (2)



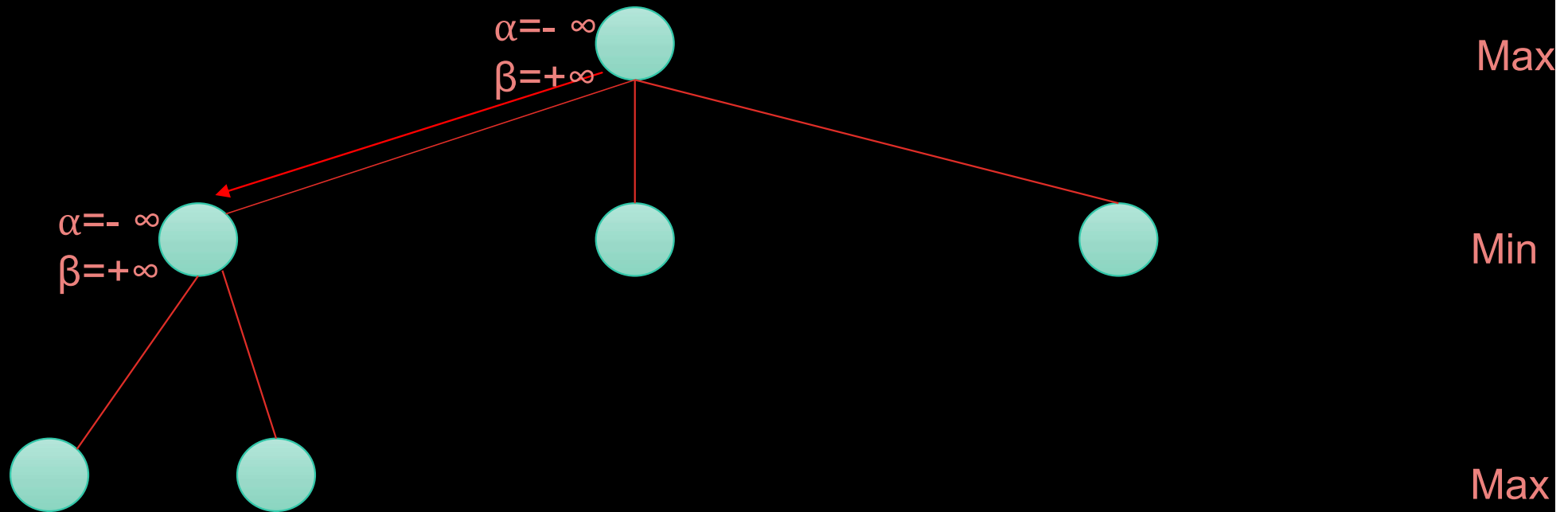
Max

Min

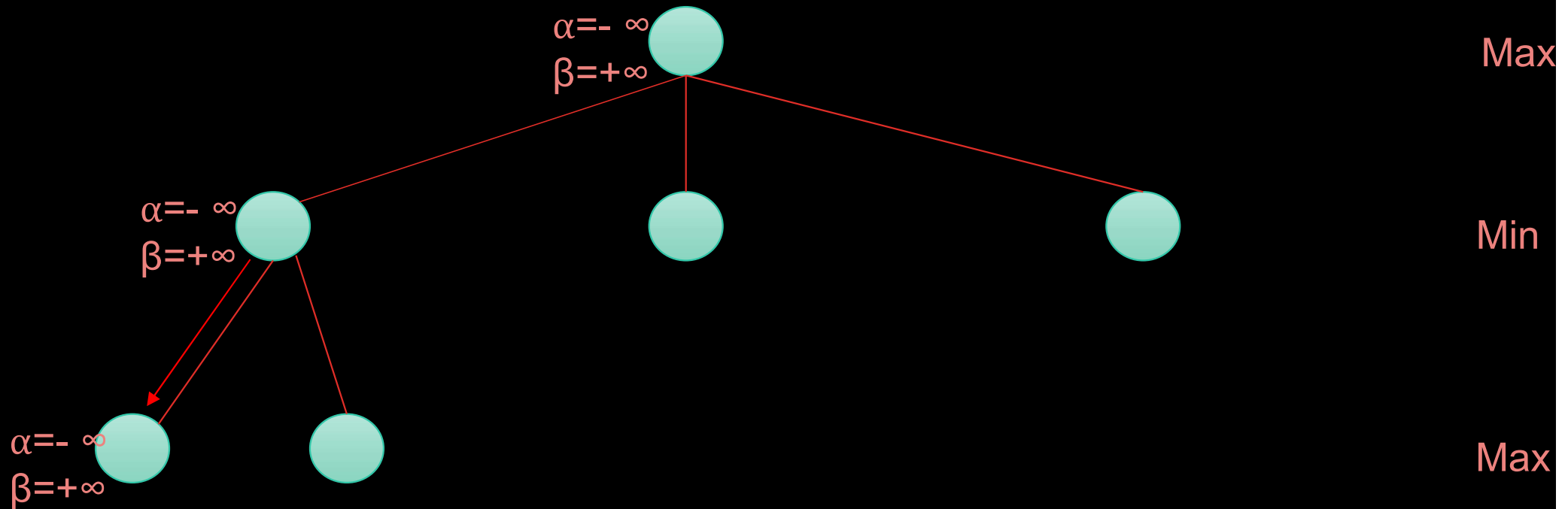
# EXEMPLO ALFABETA (3)



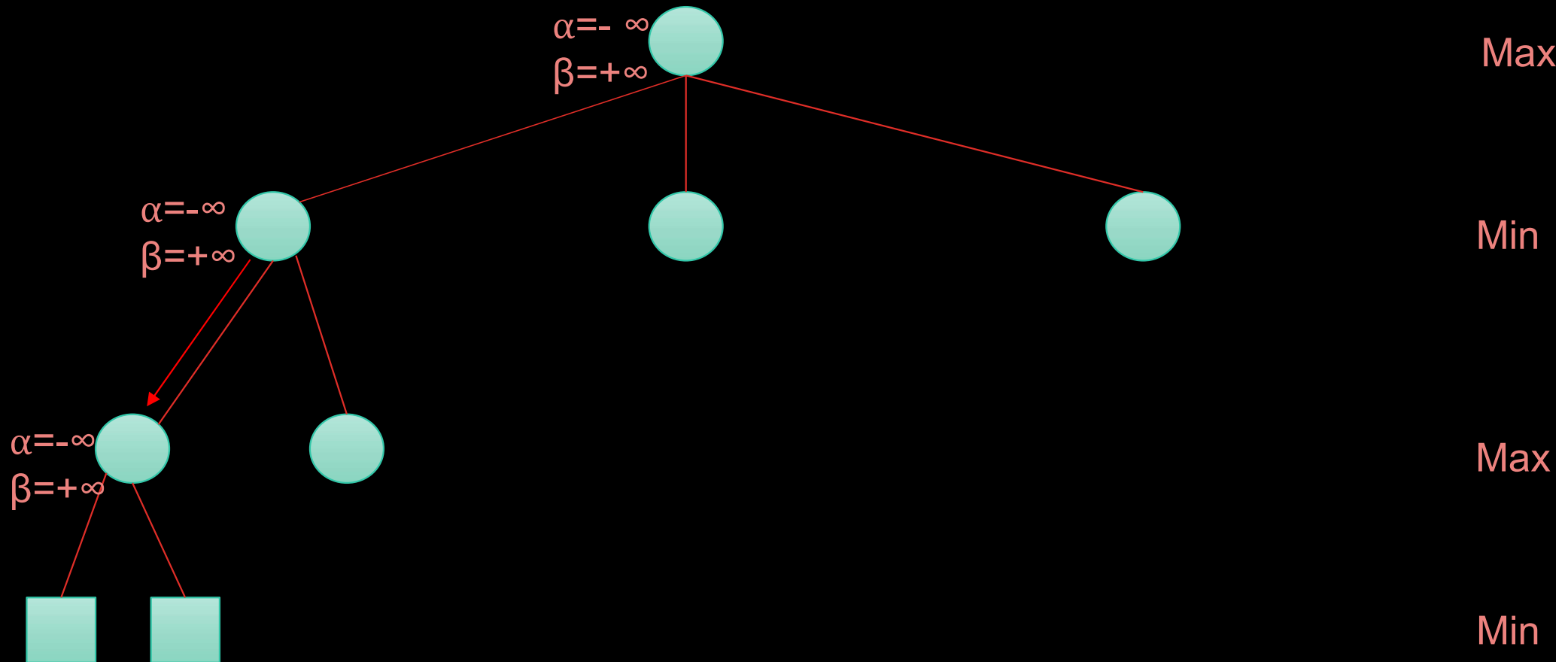
# EXEMPLO ALFABETA (4)



# EXEMPLO ALFABETA (5)

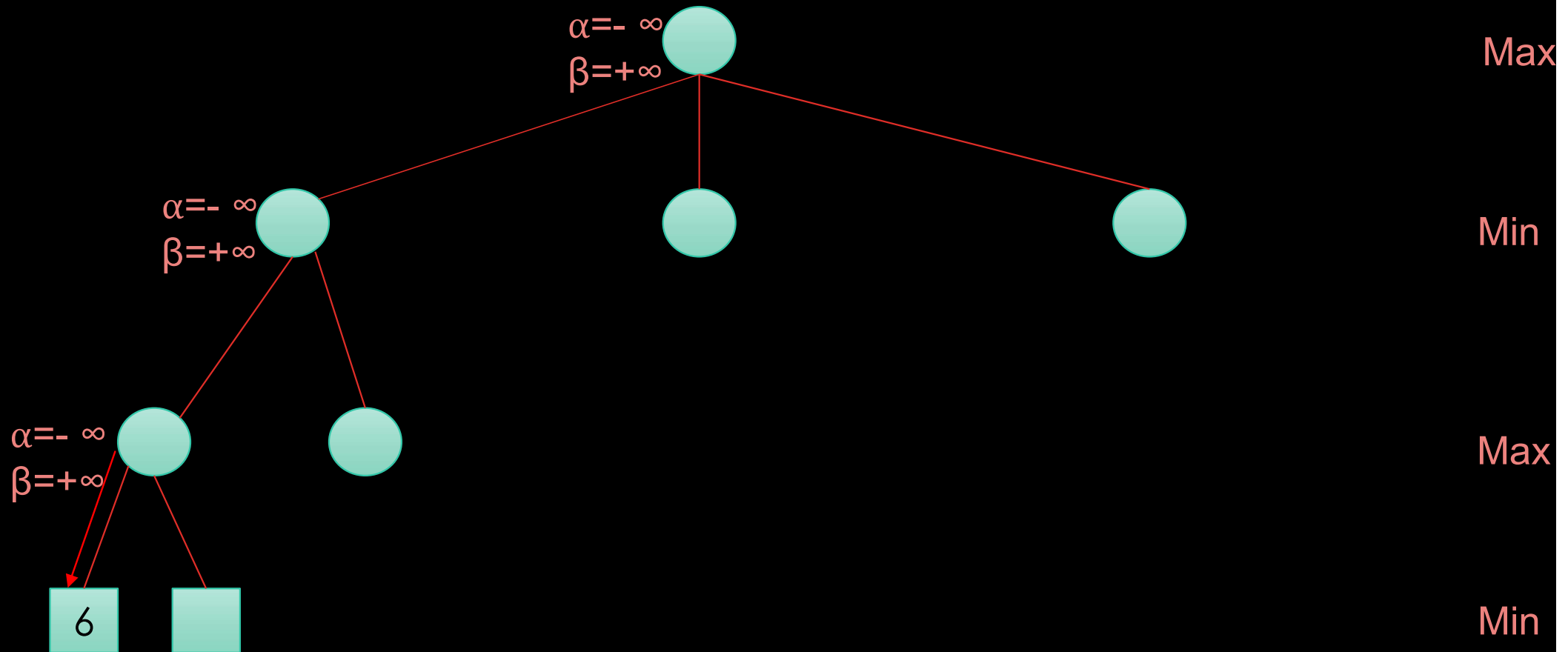


# EXEMPLO ALFABETA (6)

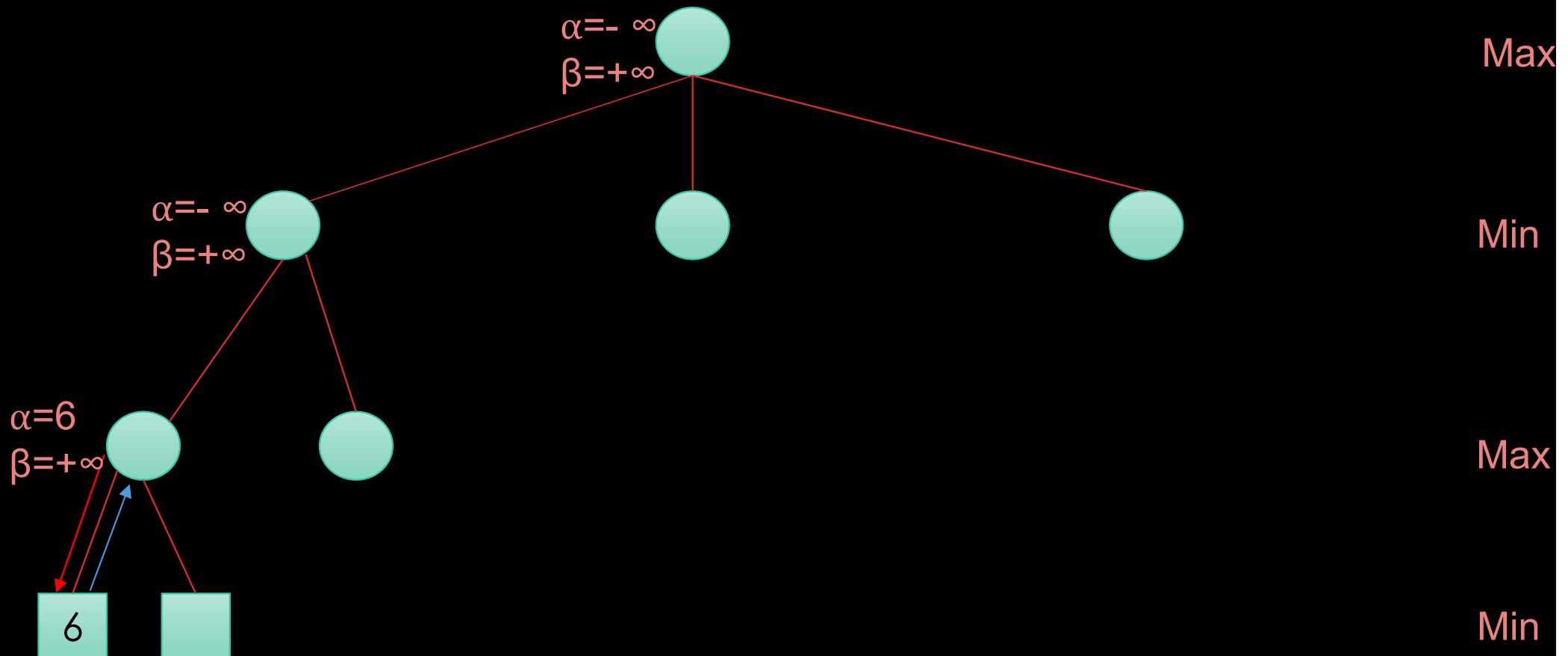




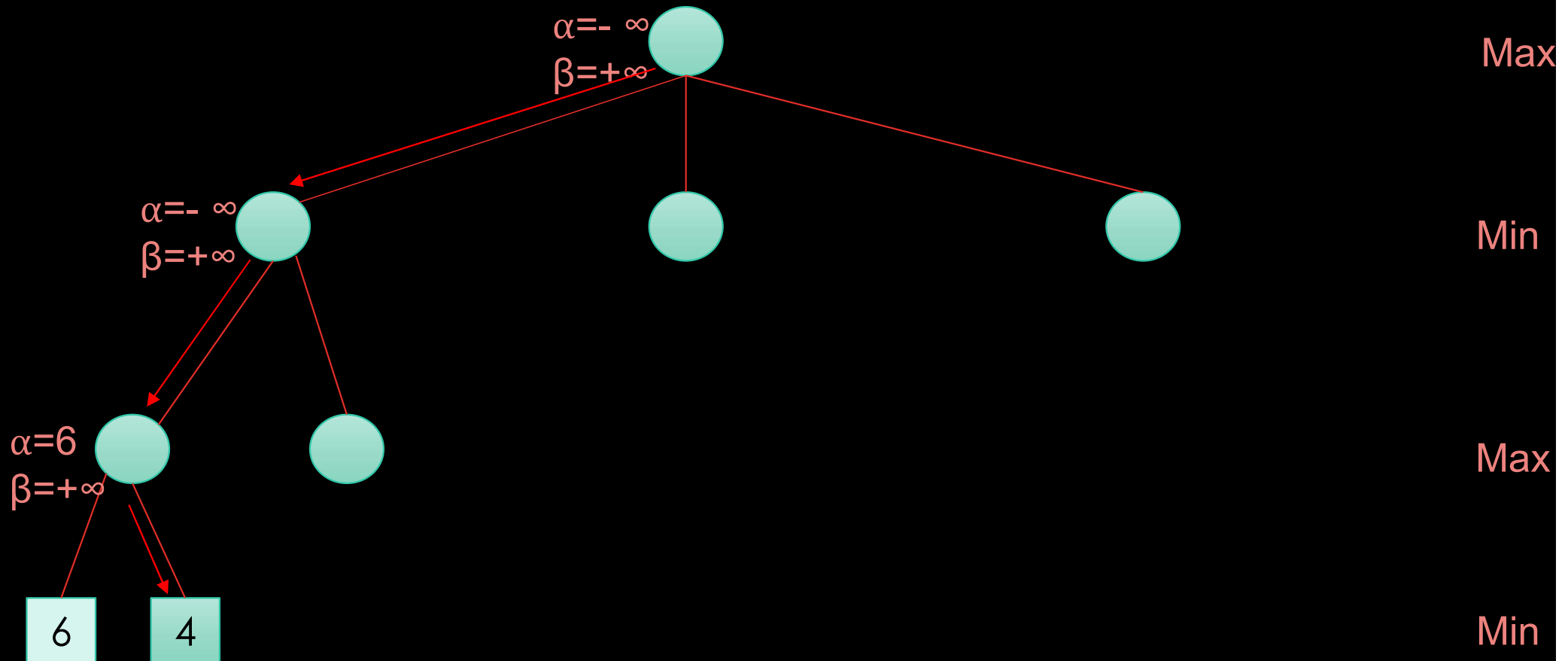
# EXEMPLO ALFABETA (7)



