Ricerca Combinatoria

Theo Gonella

27 marzo 2005

1 Generalità

Gli algoritmi combinatori servono per risolvere problemi di due tipi: - Decision Problem, che richiede la semplice ricerca di una soluzione. Il programma in questo caso si occupa di trovare la prima soluzione disponibile. - Optimization Problem, che richiede la miglior (secondo i parametri del problema) soluzione disponibile. La ricerca combinatoria quindi non si accontenta di trovare una soluzione qualunque, ma deve controllare che non sia possibile trovarne una migliore (vedi BRANCH & BOUND e DISTRIBUITED TERMINATION DETECTION (evoluto)) Una ricerca combinatoria può essere rappresentato da un albero (del tutto simile come struttura ideologica agli alberi informatici. Alla radice dell'albero (root) sta il problema iniziale. A partire da questo, vengono man mano esplorate tutte le possibili situazioni derivate dal problema iniziale. Ad esempio: Se si volesse vincere una partita di scacchi attraverso una ricerca combinatoria, la radice rappresenterebbe la situazione attuale della scacchiera. Al primo livello dell'albero si troverebbero tutte mosse possibili derivate dalla situazione precedente. Via via ogni sub-albero diventa una nuova *radice*, e si esplorano tutte (fino ad una certa profondità, ovviamente) le possibilità della partita. Quando la ricerca si imbatte in una situazione di scacco matto, cioè di vittoria, vengono ripercorse tutte le mosse necessarie per giungere a quella situazione. Arrivati al primo livello della ricerca, il giocatore sà quale sarà la mossa più conveniente. Più la ricerca combinatoria è profonda, più probabilità di vincere (cioè di scovare uno scacco matto in ogni situazione) si hanno.

Gli alberi di ricerca combinatoria possono avere dei nodi AND, OR o misti (AND/OR). I nodi AND richiedono che tutte i loro *figli* (cioè le loro diramazioni) siano risolte. I nodi OR richiedono la soluzione di uno qualunque dei loro figli. I nodi misti sono presenti soprattutto nelle ricerche simili all'esempio, nelle quali occorre esplorare le scelte dell'avversario (nodo AND) e nostre (nodo AND). Esistono vari tipi di tecnica di ricerca:

- Divide and Conquer (nodi AND)
- Backtrack Search, Branch & Bound (nodi OR)
- Minimax, Alpha-Beta Pruning (nodi misti)

2 Divide and Conquer

La ricerca divide and conquer consiste nel dividere il problema in sub-problemi, risolverli, e combinarli fra loro. I nodi della ricerca divide and conquer sono di tipo AND: occorre esaminare e risolvere tutto l'albero per comporre la soluzione finale. E' una metologia ricorsiva, per la quale un'implementazione parallela è estremamente problematica (passaggio fra processi di soluzioni dei sub-problemi, sincronizzazione dei messaggi ecc..). Ci sono due modi di rendere parallela una ricerca del genere.

- Il problema iniziale e la soluzione finale si trovano nella memoria di un solo processo (processo madre). In una prima fase il processo madre distribuisce il problema fra gli altri processi. Nella seconda fase ogni processo è impegnato nel calcolare le soluzioni. Nella fase finale il processo madre riceve le soluzioni, le combina ed elabora la soluzione. In questa struttura solo nella seconda fase si ha uno sfruttamento della parallelizzazione del problema, non permettendo quindi un reale incremento delle prestazioni.
- Mantenere il problema iniziale e la sua soluzione in ogni processo. In questo modo ogni processo è attivo fin dall'inizio, ma è estremamente problematico (vedi sopra) implementare una struttura del genere.

3 Backtrack Search

La ricerca backtrack può essere rappresentata attraverso un albero OR. Se si giunge ad un *punto morto*, detto anche *foglia*, si torna al nodo precedente e la ricerca continua, finchè non si giunge alla soluzione del problema. Questa tipologia di ricerca combinatoria è utile per problemi che richiedono una soluzione qualunque (Decision Problem), non una ottimizzata (esempio di soluzione qualunque: trovare il modo di vincere una partita di dama, ottimizzata :vincere la partita nel minor numero di mosse possibili). La ricerca backtrack infatti si blocca non appena viene trovata una soluzione, che non deve essere per forza la migliore.

Un modo per svolgere una ricerca di questo tipo in modo parallelo consiste nel distribuire la ricerca di sub-alberi tra i vari processi. La situazione ideale si manifesta quando il *branching factor* (ossia le diramazioni di ogni nodo) sono B, e il numero di processi (P) è tale che P=B^k. K quindi sarà il livello a partire dal quale sarà possibile distribuire un sub-albero per ogni processo. Questo metodo è tanto migliore quanto la profondità della ricerca totale è maggiore di K, in modo che lo scarso sfruttamento di processi nella prima fase possa essere ininfluente nella velocità della ricerca. Se non esiste un K tale che P=B^k, allora a partire da un certo livello M i processi si spartiranno i nodi più egualmente possibile.

