Giochi a due giocatori

Caratteristiche :

- mosse alternate
- non c'è intervento del caso (no giochi di carte…)
- ogni giocatore ha informazioni complete sullo stato del gioco.

Esempi: scacchi, dama, filetto, ecc.

Si può applicare il metodo di riduzione a sottoproblemi per trovare una strategia vincente mediante la "*dimostrazione*" che il gioco può essere vinto.

Giocatori: Più e Meno

Configurazione iniziale: X^t con $t = \pm$

X^+ : la prossima mossa spetta a Più

X^- : la prossima mossa spetta a Meno

Si vuol provare che Più vince (o patta), ovvero si vuol dimostrare $W(X^t)$

Se tocca al Più muovere, cioè la configurazione è X^+, e se le configurazioni legali successive sono $X_1^{t_1}, X_2^{t_2}, ..., X_n^{t_n}$ allora provare $W(X^+)$ significa provare *uno* dei $W(X_i^{t_i})$ (cioè ramo OR).

Se tocca a Meno muovere e da X^- seguono $Y_1^{s_1}, Y_2^{s_2}, ..., Y_m^{s_m}$, dimostrare $W(X^-)$ comporta dimostrare *tutti* i $W(Y_i^{s_i})$ (ramo AND).

Applicando gli operatori di riduzione si genera *l'albero del gioco*.

Il processo di dimostrazione prosegue finché non si è trovato un albero risolutivo che termina nei problemi primitivi (vale solo per i giochi semplici!)

Esempio: Gioco di Grundy

Regole:

- 2 giocatori, una pila di oggetti (per esempio monete)
- Il primo giocatore divide la pila in 2 parti *diseguali*
- successivamente, a turno, ogni giocatore fa lo stesso con una delle pile restanti
- terminazione: tutte le pile di 1 o 2 oggetti (vince chi arriva a questa configurazione).

Ipotesi: 7 monete e muove Meno.

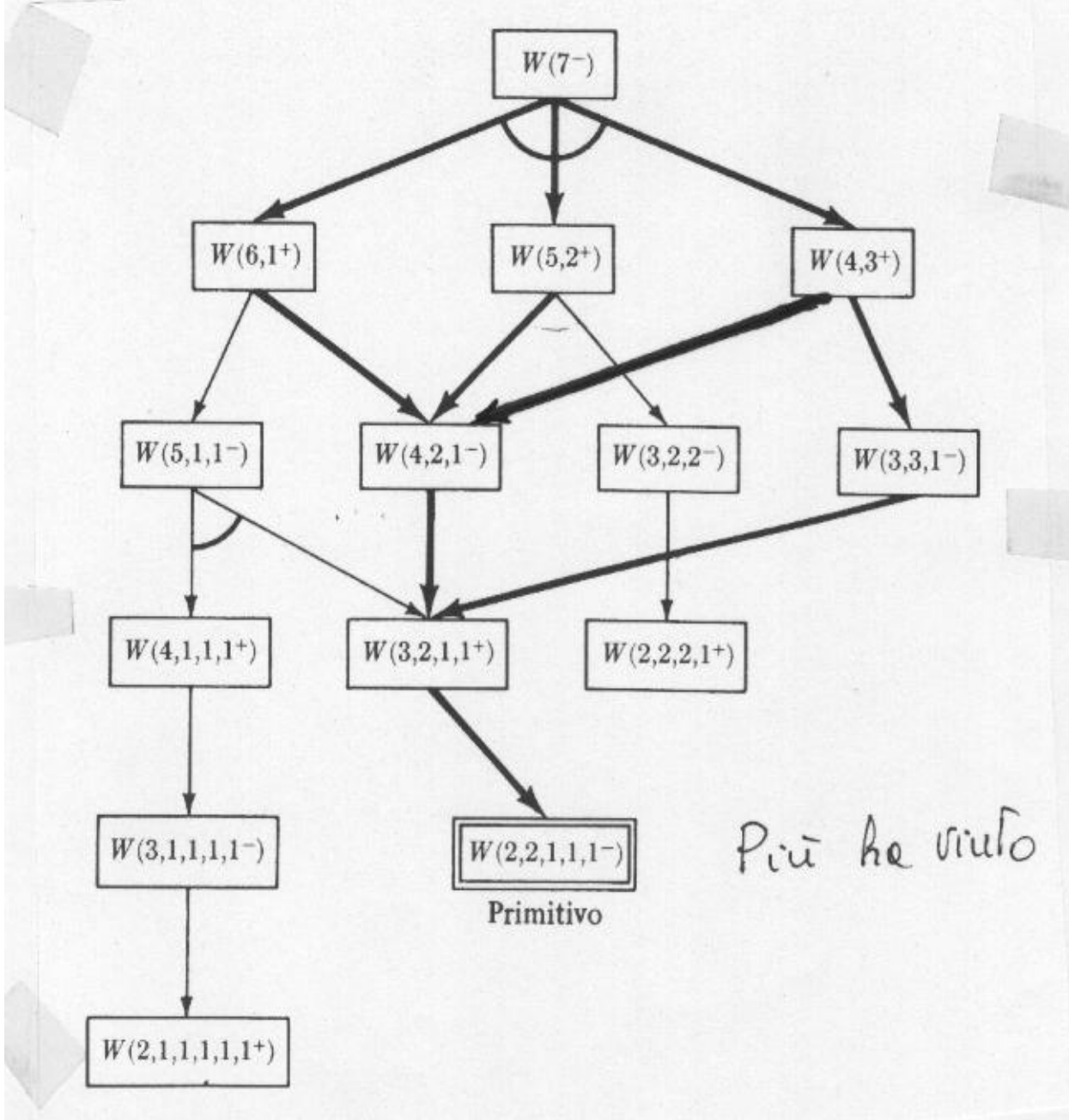
Obiettivo far vincere Più.

Soluzione:

Meno muove da 7^- e può generare $(6,1^+)$ $(5,2^+)$ $(4,3^+)$.

Tocca a Più. L'albero risolutivo è:

Un grafo AND/OR per il gioco di Grundy.



La procedura minimax

Nei giochi complessi (dama, scacchi) non si può dimostrare la vittoria (alberi con 10^{40} e 10^{120} nodi!).

Ci si limita a cercare una "buona" mossa (ricerca limitata in profondità). Schematicamente:

- Guarda se è una mossa vincente. Se sì, la considera la migliore dandole un valore molto alto.

- In caso contrario considera tutte le mosse che l'avversario può fare successivamente. Vede qual è la peggiore per il giocatore (chiamando ricorsivamente questa procedura). Assume che l'avversario farà la mossa che pensa sia la peggiore (e che l'avversario considererà la migliore).

- Qualunque sia il valore della mossa peggiore, lo passa come il valore del caso che sta considerando.

- Il caso migliore è allora quello con valore più alto.

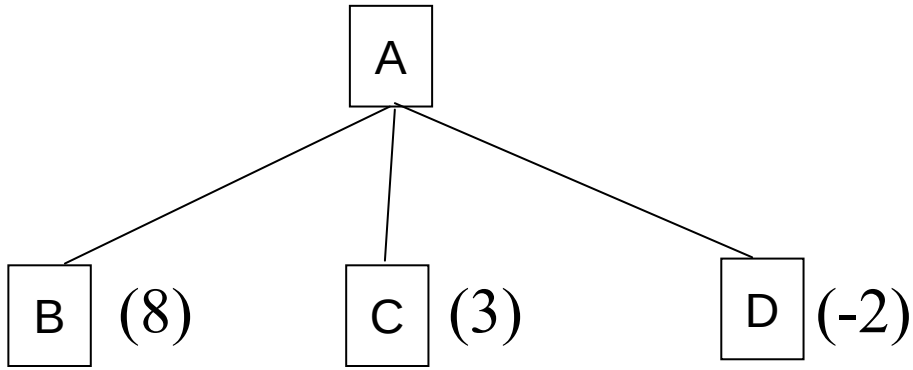
(l'ipotesi è che l'avversario farà la mossa per lui più conveniente!)

Valutazione a 1 stadio:

Si parte dalla posizione attuale, si generano le posizioni successive e si effettua una valutazione di ogni nodo (valutazione statica), si sceglie la migliore. Si riporta questo valore all'indietro (*backed-up value*) e lo si attribuisce al nodo corrente. In questo stadio *si massimizza*.

Esempio:

(Si usano valori di valutazione nell'intervallo -10 ÷ +10)



Si attribuisce ad A un valore della funzione di valutazione statica riportando il valore del nodo B (=8).

N.B: La funzione di valutazione misura il "valore" di una posizione foglia.

Esempio per gli scacchi: vantaggio nel punteggio dei pezzi, controllo del centro, controllo di re, ecc.