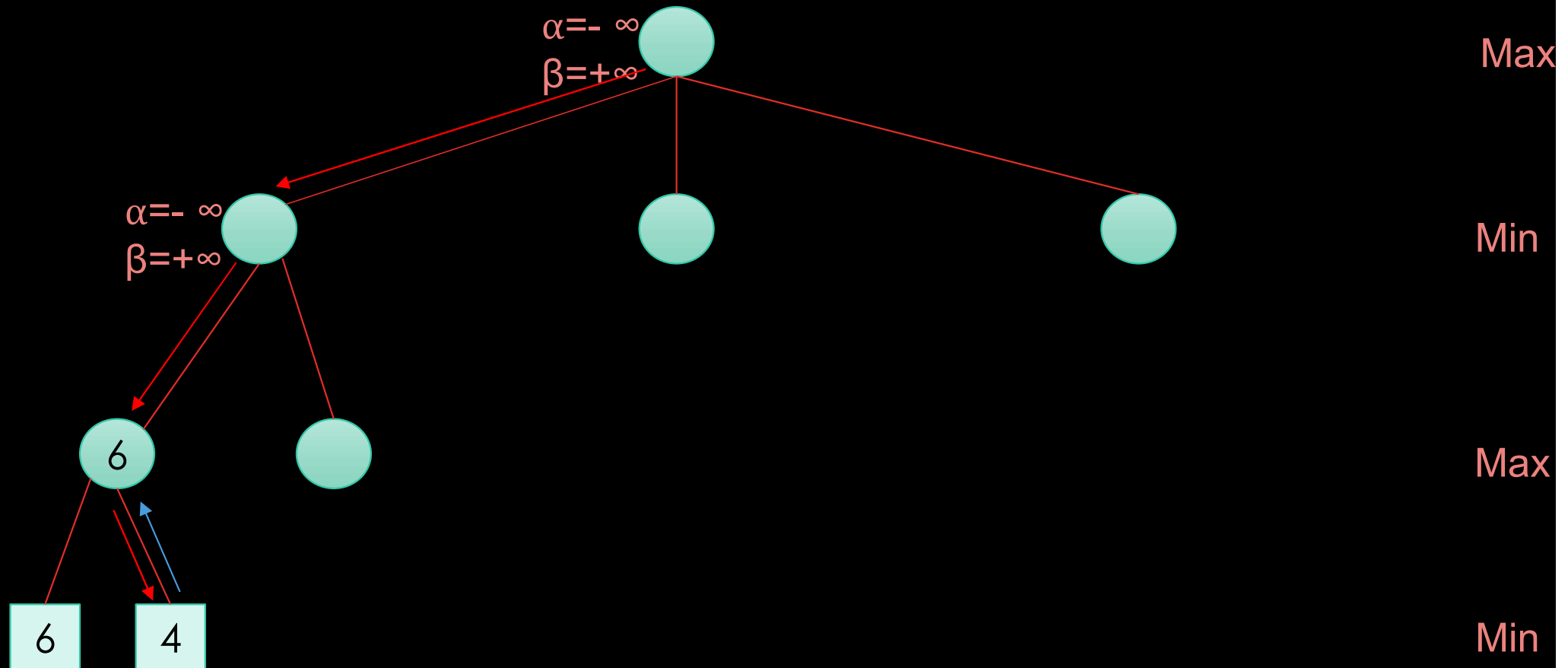
# EXEMPLO ALFABETA (8)



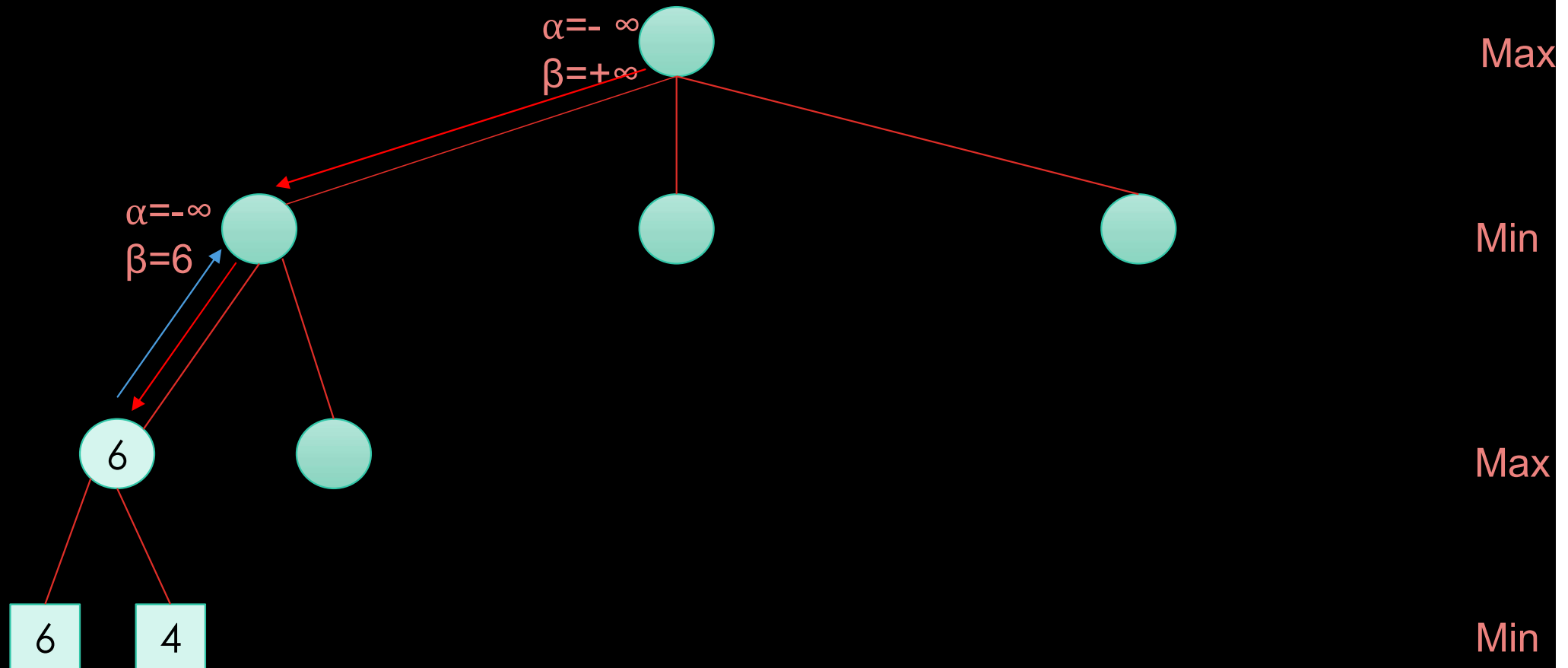
# EXEMPLO ALFABETA (9)



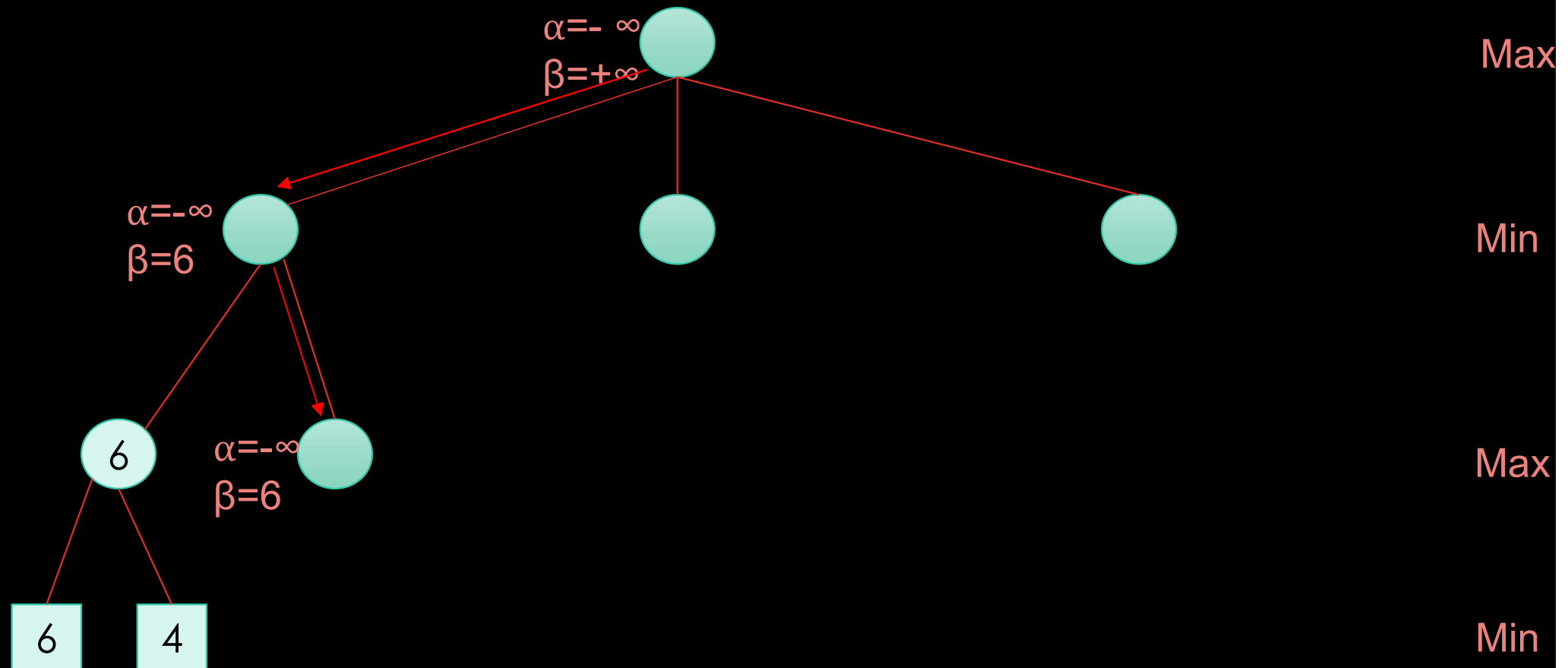
# EXEMPLO ALFABETA (10)



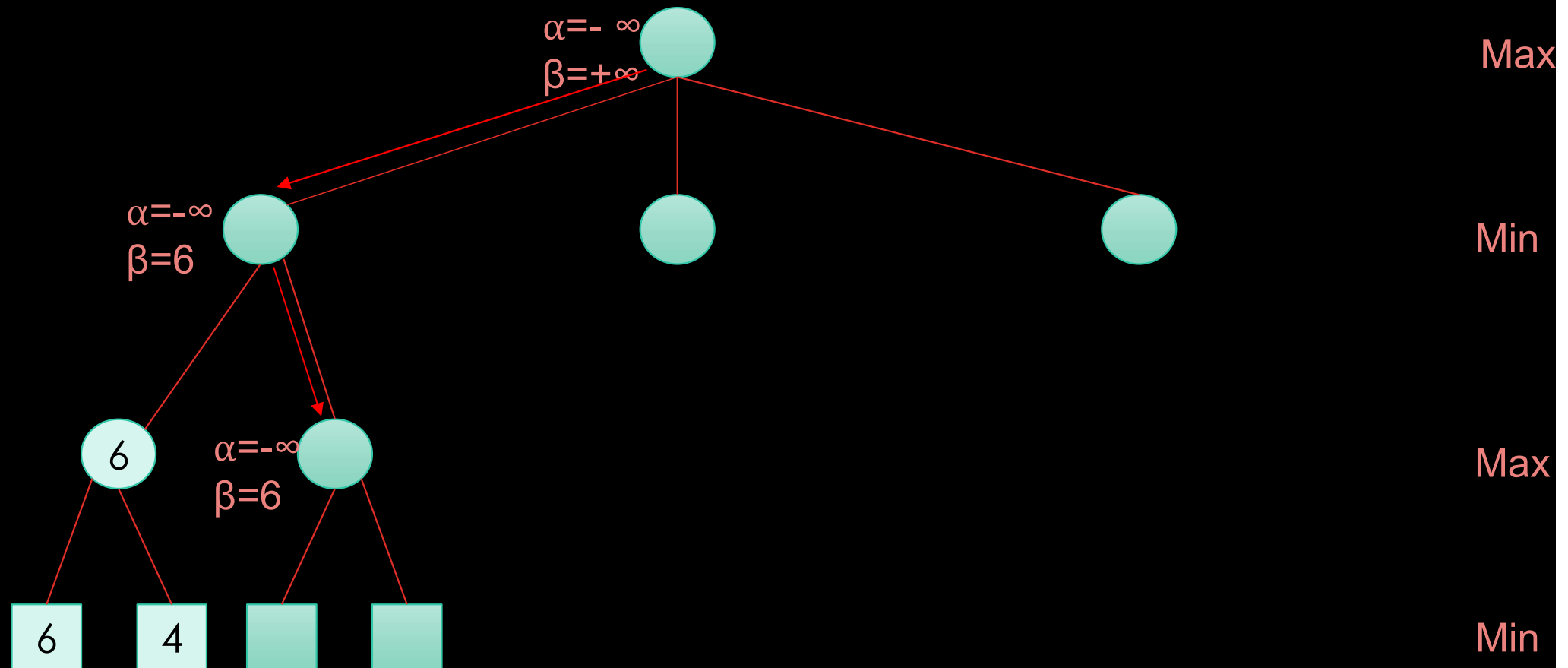
# EXEMPLO ALFABETA (11)



