

# ALGORITMI DI DECISIONE SU ALBERI

PIETRO DI LENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA  
UNIVERSITÀ DI BOLOGNA

ALGORITMI E STRUTTURE DI DATI  
ANNO ACCADEMICO 2021/2022



# INTRODUZIONE

- Game tree (teoria dei giochi): albero che rappresenta tutte le possibili partite in un gioco a turni
  - nodi: rappresentano una situazione di gioco
  - archi: mosse giocabili a partire da un nodo
  - foglie: stati finali di gioco (vittoria, sconfitta, patta)
- Usati per lo sviluppo di algoritmi di decisione per giochi a turni
- Un Game Tree completo permette di determinare la mossa migliore in ogni possibile situazione di gioco
- Problema: spazio di gioco tipicamente troppo ampio da permettere la visita/generazione dell'intero albero
- Cosa vedremo:
  - Algoritmo di decisione MINIMAX su Game Tree
  - Ottimizzazione ALPHABETA pruning per MINIMAX
  - Ricerca in ampiezza ITERATIVEDEEPENING

} visita in  
profondità

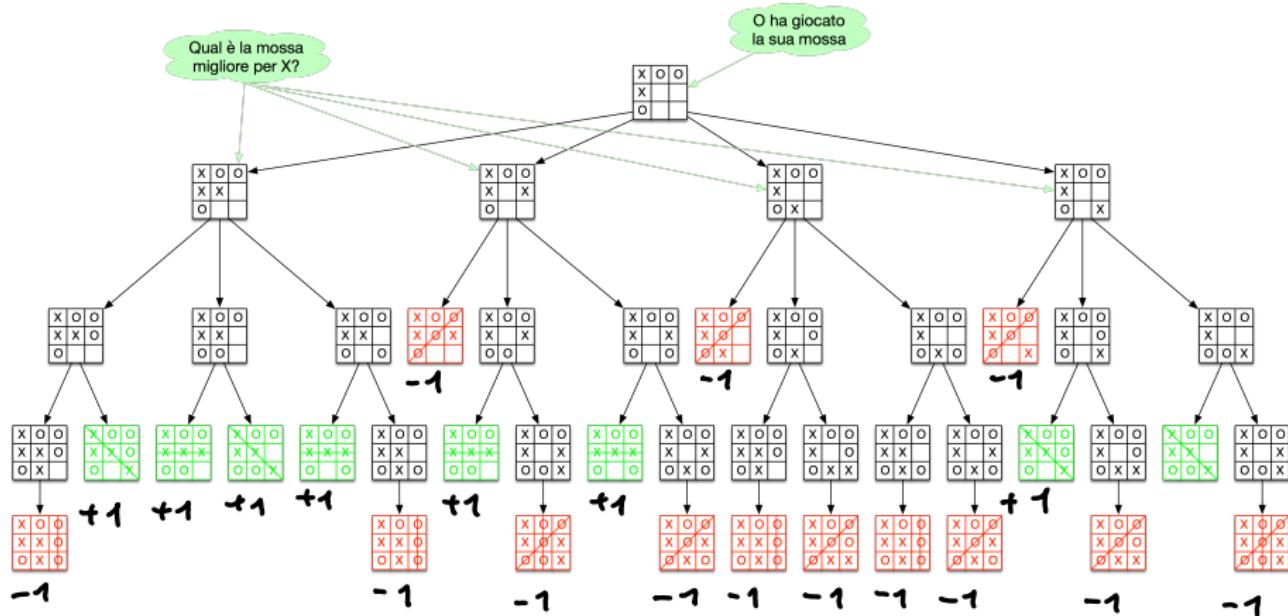
# ALGORITMO MINIMAX

- Minmax: criterio per **minimizzare la massima perdita possibile**
  - Teorema Minimax (Von Neumann, 1928)
- **Algoritmo MINIMAX**: algoritmo **ricorsivo** per individuare la migliore mossa possibile in un gioco secondo il criterio di minimizzare la massima perdita possibile
- Algoritmo **esatto** quando è possibile visitare l'intero Game Tree
  - Siamo sicuri di trovare la mossa migliore in ogni situazione

# ALGORITMO MINIMAX

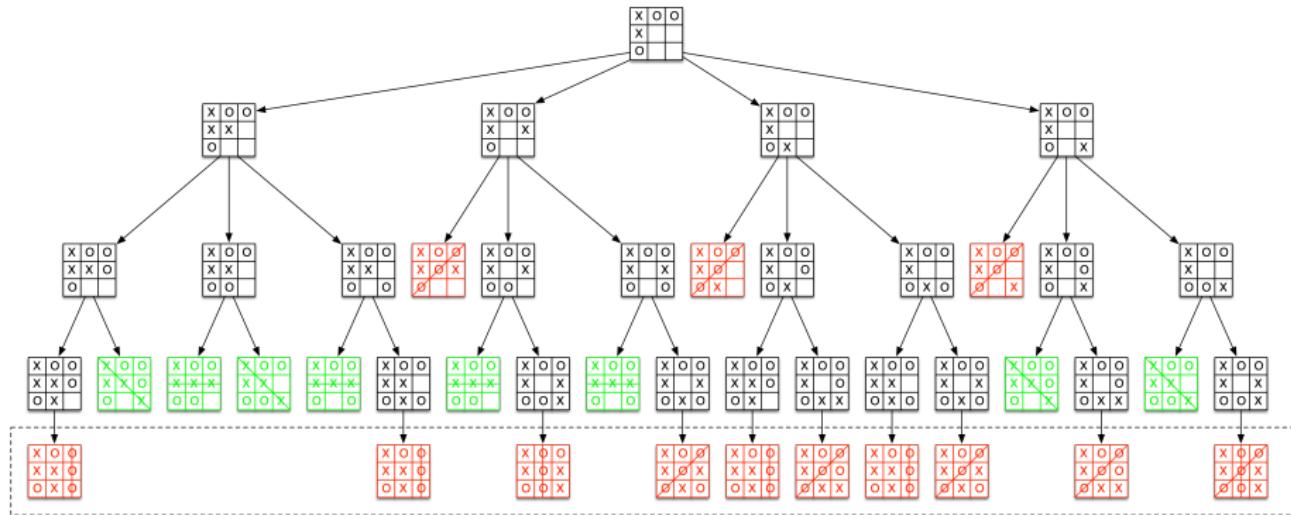
- Assumiamo di poter visitare l'intero albero di gioco
- Etichettiamo le configurazioni finali di gioco (foglie dell'albero)
  - $1 =$  vittoria
  - $0 =$  patta
  - $-1 =$  sconfitta
- Propaghiamo le label dalle foglie fino alla radice:
  - Turno (nodo) del Giocatore A (**massimizzazione**): assegniamo al nodo il **massimo** tra i valori assegnati ai figli (mosse del Giocatore B)
  - Turno (nodo) del Giocatore B (**minimizzazione**): assegniamo al nodo il **minimo** tra i valori assegnati ai figli (mosse del Giocatore A)
- Nota: Giocatore A  $\Rightarrow$  vittoria= $-1$ , Giocatore B  $\Rightarrow$  vittoria= $1$

# ESEMPIO: ALGORITMO MINIMAX



- Verde (+1): X vince (giocatore che minimizza)
- Rosso (-1): O vince (giocatore che massimizza)
- Non può finire in patta (0)

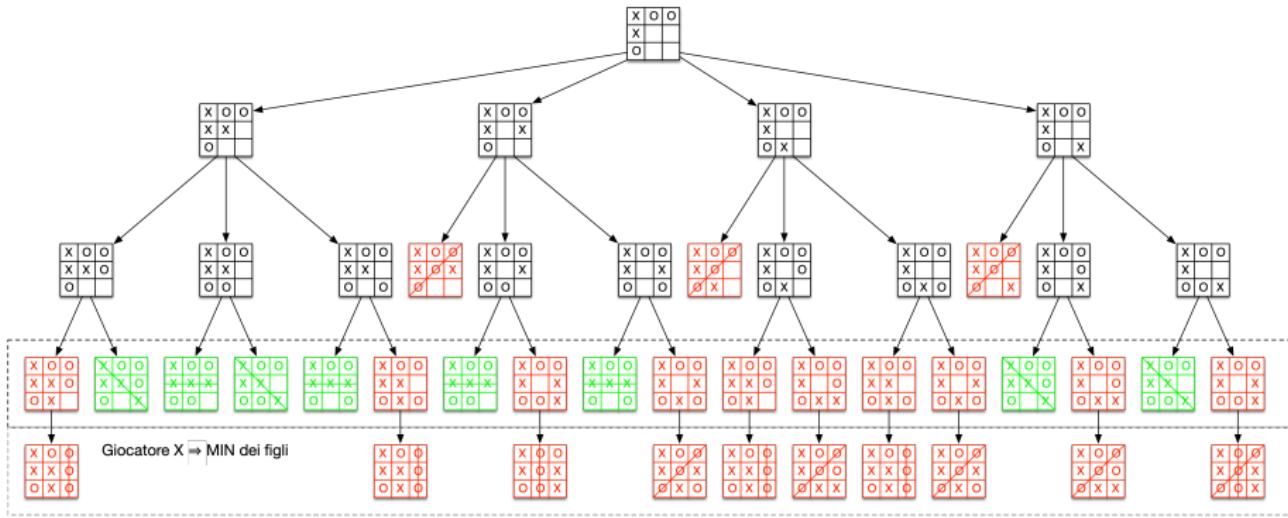
### ESEMPIO: ALGORITMO MINIMAX



**Giocatore O** → MAX dei figli (in questo caso sono tutte foglie quindi assegniamo solo lo score di fine partita: vince sempre O)

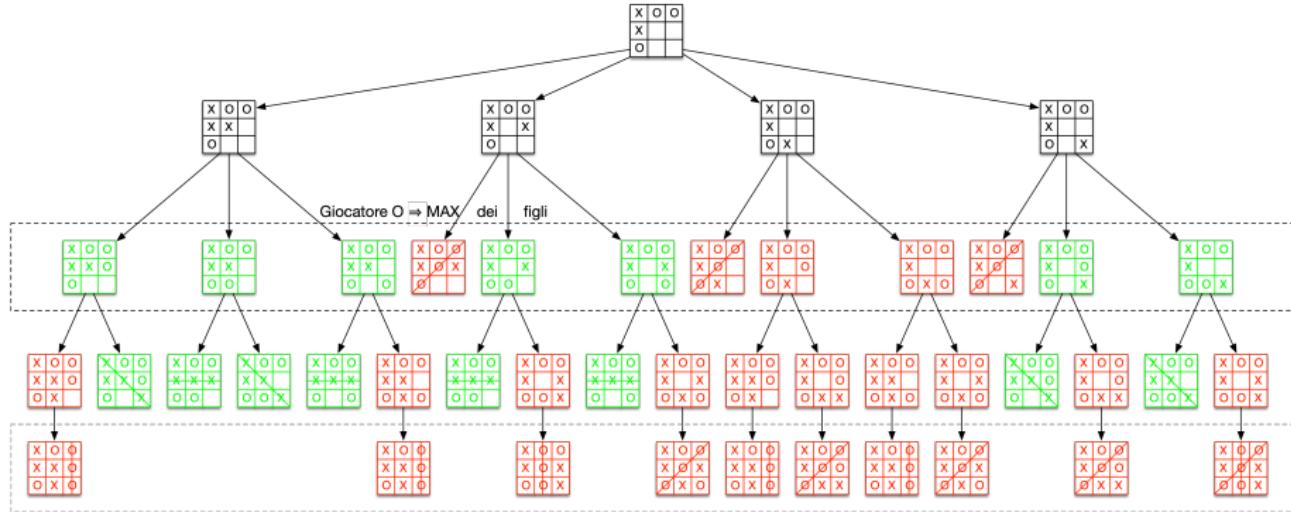
- Verde (-1): X vince
  - Rosso (-1): O vince

