Algoritmo minimax ricorsivo

```
function MINIMAX-DECISION(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) 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
```

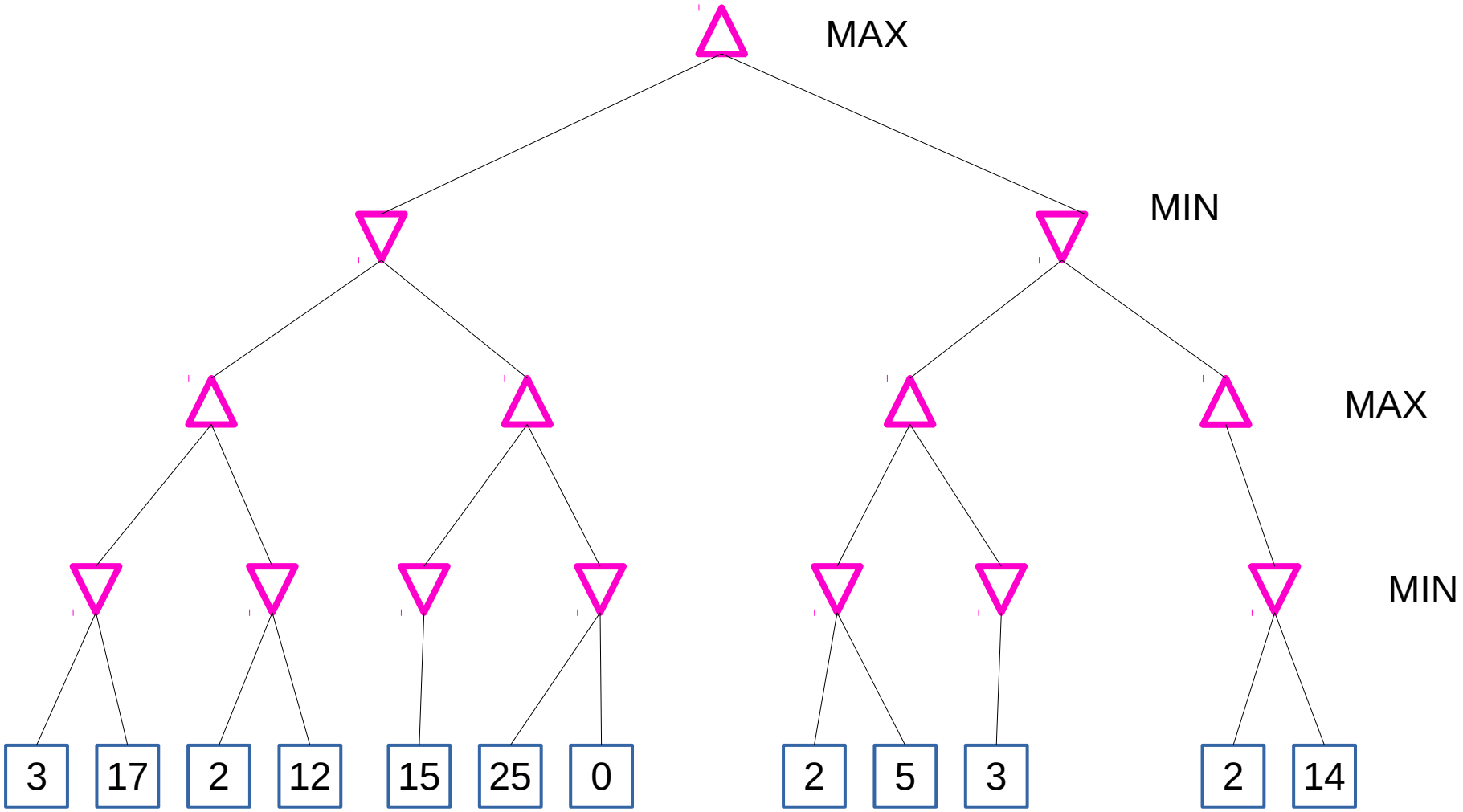
Figure 6.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state.

Visita completa in profondità dell'albero di gioco

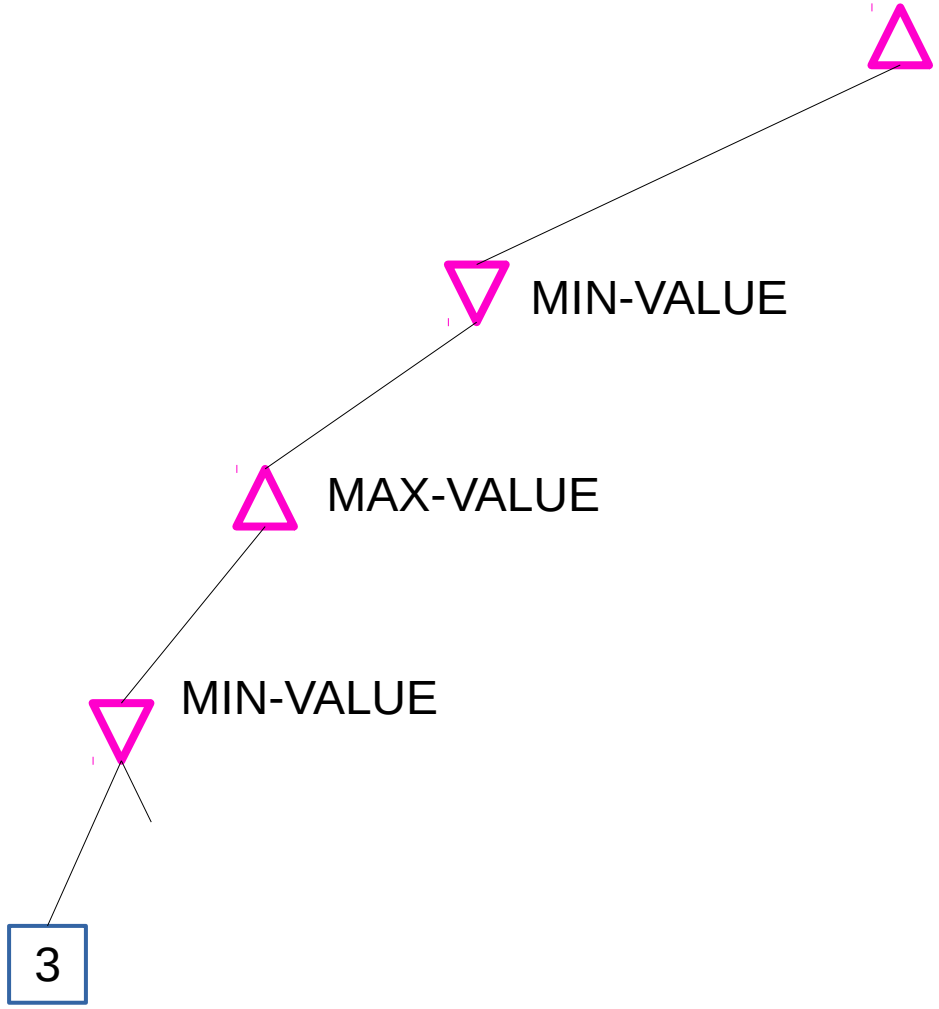
Complessità temporale: **$O(b^m)$**
con b = branching factor ed
m = profondità massima

Complessità spaziale: **O(bm)**
se i successori sono generati contemporaneamente

Esempio: albero di gioco completo



Esempio: albero di gioco completo

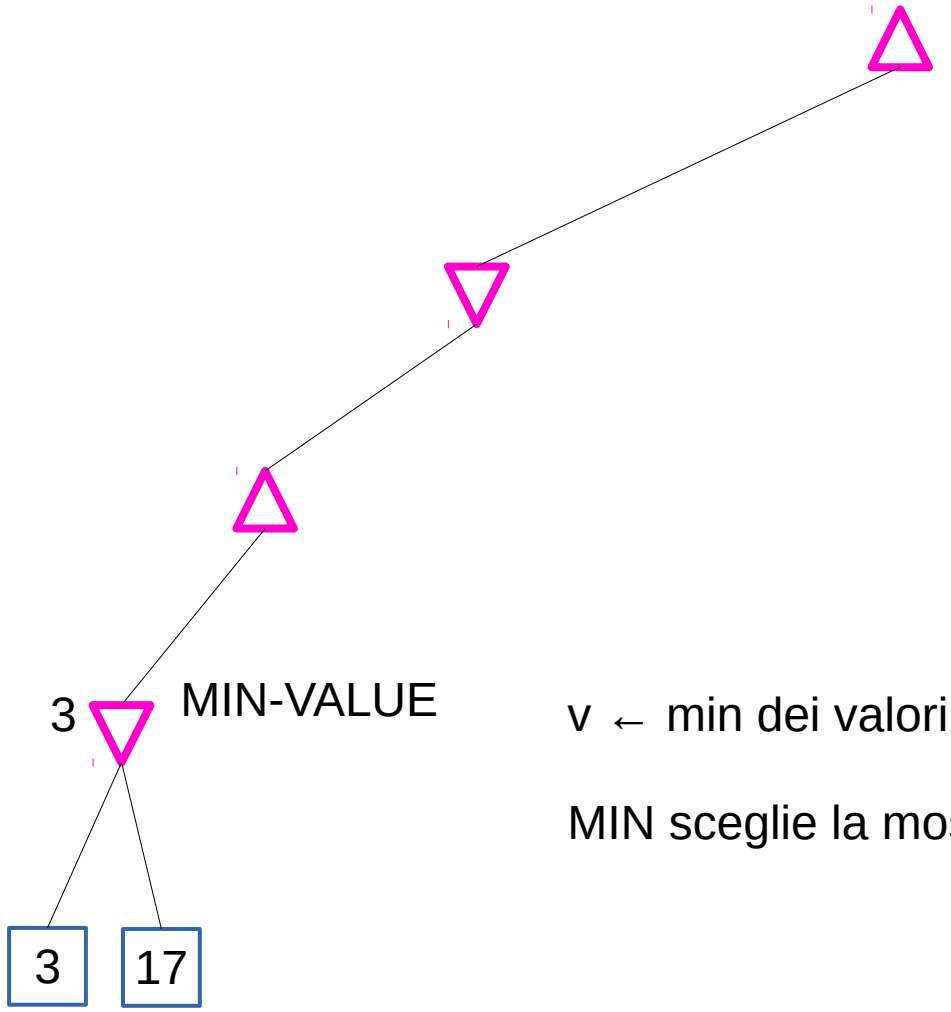


Esplorazione ricorsiva dell'intero albero con chiamate alternate a max-value e a min-value.

I nodi assumeranno un valore alla chiusura delle rispettive chiamate ricorsive.

I nodi terminali hanno come valore la loro utilità.