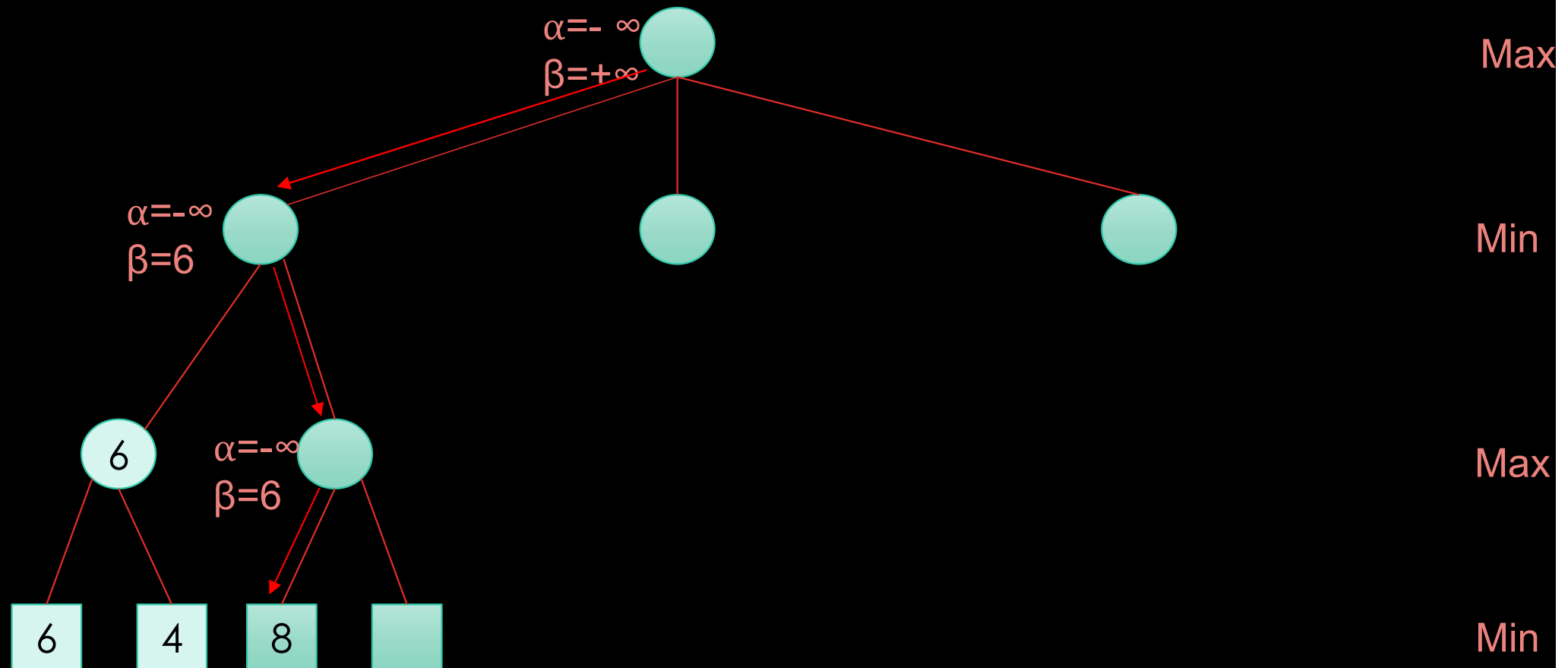
# EXEMPLO ALFABETA (12)



# EXEMPLO ALFABETA (13)

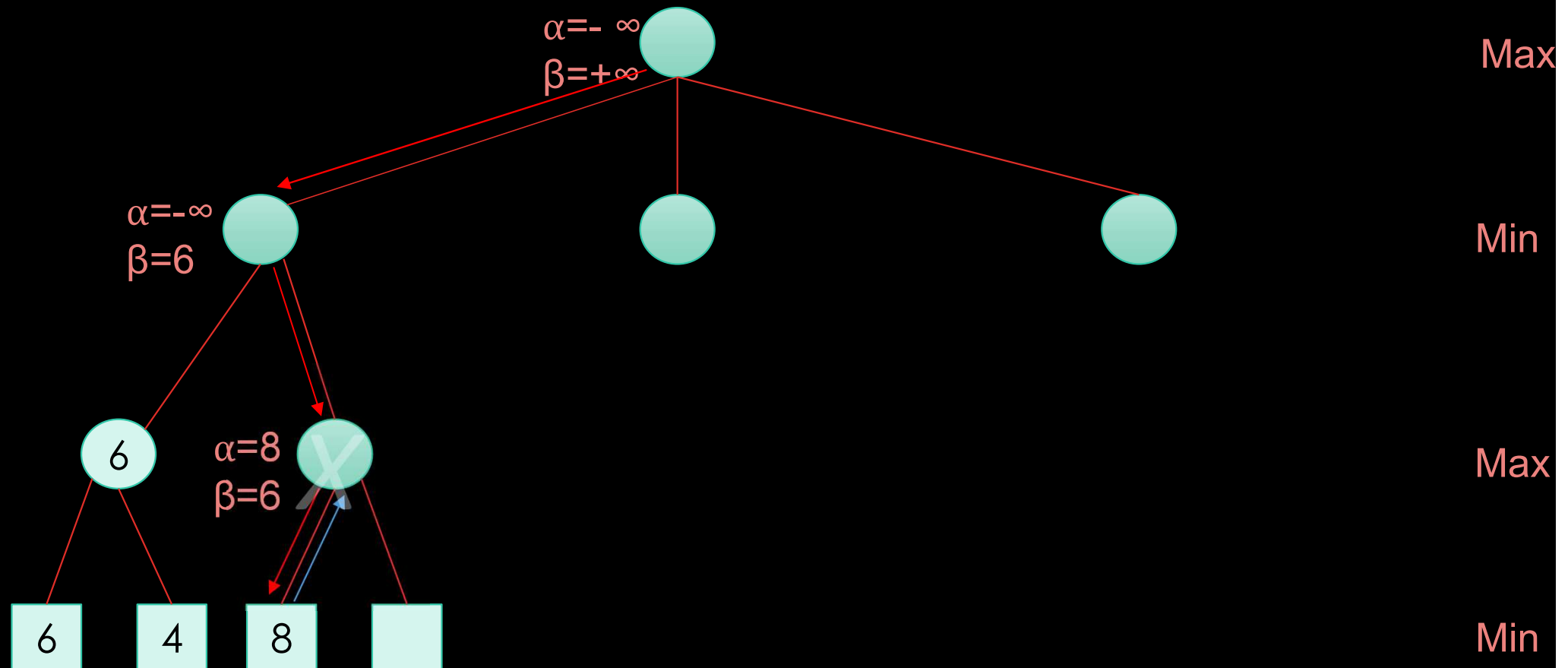


# EXEMPLO ALFABETA (14)

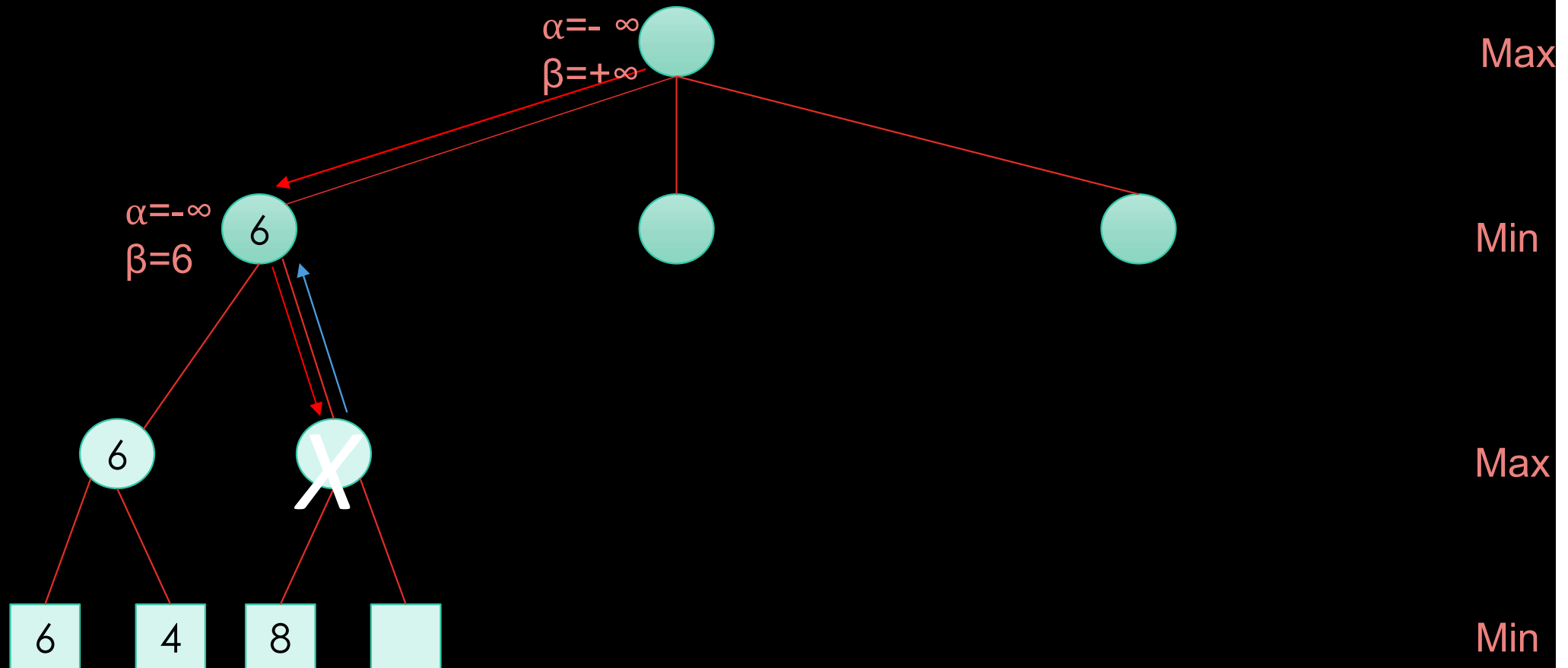




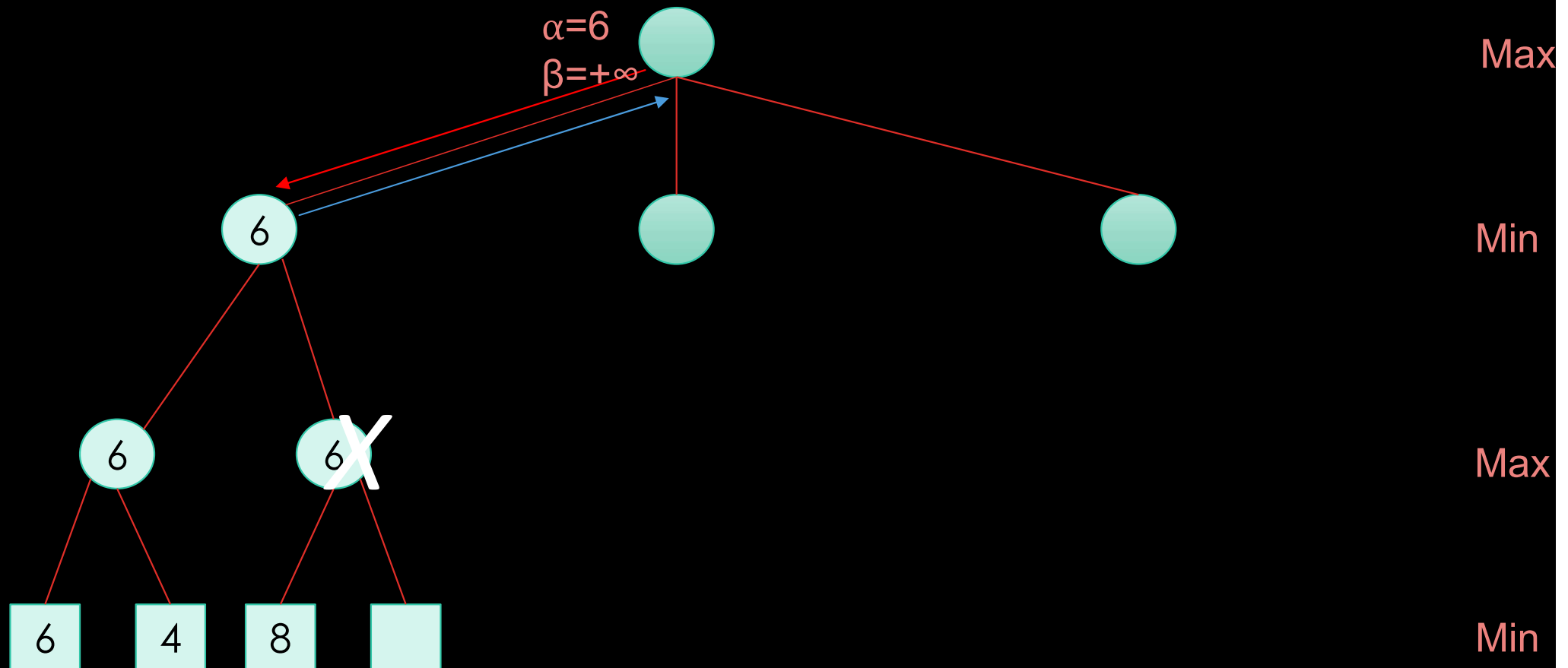
# EXEMPLO ALFABETA (15)



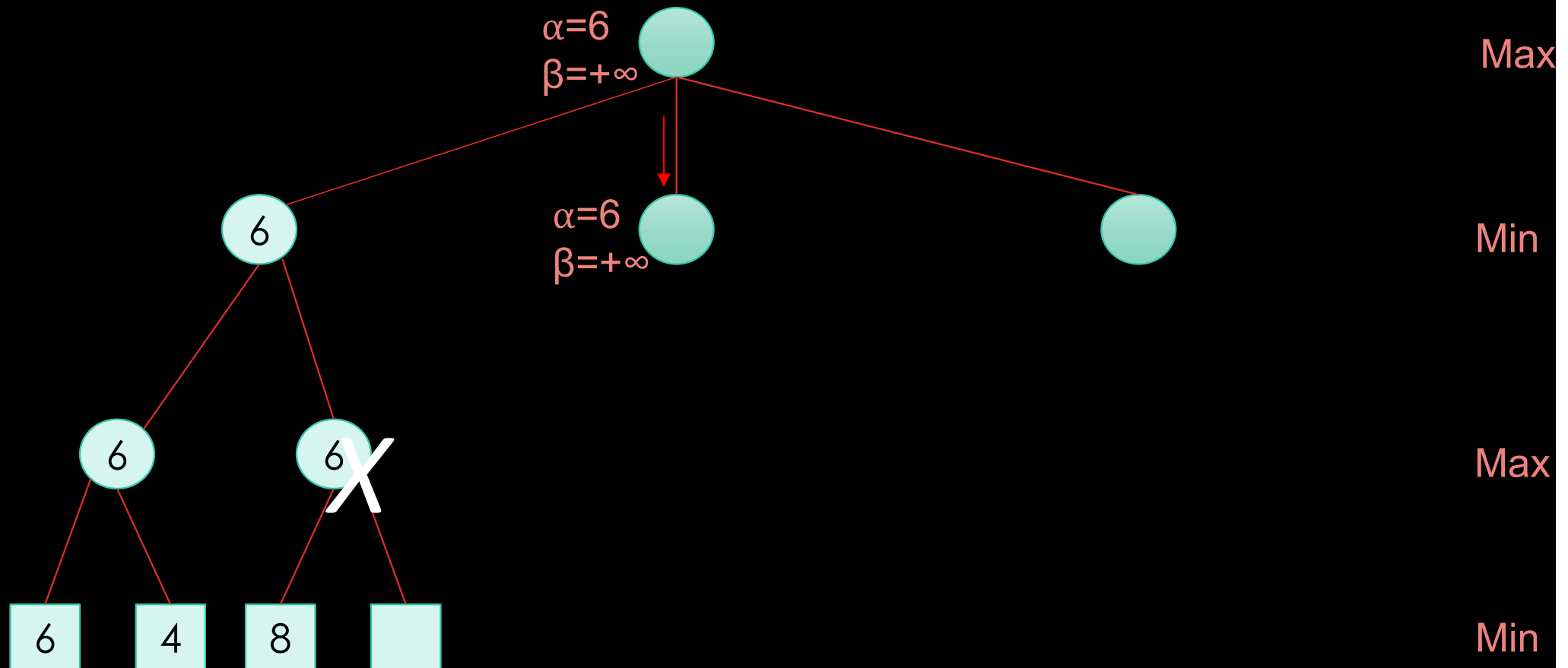
# EXEMPLO ALFABETA (16)