# ESEMPIO: ALGORITMO MINIMAX



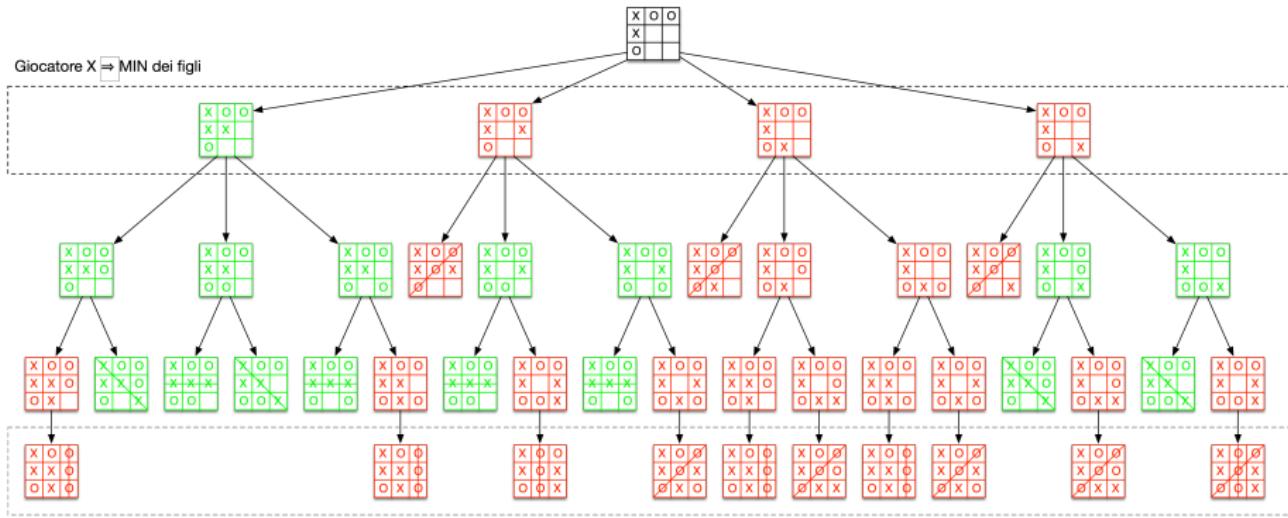
- **Verde (-1):** X vince
- **Rosso (-1):** O vince

# ESEMPIO: ALGORITMO MINIMAX



- **Verde (-1):** X vince
- **Rosso (-1):** O vince

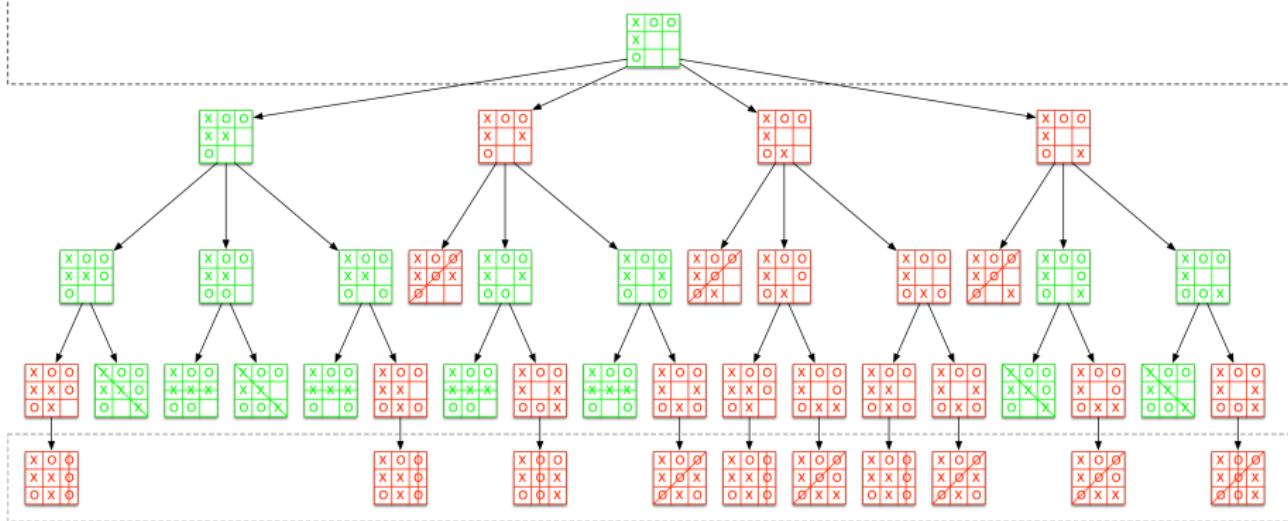
# ESEMPIO: ALGORITMO MINIMAX



- **Verde ( 1):** X vince
- **Rosso (-1):** O vince

# ESEMPIO: ALGORITMO MINIMAX

Giocatore O  MAX dei figli



Giocatore O  MAX dei figli (in questo caso sono tutte foglie quindi assegnano solo lo score di fine partita: vince sempre O)

- **Verde ( -1):** X vince
- **Rosso ( -1):** O vince

# ALGORITMO MINIMAX: PSEUDOCODICE

+1 A  $\Rightarrow$  O massimizza | B  $\Rightarrow$  X minimizza -1

```
1: function MINIMAX(TREE  $T$ , BOOL  $playerA$ )  $\rightarrow$  INT
2:   if ISLEAF( $T$ ) then
3:     eval = EVALUATE( $T$ )
4:   else if  $playerA == \text{true}$  then                                 $\triangleright MAX\ player$ 
5:     eval =  $-\infty$ 
6:     for  $c \in \text{CHILDREN}(T)$  do
7:       eval = MAX(eval,MINIMAX( $c,\text{true}$ ))
8:   else                                                  $\triangleright MIN\ player$ 
9:     eval =  $\infty$ 
10:    for  $c \in \text{CHILDREN}(T)$  do
11:      eval = MIN(eval,MINIMAX( $c,\text{false}$ ))
12:    return eval
```

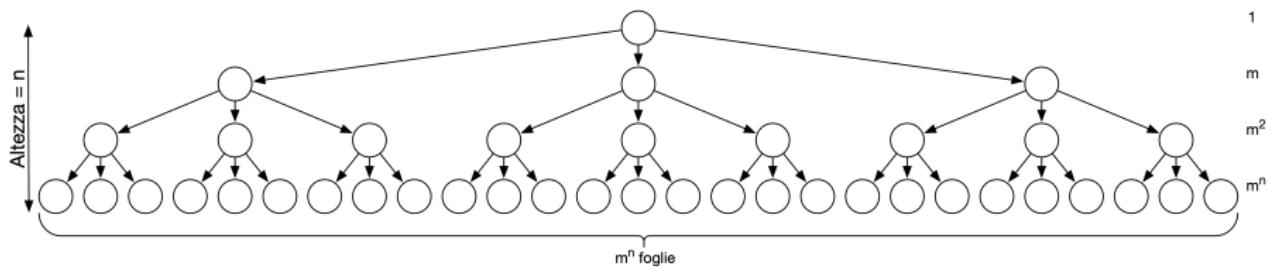
- ISLEAF( $T$ ): verifica che il nodo  $T$  sia o meno una foglia
- CHILDREN( $T$ ): ritorna la lista di figli del nodo  $T$
- EVALUATE( $T$ ): ritorna  $-1$  (sconfitta),  $0$  (patta) o  $1$  (vittoria)

# ANALISI DELL'ALGORITMO MINIMAX

- Che tipo di visita implementa l'algoritmo MINIMAX?
  - Visita in profondità post-ordine: bisogna valutare tutti i figli prima di poter assegnare una valutazione al nodo corrente
- Quanto costa l'algoritmo MINIMAX?
  - Tempo:  $\Theta(n)$  (visita ogni nodo esattamente una volta)
  - Memoria:  $O(h)$  ( $h = \text{altezza dell'albero}$ )
- Quanto è grande l'albero di gioco?
  - Dipende dal gioco
  - E' complesso calcolare il numero esatto di nodi di un Game Tree
  - Possiamo trovare un upper bound al Game Tree effettivo ammettendo configurazioni di gioco *illegali*, cioè configurazioni che non possono mai apparire in una partita reale

# COMPLESSITÀ: GIOCO CON $m$ MOSSE E $n$ TURNI

- **Definizione.** Un giocatore ha a disposizione al massimo  $m$  mosse possibili per turno e il gioco termina in massimo  $n$  turni
  - Ex. Forza 4,  $m = 7$  (colonne matrice),  $n = 6 \times 7$  (righe  $\times$  colonne)
- Upper bound al Game Tree effettivo (altezza =  $O(n)$ )

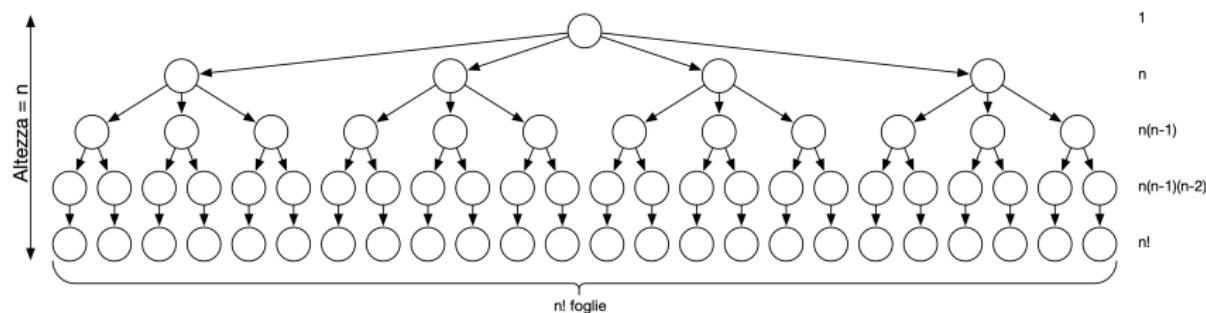


- Il numero di partite possibili  $P(m, n)$  è dato dal numero di foglie
$$P(m, n) \leq m^n = O(m^n)$$
- Il numero totale di nodi  $T(m, n)$  del Game Tree è limitato da:

$$T(m, n) \leq 1 + m + m^2 + \cdots + m^n = \sum_{k=0}^m m^k = \frac{m^{n+1} - 1}{m - 1} = O(m^n)$$

# COMPLESSITÀ: GIOCO CON $n$ OGGETTI 1/2

- **Definizione.** Un giocatore può scegliere/rimuovere un solo oggetto per turno su  $n$  oggetti totali a disposizione
  - Ex. Tris,  $n = 3 \times 3$  (righe  $\times$  colonne della matrice)
- Upper bound al Game Tree effettivo (altezza =  $O(n)$ )



