UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA

Laurea triennale in Informatica

Fondamenti di Intelligenza Artificiale

Lezione 11 - Ricerca con avversari (2)

Ricerca con Avversari

Algoritmo Minimax, performance

La ricerca minimax non è altro che una ricerca in profondità e, pertanto, eredita molti degli aspetti già visti nelle precedenti lezioni:

Completezza: L'algoritmo è completo se l'albero è finito - molti giochi, inclusi gli scacchi, hanno regole specifiche per garantire il termine del gioco.

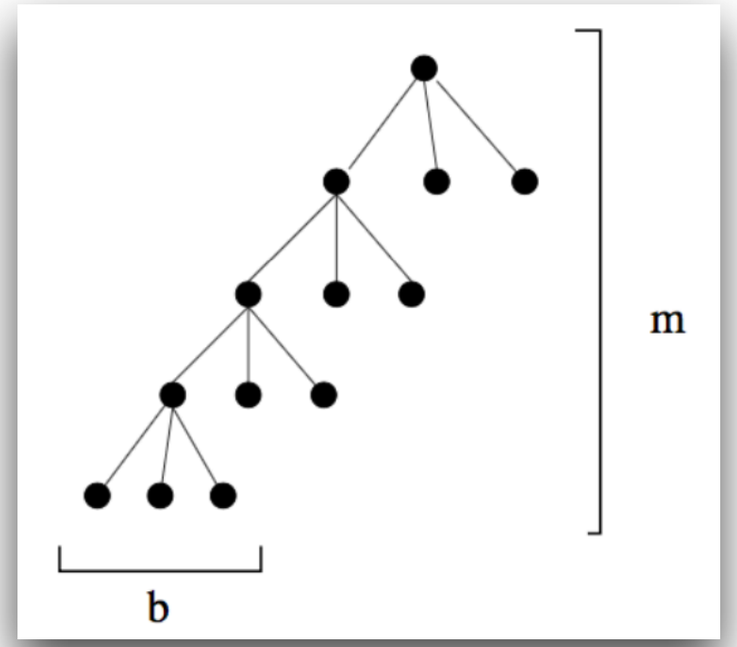
Ottimalità: Come già detto, l'algoritmo è ottimo nel caso in cui l'avversario è ottimo.

Complessità temporale: La complessità è governata dalla dimensione dell'albero di gioco. Sia *m* la profondità massima, nel caso pessimo la ricerca genererà $O(b^m)$ nodi.

Per i giochi reali, questa complessità temporale è improponibile! Nella prossima lezione vedremo come migliorare l'usabilità di questo algoritmo.

cammino. Una volta che un nodo è stato espanso, può essere rimosso dalla memoria non appena tutti i suoi discendenti sono stati esplorati completamente.

Per un albero di gioco con fattore di ramificazione *b* e profondità *m*, la complessità sarà di O(bm).



Ricerca con Avversari

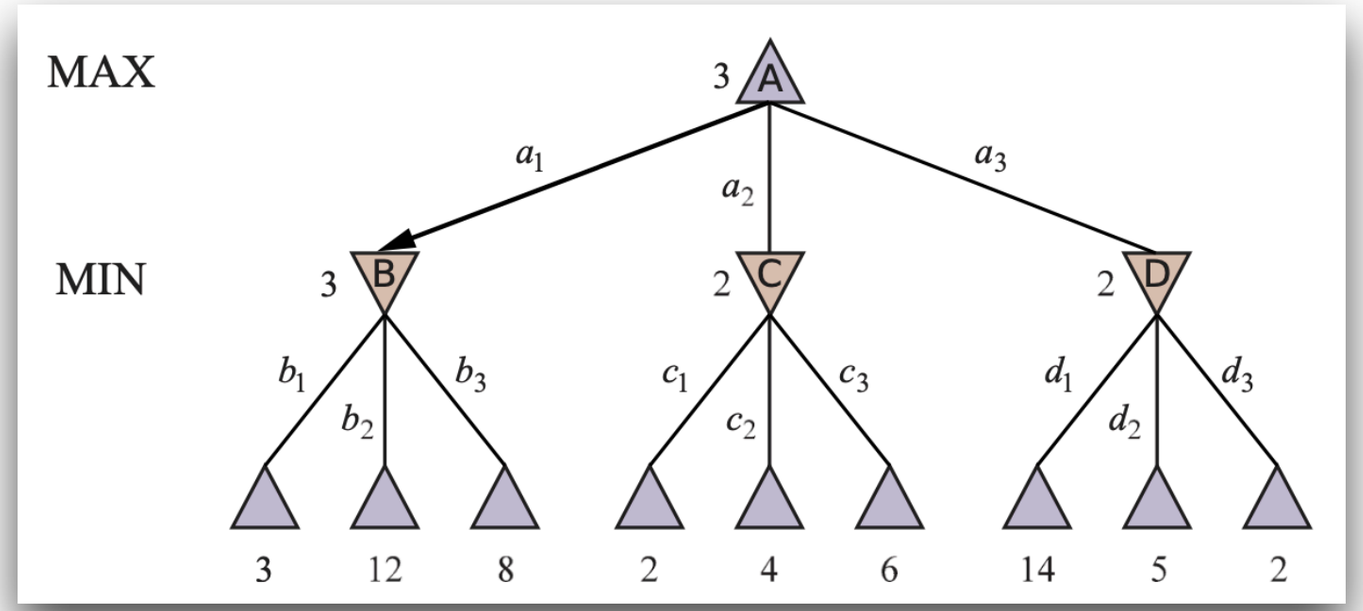
Algoritmo Minimax, considerazioni

Sfortunatamente, non possiamo eliminare l'esponente, ma possiamo almeno dimezzarlo: è di fatti possibile calcolare la decisione minimax corretta **senza** guardare tutti i nodi dell'albero di gioco.

Parleremo di **potatura**, ovvero la tecnica che consente di evitare di prendere in considerazione grandi porzioni dell'albero.