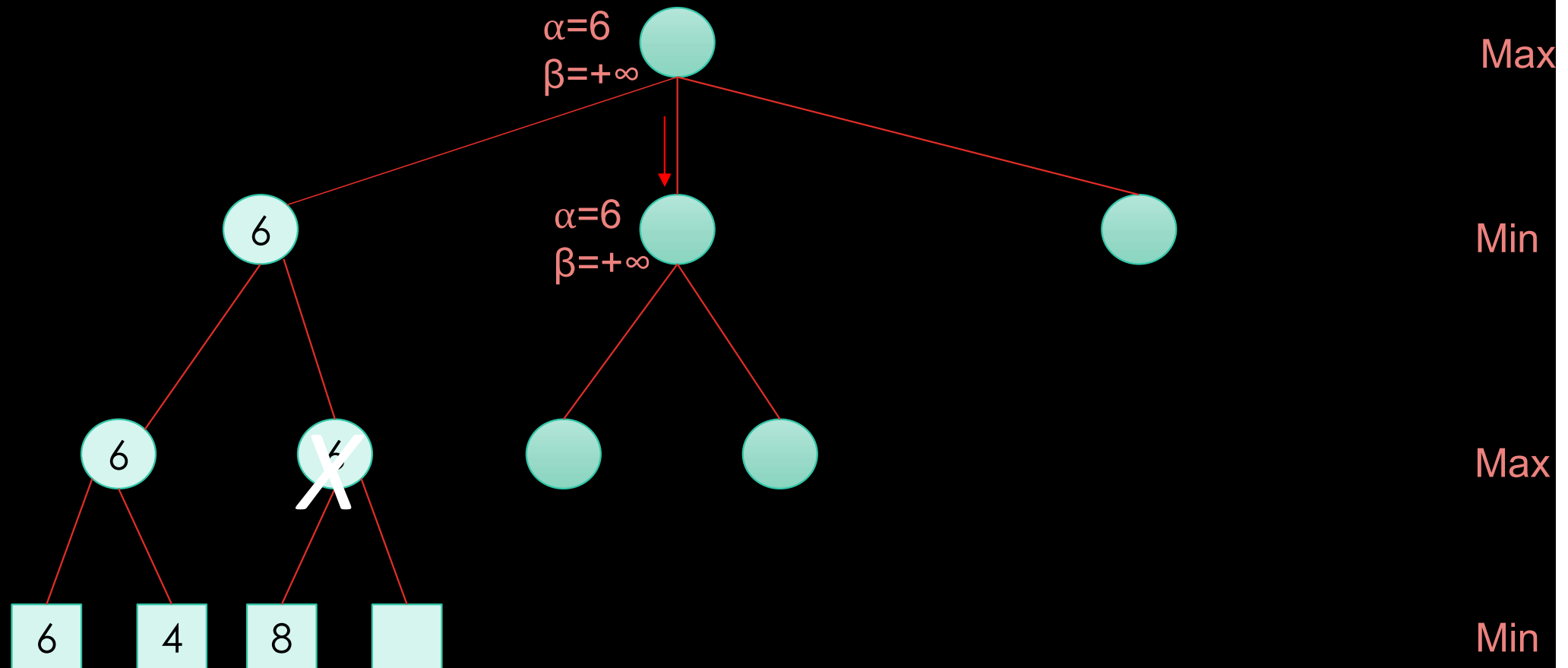
# EXEMPLO ALFABETA (17)



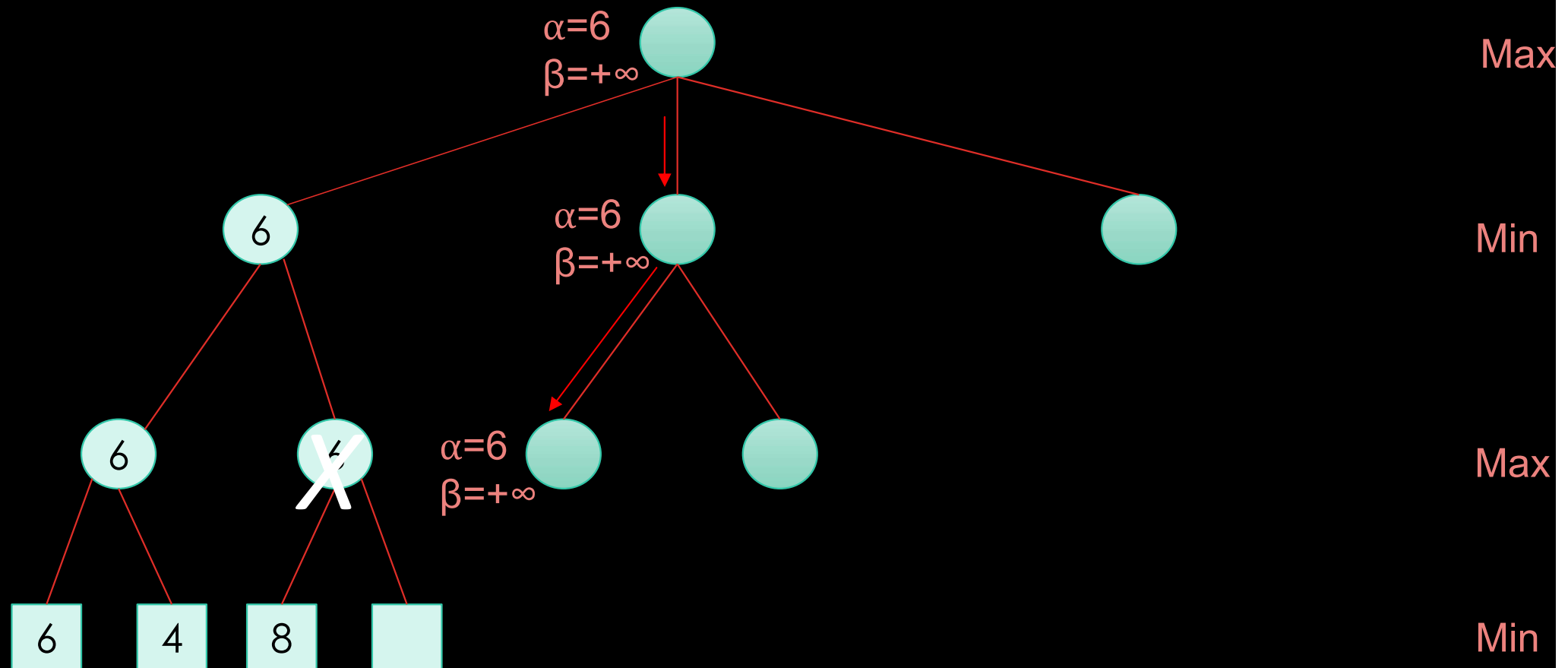
# EXEMPLO ALFABETA (18)



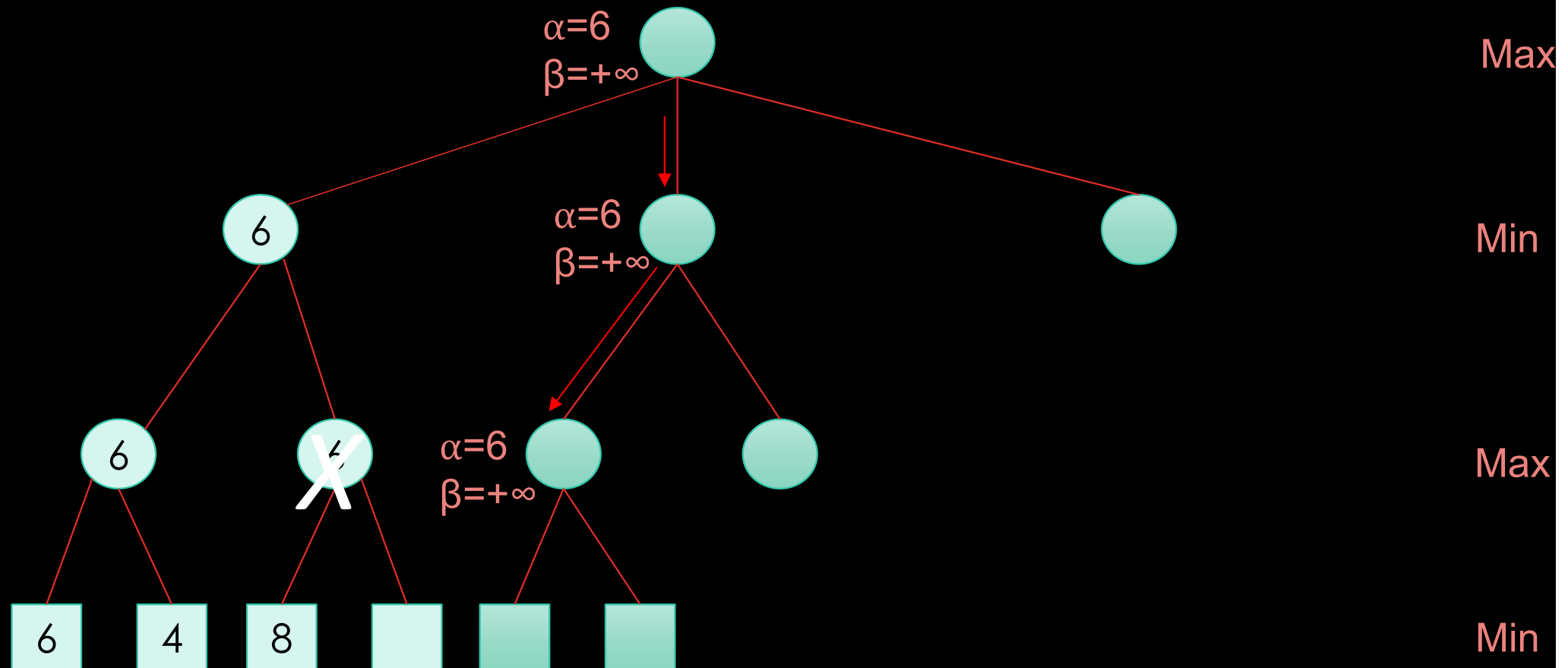
# EXEMPLO ALFABETA (19)