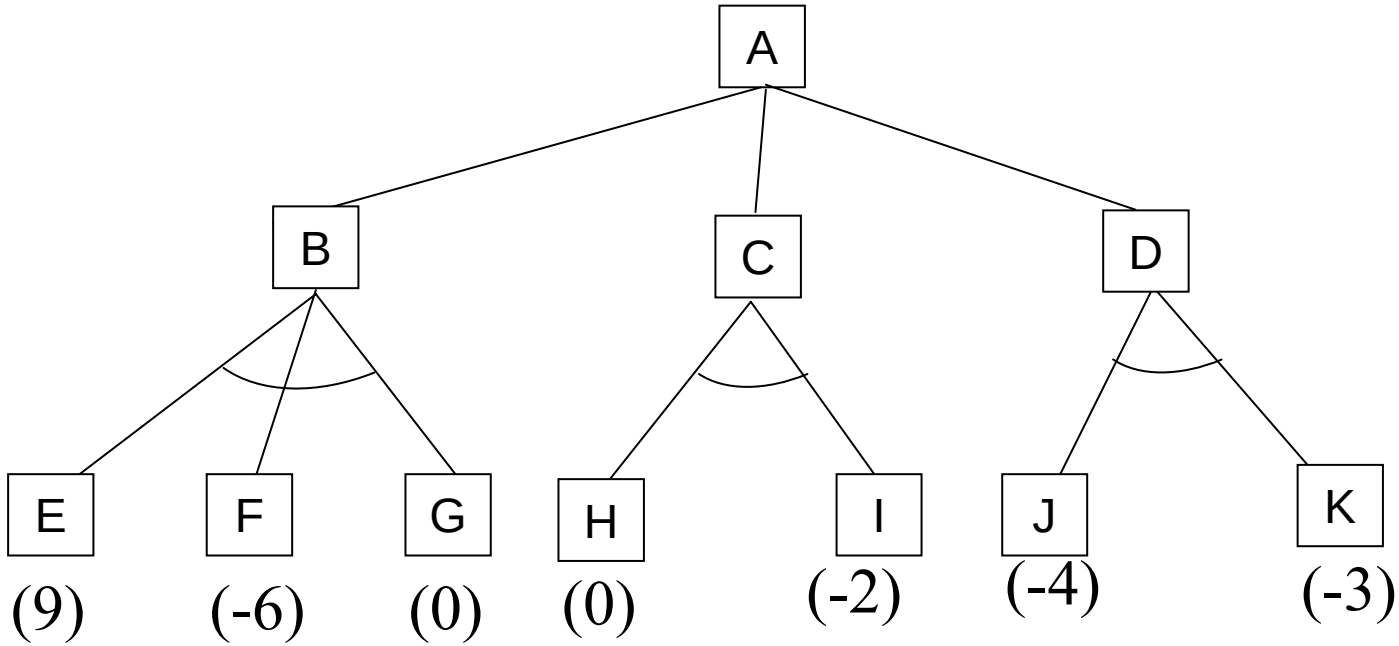
Valori positivi per posizioni favorevoli a Più

Valori negativi per posizioni favorevoli a Meno

Valutazione a 2 stadi:

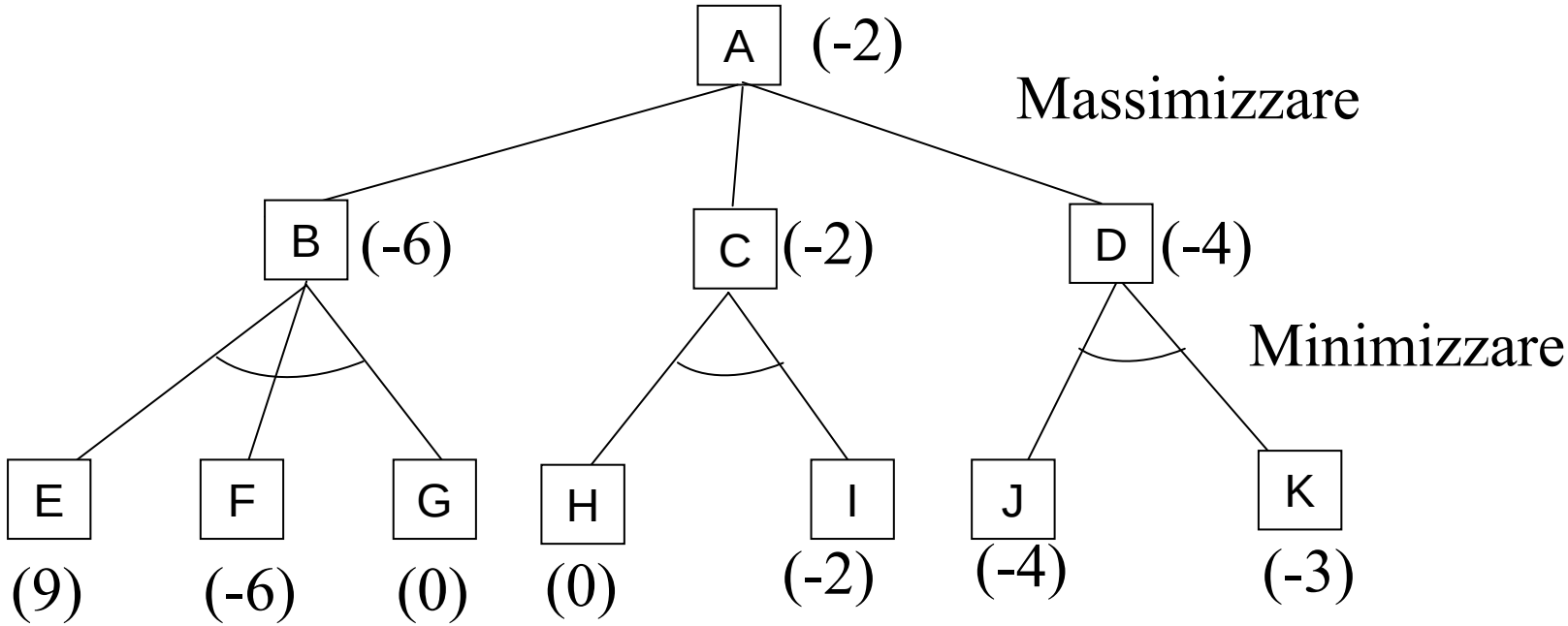
Si valutano anche le possibili mosse dell'avversario. Per ogni nodo si applica il generatore di mosse plausibili.

Le mosse dell'avversario sono da considerare come AND (per ciascun nodo OR).



La meta dell'avversario è di minimizzare la funzione di valutazione (cioè, se Più scegliesse B, Meno sceglierebbe F (=-6)!).

Pertanto occorre propagare all'indietro i valori trovati nell'albero, scegliendo come mossa dell'avversario quella con il valore minimo. La situazione diventa così:



La mossa migliore è ***C***, che comporta un valore della funzione di valutazione pari a -2 (*il massimo dei minimi*, donde il nome di *minimax* dell'algoritmo).

Ovviamente si può proseguire su più stadi, utilizzando la stessa tecnica.

Algoritmo minimax

Procedura ricorsiva che utilizza

GEN MOSSE(Pos): è il generatore di mosse accettabili che fornisce una lista di nodi che rappresentano le mosse che potrebbero essere fatte a partire da Pos.

STATICA(Pos, Profondità): è la funzione di valutazione statica che fornisce un numero che rappresenta la bontà di Pos dal punto di vista corretto. Valori alti indicheranno una buona posizione per la parte che sta per muovere, come indicato dal valore di Profondità pari o dispari. Chi ha chiamato STATICA cercherà di massimizzare il suo punteggio. Quando il valore calcolato viene passato indietro al livello più alto successivo, verrà negato per riflettere la possibilità della situazione dal punto di vista del giocatore avversario.

Invertendo i valori a livelli alternati, la procedura MINI-MAX può essere molto semplice; in effetti fra i valori a disposizione sceglie sempre quello massimo.

Elementi che influenzano la decisione di fermare la ricorsione (e quindi chiamare la funzione di valutazione statica):

- Una delle due parti ha vinto?
- Quanti stadi abbiamo già esplorato?
- Quanto è promettente questo cammino?
- Quanto tempo rimane?
- Quanto è stabile la configurazione?

Tutti questi elementi (e altri) sono valutati dalla funzione booleana

ABBASTANZA_PROFONDO che può assumere i valori:

$$\begin{cases} TRUE \rightarrow smettere \\ FALSE \rightarrow continuare \end{cases}$$

MINI-MAX restituisce:

- Il valore riportato indietro del cammino che sceglie (VALORE)

- Il cammino stesso. Avremo l'intero cammino anche se probabilmente è effettivamente necessario solo il passo più alto (CAMMINO).