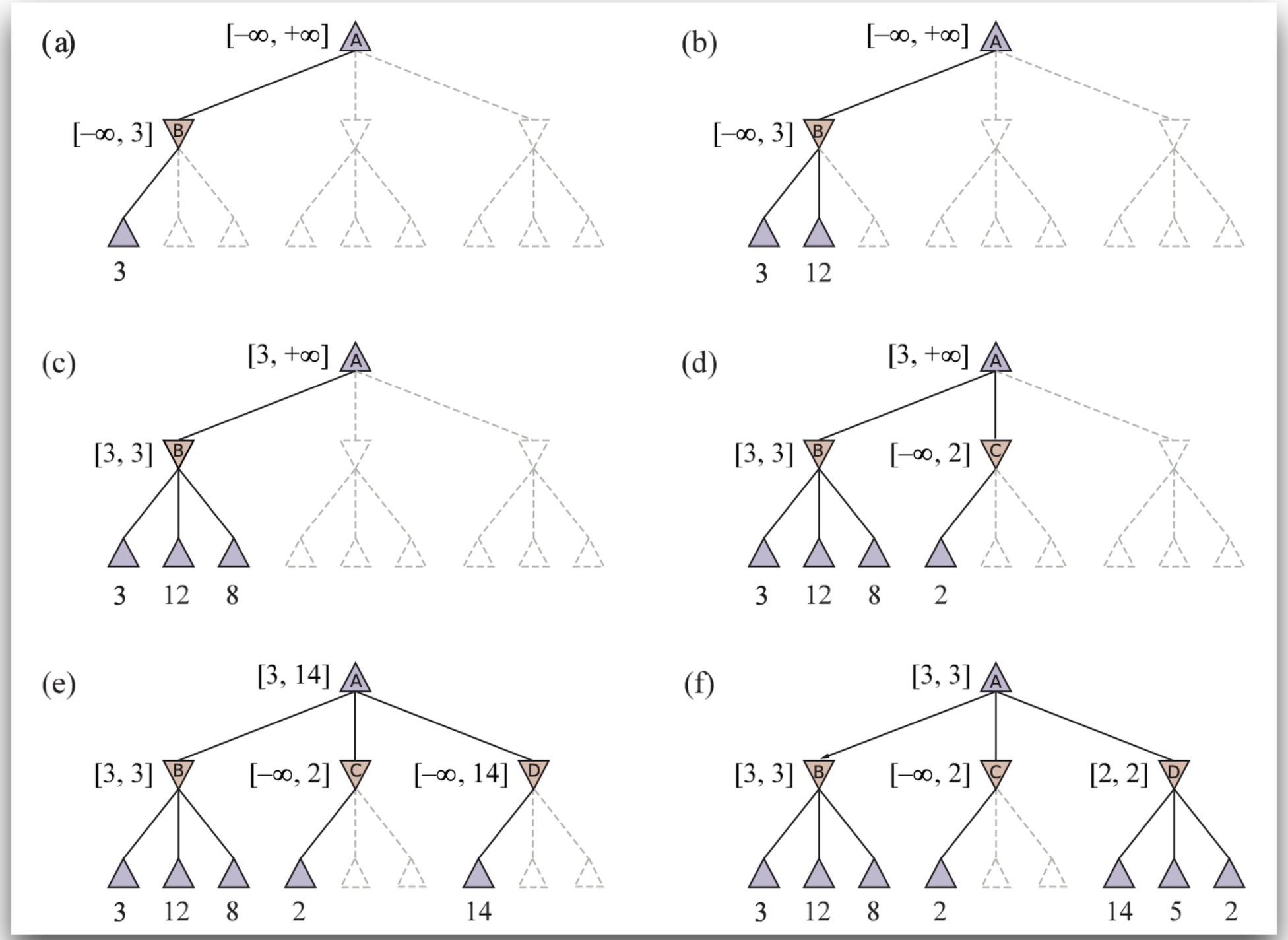
La specifica tecnica che esamineremo è chiamata **potatura alfa-beta**, che applicata ad un albero minimax restituisce lo stesso risultato della tecnica minimax standard, ma "pota" i rami che non possono influenzare la decisione finale.

Per capirne il funzionamento, consideriamo nuovamente l'albero di gioco discusso nella lezione precedente.