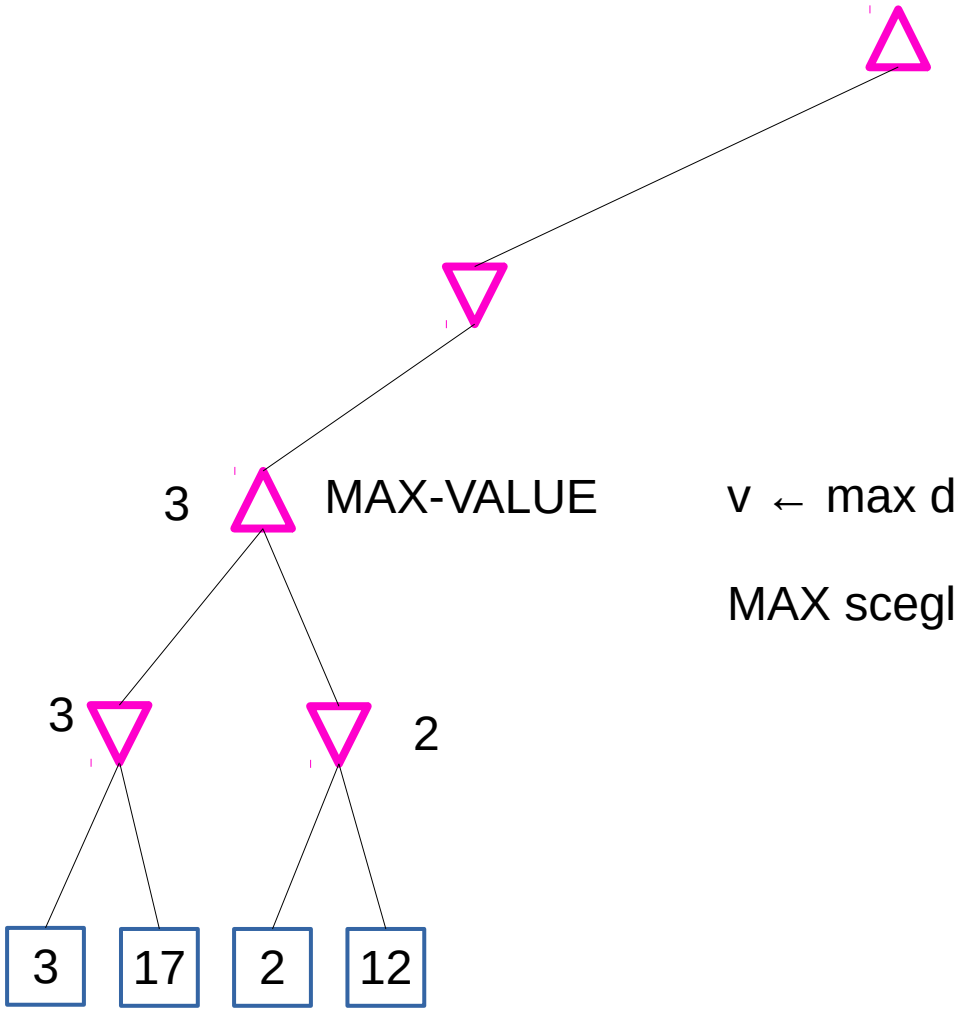
Esempio: albero di gioco completo



v ← min dei valori calcolati per i successori

MIN sceglie la mossa meno vantaggiosa per MAX

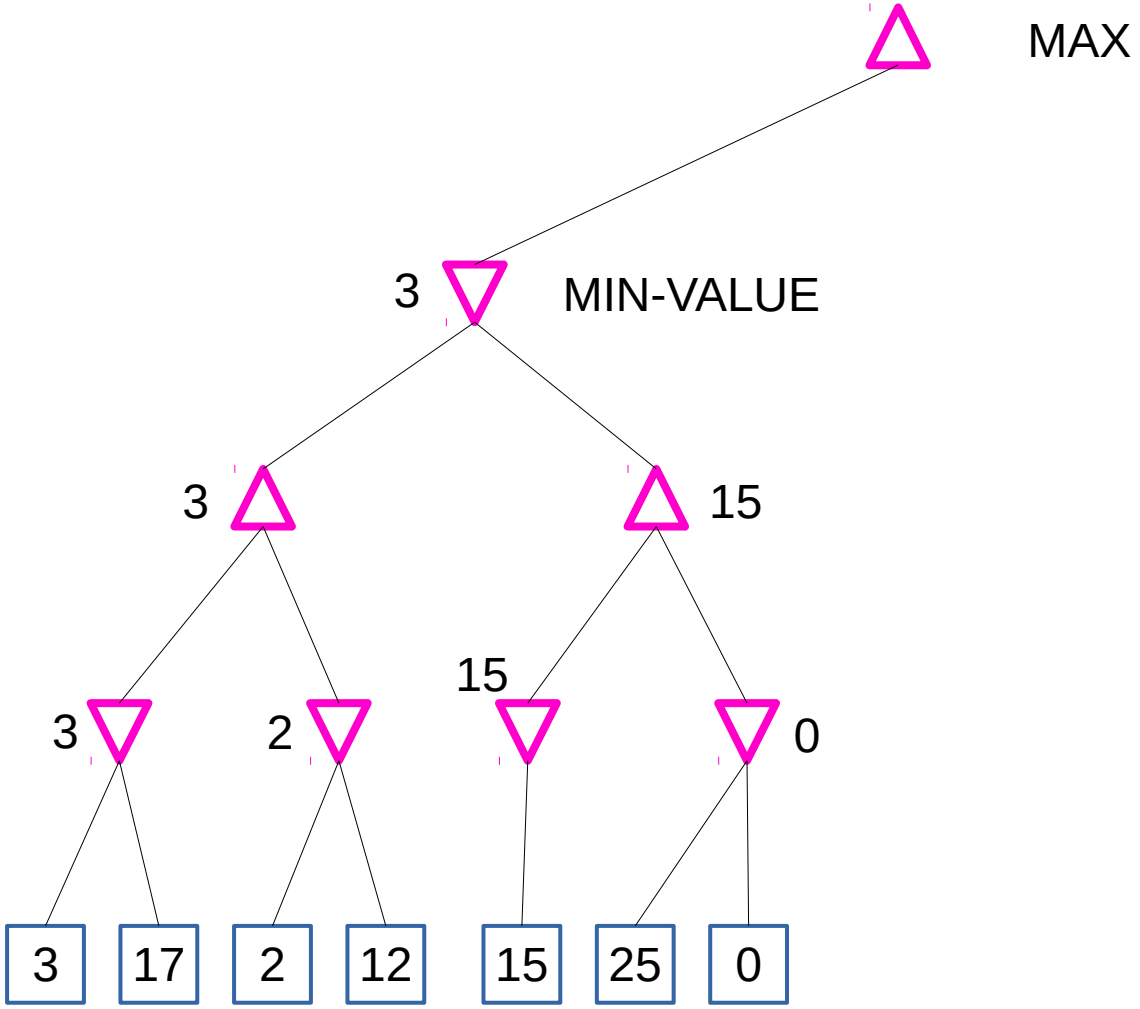
Esempio: albero di gioco completo



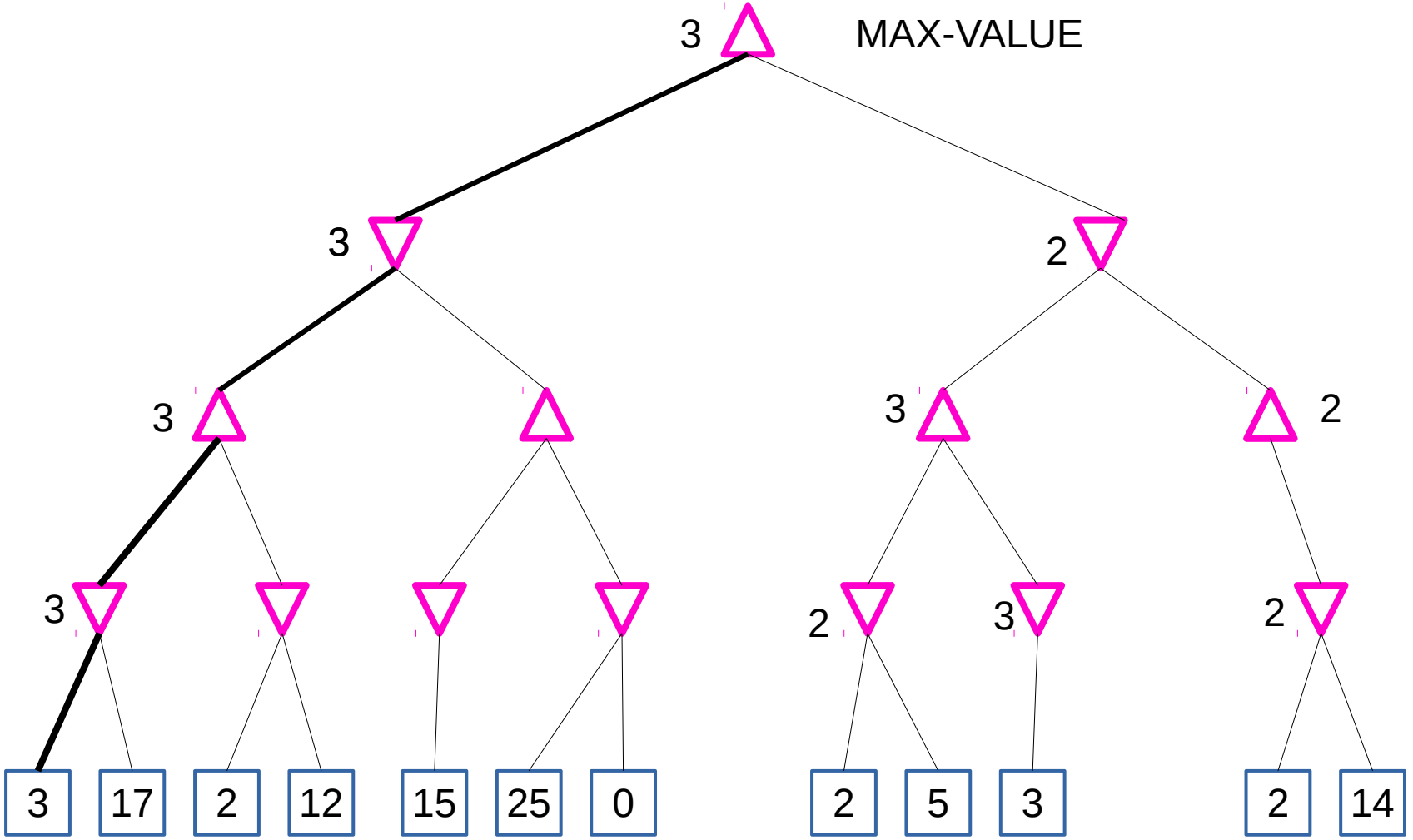
v ← max dei valori calcolati per i successori

MAX sceglie la mossa più vantaggiosa per sè

Esempio: albero di gioco completo



Esempio: albero di gioco completo



Giochi con più di due giocatori

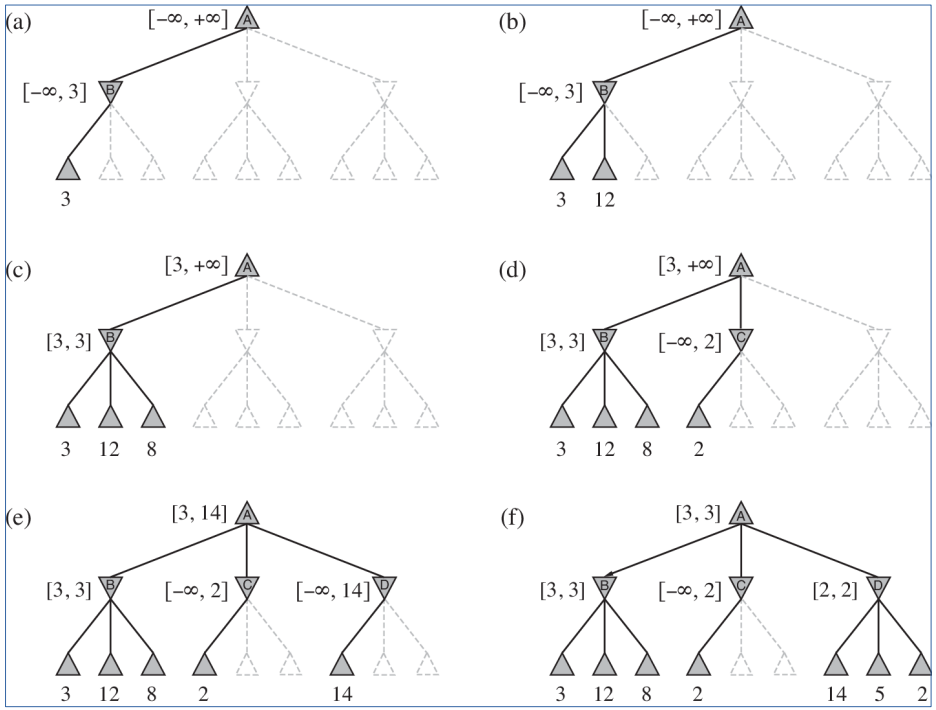
- Minimax può essere esteso a più giocatori
- Utilità di ogni nodo data da un vettore $<u_1, \ldots, u_k>$ dove u_i indica l'utilità del nodo per il giocatore i
- Un aspetto dei giochi a più di due giocatori è la formazione di alleanze

Valutazione

- Effettua una visita in profondità completa quindi la complessità temporale è esponenziale e quella spaziale è lineare
- È completo in grafi finiti
- È ottimale se MAX e MIN giocano in modo ottimale

Minimax con potatura Alfa-Beta

- In termini di tempo, minimax è molto costoso
- Per ridurre i tempi di ricerca: potare i rami meno promettenti senza seguirli
- Alfa-beta è un algoritmo di potatura



nell'esempio le foglie figlie del nodo centrale non espanse hanno valutazioni più alte di 2. È inutile considerarle perché darebbero un vantaggio all'avversario.

Potatura alfa-beta

- Ogni nodo ha associato un valore N che cambia man mano che l'esplorazione dei suoi sottoalberi procede
- L'esplorazione si porta dietro (e aggiorna) gli estremi di un intervallo [α , β]:
  - α = massimo lower bound delle soluzioni possibili (è posto dai nodi MAX)
    = valore della scelta migliore per MAX trovata in qualsiasi punto di scelta lungo il cammino (inizialmente – ∞)
  - β = minimo upper bound delle soluzioni possibili (è posto dai nodi MIN)
    = valore della scelta migliore per MIN trovata in qualsiasi punto di scelta lungo il cammino (inizialmente ∞)
- Ha senso esplorare un nodo se e solo se il suo valore stimato N è compreso fra α e β

http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html

Potatura alfa-beta

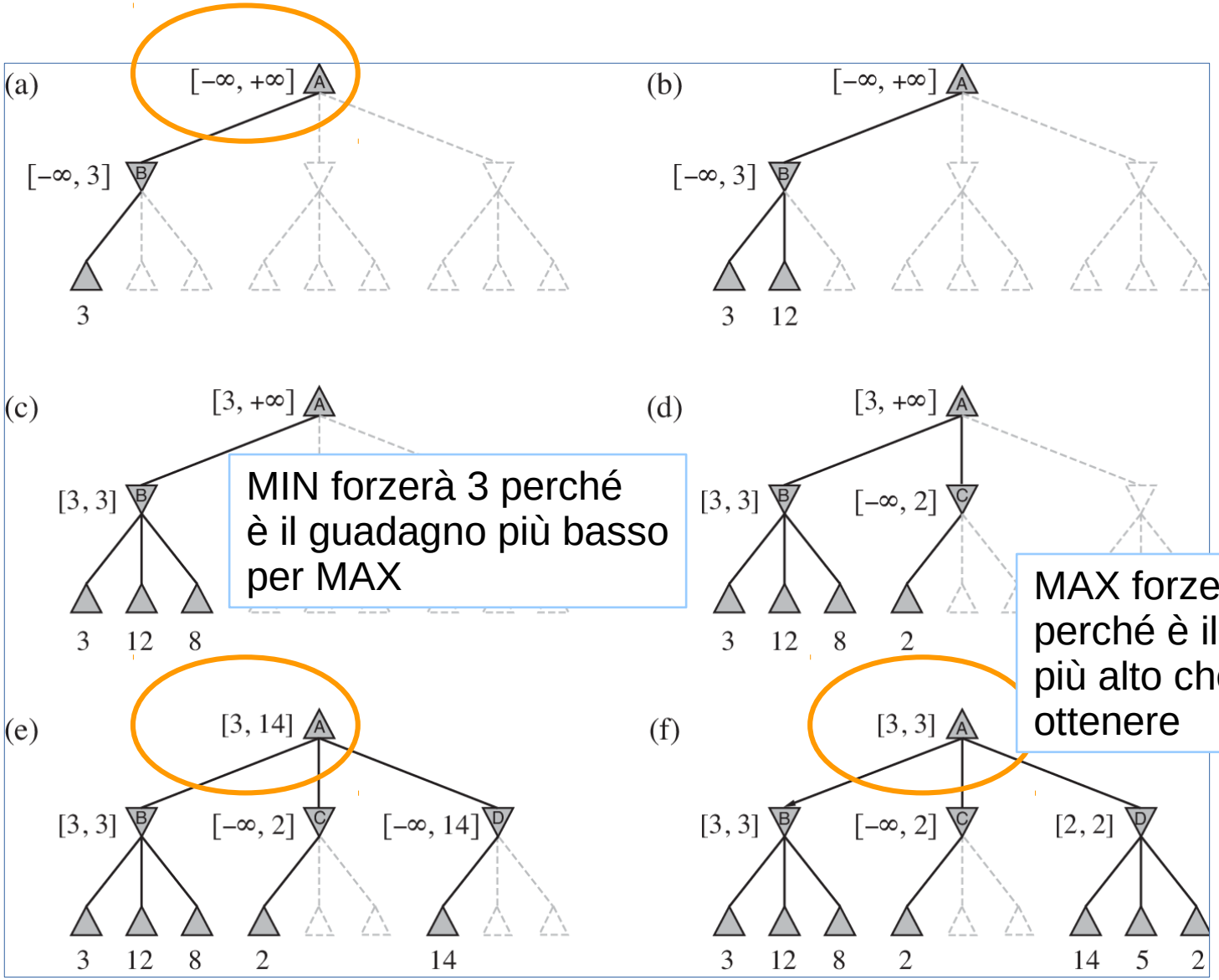
- I valori α e β sono aggiornati man mano che la ricerca procede
- Idealmente α e β sono gli estremi di un intervallo che si restringe con la ricerca:

- I nodi che hanno valutazioni fuori dall' intervallo sono esclusi
- Se a un certo punto per un certo nodo i due estremi si invertono, si può evitare di condurre l' esplorazione dei sottoalberi rimanenti di quel nodo

- Nessun valore potrà infatti essere minore di β e maggiore di α

http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html

Esempio


(a)
[−∞, +∞]
A
[−∞, 3]
B
3
(b)
[−∞, +∞]
A
[−∞, 3]
B
3
12
(c)
[3, +∞]
A
[3, 3]
B
3
12
8
MIN forzerà 3 perché è il guadagno più basso per MAX
(d)
[3, +∞]
A
[3, 3]
B
[−∞, 2]
C
3
12
8
2
MAX forzerà 3 perché è il guadagno più alto che può ottenere
(e)
[3, 14]
A
[3, 3]
B
[−∞, 2]
C
[−∞, 14]
D
3
12
8
2
14
(f)
[3, 3]
A
[3, 3]
B
[−∞, 2]
C
[2, 2]
D
3
12
8
2
14
5
2

Minimax con potatura alfa-beta: algoritmo

```
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
```

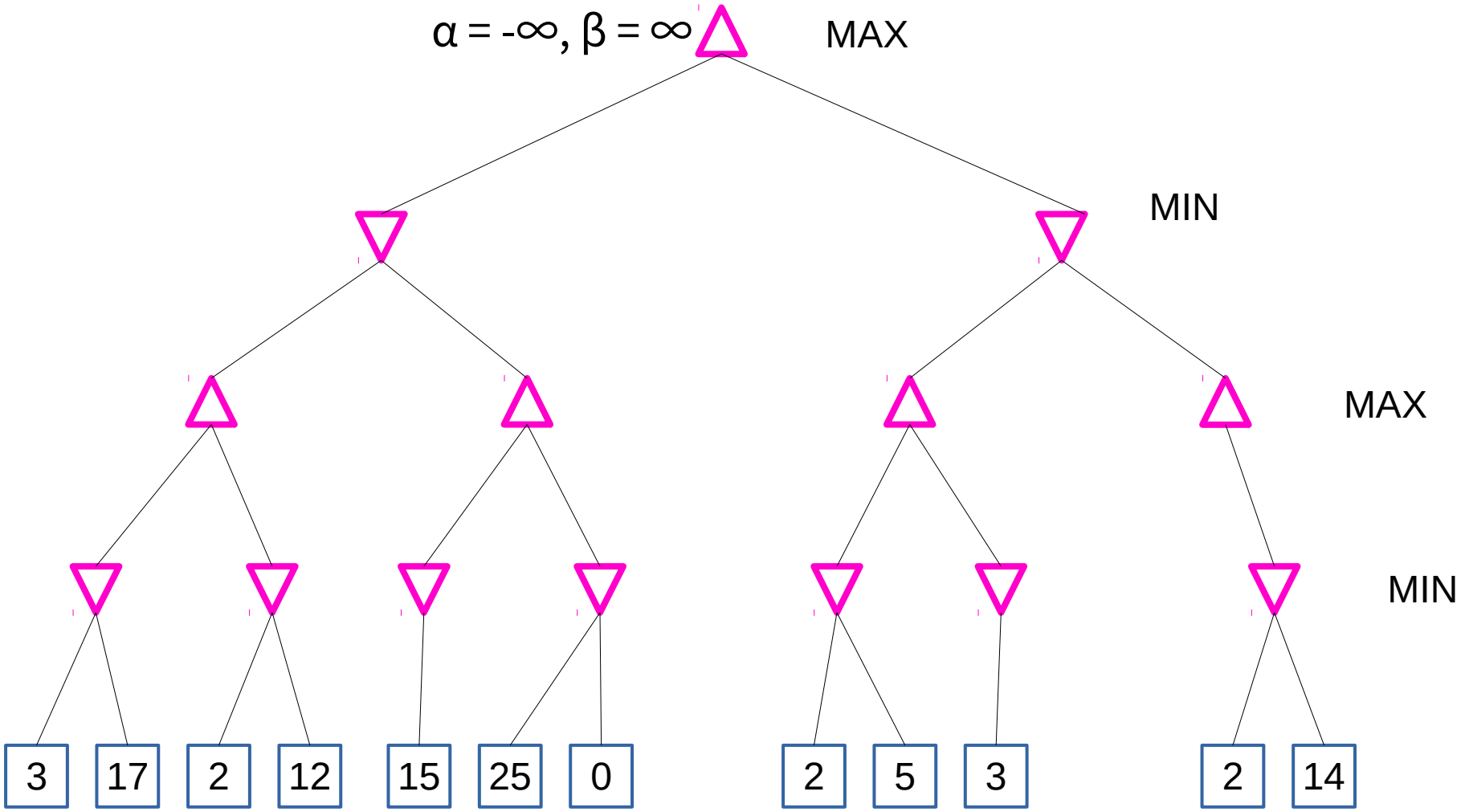
```
function MAX-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  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
```

MAX-VALUE modifica il lower bound

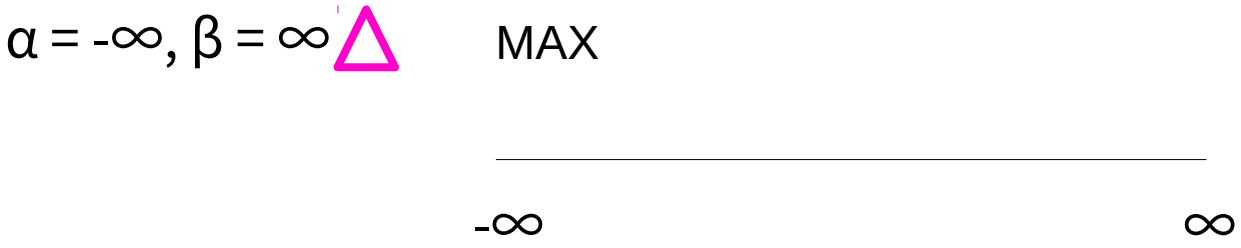
```
function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  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
```

MIN-VALUE modifica l'upper bound

Esempio: albero di gioco completo



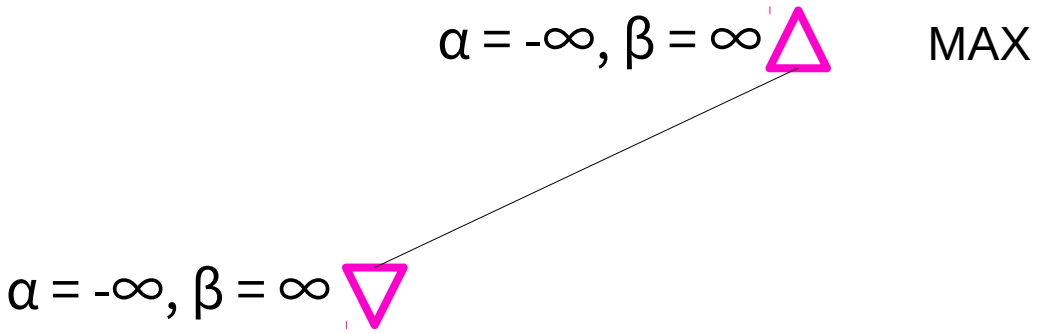
Esempio: parto dalla radice



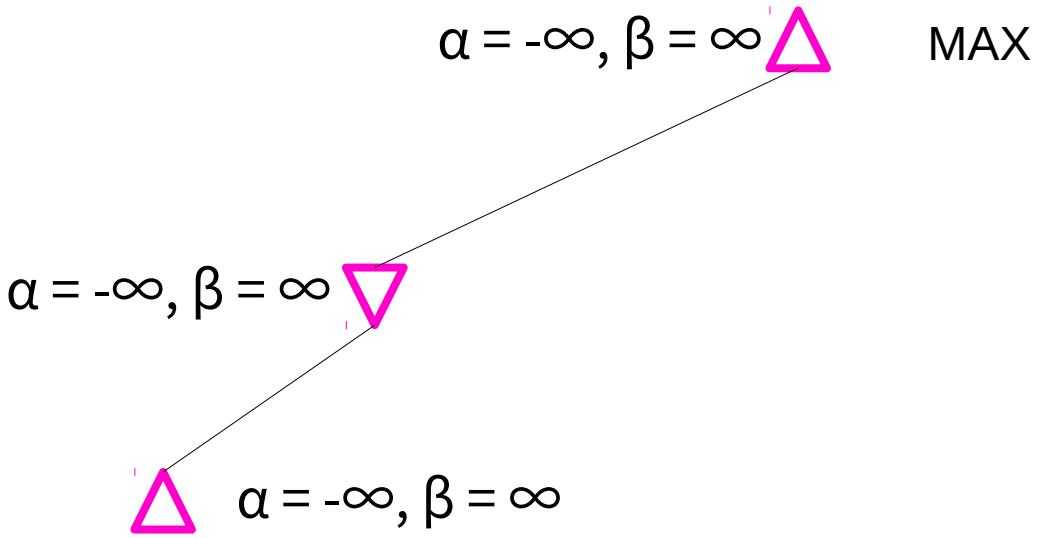
I nodi MIN che produrremo restringono l'upper bound α
I nodi MAX invece restringono il lower bound β