- Il numero di partite possibili  $P(n)$  è dato dal numero di foglie

$$P(n) \leq n! = O(n!)$$

## COMPLESSITÀ: GIOCO CON $n$ OGGETTI 2/2

- Il numero totale di nodi  $T(n)$  del Game Tree è limitato da:

$$T(n) \leq 1 + n + n(n-1) + n(n-1)(n-2) + \cdots + n!$$

$$= \sum_{k=0}^n \frac{n!}{(n-k)!}$$

$$= n! \sum_{k=0}^n \frac{1}{(n-k)!} \quad n! \left( \frac{1}{n!} + \frac{1}{(n-1)!} + \cdots + \frac{1}{0!} \right)$$

$$= n! \sum_{k=0}^n \frac{1}{k!}$$

$$\leq n! \sum_{k=0}^{\infty} \frac{1}{k!}$$

$$= n!e$$

(numero di Eulero  $e \approx 2.72$ )

$$= O(n!)$$

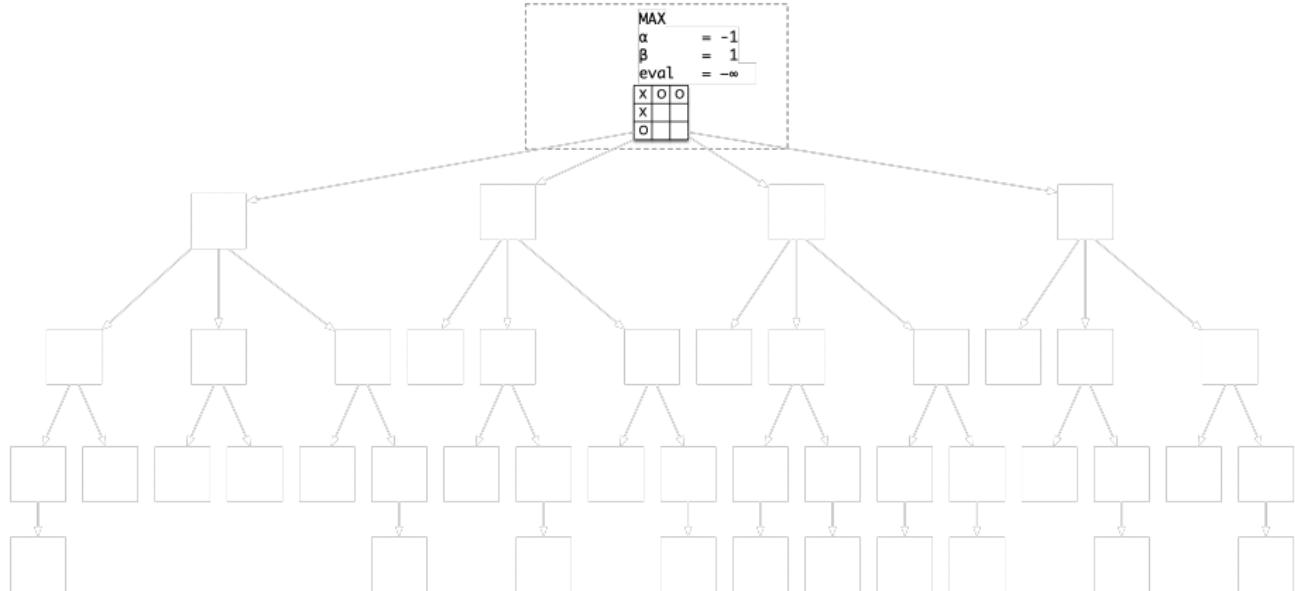
# ALGORITMO ALPHABETA PRUNING

- Ottimizzazione per minimizzare il numero di nodi valutati da MINIMAX
- E' davvero necessario visitare tutto l'albero?
  - Interrompiamo la visita su un sottoalbero quando siamo sicuri di non poter trovare una soluzione migliorare di quella attuale
  - **MIN**: stop quando uno dei figli ha score **minimo assoluto** (Es. -1)
  - **MAX**: stop quando uno dei figli ha score **massimo assoluto** (Es. 1)
- ALPHABETA generalizza questa idea: aggiorniamo iterativamente
  - $\alpha$ : **punteggio minimo** ottenibile dal **Giocatore A** (che massimizza)
  - $\beta$ : **punteggio massimo** ottenibile dal **Giocatore B** (che minimizza)
  - $[\alpha, \beta]$  rappresenta il range di possibili risultati per un sottoalbero
  - Se ad un certo punto  $\beta \leq \alpha$  allora il sottoalbero relativo non potrà contenere una soluzione ottima e lo possiamo ignorare (in altri termini, *potiamo* il ramo relativo al sottoalbero)
- Calcola la stessa soluzione di MINIMAX visitando meno nodi

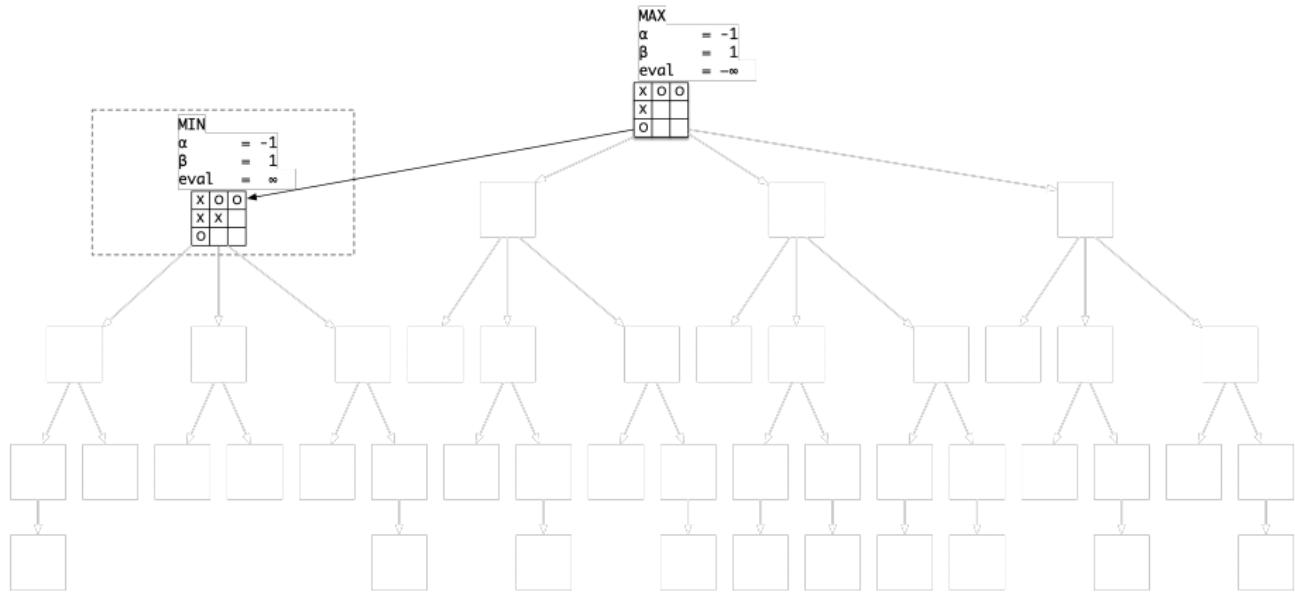
# ALGORITMO ALPHABETA: PSEUDOCODICE

```
1: function ALPHABETA(TREE  $T$ , BOOL  $playerA$ , INT  $\alpha$ , INT  $\beta$ )  $\rightarrow$  INT
2:   if ISLEAF( $T$ ) then
3:     eval = EVALUATE( $T$ )
4:   else if  $playerA == \text{true}$  then                                 $\triangleright MAX\ player$ 
5:     eval =  $-\infty$ 
6:     for  $c \in CHILDREN(T)$  do
7:       eval = MAX(eval, ALPHABETA( $c, \text{true}, \alpha, \beta$ ))
8:        $\alpha = \text{MAX}(eval, \alpha)$ 
9:       if  $\beta \leq \alpha$  then                                          $\triangleright \beta\ cutoff$ 
10:        break
11:   else                                                  $\triangleright MIN\ player$ 
12:     eval =  $\infty$ 
13:     for  $c \in CHILDREN(T)$  do
14:       eval = MIN(eval, ALPHABETA( $c, \text{false}, \alpha, \beta$ ))
15:        $\beta = \text{MIN}(eval, \beta)$ 
16:       if  $\beta \leq \alpha$  then                                          $\triangleright \alpha\ cutoff$ 
17:         break
18:   return eval
```