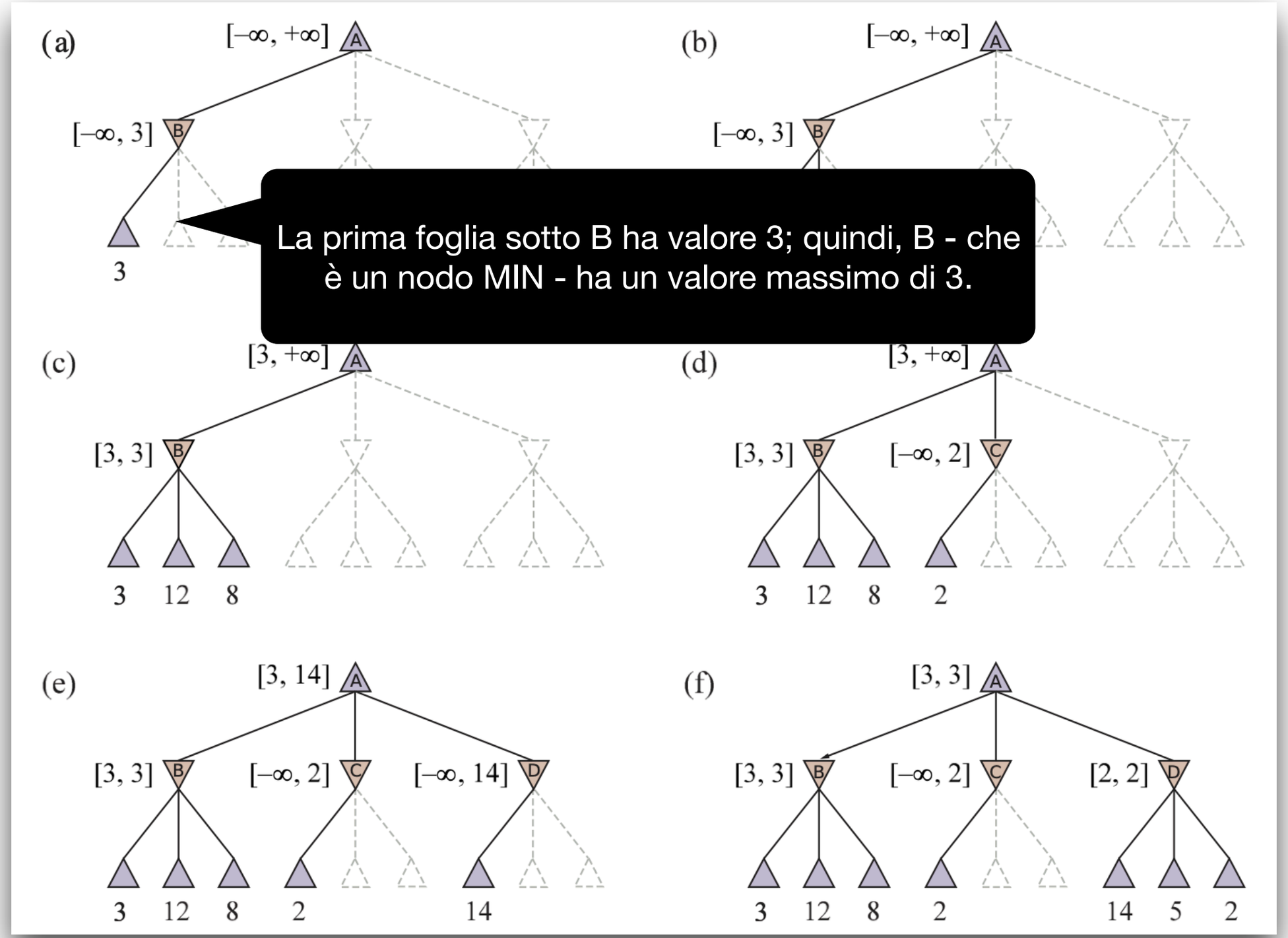
Ricerca con Avversari

Algoritmo Minimax con Potatura Alfa-Beta



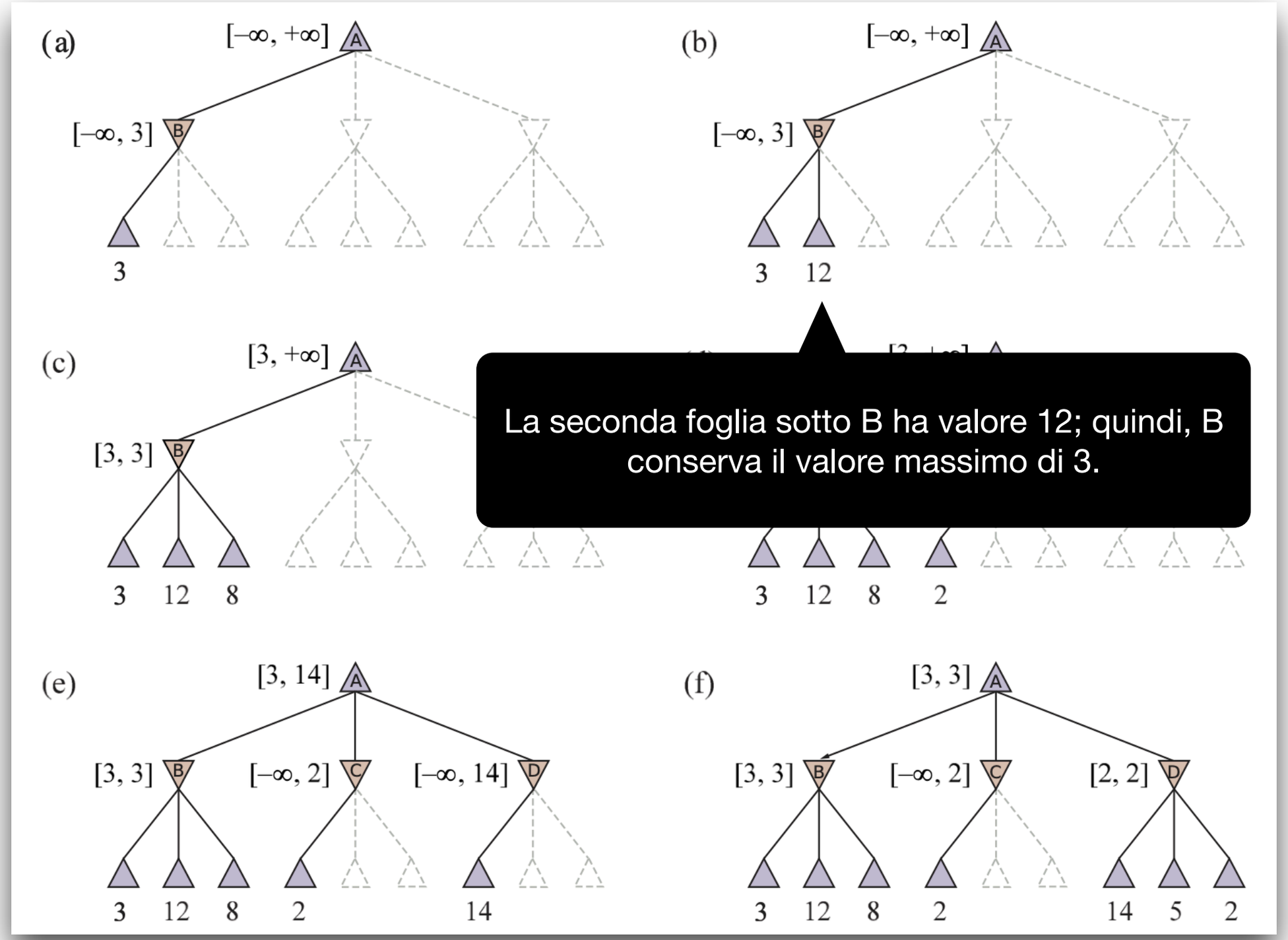
Ricerca con Avversari

Algoritmo Minimax con Potatura Alfa-Beta



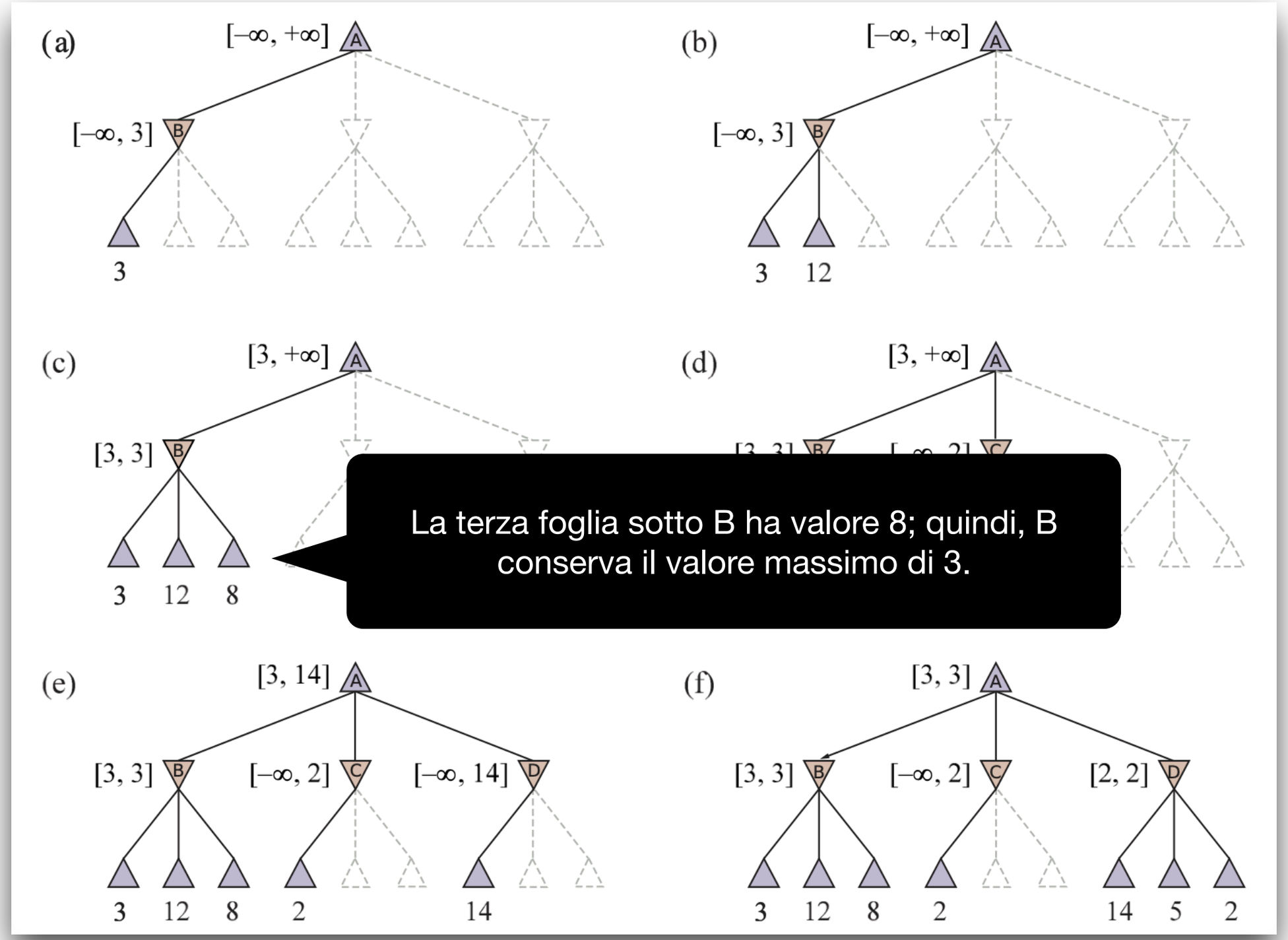