Sfortunatamente la maggior parte dei casi l'albero di una ricerca backtrack non è bilanciato, per cui ci potrebbero essere degli squilibri nell'avanzare della ricerca. Un metodo per alleviare questo problema consiste nell'aumentare la profondità dell'albero, in modo da assegnare a ciascun processo un gran numero di sub-nodi. Questa strategia si basa su un calcolo statistico: se ogni processo elabora un gran numero di sub-alberi, allora le differenze nel tempo impiegato dai processi si abbasseranno, convergendo verso un valore costante per tutti.

4 Distribuited Termination Detection (*base*)

L'algoritmo di ricerca backtrack, come altre tipologie di ricerca, trova potenzialmente ogni soluzione al problema. Siccome ne basta una sola, occorre *terminare il programma* non appena un processo trova la soluzione. Le circostanze per cui un processo termina sono quindi:

1. ha trovato una soluzione e mandato il messaggio di termine agli altri;
2. ha ricevuto un messaggio di termine da un altro processo
3. ha finito di cercare nel suo sub-albero

Sfortunatamente, se un processo chiama `MPI_Finalize()` prima di un altro, che lo fa subito dopo, si ha un run-time error. Esempio: il processo A trova una soluzione e chiama `MPI_Finalize( )`. Subito dopo anche B trova una soluzione e chiama anch'esso `MPI_Finalize( )`, prima di ricevere il messaggio da A. In questa situazione si ha un errore in esecuzione.

Per ovviare questo problema occorre controllare che non ci siano messaggi in circolo nell'anello. Si utilizza un metodo inventato da famosi studiosi che si basa sul passaggio di un *testimone* fra i processi, prima dell'invio del messaggio di termine. Questo testimone viene passato da processo a processo, partendo da quello 0.

Il testimone ha due proprietà: il *colore* e il *contatore*. Anche ogni processo ha un colore e un contatore di messaggi, con i quali modifica quelli del testimone. Quando un processo inizia il suo lavoro nel programma parallelo, pone il suo contatore a 0. Un processo (da bianco) diventa nero quando riceve o manda messaggi. Quando riceve messaggi incrementa il suo contatore di 1, quando li manda lo decrementa di 1.

Il testimone viene lanciato quando un processo ha trovato una soluzione. All'inizio è bianco ed il suo contatore è a 0. Viene passato di processo in processo, fino a tornare a quello 0. Di volta in volta gli viene aggiunto il contatore di messaggi del processo dal quale si trova. Cambia il colore in nero solo se il processo è nero, altrimenti rimane bianco. Un processo nero cambia il colore del testimone, ma poi cambia in bianco il suo colore.

Tornato dal processo 0, quest'ultimo controlla il testimone. Se questo è bianco e ha il contatore pari a 0, significa che il sistema è quiescente, per cui si può terminare il programma.

5 Branch & Bound

Ecco un problema che ben si adatta ad una ricerca branch & bound:

Considerare il puzzle-8, un costituito da 3 righe e 3 colonne, contenenti caselline numerate da 1 a 8, che si possono spostare solo su l'unico spazio vuoto, lasciato dalla casellina 9 mancante. Si desidera ordinare le caselline in modo crescente, nel minor numero di mosse possibili.

La ricerca B&B, a differenza della backtrack, cerca la soluzione migliore del problema, cioè quella che impiega il minor numero di mosse per la risoluzione del puzzle. Quindi l'elemento nuovo del B&B è una funzione che sia in grado di *valutare la bontà* della strada intrapresa, scegliendo quindi la migliore. L'albero della ricerca questa volta sarà ancora di tipo OR, cioè non è necessario risolvere tutti i nodi. Tornando al puzzle, la funzione F (controllo qualità) consiste nel contare, di ogni casellina fuori posto, la sua distanza dalla posizione ideale.

La distanza viene calcolata considerando le mosse legali(orizzontale o verticale, non obliquo) che la casella deve svolgere per raggiungere la posizione giusta. F calcola la somma delle distanze per ogni figlio di un nodo, e sceglie la strada da intraprendere.