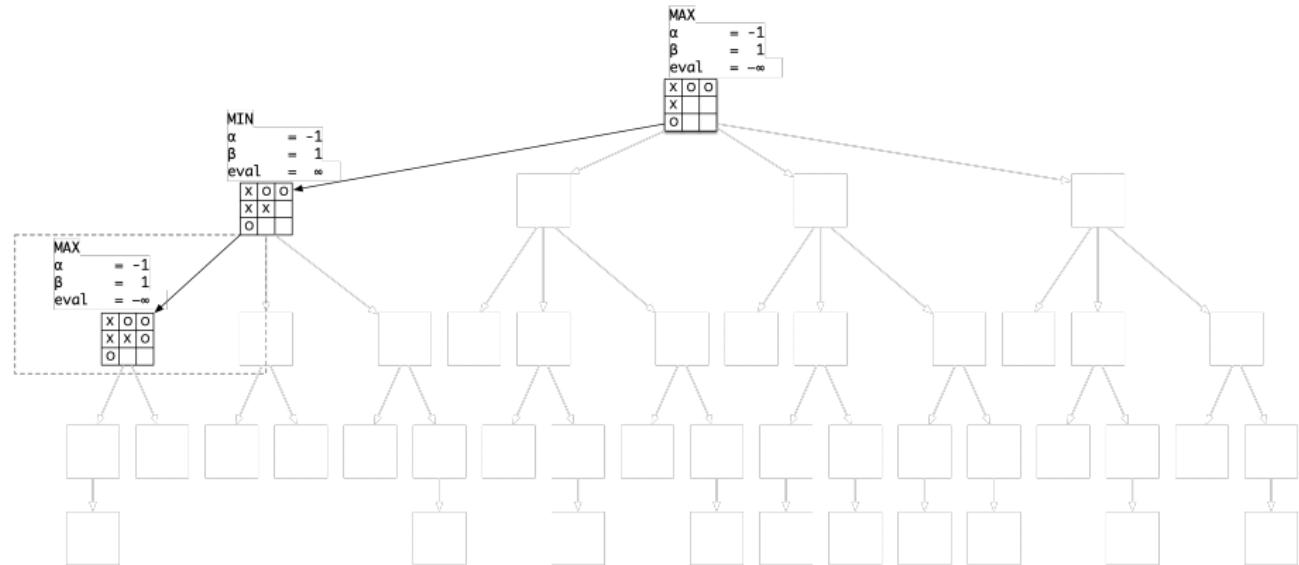
# ESEMPIO: ALGORITMO ALPHABETA



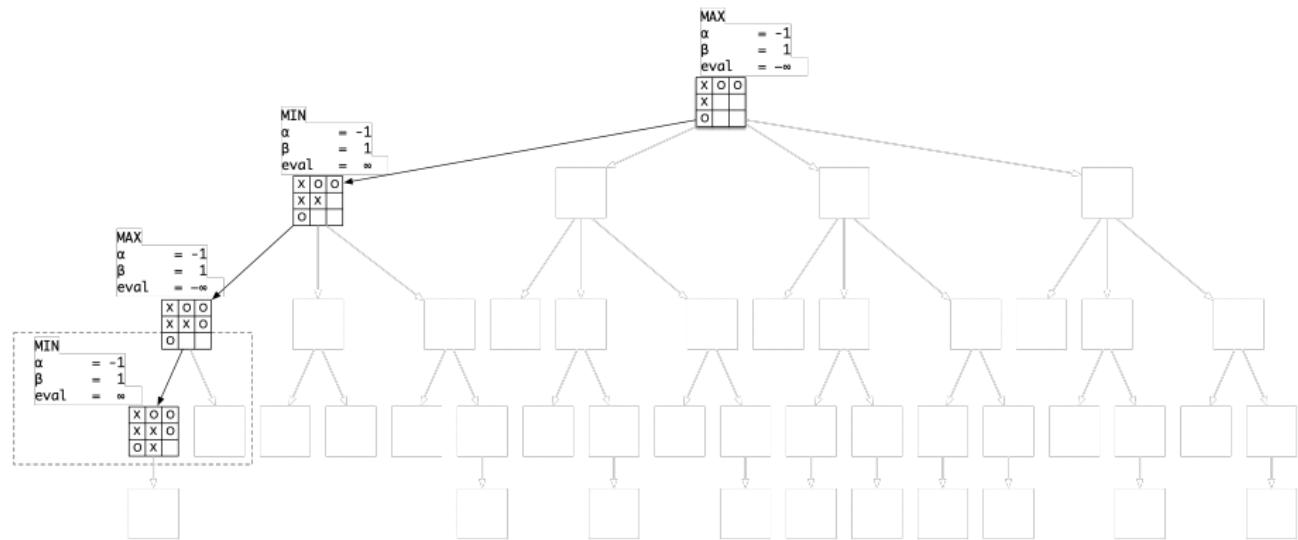
# ESEMPIO: ALGORITMO ALPHABETA



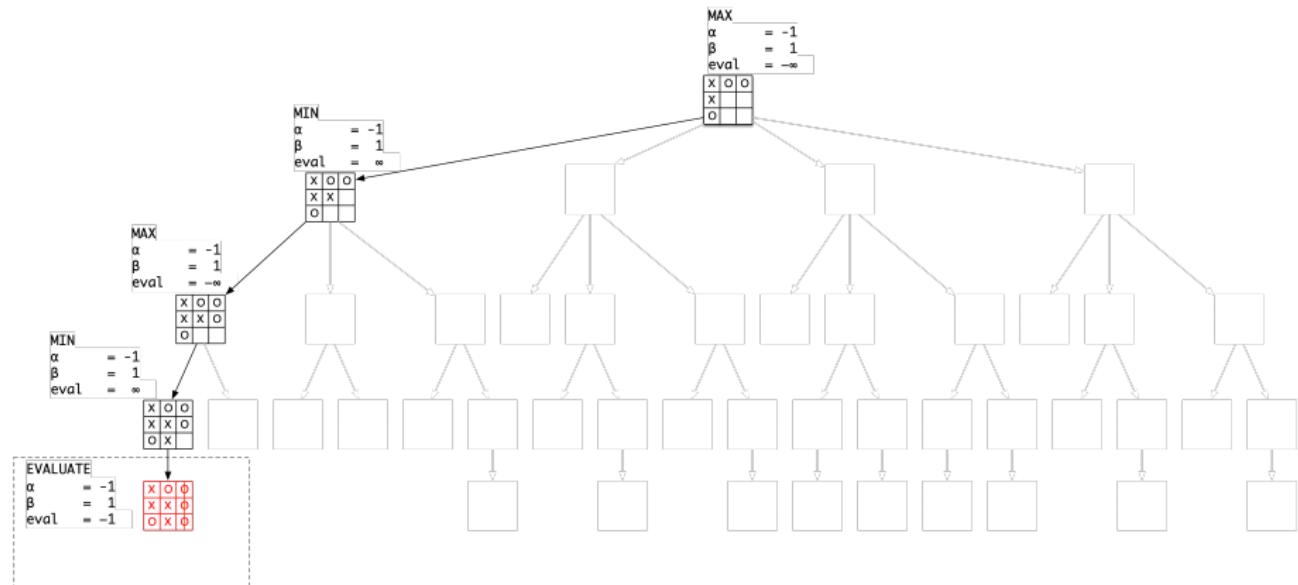
# ESEMPIO: ALGORITMO ALPHABETA



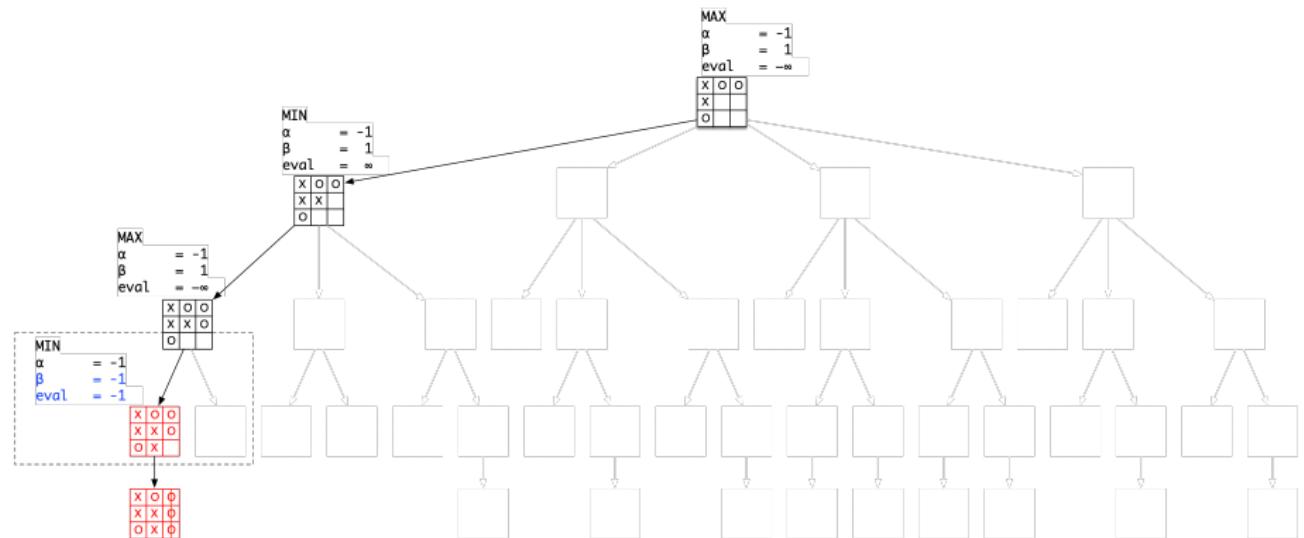
# ESEMPIO: ALGORITMO ALPHABETA



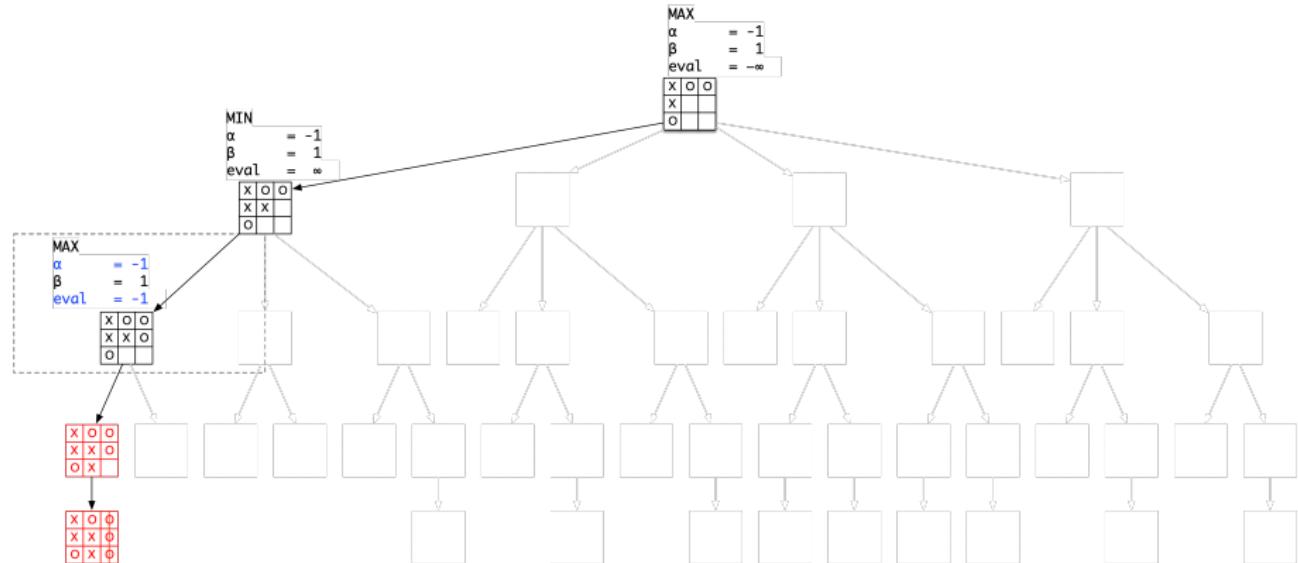
# ESEMPIO: ALGORITMO ALPHABETA



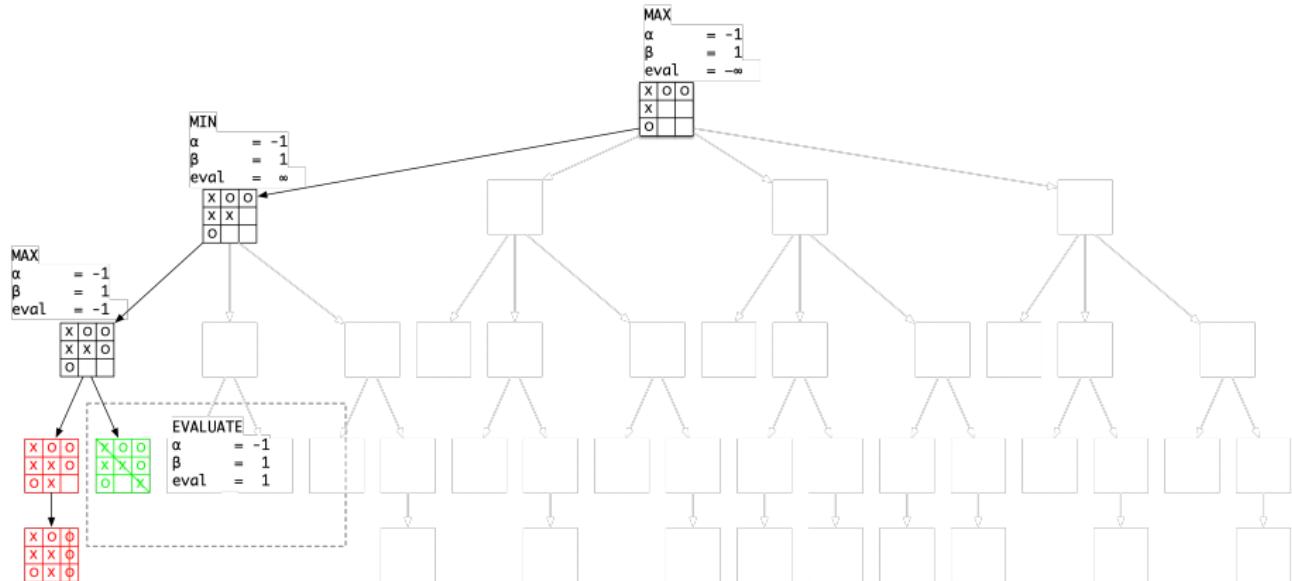
### ESEMPIO: ALGORITMO ALPHABETA



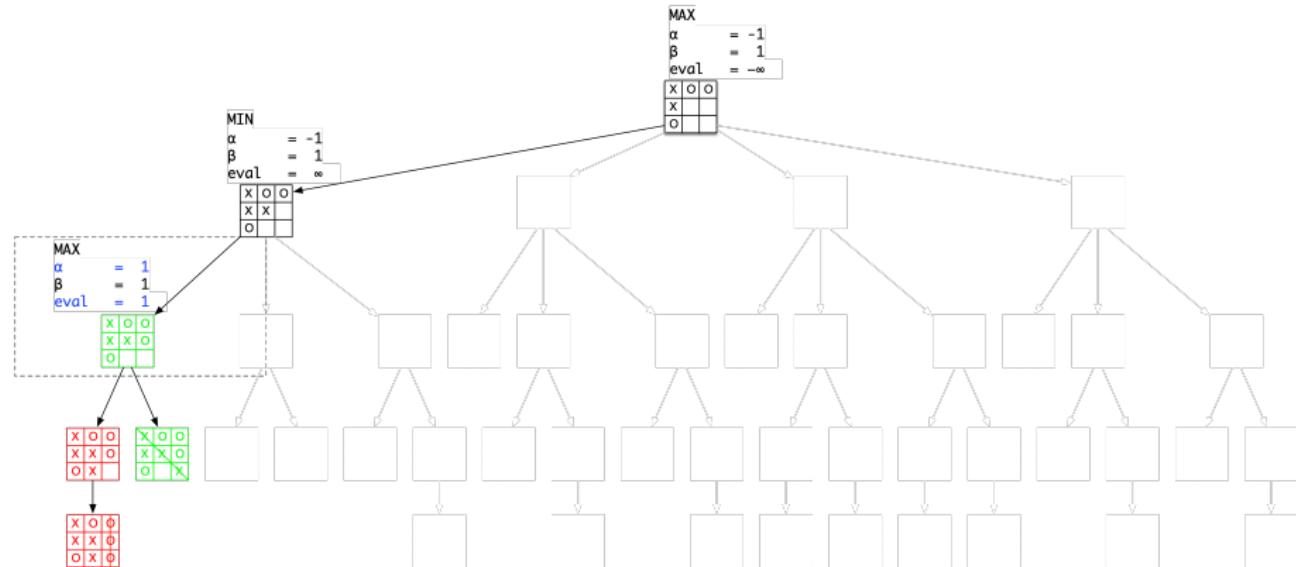
# ESEMPIO: ALGORITMO ALPHABETA



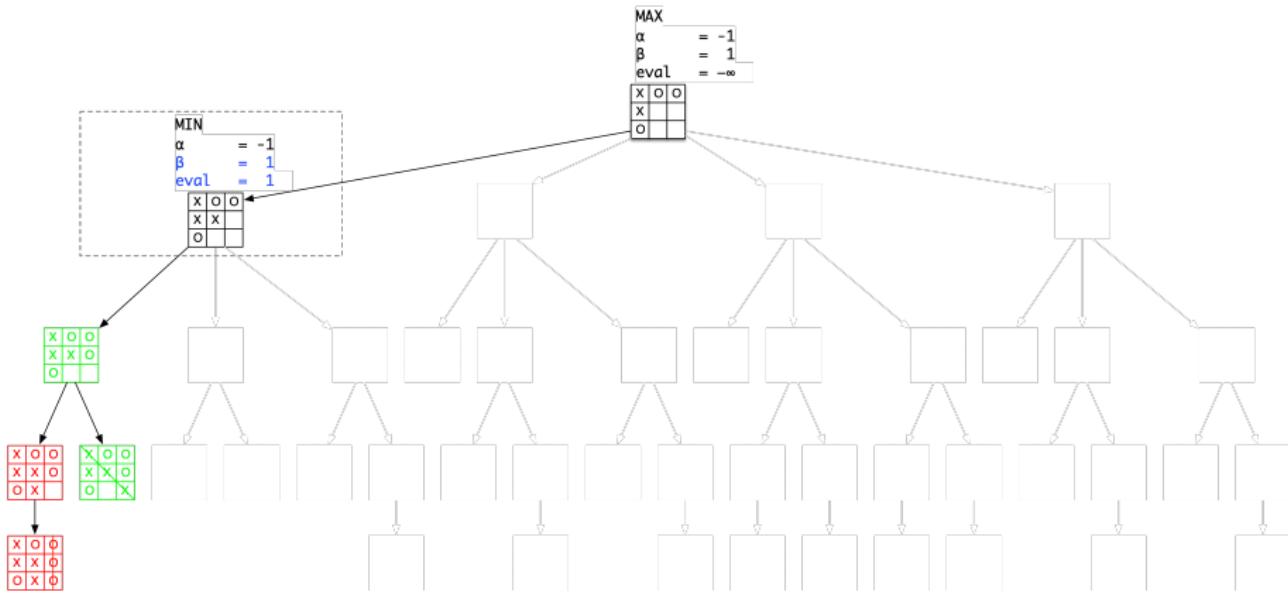
# ESEMPIO: ALGORITMO ALPHABETA



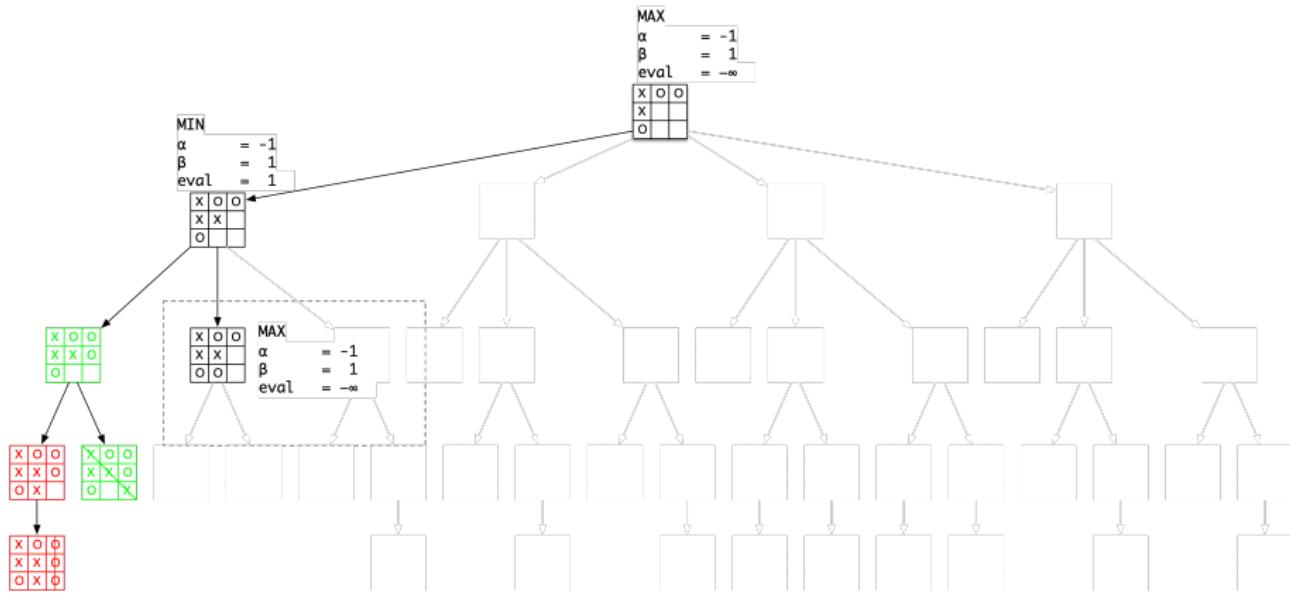
### ESEMPIO: ALGORITMO ALPHABETA



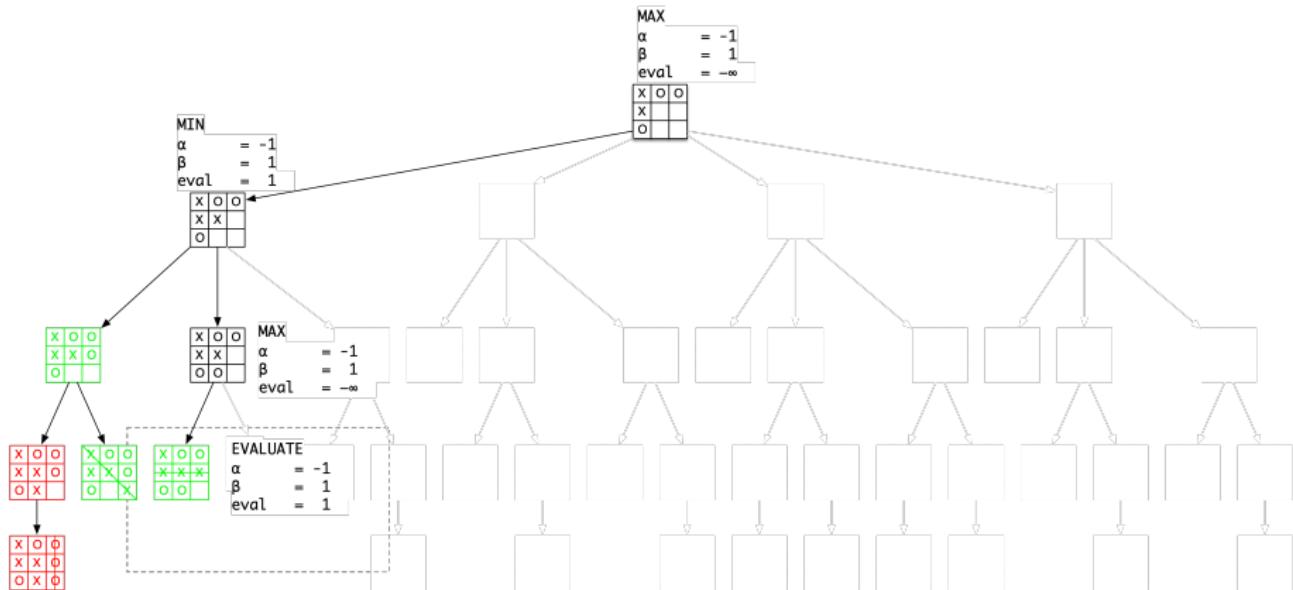
# ESEMPIO: ALGORITMO ALPHABETA



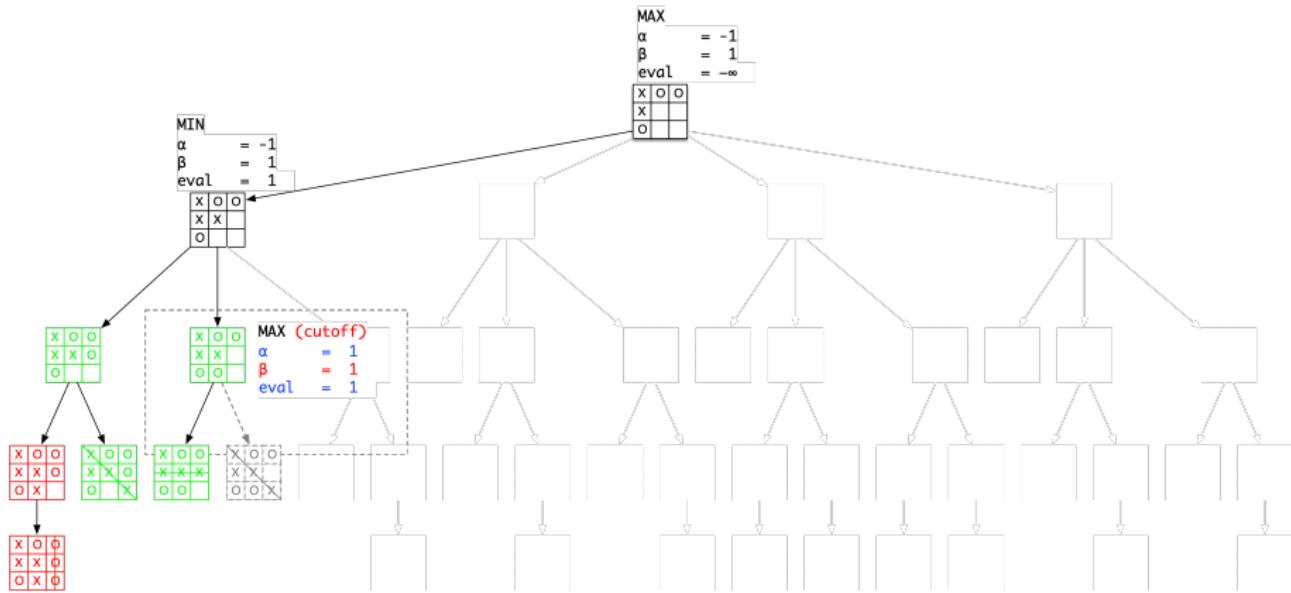
# ESEMPIO: ALGORITMO ALPHABETA



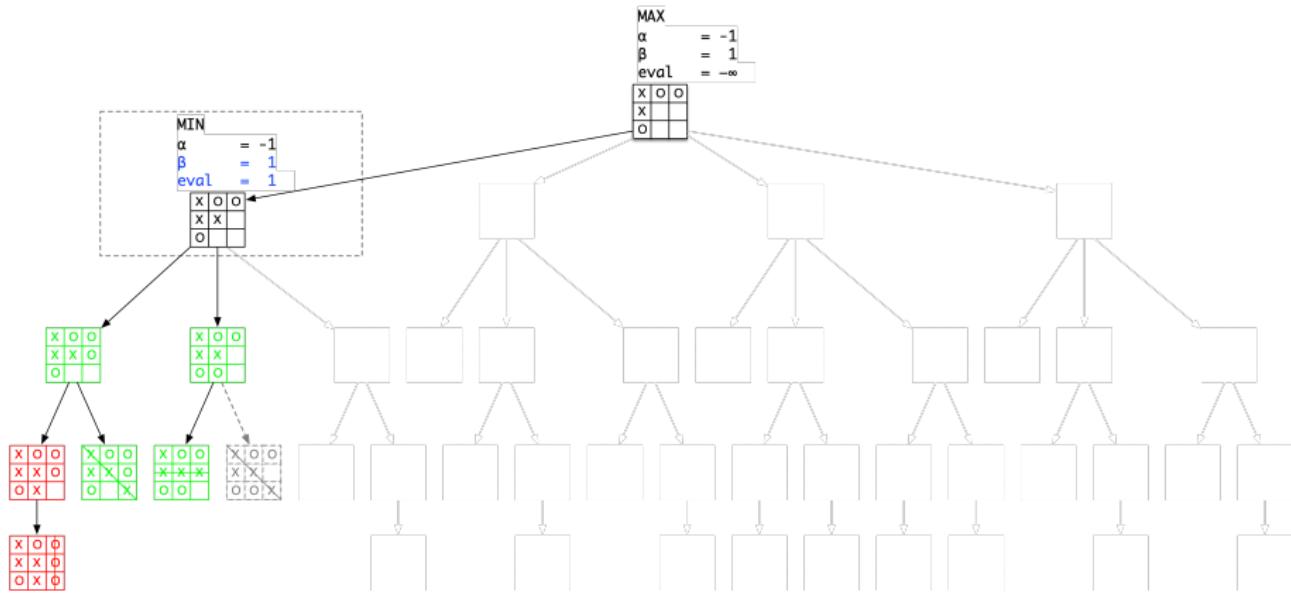