Chiamata iniziale:

MINIMAX (ATTUALE,0)

- ATTUALE è il nodo a partire dal quale si vuole la valutazione

- 0 indica la profondità nulla del nodo iniziale

MINIMAX (POSIZIONE, PROFONDITÀ)

1. Se ABBASTANZA-PROFONDO (PROFONDITA), allora fornisce la struttura
   VALORE = STATICA(POSIZIONE, PROFONDITA);
   CAMMINO = nil
   Ciò indica che non vi è un cammino a partire da questo nodo e che il suo valore è quello determinato dalla funzione di valutazione statica.

2. Altrimenti genera un altro stadio dell'albero chiamando GENMOSSE(POSIZIONE) e dando a SUCCESSORI il valore della lista che fornisce.

3. Se SUCCESSORI è vuoto, allora non vi sono mosse da fare, quindi si ha la stessa struttura che si sarebbe avuta se ABBASTANZA-PROFONDO avesse dato vero.

4. Se SUCCESSORI non è vuoto, allora lo attraversa esaminando ogni elemento e tenendo traccia del migliore, nel modo seguente.

5. Inizializza PUNTEGGIO-MIGLIORE col valore minimo che possa dare STATICA; esso verrà aggiornato per riflettere il punteggio ottimo che può essere raggiunto da un elemento di SUCCESSORI.

6. Per ogni elemento di SUCCESSORI (che chiameremo SUCC) si fanno le operazioni seguenti:
   a. Si attribuisce a RISULTATO-SUCC il valore MINIMAX (SUCC, PROFONDITA + 1). Questa chiamata ricorsiva di MINIMAX attuerà effettivamente l'esplorazione di SUCC.
   b. Si attribuisce a NUOVO-VALORE il valore meno VALORE(RISULTATO-SUCC). Ciò farà in modo che rifletta le possibilità della posizione dalla prospettiva opposta a quella del livello più successivo.

c. Se NUOVO-VALORE > PUNTEGGIO-MIGLIORE, allora abbiamo trovato un successore migliore di tutti quelli esaminati fino ad ora e lo si memorizza nel modo seguente:
   i. Si attribuisce a PUNTEGGIO-MIGLIORE il valore NUOVO-VALORE.
   ii. Il cammino noto migliore è ora quello che va da ATTUALE a SUCC e poi lungo il cammino appropriato che parte da SUCC come determinato dalla chiamata ricorsiva di MINIMAX. Quindi si attribuisce a CAMMINO-MIGLIORE il risultato derivante dall'aver appeso SUCC a CAMMINO(RISULTATO-SUCC).

7. Ora che sono stati esaminati tutti i successori, sappiamo qual è il valore di NODO, e quale cammino si deve prendere a partire da esso. Quindi abbiamo la struttura VALORE = PUNTEGGIO-MIGLIORE CAMMINO = CAMMINO-MIGLIORE

Esempio: gioco del filetto

Più → X Meno → O

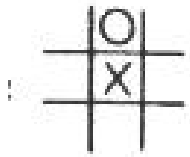
Si usa una funzione di valutazione *e(p)* definita come :

1. se *p* non è una posizione vincente,
   e(p) = (numero delle righe, colonne e diagonali complete ancora aperte a Più) - (numero delle righe, colonne e diagonali complete ancora aperte a Meno);

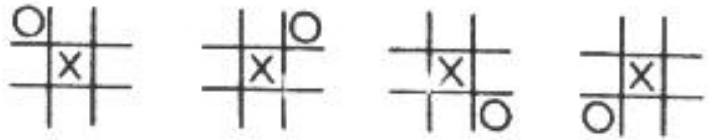
2. se *p* è una posizione vincente per Più, *e(p)* = ∞ (∞ denota un numero positivo molto grande);

3. se *p* è una posizione vincente per Meno, *e(p)* = -∞

Così, se *p* è

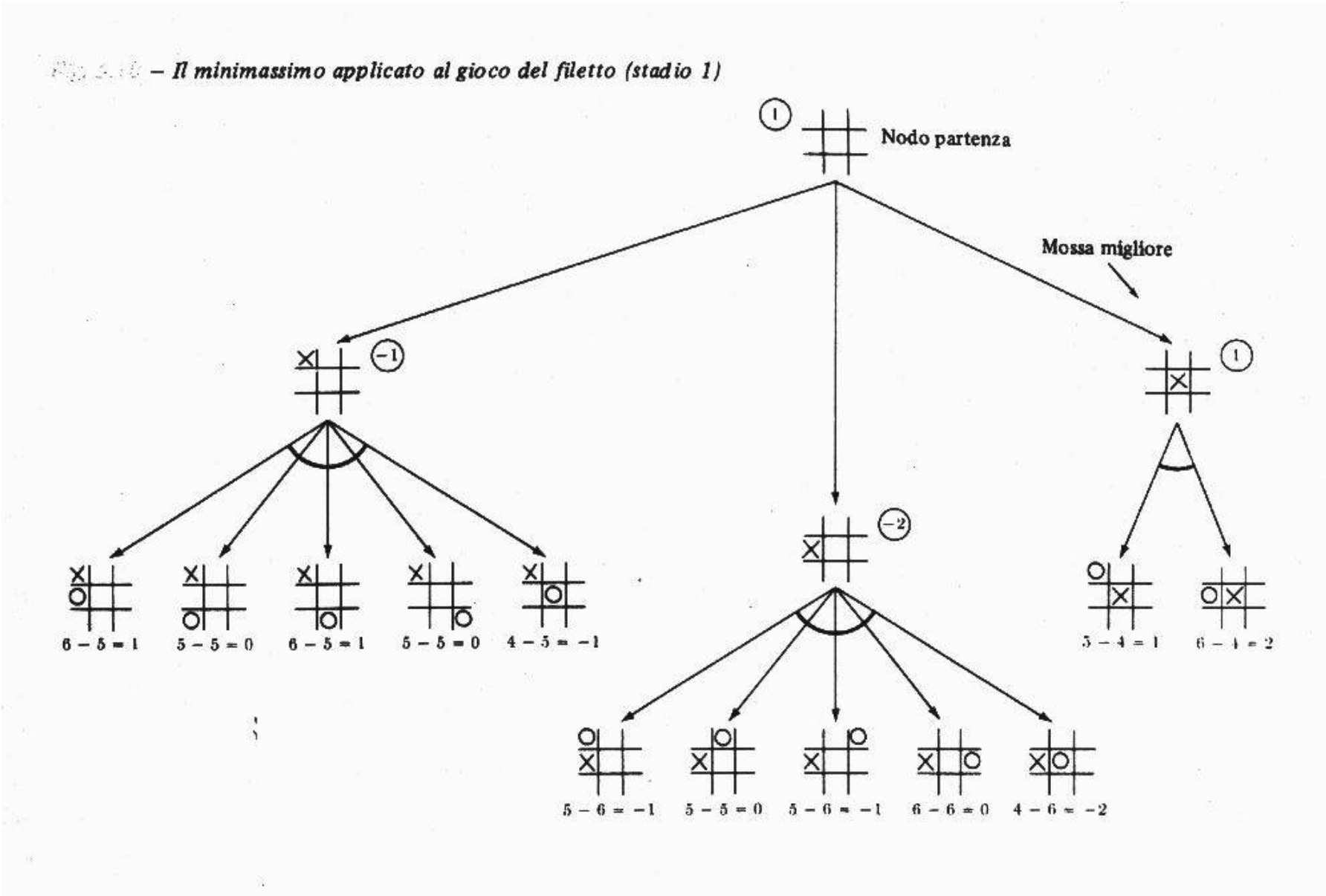


abbiamo *e(p)* = 6 - 4 = 2. Nella generazione delle posizioni sfrutteremo le simmetrie; ovvero,