Un nodo viene suddiviso in numero di figli uguale al numero di mosse legali (da un massimo di 4- solo nella posizione centrale- ad un minimo di 2- negli angoli). F quindi calcola la somma delle distanze per ogni figlio di un nodo, scegliendo quello con il valore più basso *(valori più bassi = meno mosse = soluzione migliore).*

Tuttavia la ricerca in questo modo lascia da parte gran parte dei nodi dell'albero, che potenzialmente potrebbero contenere comunque delle soluzioni accettabili. Quindi in una ricerca parallela B&B è bene utilizzare una lista d'attesa per ognuno dei processi, nella quale sono elencati i sub-problemi da esaminare di volta in volta, ordinati secondo l'indice di bontà (dal migliore al peggiore). Ogniqualvolta un nodo si divide in sub-nodi, questi vengono inseriti nella lista d'attesa. Quando questa è piena, vengono eliminati i sub-nodi con indice di qualità meno vantaggioso. La migliore soluzione trovata localmente (per ogni processo), anche se non ottimale (vedi prossimo paragrafo), viene salvata.

Nella fase iniziale il processo 0 esamina il problema nel suo stato iniziale. Alla prima suddivisione della radice può già distribuire problemi agli altri processi, migliorando la velocità d'esecuzione. Man mano che vengono generati più nodi ogni processo viene reso operativo. Tuttavia l'efficienza del B&B non è sempre garantita, in quanto ogni processo esamina le situazioni che hanno indice di qualità migliore solo nei sub-alberi a loro assegnati, per cui spesso capita di cercare in zone inutili, e quindi di sprecare tempo in esecuzione.

Per questo sarebbe bene garantire la distribuzione di problemi fra processi, che contribuisce a mantenere una certa qualità nella ricerca complessiva. Infatti verrebbero in questo modo considerati solo i nodi con migliore indice di qualità.

6 Distribuited Termination Detection (*evoluto*)

Diversamente dal Backtrack Search, il B&B richiede la migliore soluzione, per cui, quando un processo trova una soluzione, occorre controllare anche che sia la migliore soluzione raggiungibile. Ciò corrisponde nel confrontare l'indice di qualità della soluzione appena trovata con quelli dei sub-nodi degli altri processi. Se l'indice è minore o uguale (oppure maggiore o uguale, dipende dai casi) a quelli degli altri, allora si tratta della migliore soluzione che si può ottenere, per cui si può terminare la ricerca. Il testimone quindi è provvisto di altre nuove qualità. Con sé porta ancora il colore ed il contatore, ma anche l'indice di qualità della soluzione appena trovata, e la soluzione stessa (le mosse svolte).

Ogni processo che riceve il testimone, ne cambia innanzitutto il colore (quando necessario) e il contatore di messaggi. Successivamente, controlla la soluzione contenuta nel testimone: se il processo ne ha trovata preceden-

temente una migliore, la salva nel testimone eliminando quella vecchia. Poi controlla gli indici di bontà dei sub-problemi presenti nella sua lista d'attesa. Se possiede dei migliori indici, il testimone informerà il processo 0 che non è il caso di accettare quella soluzione come migliore. Se invece non ci sono migliori speranze, il processo ri-inizializza la sua lista d'attesa con nuovi problemi, dato che comunque non potrebbe trovare soluzioni migliori.

Infine il testimone torna al processo 0, che determina se ci sono messaggi circolanti nell'anello, e se la soluzione trovata è la migliore. Solo in questo caso si ha il termine del programma.

In questa evoluzione quindi il testimone ha una *duplice funzione*: quella di rappresentare lo stato delle comunicazioni e delle soluzioni, ma anche quello di informare i vari processi se stanno cercando effettivamente soluzioni apprezzabili.

7 Alberi di Ricerca per Giochi

La maggior parte dei programmi usati per giocare ai cosiddetti giochi "a somma 0" sono basati su algoritmi di ricerca. Questi considerano tutte le possibili mosse e contromosse fino ad un certo livello di ricerca, e individuano, percorrendo all'indietro tutto l'albero, la mossa iniziale più vantaggiosa. L'algoritmo minimax, con le sue evoluzioni, è il più utilizzato per questo tipo di ricerca.

7.1 Algoritmo Minimax

Gli scacchi sono un gioco a somma 0 in quanto il vantaggio di un giocatore equivale allo svantaggio dell'avversario, e viceversa. La situazione di vantaggio o svantaggio in un gioco a somma 0 può essere rappresentato attraverso un valore numerico. Attraverso questo valore si potranno individuare durante la ricerca le situazione più o meno favorevoli, scegliendo la migliore di conseguenza. La funzione che assegna questo punteggio considererà parametri propri del gioco, e sarà fondamentale per la qualità di gioco del programma.

