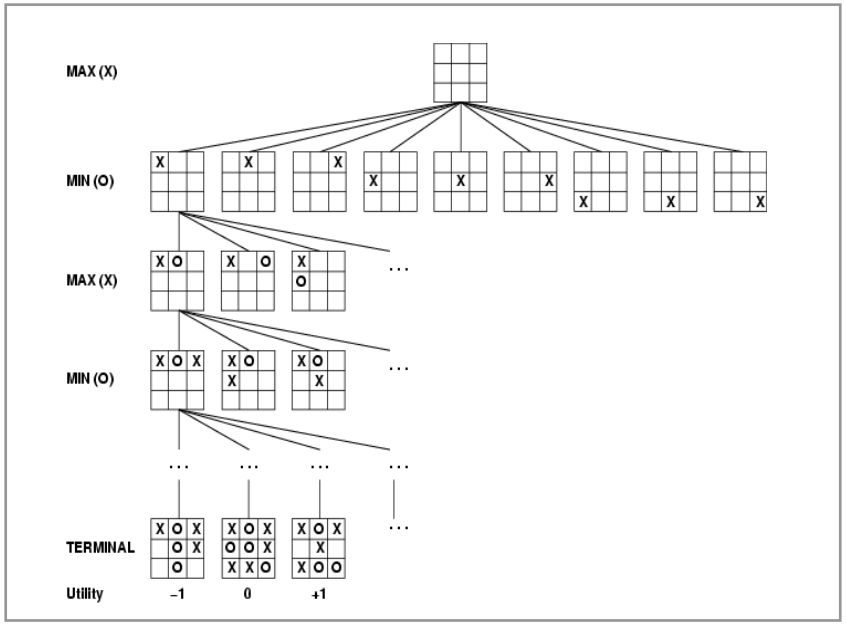
Jogos e Busca



Silvio Lago
slago@ime.usp.br

Sumário

- Jogos adversariais
- Algoritmo MINIMAX
- Algoritmo de poda α-β
- Função de avaliação e corte
- Jogos de sorte

Jogos

- Ambientes competitivos, em que as metas dos agentes estão em conflito, dão origem a problemas de busca competitiva.
- Em IA, geralmente, jogos são problemas de busca competitiva em ambientes determinísticos e completamente observáveis, em que:
  - existem dois agentes cujas ações se alternam;
  - os valores de utilidade final são sempre simétricos.

Jogos

- Primeiros pesquisadores de jogos (1950):
  - Conrad Zuse, Claude Shannon e Alan Turing
- Jogos são considerados:
  - um desafio à inteligência humana:
    - são problemas muito difíceis de se resolver.
  - um assunto atraente para estudo em IA:
    - é fácil representar o estado corrente de um jogo;
    - há um pequeno número de ações cujos resultados são definidos por regras precisas;
    - apresentam enormes espaços de busca.

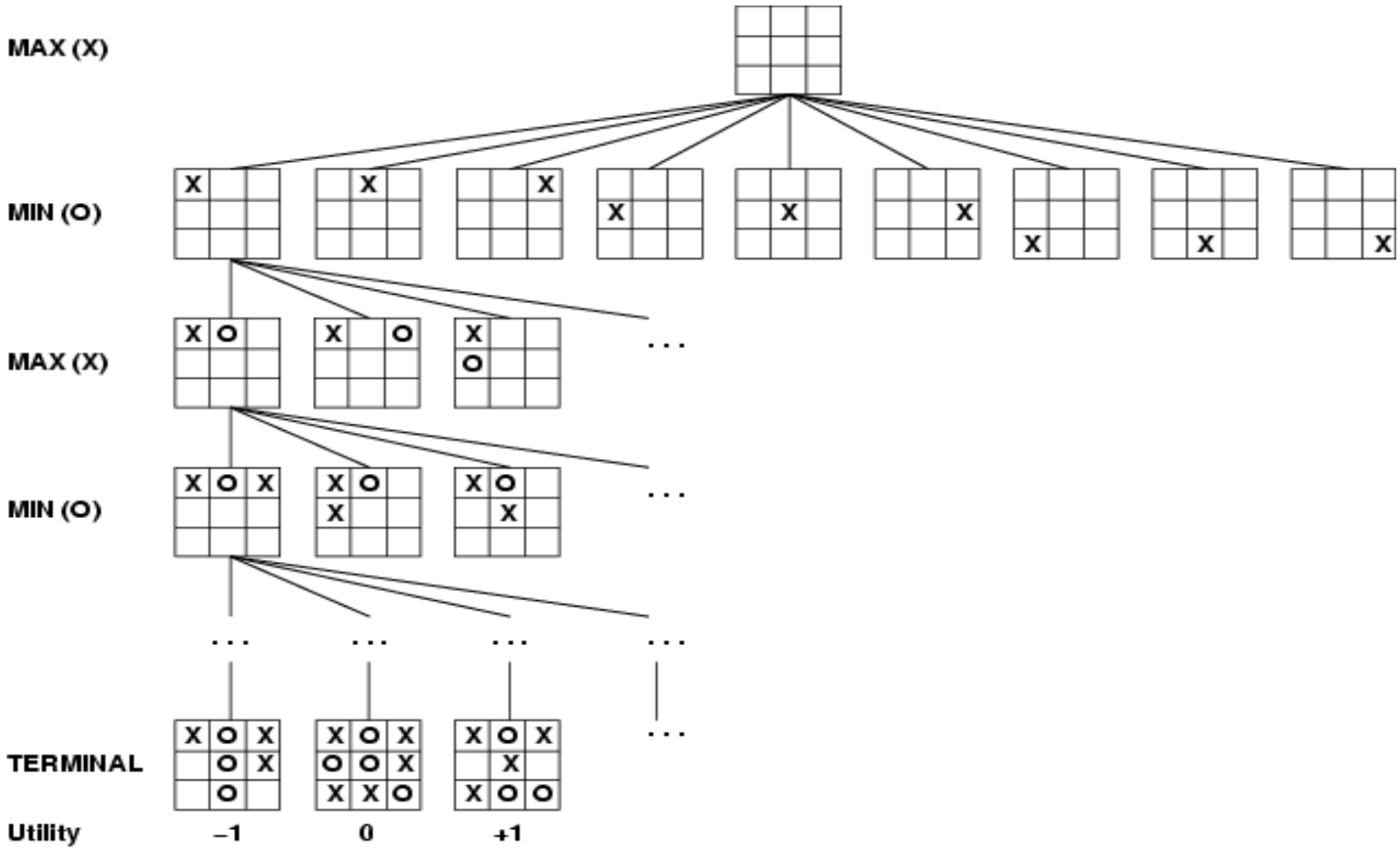
Exemplo: xadrez

- requer inteligência (raciocínio correto)
- fácil representar a configuração do tabuleiro
- pequeno número de regras de movimentação
- problema de busca complexo:
  - *fator de ramificação médio* ≈ 35
  - *número de movimentos por jogador* ≈ 50
  - *profundidade da solução* ≈ 100
  - *árvore de busca* $\approx 35^{100} \approx 10^{154}$

Jogos Adversariais

- Características de um jogo adversarial:
  - jogadores adversários que se revezam (MAX e MIN)
  - MAX faz o primeiro movimento (planejamos para ele)
  - vencedor (perdedor) recebe pontos (penalidades)
  - no final, a soma de pontos e penalidades é nula
- Jogo adversarial é um problema de busca com:
  - *estado inicial*
  - *função sucessor*
  - *teste de término*
  - *função utilidade*

Exemplo: árvore para jogo-da-velha



Estratégia ótima

- MAX deve:
  - maximizar sua chance de vitória
  - considerar que MIN minimizará sua chance vitória
- Questão:
  - Como usar a árvore de jogo para encontrar uma estratégia ótima que, ***independentemente das escolhas feitas pelo adversário***, garanta que MAX fará sempre o melhor possível?

VALOR-MINIMAX

- Suposição: jogadores jogam otimamente!!!
- Uma estratégia ótima pode ser determinada usando-se o valor-minimax de cada nó:

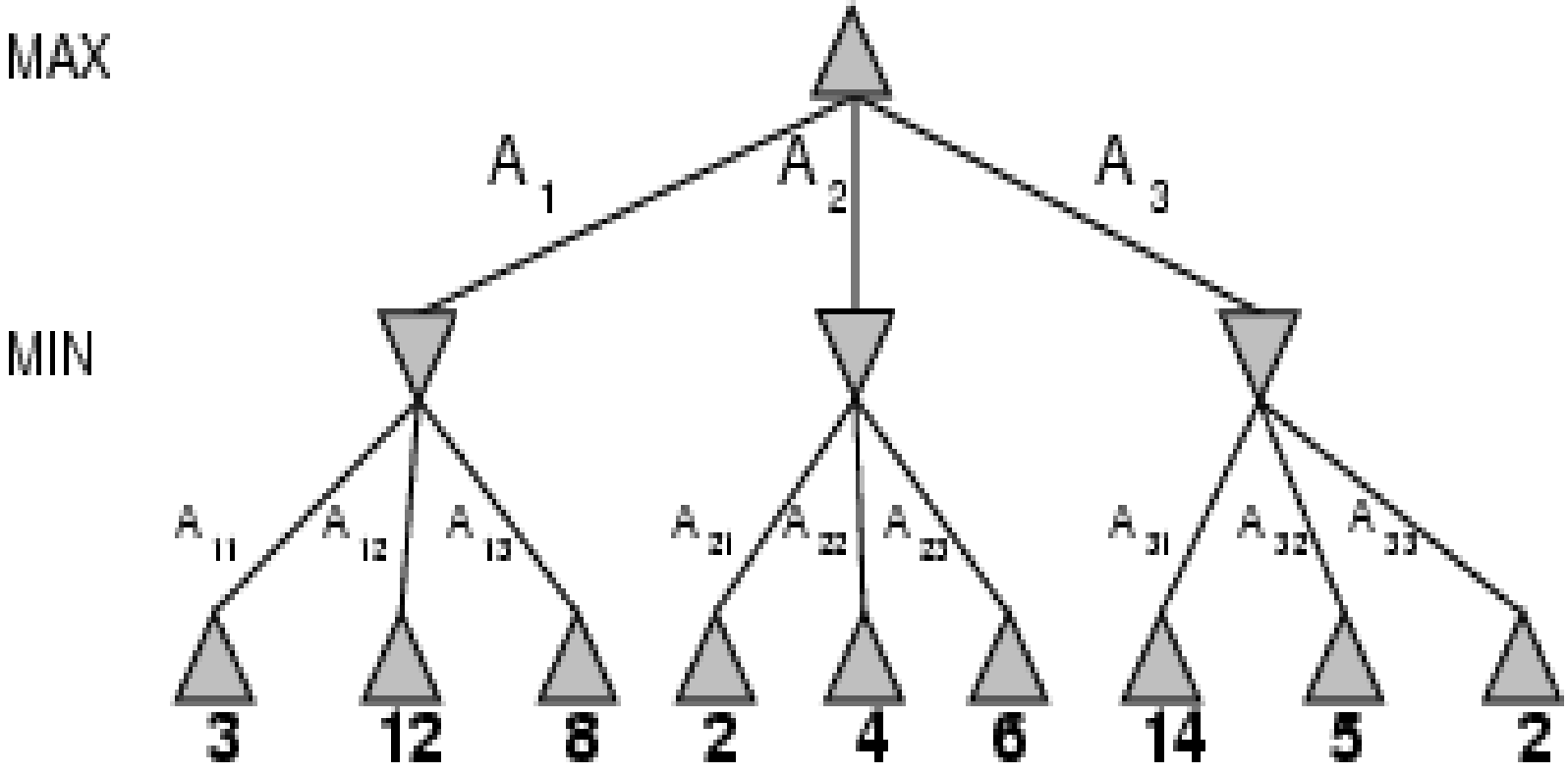
VALOR-MINIMAX(n) =

UTILIDADE(n)	se n é um nó terminal
$\max_{s \in sucessores(n)}$ VALOR-MINIMAX(s)	se n é um nó max
$\min_{s \in sucessores(n)}$ VALOR-MINIMAX(s)	se n é um nó min