Ricerca con Avversari

Algoritmo Minimax con Potatura Alfa-Beta



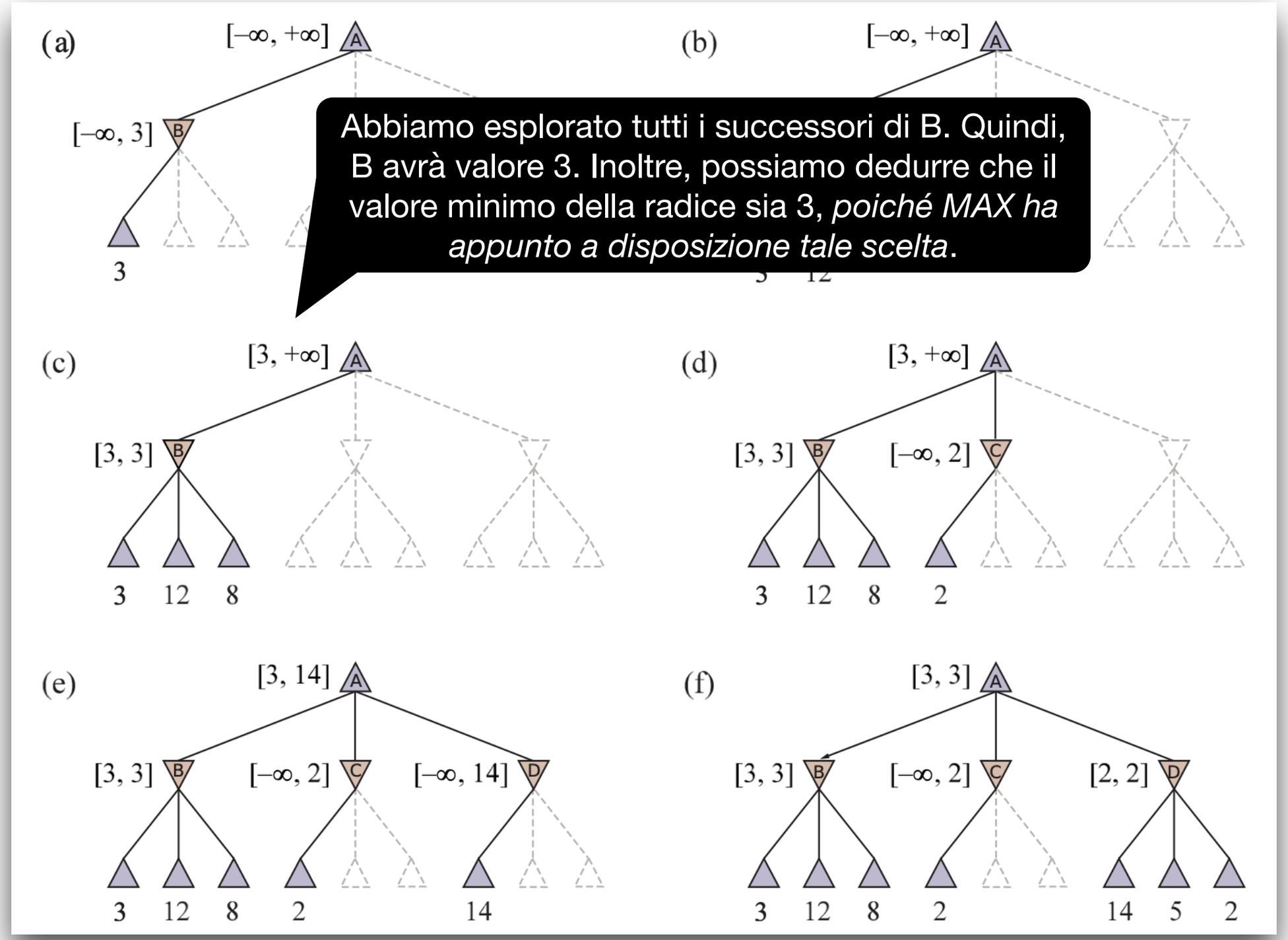
Ricerca con Avversari

Algoritmo Minimax con Potatura Alfa-Beta



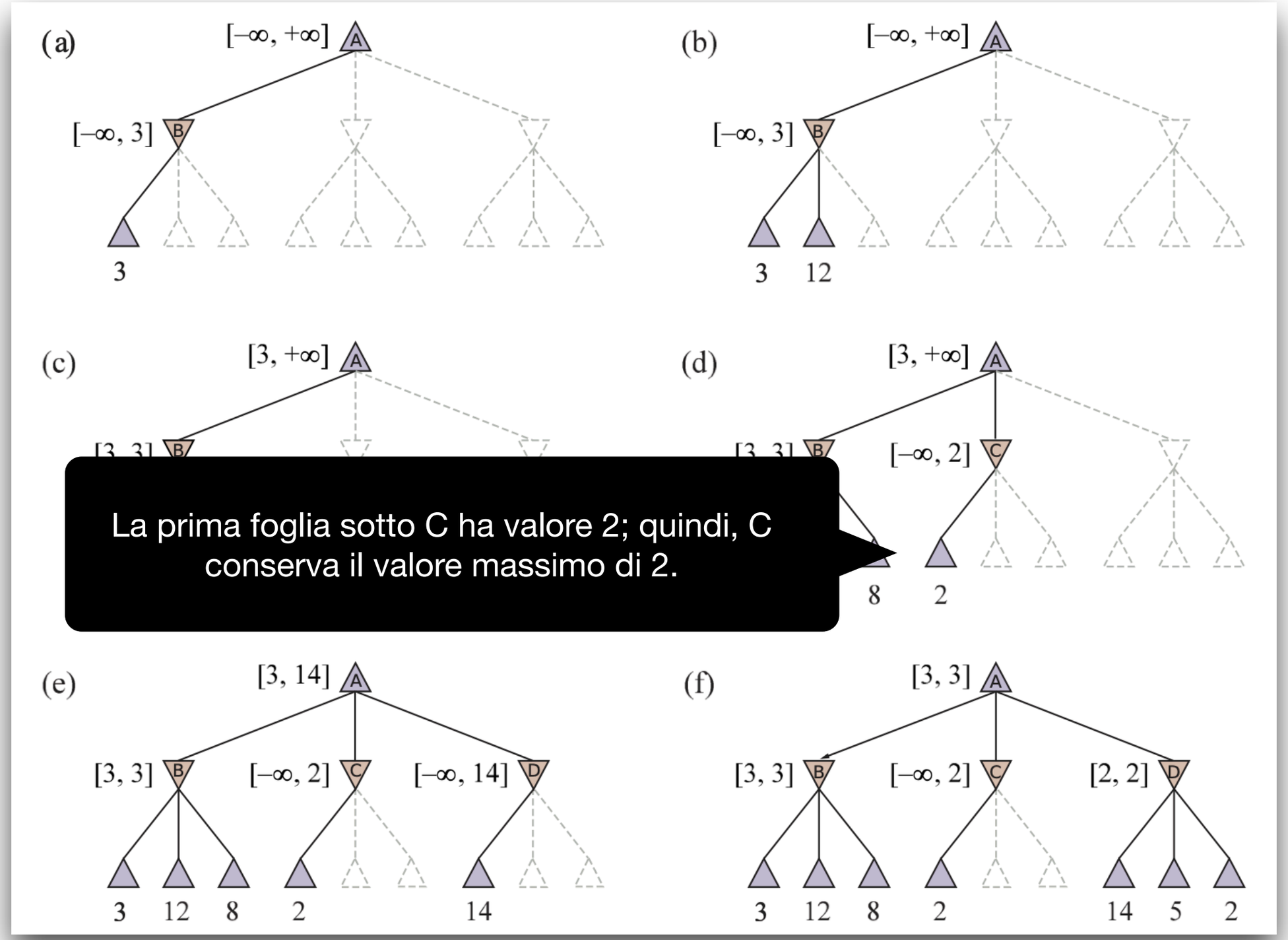
Ricerca con Avversari

Algoritmo Minimax con Potatura Alfa-Beta