http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html

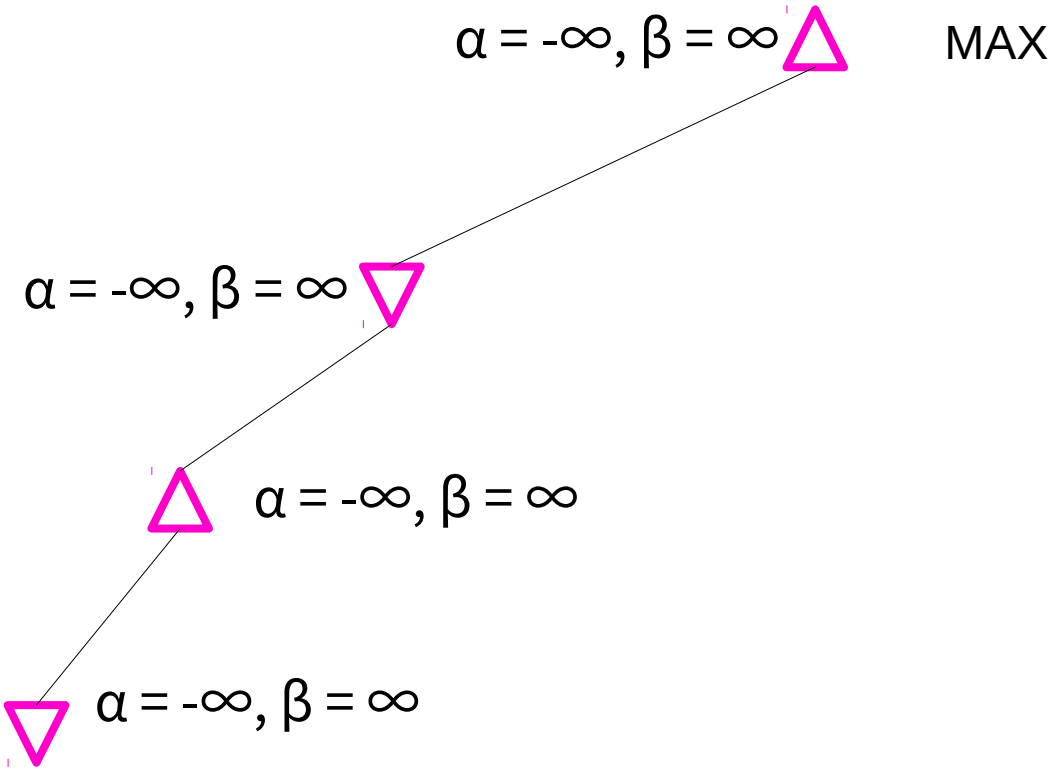
Esempio: costruisco il primo successore



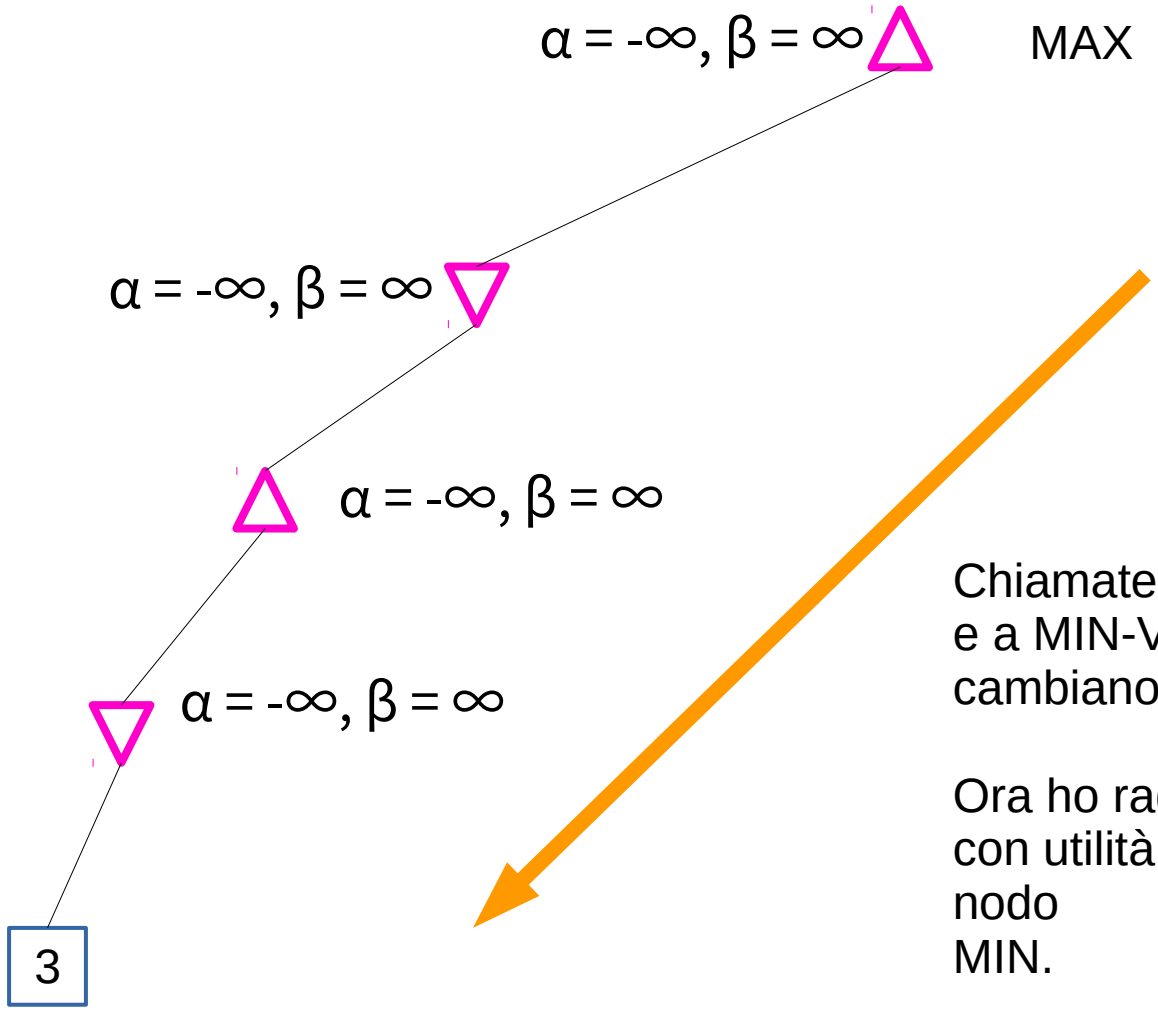
Esempio: continuo



Esempio: continuo



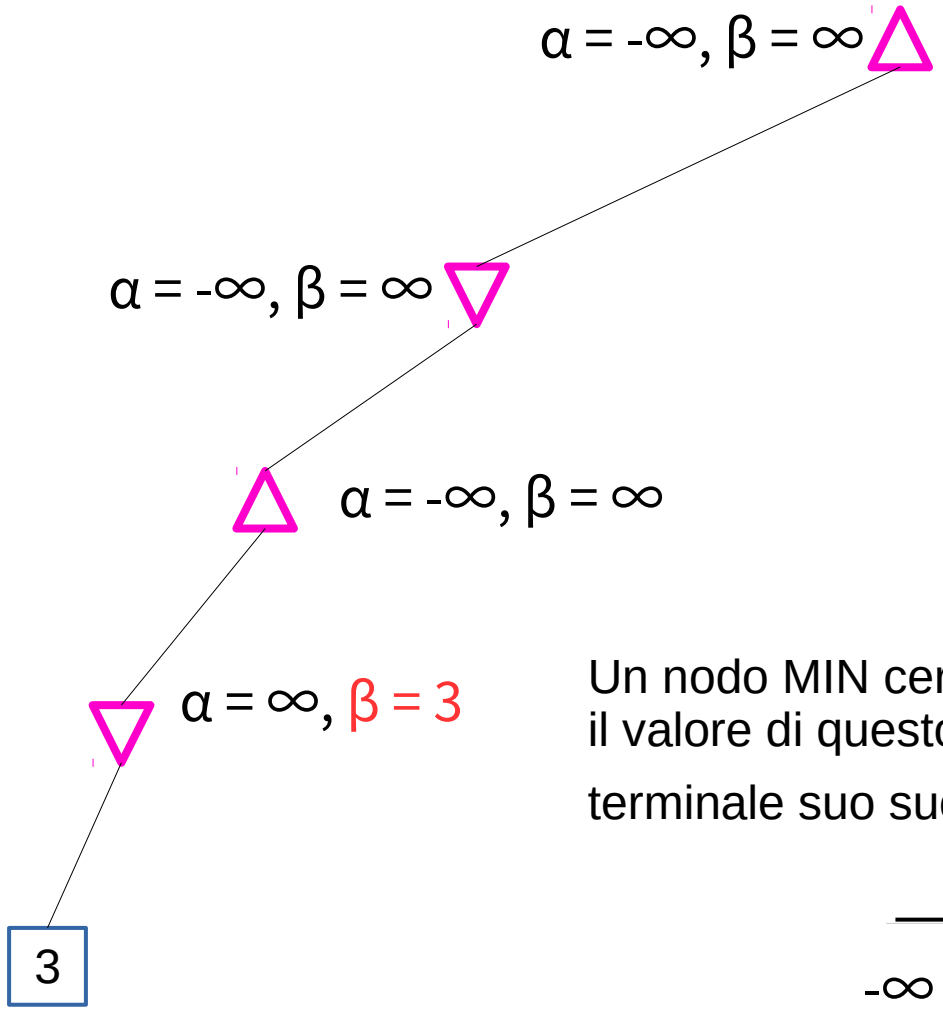
Esempio: rsggiungo il primo terminale



Chiamate ricorsive a MAX-VALUE e a MIN-VALUE. I due estremi non cambiano scendendo ricorsivamente.

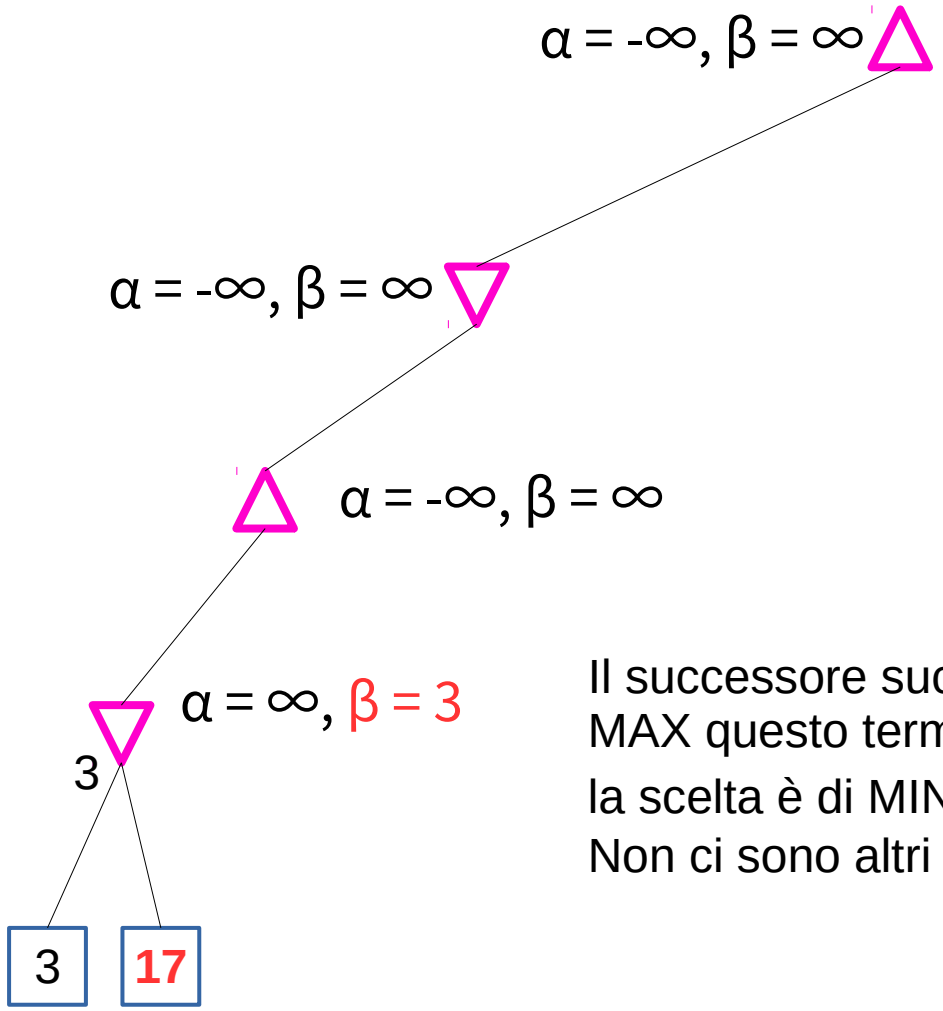
Ora ho raggiunto un nodo terminale con utilità 3. L'ho raggiunto da un nodo
MIN.

Esempio: comincio a retropropagare



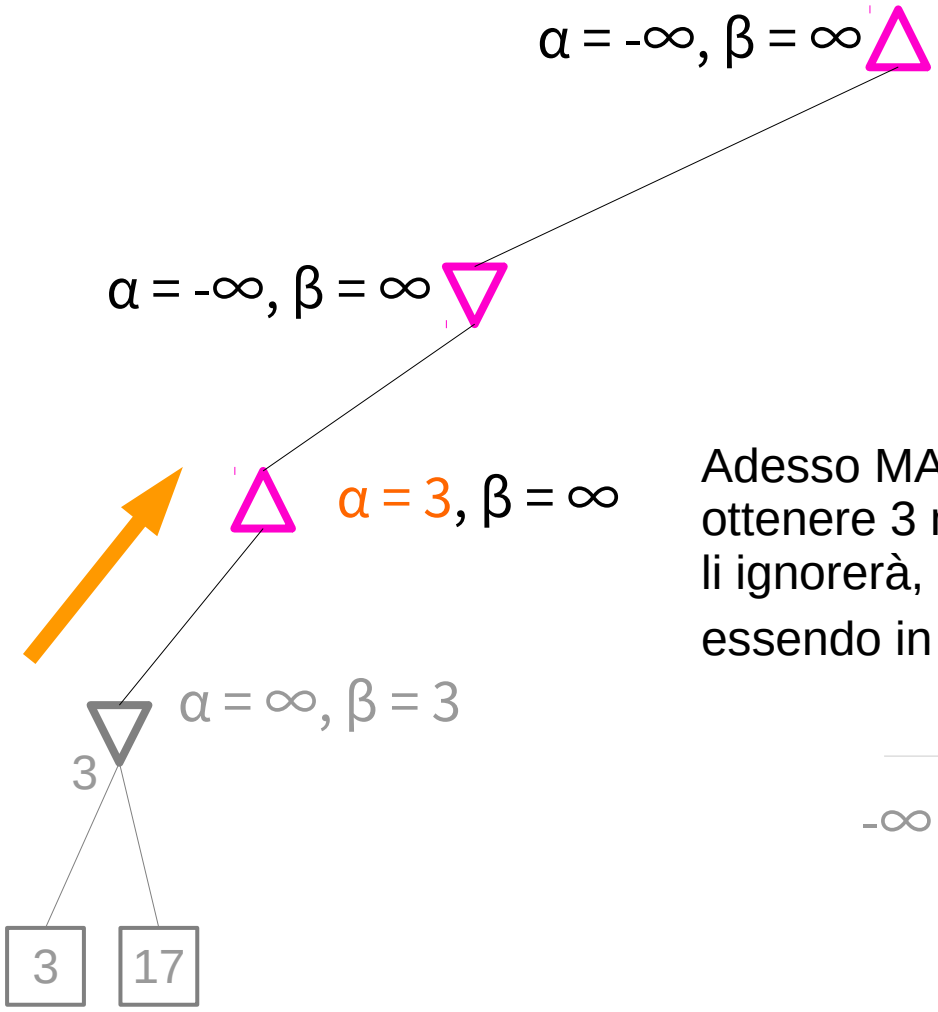
Un nodo MIN cerca di ridurre il guadagno di MAX, quindi il valore di questo nodo non sarà superiore al valore dell'unico terminale suo successore generato, cioè 3. Si modifica β.