# EXEMPLO ALFABETA (20)

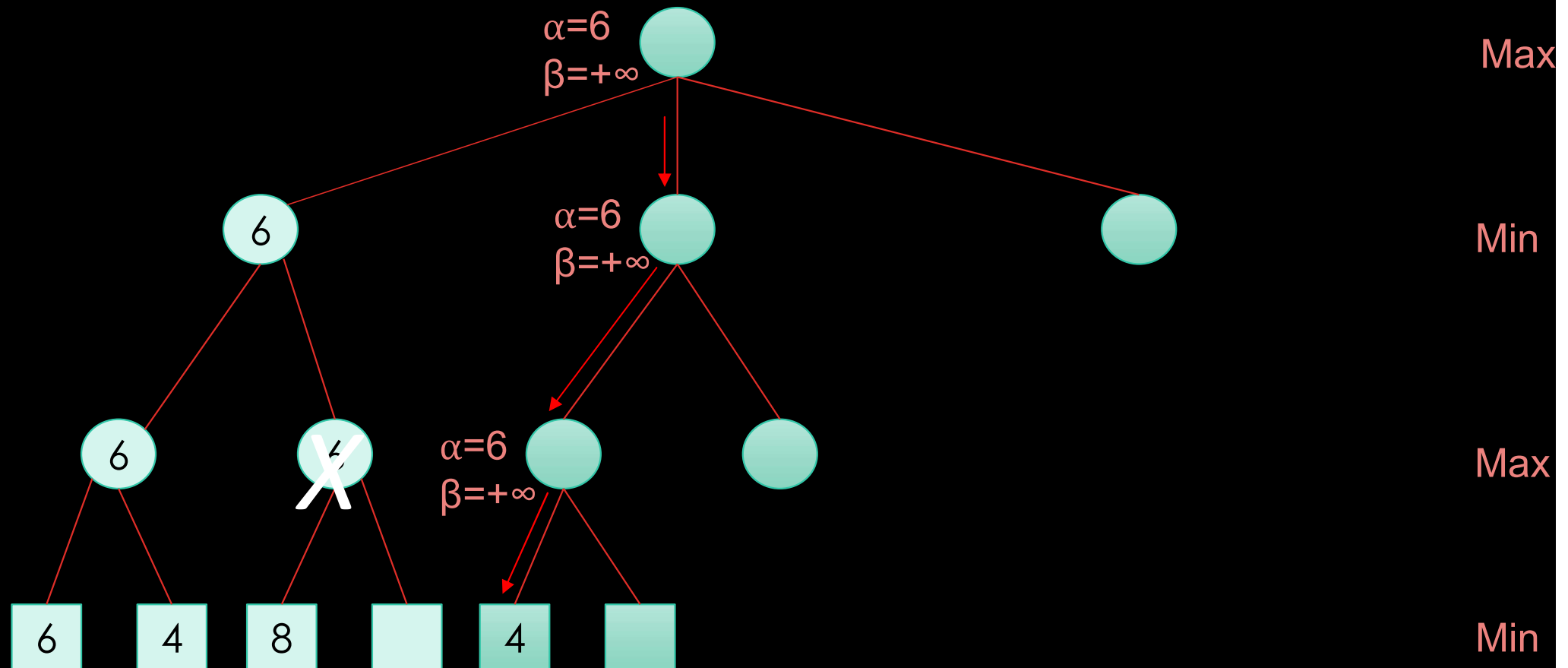


# EXEMPLO ALFABETA (21)

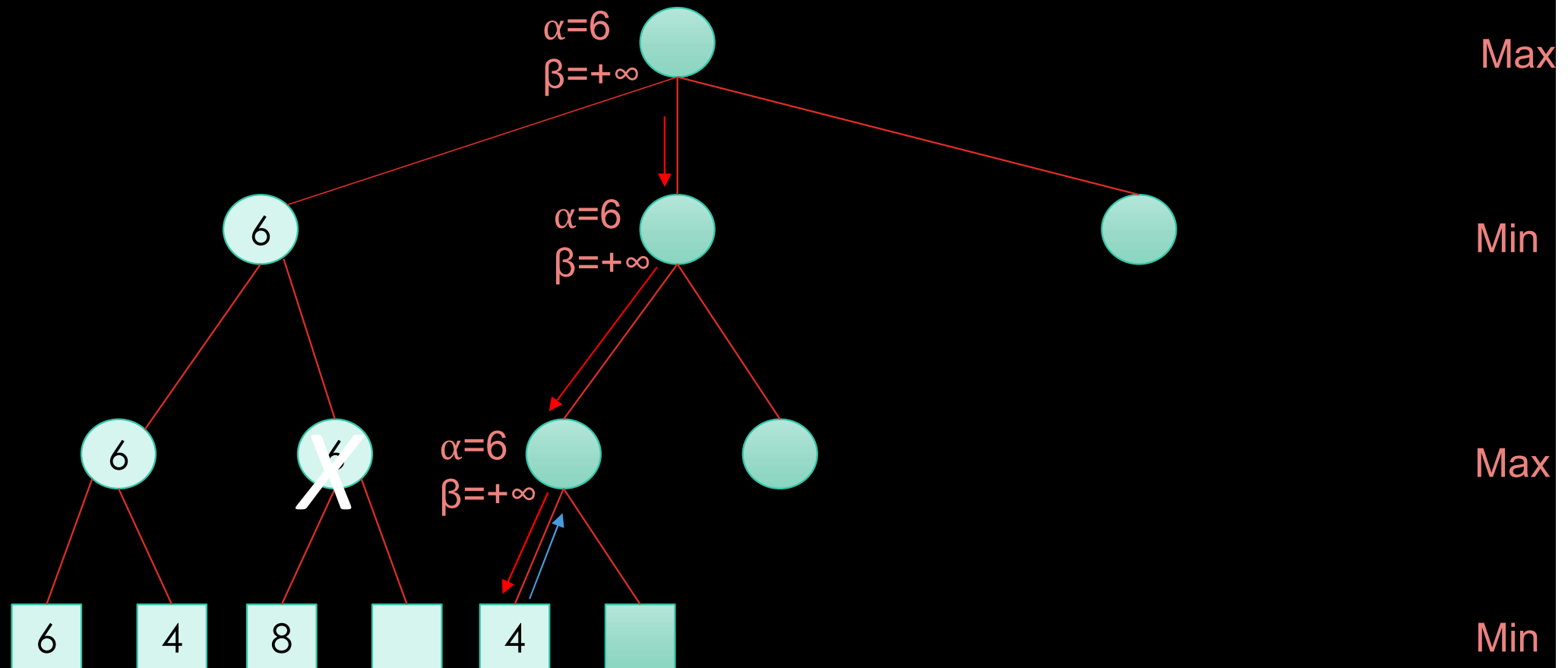




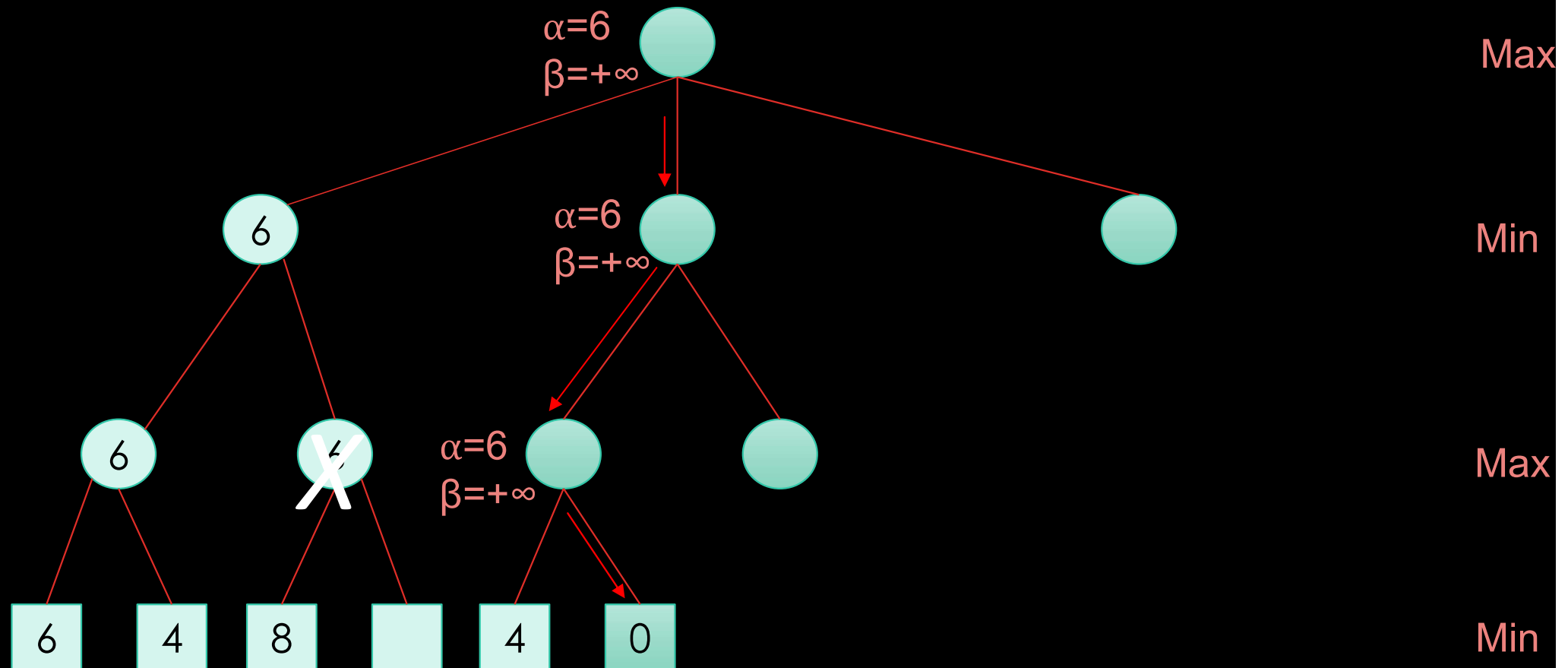
# EXEMPLO ALFABETA (22)



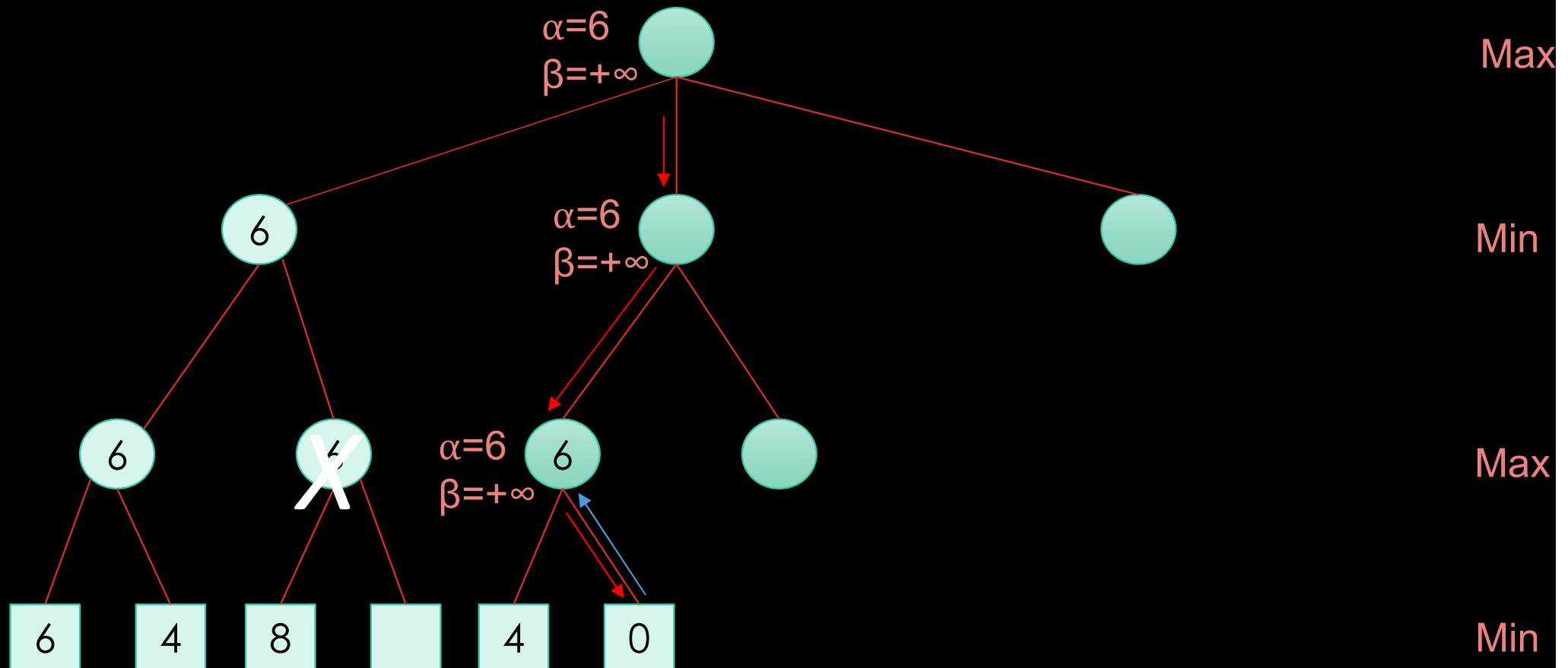
# EXEMPLO ALFABETA (23)



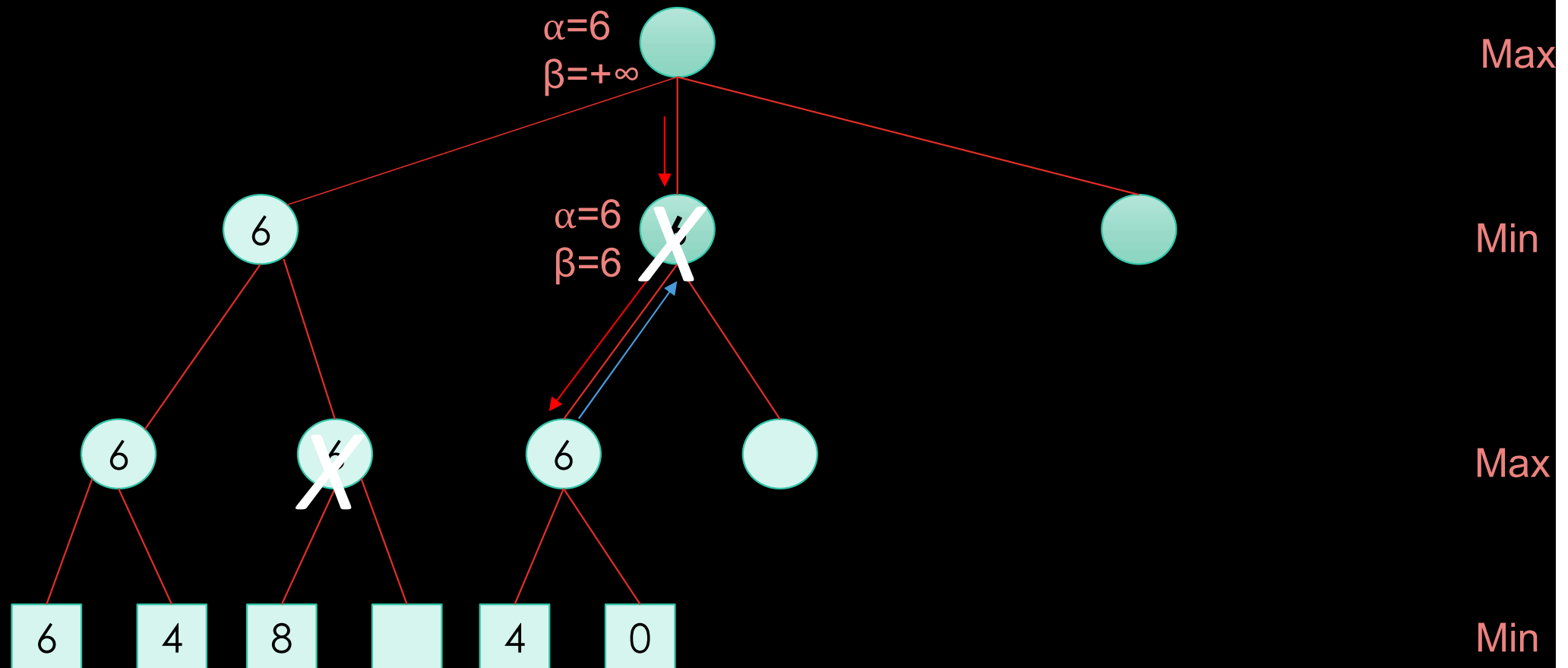
# EXEMPLO ALFABETA (24)



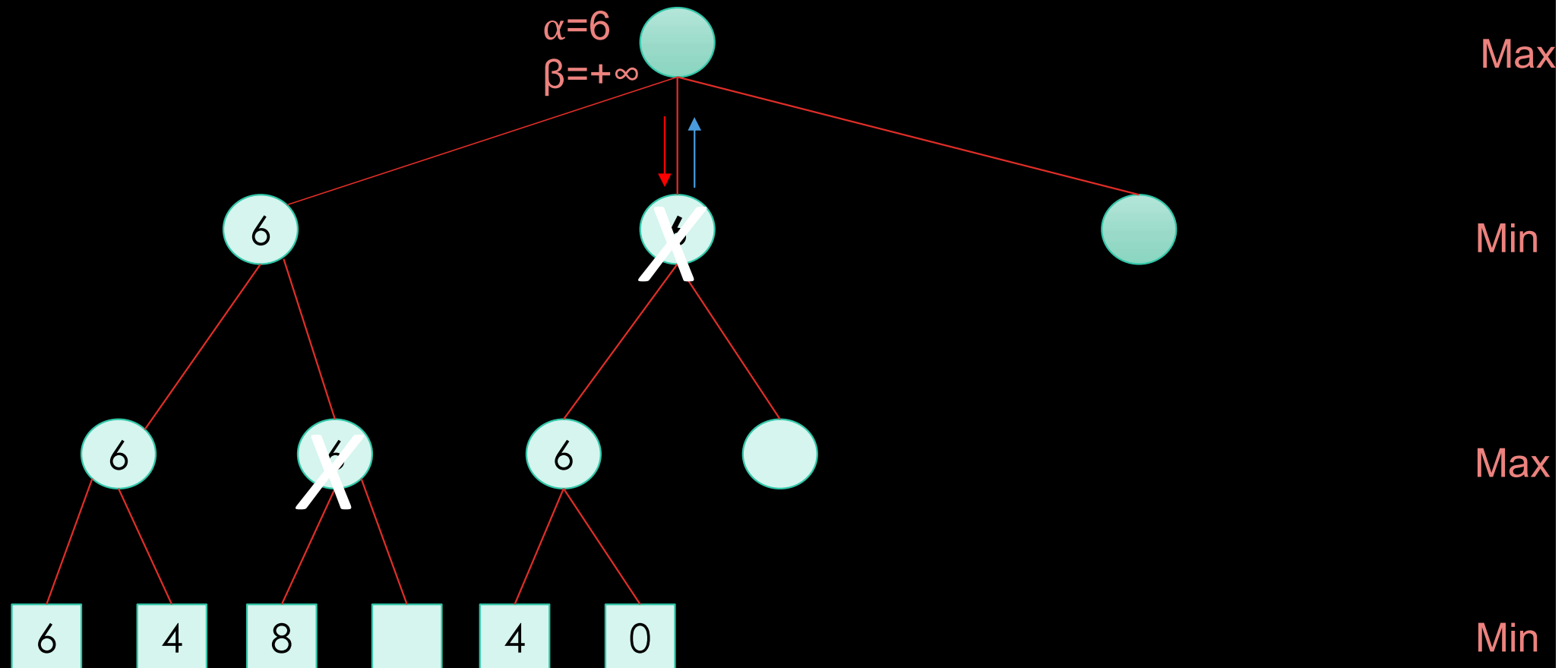
# EXEMPLO ALFABETA (25)



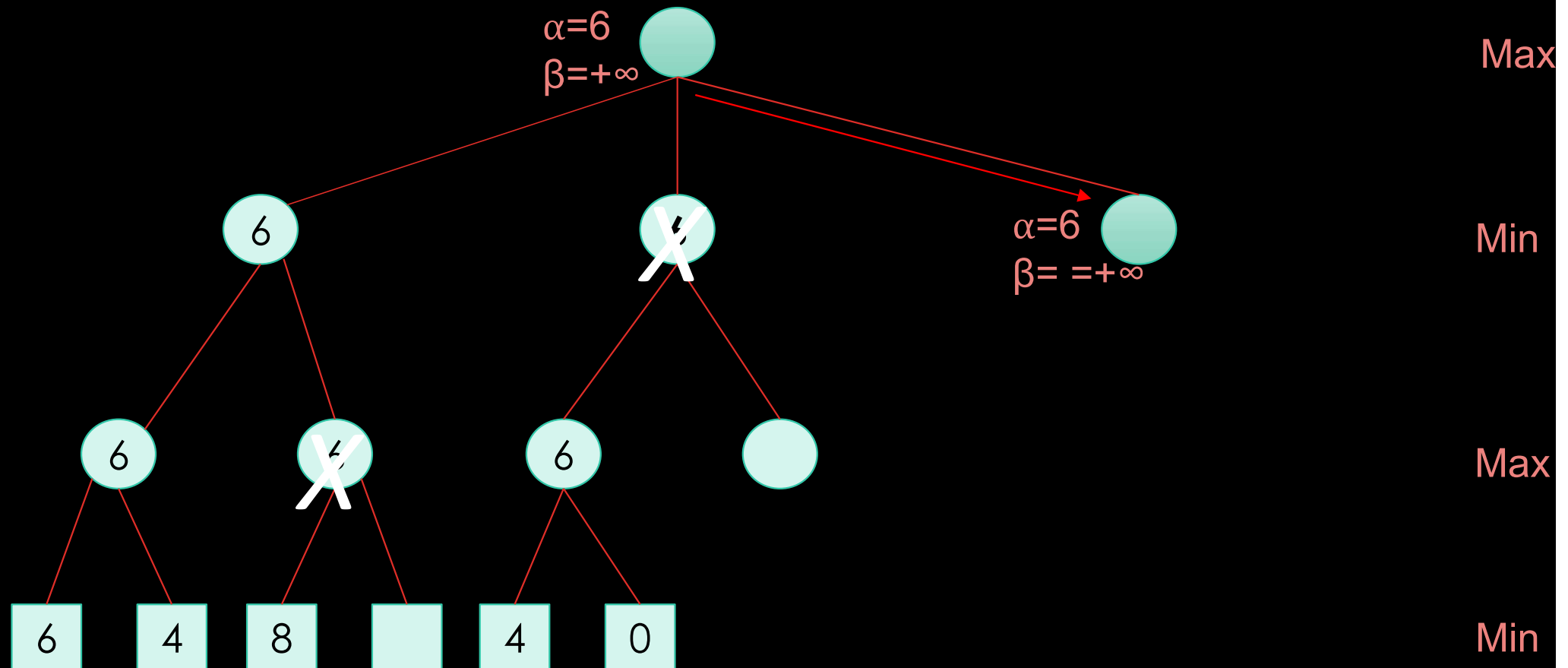
# EXEMPLO ALFABETA (26)