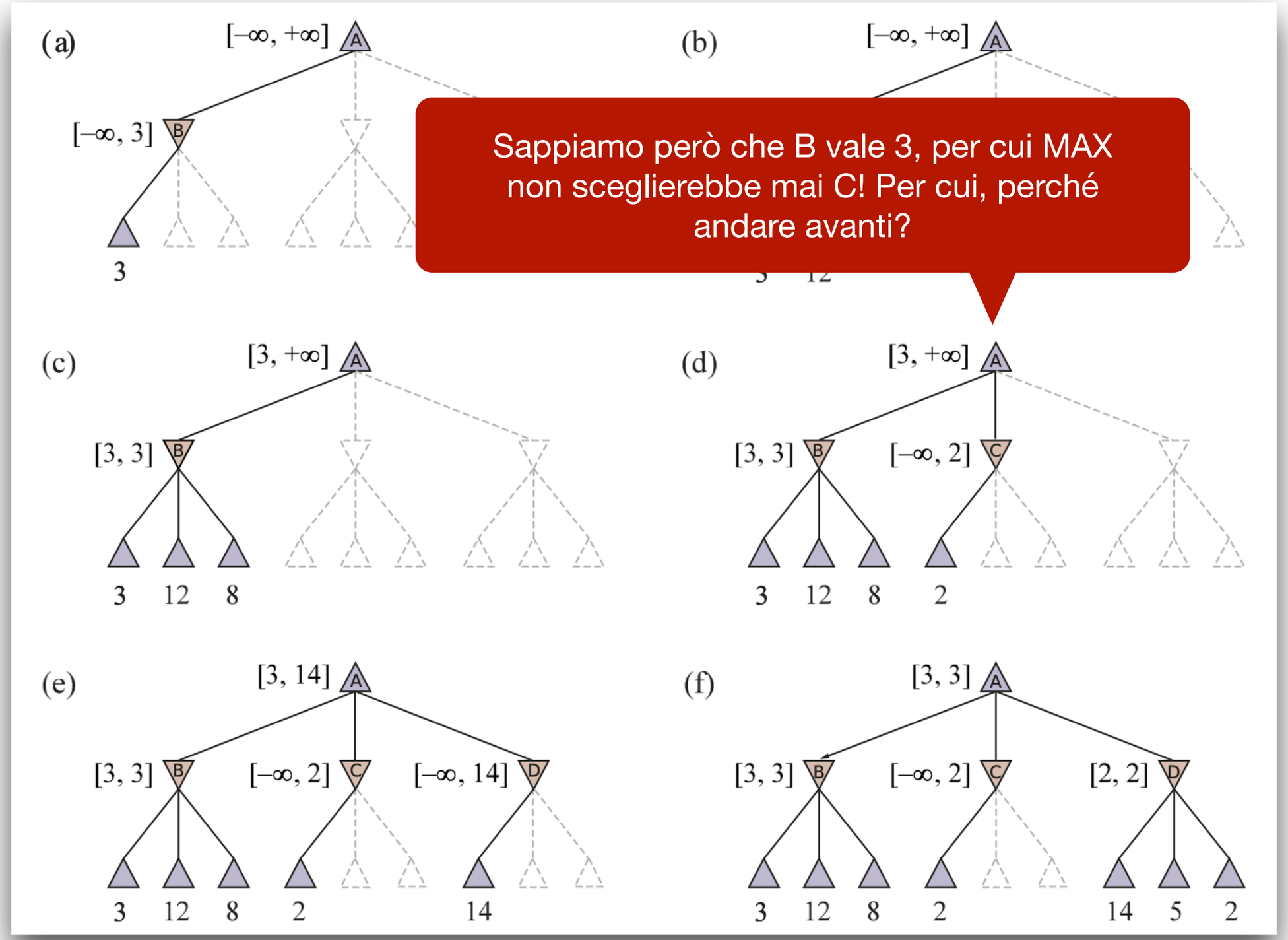
Ricerca con Avversari

Algoritmo Minimax con Potatura Alfa-Beta



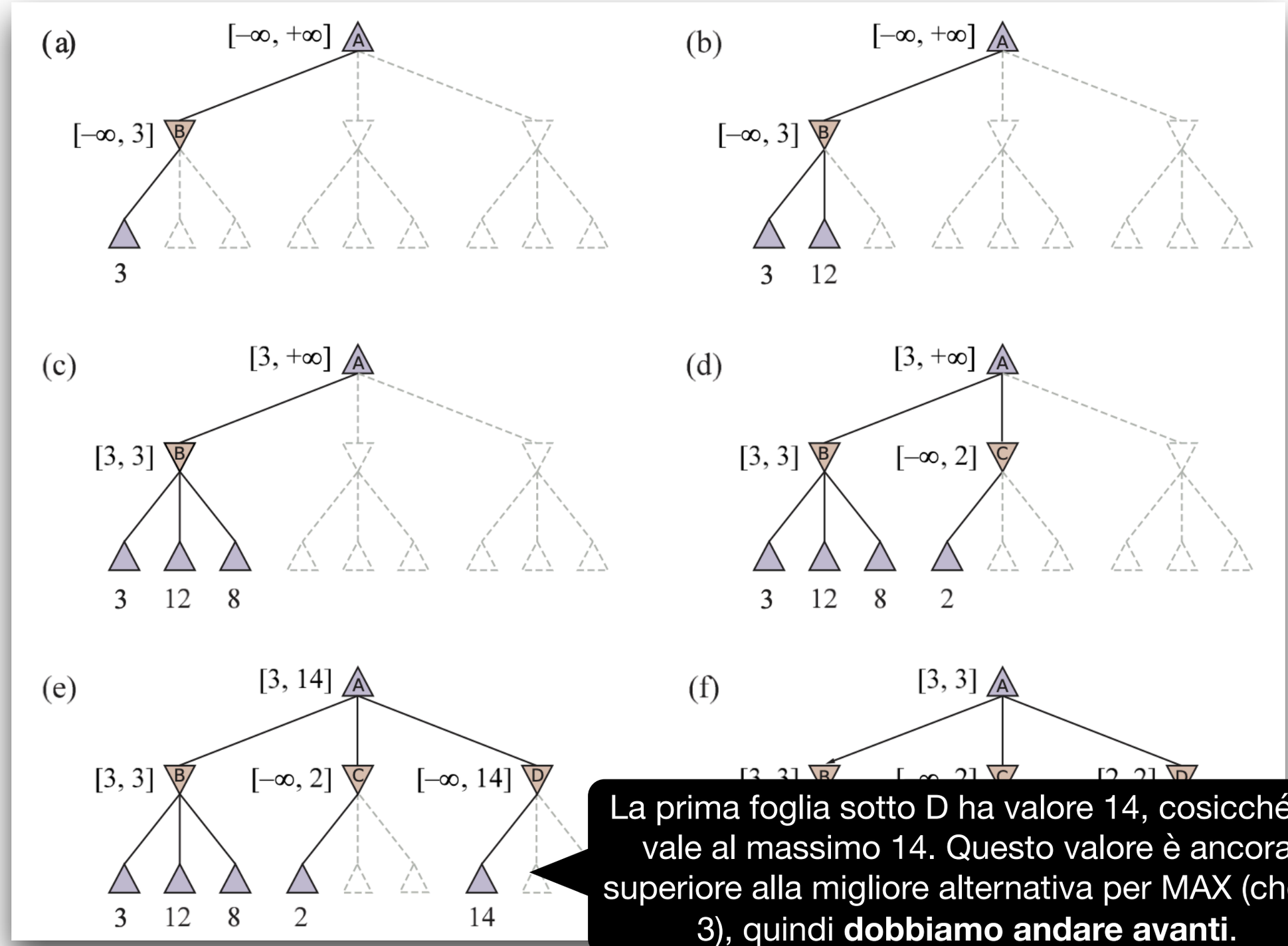
Ricerca con Avversari

Algoritmo Minimax con Potatura Alfa-Beta



Ricerca con Avversari

