# Tecniche di potatura alpha-beta nei giochi combinatori a somma nulla

Sebastián Matías Camponovo

September 2, 2022

**Abstract**

Nel corso del seminario siamo interessati a descrivere l'applicazione di algoritmi basati su tecniche di *alpha-beta pruning* indirizzate a ottimizzare l'esplorazione di alberi di gioco; ci interessiamo a giochi sequenzialmente finiti a somma nulla con informazioni perfette. L'idea è di migliorare l'algoritmo minimax fornendo un criterio per *potare* i rami del nostro albero di ricerca che non risultino essere rilevanti per la ricerca della strategia di gioco ottimale; rendendo possibile un'analisi accurata anche là dove la complessità del gioco impedisca di esplorare ogni nodo.

## 1 Giochi combinatori a somma nulla

Iniziamo definendo formalmente la tipologia di giochi che siamo interessati a studiare:

**Definizione 1.1.** Un gioco combinatorio è un gioco:

- a due giocatori, i quali giocano succedendosi in turni,
- ad informazione perfetta, ovvero i giocatori conoscono tutte le possibili configurazioni del gioco,
- deterministico, ogni mossa determina la nuova configurazione del gioco.

In particolare considereremo giochi combinatori che siano:

- a somma nulla, quindi la funzione di payoff di un giocatore è "meno" la funzione di payoff dell'altro giocatore,
- sequenzialmente finiti, ovvero data la posizione iniziale del gioco, esiste un numero finito di configurazioni raggiungibili tramite le mosse dei giocatori e non sono permesse successioni infinite di mosse.

Una partita di un gioco combinatorio può essere descritta formalmente tramite i seguenti elementi:

- $S_0$: Lo stato iniziale che descrive la struttura di partenza del gioco.
- Mossa($s$): Il giocatore cui tocca il turno di muovere allo stato s.
- Azioni($s$): L'insieme di mosse legali allo stato s.
- Risultato(s,a): Il modello di transizione che definisce lo stato risultante dall'eseguire l'azione $a$ allo stato $s$. Abbreviata nel seguito in R(s,a).
- Terminale(s): Un test terminale che risulta essere vero quando il gioco termina allo stato $s$ e falso quando $s$ non è stato conclusivo di una partita. Uno stato $s$ che restituisce vero è detto *stato terminale.*
- Utilità($s$, $p$): Una funzione payoff che attribuisce un valore finale numerico al giocatore $p$ quando il gioco si conclude con un certo stato terminale $s$. Nel seguito utilizzeremo $U(s, p)$ per riferirci alla funzione utilità.

# 2 Albero di ricerca

Lo stato iniziale, assieme alle funzioni Azioni e Risultato, definiscono un grafo dove i nodi $s_i$ sono rappresentati dagli stati del gioco ed un arco $s_i \to s_j$ corrisponde ad una mossa che consente di passare dal primo al secondo.
Possiamo quindi associare al grafo un albero di ricerca e in accordo con ciò utilizzeremo alternativamente i termini *foglie* per riferirci agli stati terminali, *rami* per gli archi e *radice* per lo stato iniziale.

**Definizione 2.1.** L'albero di ricerca completo è l'albero che segue ogni possibile successione di mosse fino al raggiungimento di uno stato terminale.

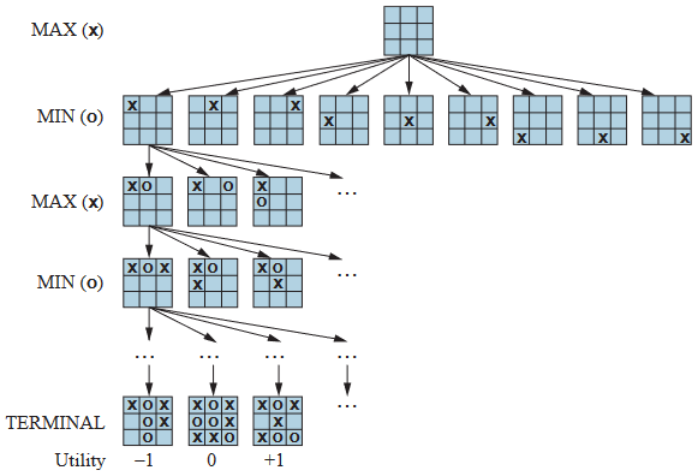



Figure 1: Un esempio di albero di gioco relativo al tic-tac-toe. La semplicità del gioco permette l'esplorazione dell'albero tramite brute force, il gioco è quindi di fatto *risolto*, e gioco perfetto da parte di entrambi i giocatori conduce a un pareggio (da [SP22]).

Ricordiamo due definizioni relative alla teoria degli alberi di gioco che utilizzeremo nel seguito:

**Definizione 2.2.** La profondità dell'albero è la distanza massima fra lo stato radice ed uno stato terminale.

**Definizione 2.3.** Il fattore di diramazione dell'albero è il numero di nodi figlio per ogni nodo dell'albero.

**Osservazione 2.1.** Spesso per semplificare calcoli si considera il fattore di diramazione medio, e.g. nel gioco degli scacchi è stimato un fattore di diramazione medio di 35 a discapito delle 20 mosse legali alla prima ply.

## 2.1 Formalismi Max/Min e Negamax

Prima di iniziare a studiare gli alberi stabiliamo le relazioni fra due convenzioni in uso nello studio di giochi combinatori. Per farlo introduciamo il problema di assegnare ad ogni stato un valore, così da renderli confrontabili. Se ci troviamo ad uno stato terminale $s$ possiamo richiamare la funzione Utilità ed attribuire un *punteggio* $U(s, p)$ ai giocatori (ricordiamo che per ipotesi sui giochi combinatori a somma nulla se per un giocatore $p_1$ il nodo $s$ ha valore $U(s, p_1)$, la valutazione dell'avversario è $-U(s, p_1)$).
Più in generale se muovendo allo stato $s$ sono raggiungibili gli stati $s_1, ..., s_d$ $(d > 1)$ siamo interessati alla scelta della *mossa migliore.*
Assumiamo che la nostra mossa migliore sia quella che massimizza il valore della funzione Utilità supponendo gioco ottimale da entrambi i giocatori a partire dallo stato $s$.
Sia $F(s)$ la funzione che realizza queste richieste dal punto di vista del giocatore che muove allo stato

$s$; osservando che il valore per questo giocatore dopo aver mosso in una posizione successore $s_i$ sarà proprio $-F(s_i)$, possiamo definire ricorsivamente una formula per la funzione che stavamo cercando:

$$F(s) = \begin{cases} U(s) & se\ d = 0 \\ max(-F(s_1), ..., -F(s_d)) = & se\ d > 0 \end{cases} \tag{1}$$

Adesso siamo interessati a ri-definire rigorosamente questa funzione anche in accordo con il formalismo Max e Min dove i valori vengono espressi "con la prospettiva di Max". Sia quindi $s$ uno stato terminale con Max che deve muovere, il valore della funzione è $U(s, p) = f(s)$ (rimuoviamo la dipendenza della funzione dal giocatore; informazione che viene *pesata* dal segno), se invece la mossa è di Min il valore è $g(s) = -f(s)$.
Chiaramente, in accordo con la nomenclatura del formalismo, Max vuole massimizzare la funzione introdotta e Min cerca di minimizzarla. Possiamo quindi riassumere le nostre richieste tramite la seguenti funzioni equivalenti a (1):

$$F(s) = \begin{cases} f(s) & se\ d = 0 \\ max(G(s_1), ..., G(s_d)) & se\ d > 0 \end{cases} \tag{2}$$

$$G(s) = \begin{cases} g(s) & se\ d = 0 \\ min(F(s_1), ..., F(s_d)) & se\ d > 0 \end{cases} \tag{3}$$

**Osservazione 2.2.** La convenzione "minimax" è preferibile per evitare confusione; mantentendo costante la prospettiva dalla quale stiamo valutando le posizioni di gioco riduciamo il rischio di incappare in errori.
D'altro canto abbiamo fin da subito un esempio delle problematiche del formalismo "minimax" osservando che la funzione (1) si è *sdoppiata* in due funzioni che tengono in considerazione il ruolo non simmetrico di Max e Min. Per evitare l'appesantirsi della notazione (e semplificare le dimostrazioni che seguiranno) ci proponiamo quindi di considerare la notazione *negamax* di (1) dove conveniente.

**Osservazione 2.3.** All'atto pratico nella definizione di $F(s)$ stiamo seguendo una via molto rigida e conservativa, supponendo sempre la miglior risposta da parte di Min. Nel momento in cui Max creda di trovarsi di fronte ad un avversario subottimale possono essere attuabili strategie più *rischiose* che preferiscano una mossa inferiore, ma con più prospettive di vittoria, ad una migliore che però porta a stati di pareggio.
Non siamo però interessati a tali dilemmi ed osserviamo come la strategia minimax garantisca sempre il miglior risultato in caso di gioco ottimale da entrambe le parti, ipotesi che richiediamo nell'analisi degli algoritmi che seguono.