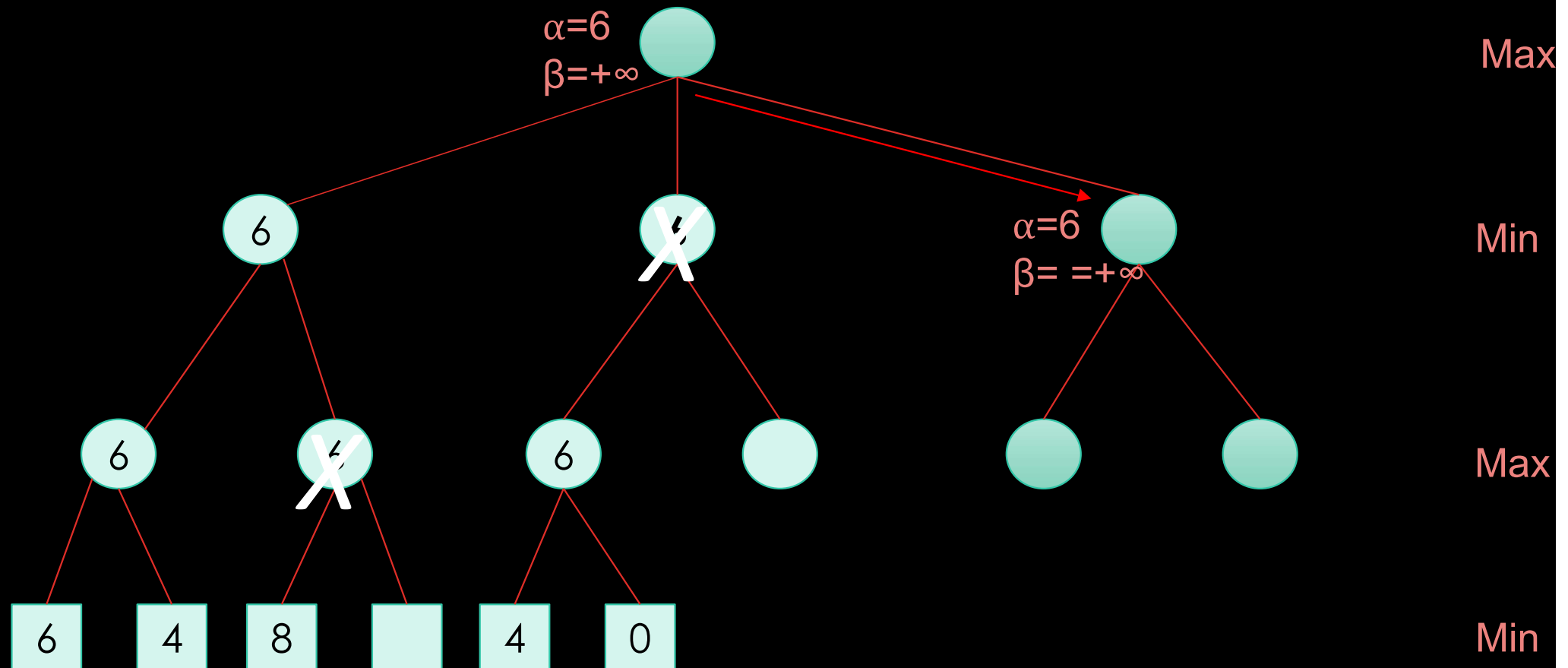
# EXEMPLO ALFABETA (27)



# EXEMPLO ALFABETA (28)

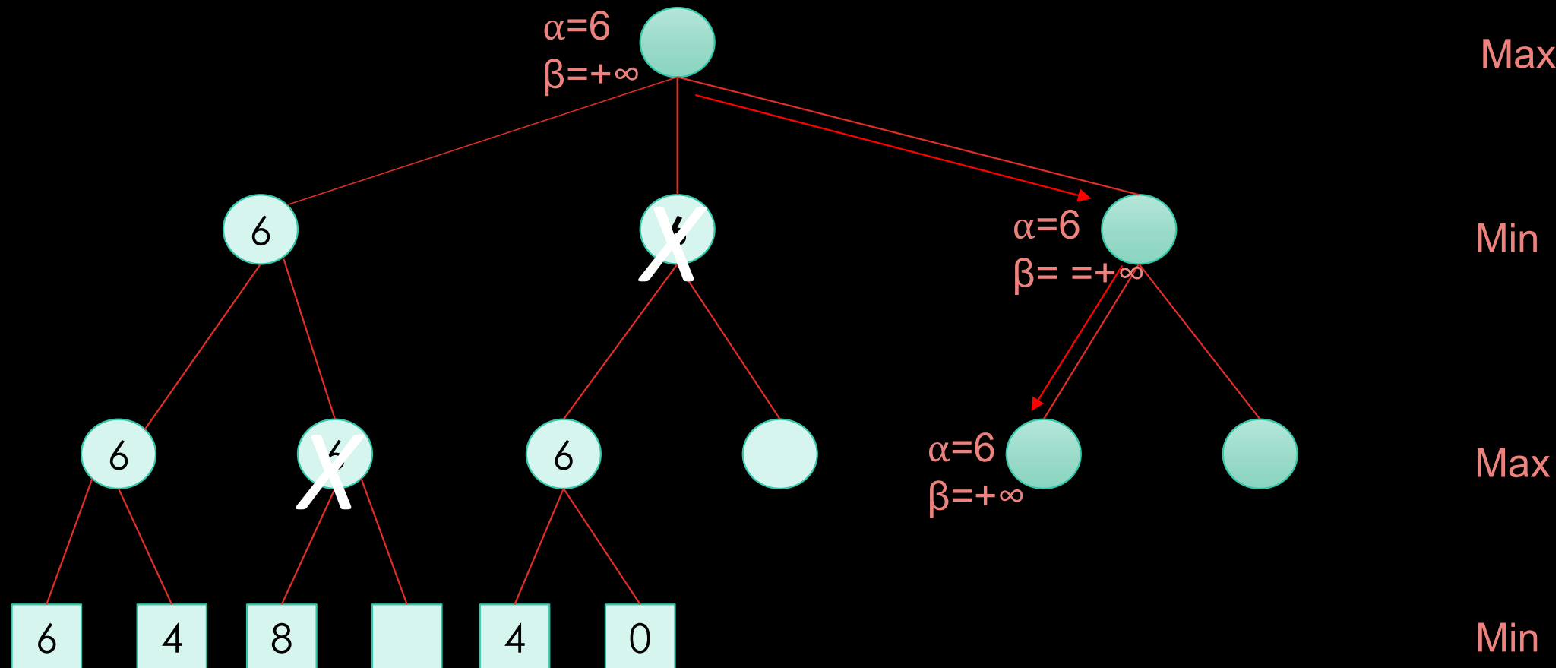


# EXEMPLO ALFABETA (29)

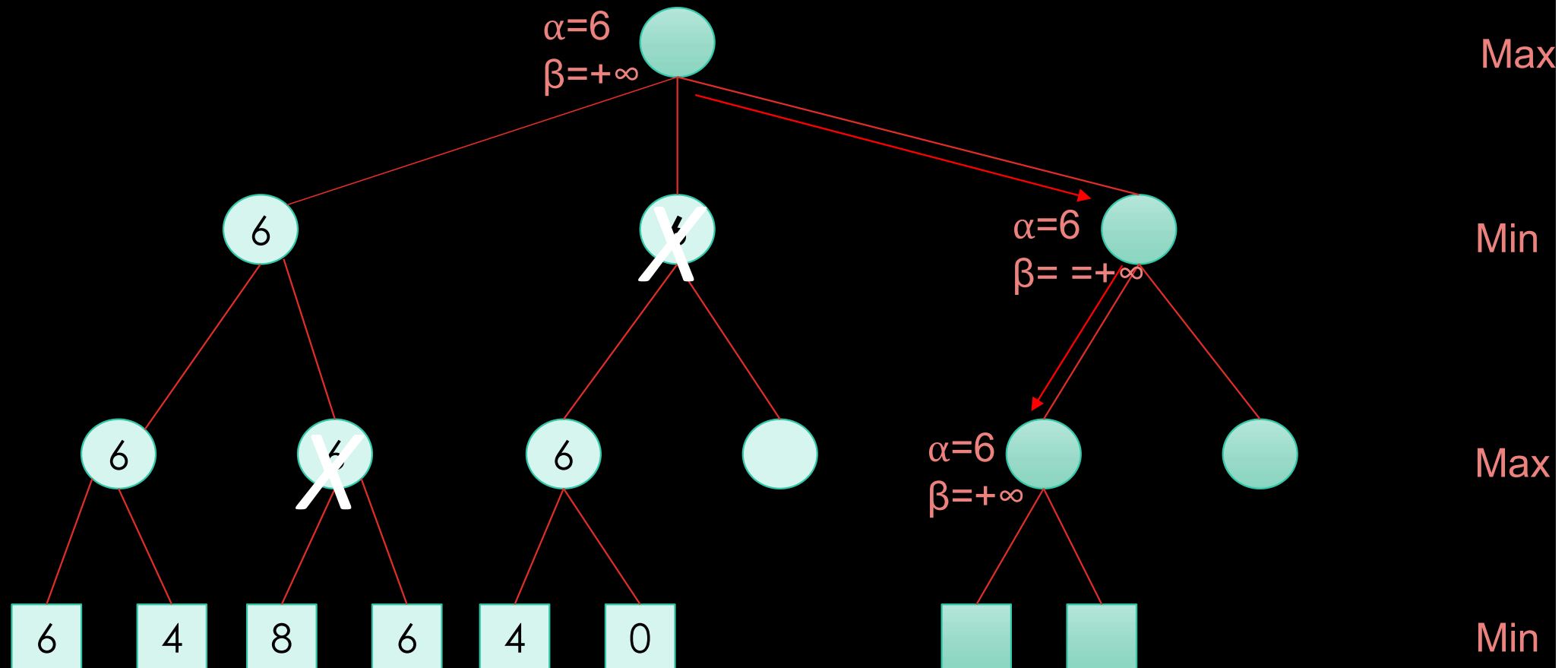




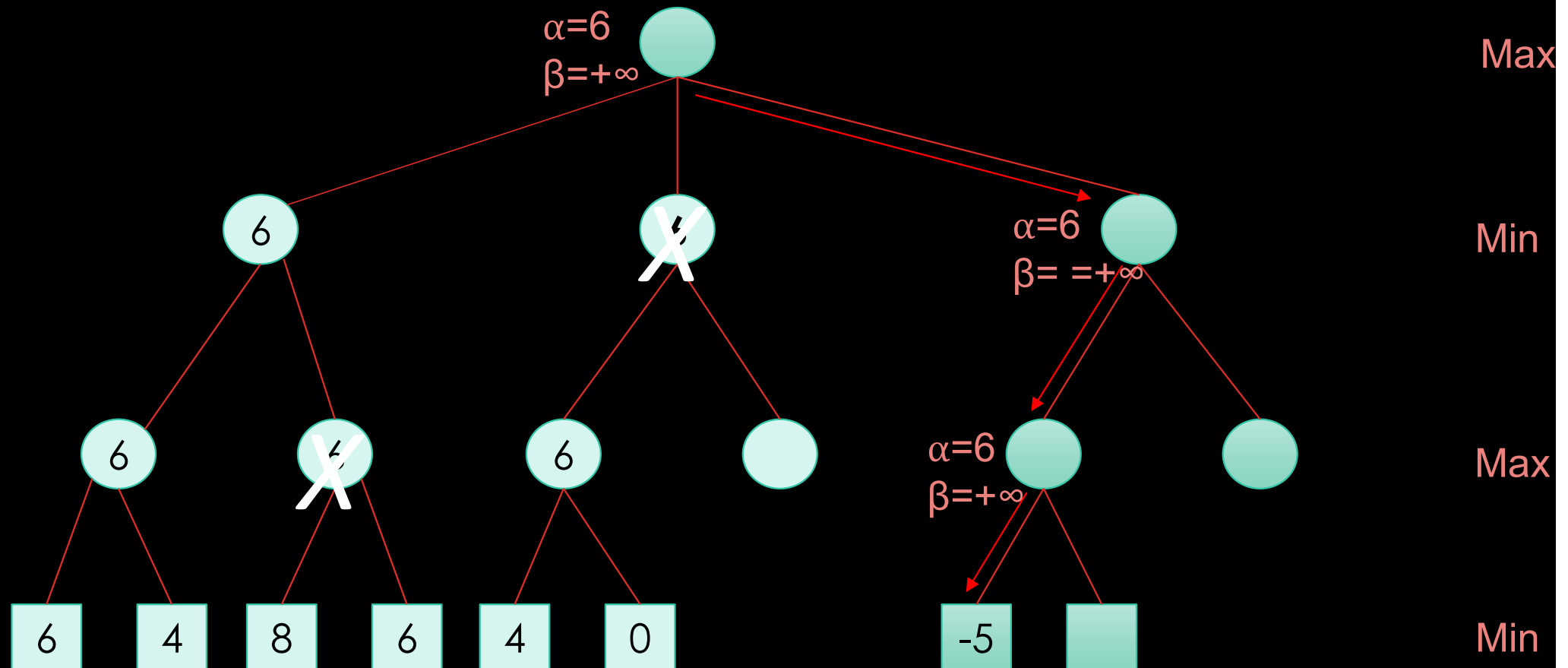
# EXEMPLO ALFABETA (30)



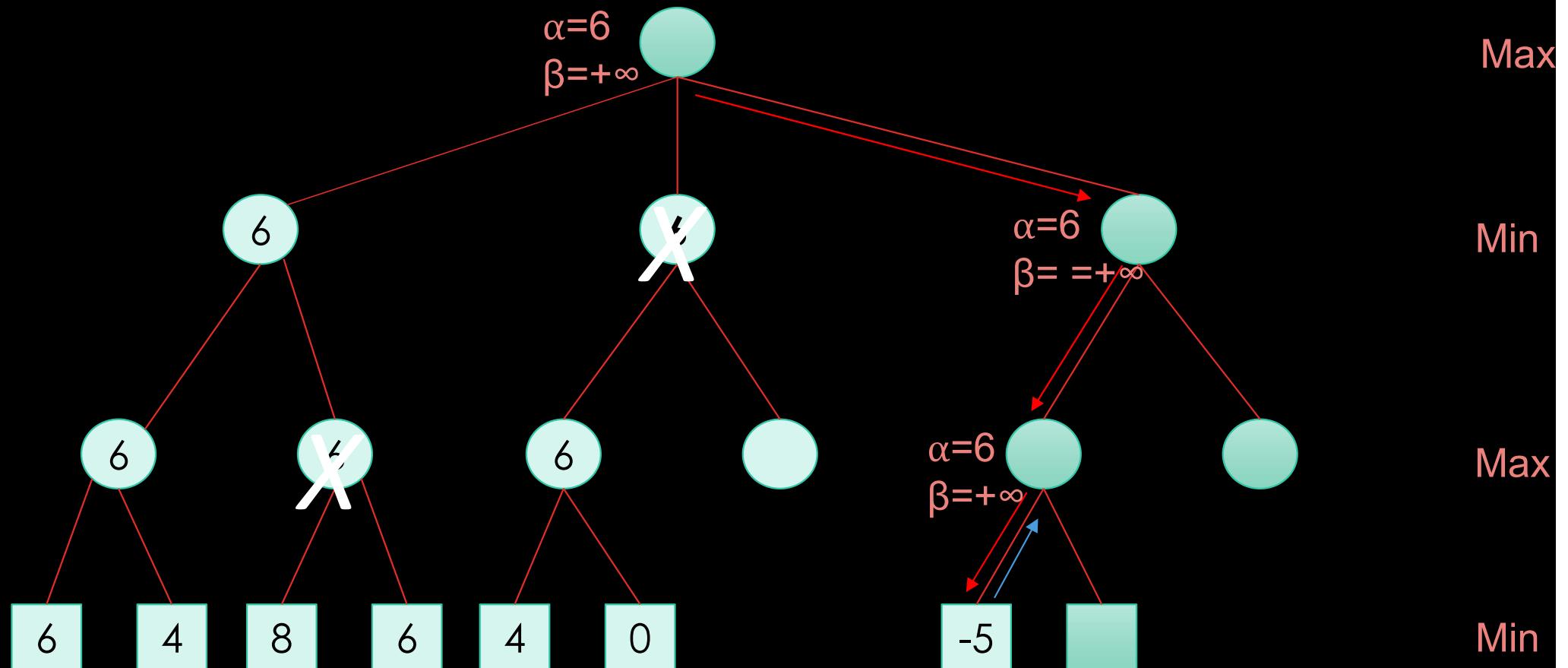
# EXEMPLO ALFABETA (31)