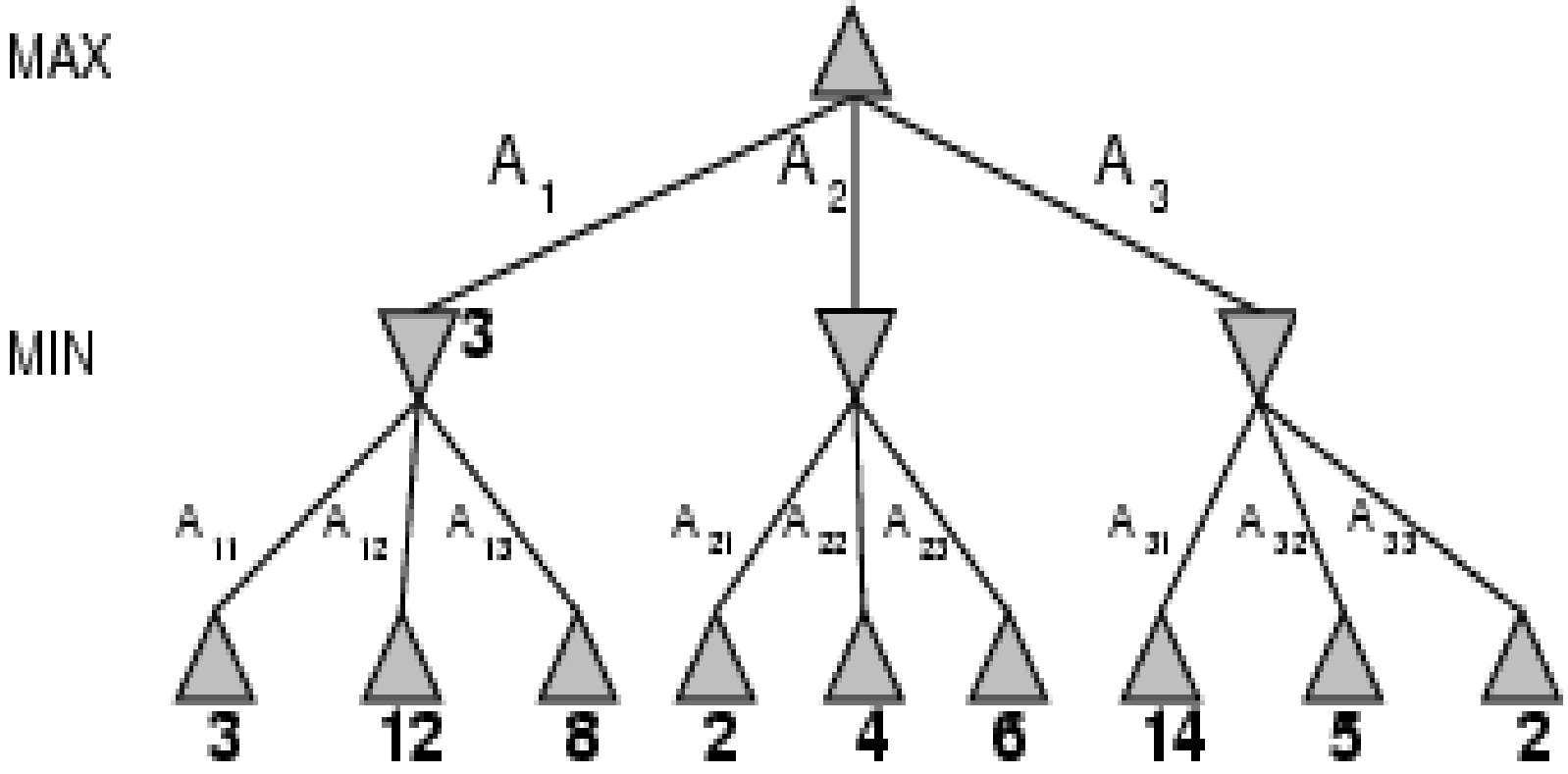
Escolha da melhor ação num estado *s*

- gerar a árvore de busca completa, a partir de *s*
- aplicar a função utilidade a cada estado terminal
- propagar os valores de utilidade, a partir das folhas, até a raiz *s* (MIN escolhe mínimo e MAX escolhe máximo)
- escolher a ação que produz o valor utilidade associado ao estado *s*.