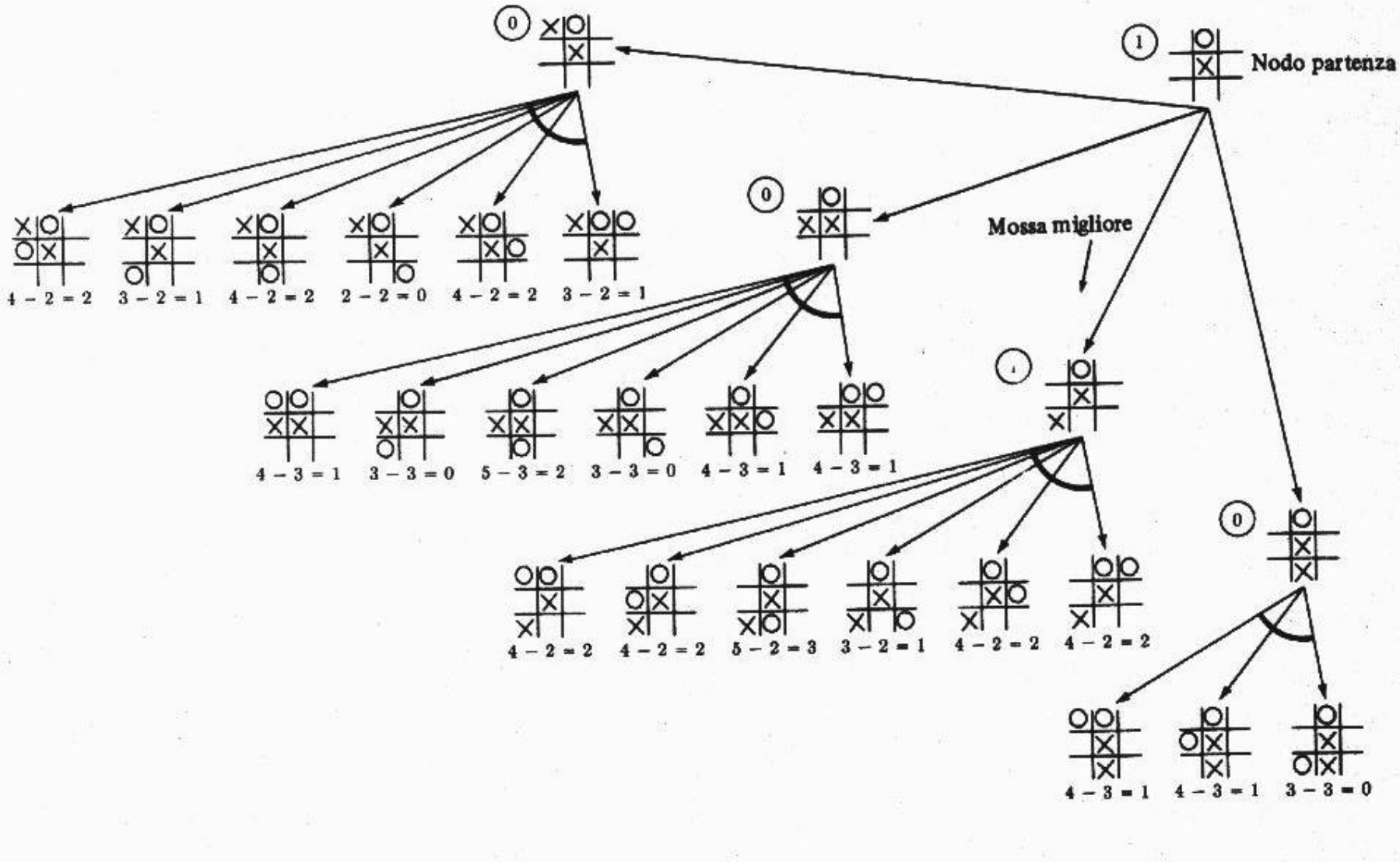


saranno considerate identiche (nella fase iniziale, il fattore di ramificazione del filetto è contenuto dalle simmetrie; nella fase finale, dal piccolo numero di spazi liberi disponibili).