# ESEMPIO: ALGORITMO ALPHABETA



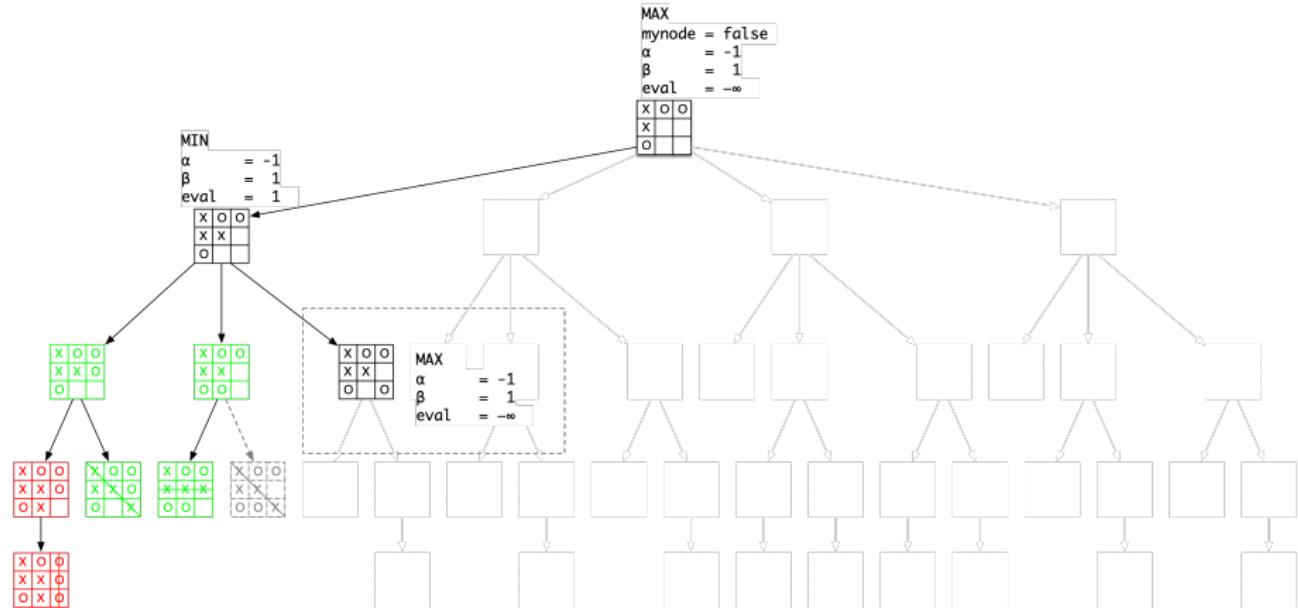
### ESEMPIO: ALGORITMO ALPHABETA



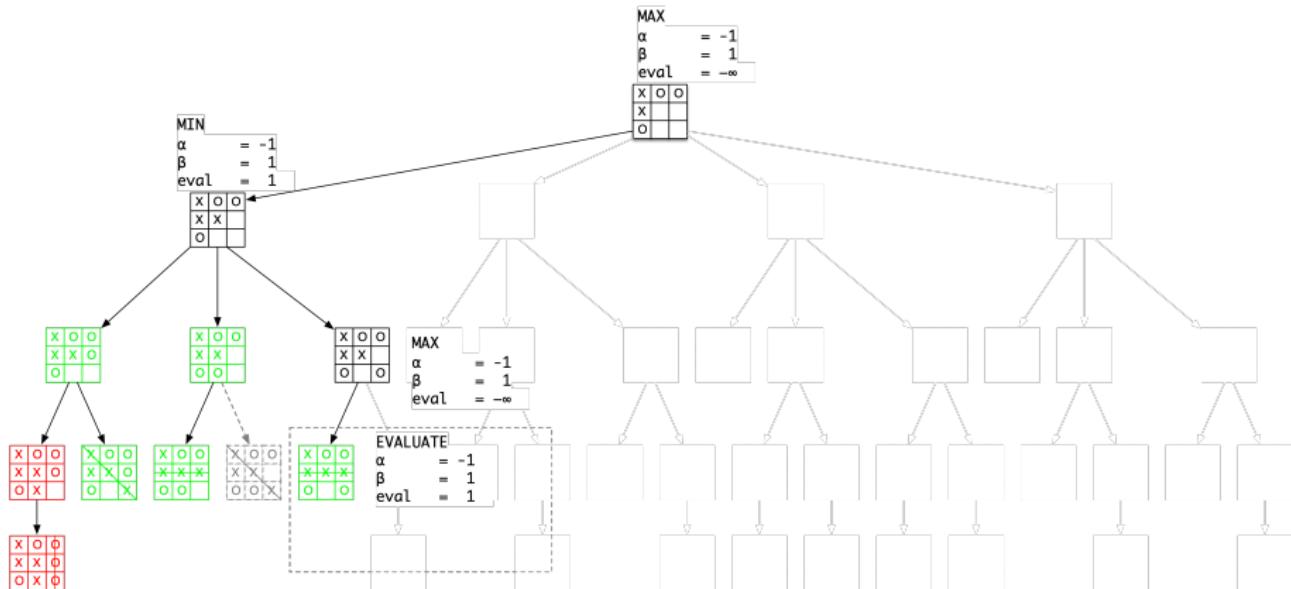
# ESEMPIO: ALGORITMO ALPHABETA



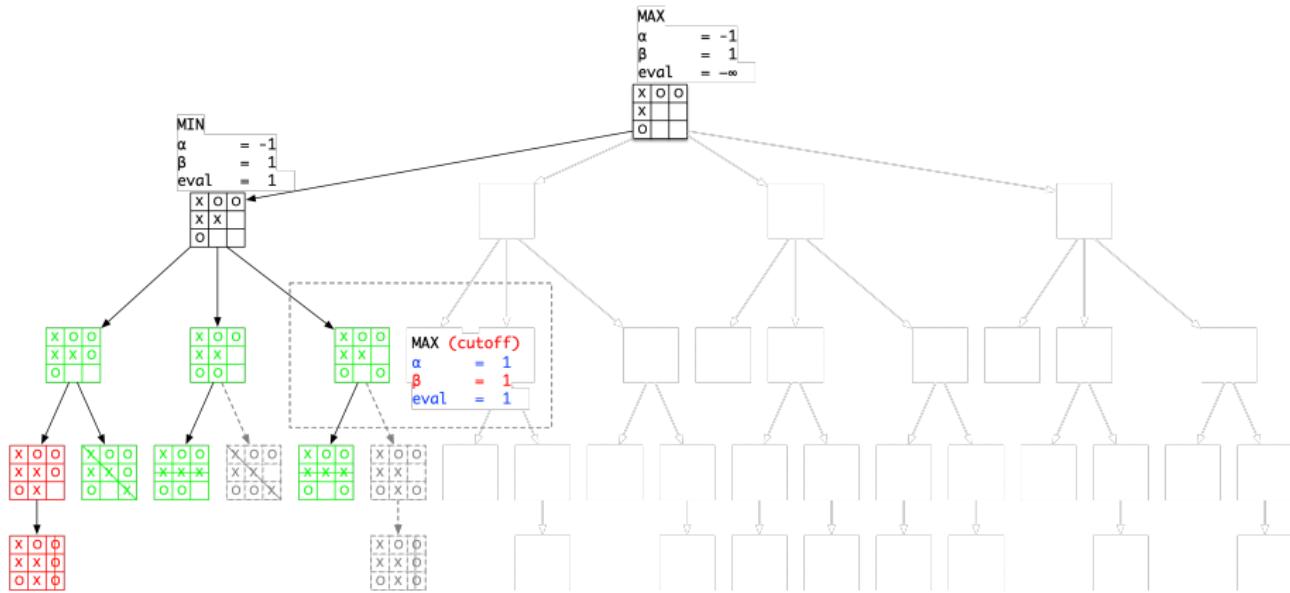
### ESEMPIO: ALGORITMO ALPHABETA



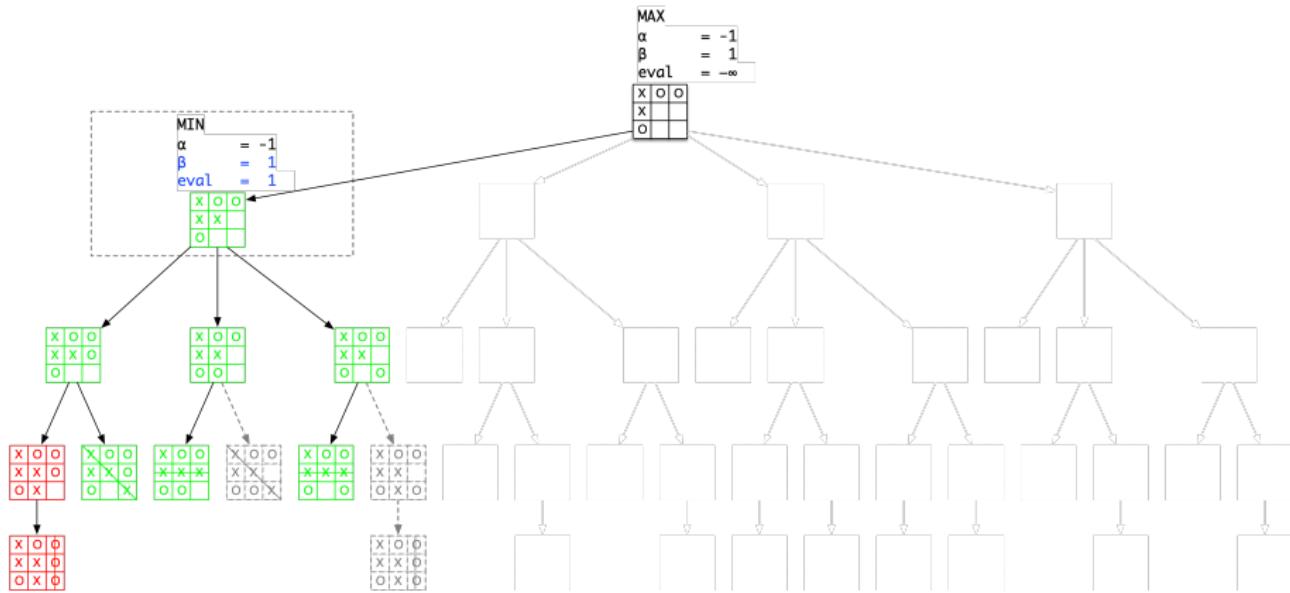
# ESEMPIO: ALGORITMO ALPHABETA



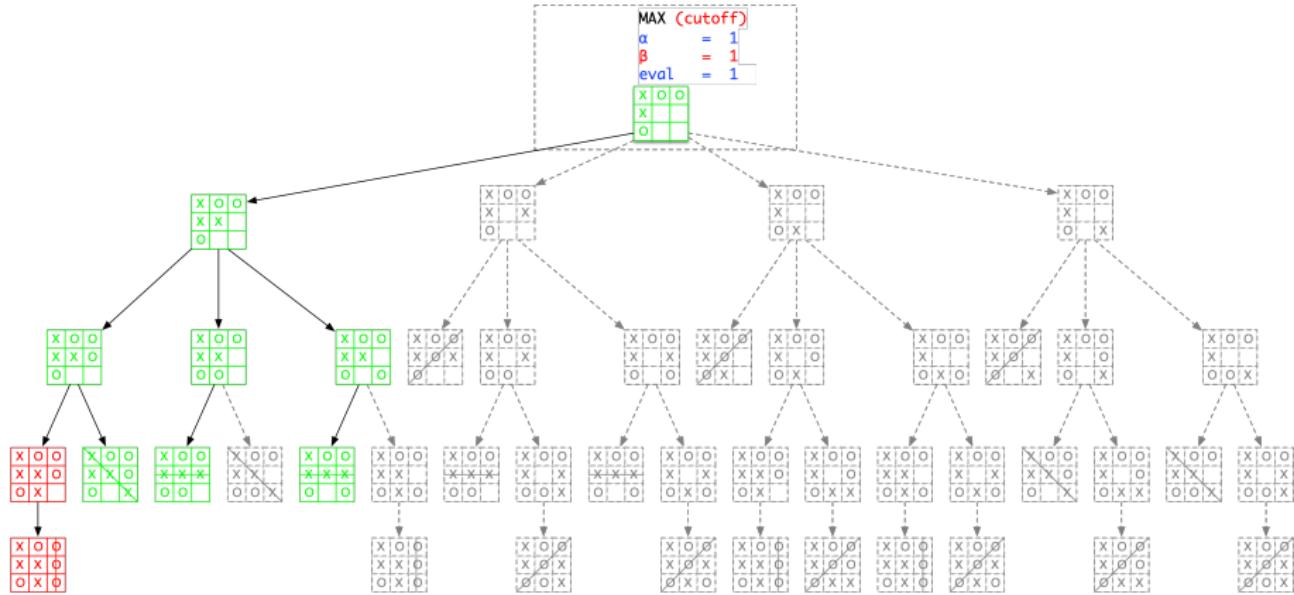
# ESEMPIO: ALGORITMO ALPHABETA



# ESEMPIO: ALGORITMO ALPHABETA



# ESEMPIO: ALGORITMO ALPHABETA

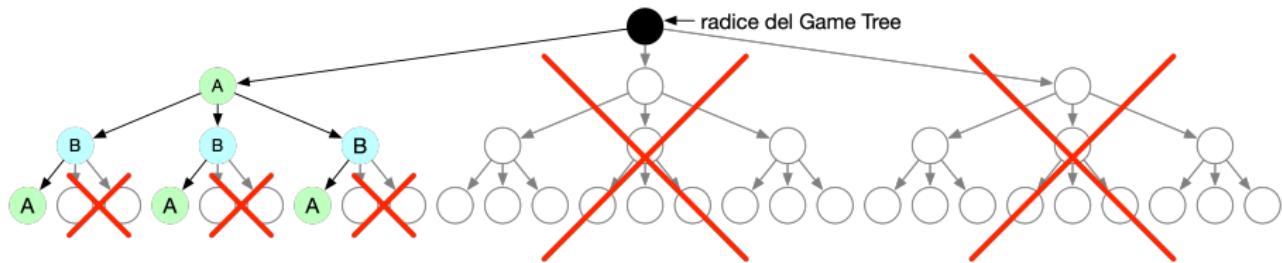


# ANALISI DELL'ALGORITMO ALPHABETA

- Quanto costa l'algoritmo ALPHABETA?
  - Tempo:  $O(n)$  (nel caso pessimo visita ogni nodo dell'albero)
  - Memoria:  $O(h)$  ( $h =$  altezza dell'albero)
  - E' davvero conveniente rispetto a MINIMAX?
- Proviamo ad analizzare quando funziona la potatura  $\alpha/\beta$ 