# 3 Sviluppo dell'algoritmo

Il seguente algoritmo riportato in pseudocodice descrive una procedura esplicita per calcolare $F(s)$, la funzione definita da (1), che associa ad ogni stato $s$ il più grande valore ottenibile giocando contro una strategia difensiva ottimale:

**Algoritmo 1** Minimax

```
1: procedure F(s)
2:     inizializza m, i, t, d
3:     determina stati successivi s_1, ..., s_d;
4:     if d = 0 then
5:         F = f(s)
6:     else
7:         m = −∞;
8:         for i ← 1, d do
9:             t ← −F(s_i)
10:            if t > m then
11:                m = t;
12:                end;
13:                F = m;
14:                end;
15:                end.
16:
```

Questo algoritmo è la più semplice forma di *brute force* che esplora ogni possibile continuazione del gioco, seguendo in profondità ogni ramo dell'albero fino ad uno stato terminale.

**Osservazione 3.1.** Il minimax è quindi un algoritmo di natura *depth first*, elemento che condivide con le *evoluzioni* che andremo a studiare.

Una prima idea per migliorare l'efficienza, senza cambiare la soluzione finale, dell'algoritmo di ricerca consiste nella tecnica del "branch-and-bound" ovvero nell'ignorare mosse incapaci di migliorare strategie che conoscevamo anticipatamente. Semplicemente interrompiamo la ricerca non appena troviamo una successione di mosse che **performa peggio** di una esplorata in precedenza. Formalizziamo questa idea (in terminologia classica si parla di *rifiutare una mossa*), tramite il seguente algoritmo $F1$ dipendente dallo stato in cui ci troviamo ed un bound definito ricorsivamente. L'obiettivo è quello di ridurre il costo computazionale della procedura $F$ tramite l'introduzione delle seguenti condizioni aggiuntive:

$$\begin{cases} F1(s, bound) = F(s) & F(s) < \ bound \\ F1(s, bound) \geq bound & \text{altrimenti} \end{cases} \tag{4}$$

più concretamente uno pseudocodice che simula un'applicazione del metodo è il seguente:

**Algoritmo 2** Branch-and-Bound

```
procedure F1(s, bound)
    inizializza m, i, t, d
    determina stati successivi s_1, ..., s_d;
    if d = 0 then
        F1 = f(s)
    else
        m = −∞;
        for i ← 1, d do
            t ← −F1(s_i, −m)
            if t > m then
                m = t;
                if m ≥ bound then
                    break
                    end;
                    F1 = m;
                    end;
                    end.
```

Vediamo che la procedura $F1$ rispetta le richieste dell'equazione 4: all'inizio della $i-$esima iterazione dell'algoritmo del **for** abbiamo la condizione

$$m = max(-F(s_1), ..., -F(s_{i-1}))$$

come nella procedura $F$. Ora, se $-F(s_i) > m$, allora $F_1(s_i, -m) = F(s_i)$, per come sono state definite le procedure. Mentre se $max(-F(s_1), ..., -F(s_i)) \geq bound$ per ogni $i$, allora $F(s) \geq bound$. La condizione 4 è quindi verificata da ogni posizione $s$.
In realtà possiamo fare ancora di meglio introducendo lower ed upper bounds e lavorando quindi *in due direzioni* rispetto al semplice branch and bound. Il miglioramento è noto in letteratura come *alpha-beta pruning* ed è definito da un algoritmo $F2$ a tre parametri (stato $s$ e per l'appunto un certo alpha<beta) che rispettino le seguenti condizioni:

$$\begin{cases} F2(s, alpha, beta) \leq alpha & F(s) \leq \text{ alpha} \\ F2(s, alpha, beta) = F(s) & alpha \leq F(s) \leq beta \\ F2(s, alpha, beta) \geq beta & F(s) \geq beta. \end{cases} \tag{5}$$

Osserviamo che queste condizioni garantiscono il verificarsi di

$$F2(s, -\infty, +\infty) = F(s).$$

Riportiamo infine uno pseudocodice che simula un'applicazione dell'alpha-beta pruning e procediamo poi a confrontare i metodi:

**Algoritmo 3** Alpha-Beta Pruning

```
1:  procedure F2(s, alpha, beta)
2:      inizializza m, i, t, d
3:      determina posizione successive s_1, ..., s_d;
4:      if d = 0 then
5:          F2 = f(s)
6:      else
7:          m = alpha;
8:          for i ← 1, d do
9:              t ← −F2(s_i, −beta, −m)
10:             if t > m then
11:                 m = t;
12:                 if m ≥ beta then
13:                     break
14:                     end;
15:                     F2 = m;
16:                     end;
17:                     end.
18:
```

L'origine del nome dell'algoritmo è dovuta alle due variabili di controllo:

- $\alpha$ è il punteggio minimo che **Max** può raggiungere a partire dalla posizione in cui ci troviamo, all'inizio è posta a $-\infty$. È il valore migliore ("Max ha garantito almeno") che abbiamo trovato fino allo stato $s$.
- $\beta$ è il punteggio massimo che **Min** può raggiungere dalla stessa posizione. All'inizio dell'algoritmo è posto a $+\infty$.

La verifica della validità di $F2$ è analoga a quella dell'algoritmo di Branch-and-Bound; la condizione è adesso

$$m = max(alpha, -F(s_1), ..., -F(s_{i-1}))$$

con l'aggiunta della richiesta $m < beta$. Ora se $-F(s_i) \geq beta$, allora $-F2(s_i, -beta, -m)$ sarà ancora maggiore o uguale di $beta$, e se $m < -F(s_i) < beta$, allora $-F2(s_i, -beta, -m) = -F(s_i)$, e concludiamo per induzione.

**Osservazione 3.2.** In accordo con l'esposizione di Knuth e Moore in [KM75], gli pseudo-codici sono stati espressi utilizzando la convenzione del "negamax" che permette di *ripulire* il codice mediante una singola ricorsione. Si tratta comunque semplicemente di un elemento legato alle convenzioni definite nella sezione 2.1 e tutte le procedure possono essere *tradotte* in linguaggio "minimax". Riportiamo l'esempio di $F2$ che sarà la procedura che andremo ad analizzare più nel dettaglio:

**Algoritmo 4** Alpha-Beta Pruning, notazione "minimax"

```
1: procedure F2(s, alpha, beta)
2:     inizializza m, i, t, d
3:     determina posizione successive s_1, ..., s_d;
4:     if d = 0 then
5:         F2 = f(s)
6:     else
7:         m = alpha;
8:         for i ← 1, d do
9:             t ← G2(s_i, m, beta)
10:            if t > m then
11:                m = t;
12:                if m ≥ beta then
13:                    break
14:                    end;
15:                    F2 = m;
16:                    end;
17:                    end.
18:                    procedure G2(s, alpha, beta)
19:                        inizializza m, i, t, d
20:                        determina posizione successive s_1, ..., s_d;
21:                        if d = 0 then
22:                            G2 = g(s)
23:                        else
24:                            m = alpha;
25:                            for i ← 1, d do
26:                                t ← F2(s_i, alpha, m)
27:                                if t < m then
28:                                    m = t;
29:                                    if m ≤ alpha then
30:                                        break
31:                                        end;
32:                                        F2 = m;
33:                                        end;
34:                                        end.
35:
```

**Osservazione 3.3.** Riportiamo l'equivalenza fra gli algoritmi minimax ed alpha-beta anche con la notazione di *Max/Min*, la quale chiarifica il ruolo speculare delle variabili $\alpha, \beta$:

- Dato uno stato MAX $s_0$ con utilità attuale $U(s_0) \geq \alpha$ per farla crescere si dovrebbe trovare uno stato figlio $s_1$ di tipo MIN per cui valga $U(s_1) > \alpha$; perché ciò sia verificato serve che *tutti* i figli di $s_1$ abbiano utilità maggiore di $\alpha$, quindi appena si trova anche solamente un figlio $s_2$ con $U(s_2) \leq \alpha$ possiamo potare il sottoalbero relativo a $s_1$.
- Dato uno stato MIN $s_0$ con utilità attuale $U(s_0) \leq \beta$ per farla diminuire si dovrebbe trovare uno stato figlio $s_1$ di tipo MIN per cui valga $U(s_1) < \beta$; perché ciò sia verificato serve che *tutti* i figli di $s_1$ abbiano utilità minore di $\beta$, quindi appena si trova anche solamente un figlio $s_2$ con $U(s_2) \geq \alpha$ possiamo potare il sottoalbero con radice $s_1$.

# 4 Spiegazione dell'algoritmo

Le procedure definite nella sezione 3 sono sufficienti a descrivere l'algoritmo di alpha-beta pruning come *evoluzione* del minimax. Nel paragrafo che segue descriviamo applicazioni di tali procedure ed esibiamo esempi di *potature* di rami da alberi di gioco prima di passare ad analisi più dettagliate.
Consideriamo inizialmente il seguente albero di gioco dove i nodi rappresentati da $\Delta$ sono corrispondenti a Max ed i nodi $\nabla$ a Min.