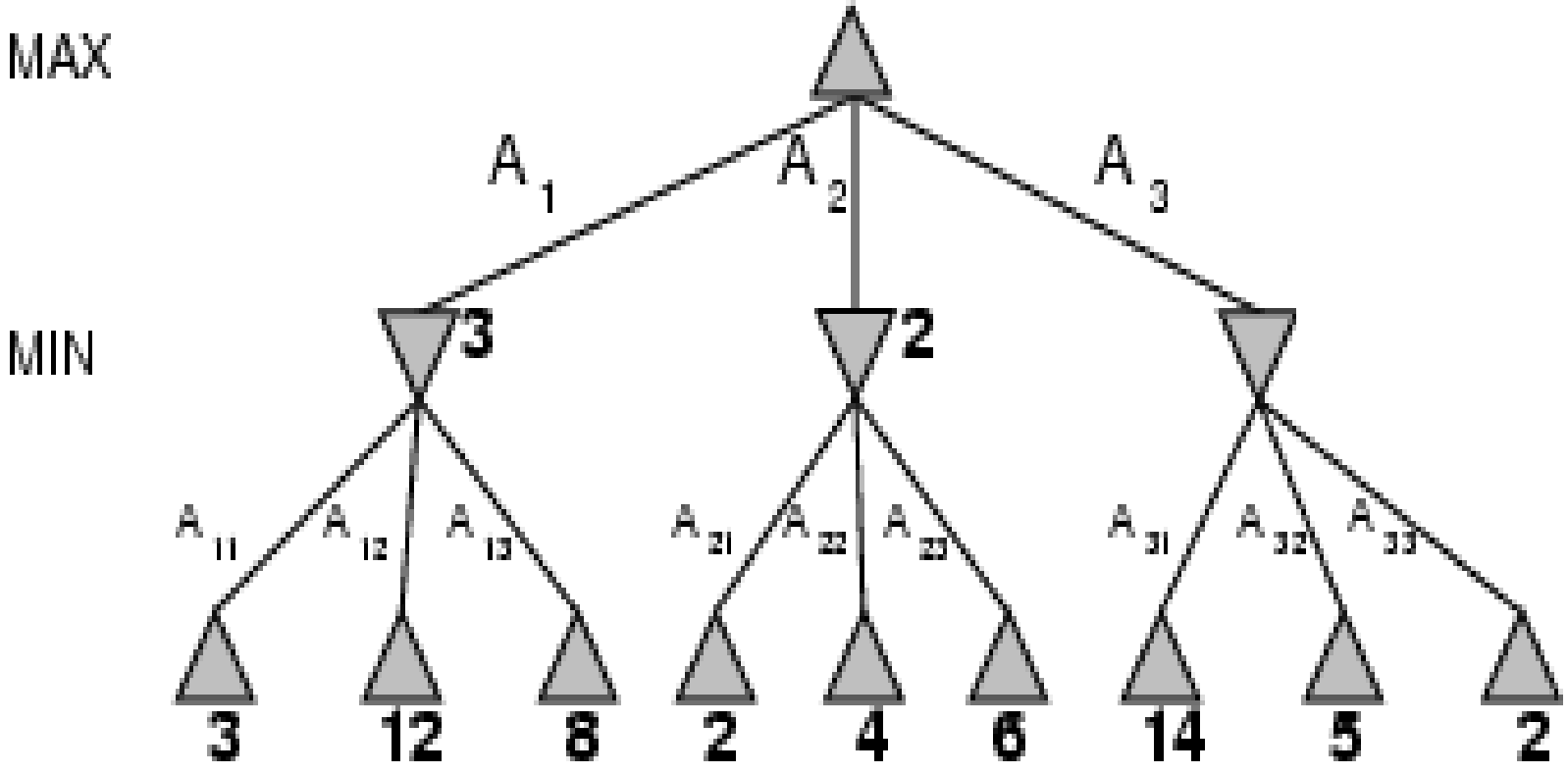
Exemplo: um jogo de duas jogadas



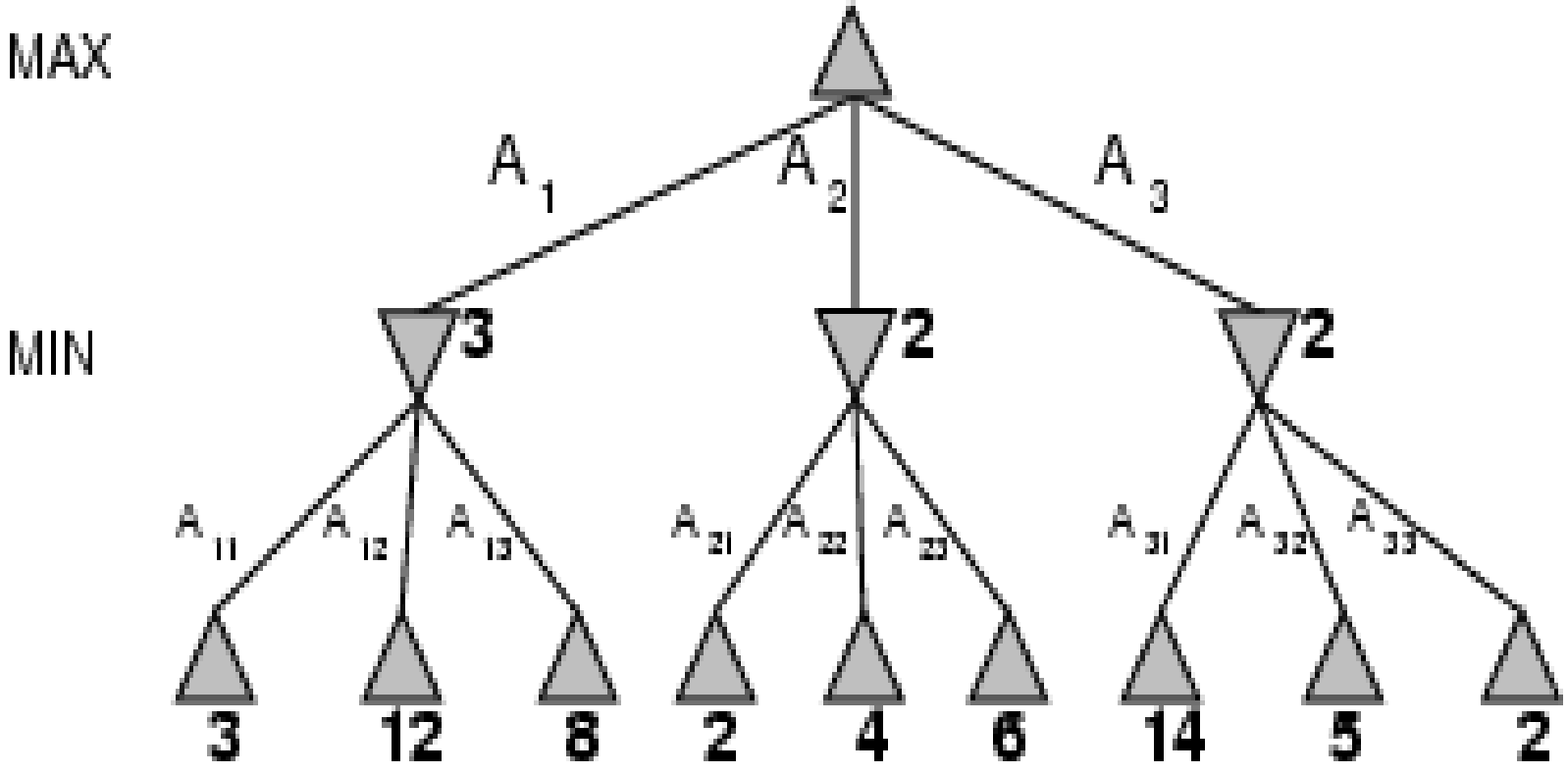
Exemplo: um jogo de duas jogadas



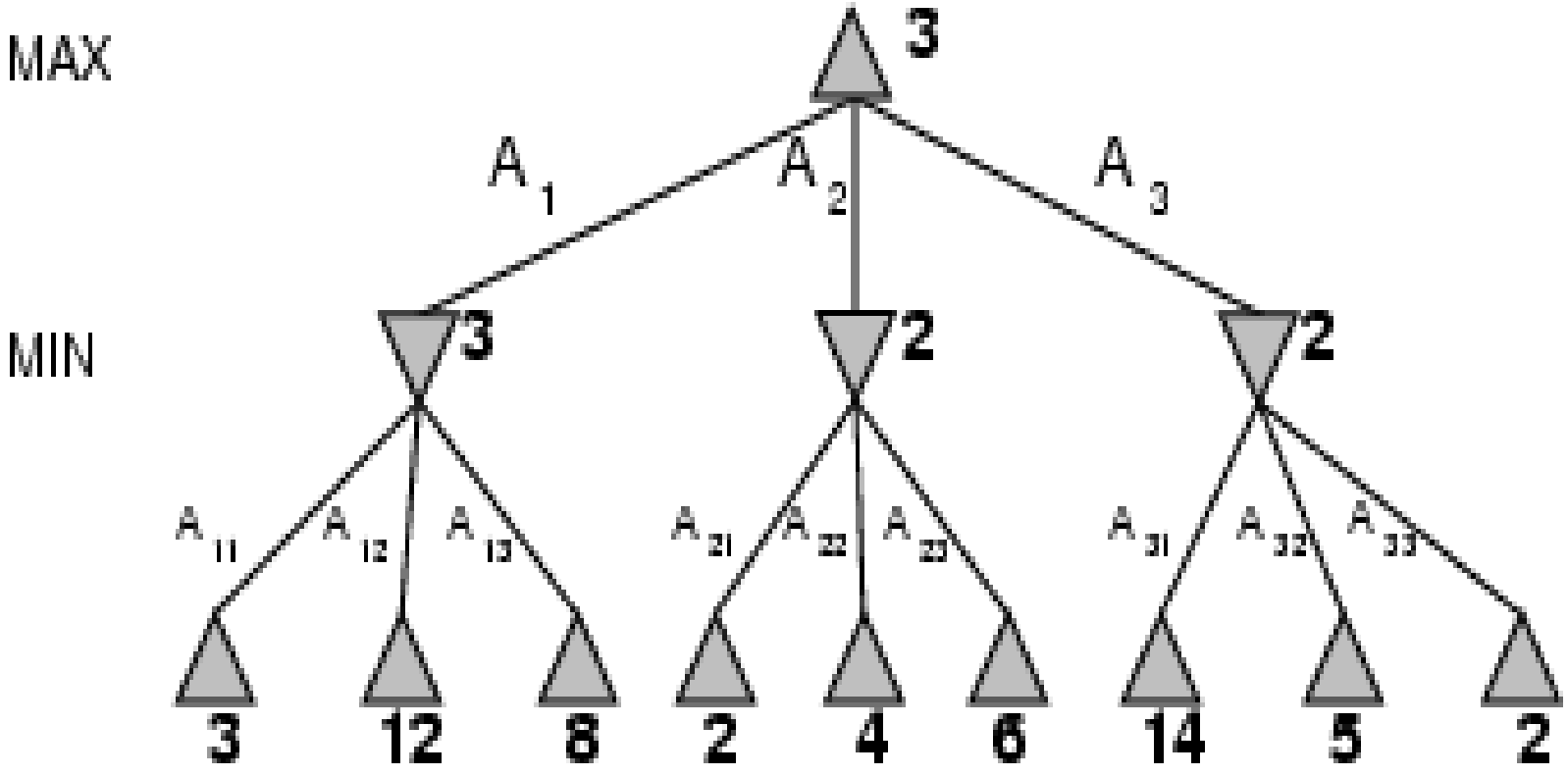
Exemplo: um jogo de duas jogadas



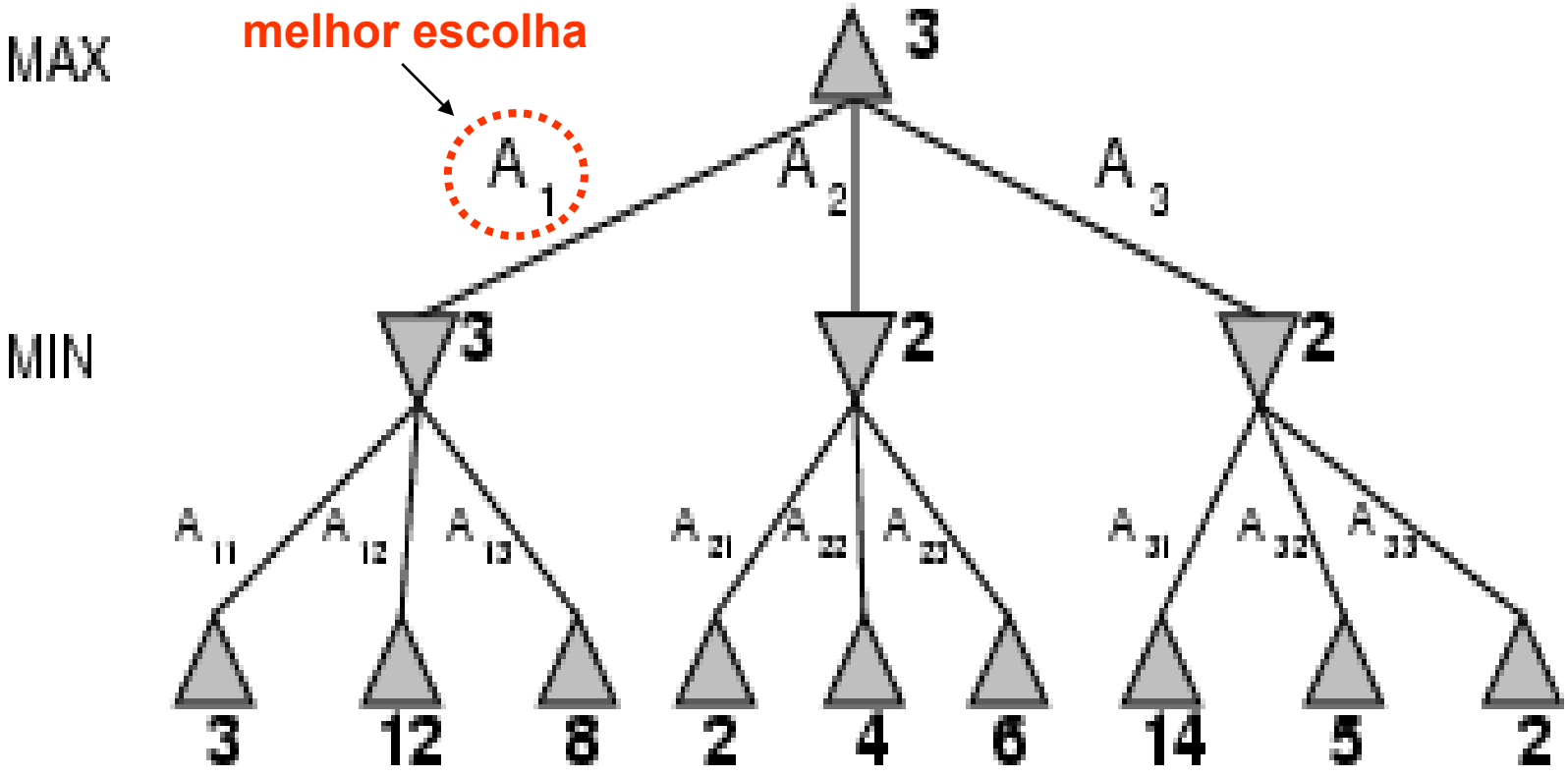
Exemplo: um jogo de duas jogadas



Exemplo: um jogo de duas jogadas



Exemplo: um jogo de duas jogadas



Melhor escolha: maximiza o resultado para MAX no pior caso!!!

Algoritmo MINIMAX

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

Propriedades do MINIMAX

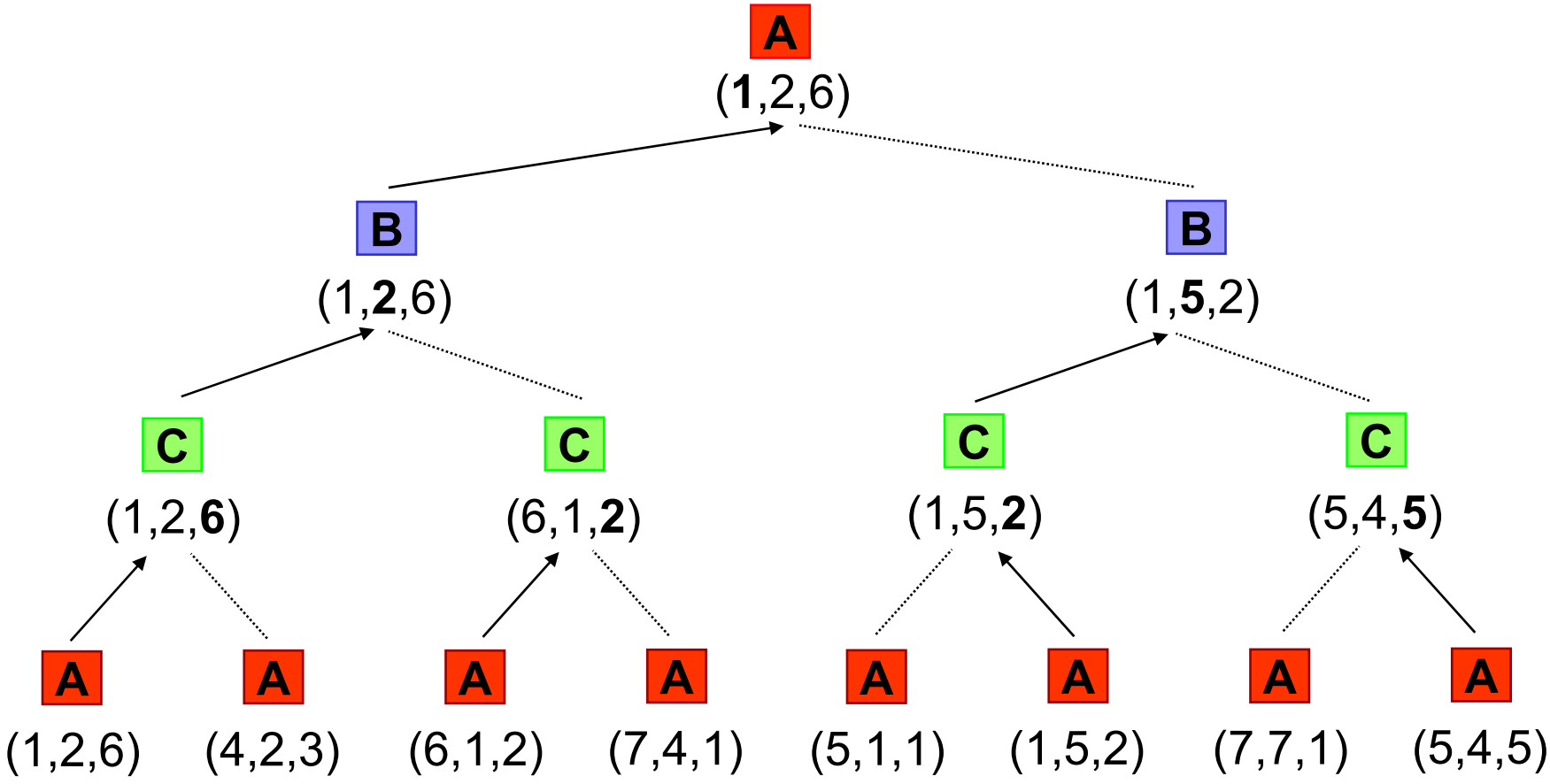
- completo? sim, se a árvore for finita
- ótimo? sim, se o adversário for ótimo
- tempo? $O(b^m)$
- espaço? $O(b \times m)$ ou $O(m)$