  - Se durante il turno del giocatore A valutiamo come **prima mossa** quella che porta A alla vittoria, allora ALPHABETA evita di valutare (*potatura*) tutti i sotto-alberi relativi alle ulteriori mosse a disposizione di A nel suo turno.
  - D'altra parte, se una mossa scelta da A porta alla vittoria allora questo implica il dover verificare **tutte** le mosse successive per il giocatore B (cerchiamo una contromossa per B senza successo)
  - Lo stesso vale simmetricamente per il giocatore B
- Riusciamo a quantificare il vantaggio offerto da ALPHABETA?

# ANALISI DI ALPHABETA: $m$ MOSSE, $n$ TURNI

- Caso pessimo:  $O(m^n)$  (visitiamo tutto l'albero)
- Caso ottimo: inizia il giocatore  $A$  e ad ogni turno di  $A$  scegliamo sempre come prima mossa da valutare quella che porta  $A$  alla vittoria (N.B. non è l'unica situazione ottima, potrebbe iniziare  $B$ ).



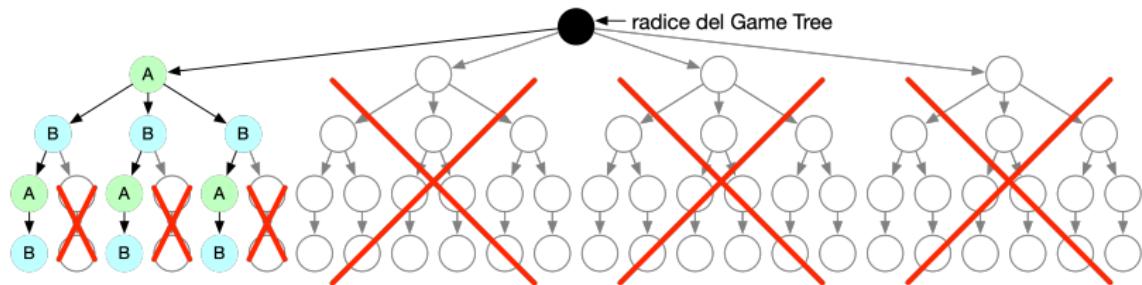
Il numero di nodi  $T(m, n)$  è limitato da (per comodità  $n$  dispari):

$$\begin{aligned} T(m, n) &\leq 1 + 1 + m + m + m^2 + m^2 + \cdots + m^{\lfloor n/2 \rfloor} + m^{\lfloor n/2 \rfloor} \\ &= \sum_{k=0}^{\lfloor n/2 \rfloor} 2m^k = 2 \frac{m^{\lfloor n/2 \rfloor + 1} - 1}{m - 1} = O\left(m^{n/2}\right) = O\left(\sqrt{m^n}\right) \end{aligned}$$

Conclusione: speed-up quadratico nel caso ottimo

# ANALISI DI ALPHABETA: GIOCO CON $n$ OGGETTI 1/2

- Caso pessimo:  $O(n!)$  (visitiamo tutto l'albero)
- Caso ottimo: inizia il giocatore  $A$  e ad ogni turno di  $A$  scegliamo sempre come prima mossa da valutare quella che porta  $A$  alla vittoria (N.B. non è l'unica situazione ottima, potrebbe iniziare  $B$ ).