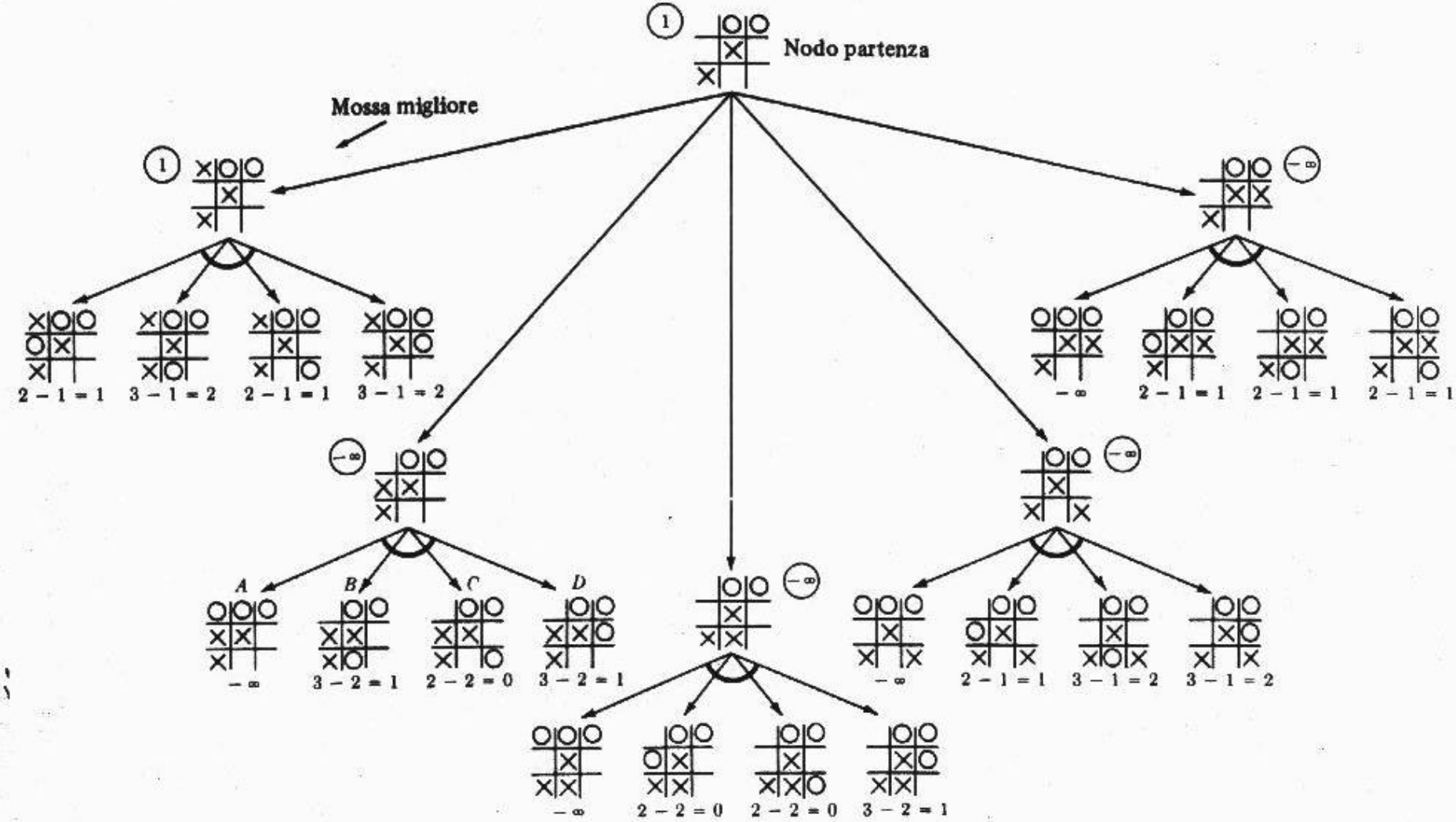
Profondità di ricerca: livello 2



– *Il minimassimo applicato al gioco del filetto (stadio 2)*



– *Il minimassimo applicato al gioco del filetto (stadio 3)*



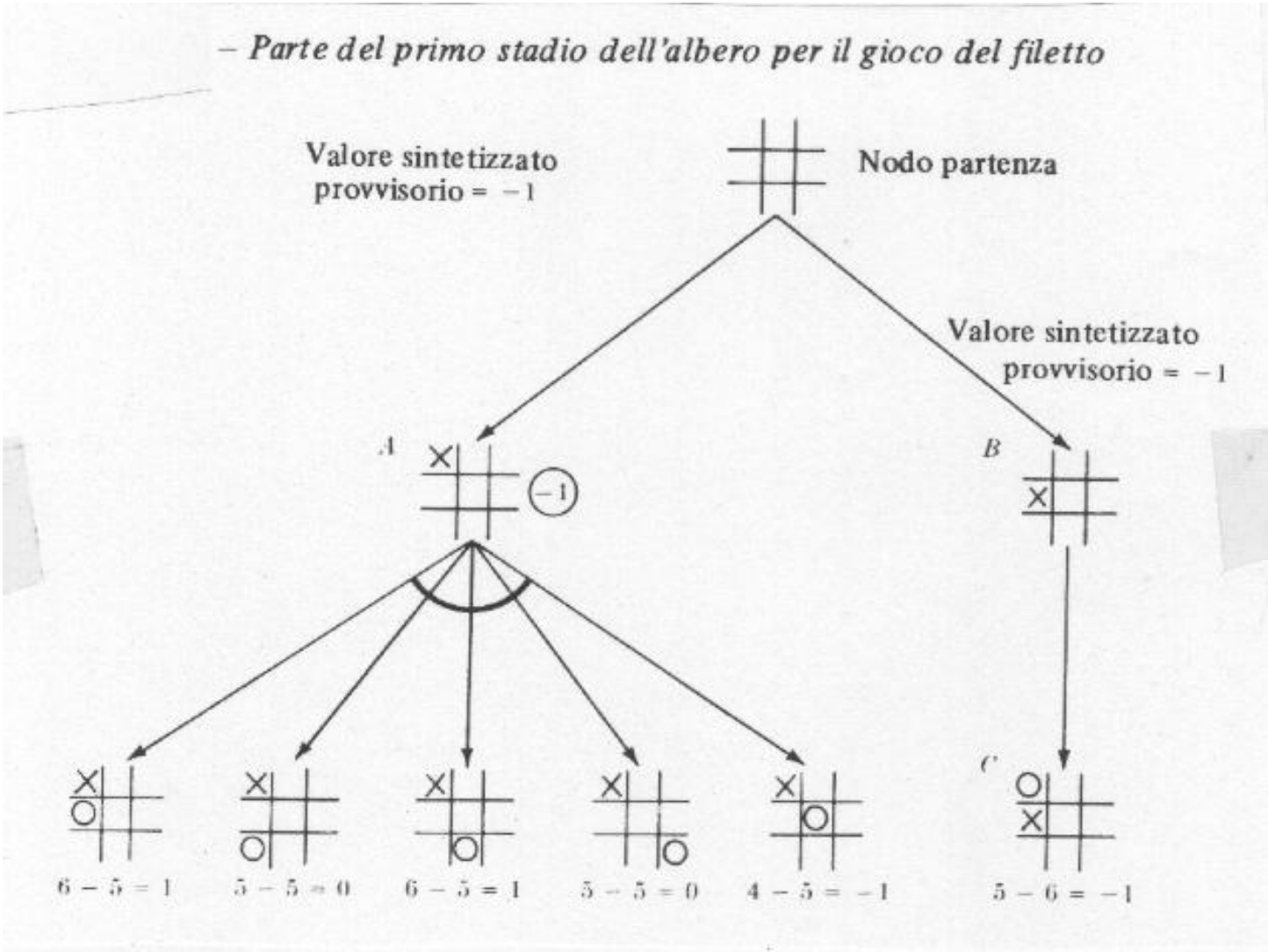
Potatura alfa-beta

Posporre la *valutazione* alla completa *generazione* dell'albero di ricerca è inefficiente: meglio valutare mentre si genera.

Esempio: ultimo stadio ricerca per il filetto (vedi figura precedente).

Se si genera e valuta *A* (= -∞), è inutile generare *B*, *C* e *D* (*A* è la migliore scelta per Meno). Basta riportare all'indietro il valore -∞ di *A*, con notevole risparmio.

Per generalizzare questo procedimento consideriamo un caso come il seguente:



Dopo aver generato *A* e i suoi successori e dopo averli valutati, ad *A* viene dato un valore sintetizzato pari a -1.

Al nodo in esame allora può essere attribuito un *valore sintetizzato provvisorio* (VSP) pari a -1.

Si passa a generare e valutare *B* e *C*. Poiché il valore in *C* è -1, a *B* si attribuisce il valore sintetizzato provvisorio -1.

Ora se si cercassero altri successori, la valutazione di *B* potrebbe *scendere*, *mai salire*.

Pertanto *A* è da preferire a *B*, ed è inutile generare gli altri nodi successori di *B*.

È quindi sufficiente memorizzare i valori sintetizzati provvisori, e aggiornarli man mano che si generano i successori.

Detti nodi MAX quelli sotto i nodi AND (in cui si massimizza) e nodi MIN quelli sotto i nodi OR (in cui si minimizza), si osserva che:

- i VSP (detti valori *alfa*) dei nodi MAX non possono mai diminuire;

- i VSP (detti valori *beta*) dei nodi MIN non possono mai aumentare.

Pertanto le regole per interrompere la ricerca sono:

a) si può tralasciare la ricerca sotto i nodi MIN aventi un valore beta *minore o uguale* al valore alfa di uno dei loro antenati MAX; a tali nodi MIN si può assegnare il loro valore beta come valore sintetizzato finale;

b) si può tralasciare la ricerca sotto i nodi MAX aventi un valore alfa *maggiore o uguale* al valore beta di uno dei loro antenati MIN; a tali nodi MAX si può assegnare il loro valore alfa come valore sintetizzato finale.

Durante la ricerca, i VSP (valori alfa e beta) sono calcolati come segue:

1. il valore alfa dei nodi MAX è posto uguale al corrente valore finale *più grande* riportato all'insù dei suoi successori;

2. il valore beta dei nodi MIN è posto uguale al corrente valore finale *più piccolo* riportato all'insù dei suoi successori.

Nel caso a) si ha un *troncamento alfa*
Nel caso b) si ha un *troncamento beta*

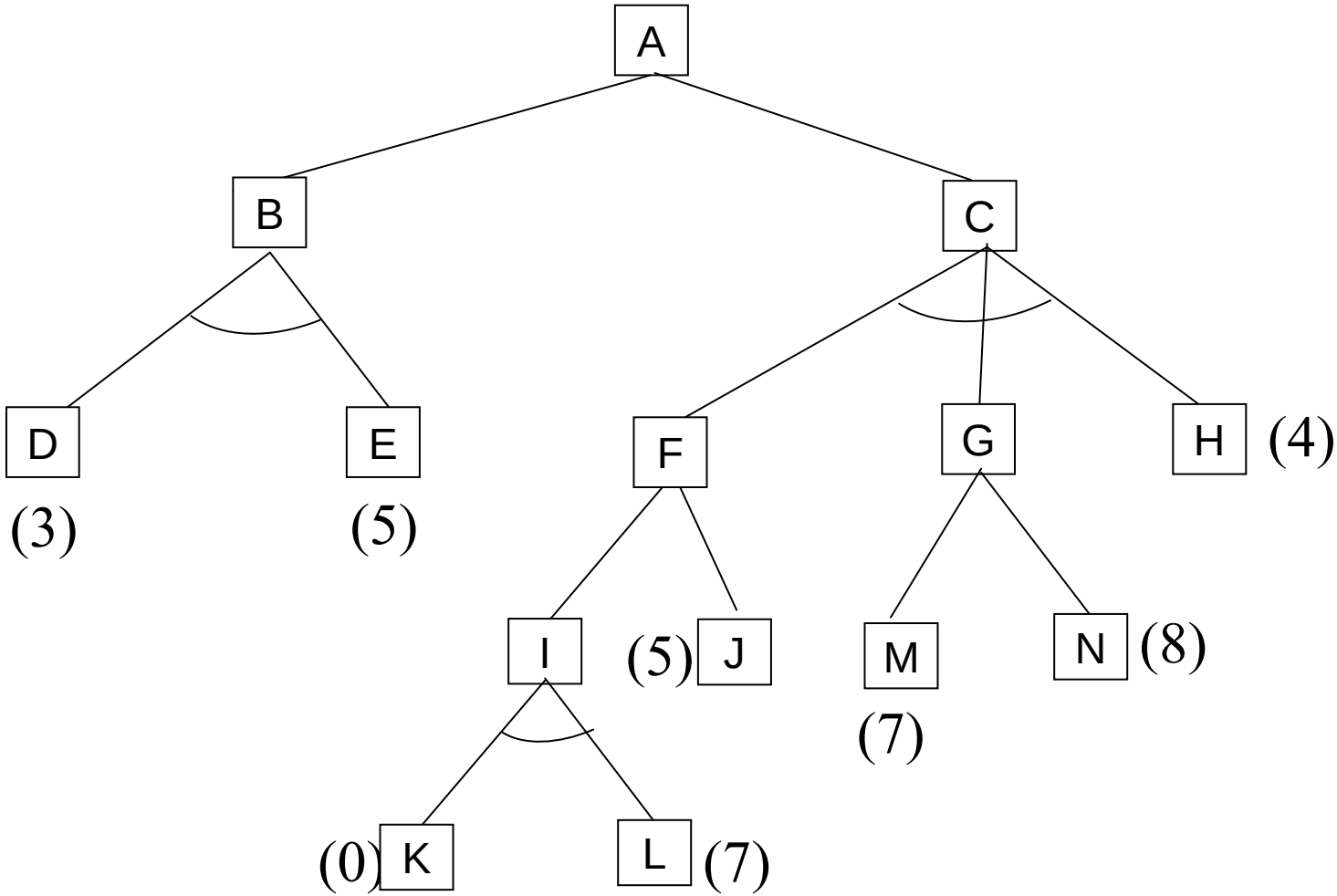
Da qui la dizione di *procedura alfa-beta*.

La procedura termina quando tutti i successori del nodo di partenza hanno ricevuto un valore sintetizzato finale.

La migliore mossa è quella corrispondente al successore con il massimo valore sintetizzato finale.