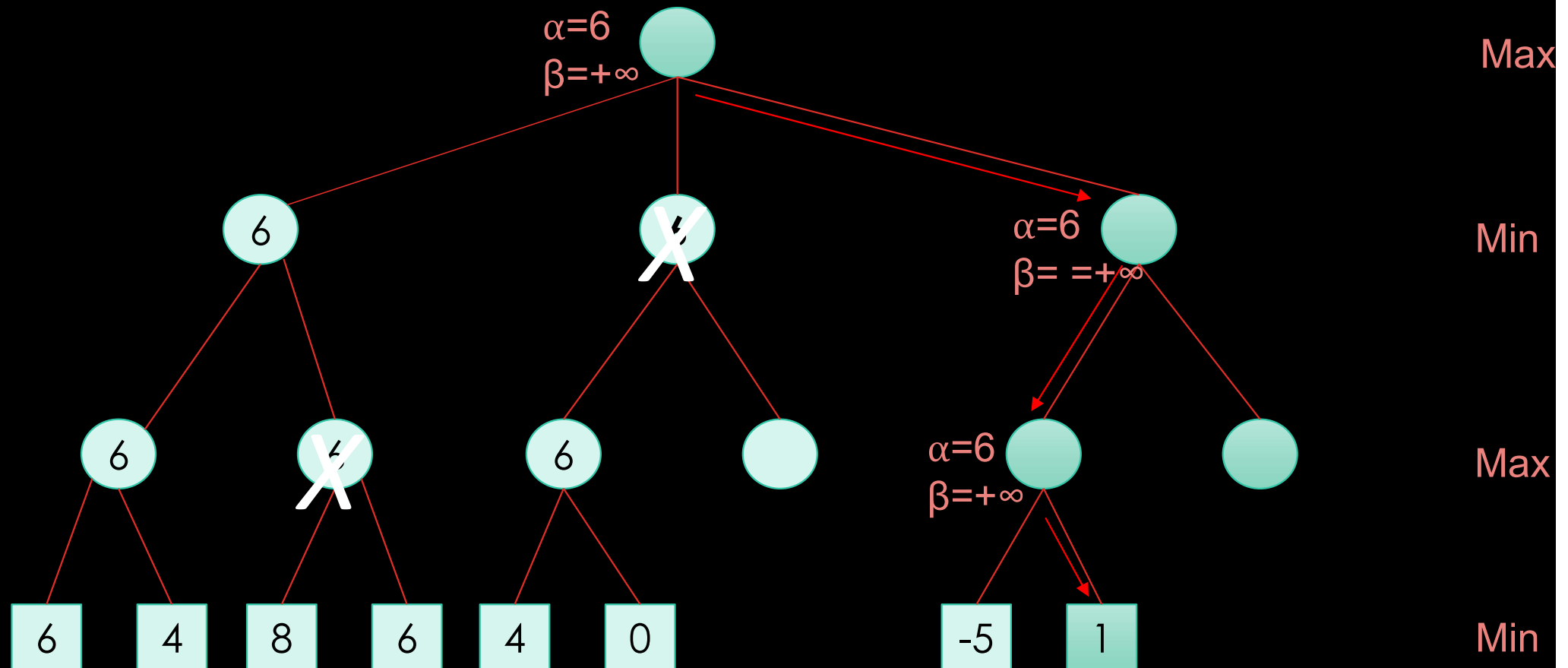
# EXEMPLO ALFABETA (32)



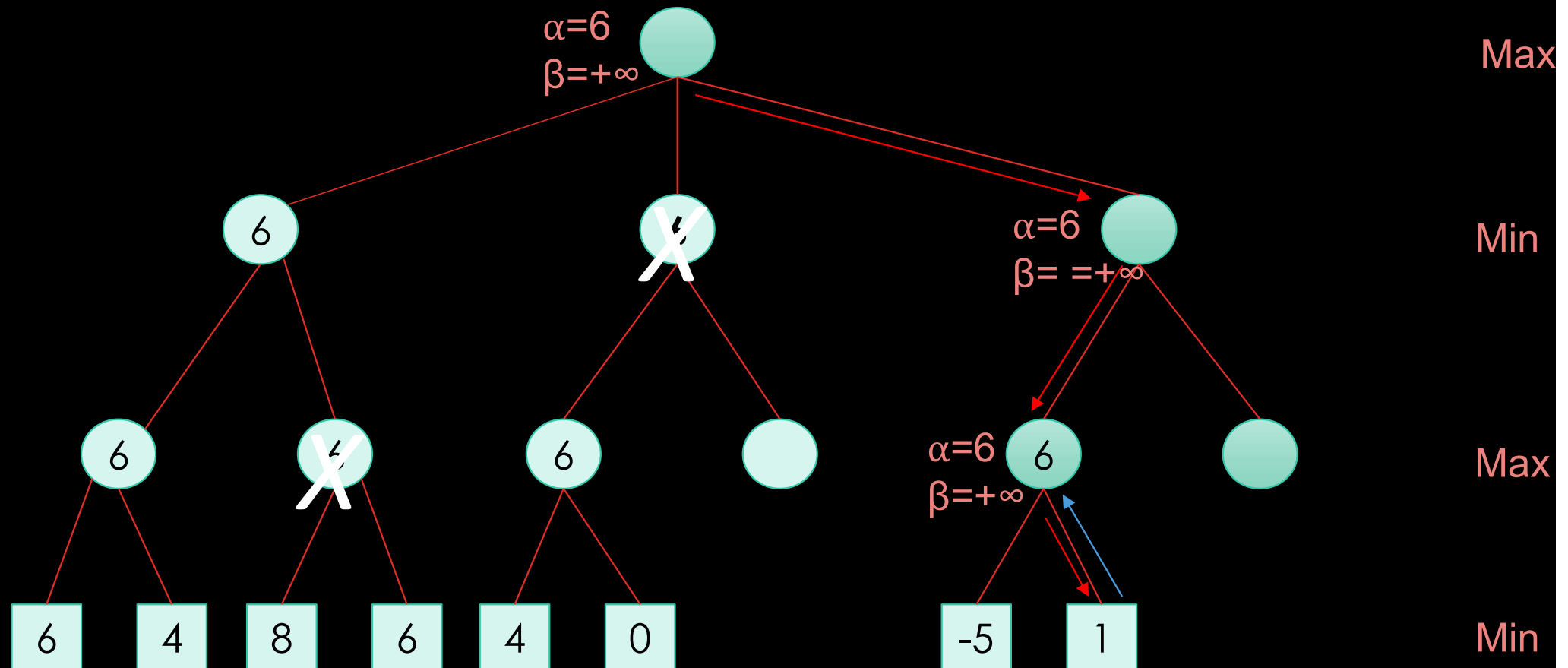
# EXEMPLO ALFABETA (33)



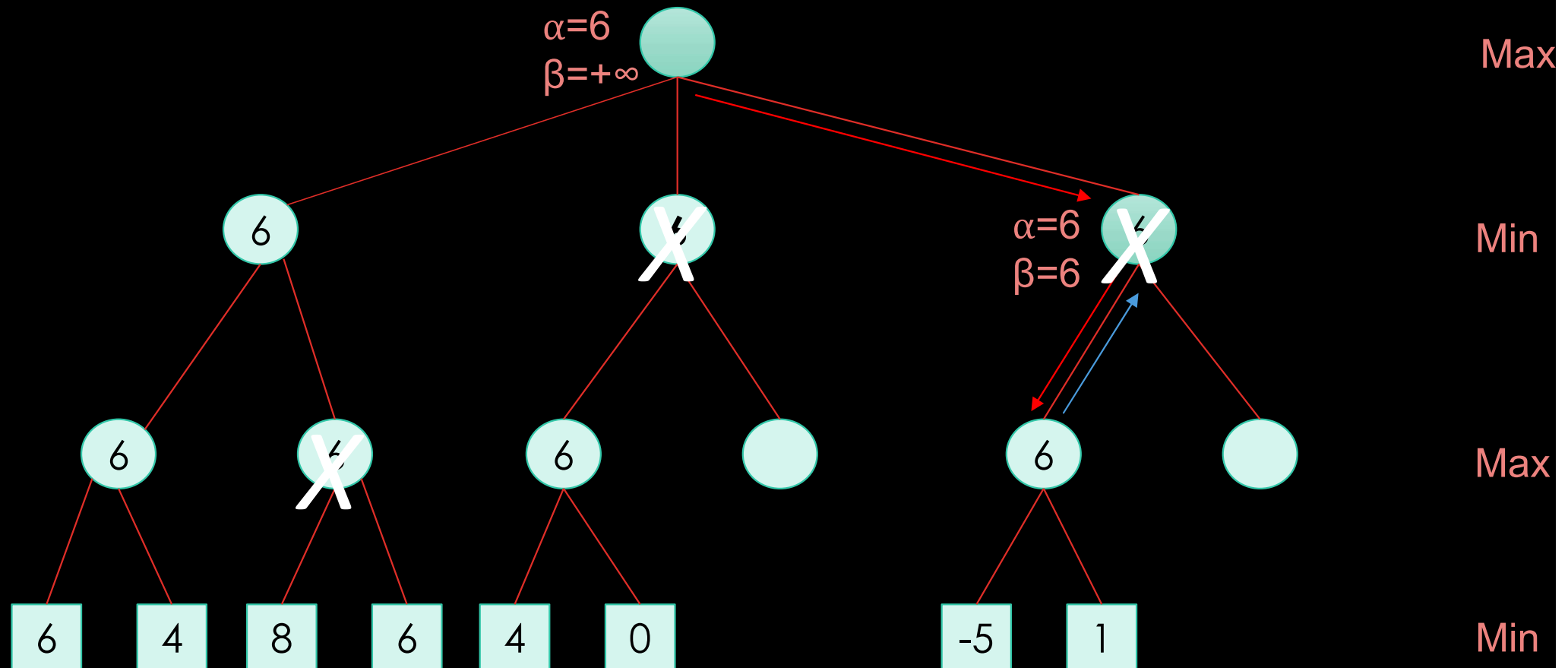
# EXEMPLO ALFABETA (34)



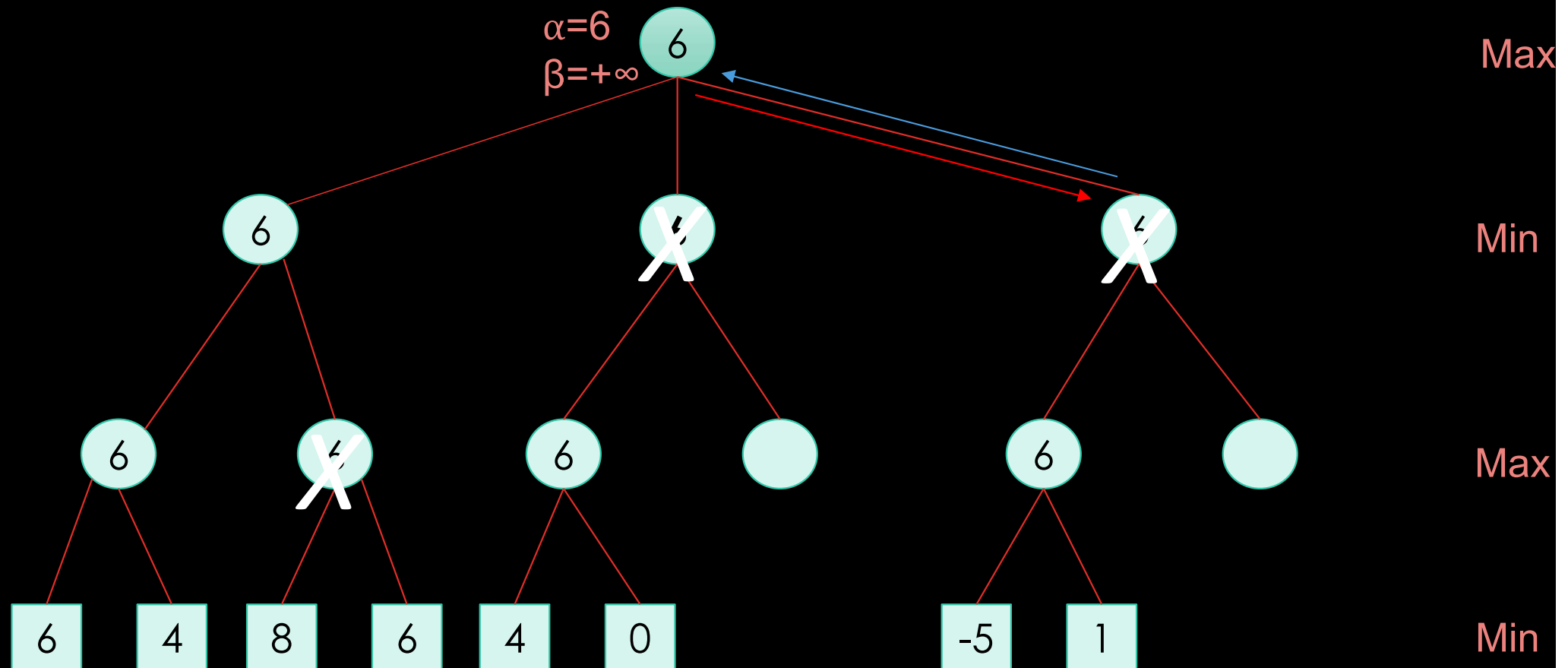
# EXEMPLO ALFABETA (35)