Esempio: passo all'altro successore

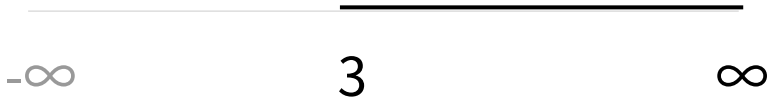


Il successore successivo ha utilità (per MAX) di 17, cioè per MAX questo terminale è migliore del predente. Poiché però la scelta è di MIN, che vuole sfavorire MAX, non si modifica β. Non ci sono altri successori. Possiamo dare un valore al nodo

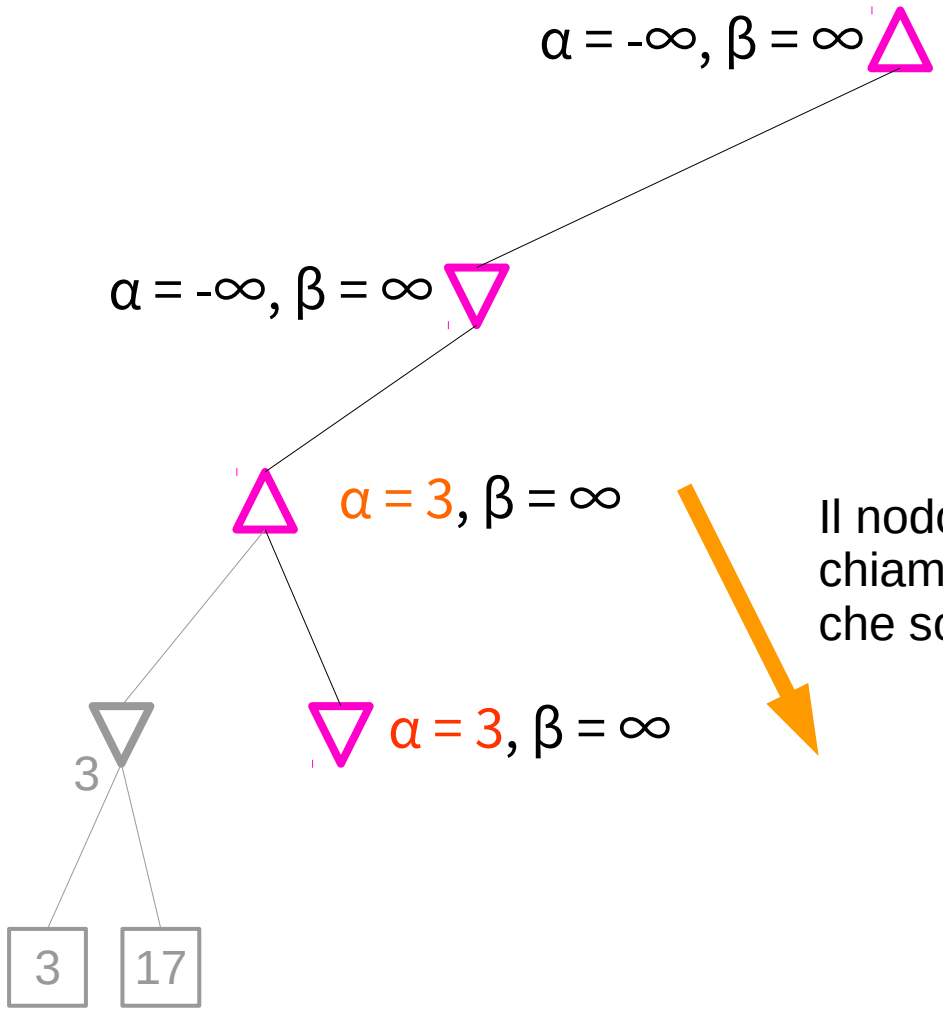
Esempio: retropropago



Adesso MAX sa che lungo il successore espanso può ottenere 3 ma deve espandere altri successori. Se peggiori li ignorerà, quindi 3 è il valore minimo che può ottenere essendo in questo nodo. Si modifica α.



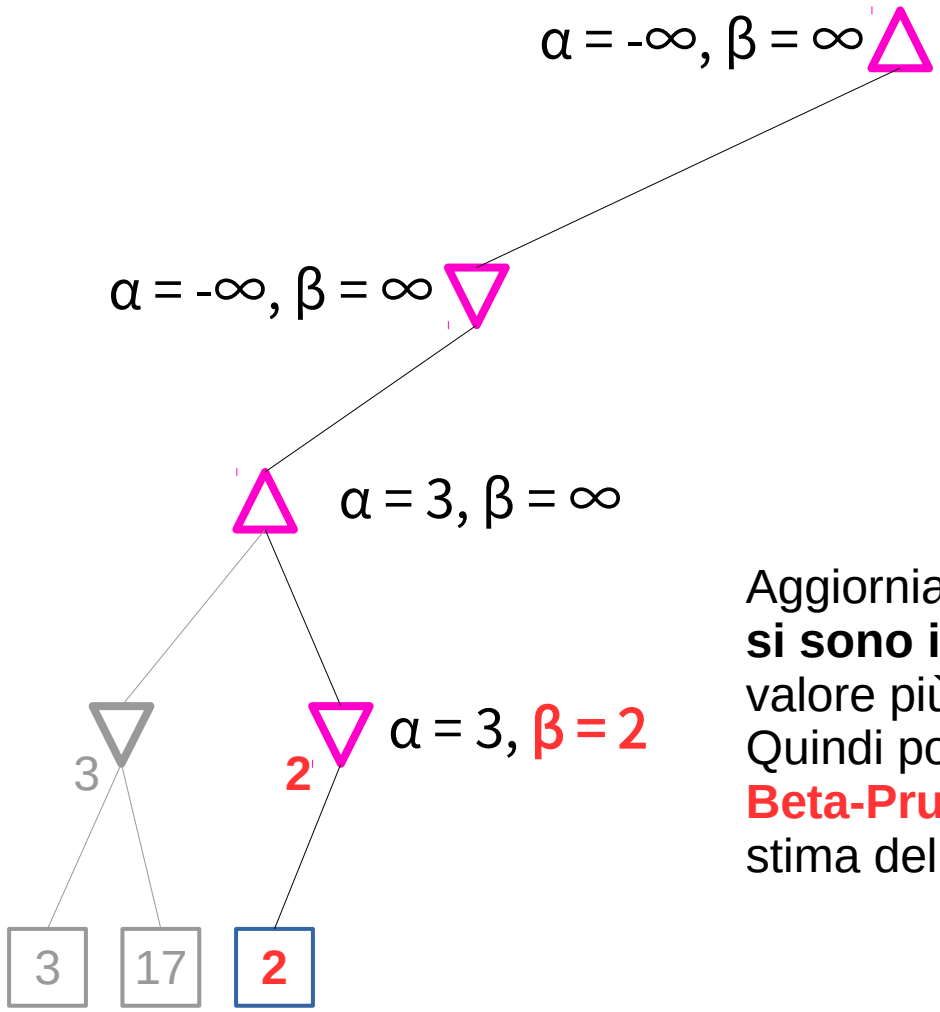
Esempio: scendo lungo un altro cammino



Il nodo MAX ha un altro successore, fa la chiamata ricorsiva passando i valori di α e β che sono stati calcolati

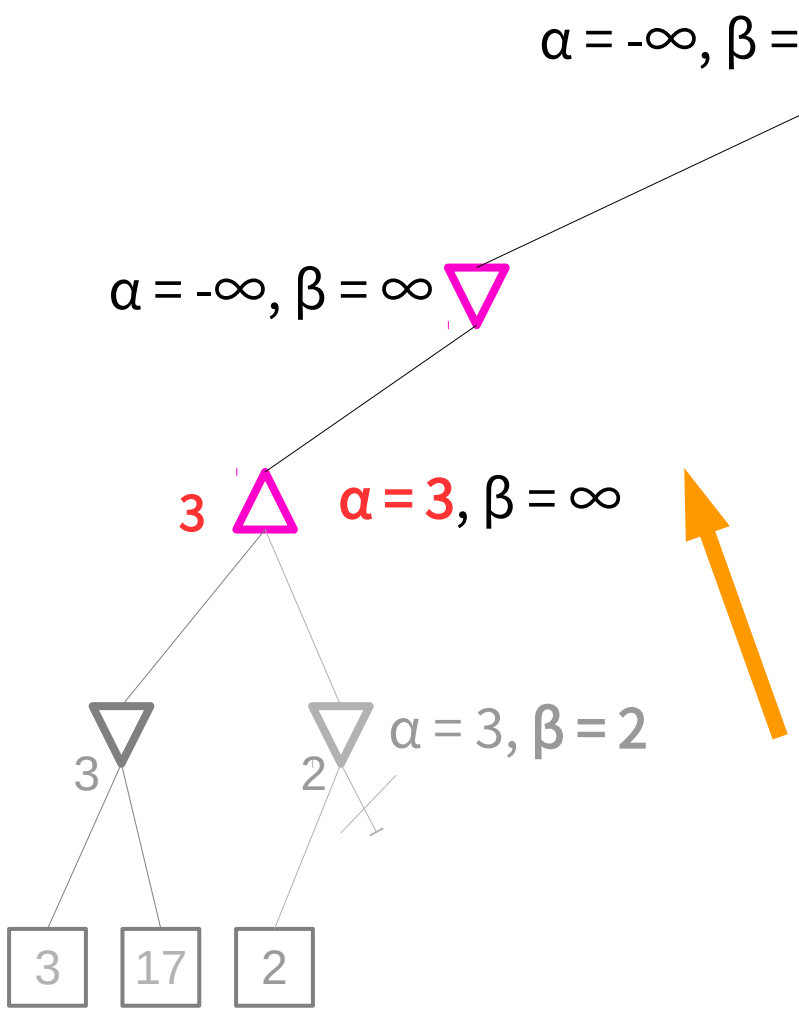
Questo nodo non è terminale, quindi creiamo il suo primo successore, che sarà un terminale

Esempio: raggiungo un terminale



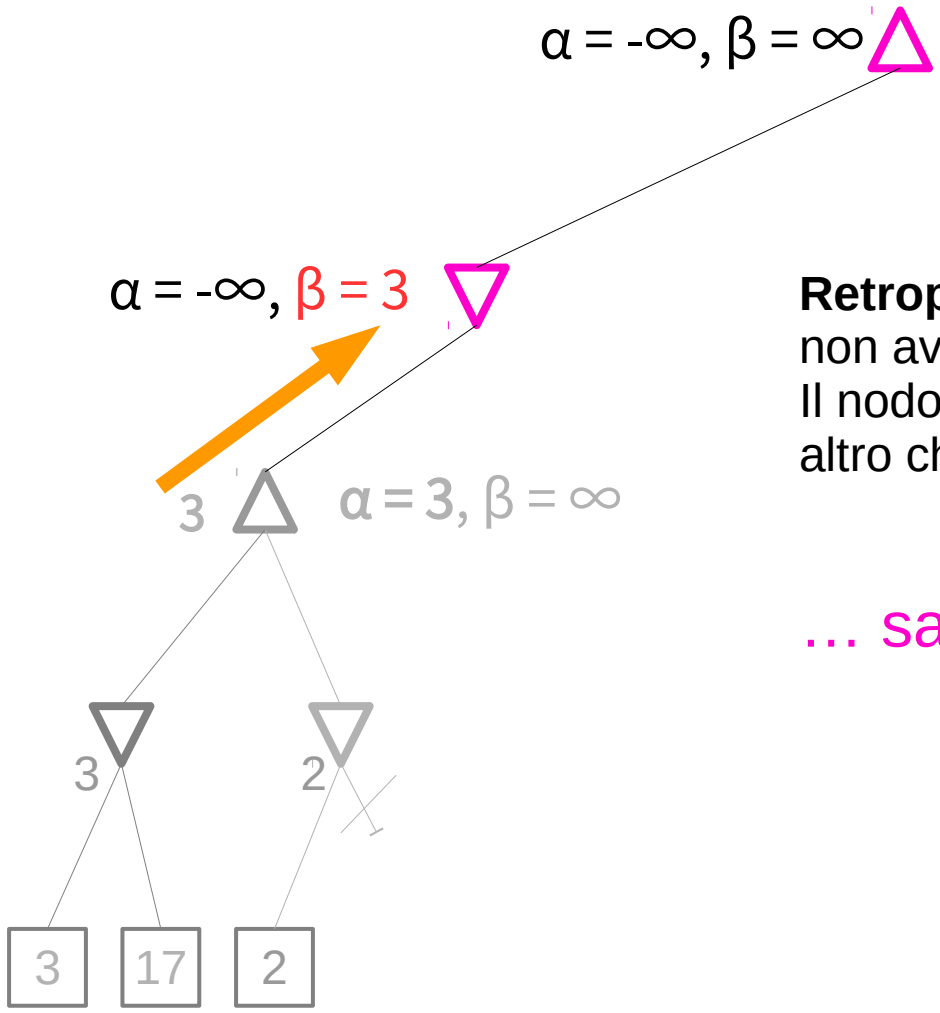
Aggiorniamo β e notiamo che **i due estremi dell'intervallo si sono incrociati.** Questo nodo MIN dovrebbe avere un valore più grande di 3 ma più piccolo di 2. **È impossibile!!** Quindi possiamo smettere di espanderlo. Si parla di **Beta-Pruning.** Il nodo prende valore 2, che è solo una stima del valore vero (abbiamo interrotto la ricerca)

Esempio: retropropago



La chiamata ricorsiva si chiude.
Dobbiamo **retropropagare** i valori calcolati.
Il valore 2 è meno buono per MAX rispetto a 3, quindi **α non cambia**.
Non ci sono altri figli, quindi possiamo dare valore al nodo (il valore 3)