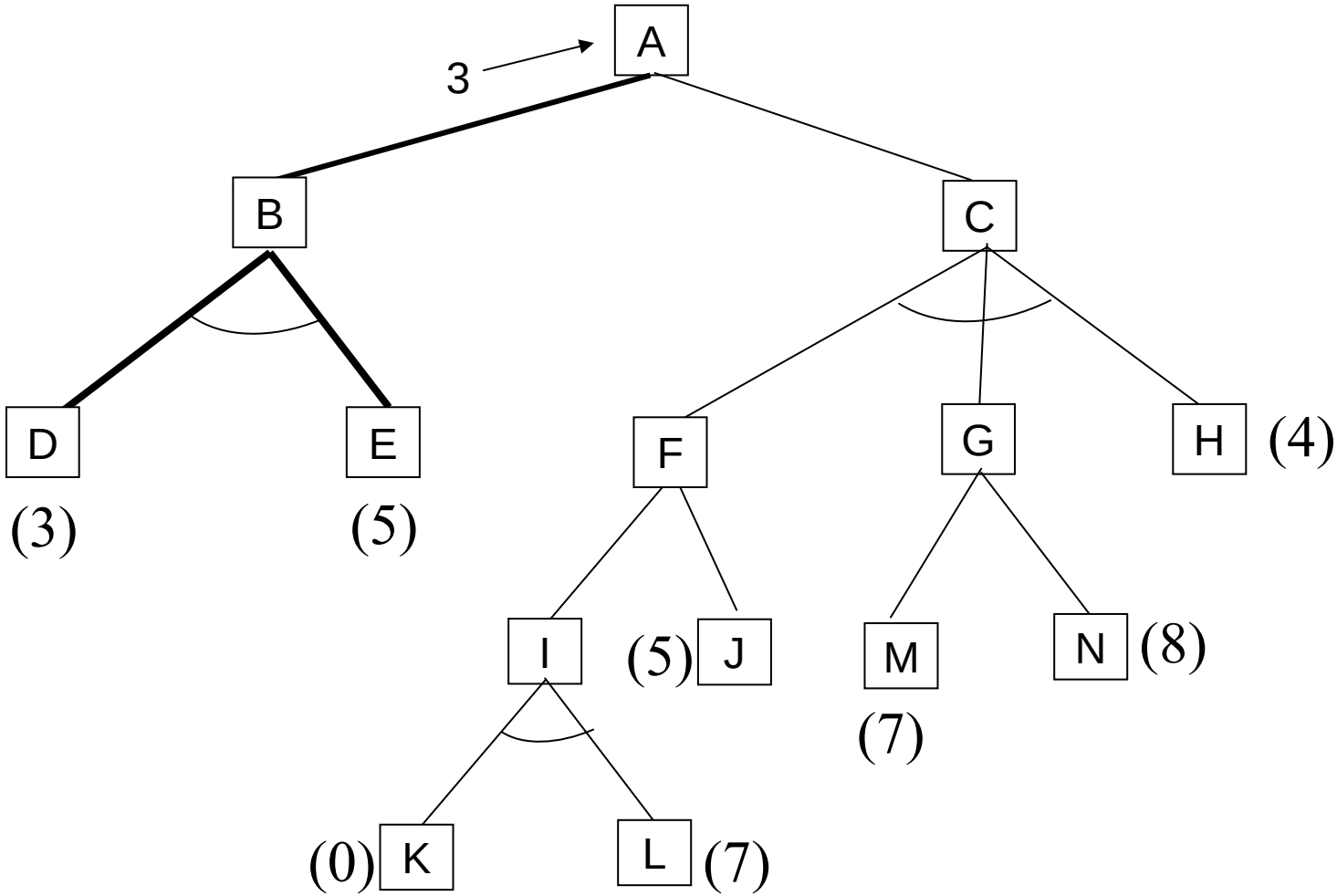
N.B: stessa mossa di MINI-MAX ma con meno ricerca.

Esempio:



Partendo da A, si genera B e poi D ed E.

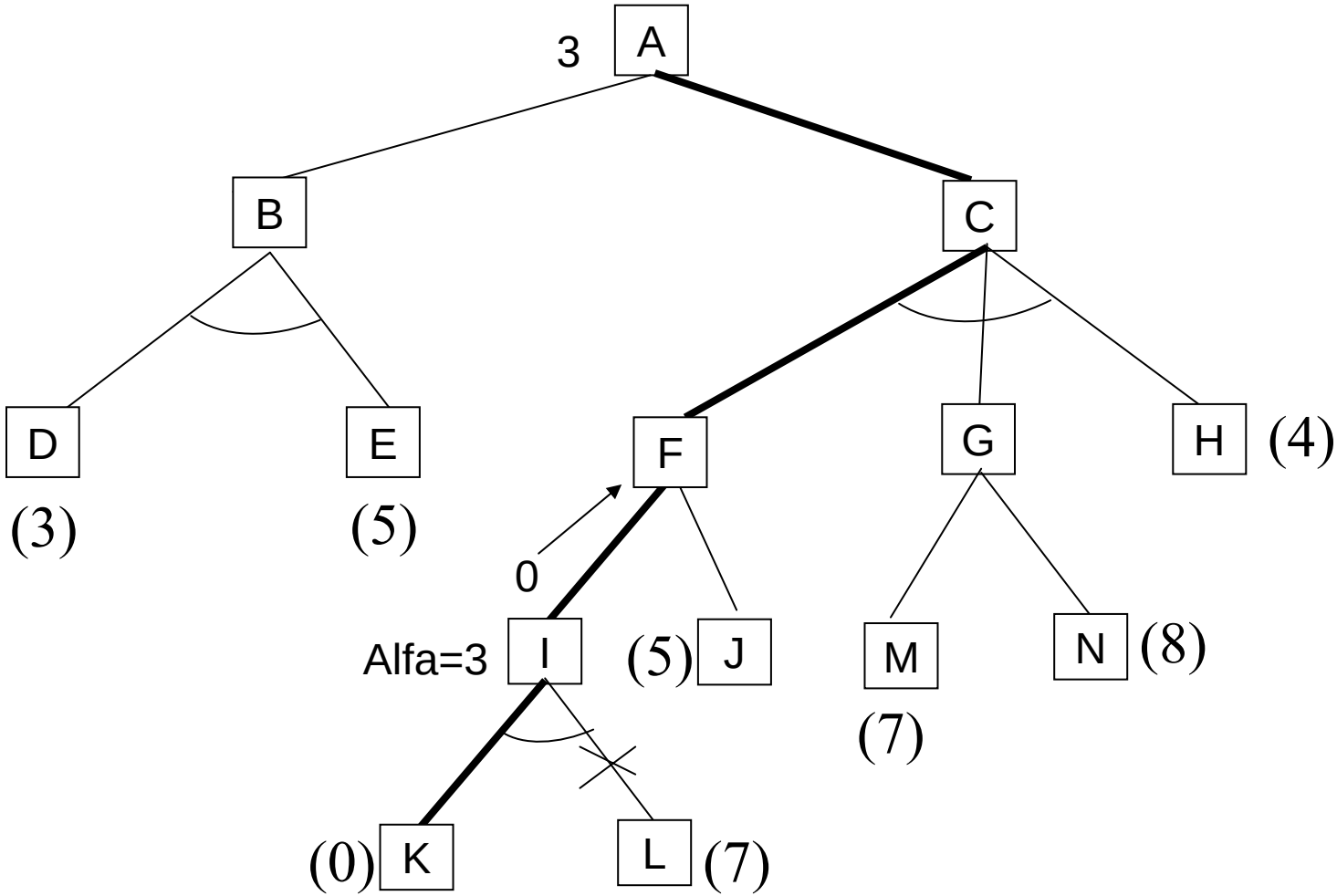
Si riporta indietro al nodo A la valutazione provvisoria pari a 3.


A
3
B
C
D
E
F
G
H
(4)
(3)
(5)
I
(5)
J
M
N
(8)
(7)
(0)
K
L
(7)