Para jogos de xadrez, com b ≈ 35 e m ≈ 100, é praticamente impossível encontrar uma solução exata!!!

Jogos com vários jogadores

- muitos jogos permitem mais de dois jogadores
- valor utilidade $\Rightarrow$ vetor de valores utilidades
- alianças $\Rightarrow$ maximiza (minimiza) subvetor

Exemplo: minimax com três jogadores



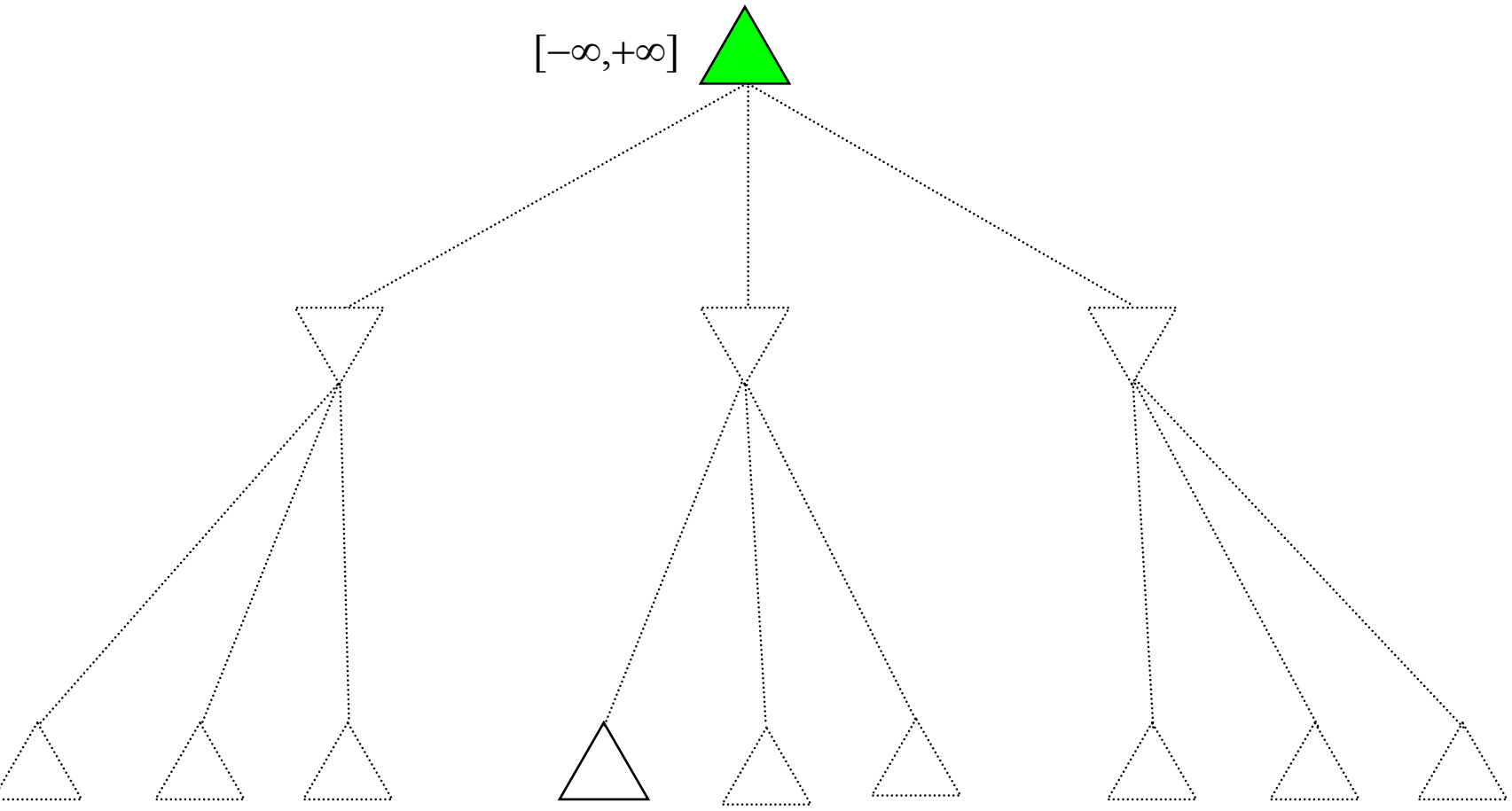
Poda α-β

- na maioria dos casos, não é possível explorar toda a árvore de busca
- a poda α-β consiste numa estratégia segura para eliminar ramos da árvore de busca, sem ter que examiná-los

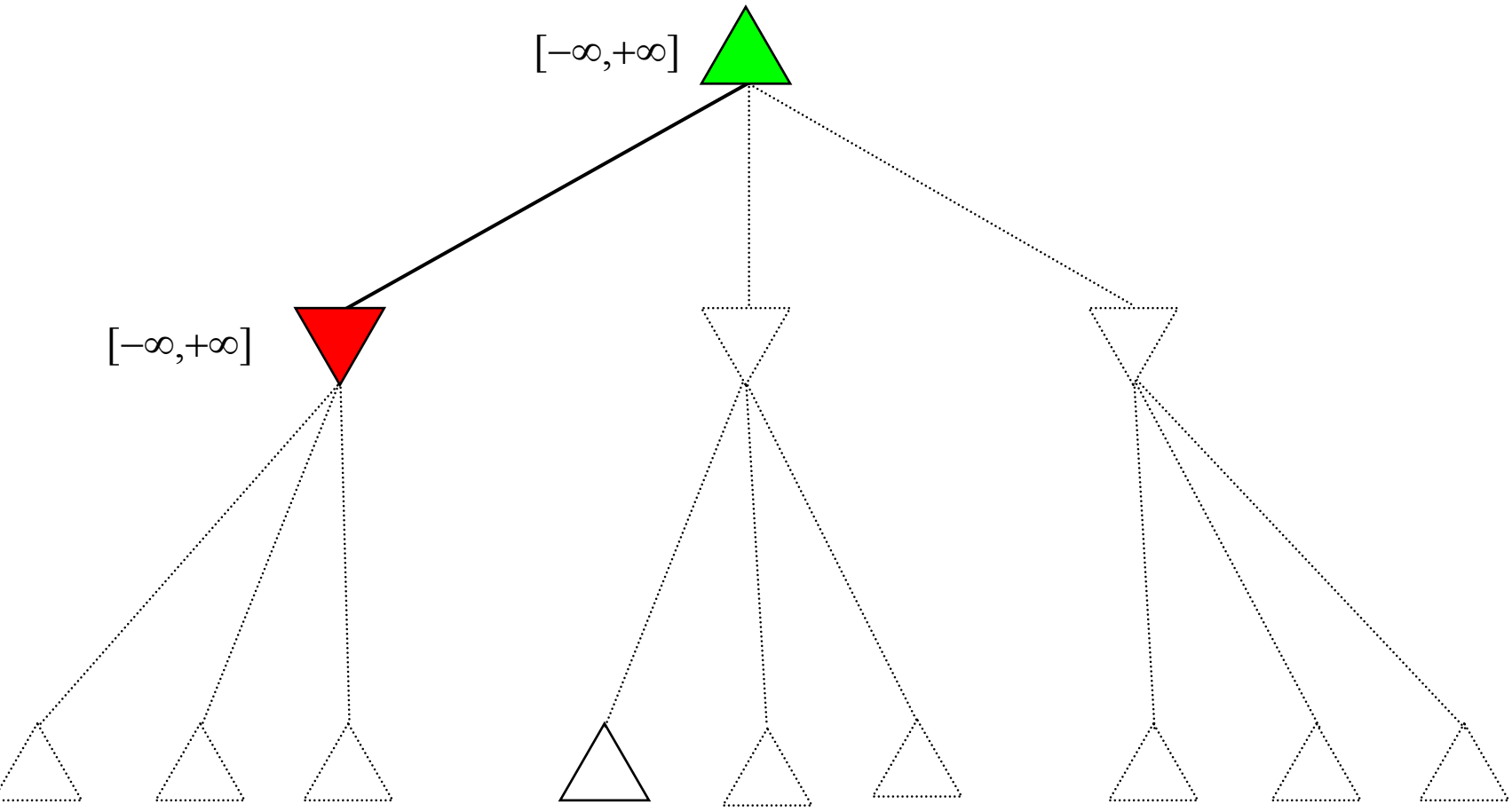
Poda α-β

- poda α-β devolve a mesma ação que MINIMAX, mas poda ramificações que não influenciam na escolha dessa ação
- poda α-β pode reduzir a complexidade da busca, dependendo da ordem em que os nós são examinados (no xadrez, pode reduzir *b* de 35 para 6 [capturar<ameaçar<avançar<retroceder])