L'albero di ricerca ha *due diversi tipi di nodo*: uno che rappresenta il turno del giocatore corrente (d'ora in poi giocatore 1, quello che il programma vuole far vincere,) e l'altro che rappresenta il turno dell'avversario. Ad ogni livello infatti si alternano i due tipi di nodo, come nel gioco vero e proprio. I nodi che rappresentano l'avversario (di tipo AND) sono anche detti *MIN-Node*, perché sceglieranno, fra i loro figli, quelli con il valore più basso. Questo perché l'avversario sceglierà la mossa migliore per lui, cioè quella meno vantaggiosa

per il giocatore 1. Di conseguenza i nodi (di tipo OR) nei quali è il giocatore 1 a scegliere sono detti *MAX-Node* (da qui il nome MINIMAX dell'algoritmo).

La radice dell'albero rappresenta la situazione attuale della partita. A partire dalla radice il programma genera le evoluzioni possibili della partita (attraverso un'efficiente funzione generatrice di mosse legali), seguendo il turno dei giocatori. Più la ricerca è profonda più alta sarà la probabilità di vincere, a fronte però di un impiego di tempo sempre maggiore. Interrotta la ricerca ad un livello predefinito, quindi, verranno calcolati i "punteggi" delle foglie. Ora non resterà altro che percorrere l'albero al contrario, controllando il tipo del nodo subito superiore: se è di tipo MIN, allora andrà scelto il valore più basso tra i suoi figli, se è di tipo MAX andrà scelto il valore maggiore. Arrivati alla radice è possibile individuare la mossa migliore, semplicemente scegliendo fra i suoi figli quello con il valore maggiore.

E' facile intuire che le dimensioni e il numero di nodi dell'albero cambiano di gioco in gioco. Gli scacchi rappresentano il gioco più difficile da implementare. Non a caso alcuni tra i più famosi e potenti super-calcolatori sono costruiti appositamente per questo gioco.

7.2 Alpha-Beta Pruning

Più una ricerca minimax è profonda, migliore sarà la qualità del gioco. Per questo la potatura alpha-beta è tanto preziosa. Questa tecnica è una forma di algoritmo B&B perché esclude dalla ricerca quei nodi che non possono in alcun modo influire sulla decisione della mossa. La potatura (pruning) dell'albero permetterà quindi di raggiungere una maggiore profondità a parità di tempo di programma minimax. L'algoritmo prende nome da due nuovi parametri: a e ß, che rappresentano una sorta di finestra del gioco. La regola fondamentale dell'algoritmo è

$$\alpha < val < \beta$$

Con val che rappresenta il punteggio del nodo in questione.

Inizialmente $\alpha = \infty$ e $\beta = -\infty$. La ricerca con alpha-beta pruning avviene in modo leggermente diverso da quella minimax. Non vengono generati immediatamente tutti i nodi e le foglie, bensì si prosegue per un unico percorso (a caso). Quando viene generato un nuovo nodo, questo eredita i valori di a e ß da quello immediatamente superiore, fino ad arrivare alla foglia, della quale viene calcolato il punteggio.

Ora la ricerca prosegue in ordine inverso, salendo di nodo in nodo solo quando tutti i suoi figli sono stati esaminati e il suo valore (a seconda che sia MIN o MAX) determinato. La possibilità di potare l'albero dipende dai

valori di α e β. Più in particolare, *i ruoli di α e β cambiano da MIN-Node e MAX-Node.*

- I MIN-Node possono modificare solo β, perché rappresenta il loro limite massimo. Quando un MIN-Node ha esaminato una soluzione di un suo figlio, allora la confronterà con il valore di β. Se il valore della soluzione è minore di β, allora significa quel nodo potrà in ogni caso scegliere una soluzione migliore (che per i MIN-Node è data dal punteggio minore), per cui β diventerà il valore appena trovato. Per i l'avversario quindi β rappresenta la soluzione migliore ottenibile.

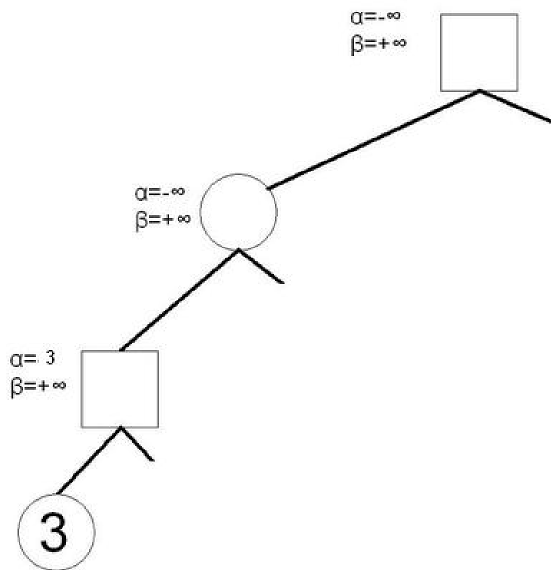
- Al contrario, i MAX-Node possono modificare solo il proprio α. Infatti, quando trovano una soluzione, la confrontano con α. Se è maggiore significa che il giocatore 1 potrà raggiungere una soluzione migliore, quindi aggiornano il valore dell'indice.