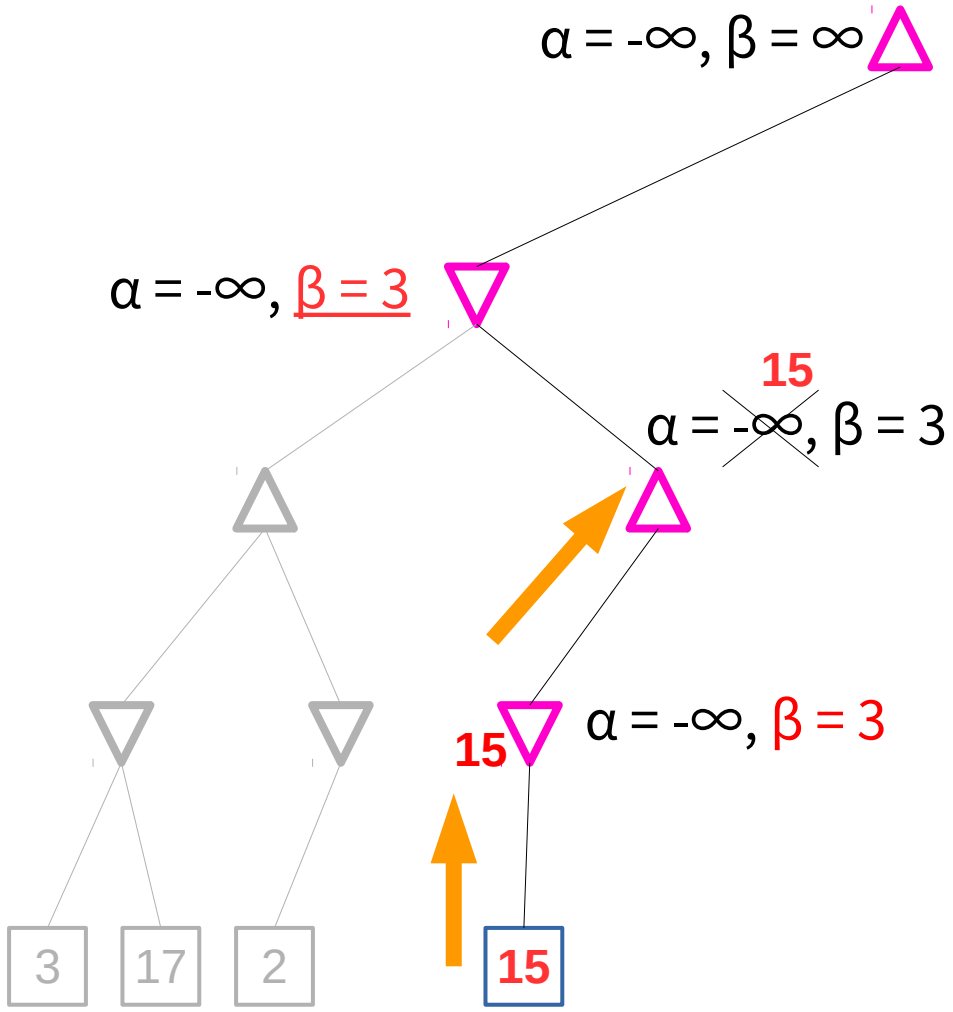
Esempio: retropropago



Retropropaghiamo ancora perché il nodo non aveva altri successori.
Il nodo MIN al momento non può garantire altro che impedire a MAX di fare meglio di 3

… saltiamo qualche mossa ...

Esempio: propagazione dei valori verso l'alto



MAX

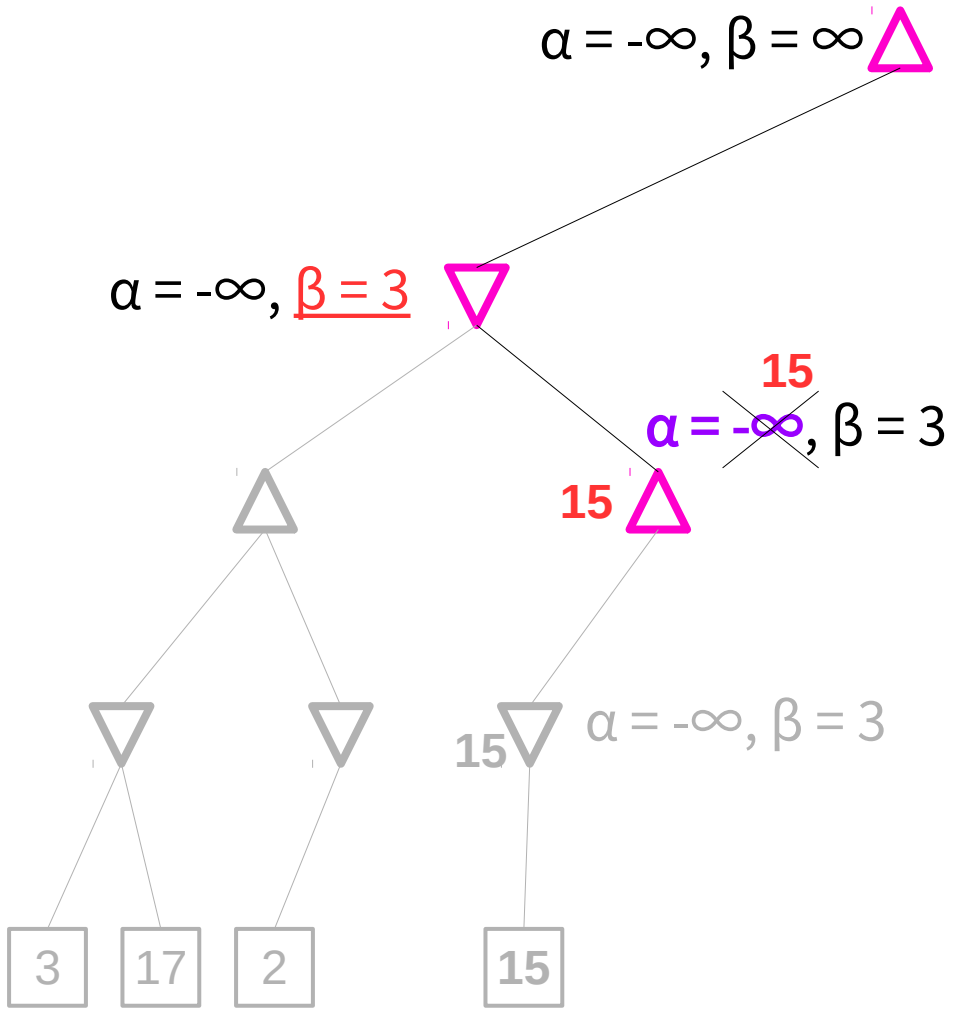
Questo è un nodo MAX, MAX qui può fare una mossa che lo porta a un nodo di valore 15, quindi il suo guadagno minimo di qui in poi è almeno 15.

Cambiamo α

Questo nodo ha un successore solo (terminale) assume come valore quello del terminale.

β ci dice che lungo il percorso c'è un alto nodo MIN (sottolineato) che può forzare MAX a un risultato peggiore quindi non cambiamo β

Esempio: propagazione dei valori verso l'alto



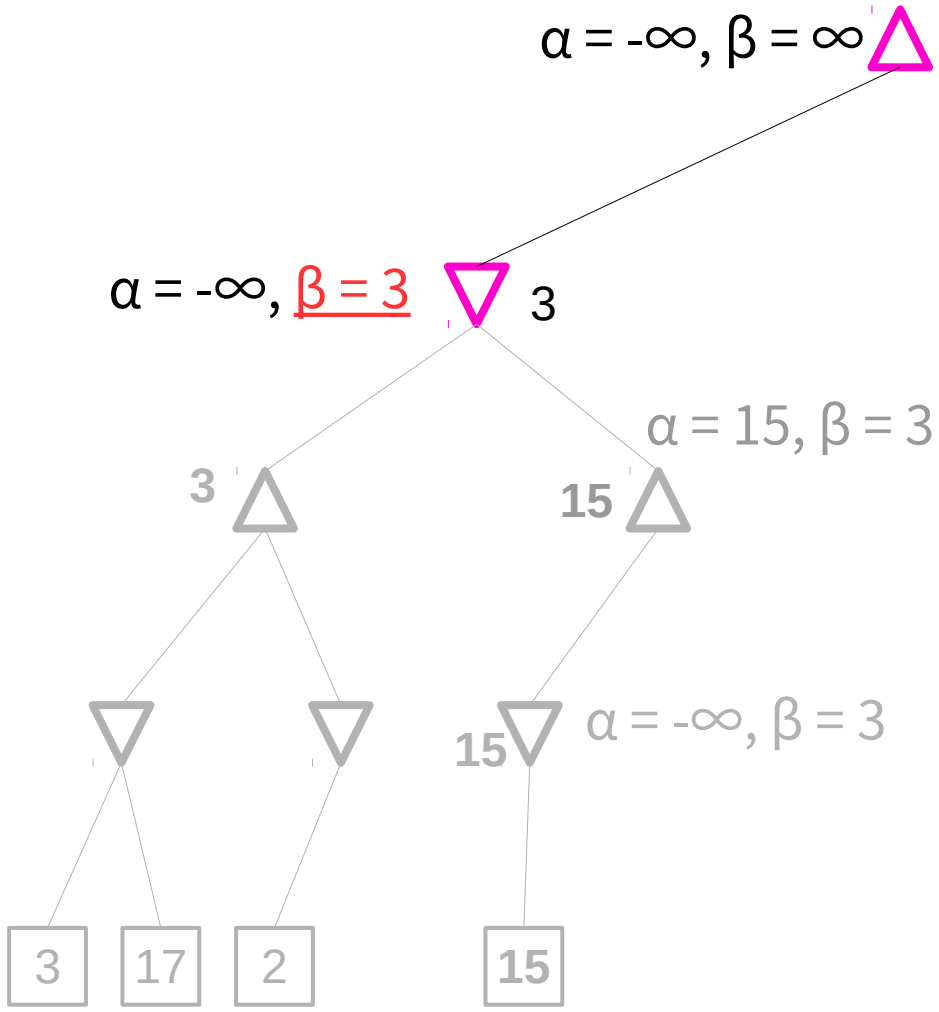
Cambiamo α

Il valore del nodo deve essere maggiore di 15 e minore di 3. Impossibile!!

L' esplorazione si ferma (**alfa pruning**).

Il nodo assume il valore stimato 15.

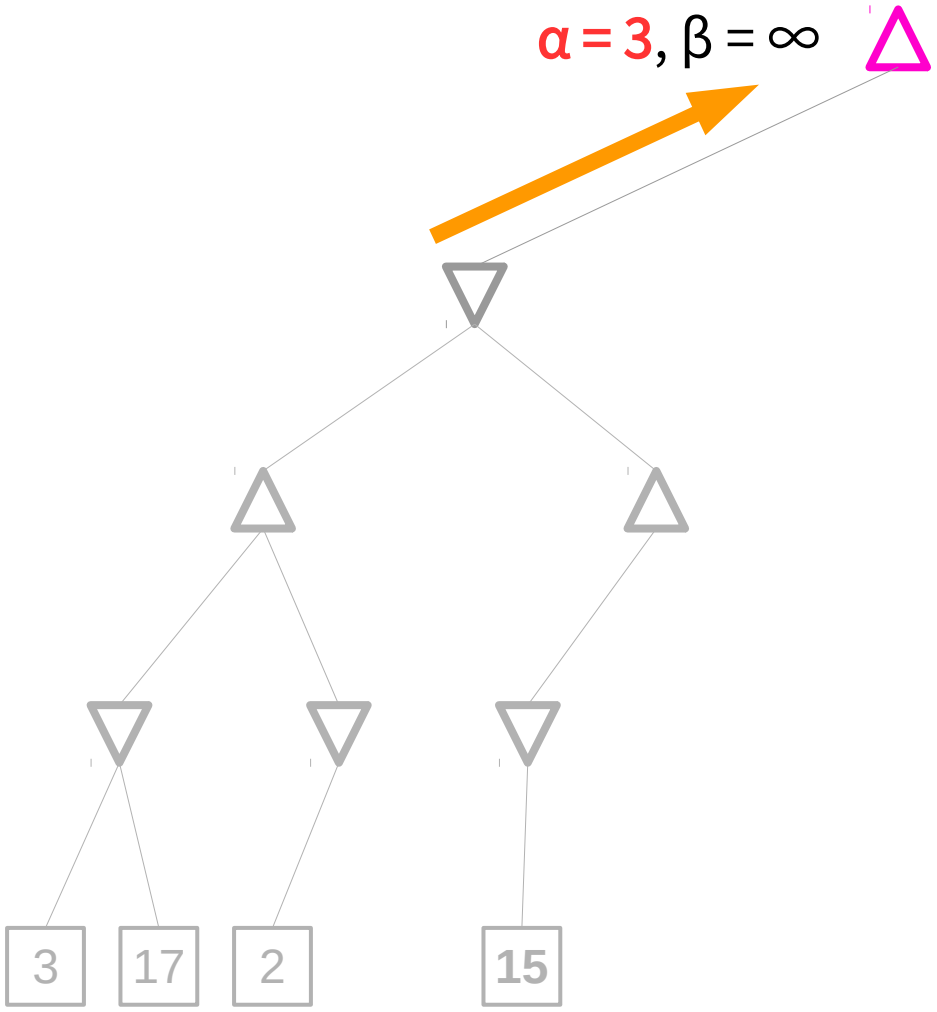
Esempio: propagazione dei valori verso l'alto



MAX

Il nodo MIN non ha altri successori. Ha il controllo su far guadagnare a MAX 3 oppure 15. Volendolo svantaggiare non modifica β