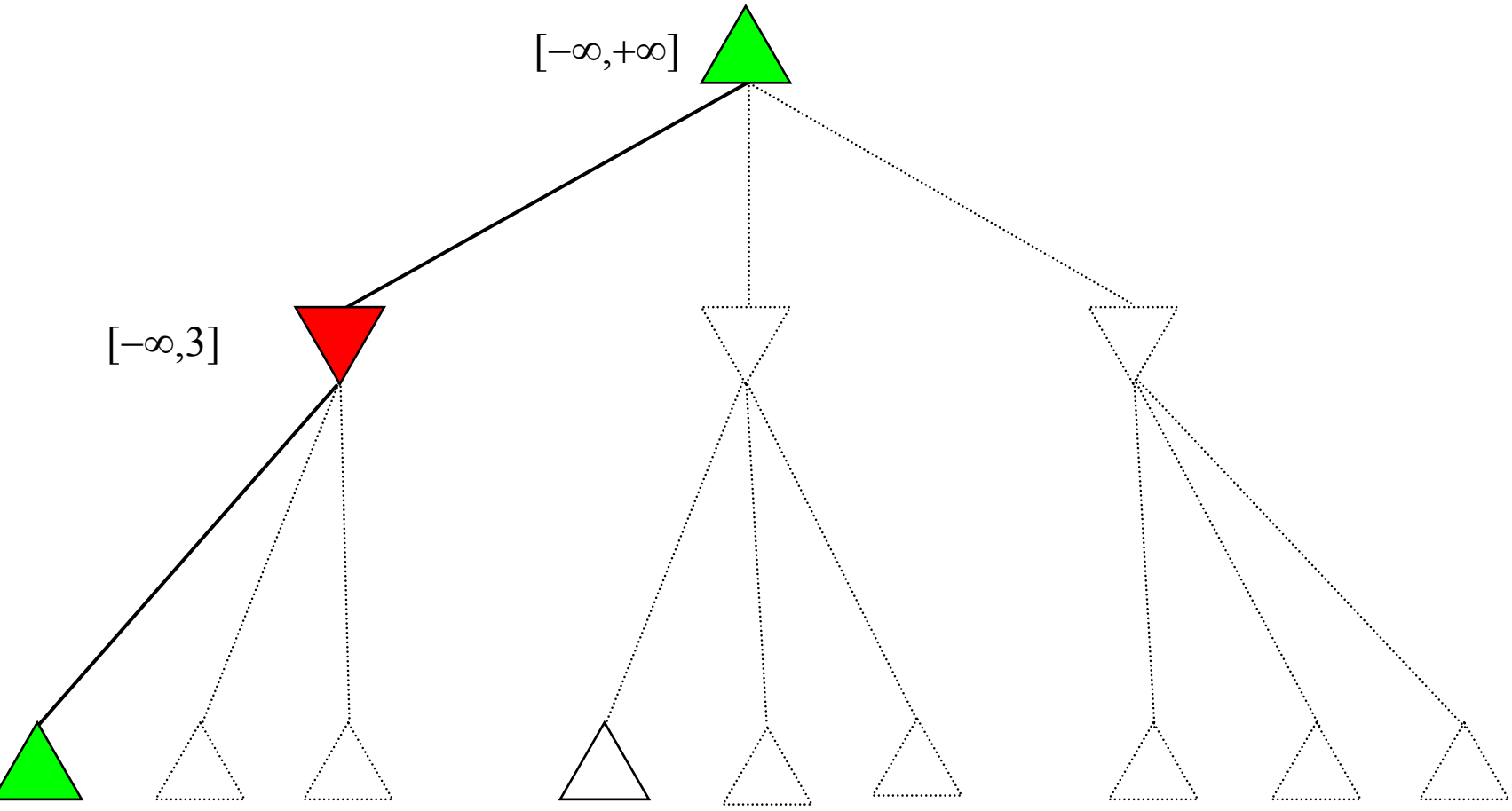
Exemplo de poda α-β



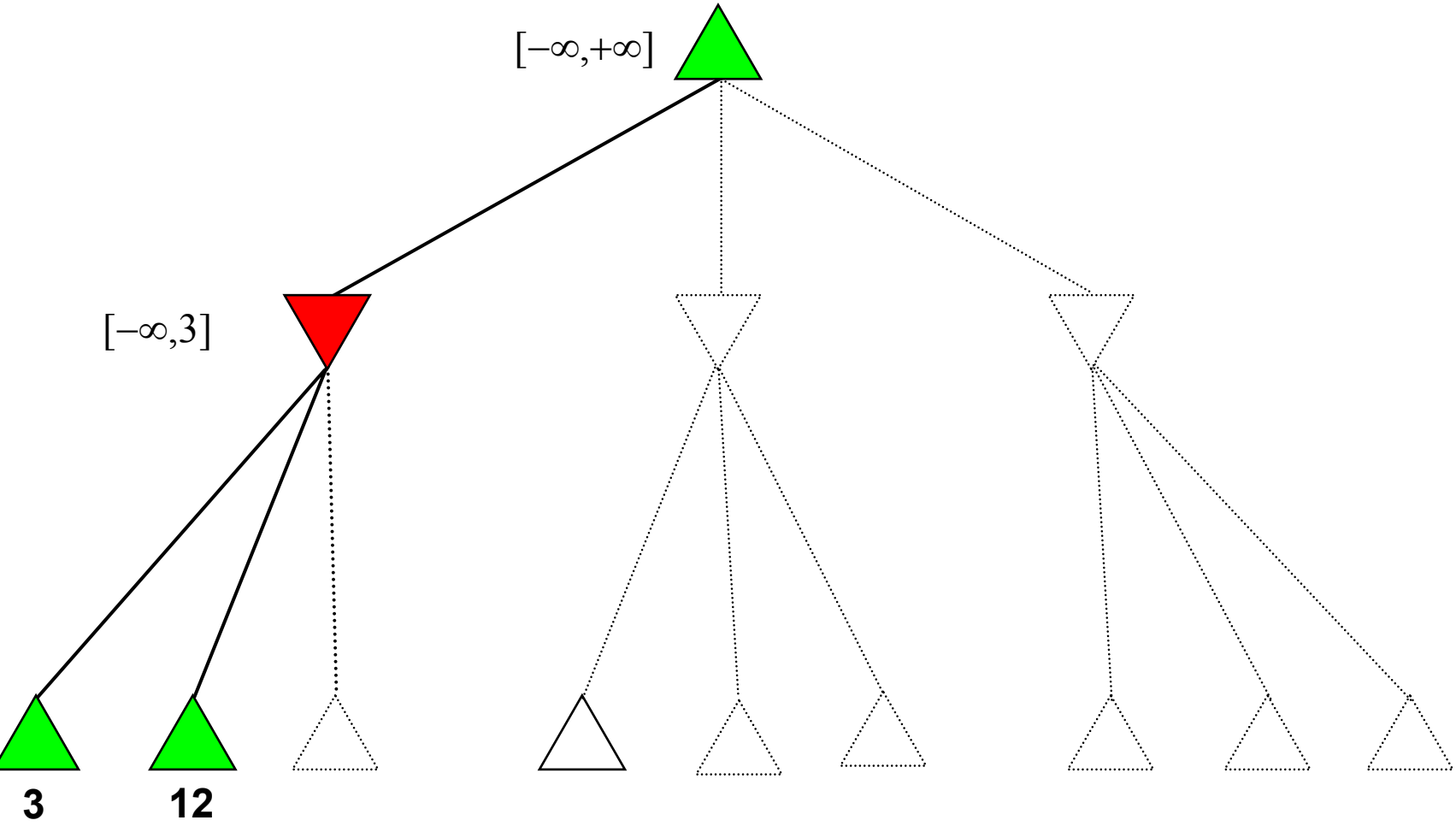
Exemplo de poda α-β



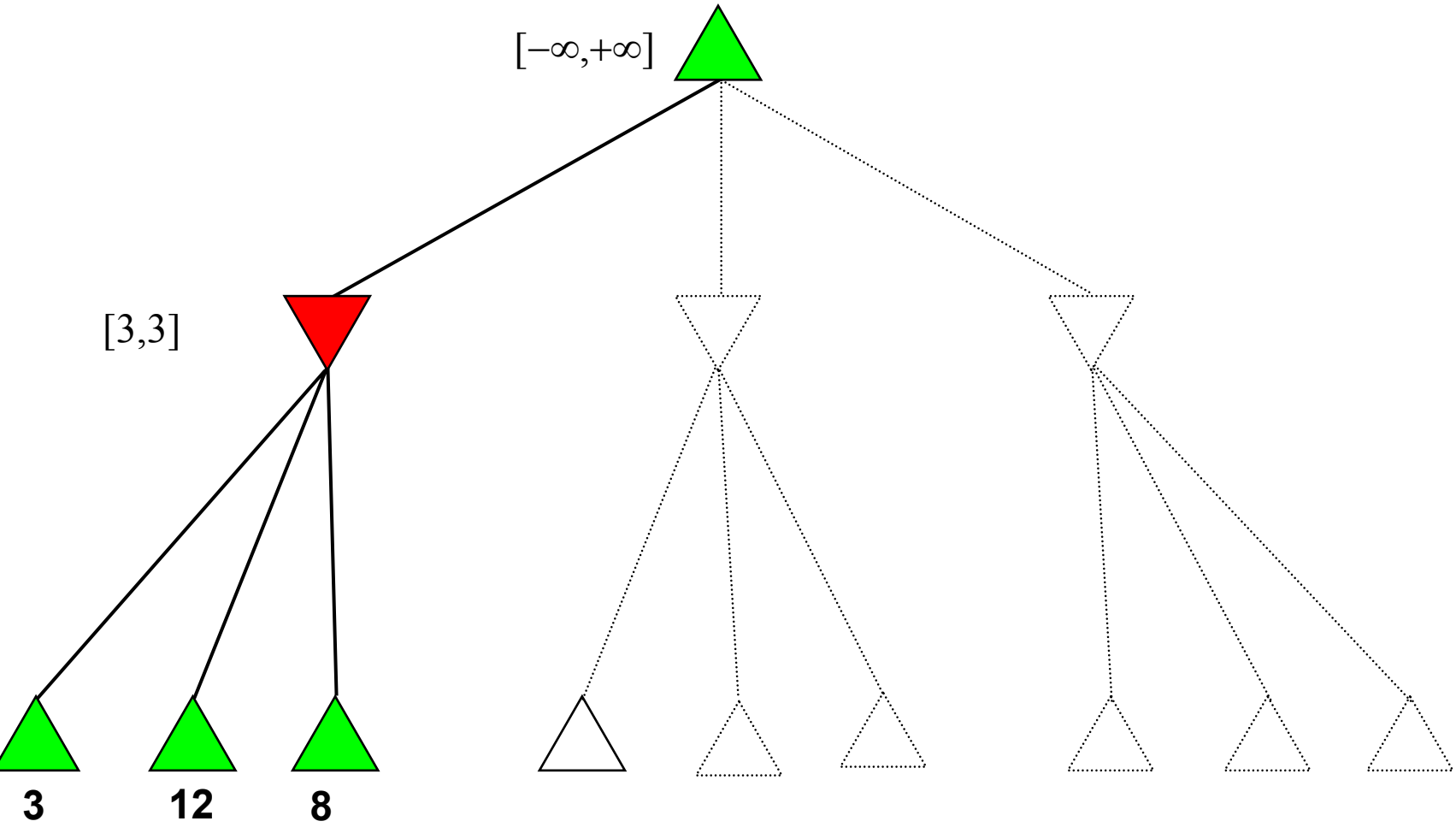
Exemplo de poda α-β



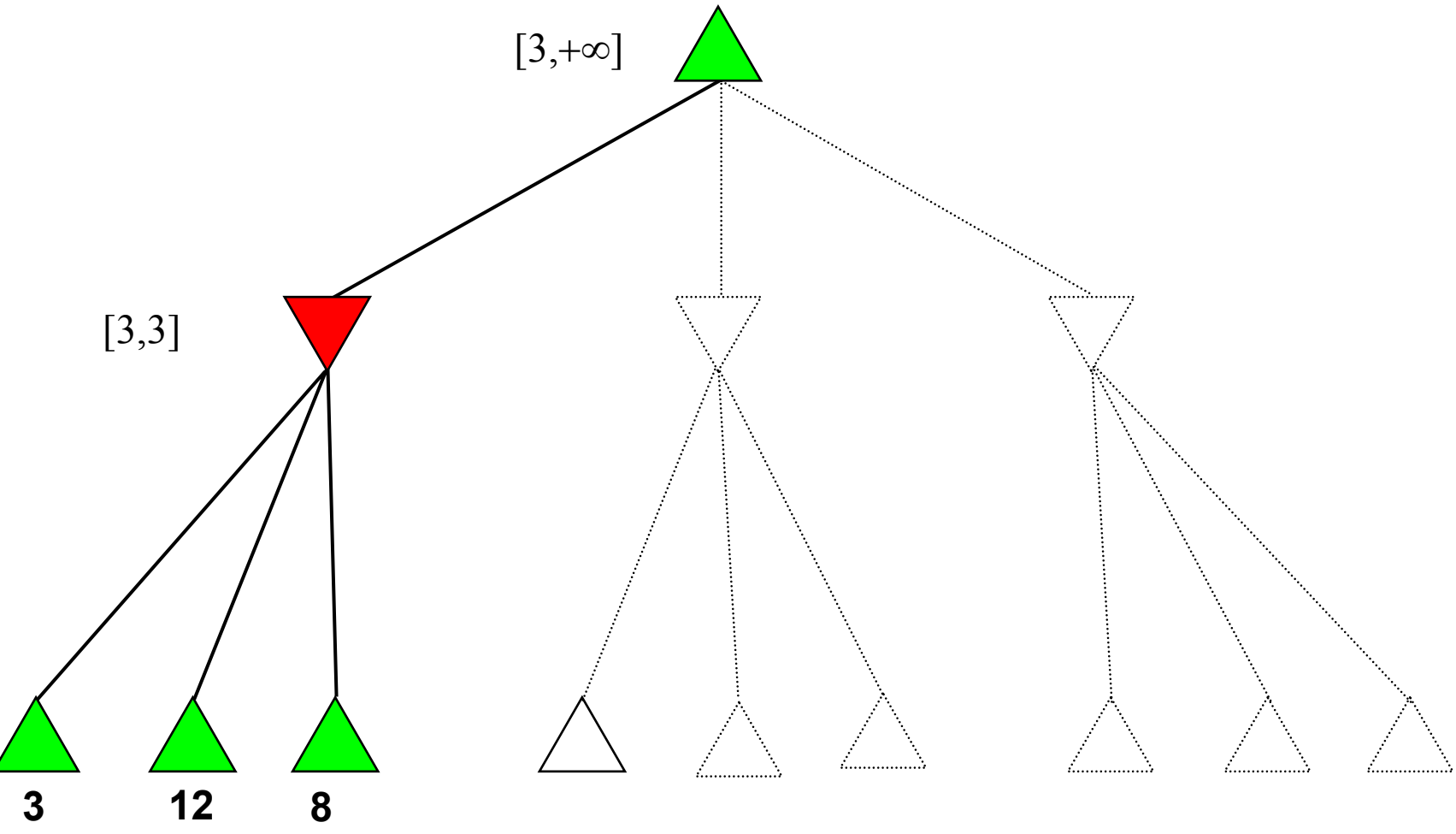
Exemplo de poda α-β



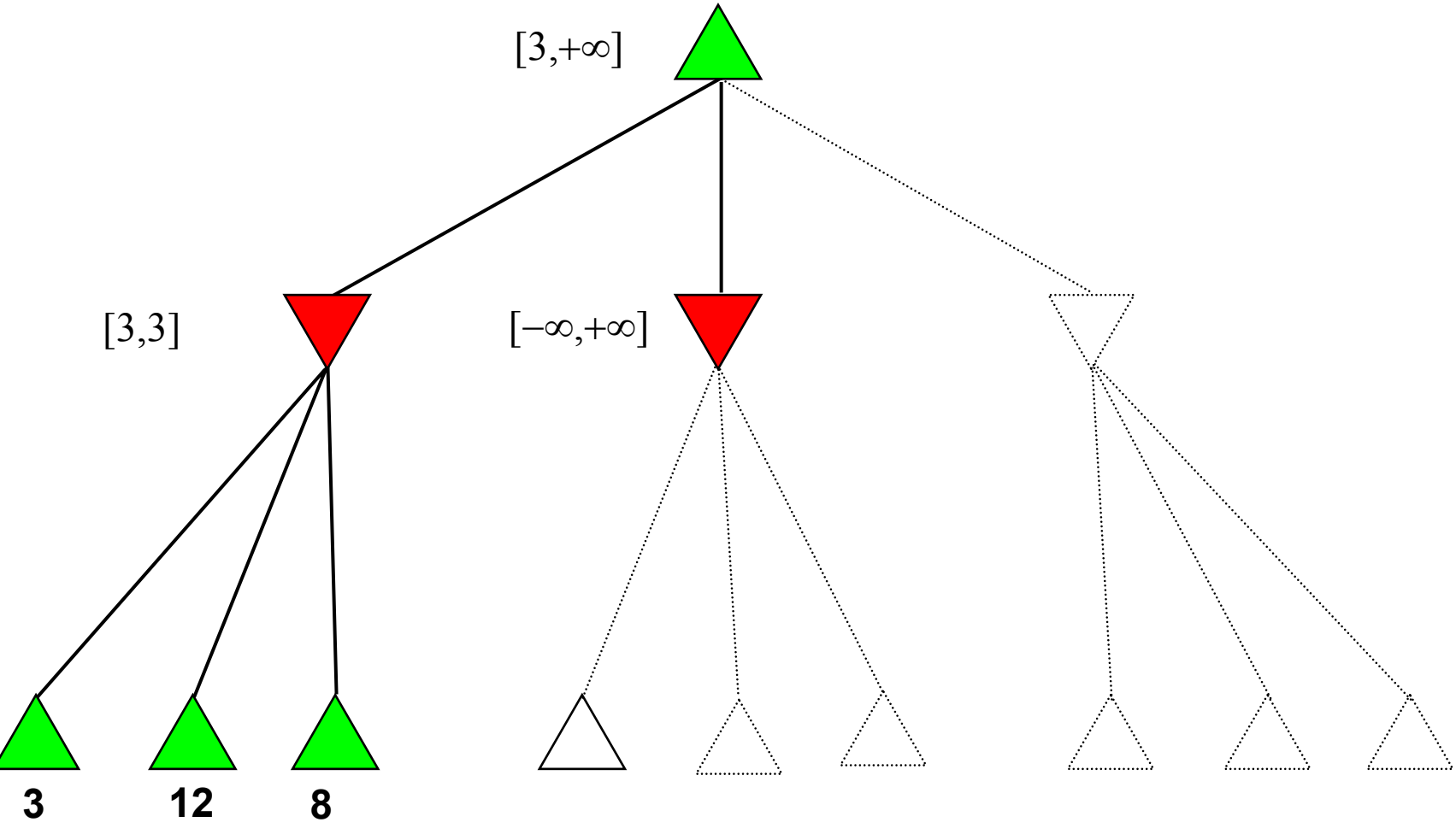
Exemplo de poda α-β



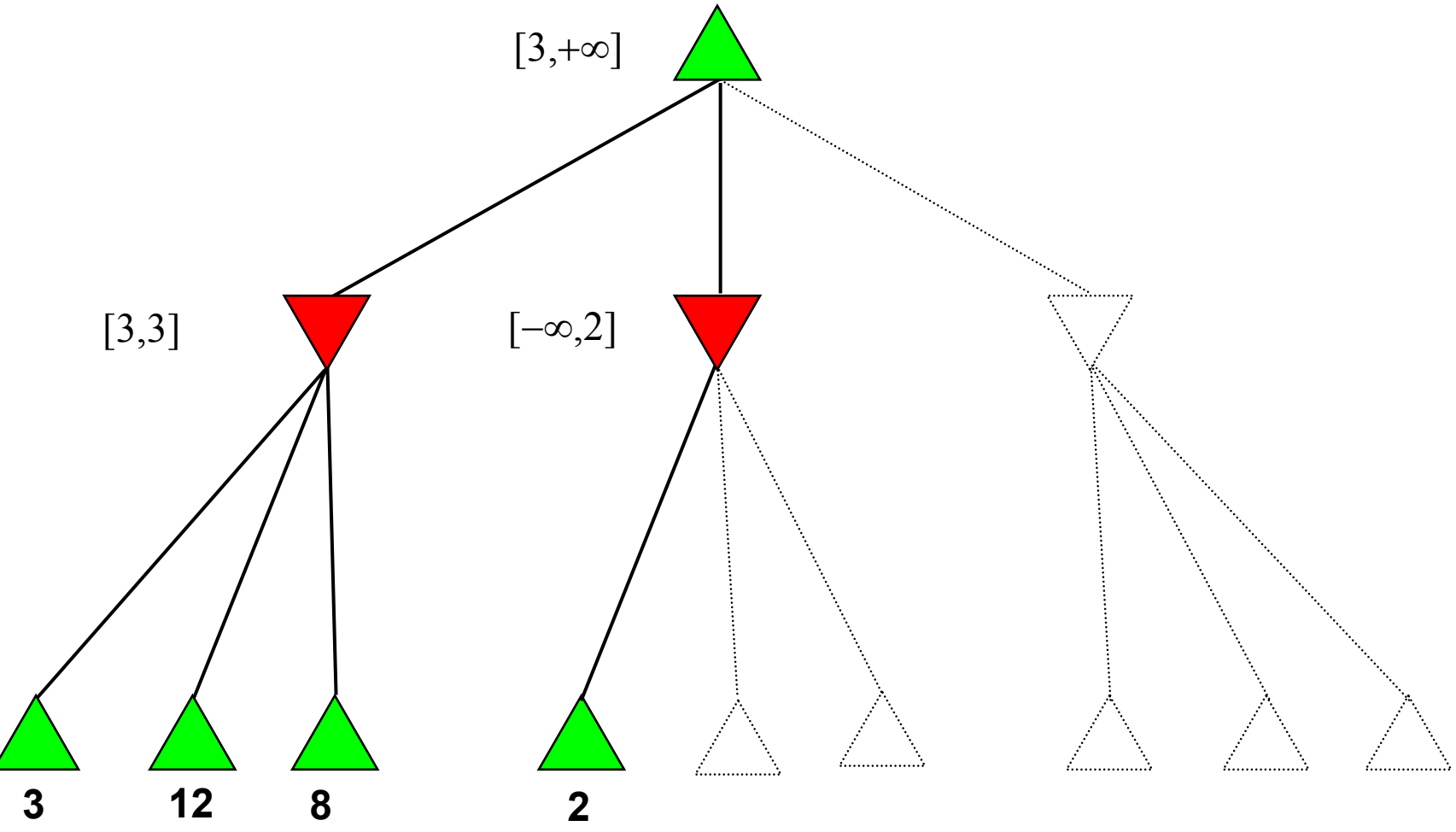
Exemplo de poda α-β



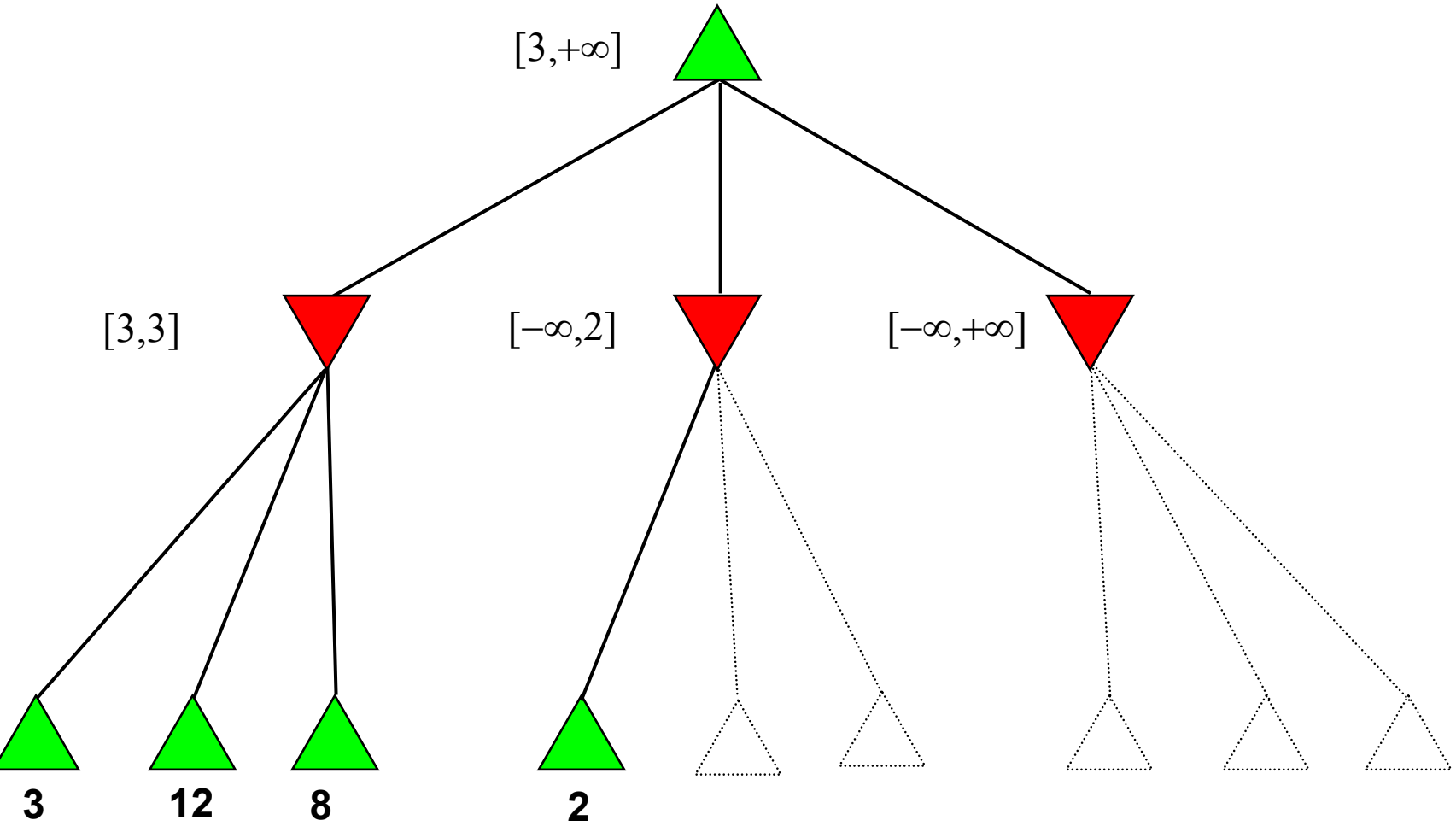
Exemplo de poda α-β



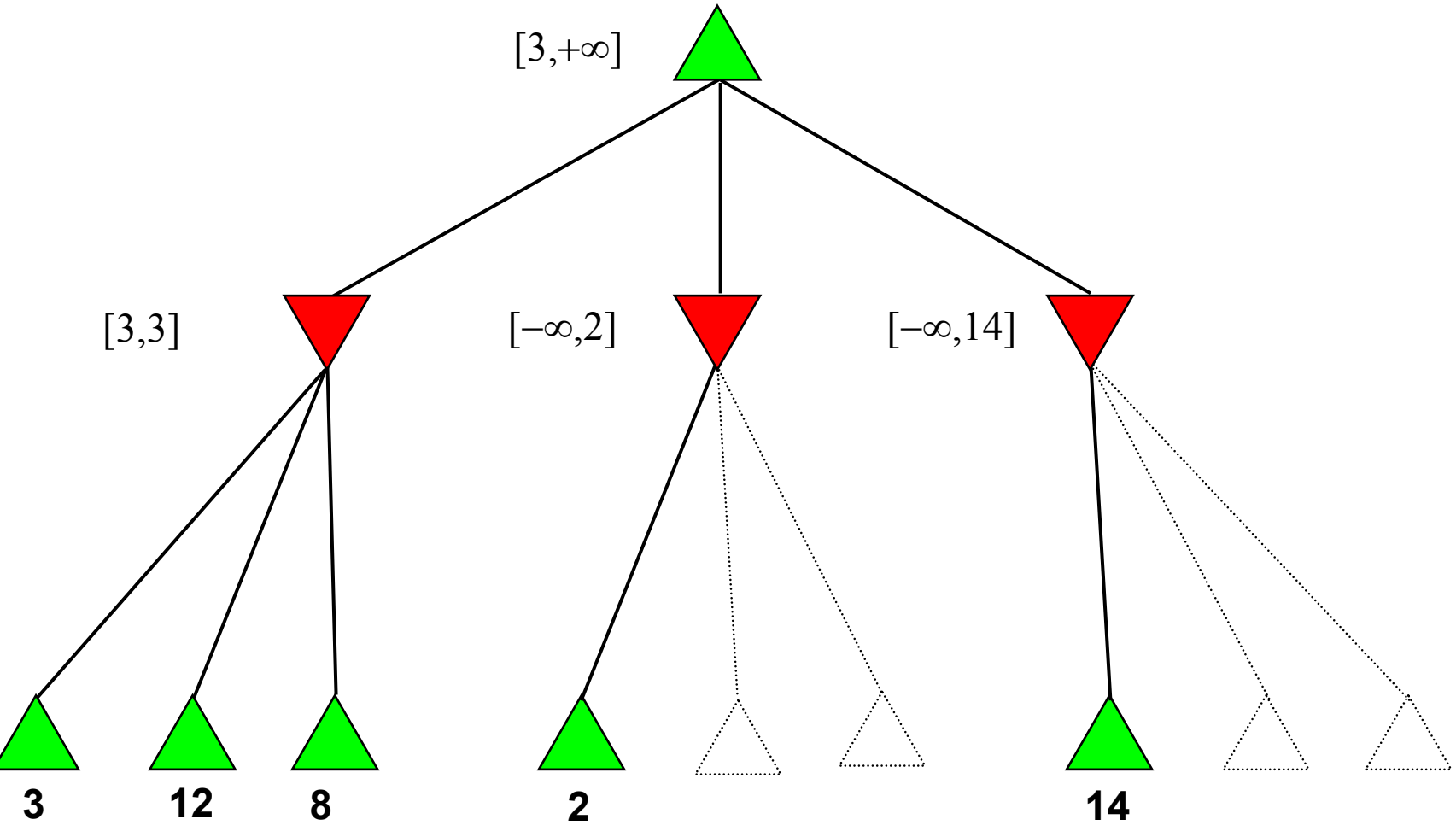
Exemplo de poda α-β



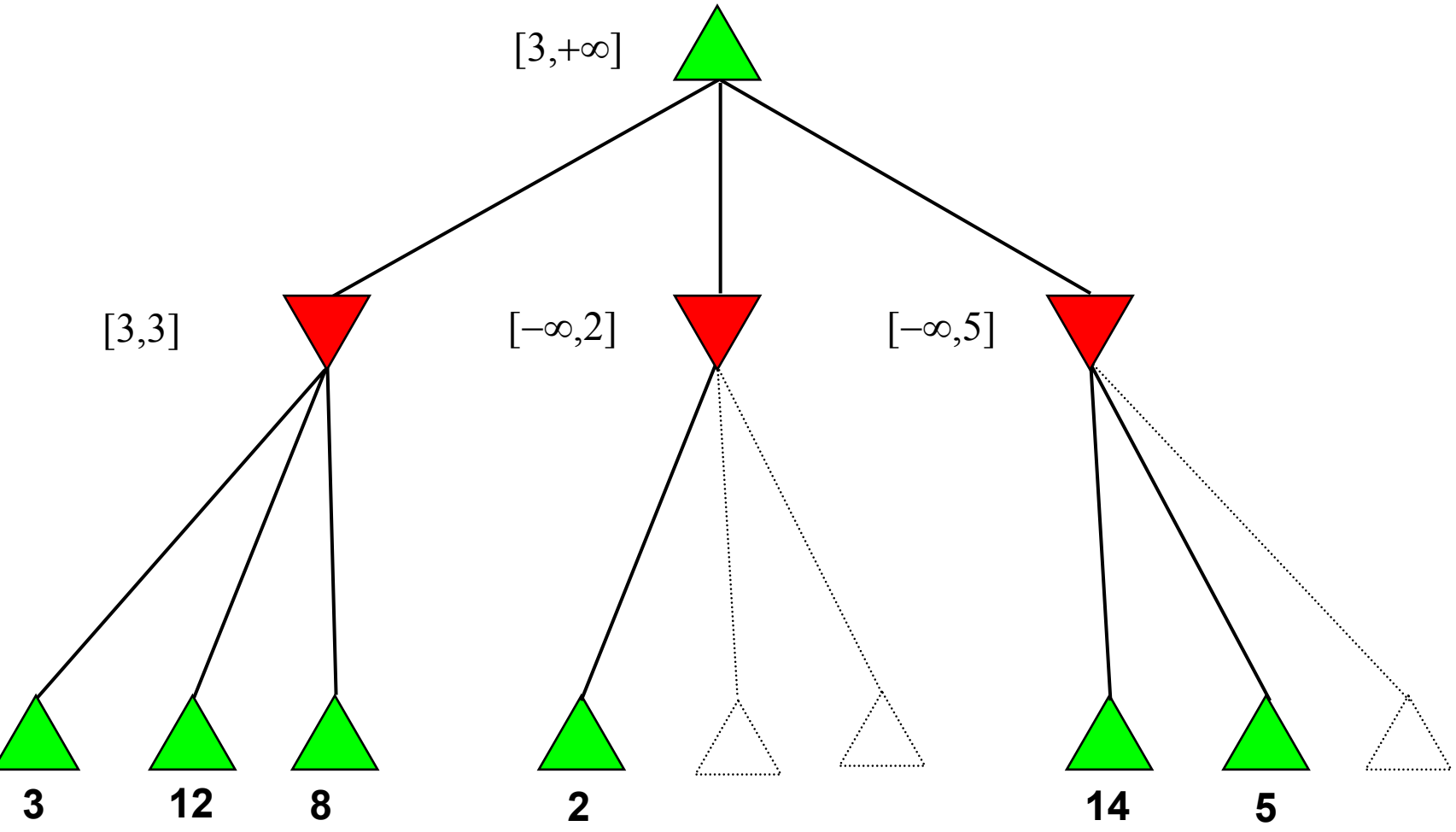
Exemplo de poda α-β



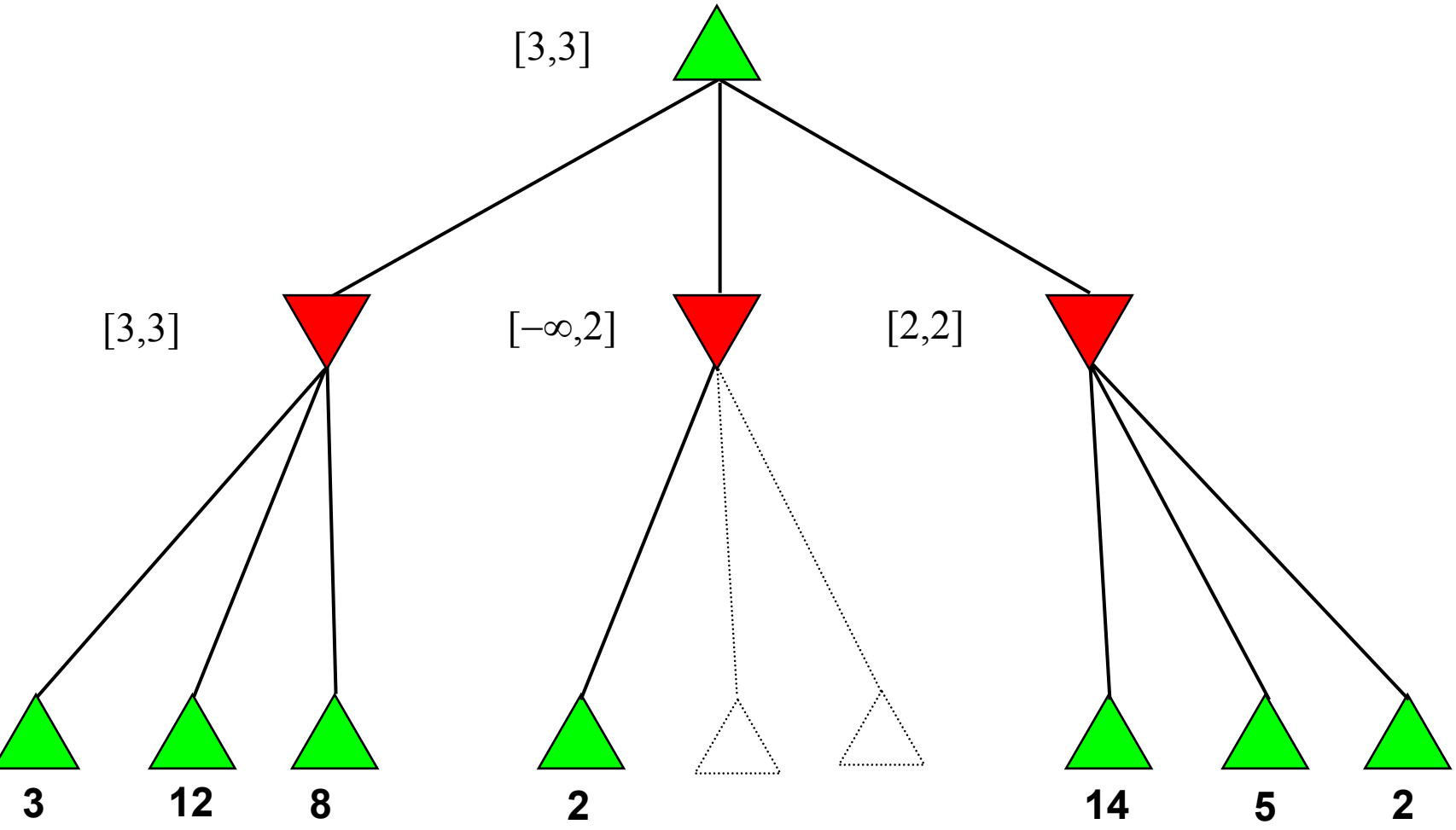
Exemplo de poda α-β



Exemplo de poda α-β



Exemplo de poda α-β



Poda α-β como simplificação

VALOR-MINIMAX(*raiz*)

$= max(min(3,12,8), min(2,x,y), min(14,5,2))$

$= max(3, min(2,x,y), 2)$

$= max(3, z, 2)$, onde $z \leq 2$

$= 3$

Algoritmo de poda α-β

```
function ALPHA-BETA-SEARCH(state) returns an action
    inputs: state, current state in game
    v ← MAX-VALUE(state, −∞, +∞)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s, α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
```

Algoritmo de poda α-β

```