La particolarità della potatura $\alpha - \beta$ sta nel fatto che, attraverso il valori di α e β, è possibile *bloccare la ricerca* di alcuni rami, che non influiranno certamente nella ricerca (se non allungandone il tempo). Infatti, ogni nodo, sia MIN che MAX, possiede tutti e due i parametri (che eredita dal nodo soprastante) quando viene creato. Per un MIN-Node, α rappresenta il valore che il nodo soprastante (ovviamente un MAX-Node) può raggiungere. Il MIN-Node, come detto prima, se troverà una soluzione con punteggio minore di β, cambierà il valore di quest'ultimo. A questo punto occorrerà controllare che sia ancora valida la regola $\alpha < val < \beta$. Infatti se i valori di α e β si fossero superati, è possibile interrompere la ricerca per quel nodo. Ma come mai è possibile "potare l'albero"?

Si immagini di trovarsi, nello scorrere della ricerca, in un MIN-Node. Esso ha ereditato i valori di α e β da un nodo superiore. In particolare, il valore di α indica il valore che il giocatore 1 può raggiungere. Se il MIN-Node trova una soluzione migliore di β, che viene quindi aggiornato, sicuramente non sceglierà soluzioni peggiori (cioè dal punteggio maggiore) di β. Si immagini che $\beta < \alpha$: anche se il MIN-Node dovesse trovare soluzioni dal punteggio più basso, il nodo MAX soprastante sceglierà sicuramente α, che per lui è più vantaggioso.

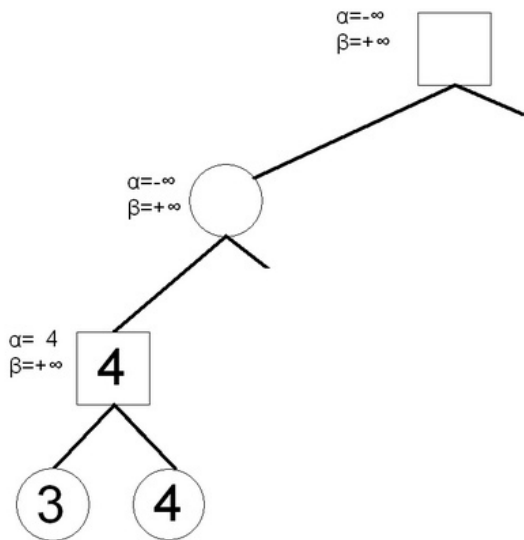
La stessa cosa dicasi per il MAX-Node, che ogni volta che aggiornerà il valore di α, dovrà controllare il valore di β.L'efficienza dell'algoritmo $\alpha - \beta$ migliora quanto più è bilanciato l'albero, cioè se le soluzioni migliori ad ogni nodo sono sempre le prime ad essere esaminate. Se si è così fortunati, degli studi hanno determinato che un programma basato su algoritmo $\alpha - \beta$ può esplorare fino al doppio della profondità, nello stesso tempo, di un altro basato su normale algoritmo minimax.

Breve Esempio di Ricerca con Alpha-Beta Pruning Attraverso questo esempio si cercherà di rendere più chiaro lo svolgersi di una ricerca che adotta l'alhpa-beta pruning. Per convenzione, i MAX-Node vengono rappresentati con dei quadrati, i MIN-Node con dei cerchi. Come già detto, questa ricerca avviene in modo diverso da quella minimax: l'albero viene esplorato inizialmente per un unico percorso, fino a giungere al livello considerato soddisfacente.

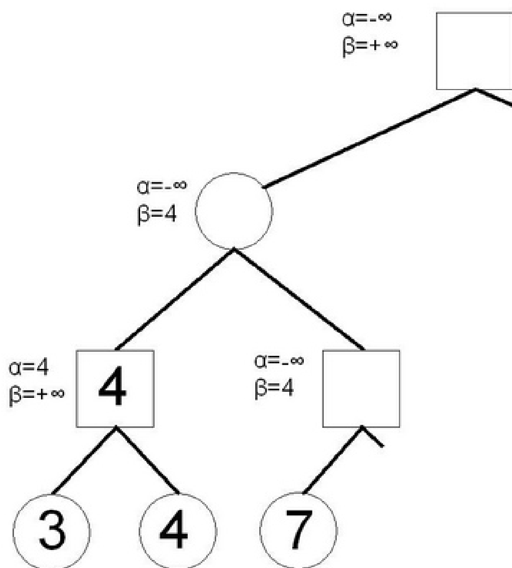


Gli iniziali valori di α e β (rispettivamente $-\infty$ e $+\infty$) vengono trasmessi dalla root per tutti i suoi figli. Questa è una regola generale dell'alpha-beta pruning: **i vari nodi, al momento della loro creazione, ereditano i valori di α e β dal nodo genitore**. L'ultimo nodo, di tipo MAX, ha trovato come soluzione 3. Come detto sopra, i MAX-Node possono cambiare solo il proprio α, che sta ad indicare il limite inferiore del giocatore 1. Il fatto di aggiornare il valore di α con 3 è analogo ad affermare: "*il giocatore 1 può totalizzare **al minimo** 3*".