# EXEMPLO ALFABETA (36)

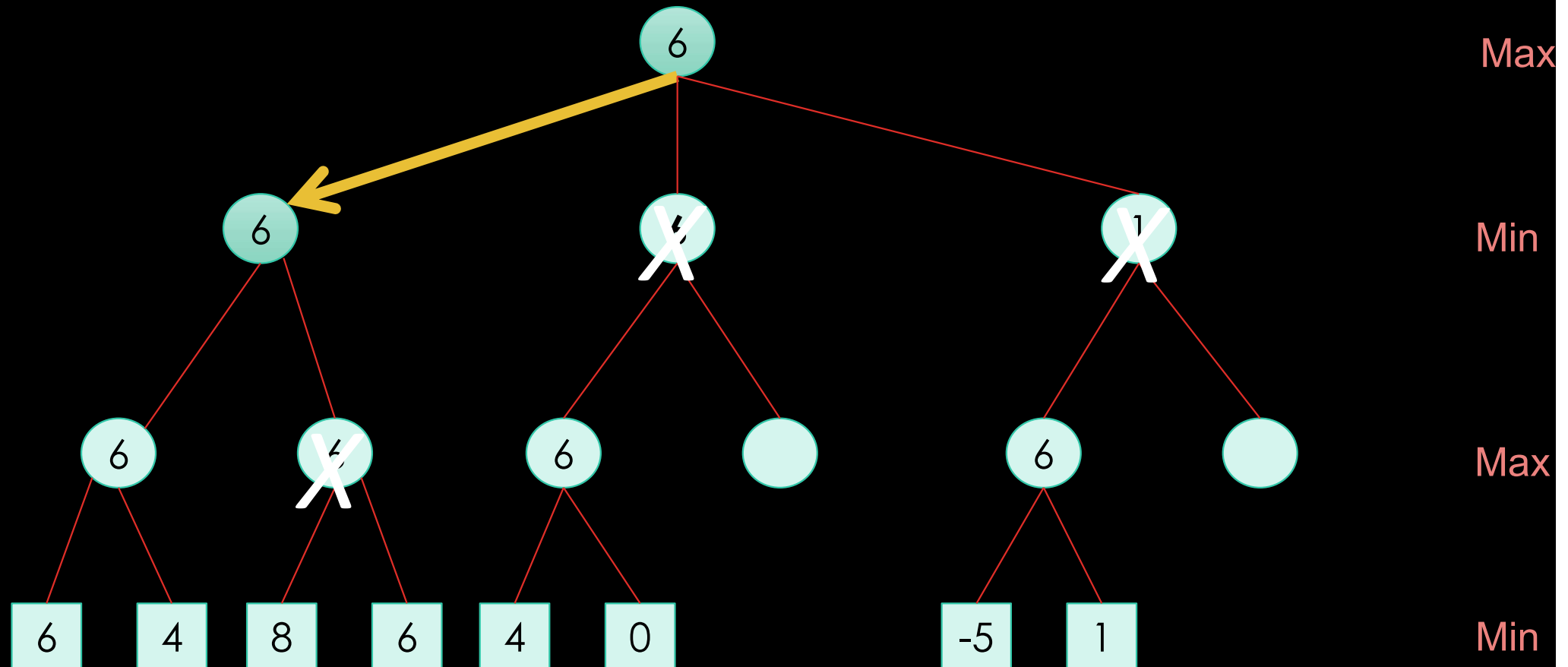


# EXEMPLO ALFABETA (37)





# EXEMPLO ALFABETA (38)



# OUTROS ALGORITMOS

- SSS\*
  - Realiza uma procura do tipo A\* em vez de DF
- DUAL\*
  - Similar ao SSS\* mas invertendo as operações do SSS\*
- SCOUT
- PVS (Principal Variation Search)  
também designado por NegaScout
- MTD(F)
- BNS

# SSS\*

- O MINIMAX com cortes alfa-beta usa uma estratégia de análise da árvore de procura do tipo depth-first.
- Seria expectável que formas de percorrer a árvore similares ao A\* pudessem dar melhores resultados.
  - Um algoritmo que usa esta estratégia é o SSS\* (George Stockman, 1979)
    - Existe uma lista de ABERTOS ordenada por ordem decrescente do seu valor (heurístico).
    - O problema é que em cada passo é preciso remover o nó com o maior valor da lista de ABERTOS e sempre que um nó MAX interior (não-folha) é resolvido todos os descendentes diretos e indiretos têm de ser removidos de ABERTOS. Estes 2 passos ocupam mais de 90% do tempo de CPU.
    - O SSS\* foi por isso declarado impraticável.

# SCOUT

- Proposto por Judea Perl em 1980
- A ideia é:
  - Quando se está a analisar um nó MAX (por exemplo) admite-se que ele deverá ter um valor mínimo admissível  $v_{\min}$  e por isso primeiro verifica-se se cada sucessor desse nó poderá devolver um valor maior do que  $v_{\min}$ .
    - Se não: então não há necessidade de analisar esse sucessor
    - Se sim: analisar o sucessor
  - Para os nós MIN é o simétrico
- Este algoritmo requer uma heurística para determinar o valor de  $v_{\min}$ .

# SCOUT (ALGORITMO)

- Em cada lance o jogador tem 1 de  $b$  possibilidades legais (*branching*)
- O jogo é pesquisado até à profundidade  $d$  (*depth*)
- Os nós folha são avaliados com uma função estática  $v_0$
- O algoritmo usa duas funções recursivas: EVAL e TEST.
  - EVAL( $S$ ) devolve o valor minimax da posição  $S$ ,  $V(S)$ .
  - TEST( $S, v, >$ ) devolve o valor lógico da desigualdade  $V(S) > v$  em que  $v$  é um dado valor de referência.

# SCOUT (TEST & EVAL)

TEST( $S, v, >$ )

Aplica-se o teste aos sucessores:

- Se  $S$  é MAX dá Verdade logo que um sucessor é maior que  $v$ ; Falso c.c.
- Se  $S$  é MIN dá Verdade logo que um sucessor é menor ou igual a  $v$ ; Falso c.c.

EVAL( $S$ ) – para um nó MAX:

- Avalia começando por avaliar (i.e. EVAL( $S_1$ )) o seu primeiro sucessor;
- Depois faz o “scouting” dos restantes sucessores fazendo o respetivo TEST:  $V(S_k) > V(S_1)$ .
- Se o resultado para  $S_k$  for Verdade então o nó é realmente avaliado usando EVAL( $S_k$ ) e o seu valor passa a ser usado para o “scouting” dos restantes sucessores.
- Após todos os sucessores terem sido avaliados, o último valor obtido é dado como  $V(S)$ .
- O EVAL( $S$ ) de um nó MIN é similar e simétrico fazendo o TEST:  $V(S_k) < V(S_1)$ .

# SCOUT (CONCL.)

- O algoritmo SCOUT parece desperdiçar tempo pois qualquer nó  $S_k$  que passe o teste é submetido de novo para avaliação.
- No entanto, uma análise mais cuidada revela que o desperdício não é muito significativo e apesar de algum esforço duplicado, a redução do fator de ramificação compensa para análises com 4 ou mais níveis de profundidade.
- Esta vantagem do SCOUT pode deteriorar-se para fatores de ramificação muito elevados.

# SCOUT VS. ALFABETA

Search Depth	Random Ordering			Dynamic Ordering		
	SCOUT	$\alpha$ - $\beta$	% Improvement	SCOUT	$\alpha$ - $\beta$	% Improvement
2	82	70	-17.0	39	37	-5.4
3	394	380	-3.7	62	61	-1.6
4	1173	1322	+11.3	91	96	-1.1
5	2514	4198	+40.1	279	336	+17.0
6	5111	6944	+26.4	371	440	+15.7

Pearl, J. (1980) **SCOUT**: "A SIMPLE GAME-SEARCHING ALGORITHM WITH PROVEN OPTIMAL PROPERTIES", in AAAI-80 Proceedings.



# PVS/NEGASCOUT

## PRINCIPAL VARIATION SEARCH

- O algoritmo NEGASCOUT é similar ao NEGAMAX mas para o SCOUT.
- O NEGASCOUT (aliás à semelhança do SCOUT) funciona melhor se o primeiro nó for de facto o melhor do seu nível (variação principal) para que o TEST dos restantes os isente do EVAL.
- Se estiver a ser aplicado um método *iterative deepening*, o melhor nó do nível  $k$  deverá ser o primeiro a ser considerado no nível  $k+1$  (variação principal).

# MTD(F) (MEMORY-ENHANCED TEST DRIVER)

- Algoritmo desenvolvido em 1994 por Aske Plaat et al.
- Usa uma função de teste similar à do algoritmo “TEST” de Judea Pearl, que realiza procuras alfa-beta com janelas nulas.
- Uma janela nula causa mais cortes mas devolve menos informação: apenas a fronteira do valor minimax.
- Para encontrar o valor minimax, o MTD(f) chama o alfabeta várias vezes, convergindo para o valor exato.
- A utilização de uma tabela de transposição permite evitar a reexploração das partes da árvore já anteriormente exploradas.

# PSEUDO-CÓDIGO DO MTD(F)

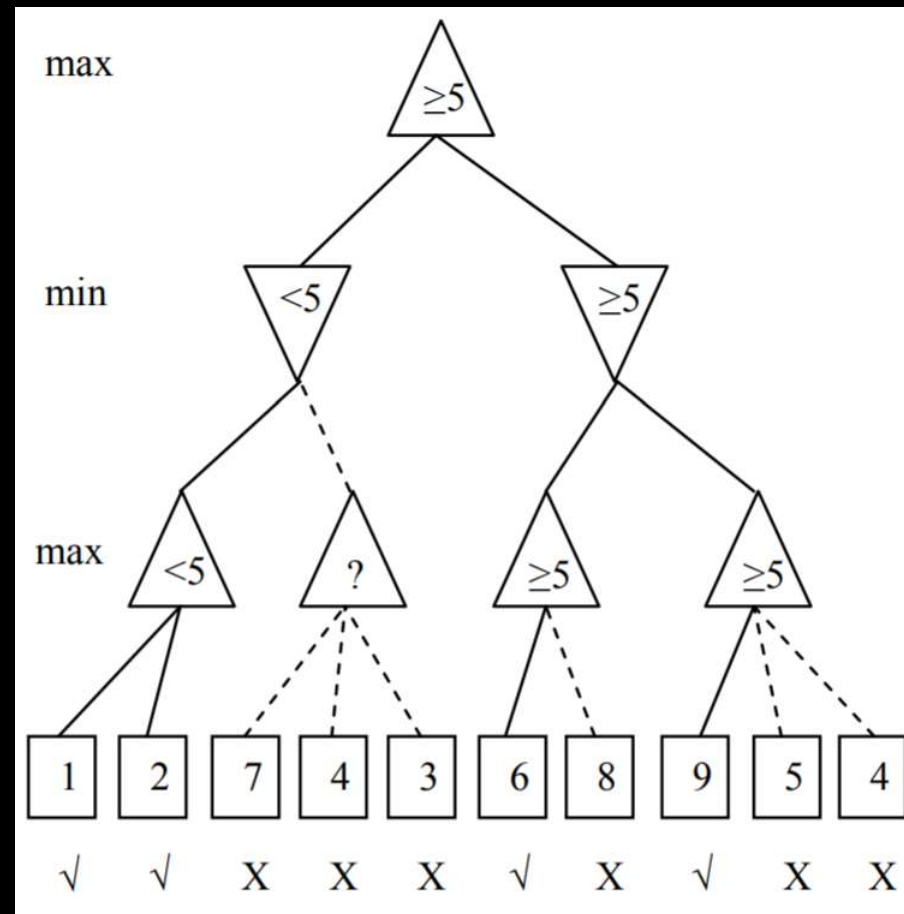
```
function MTDF(root, f, d)
  g := f
  upperBound :=  $+\infty$ 
  lowerBound :=  $-\infty$ 
  while lowerBound < upperBound
     $\beta := \max(g, \text{lowerBound} + 1)$ 
    g := AlphaBetaWithMemory(root,  $\beta - 1$ ,  $\beta$ , d)
    if g <  $\beta$  then
      upperBound := g
    else
      lowerBound := g
  return g
```

<https://askeplaat.wordpress.com/534-2/mtdf-algorithm/>

# BEST NODE SEARCH (BNS)

- Desenvolvido em 2011. Aparentemente, experiências realizadas com árvores aleatórias mostram que é o algoritmo minimax mais eficiente.
- É uma evolução do MTD(F).
- Indica qual a jogada ideal do ponto de vista do minimax mas não indica o seu valor.
- Determina qual a sub-árvore em que o minimax é maior ou menor que um dado valor (adivinhado), mudando este valor até que o alfa e o beta estão suficientemente próximos ou apenas uma subárvore é maior que o valor adivinhado.

# BNS



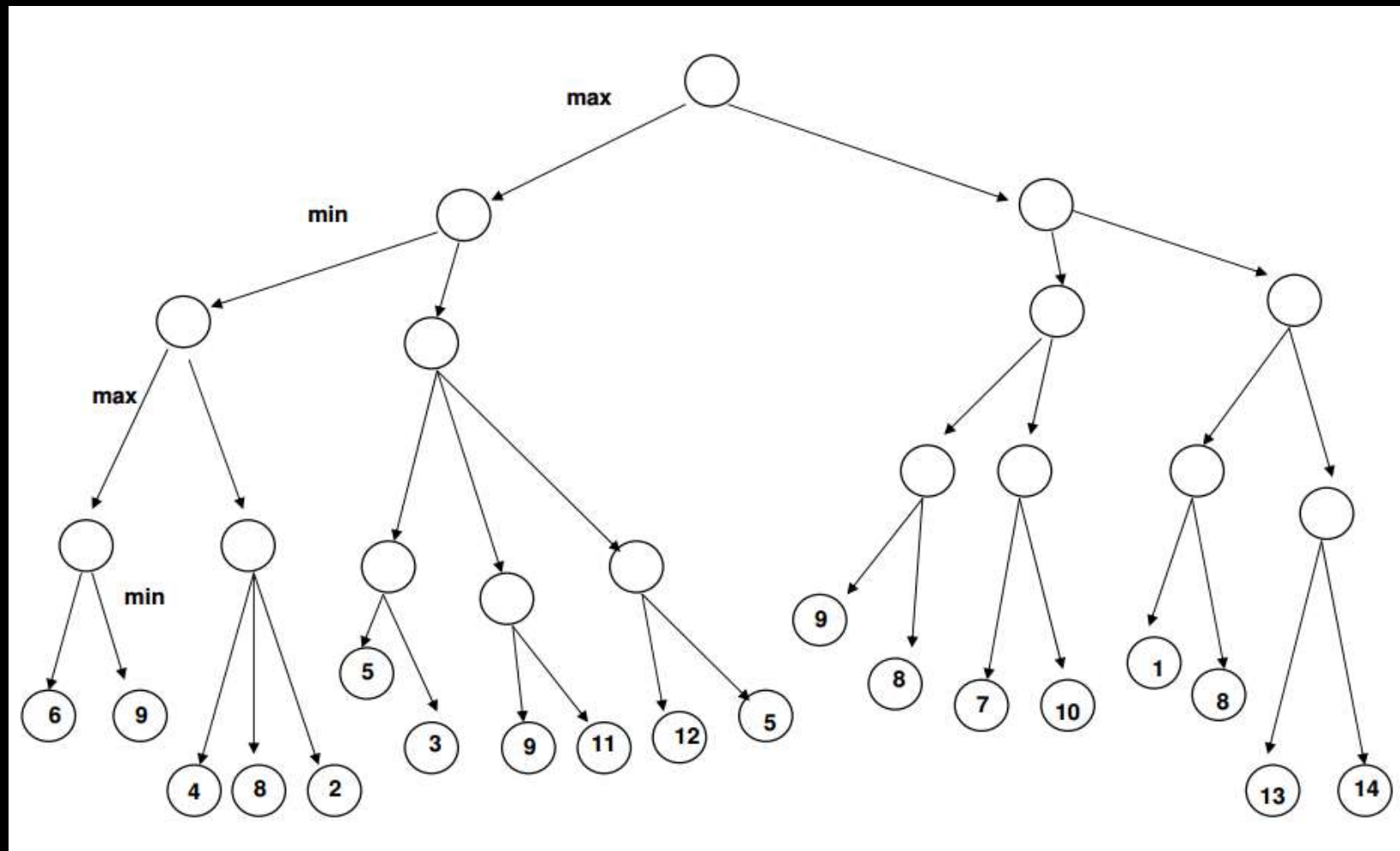
Rutko, D. (2011) "Fuzzified Algorithm for Game Tree Search with Statistical and Analytical Evaluation" in Scientific Papers, University of Latvia. Vol. 770

# PSEUDO-CÓDIGO (BNS)

```
function BNS(node,  $\alpha$ ,  $\beta$ )
subtreeCount := number of children of node
do
  test := NextGuess( $\alpha$ ,  $\beta$ , subtreeCount)
  betterCount := 0
  foreach child of node
    bestVal := -AlphaBeta(child, -test, -(test - 1))
    if bestVal  $\geq$  test
      betterCount := betterCount + 1
      bestNode := child
  //update number of sub-trees that exceeds separation test value
  //update alpha-beta range
while not(( $\beta$  -  $\alpha$  < 2) or (betterCount = 1))
return bestNode
```

# EXERCICIO

- Aplicar o alfa-beta para calcular o valor do nó raiz.





# RESOLUÇÃO

- Percorrendo a árvore da esquerda para a direita:  
*Os valores indicados nos nós são os do minimax e não do alfabeto*