function MIN-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← +∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s, α, β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
```

Problemas com MINIMAX e poda α-β

- MINIMAX gera o espaço de busca do jogo inteiro
- poda α-β permite eliminar parte desse espaço, mas ainda precisa fazer a busca até os estados terminais em pelo menos um dos ramos
- em geral, essa profundidade é proibitiva (há limite de tempo para cada jogada)
- Shannon propôs tratar nós não-terminais como se fossem folhas:
  - função de utilidade ⇒ função de avaliação (heurística)
  - teste de terminal ⇒ teste de corte

Função de avaliação

- fornece uma estimativa da utilidade de um estado *s* (semelhante a uma função heurística)
- o desempenho do programa depende da qualidade da função de avaliação
- uma função avaliação mal projetada poderá guiar o agente em direção à derrota

Função linear ponderada

- combina características do estado que podem ser julgadas de maneira independente:

$$\text{AVALIAÇÃO}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- Exemplo: *valor material* de cada peça no xadrez:

  peão = 1
  cavalo e bispo = 3
  torre = 5
  rainha = 9

$$\text{AVALIAÇÃO}(s) = p(s) + 3c(s) + 3b(s) + 5t(s) + 9r(s)$$

Uma boa função de avaliação deve:

- ordenar os estados terminais da mesma forma que a função de utilidade
- ser de fácil computação (precisão *vs*. custo)
- refletir precisamente a *chance* de vitória que o estado oferece (a avaliação não é exata!)

Busca com corte

- usa função de avaliação quando for apropriado cortar a busca num determinado estado:

  se TESTE-TERMINAL(*s*) **então devolva** UTILIDADE(*s*)

  deve ser substituído por:

  se TESTE-CORTE(*s*) **então devolva** AVALIAÇÃO(*s*)

- usa *profundidade limitada* ou *profundidade iterativa* como critério de corte

Escolha do ponto de corte

- dependendo do jogo, um estado favorável pode tornar-se extremamente desfavorável após um único movimento
  - a função de avaliação deve ser aplicada apenas a estados *quiescentes* (estados em que é improvável que ocorram mudanças num futuro próximo)
  - busca de quiescência (restringe-se às ações que afetam diretamente a função de avaliação)

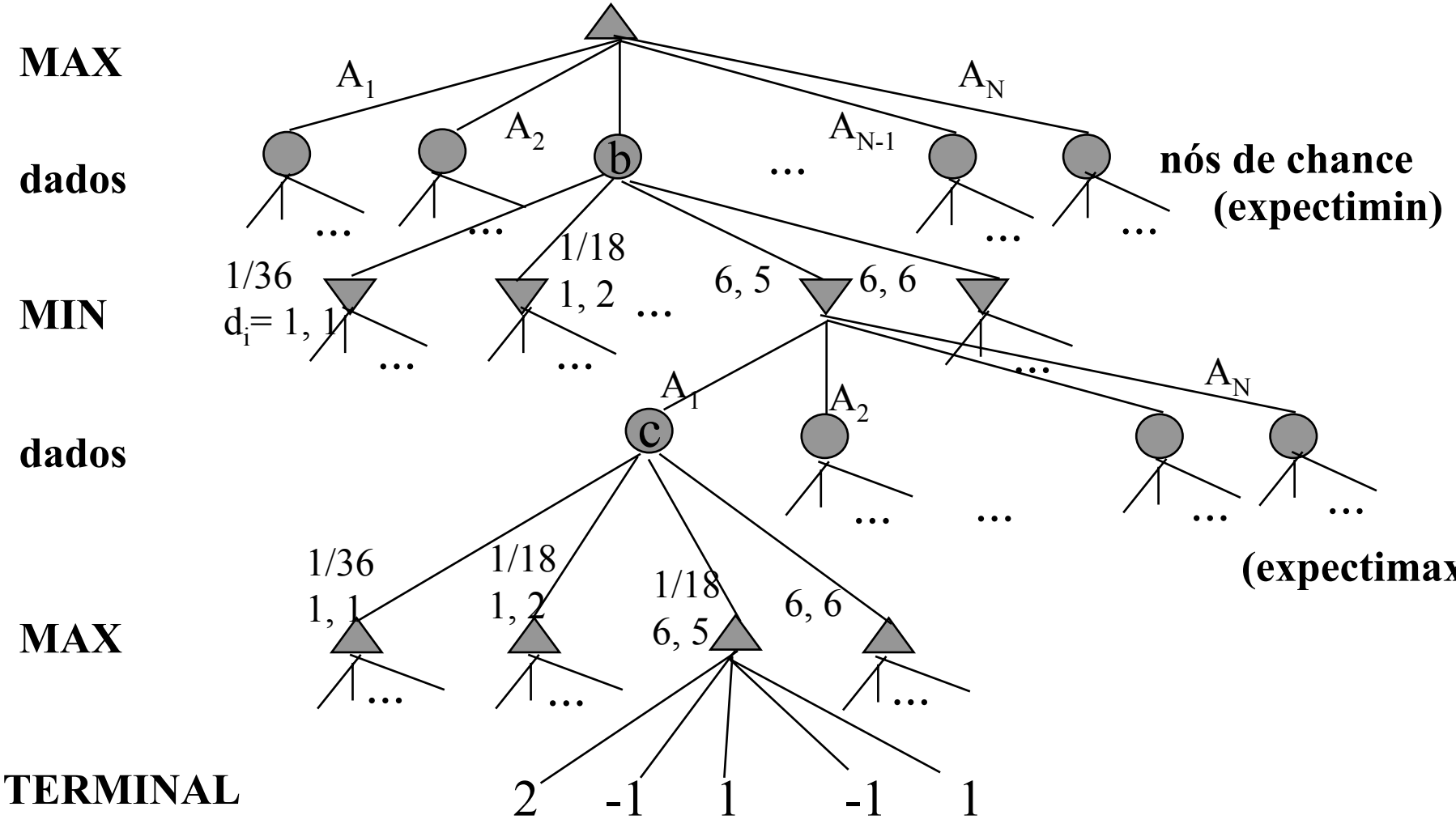
Jogos de sorte