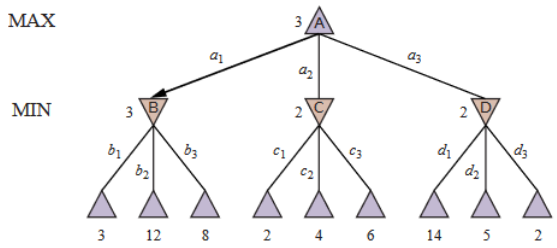



Figure 2: (da [SP22])

La mossa migliore di Max alla radice è $a_1$, in quanto porta ad uno stato con valore massimo, ed è seguita dalla miglior riposta di Min: $b_1$ che minimizza il valore dello stato raggiunto successivamente. Osserviamo che i valori ai nodi sono stati attribuiti ricorsivamente mediante l'utilizzo della funzione $F(s)$; questo fatto esemplifica la natura di algoritmo di ricerca in profondità del minimax e la conseguenziale crescita esponenziale della complessità. Se il nostro albero ha profondità $h$ e vi sono $d$ mosse legali ad ogni punto il nostro algoritmo ha infatti complessità $O(d^h)$; quantità ingestibile da un punto di vista pratico che giustifica il nostro interesse a tagliare, dove possibile, rami dell'albero.
Mostriamo quindi come tramite Alpha-Beta pruning sia possibile evitare la visita di alcuni stati ottenendo comunque un albero di gioco *equivalente.*

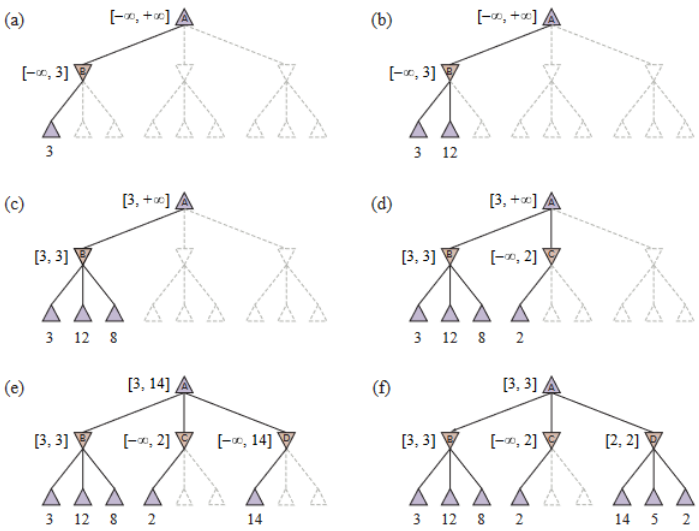



Figure 3: (da [SP22])

A Iniziamo considerando valori $[-\infty, +\infty]$ relativi alla radice dell'albero e consideriamo il primo successore raggiungibile: $B$. Dopo aver esaminato la prima foglia raggiungibile da $B$, questa ha valore 3, modifichiamo i parametri della foglia $B$ ricordando che si tratta di un nodo di Min che avrà, quindi, *al massimo* valore 3.

B La seconda foglia raggiungibile da $B$ ha una valutazione di 12. $B$ non giocherebbe mai questa mossa, il valore del nodo $B$ rimane quindi stabile.

C La terza mossa raggiungibile da $B$ ha una valutazione di 8, nuovamente $B$ non è *interessato* a giocare la posizione. Osserviamo che con questa iterazione abbiamo anche terminato le mosse che seguono da $B$. La radice $A$ ha quindi *almeno* 3 di valutazione in quanto Max ha una tale possibilità di giocata.

D Passiamo adesso ad esaminare $C$. La prima foglia al di sotto di $C$ ha valore 2. Quindi $C$, un nodo relativo a Min, ha valore di *al massimo* 2; ma $B$ ha valore 3, quindi Max non sarà mai interessato a scegliere $C$ e possiamo evitare di controllare gli stati successivi a $C$ operando quindi un primo taglio dell'alpha-beta.

E Passiamo al nodo $D$. La prima foglia successiva ha valore 14, quindi $D$ vale *almeno* 14. Questo valore è più alto del valore provvisorio della radice (attualmente 3), dobbiamo quindi continuare l'esplorazione in quanto $D$ potrebbe essere effettivamente la mossa migliore per le informazioni ottenute fino a questo punto.

F Il secondo successore di $D$ vale 5, ancora più di 3, dobbiamo quindi continuare nella ricerca. Finalmente concludiamo l'esplorazione dell'albero con il 2 dell'ultima foglia, la decisione di Max alla radice è infine di muoversi verso $B$, riottenendo quindi la partita $a_1b_1$ vista con il metodo di minimax.

**Osservazione 4.1.** Il vero pregio della procedura alpha-beta rispetto a $F1$ è la possibilità di *deep-cutoff*, potature in avanti a profondità superiore a 1. Queste permettono di risparmiare potenza di calcolo sia sulle valutazione statiche che sulla generazione di nodi. Il fenomeno si rende *visibile* per alberi con profondità maggiore o uguale a quattro e permettono di di potare zone *considerevoli*.

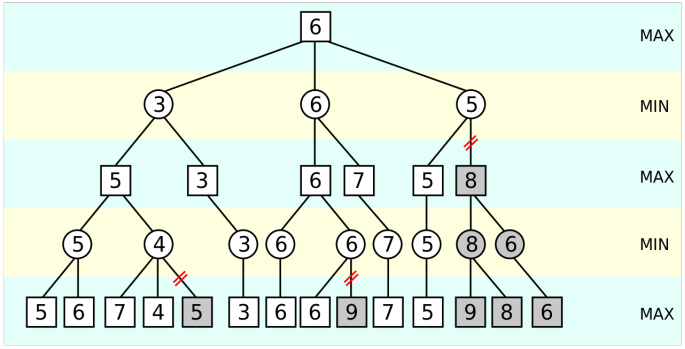



Figure 4: Il taglio *generato* dal secondo successore dell'ultimo nodo del primo livello è un esempio della casisitica di deep-cutoff.

# 5 Applicazioni dell'algoritmo

Dimenticando momentaneamente l'aspetto puramente teorico siamo interessati all'implementazione degli algoritmi tramite calcolatori e alle naturali restrizioni computazionali che emergono nel processo. Tornando al precedente albero di gioco del tic-tac-toe osserviamo come questo sia composto da meno di 9! foglie e quindi ancora esplorabile tramite un semplice algoritmo di brute force, una soluzione impraticabile per giochi di complessità più elevata.
Tale problematica è stata notata sin da Shannon (1950), [Sha50] ("*Programming a Computer for Playing Chess*") autore della prima pubblicazione che affronta l'implementazione al computer di programmi capaci di giocare a scacchi. Per affrontare tale tematica Shannon propose, e distinse, due possibili strategie: una prima, denotata dall'autore come di **tipo A**, considera tutte le mosse possibili fino ad una determinata altezza dell'albero e tramite una *funzione di valutazione euristica* si stima l'utilità degli stati all'altezza prescelta.
La seconda strategia proposta, detta di **tipo B**, consiste invece nell'ignorare mosse che sembrino cattive e studiare *il più possibile* mosse che sembrino essere più promettenti.
L'idea che riassume le due strategie è quindi considerare posizioni *sufficientemente profonde* nell'albero

come stati terminali, la diversificazione avviene nel come determinare uno stato come *similterminale*. Un ulteriore raffinamento della strategia di tipo B (originariamente dovuto a Floyd 1965) può essere operato tramite una funzione di probabilità che diversifica la profondità dei tagli (come esempio concreto nel gioco degli scacchi una mossa obbligata ha probabilità 1, mentre una ricattura avrà una probabilità superiore a quella di un sacrificio di regina in quanto *tendenzialmente* mossa migliore) e possiamo scegliere di considerare come stati terminali quelli raggiunti tramite stati il cui prodotto delle funzioni di probabilità sia inferiore a una certa soglia scelta preliminarmente. Stiamo quindi attribuendo "più importanza" ai rami "più probabili" i quali vengono studiati più in profondità.
Le considerazioni appena espresse introducono immediatamente un nuovo problema: nel momento in cui *studiamo* stati ad altezza arbitraria come terminali non possiamo valutarli semplicemente tramite la nostra $U(s,p)$ e dobbiamo trovare una *funzione di valutazione* che sia capace di produrre un valore da attribuire a suddetti stati, una casistica che verrà presentata più nel dettaglio nella sezione 6.1 dopo aver descritto ulteriori problematiche dell'implementazione di alpha-beta.

**Osservazione 5.1.** Storicamente la maggior parte di programmi scacchistici ricadevano nella categoria di tipo A mentre per il Go la strategia preferita era spesso quella di tipo B (a causa del fattore di branching molto più elevato del secondo gioco rispetto al primo).

# 6 Problematiche di Alpha-Beta

Per quanto sia estremamente più rapido dell'algoritmo *minimax* anche l'alpha-beta non è esente da difetti.
Andiamo a fornire un elenco dei principali problemi dell'algoritmo per poi fornire idee di soluzioni ad alcuni di questi. Più in generale le seguenti tematiche vengono trattate da Junghanns in [And00]

- Errore euristico: come accennato nella sezione 5 alpha-beta necessita di una funzione di valutazione. La funzione di valutazione viene ritenuta *corretta* dalla procedura ma in generale ciò può portare ad errori.
- Espansione: l'alpha beta è un algoritmo DFS (ricerca in profondità) totalmente dominato dai parametri alpha e beta. La non correlazione fra *nodi distanti* impedisce di utilizzare informazioni ottenute in altre parti dell'albero, una caratteristica molto poco flessibile.
- Tempo di arresto: in generale l'algoritmo non ha un criterio di arresto ed una posizione viene esplorata fino a una determinata altezza, il tempo speso per la ricerca è quindi indipendente dalla *qualità* delle mosse, una caratteristica che vorremmo correggere. La proposta di Floyd citata nella sezione 5 risponde teoricamente a questa evenienza ma il modello concreto è fortemente dipendente dal gioco che vogliamo studiare.
- "The Horizon effect": con questo termine ci riferiamo ad un fenomeno che può capitare in giochi con numero di stati *considerevole;* può capitare che mosse preferibili vengano scartate in quanto il loro *effettivo valore* viene rilevato in nodi che sottostanno alla ricerca effettuata (quindi oltre il nostro *orizzonte*, da qui il nome), l'idea, esplorata per primo da Berliner nel 1973, viene spiegata con esempi concreti in 6.3.