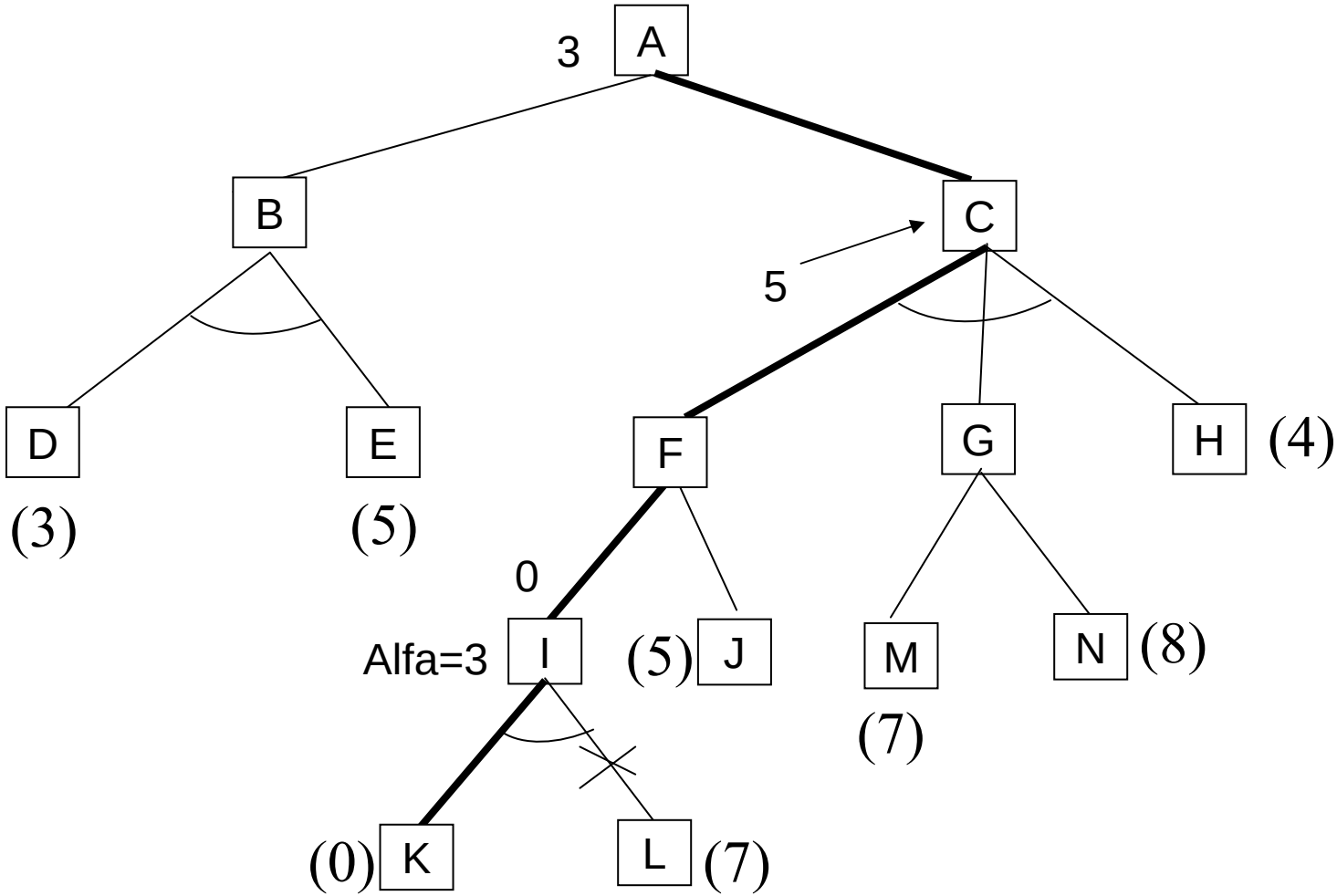
Questo valore *alfa* viene passato all'ingiù a C, F ed I. Da K viene su una valutazione provvisoria pari a 0 che viene (provvisoriamente) attribuita a F e a C.

Ma (provvisoriamente) B è migliore di C (e C verrebbe scelto solo per un valore superiore a 3), mentre in I la valutazione può solo scendere.

In I si effettua un taglio (L non viene generata).

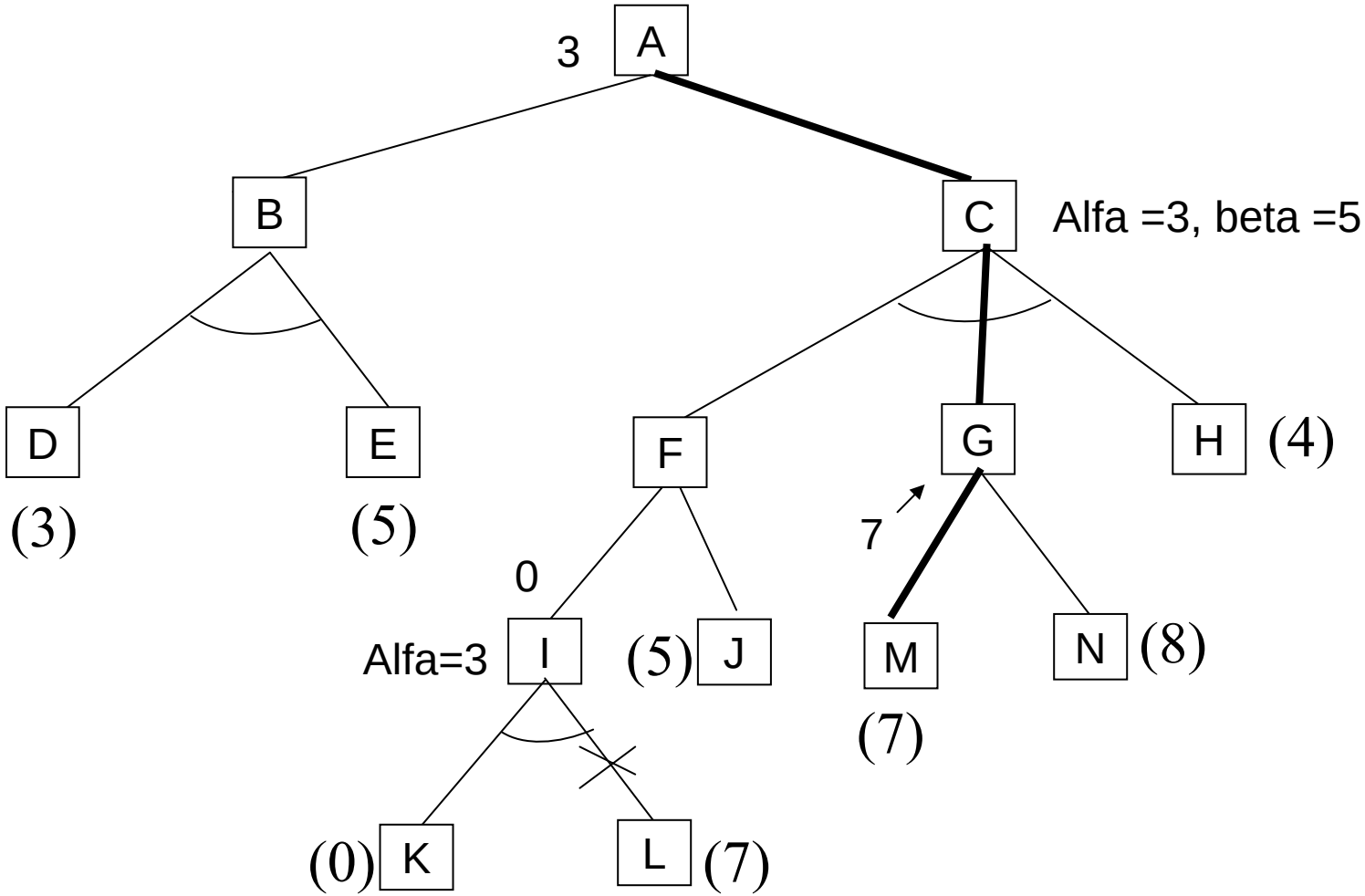

3
A
B
C
D
E
F
G
H
(4)
(3)
(5)
0
Alfa=3
I
(5)
J
M
N
(8)
(7)
(0)
K
L
(7)

F, che ha una valutazione provvisoria di 0, genera J e sceglie la valutazione 5 come sua propria valutazione, passandola a C.