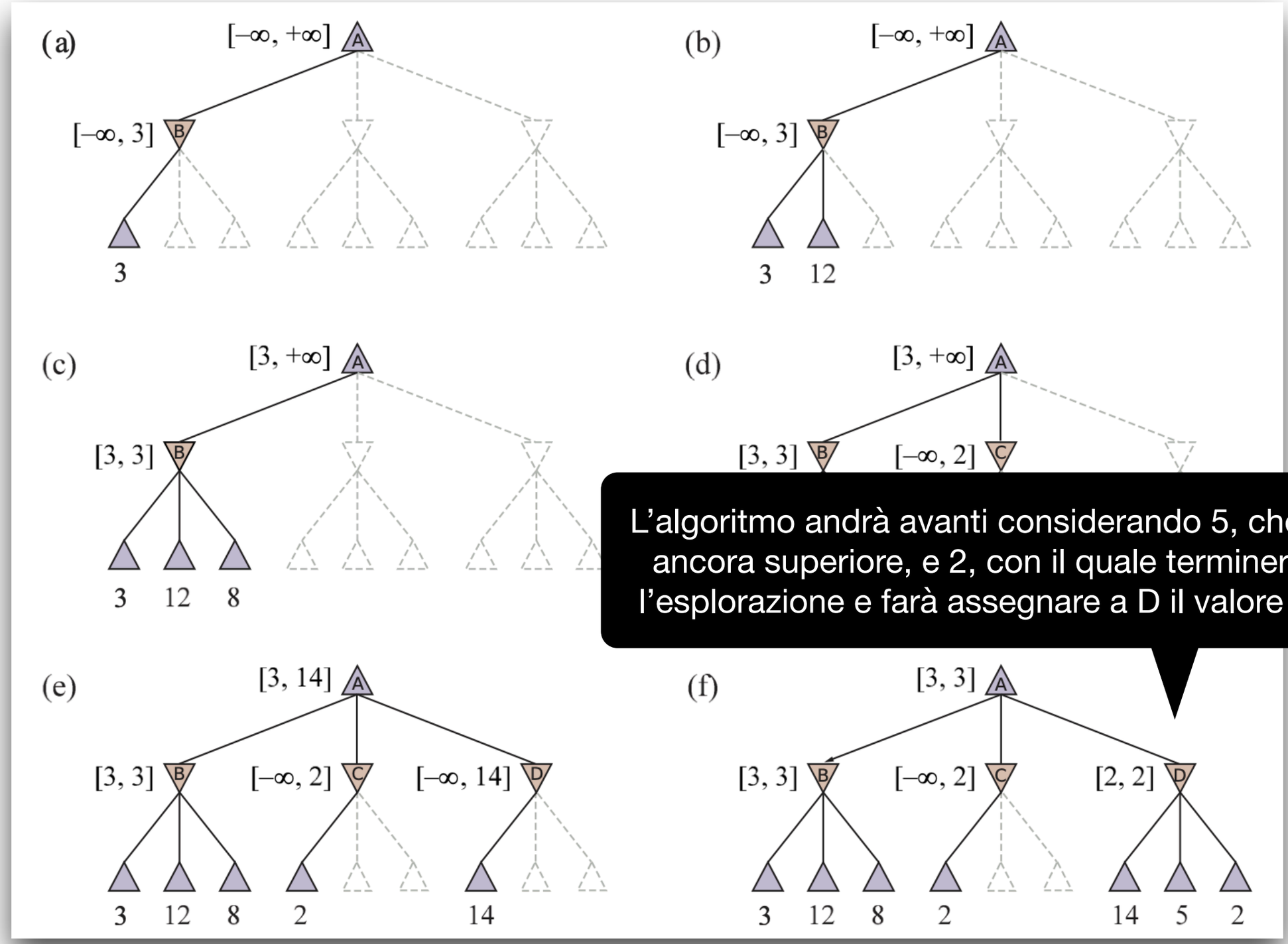
Algoritmo Minimax con Potatura Alfa-Beta



La prima foglia sotto D ha valore 14, cosicché D vale al massimo 14. Questo valore è ancora superiore alla migliore alternativa per MAX (che è 3), quindi **dobbiamo andare avanti**.