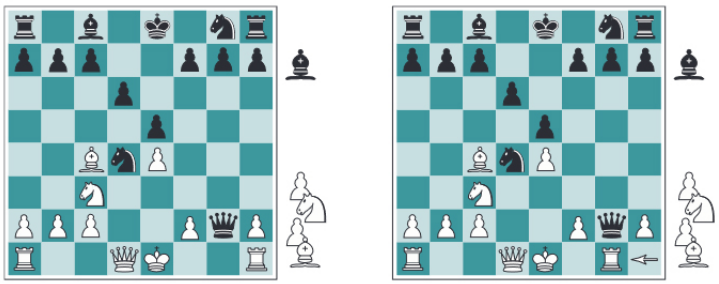
## 6.1 Alpha-Beta euristico

Come citato in precedenza, nella maggior parte delle applicazioni dell'algoritmo, l'esplorazione fino ai nodi terminali dell'albero è computazionalmente intrattabile ed è necessario tagliare l'albero per considerare nodi non-terminali come se lo fossero, e quindi occorre sostituire la funzione Utilità con una *funzione di valutazione*. Considerato che vogliamo tagliare a nodi *arbitrari* il massimo che possiamo richiedere alla nostra funzione di valutazione è di simulare la probabilità di vittoria ad un certo stato. L'effettiva messa a punto di una funzione di valutazione è dipendente dal gioco considerato ma in generale si tendono a selezionare caratteristiche oggettive di uno stato (per esempio negli scacchi il numero di copie di un determinato pezzo presenti sulla scacchiera) e a partizionare gli stati rispetto a relazioni di equivalenza relative alle caratteristiche. A questo punto è necessario attribuire un peso ad ognuna di queste relazioni di equivalenza ed infine un valore atteso alla posizione compreso negli estremi della nostra funzione payoff.

**Osservazione 6.1.** Un esempio molto semplicistico di funzione di valutazione può essere la seguente funzione lineare pesata:

$$Eval(s) = \sum_{i=1}^{n} w_i f_i(s) \tag{6}$$

Nell'esempio scacchistico potremmo considerare le $f_i(s)$ come il numero di pezzi di tipo $i$ presenti allo stato $s$ ed i $w_i$ come i *pesi* (moralmente la "potenza materiale" di un pezzo) da attribuire agli stessi. Un tale tipo di funzione ha però chiari difetti (come mostrato dalla figura) ed in generale rimane complicato capire quali siano le caratteristiche di interesse di uno stato ed i relativi pesi all'interno della posizione; negli ultimi sviluppi della ricerca spesso si utilizzano tecniche di machine learning proprio nella ricerca di tali funzioni di valutazione.



(a) White to move (b) White to move

Figure 5: Le due scacchiere differiscono solamente per la posizione della torre e di conseguenza sono scacchiere equivalenti agli occhi della funzione definita dall'equazione 6. Nella prima scacchiera, però, la posizione del bianco è perdente; nella seconda la cattura della regina da parte della torre conduce a una partita vincente per il bianco (da [SP22])

## 6.2 Stati Quiescenti

Il problema di errore euristico introdotto dall'esempio precedente può essere risolto con un'ulteriore differenziazione fra stati. Definiamo uno stato *quiesciente* se non presenta successori possibili che causino una grande variazione nella funzione di valutazione, a questi applichiamo la procedura di valutazione mentre per uno stato *non-quiesciente* continuiamo la ricerca in profondità fino ad uno stato quiesciente. In generale questa ricerca di nuova tipologia di stato può essere velocizzata venendo ristretta a particolari tipologie di mosse (e.g. negli scacchi possiamo limitarci alle catture, mosse spesso impattanti sul valore di una posizione).

## 6.3 The Horizon Effect

Descriviamo adesso nel dettaglio l'*Horizon Effect* accennato all'inizio della sezione; l'effetto è caratteristico di algoritmi di ricerca in profondità con interruzione programmata. Quest'ultima caratteristica ha infatti il difetto di condurre a valutazioni errate mediante mosse che *ritardino* uno stato quiesciente al di sotto dell'interruzione programmata. Descriviamo un esempio dell'effetto che chiarifichi la casistica.

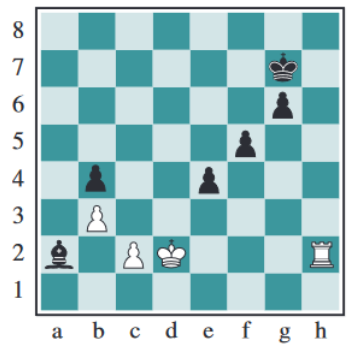


Figure 6: La posizione descritta dall'immagine può ricadere nella casistica dell'horizon effect, se si sceglie un'interruzione programmata dopo un numero basso di mosse (da [SP22])

Nella posizione rappresentata nel precedente diagramma l'alfiere nero è *intrappolato* e a gioco corretto il bianco è in situazione vincente, avendo la possibilità di cattura tramite 3 mosse di torre. Il nero però può essere erroneamente condotto a rendere ancora peggiore la propria situazione: causare uno scacco al re bianco tramite avanzate di pedoni può *spingere* la cattura dell'alfiere oltre la *linea di orizzonte* posta all'inizio della ricerca. L'introduzione di mosse preferibili, e quindi un'interruzione a differenti profondità, tramite un algortimo simile a quello proposto da Floyd, è la soluzione utilizzata nella maggior parte delle applicazioni pratiche dell'alpha-beta.

## 6.4 Forward Pruning

I tagli ad altezza arbitraria dell'alpha-beta possono eventualmente causare tagli erronei di mosse *buone*. Una possibile soluzione rientra nella casistica di strategie di tipo B descritte da Shannon [Sha50], ed è simile alla metodologia di gioco *umana*, si considera solamente le mosse che *sembrano* buone. Un'approccio elementare è *la ricerca a raggio* che considera solamente le $n$ mosse migliori secondo la funzione di valutazione, questa è però una scelta pericolosa in quanto non garantisce di non tagliare la scelta migliore.
Buro, nel 1995, ha sviluppato le idee di Floyd in un *taglio probabilistico* tramite un algoritmo che utilizza statistiche ottenute dalle precedenti esperienze di potatura per ridurre il rischio di tagliare la mossa migliore.
Effettivamente la semplice modifica probabilistica dell'algoritmo effettuata al programma *Logistello* (creato dallo stesso Buro per giocare a Othello) riuscì a battere il programma originale nel 64% delle partite simulate pur giocando con la metà del tempo disponibile.
Per comprendere l'importanza delle decisioni attuate nell'applicazione del forward pruning citiamo il seguente esempio: supponiamo di aver implementato una funzione di valutazione per gli scacchi ed un test *cutoff* legato agli stati quiescenti. A causa dell'elevato fattore di diramazione degli scacchi (circa 35) supponendo di poter generare e valutare circa un milione di nodi al secondo, con l'algoritmo minimax riusciremmo a *guardare avanti* solamente 5 ply al minuto; ottenendo un programma capace di giocare ma in grado di essere battuto da uno scacchista amatoriale.
Tramite l'implementazione di alpha-beta, e tavole di trasposizione, è possibile estendere la ricerca a 14 ply ottenendo un livello di gioco simil-professionale.
I programmi di scacchi più forti attualmente, basati su migliorie di alpha-beta, come *Stockfish* raggiungono fino a 30 ply di profondità e sono in grado di sconfiggere qualsiasi avversario umano.

## 6.5 Ricerca contro Consultazione

Un'ulteriore miglioria pratica ad algoritmi di gioco consiste nel ridurne la necessità di calcolo affidandosi a *tavole di gioco* già compilate. Ripetiamo l'esempio degli scacchi ed osserviamo che la *teoria delle aperture*, termine generico con il quale ci riferisce alle prime *fasi* di gioco, ha storia ultrasecolare e l'utilizzo di questi risultati, unito ad un database di partite passate, può evitare calcoli e velocizzare sensibilmente il programma. Solitamente nell'arco di 10/15 mosse le posizioni sono *novelty*, mai giocate in precedenza, ed i programmi *iniziano* le loro effettive computazioni (per esempio IBM Deep Blue, il primo computer in grado di battere il campione mondiale di scacchi in carica, utilizzava una tablebase

per le aperture). Una tale tipologia di *affidamento* a tablebase diventa ancora più efficiente nei momenti terminali della partita, dove sono stati eliminati pezzi dalla scacchiera e la riduzione di complicazioni permette di *risolvere* il gioco tramite ricerca *all'indietro:* si considerano posizioni terminali del gioco, e tramite analisi all'indietro vengono simulate le partite che possono aver condotto a tale posizione. Attualmente per il gioco degli scacchi sono note, e disponibili per download e consultazione a [Guo18], le tablebase che contengono fino a 7 pezzi.

# 7 Analisi numerica dell'Alpha Beta Pruning

Dopo aver descritto l'idea dietro l'algoritmo ed una sua applicazione nel concreto, tramite l'ausilio di funzioni e stratagemmi euristici, siamo interessati ad un'analisi numerica del costo computazionale dell'algoritmo alpha-beta in diversi scenari; cerchiamo quindi una risposta alla naturale domanda di **quanto** albero si debba effettivamente esplorare tramite l'alpha-beta.