Esempio: propagazione dei valori verso l'alto

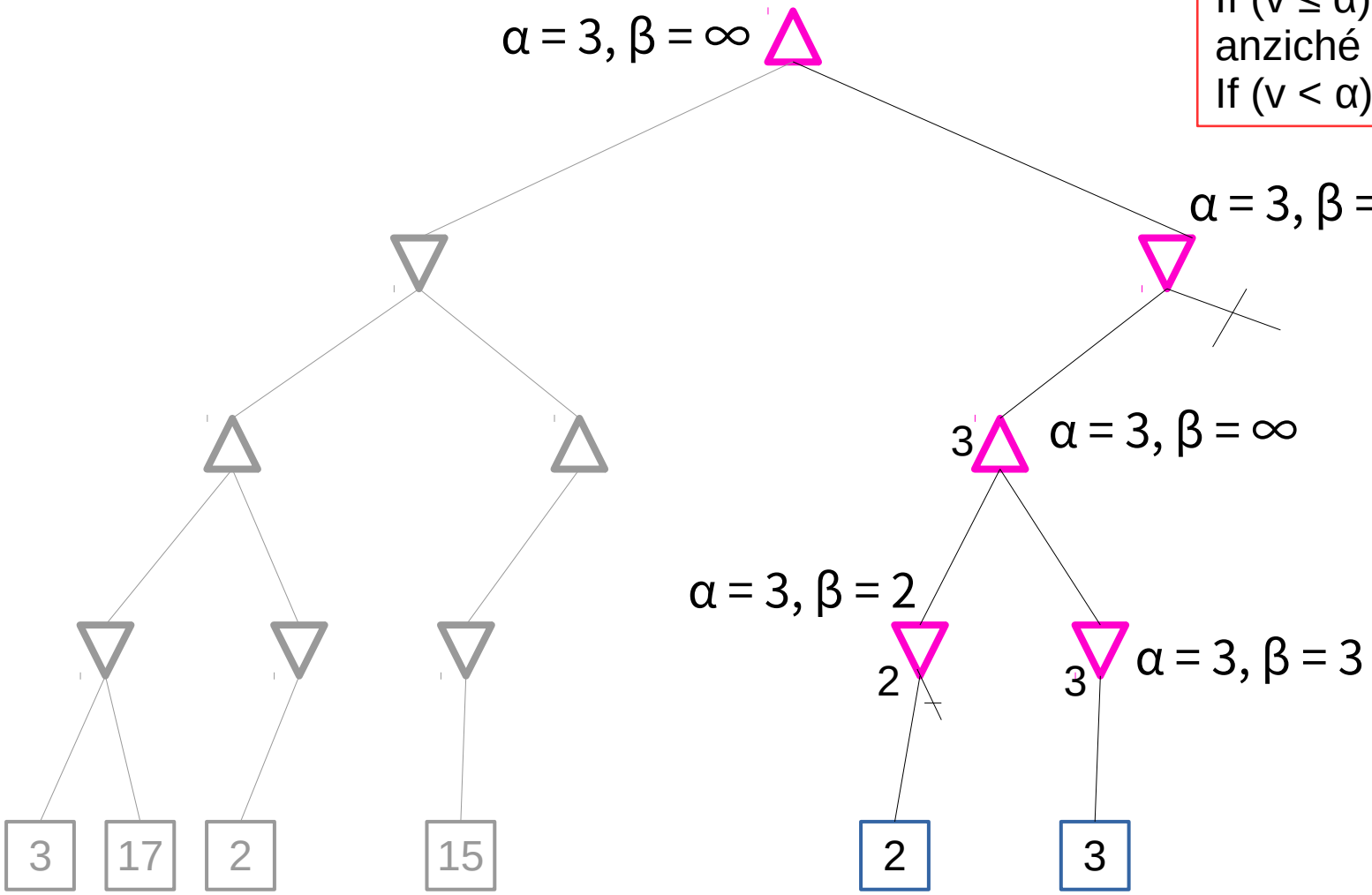


MAX

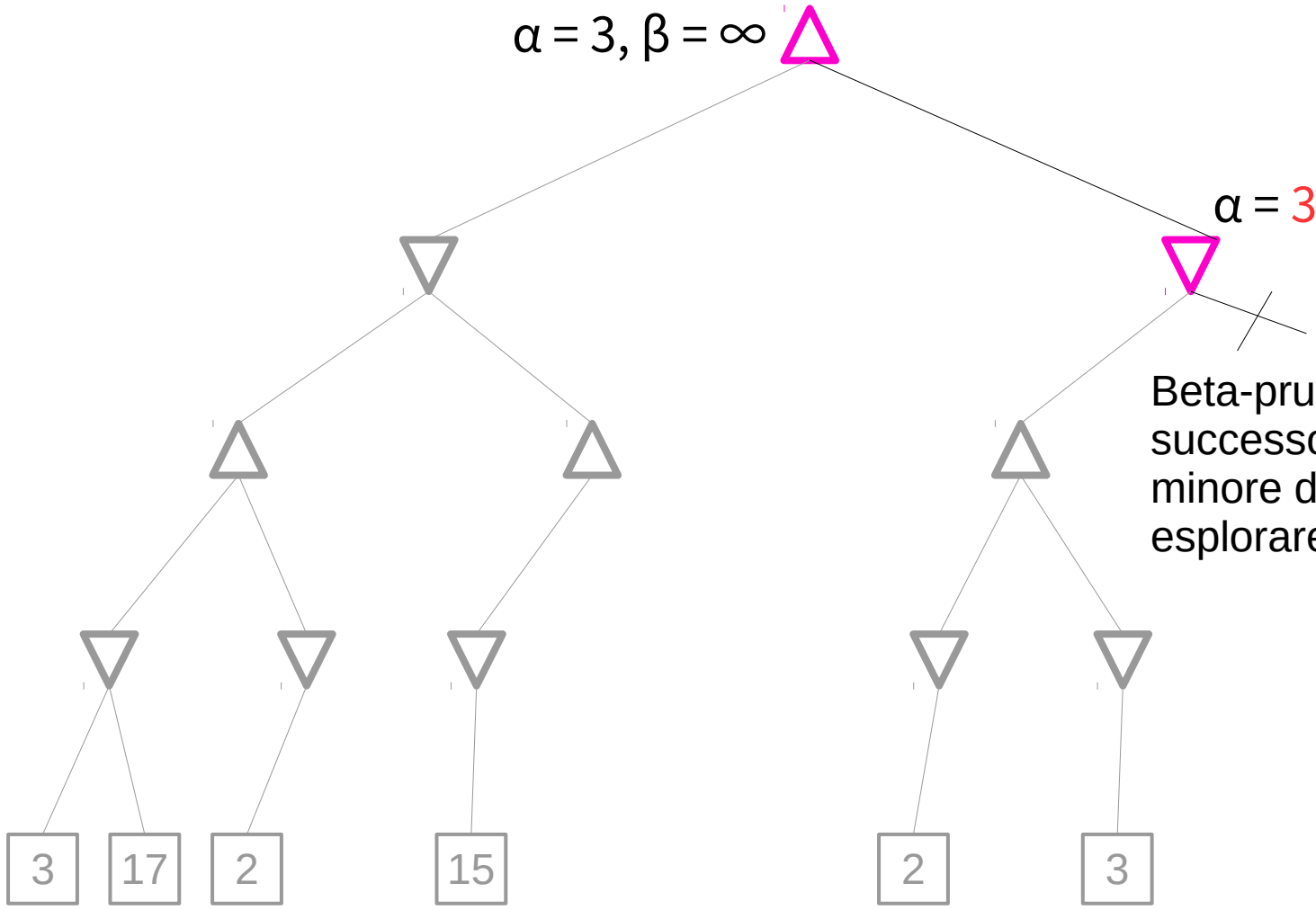
Infine retropropaghiamo alla radice

Esempio: albero di gioco completo

NB: il pruning si ha solo se min-value esegue
If (v ≤ α) return value
anziché
If (v < α) return value



Esempio: albero di gioco completo

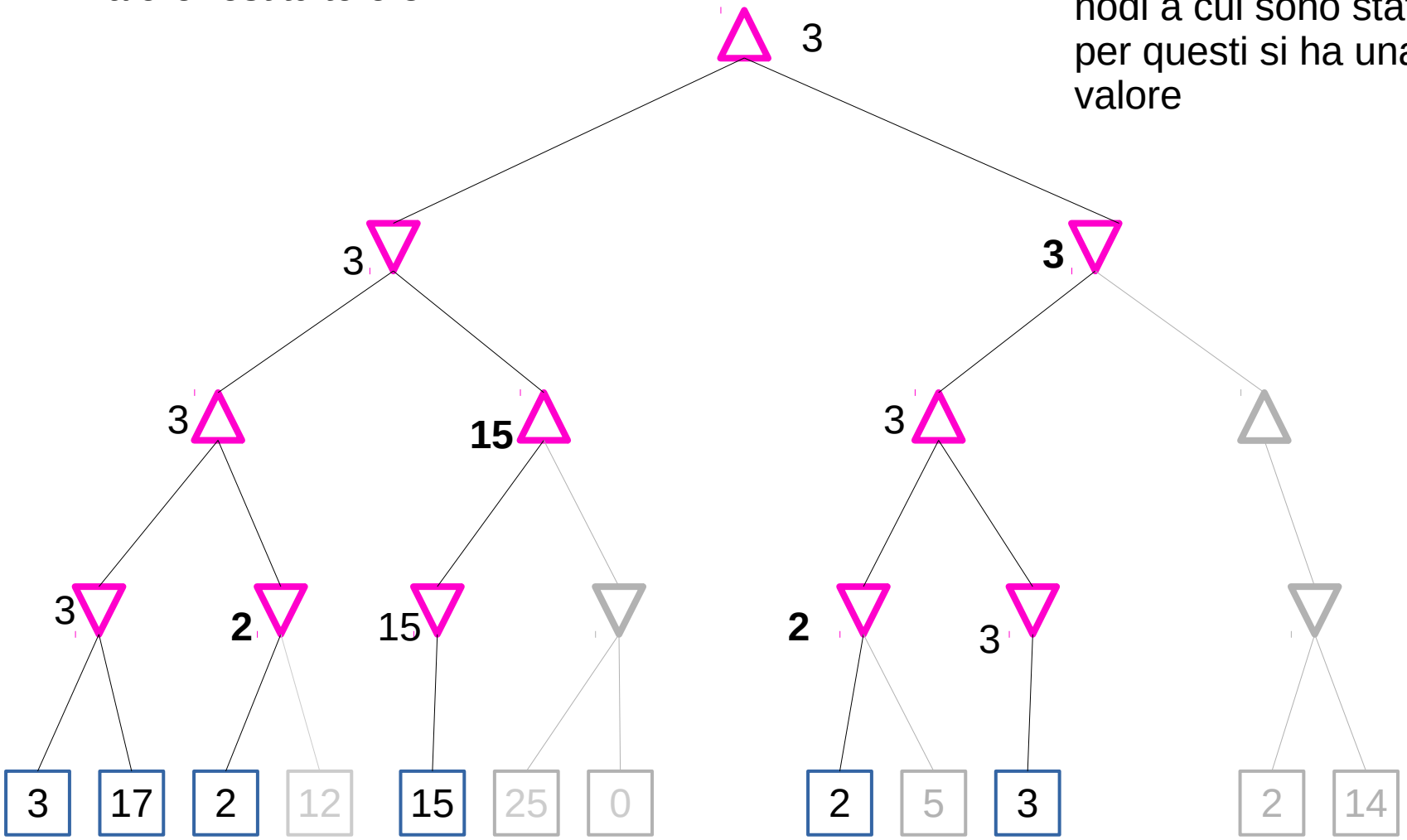


Beta-pruning. Nessun'altro successore può dare un limite minore di 3 quindi è inutile esplorare il resto dell'albero

Esempio: albero di gioco completo

Esaminati 11 su 20 nodi
Il valore restituito è 3

I valori calcolati per i nodi sono gli stessi di minimax tranne per i nodi a cui sono stati potati figli, per questi si ha una stima del valore



Confronto con minimax

- Gli algoritmi **alpha-beta pruning** e **minimax** sono equivalenti nel senso che:
  - entrambi trovano la stessa mossa ottima per il nodo radice
  - ed attribuiscono al nodo radice la stessa valutazione

- Alfa-beta raggiunge questo risultato espandendo molti meno nodi, la complessità temporale può essere $O(b^{m/2})$ contro $O(b^m)$
  - Piccolo esempio intuitivo: 2^4 contro 2^8 significa 16 contro 256 nodi espansi

Attenzione!

- Alpha-beta pruning è più o meno efficace a seconda dell' ordine con cui i successori di ciascun nodo sono considerati:
- se il successore più promettente è l' ultimo a essere considerato non è possibile evitare di esplorare i sottoalberi dei suoi fratelli
- La complessità vista nella slide precedente vale quando è possibile espandere i figli più promettenti (killer move) per primi (ordinamento dei successori)
- In questo caso il branching factor diventa circa la radice quadrata del branching factor del problema (esempio: negli scacchi il branching factor è 35, con ordinamento e alpha-beta pruning diventa 6)
- **Quando l' ordinamento dei successori non è possibile, la complessità diventa $O(b^{3m/4})$**

Come trovare le killer moves?