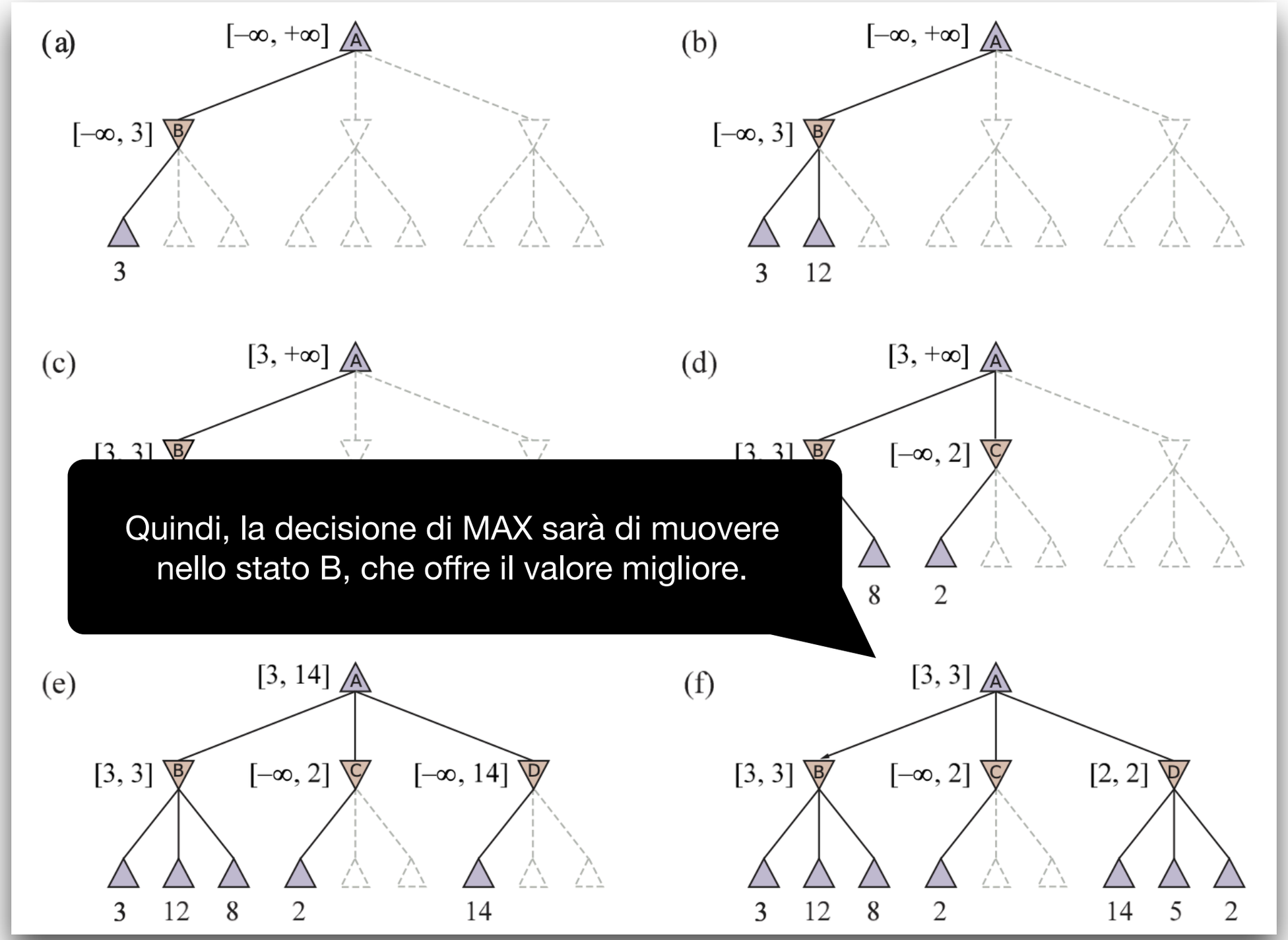
Ricerca con Avversari

Algoritmo Minimax con Potatura Alfa-Beta



Ricerca con Avversari

Algoritmo Minimax con Potatura Alfa-Beta



Quindi, la decisione di MAX sarà di muovere nello stato B, che offre il valore migliore.