## 7.1 Analisi del Best Case

Prima di enunciare proposizioni stabiliamo una convenzione per assegnare coordinate ad ogni nodo dell'albero. In accordo con [KM75] utilizziamo la "classificazione decimale Dewey". Ad ogni posizione al livello $l$ viene assegnata una successione di interi $a_1a_2, ..., a_l$. Il nodo radice corrisponde alla successione vuota, ed i $d$ successori della posizione $a_1, ..., a_l$ vengono codificati con le successioni: $a_1, ..., a_l, 1$; $a_1, ..., a_l, 2$; $a_1, ..., a_l, d$ (elideremo le virgole dalla notazione di Dewey dove questo non vada a causare confusione).

**Definizione 7.1.** Diciamo che una posizione $a_1...a_l$ è critica se $a_i = 1$ per ogni valore pari o per ogni valore dispari di $i$.

L'interesse della definizione è da ricercare nel seguente teorema che considera la casistica fortunata in cui la *prima* mossa disponibile sia sempre la migliore:

**Teorema 7.1.** *Dato un albero di gioco il cui valore alla radice sia diverso da $\pm\infty$, e per il quale il primo successore di ogni posizione sia ottimale, i.e.*

$$F(a_1...a_l) = \begin{cases} f(a_1...a_l) & \text{se } a_1...a_l \text{ è terminale,} \\ -F(a_1...a_l1) & \text{alternativamente} \end{cases} \tag{7}$$

*Allora la procedura alpha-beta F2 esamina precisamente le posizioni critiche dell'albero di gioco.*

*Proof.* Consideriamo una posizione critica $a_1...a_l$ , diremo che questa è di tipo 1 se $a_i = 1\ \forall i$, di tipo 2 se data $a_j$ prima entrata $> 1$ vale che $l - j$ sia pari o di tipo 3 altrimenti. Vista la struttura di $F2$ valgono le seguenti affermazioni:

1. Una posizione $s$ di tipo 1 viene esaminata *invocando* $F2(s,\ -\infty,\ +\infty)$. Se lo stato non è terminale allora la posizione successiva $s_1$ è critica di tipo 1 e $F(s) = -F(s_1) \neq \pm\infty$. Le altre posizioni successive $s_2, ..., s_d$ sono di tipo 2 e vengono tutte esaminate richiamando $F2(s_i,\ -\infty,\ F(s_1))$.

2. Una posizione $s$ di tipo 2 è esaminata per via di $F2(s,\ -\infty,\ beta)$, con le condizioni $-\infty < beta \leq F(s)$. Se non è terminale, il suo successore $s_1$ è di tipo 3, vale $F(s) = -F(s_1)$ e quindi per la definizione di $F2$, i successori $s_2, ..., s_d$ non sono esaminati.

3. Una posizione $s$ di tipo 3 viene esaminata nel caso in cui si richiami $F2(s,\ alpha,\ +\infty)$ con la condizione $\infty > alpha \geq F(s)$. Se non è terminale ogni stato successivo a $s_l$ è di tipo 2 e viene esaminato per mezzo di $F2(s_i,\ -\infty,\ -alpha)$.

Per induzione su $l$ otteniamo che ogni posizione critica viene raggiunta ed esaminata. Otteniamo anche il seguente corollario: □

**Corollario 7.1.** *Se ogni posizione ai livelli 0,1,...,l-1 di un albero di gioco che soddisfa le condizioni del teorema 7.1 ha esattamente d successori, per d fissato, allora la procedura alpha-beta esamina esattamente*

$$d^{\lfloor l/2 \rfloor} + d^{\lceil l/2 \rceil} - 1 \tag{8}$$

*Proof.* Esistono esattamente $d^{\lfloor l/2 \rfloor}$ successioni $a_1, ..., a_l$ con $1 \leq a_i \leq d$ per ogni $i$, tale che $a_l = 1$ per ogni valore dispari di $i$; esistono invece $d^{\lceil l/2 \rceil}$ successioni con $a_i = 1$ per ogni $i$ pari, dobbiamo togliere 1 al conto per la successione $1, ..., 1$ costituita solamente da $l$ ripetizioni di 1 che è stata contata due volte. □

Possiamo estendere il risultato ottenuto per un certo $d$ (numero di mosse fissato) ad un caso più generale considerando una distribuzione di probabilità che *conti* il numero di mosse disponibili al livello $l$ :

**Corollario 7.2.** *Consideriamo un albero di gioco tale per cui ogni posizione al livello $j$ abbia probabilità $q_j$ di essere non-terminale, con media di $d_j$ successori. Allora il numero atteso di posizioni al livello $l$ è $d_0 d_1 ... d_{l-1}$; ed il numero atteso di posizioni esaminate dall'alpha-beta in accordo con le ipotesi di 7.1 è:*

$$d_0 q_1 d_2 q_3 ... d_{l-2} q_{l-1} + q_0 d_1 q_2 d_3 ... q_{l-2} d_{l-1} - q_0 q_1 ... q_{l-1} \text{ se } l \text{ è pari;} \tag{9}$$

$$d_0 q_1 d_2 q_3 ... q_{l-2} d_{l-1} + q_0 d_1 q_2 d_3 ... d_{l-2} q_{l-1} - q_0 q_1 ... q_{l-1} \text{ con } l \text{ dispari.} \tag{10}$$

**Osservazione 7.1.** Prima di procedere alla dimostrazione del corollario una precisazione sulle ipotesi di diramazione casuale. Supponiamo che il livello $j+1$-esimo dell'albero sia formato a partire dal $j-$esimo nel modo che segue: ad ogni stato $s$ al livello $j-$esimo è assegnata una distribuzione di probabilità $< r_0(s), r_1(s), ... >$, dove $r_d(s)$ è la probabilità che $s$ abbia $d$ successori. Le distribuzioni possono differire al variare dello stato $s$, ma devono soddisfare $r_0(s) = 1 - q_j$, ed ognuna deve avere valore atteso $r_1(s) + 2r_2(s) + ... = d_j$. Il numero di successori di $s$ è scelto quindi in maniera casuale da questa distribuzione, in maniera indipendente dalle altre posizioni che condividono il livello di $s$.

*Proof.* Sia $x$ il numero atteso di posizioni di un certo tipo al livello $j$, chiamiamo $xd_j$ il numero atteso di successori di tali posizioni, e con $xq_j$ il valore atteso di posizioni di "numeri 1" successori. Segue allora dal corollario 7.1 la veridicità delle equazioni dell'enunciato; per esempio $q_0 q_1 ... q_{l-1}$ è il numero atteso di posizioni al livello $l$ con coordinate tutte 1. □

**Osservazione 7.2.** In generale potrebbe valere l'idea intuitiva secondo la quale alpha-beta pruning raggiunga la massima efficienza con ipotesi di ordine crescente nel sottoalbero di interesse. In realtà, come mostra il controesempio che segue, la assunzione è falsa.

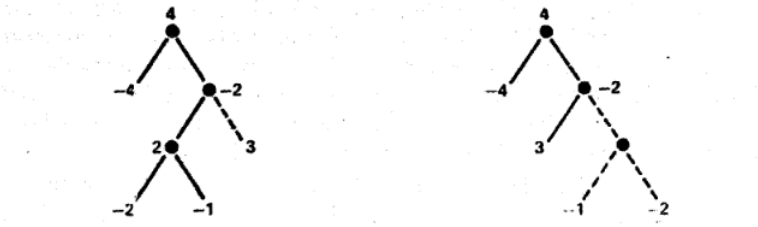



Figure 7: Un esempio che mostra come l'ordinamento dell'albero influisca sulle potature. Per questo, prima di procedere con alpha-beta, spesso si agisce sull'albero tramite un algoritmo di riordinamento. (da [KM75])

In generale non è quindi banale decidere quale sia l'ordine ottimale da attribuire agli stati per ridurre il costo computazionale di alpha-beta.
Quello che si può invece dimostrare è l'esistenza dell'ordine ottimale che renda alpha-beta l'algoritmo *migliore* (rispetto al numero di posizioni terminali esaminate) come precisato nel seguente teorema:

**Teorema 7.2.** *Alpha-beta è ottimale nel senso che segue: dato un qualsiasi albero di gioco ed un qualsiasi algoritmo che computa il valore della radice, esiste sempre una permutazione dell'albero (che eventualmente riordina i successori di date posizioni) tale per cui ogni stato terminale esaminato da alpha-beta rispetto a questa permutazione sia esaminato anche dal dato algoritmo. In generale se il valore della radice non è $\pm\infty$, l'alpha-beta esamina esattamente le posizioni che sono critiche rispetto a questa permutazione di riordinamento.*

*Proof.* Come prima cosa definiamo due funzioni $F_l, F_u$ con lo scopo di misurare i migliori bound sul valore di uno stato $s$, basandoci sulle posizioni terminali esaminate dall'algoritmo dato:

$$F_l(s) = \begin{cases} -\infty & \text{se } s \text{ è terminale e non esaminato} \\ f(s) & \text{se } s \text{ è terminale e viene esaminato} \\ max(-F_u(s_1), ..., -F_u(s_d)) & \text{alternativamente} \end{cases} \tag{11}$$

$$F_u(s) = \begin{cases} +\infty & \text{se } s \text{ è terminale e non esaminato} \\ f(s) & \text{se } s \text{ è terminale e viene esaminato} \\ max(-F_l(s_1), ..., -F_l(s_d)) & \text{alternativamente} \end{cases} \tag{12}$$