Ora viene generato l'altro figlio del nodo MAX. Si ricorda che si può salire di livello solo quando sono stati considerati (o potati) tutti i figli del nodo.



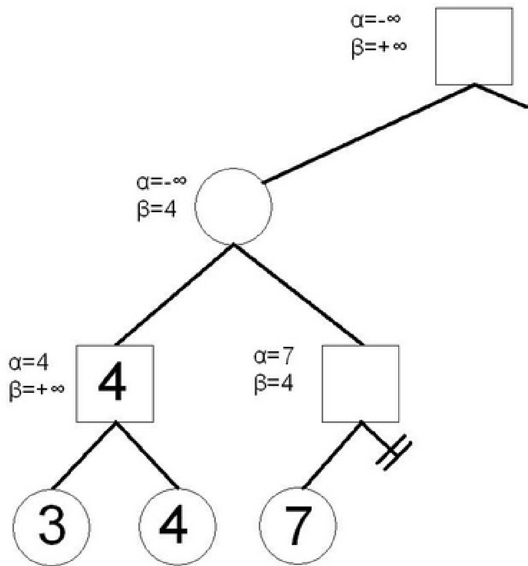
Il secondo figlio ha come punteggio 4. Siccome 4>3, il valore di α diventa 4, poichè ora il giocatore 1 è in grado di raggiungere un punteggio più alto. Se il nodo non ha più figli, siccome è di tipo MAX, sceglierà come proprio valore quello più alto fra i suoi figli.



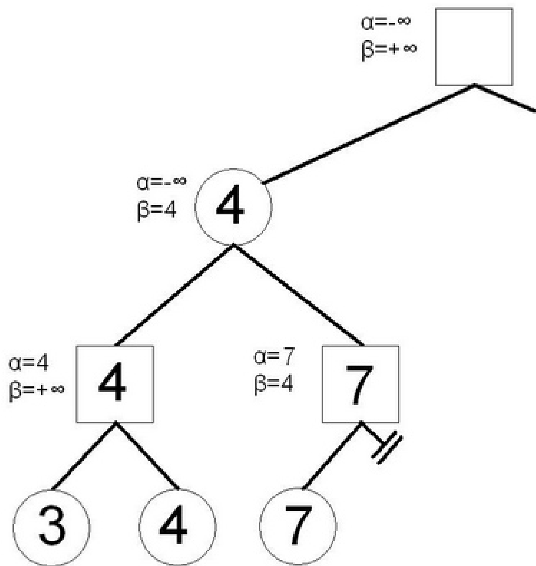
Ora la ricerca si sposta al nodo MIN, del quale il valore di β passa da $+\infty$ a 4. Infatti, quando la ricerca sale di livello, se i valori di α o β sono stati modificati

(rispetto alla situazione iniziale), essi modificano quelli del nodo genitore, ma in modo inverso (α modifica β, e β modifica α).

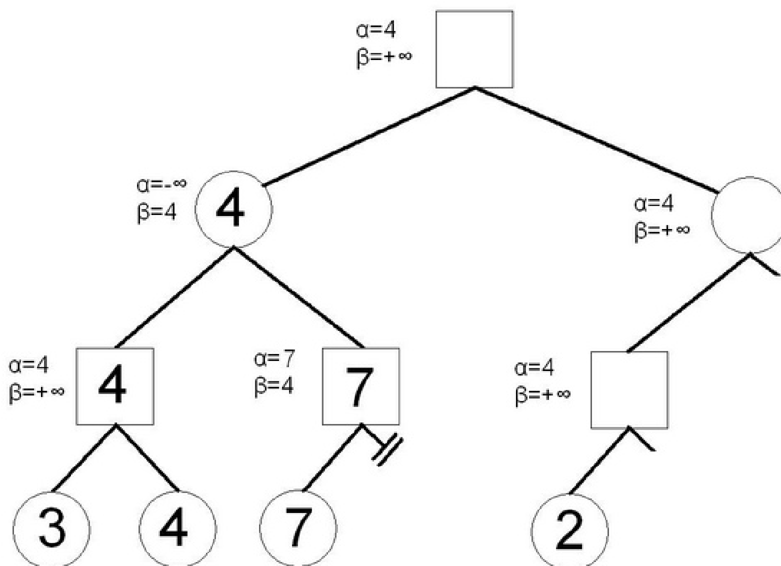
Ora viene generato il primo figlio del nodo, che erediterà i valori di α e β. Di questo si genera i figli. Il primo ha punteggio 7.