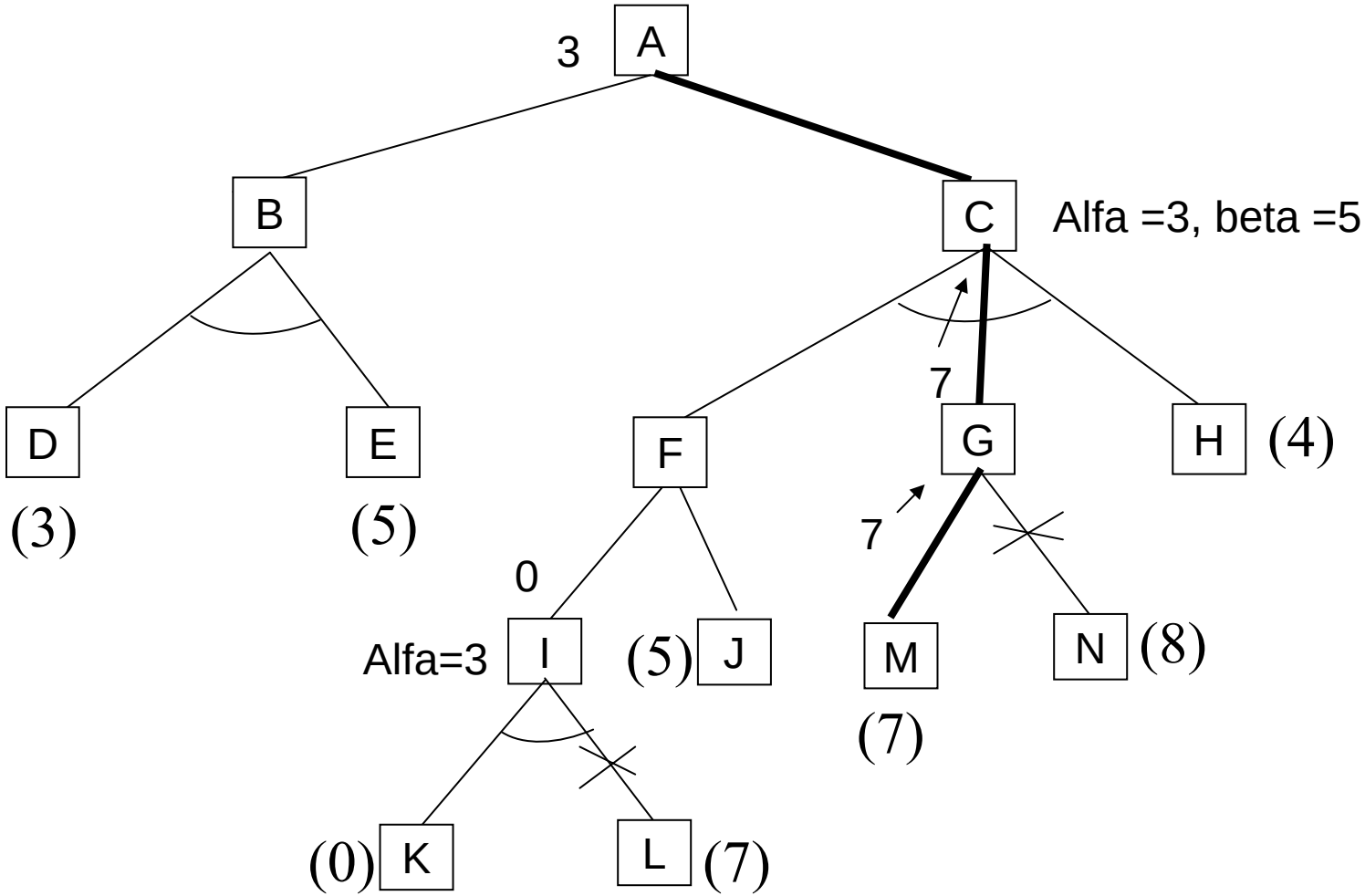

3
A
B
C
5
D
E
F
G
H
(4)
(3)
(5)
0
Alfa=3
I
(5)
J
M
N
(8)
(7)
(0)
K
L
(7)

Questo diventa il valore di *beta* al nodo C, che lo passa a G.

Questo espande M e da questo riceve la valutazione provvisoria 7, che viene confrontata con beta (5).


3
A
B
C
Alfa =3, beta =5
D
E
F
G
H
(4)
(3)
(5)
7
0
Alfa=3
I
(5)
J
M
N
(8)
(7)
(0)
K
L
(7)

Essa è maggiore mentre C deve minimizzare. Poiché G deve invece massimizzare, è inutile proseguire la ricerca sotto G: si effettua un taglio (la valutazione di G resta 7). E così via.


3
A
B
C
Alfa =3, beta =5
D
E
F
G
H
(4)
(3)
(5)
7
7
0
Alfa=3
I
(5)
J
M
N
(8)
(7)
(0)
K
L
(7)

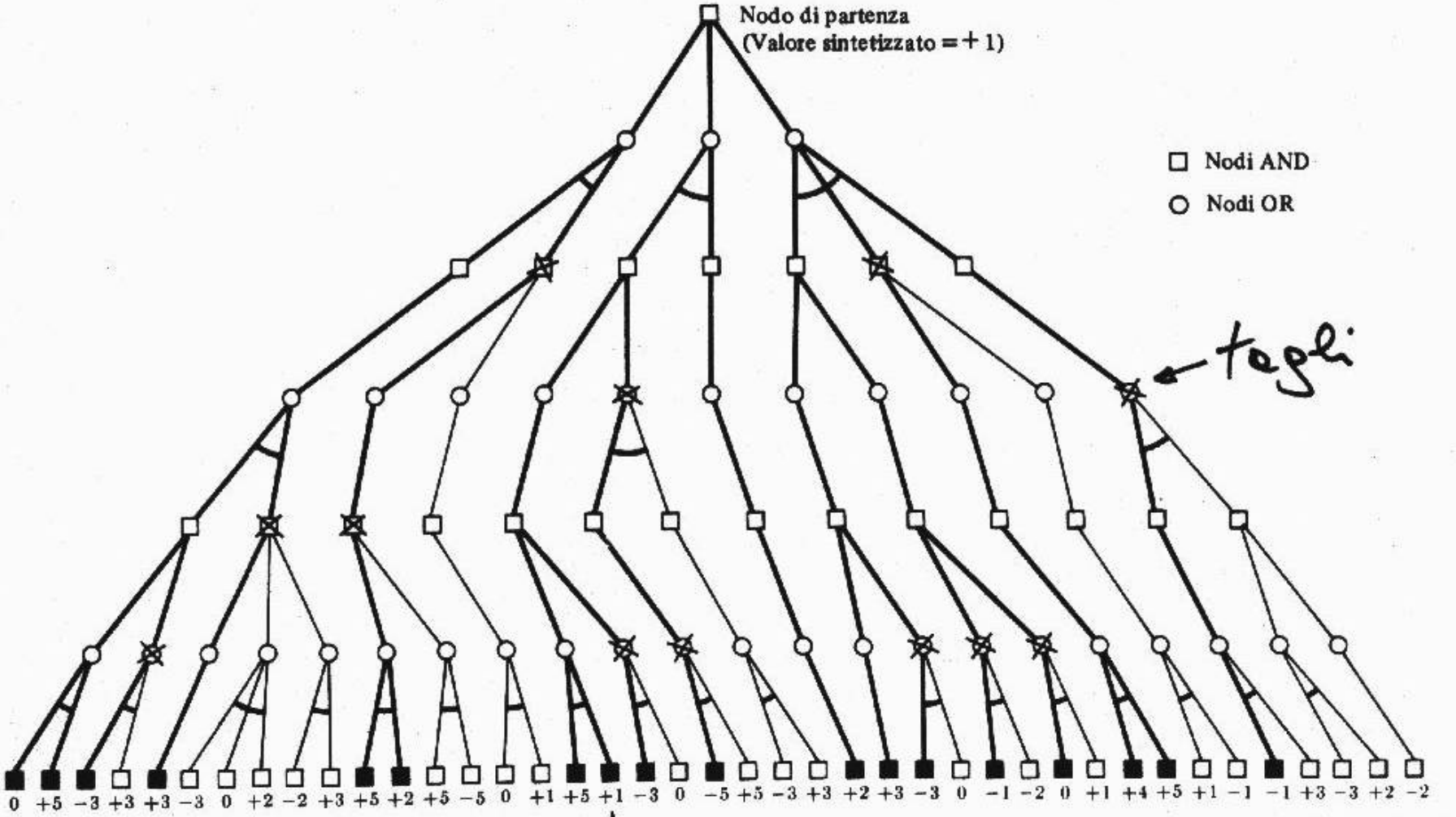
In conclusione l'effettivo uso di alfa e beta consiste nel terminare la ricerca

- ad un livello di minimizzazione quando viene scoperto un valore minore di alfa
- ad un livello di massimizzazione quando è stato trovato un valore maggiore di beta.

Esempio più complesso:

Si osservi che i tagli (e la riduzione di nodi visitati), dipendono dall'ordine con cui vengono generati i nodi (provare, per esercizio, a valutare i tagli se la visita dell'albero seguente avviene da destra verso sinistra).

— Un esempio che illustra la procedura di ricerca alfa-beta



In grassetto il sottoalbero generato dalla procedura alfa-beta. Si valutano 18 foglie contro le 41 di minimax.

Algoritmo ricorsivo per la ricerca minimax con potatura alfa-beta

(The following version, which evaluates the minimax value of a node *n* relative to the cut-off values α and β, is adapted from [Pearl 1984, p. 234])

AB(n; α, β)

1. If n at depth bound, return AB(n) = static evaluation of n. Otherwise, let n_1, … n_k, ... n_b be the successors of n (in order), set $k \leftarrow 1$ and, if n is a *MAX* node, go to step 2; else go to step 2′.

2. Set $\alpha \leftarrow \max[\alpha, \text{AB}(n_k; \alpha, \beta)]$.
3. If $\alpha \geq \beta$, return β; else continue.
4. If $k = b$, return α; else proceed to n_{k+1}, i.e., set $k \leftarrow k + 1$ and go to step 2.

2'. Set $\beta \leftarrow \min[\beta, \text{AB}(n_k; \alpha, \beta)]$.

3'. If $\beta \leq \alpha$, return α; else continue.

4'. If $k = b$, return β; else proceed to n_{k+1}, i.e., set $k \leftarrow k + 1$ and go to step 2'.

We begin an alpha-beta search by calling AB(s; $-\infty$, $+\infty$), where s is the start node.

Throughout the algorithm, $\alpha < \beta$.

The ordering of nodes in step 1 of the algorithm has an important effect on its efficiency.