In generale vale $F_l(s) \leq F_u(s)$ per ogni $s$. Facendo variare in maniera indipendente i valori nelle posizioni terminali al di sotto di $s$, possiamo far assumere a $F(s)$ ogni valore compreso fra i due bound, ma mai eccedere nessuno dei due. Di conseguenza quando $s$ è lo stato radice deve valere $F_l(s) = F_u(s) = F(s)$. Assumendo che questo valore sia diverso da infinito, in modulo, facciamo vedere come permutare l'albero così che ogni posizione terminale critica (rispetto al nuovo ordinamento) venga esaminata dall'algortmo dato e che precisamente le posizione critiche siano quelle esaminate da alpha-beta secondo $F2$. In accordo con le defizioni sul *tipo* di posizioni critiche date in 7.1 valgono le seguenti:

1. Una posizione $s$ di tipo 1 è tale per cui $F_l(s) = F_u(s) = F(s) \neq \pm\infty$, e viene esaminata da alpha-beta via $F2$ con argomento $(s, -\infty, +\infty)$. Se $s$ è terminale deve venire esaminato dall'algoritmo dato che $F_l(s) \neq -\infty$. Se non lo è siano $j, k$ indici tali per cui $F_l(s) = -F_u(s_j)$ e $F_u(s) = -F_l(s_k)$. Allora per le definizioni delle funzioni deve valere:

   $$F_l(s_k) \leq F_l(s_j) \leq F_u(s_j) = -F(s) = F_l(s_k)$$

   quindi $F_l(s_j) = F_l(s_k)$ e possiamo supporre $j = k$. Permutando stati successivi possiamo assumere $j = k = 1$. Ora la posizione $s_1$ è di tipo 1; le successive $s_2, ..., s_d$ sono di tipo 2, e vengono esaminate da $F2(s_i, -\infty, -F(s_1))$.

2. Una posizione $s$ di tipo 2 è tale per cui $F_l(s) > -\infty$, e viene esaminata durante alpha-beta tramite $F2(s, -\infty, beta)$, con $-\infty < beta \leq F_l(s)$. Se $s$ è terminale deve venire esaminato dall'algoritmo dato. Alternativamente sia $j$ tale che $F_l(s) = F_u(s_j)$, e permutiamo i successori sino ad avere $j = 1$. Ora $s_1$ è di tipo 3 e viene esaminata da $F2(s_1, -beta, +\infty)$. Dato che $F_u(s_1) = -F_l(s) \leq -beta$, $F2$ *ritorna* un valore inferiore o uguale a $-beta$; di conseguenza i successori $s_2, ..., s_d$ (che non sono posizioni critiche) non vengono esaminate da alpha-beta ma *potate*, evitando quindi anche gli stati che vi discendono.

3. Uno stato $s$ di tipo 3 ha $F_u(s) < \infty$, e viene esamianto durante $F2(s, alpha, +\infty)$ dove $F_u(s) \leq alpha < \infty$. Se $p$ è terminale deve venire esaminato dall'algoritmo. Se non lo è ogni successore $s_i$ è di tipo 2, e viene esaminato da $F2(s_i, -\infty, -alpha)$. (Qua l'ordine di esplorazione non compie nessuna differenza.)

□

**Osservazione 7.3.** Nel corso della dimostrazione abbiamo osservato come l'ordinamento dei successori di una posizione di tipo 3 in un albero ottimale non abbia nessuna influenza su alpha-beta. Posizioni di tipo 1 sono spesso chiamate *varianti principali*, e corrispondono alla miglior strategia da parte di entrambi i giocatori. Le *risposte alternative* a mosse principali coincidono con le posizioni di tipo 2, mentre una posizione di tipo 3 accade quando la migliore mossa viene eseguita da uno stato di tipo 2.

Cosideriamo la seguente definizione relativa ad alberi che ci consente di enunciare un nuovo corollario di 7.2:

**Remark:** Un albero di gioco è detto uniforme di grado $d$ ed altezza $h$ se ogni posizione ai livelli $0, 1, ..., h-1$ ha esattamente $d$ successori ed ogni posizione al livello $h$ è terminale.

**Osservazione 7.4.** Ogni permutazione di un albero uniforme è nuovamente un albero uniforme. Questo permette di enunciare:

**Corollario 7.3.** *Ogni algoritmo che valuti un albero di gioco uniforme di altezza $h$ e grado $d$ deve valutare almeno*

$$d^{\lfloor h/2 \rfloor} + d^{\lceil h/2 \rceil} - 1$$

*posizioni terminali. La procedura alpha-beta realizza questo lower bound se la mossa migliore viene considerata ad ogni posizione di tipo 1 e 2.*

## 7.2 Alberi uniformi senza tagli profondi

Abbiamo studiato fino ad adesso il best case realizzabile con alpha-beta pruning. Cerchiamo adesso di stimare il caso pessimistico. Dato un albero finito è sempre possibile trovare una successione di valori per le posizioni terminali tali per cui alpha-beta debba visitare ogni nodo dell'albero senza mai potare nessun ramo.

**Osservazione 7.5.** Questo succede se i valori sono disposti in maniera tale per cui ad ogni *invocazione* di $F2(s, alpha, beta)$ si abbia $-alpha > F(s_1) > F(s_2) > ... > F(s_d) > -beta$.

In applicazione reali, però, alpha-beta pruning viene sempre utilizzato in maniera complementare ad una qualche strategia di ordinamento dell'albero di gioco ed il worst case descritto in precedenza è quindi di ridotto interesse pratico. Cerchiamo, dunque, un upper bound nel caso più rilevante che potremmo ipotizzare in un esperimento reale: supponiamo quindi di avere dei dati completamente casuali.
Osservando che siamo interessati ad un upper bound, e non a disuguaglianze *sharp*, per semplificare i calcoli, consideriamo la procedura $F1$, la cui *debolezza* rispetto all'alpha-beta pruning consiste nel non rilevare nessun *taglio profondo.*

Supponiamo quindi di considerare un albero uniforme di grado $d$ ed altezza $h$ costruito con valori assegnati in maniera causale alle $d^h$ posizioni terminali e definiamo $T(d, h)$ il numero atteso di posizioni terminali esaminate dalla procedura $F1$ quando applicata all'albero.
Come prima cosa osserviamo che la procedura dipende solamente dall'ordine relativo delle valutazioni e non dai loro valori assoluti, possiamo quindi assumere che questi siano permutazioni dell'insieme $\{1, 2, ..., d^h\}$ ed ogni permutazione venga assunta con probabilità $\frac{1}{(d^h)!}$. Diretta conseguenza di questo fatto è la casualità dell'ordine dei $d^l$ valori associati alle posizioni ad altezza $l$ per ogni $0 \leq l \leq h$. Ancora di più, dato che i tagli effettuati da $F1$ dipendono da $F$, possiamo estendere l'analisi ad ogni livello $l$ e supporre che il numero atteso di posizioni valutate sia $T(d, l)$ per ogni $l$.
Possiamo quindi restringerci a studiare un singolo livello $h$.

**Teorema 7.3.** *Il numero atteso di posizione terminali esaminate dalla procedura alpha-beta senza tagli profondi, in un albero di gioco casuale e uniforme di grado $d$ e altezza $h$, soddisfa*

$$T(d, h) < c^*(d) r^*(d)^h, \tag{13}$$

*dove $c^*(d)$ è una costante e $r^*(d)$ il più grande autovalore della matrice: $M_d^* = (\sqrt{p_{ij}})$ al variare di $i, j$ da 1 a $d$. Dove i $p_{ij}$ sono la probabilità che valga*

$$\max_{1 \leq k < 1} (\min(Y_{k1}, ..., Y_{kd})) < \min_{1 \leq k < i} Y_{ik} \tag{14}$$

*in una successione di $(i-1)d + (j-1)$ i.i.d. variabili aleatorie $Y_{11}, ..., Y_{i(j-1)}$.*

**Osservazione 7.6.** Prima di procedere alla dimostrazione alcune osservazioni sulle probabilità introdotte in 14. Se $i = 1$ o $j = 1$, la probabilità considerata è proprio 1, con la convenzione che min e max di un insieme vuoto siano rispettivamente $+\infty$ e -$\infty$. Con $i, j > 1$ osserviamo che perché valga la condizione 14 è necessario che il minimo delle $Y_m$ sia $Y_{k_1 t_1}$ per qualche $k_1 < i$, e questo accade con probabilità $(i-1)d/((i-1)d + j - 1)$; rimuovendo ora $Y_{k_1 1}, ..., Y_{k_1 d}$ dalla considerazione, il minimo delle $Y_m$ rimanenti deve essere della forma $Y_{k_2 t_2}$ per qualche $k_2 < i$, questo succede con probabilità: $(i-2)d/((i-2)d + j - 1)$ e così via, possiamo ripetere argomento analogo per $m \in \{1, ..., d\}$.
Quindi la condizione 14 è verificata con probabilità:

$$p_{ij} = 1 / \binom{i - 1 + (j-1)/d}{i-1} \tag{15}$$

il che permette approssimazioni numeriche.