- Tramite apprendimento: il sistema "ricorda" le esperienze passate e usa questa esperienza per scegliere la mossa più promettente (efficace ma richiede tempo per imparare)
- Combinazione della potatura alfa-beta con iterative deepening
- Uso di tabelle di trasposizione (talvolta in aggiunta ad alfa-beta + iterative deepening)

Trasposizione

- In alcuni problemi, eseguendo un certo insieme di mosse, è possibile ottenere sempre uno stesso risultato anche se le mosse sono ordinate differentemente. I diversi ordinamenti sono detti trasposizioni

M1 M2 M3 M4

M2 M3 M1 M4

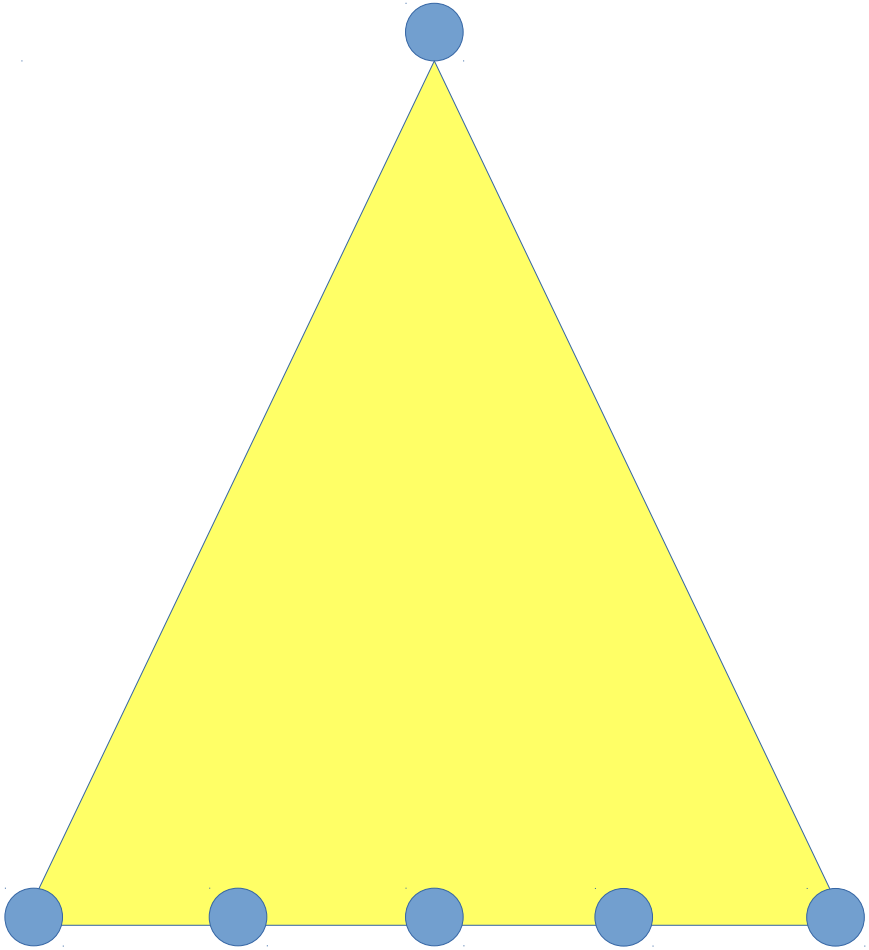
Mi mosse di un gioco
I due ordinamenti conducono a uno stesso stato

- Quando lo spazio degli stati è grande riconoscere le trasposizioni è importante per evitare di esplorare più volte gli stessi stati.
- A questo fine le trasposizioni sono conservate in una hash table (tabella delle trasposizioni) e ogni volta che si genera un nuovo stato si controlla se corrisponde a uno stato già generato da una trasposizione. Se sì non viene esplorato
- Quando lo spazio degli stati supera le capacità di memorizzazione, è talvolta necessario mantenere nella tabella delle trasposizioni soltanto quelle usate più di frequente o più di recente

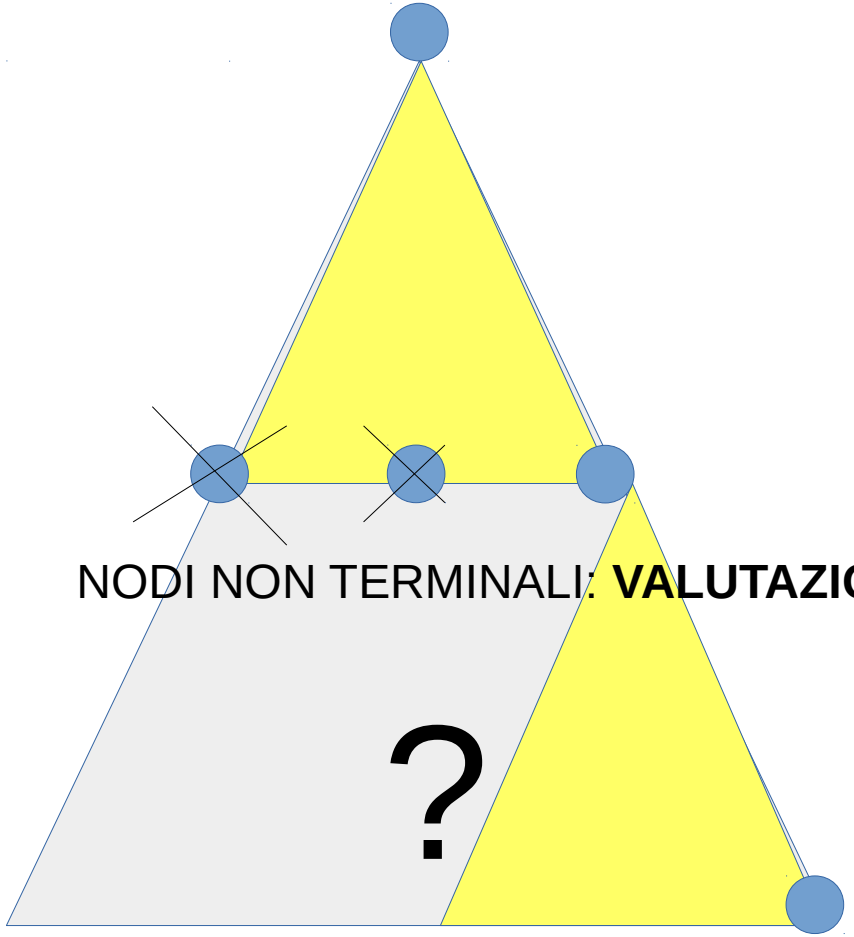
Uso di alfa-beta in contesti real-time

- Alfa-beta concentra la ricerca su una porzione limitata dello spazio degli stati ma deve comunque arrivare agli stati terminali
- Può diventare troppo lento nel produrre la risposta quando il nodo terminale è situato a grande profondità (esempio: scacchi)
- *Questo è un problema quando il problema di ricerca è in real-time, cioè tale da porre un vincolo sui tempi di risposta*
- In questo caso è necessario introdurre dei test di "cutoff" per produrre una decisione prima di raggiungere il nodo terminale

Cutoff basato su funzioni di valutazione



NODI TERMINALI: **UTILITÀ**

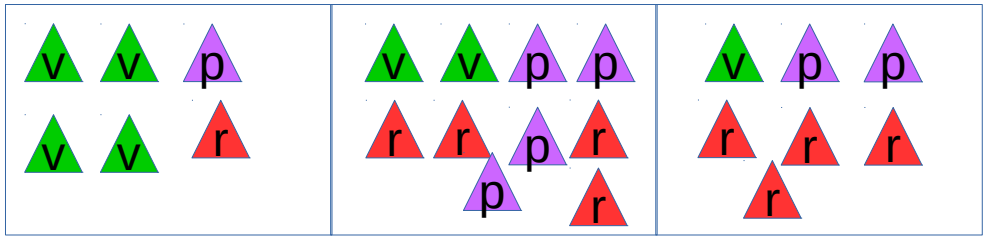


NODI NON TERMINALI: **VALUTAZIONE**

La funzione di valutazione stima il guadagno atteso essendo in un certo stato, E' usata per interrompere la ricerca lungo cammini dove la probabilità di vincere è bassa

Fz. di valutazione e categorizzazione degli stati

- In molti problemi di ricerca è possibile partizionare l' insieme degli stati basandosi su caratteristiche degli stati stessi
- Ogni insieme di stati prodotto conterrà stati che portano alla vittoria, altri che portano al pareggio, altri alla sconfitta



- Una **funzione di valutazione** fa una stima della bontà di uno stato intesa come percentuale di presenza degli stati che portano alla vittoria rispetto agli altri
- In altri termini calcola la probabilità di essere in uno stato che porta a vittoria conoscendo solo la classe di appartenenza dello stato

Troppe classi!

- Purtroppo nei problemi di interesse reale il numero delle classi che è necessario identificare perché la valutazione sia significativa è troppo alto
- **Alternativa**:
  - Identificate le caratteristiche
  - Attribuire un peso a ciascuna di esse
  - Usare una funzione lineare che combina tali caratteristiche (i loro valori o la loro presenza)
  - $\text{eval}(s) = \Sigma_{i \in [1,k]} w_i * f_i$
- **Esempio**: negli scacchi le f_i potrebbero essere il numero di pedoni, e via via degli altri pezzi ancora a disposizione del giocatore
- **NB**: quella lineare è la forma più semplice di combinazione che si possa realizzare. Ricordatevene quando parleremo di reti neurali

Alfa-beta modificato

- Data la funzione di valutazione si può modificare l'algoritmo sostituendo all'istruzione "if (TEST-TERMINALE (…) ..." l'istruzione:

  if [TEST-TAGLIO(stato, profondità)] **then** return eval(stato)

- **Quando tagliare? Tante possibilità, esempi:**

  1) Raggiunta una profondità massima predefinita

  2) Iterative deepening:

     - quando devo muovere, uso tutto il tempo disponibile per la mossa per fare cercare la mossa migliore per approfondimenti crescenti.
     - Allo scadere del tempo restituisco la mossa migliore che ho trovato

Problema dell'orizzonte

- Gli algoritmi che introducono tagli artificiali incorrono nel **problema dell' orizzonte**:
  - l' algoritmo non vede oltre il punto di taglio ma ...
  - … in alcune fasi il gioco si può capovolgere il fretta o, in altri termini, la funzione di valutazione è instabile
  - Tagliare in questi punti è prematuro perché rischioso: l' avversario potrebbe successivamente forzare un forte cambiamento della valutazione

Quiescenza

- Ne consegue che: "The decision as to whether to terminate the search at a node or continue, has to be a function of the information that exists at that node and how this relates to the **quiescence** of each and every term in the evaluation function" (tratto da: Some necessary conditions for a master chess program, Hans J. Berliner, 3rd Int. Joint Conf. On AI, 1973)
- La nozione di quiescenza concerne la **permanenza della negatività (o positività) della valutazione**
- Si taglieranno nodi la cui valutazione è quiescente mentre quelli non quiescenti richiederanno un po' di esplorazione ulteriore dei sottoalberi che li vedono come radici

- Berliner è stato il realizzatore del primo programma che abbia mai battuto un maestro in un qualsiasi gioco, nel suo caso il backgammon

Alcuni programmi che giocano

- Checkers (Samuel, Chinook)
- Othello (Logistello)
- Backgammon (TD-gammon)
- Go (AlphaGo)
- Bridge (Bridge Baron, GIB)
- Chess (DeepBlue)

DeepBlue

- Primo calcolatore a vincere una partita a scacchi contro un Campione del Mondo in carica, Garry Kasparov, con cadenza di tempo da torneo (10 febbraio 1996)
- Grande potenza computazionale:
  - un computer a parallelismo massivo a **30 nodi** basato su RS/6000, supportato da **480 processori** specifici VLSI progettati per il gioco degli scacchi
  - Sistema operativo: **AIX**
  - L'algoritmo per il gioco degli scacchi è implementato in **linguaggio C**
  - È capace di calcolare **200 milioni di posizioni al secondo**

DeepBlue

- **Algoritmo**:

  Iterative-deepening, alpha-beta search con tabella delle trasposizioni

- **Chiave del successo**: generare estensioni oltre il limite di profondità della ricerca per posizioni ritenute interessanti
- Di routine la ricerca raggiunge livello **14**, in certi casi **40**
- **Funzione di valutazione**:
  - Aveva oltre **8000 features**, inizializzate manualmente e raffinate automaticamente
  - Utilizzava un **database con 700.000 partite** di gran maestri e un ampio **database di finali di partita** (tutte quelle con 5 pezzi rimanenti e molte di quelle con 6 pezzi)
  - Kasparov ebbe il sospetto che alcune mosse fossero (scorrettamente) state suggerite da un umano

AlphaGo

- Sviluppato da google, primo programma che ha battuto senza handicap a **go** un maestro umano, su un goban di dimensioni standard (anno 2015)
- Nelle **500 partite** disputate con altri programmi per il go ha vinto:
  - **Tutte le partite meno una** quando eseguito su un solo computer
  - **Tutte** le partite quando eseguito su di un cluster che impiegava 1202 CPU e 176 GPU, circa 25 volte in più rispetto all'hardware del computer singolo
  - La versione cluster ha battuto la versione su singolo computer nel **77%** delle partite
- **Utilizza deep learning neural networks e ricerca su alberi.** Le reti neurali sono state addestrate su un **dataset di 30.000.000 di mosse** e poi raffinate giocando contro se stesse

E gli esseri umani?

- Abbiamo studiato il modo in cui diversi algoritmi di ricerca mantengono informazione sull' albero (grafo) di ricerca per determinare una soluzione
- Studio basato sul numero di nodi "da ricordare"
- **Quanti elementi è in grado di tenere contemporaneamente a mente la memoria di lavoro (o a breve termine) umana?**

Legge di Miller (1956)

- 7 ± 2

Cowan (2001)

- 4 ± 1

Cowan, N. (2001). "The magical number 4 in short-term memory: A reconsideration of mental storage capacity". Behavioral and Brain Sciences. 24: 97–185.