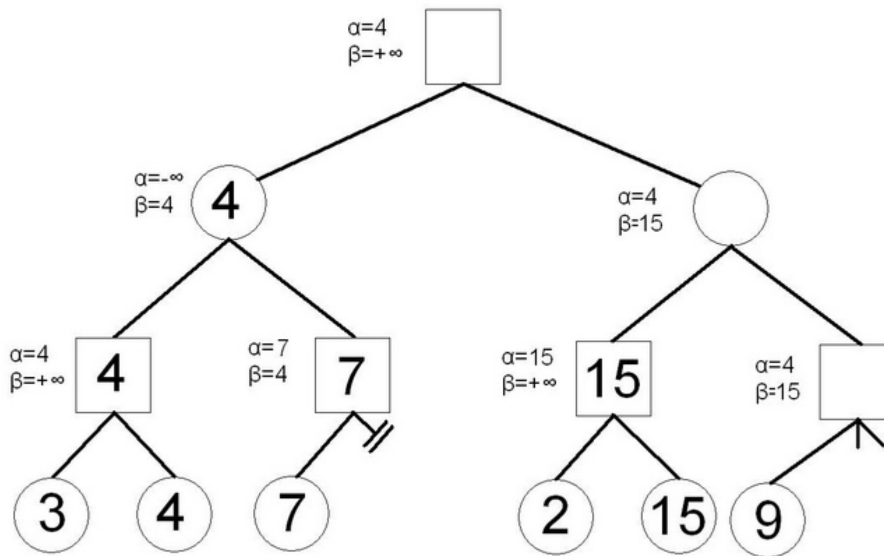
Questa è una situazione analoga a quella della seconda figura: ora il MAX-Node aggiornerà il valore di α da $-\infty$ a 7. **Ma ora è successo qualcosa di diverso:** $\alpha > \beta$. Ora è possibile interrompere la ricerca per quel nodo, perchè occorre ricordare che il MIN-Node al livello superiore sceglierà una soluzione<=4. Invece, il MAX-Node punterà a cercarne una >=7, ed i suoi sforzi di ricerca sono inutili, perchè l'avversario sceglierà la strada che gli assicura un punteggio minore (migliore per lui). In questa situazione è più facile capire il ruolo di ”limitatori” propri di α e β.



Il MAX-Node naturalmente assumerà valore 7, e il MIN-Node soprastante 4, perchè sceglie il punteggio minore. **In questo caso i valori di α e β non cambiano, perchè è avvenuta una potatura, e la situazione generale non è sostanzialmente cambiata.**

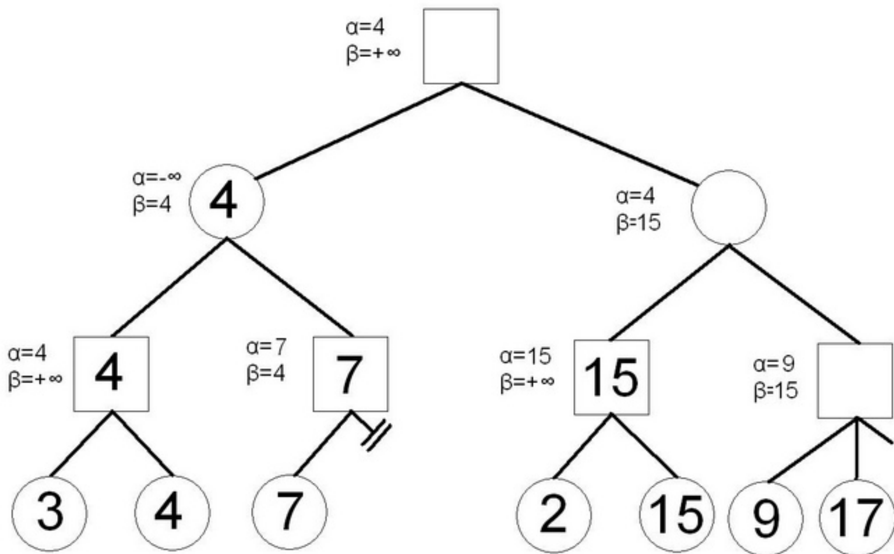


La ricerca sale di livello, e siccome il valore di β è cambiato (rispetto alla situazione iniziale), si aggiorna il valore di α del nodo superiore. Ora, come all'inizio, viene generato l'altro percorso figlio della root, fino al livello prestabilito. Tutti i figli ereditano i valori di $\alpha = 4$ e $\beta = +\infty$. Il punteggio del primo figlio del MAX-Node è 2. Siccome 2<4, non occorre cambiare il valore di α.