- com a ocorrência de eventos imprevisíveis (= lançamento de dados), não há como construir uma árvore de busca padrão
- a árvore de busca deve incluir *nós de acaso*, além dos nós MAX e MIN
- ramificações nos nós de acaso denotam lançamentos possíveis e respectivas probabilidades

Exemplo: jogo de gamão

- o lançamento de dois dados antes de cada jogada determina os movimentos possíveis para cada jogador
  - existem 36 combinações dos dois dados, todas igualmente prováveis
  - apenas 21 combinações distintas (6|5=5|6,…)
  - probabilidade para os 6 duplos (1|1,…,6|6) = 1/36
  - probabilidade para os outros 15 = 1/18
- *podemos apenas calcular um valor esperado!*

Árvore de busca com nós de acaso



EXPECTMINIMAX

VALOR-EXPECTMINIMAX(r) =

UTILIDADE(r)	se r é um nó terminal
$\max_{s \in \textit{sucessores}(r)}$ VALOR-EXPECTMINIMAX(s)	se r é um nó max
$\min_{s \in \textit{sucessores}(r)}$ VALOR-EXPECTMINIMAX(s)	se r é um nó min
$\Sigma_{s \in \textit{sucessores}(r)}\, p(s)$.VALOR-EXPECTMINIMAX($s$)	se r é um nó de acaso

Complexidade de tempo: $O(b^m n^m)$

- b é o fator de ramificação
- m é a profundidade
- n é o número de lançamentos distintos

Fim