*Proof.* Assegniamo coordinate all'albero di gioco seguendo nuovamente la classificiazione decimale Dewey. Per $l \geq 1$, si dimostra induttivamente che la posizione $a_1...a_l$ ha $bound = min\{F(a_1...a_lk) \mid 1 \leq k < a_l\}$ quando esaminata da $F1$; e quindi viene esaminata se e solo se viene esaminata $a_1...a_{l-1}$ e si verifica

$$- min_{1 \leq k < a_l} F(a_1...a_{l-1}k) < min_{i \leq k < a_{l-1}} F(a_1...a_{l-2}k) \; o \; l = 1. \tag{16}$$

Segue quindi che una posizione terminale $a_1...a_h$ viene esaminata da $F1$ se e solo se la relazione dell'equazione 16 (da adesso abbreviata in $P_l$) si verifica per ogni $l$ compreso fra 1 e $h$. Ora, la condizione $P_l$ è valida per $l \geq 2$ con probabilità $p_{ij}$, con $i = a_{l-1}$ e $j = a_l$ per come è stata definita la probabilità in precedenza. Ora $P_l$ è una funzione che dipende dai valori terminali

$$f(a_1...a_{l-2} \; j \; k \; a_{l+1}...a_h),$$

dove $j < a_{l-1}$ oppure $j = a_{l-1}$ e $k < a_l$. Quindi $P_l$ è indipendente da $P_1, P_2, ..., P_{l-2}$. Sia ora $x$ la probabilità che la posizione $a_1...a_h$ venga esaminata, e assumiamo per convenienza di notazione $h$ dispari. Allora, la parziale indipendenza delle $P_l$ ci fornisce

$$\begin{aligned} x &< p_{a_1a_2}p_{a_3a_4}...p_{a_{h-2}a_{h-1}} \\ x &< p_{a_2a_3}p_{a_4a_5}...p_{a_{h-1}a_h}; \end{aligned} \tag{17}$$

e quindi

$$x < \sqrt{p_{a_1a_2}p_{a_3a_4}...p_{a_{h-2}a_{h-1}}}$$

e il teorema segue scegliendo $c^*(d)$ grande abbastanza. □

Cerchiamo adesso una stima asintotica della crescita del fattore di diramazione della procedura $F1$ :

**Teorema 7.4.** *Il numero atteso T(d,h) di posizioni terminali visitate dalla procedura alpha-beta senza tagli profondi, in un albero di gioco casuale e uniforme di grado d e altezza h, ha un fattore di diramazione*

$$lim_{n \to \infty} T(d,h)^{1/h} = r(d) \tag{18}$$

*che soddisfa la seguente relazione per certe costanti positive* $C_3, C_4$

$$C_3 \frac{d}{logd} \leq r(d) \leq C_4 \frac{d}{logd} \tag{19}$$

*Proof.* In generale vale:

$$T(D, h_1 + h_2) \leq T(d, h_1)T(d, h_2)$$

in quanto il numero a destra è il numero di posizioni che vengono visitate da $F1$ se $bound = \infty$ per ogni posizione ad altezza $h_1$. In generale questo dimostra le seguenti relazioni:

$$\liminf_{h \to \infty} \; T(d,h) \geq r_0(d); \; \limsup_{h \to \infty} \; T(d,h) \leq r_1(d), r^*(d). \tag{20}$$

che unito ad un argomento classico sulle funzioni sub-additive assicura l'esistenza del limite che compare nell'enunciato del teorema.
Per provare la disuguaglianza vogliamo vedere che $r_0(d) \geq C_3 \frac{d}{logd}$. Ora il modulo del più grande autovalore della matrice a entrate positive $p_{ij}$ è maggiore o uguale del $min_i \; \sum_j p_{ij}$ per un teorema di Perron. Quindi per 9.1 valgono

$$r_0(d) \geq C \min_{1 \leq i \leq d} (\sum_{1 \leq j \leq d} i^{-(j-1)/d}) = C \min_{2 \leq i \leq d} (\frac{1 - i^{-1}}{1 - i^{-1/d}}) = C \frac{1 - d^{-1}}{1 - d^{-1/d}} > C \frac{d-1}{ln(d)}, \tag{21}$$

dove abbiamo $C = 0.885603 = \inf_{0 \leq x \leq 1} \Gamma(1 + x)$, dato che $d^{-1/d} = exp(-ln(d)/d) > 1 - ln(d)/d$. Per ottenere la stima dall'alto dell'equazione 19, dobbiamo mostrare che $r^*(d) < C_4 d/logd$, e per farlo utilizziamo una norma matriciale. Siano $s, t$ esponenti coniugati ($s, t \in \mathbb{R}^+$ tali per cui $\frac{1}{s} + \frac{1}{t} = 1$), allora gli autovalori $\lambda$ della matrice $A = (a_{ij})$ soddisfano:

$$|\lambda| \leq (\sum_i (\sum_j (|a_{ij}^t|^{s/t})^{1/s}. \tag{22}$$

Per dimostrarlo, sia $Ax = \lambda x$, con $x \neq 0$; per la disuguaglianza di Hölder:

$$|\lambda|(\sum_i(|x_i^3|)^{1/s} = (\sum_i(\sum_j |a_{ij}x_j|^s)^{1/s}) \leq (\sum_i(\sum_j |a_{ij}^t|)^{s/t}(\sum_j |x_j^s|))^{1/s} = (\sum_i(\sum_j |a_i j^t|)^{s/t})^{1/s}(\sum_i |x_i^s|)^{1/s} \tag{23}$$

da cui segue 22.
In generale se consideriamo $s = t = 2$ la disuguaglianza 22 implica $r^*(d) = O(d/\sqrt{logd})$, mentre facendo tendere $s$ (o simmetricamente $t$) a $+\infty$ otteniamo $O(d)$ come upper bound. La scelta di $s$, $t$ ottimali è quindi cruciale; definiamo $f(d) = \frac{1}{2}ln(d)/ln(ln(d))$, e poniamo $s = f(d)$ e $t = f(d)/(f(d)-1)$. Così facendo otteniamo:

$$r^*(d) \leq (\sum_{1\leq i\leq d}(\sum_{1\leq j\leq d} i^{-t(j-1)/2d})^{s/t})^{1/s} < (\sqrt{d}d^{s/t} + (d-\sqrt{d})(\sum_{j\geq 1}\sqrt{d^{-t(j-1)/2d}})^{s/t})^{1/s}. \tag{24}$$

Ora la somma più interna è $g(d) = 1/(1-d^{-t/4d}) = (4d/ln(d))(1+O(lnln(d)/ln(d))$, da cui

$$dg(d)^{s/t} = d^{f(d)-1/2}exp(\frac{1}{2}ln(4)ln(d)/ln(ln(d)) + ln(ln(d)) + O(1)).$$

□

Quindi la maggiorazione in 24 è proprio

$$exp(ln(d) - ln(ln(d)) + ln(4) + O((ln(ln(d))^2/ln(d));$$

e segue

$$r^*(d) \leq (4d/ln(d))(1 + O((ln(ln(d))^2/ln(d)) \; con \; d \to \infty.$$

**Osservazione 7.7.** Abbiamo quindi dimostrato che le quantità $r_0(d)$ e $r^*(d)$ crescono come $d/log(d)$, mentre $r_1(d)$ come $d/\sqrt{log(d)}$. In realtà, in accordo con tavole numeriche consultabili in [KM75] $r_1(d)$ è una stima preferibile per $d \leq 24$.

## 7.3 Discussione del modello

Abbiamo trovato un *upper bound* per il nostro modello teoretico, questa quantità non è *sharp* per le seguenti ragioni:

- non abbiamo considerato tagli profondi;
- abbiamo supposto ordinamento delle posizioni successive completamente casuale;
- abbiamo supposto che tutti i valori dei nodi terminali fossero distinti;
- abbiamo supposto che i nodi terminali fossero tutti indipendenti due a due.

Cerchiamo di dare almeno una stima del *guadagno* portato da tagli più profondi, per fare questo sviluppiamo considerazioni supponendo ordine ideale per quanto riguarda i nodi successori; nel verificarsi di tale evenienza le posizioni $a_1...a_n$ esaminate da $F1$ sono tutte e sole quelle in cui $a_i > 1$ implica $a_{i+1} = 1$. Nel caso di albero ternario ($d = 3$) definiamo i seguenti tipi di posizione:

- La radice è una posizione di tipo A.
- Ogni primo successore di una posizione non terminale è di tipo A.
- Ogni secondo successore di una posizione non terminale è di tipo B.
- Ogni terzo successore di una posizone non terminale è di tipo C.

Con queste definizioni la condizione precedente si traduce nelle seguenti ricorsioni:

$$\begin{aligned} A_0 = B_0 = C_0 = 1, \\ A_{n+1} = A_n + B_n + C_n, \\ B_{n+1} = A_n, \\ C_{n+1} = A_n, \end{aligned} \tag{25}$$

per cui abbiamo $A_{n+1} = A_n + 2A_{n-1}$. Mentre per $d$ generale vale la relazione:

$$A_0 = 1, A_1 = d, A_{n+2} = A_{n+1} + (d-1)A_n. \tag{26}$$

Ricorsione risolta da

$$A_n = \frac{1}{\sqrt{(4d-3)}}((\sqrt{(d-\frac{3}{4})} + \frac{1}{2})^{n+2} - (-\sqrt{(d-\frac{3}{4})} + \frac{1}{2})^{n+2});$$

otteniamo quindi che il tasso effettivo di crescita del fattore di diramazione è $\sqrt{(d-\frac{3}{4})} + \frac{1}{2}$, non molto di più della radice di $d$ ottenuta con il metodo che include i cutoff.