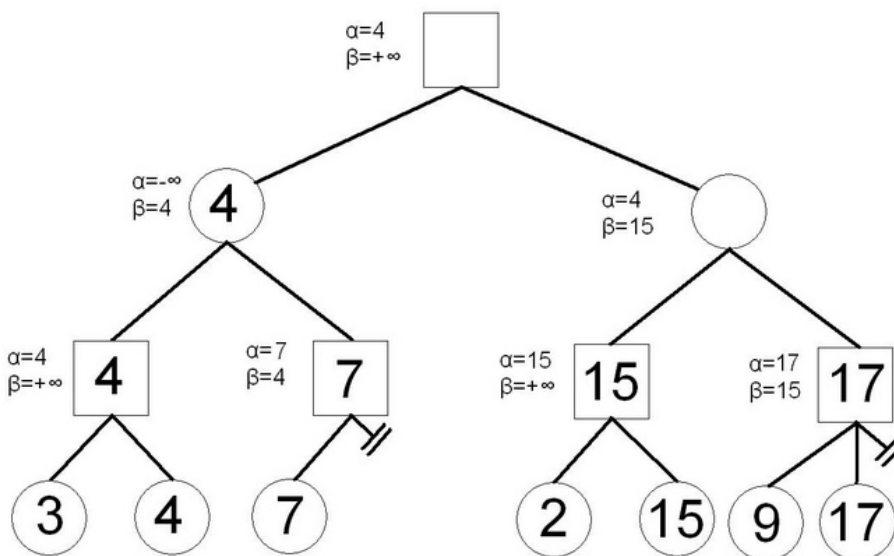


Il punteggio dell'altro figlio è 15, che è maggiore di 4, per cui il valore di α viene aggiornato. Tutti i figli del MAX-Node sono stati esplorati, per cui si può salire di livello.

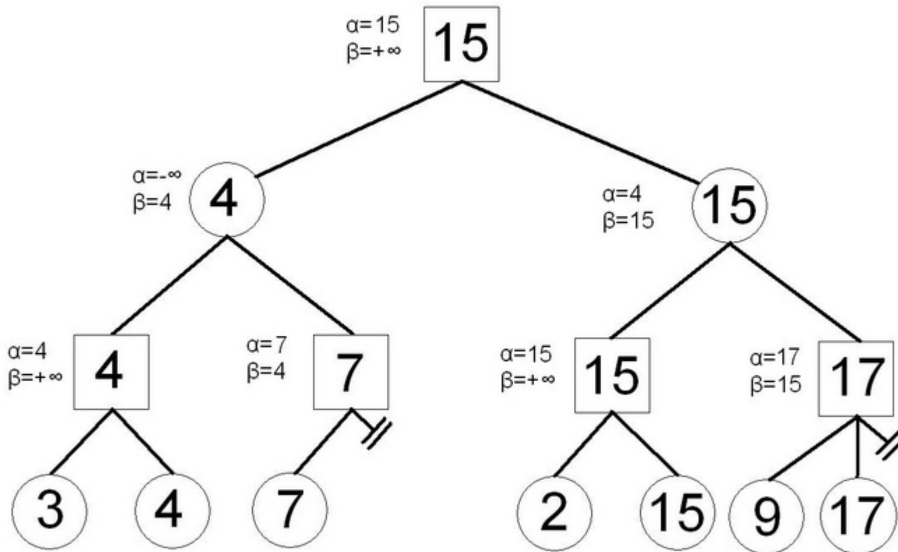
Il valore di α è cambiato, per cui il β del MIN-Node soprastante diverrà uguale a 15. Vengono generati i vari figli, che ereditano α e β, fino alla prima "foglia".



Questa ha come punteggio 9, che è maggiore di 4, per cui il valore di α viene aggiornato, e viene generato il nuovo figlio.



Questo vale 17, quindi viene aggiornato il valore di α. Ma ancora una volta $\alpha > \beta$, per cui è possibile interrompere la ricerca su quel nodo (si suppone che quest'ultimo, diversamente da tutti gli altri, avesse 3 -o più- figli).



Ora si punta al MIN-Node (i cui valori di α e β non cambiano). Tutti i suoi figli sono stati esplorati, per cui esso assume valore 15.

Ecco raggiunto l'obbiettivo dell'intera ricerca: conoscere la mossa iniziale più conveniente. E la risposta sta semplicemente nel scegliere il MIN-Node con valore maggiore. Nel nostro caso, il programma sceglierà la strada verso "destra", consapevole del fatto che il punteggio minimo raggiungibile è 15.