Il numero di nodi  $T(n)$  è limitato da (per comodità  $n$  dispari):

$$\begin{aligned} T(n) &\leq 1 + 1 + n - 1 + n - 1 + (n - 1)(n - 3) + (n - 1)(n - 3) + \dots \\ &\quad \dots (n - 1)(n - 3)(n - 5) + (n - 1)(n - 3)(n - 5) + \dots + 2 + 2 \\ &= \sum_{k=0}^{(n-3)/2} 2 \frac{(n-1)!!}{(n-1-2k)!!} \leq 2c(n-1)!! = O((n-1)!!) \quad (c \approx 3.06) \end{aligned}$$

## ANALISI DI ALPHABETA: GIOCO CON $n$ OGGETTI 2/2

- Doppio fattoriale  $n!! = \begin{cases} n \cdot (n-2) \cdot (n-4) \cdots 4 \cdot 2 & n \text{ pari} \\ n \cdot (n-2) \cdot (n-4) \cdots 3 \cdot 1 & n \text{ dispari} \end{cases}$
- Lower bound:  $n!! \geq 2^{n/2}, \forall n \geq 2$  (per dim. convergenza  $c = \sum_{n=0}^{\infty} \frac{1}{n!!}$ )
- Se  $n$  dispari

$$\begin{aligned}(n-1)!! &= (n-1) \cdot (n-3) \cdots 2 \\&= \sqrt{(n-1)^2 \cdot (n-3)^2 \cdots 2^2} \\&\leq \sqrt{n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdots 3 \cdot 2} \\&= \sqrt{n!}\end{aligned}$$

- Abbiamo quindi che  $T(n) = O(\sqrt{n!})$
- Conclusione: **speed-up quadratico** nel caso ottimo
- Nota: se inizia B riusciamo a stimare  $T(n) = O(\sqrt{(n+1)!})$

# ALPHABETA vs MINIMAX

- ALPHABETA offre uno **speed-up quadratico** rispetto a MINIMAX se riusciamo a trovare un **ordine ottimo** di valutazione delle mosse
  - **Pro:** a parità di numero di calcoli, con ALPHABETA possiamo visitare nell'albero una **profondità doppia** rispetto a MINIMAX
  - **Con:** possibile solo avendo una **strategia ottima**, che rende inutile ALPHABETA (basta usare direttamente la strategia ottima)
- Nel caso medio, ALPHABETA visita comunque meno nodi di MINIMAX
  - Il costo nel caso medio è difficile da analizzare
  - Un ordinamento **euristico** di valutazione delle mosse, dalla più promettente alla meno promettente, può migliorare il caso medio
  - Euristiche *leggere* per non sprecare il tempo risparmiato potando
    - Esempio: siamo davvero sicuri che tutte le mosse meritino di essere valutate?

# RICERCA LIMITATA IN PROFONDITÀ

- Anche nel caso ottimo, il Game Tree associato ad una configurazione di gioco potrebbe essere troppo grande per una visita completa in tempi ragionevoli con ALPHABETA
- Come gestiamo Game Tree troppo grandi?
  - Limitiamo la ricerca ad un livello massimo di profondità
  - Funzione euristica per valutare le configurazioni non-finali
  - Esempio: assumendo  $-X$  (sconfitta), 0 (patta),  $X$  (vittoria)
    - $0 < eval \leq X$  indica una configurazione favorevole
    - $-X \leq eval < 0$  indica una configurazione sfavorevole
    - $eval = 0$  indica totale incertezza o patta
- La profondità di può essere usata anche per *migliorare* la valutazione
  - Tra due mosse che portano alla vittoria, scegliamo quella che porta alla vittoria più velocemente (acceleriamo la vittoria)
  - Tra due mosse che portano alla sconfitta, scegliamo quella che porta alla sconfitta più lentamente (ritardiamo la sconfitta)

# ALGORITMO ALPHABETA CON PROFONDITÀ-LIMITE

```
1: function ALPHABETA(TREE  $T$ , BOOL  $playerA$ , INT  $\alpha$ , INT  $\beta$ , INT  $depth$ )  $\rightarrow$  INT
2:   if  $depth == 0$  or ISLEAF( $T$ ) then
3:     eval = EVALUATE( $T, depth$ )
4:   else if  $playerA == \text{true}$  then                                 $\triangleright MAX\ player$ 
5:     eval =  $-\infty$ 
6:     for  $c \in CHILDREN(T)$  do
7:       eval = MAX(eval, ALPHABETA( $c, \text{true}, \alpha, \beta, depth - 1$ ))
8:        $\alpha = \text{MAX}(eval, \alpha)$ 
9:       if  $\beta \leq \alpha$  then                                          $\triangleright \beta\ cutoff$ 
10:        break
11:   else                                                  $\triangleright MIN\ player$ 
12:     eval =  $\infty$ 
13:     for  $c \in CHILDREN(T)$  do
14:       eval = MIN(eval, ALPHABETA( $c, \text{false}, \alpha, \beta, depth - 1$ ))
15:        $\beta = \text{MIN}(eval, \beta)$ 
16:       if  $\beta \leq \alpha$  then                                          $\triangleright \alpha\ cutoff$ 
17:         break
18:      $T.\text{label} = eval$ 
19:   return eval
```

## VISITA CON LIMITI DI TEMPO

- Assumiamo di avere dei limiti di tempo per la scelta di una mossa
  - Problema: stimare il livello massimo di profondità **visitabile completamente** entro il limite di tempo
  - In generale, tale stima potrebbe essere molto complessa
- ALPHABETA implementa una visita in profondità
  - Una stima troppo ottimistica del livello massimo di profondità comporta una visita completa solo per poche mosse
  - Una stima troppo pessimistica comporta una valutazione poco accurata delle varie mosse a disposizione
- Soluzione: una **visita in ampiezza** può essere interrotta in qualsiasi momento senza dover fissare a priori un livello massimo

# ALGORITMO ITERATIVEDEEPENING

```
1: function ITERATIVEDEEPENING(TREE  $T$ , BOOL  $playerA$ , INT  $depth$ )  $\rightarrow$  INT
2:    $\alpha = \text{MINALPHA}$ 
3:    $\beta = \text{MAXBETA}$ 
4:   for  $d = 0, \dots, depth$  do
5:      $eval = \text{ALPHABETA}(T, playerA, \alpha, \beta, d)$ 
6:   return  $eval$ 
```

- Visita in ampiezza fino a profondità  $d$  del sottoalbero radicato in  $T$
- Facile da modificare rispetto ad un limite di tempo massimo
  - Non è necessario calcolare una profondità massima  $d$
  - Visitiamo il livello più profondo raggiungibile nel tempo limite
  - Se il timemout sta per scadere mentre valutiamo profondità  $d$ , scegliamo la mossa migliore individuata a profondità  $d - 1$
- Attenzione: per valutare quale scegliere tra  $m$  mosse distinte in un tempo limite la ricerca in ampiezza deve essere portata avanti **contemporaneamente** su **tutti** i sottoalberi relativi alle  $m$  mosse

# ANALISI DI ITERATIVEDEEPENING: MEMORIA

- ITERATIVEDEEPENING sembra non offrire grossi vantaggi
  - Miglioriamo la gestione del tempo ma ..
  - .. rivisitiamo continuamente gli stessi nodi (riparte dalla radice)
- Quanto costa ITERATIVEDEEPENING in termini di memoria?
  - Dipende dal costo in memoria di ALPHABETA
  - **Costo pessimo:**  $O(d)$  ( $d$  = profondità di ricerca)
  - Costo lineare rispetto alla profondità  $d$
- In confronto, BFS ha un costo  $O(k)$  ( $k$  = numero di nodi a livello  $d$ )
  - La struttura dati Coda al livello  $d - 1$  viene riempita con riferimenti a tutti i nodi del livello  $d$
  - Gioco con  $m$  mosse:  $k = m^d$  (crescita esponenziale rispetto a  $d$ )
  - Gioco con  $n$  posizioni:  $k = n!/(n - d)! \leq n^d$  (esponenziale in  $d$ )
- Conclusione: in termini di utilizzo di memoria BFS ha un costo **esponenziale** mentre ITERATIVEDEEPENING ha un costo **lineare**

# ANALISI DI ITERATIVEDEEPENING: TEMPO

- Quanto costa ITERATIVEDEEPENING in termini di tempo?
  - Nel caso pessimo, aumentiamo sensibilmente il costo di una visita fino al  $d$ -esimo livello rispetto al costo di una singola visita con ALPHABETA fino al livello  $d$ -esimo?
  - Costo pessimo di ITERATIVEDEEPENING: gioco con  $m$  mosse e  $n$  turni, limitato a  $d$  turni (analisi più complessa per gioco con  $n$  oggetti)

$$\begin{aligned}T(m, d) &\leq (d+1) + dm + (d-1)m^2 + \cdots + 2m^{d-1} + m^d = \\&= m^d \left( 1 + 2m^{-1} + \cdots + dm^{-d+1} + (d+1)m^{-d} \right) \\&= m^d \sum_{i=0}^d i \frac{1}{m^i} \leq m^d \sum_{i=0}^{\infty} i \frac{1}{m^i} = m^d \frac{1/m}{(1 - 1/m)^2} = O(m^d)\end{aligned}$$

La radice viene visitata  $d + 1$  volte, i suoi figli  $d$  volte, ecc

- Stesso ordine di crescita di una visita ALPHABETA fino al livello  $d$ !
  - Il numero di nodi al livello  $d$  domina l'intero costo

# ITERATIVEDEEPENING VS ALPHABETA E BFS

- A parità di profondità, la ricerca in ampiezza ITERATIVEDEEPENING ha lo stesso costo asintotico della ricerca in profondità ALPHABETA
  - In termini pratici, le costanti moltiplicative nascoste dalla notazione asintotica rendono ITERATIVEDEEPENING leggermente più lento di ALPHABETA
  - Questo svantaggio in termini di tempo è ripagato dalla possibilità di avere un maggiore controllo sulla visita del Game Tree quando la scelta della mossa deve essere effettuata entro certi limiti di tempo
- A parità di profondità, la ricerca in ampiezza ITERATIVEDEEPENING ha lo stesso costo asintotico della ricerca in profondità BFS
  - In termini pratici, ITERATIVEDEEPENING è più lento di BFS
  - Il vantaggio concreto di ITERATIVEDEEPENING rispetto a BFS consiste nel fatto che utilizza un quantitativo di memoria lineare rispetto alla profondità di visita, mentre BFS richiede un quantitativo di memoria che cresce esponenzialmente con la profondità

# NOTA FINALE: GESTIONE RIPETIZIONI

- Molti stati di gioco compaiono più volte nell'albero
- Possiamo ottenere uno stesso stato con differenti sequenze di mosse
- Per velocizzare la ricerca è utile riconoscere configurazioni già valutate