## 7.4 Nodi terminali dipendenti

Nel modello abbiamo anche supposto totale indipendenza fra i valori degli stati terminali, una relazione che accade raramente in giochi reali. Se ad esempio ci riferiamo al gioco degli scacchi è lecito attendersi che tutte le posizione che seguono da un errore (ie. la perdita di un pezzo) abbiano un valore di utilità basso per il giocatore che ha subito la cattura. Consideriamo quindi un modello di *dipendenza totale,* cioè tale che per ogni $i, j$, tutti i nodi terminali successori di $p_i$ abbiano tutti valore più grande (o tutti più piccolo) di tutti gli stati terminali che seguono da $p_j$. Tale assunzione è equivalente ad assegnare una permutazione $\{0, 1, ..., d-1\}$ alle mosse da ogni posizione, e poi considerare la concatenazione di tutte i numeri associati alle mosse che conducono ad una posizione terminale come valore di tale posizione.

**Teorema 7.5.** *Il numero atteso di posizioni terminali esaminate dalla procedura alpha-beta, in un albero di gioco totalmente dipendente, casuale, uniforme di grado d e altezza h è:*

$$\frac{d - H_d}{d - H_d^2}(d^{\lceil h/2 \rceil} + H_d d^{\lfloor h/2 \rfloor} - H_d^{h+1} - H_d^h) + H_d^h \tag{27}$$

*dove* $H_d = \sum_{i=1 \to d} \frac{1}{d}$

*Proof.* Iniziamo partizionando gli stati dell'albero in tre categorie differenti. Diremo che una posizione $s$ è *tipo 1* se viene esaminata da $F2(s,\ alpha,\ beta)$ e tutti gli stati terminali emanati da $s$ hanno $alpha \leq \pm f(q) < beta$ (il più o il meno dipende dalla parità del livello dell'albero contando dal basso). Se $s$ non è terminale attribuiamo un ordinamento ai successori e diremo che $s_i$ è *rilevante* se $F(s_i) \leq F(s_j)$ per ogni $1 \leq j \leq i$. La condizione implica che tutti i successori rilevanti di $s$ vengano esaminati da $F2(s_i,\ -beta\ -m)$ dove $F(s_i)$ è compreso fra $-beta$ e $-m$, quindi gli stati rilevanti sono nuovamente di tipo 1. Gli stati non-rilevanti vengono esaminati da $F2(s_i,\ -beta\ -m)$ con $F(s_i) > -m$, e li definiamo come di *tipo 2.*
Una posizione di tipo 2 $s$ viene quindi esaminata da $F2$ se i nodi terminali $q$ discendenti da $s$ soddisfano $\pm f(q) > beta$. Se $s$ è non terminale, il primo successore $s_1$ viene classificato come di *tipo 3*, e viene esaminato da $F2(s_1\ -beta,\ -alpha)$, una procedura di questo tipo *ritornerà* un valore $\leq -beta$ causando quindi un taglio.
Una posizione di tipo 3 viene visitata per tramite di $F2$ se i nodi terminali $q$ discendenti da $s$ soddisfano $\pm f(q) < alpha$. Se $s$ è non terminale i successori sono classificati come di tipo 2, vengono esaminati da $F2(s_i,\ -beta\ -alpha)$ e ritornano valori $\geq -alpha$.
Definiamo ora $A_n,\ B_n\ C_n$ il numero atteso di posizioni terminali esaminate in un albero di gioco come nelle ipotesi del teorema con radice rispettivamente di tipo 1, 2, 3. Allora per quanto argomentato valgono le seguenti relazioni:

$$\begin{aligned} A_0 = B_0 = C_0 = 1, \\ A_{n+1} = A_n + (\frac{1}{2}A_n + \frac{1}{2}B_n) + ... + (\frac{1}{d}A_n + \frac{d-1}{d}B_n) = H_d A_n + (d - H_d)B_n, \\ B_{n+1} = C_n, \\ C_{n+1} = dB_n; \end{aligned} \tag{28}$$

Risolvendo la ricorrenza otteniamo $B_n = d^{\lfloor h/2 \rfloor}$ e $A_k$ soddisfa l'enunciato. □

**Corollario 7.4.** *Se $d \geq 3$, il numero medio di posizioni esaminate da alpha-beta sotto ipotesi di totale dipendenza è limitato da una costante (dipendente dal grado d, ma non dall'altezza dell'albero) moltiplicata per il numero ottimale di posizioni del 7.3*

*Proof.* L'ordine di crescita in 27 per $\to \infty$ è nell'ordine di $d^{h/2}$. La costante che soddisfa le relazioni è approsimativamente:

$$(d - H_d)(1 + H_d)/2(d - H_d^2).$$

□

# 8 Breve storia sulla genesi dell'algoritmo

Concludiamo adesso con un breve excursus storico sullo sviluppo dell'alpha-beta.
Nonostante sia ad oggi impossibile attribuire ad un singolo la piena paternità dell'algoritmo, John McCarthy [McC06] nel 1956, propose un'idea di algoritmo simile all'alpha-beta (seppur senza nessuna specifica formale) come *critica* ad un primo programma capace di giocare a scacchi sviluppato da Bernstein.
La prima pubblicazione con esempi di *potature* è dovuta a Newell, Shaw e Simon (1958) ma non tratta di *deep-cutoff* ed è paragonabile all'algoritmo della procedura $F1$.
Sempre McCarthy coniò il nome di *alpha-beta pruning* per algoritmi di potatura in due direzioni, portando avanti il lavoro di Hart e Edwards (1961).
Contemporaneamente, ed in maniera indipendente, algoritmi paragonabili all'alpha-beta pruning vennero sviluppati in Russia. Brudno (1963) descrive, e dimostra la validità, di un algoritmo equivalente in [Bru63].
Slagle, Bursky, Dixon e Samuel (1968-69) furono i primi a portare l'alpha-beta pruning completo in letteratura anglossasone.
Knuth e Moore (1975) rifinirono l'algoritmo, dimostrandone l'efficacia e calcolando bound di efficienza.
Judea Pearl ha esteso il lavoro in direzione di alberi con dati casuali (1980) mentre Saks e Wigderson (1986) hanno dimostrato *l'ottimalità* dell'algoritmo sempre in condizione di alberi casualmente ordinati.

# 9 Appendice

Riportiamo brevemente la dimostrazione del teorema che abbiamo utilizzato nella sezione 7.2:

**Teorema 9.1.** *Se $0 \leq x \leq 1$ e $k$ è un intero positivo,*

$$k^x \leq \binom{k-1+x}{k-1} \leq k^x/\Gamma(1+x) \tag{29}$$

*(Osserviamo che $0.885603 < \Gamma(1+x) \leq 1$ per $0 \leq x \leq 1$, con il minimo in $x = 0.461632$; quindi la formula per $k^x$ rientra sempre nell'11% del valore del coefficiente binomiale.)*

*Proof.* Quando $0 \leq x \leq 1$ e $t > -1$ vale:

$$(1+t)^x \leq 1 + tx, \tag{30}$$

dato che la funzione $f(x) = (1+t)^x/(1+tx)$ soddisfa $f(0) = f(1) = 1$, e dato che:

$$f''(x) = ((ln(1+t) - t/(1+tx))^2 + t^2/(1+tx)^2)f(x) > 0.$$

Utilizzando 30 con $t = 1, \frac{1}{2}, \frac{1}{3}, \ldots$ vale:

$$1 \leq \frac{1+x}{2^x} \leq \frac{1+x}{2^x} \cdot \frac{1+\frac{1}{2}x}{(\frac{3}{2})^x} \leq \ldots$$

$$\leq \lim_{m\to\infty} \frac{(1+x)}{1}\frac{(2+x)}{2}\ldots\frac{(m+x)}{m}\frac{1}{(m+1)^x} = \frac{1}{\Gamma(1+x)}$$

e il $k-$esimo termine delle disuguaglianze è proprio $\binom{k-1+x}{k-1}/k^x$. □

# Bibliografia


[Sha50] Claude E. Shannon. "Programming a Computer for Playing Chess". In: *Philosophical Magazine, Ser.7, Vol. 41, No. 314* (Mar. 1950).

[Bru63] Alexander L'vovich Brudno. "Bounds and valuations for shortening the scanning of variations." In: *Problemy Kibernet* 10 (1963), pp. 141–150.

[KM75] Donald E. Knuth and Ronald W. Moore. "An analysis of alpha-beta pruning". In: *Artificial Intelligence* 6.4 (1975), pp. 293–326. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(75)90019-3. URL: https://www.sciencedirect.com/science/article/pii/0004370275900193.

[And00] Junghanns Andreas. "Are There Practical Alternatives To Alpha-Beta in Computer Chess?" In: *CCA Journal* (Sept. 2000).

[McC06] John McCarthy. *Human-level AI is harder than it seemed in 1955*. 2006. URL: http://www-formal.stanford.edu/jmc/slides/wrong/wrong-sli/wrong-sli.html (visited on 08/15/2022).

[Guo18] Bojun Guo. *Syzygy tablebases*. 2018. URL: https://syzygy-tables.info/ (visited on 08/15/2022).

[SP22] S.Russell and P.Norvig. *Artificial intelligence: a modern approach,* Pearson 2022 (4 ed), 2022.