Ricerca con Avversari

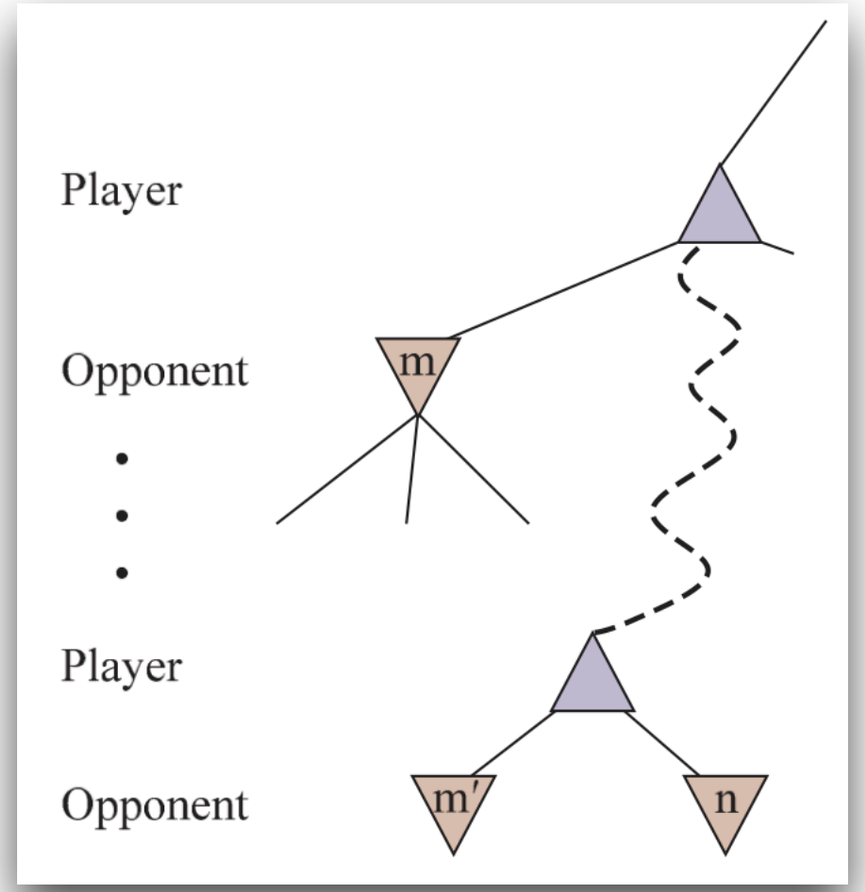
Algoritmo Minimax con Potatura Alfa-Beta

La potatura alfa-beta può essere applicata ad alberi di qualunque profondità e spesso è possibile potare interi sotto-alberi, rendendo la potatura più efficace rispetto all'esempio fatto in precedenza.

Il principio generale è questo: consideriamo un nodo *n* da qualche parte dell'albero tale che il giocatore abbia la possibilità di spostarsi in quel nodo.

Se esiste una scelta migliore *m* a livello del nodo padre o di qualunque nodo precedente, *allora n non sarà mai raggiunto in tutta la partita*.

Di conseguenza, possiamo potare *n* non appena abbiamo raccolto abbastanza informazioni per poter giungere a questa conclusione.



Alfa. Il valore della scelta migliore (quella con il valore più alto) per MAX che abbiamo trovato sin qui in un qualsiasi punto di scelta lungo il cammino.

Beta. Il valore della scelta migliore (quella con il valore più basso) per MIN che abbiamo trovato sin qui in un qualsiasi punto di scelta lungo il cammino.

Ricerca con Avversari

Algoritmo Minimax con Potatura Alfa-Beta

```
function RICERCA-ALFA-BETA(s) returns un'azione
   v <- VALORE-MAX(s, -∞, +∞)
   return l'azione in AZIONI(s) con valore v
```

```
function VALORE-MAX(s) returns un valore di utilità
   if TEST-TERMINAZIONE(s) then return UTILITÀ(s)
   v <- -∞

   for each a in AZIONI(s) do
      v <- MAX(v, VALORE-MIN(RISULTATO(s, a), alfa, beta))
      if v >= beta then return v
      alfa <- MAX(alfa, v)

   return v
```

```
function VALORE-MIN(s) returns un valore di utilità
   if TEST-TERMINAZIONE(s) then return UTILITÀ(s)
   v <- +∞

   for each a in AZIONI(s) do
      v <- MIN(v, VALORE-MAX(RISULTATO(s, a), alfa, beta))
      if v <= alfa then return v
      beta <- MIN(beta, v)

   return v
```

Ricerca con Avversari

Algoritmo Minimax con Potatura Alfa-Beta

```
function RICERCA-ALFA-BETA(s) returns un'azione
   v <- VALORE-MAX(s, -∞, +∞)
   return l'azione in AZIONI(s) con valore v
```

```
function VALORE-MAX(s) returns un valore di utilità
   if TEST-TERMINAZIONE(s) then return UTILITÀ(s)
   v <- -∞

   for each a in AZIONI(s) do
      v <- MAX(v, VALORE-MIN(RISULTATO(s, a), alfa, beta))
      if v >= beta then return v
      alfa <- MAX(alfa, v)

   return v
```

```
function VALORE-MI
   if TEST-TERMINA
   v <- +∞

   for each a in AZIONI(s) do
      v <- MIN(v, VALORE-MAX(RISULTATO(s, a), alfa, beta))
      if v <= alfa then return v
      beta <- MIN(beta, v)

   return v
```

/* La procedura di base è la stessa dell'algoritmo standard. Aggiungiamo però il codice necessario ad aggiornare alfa e beta e il codice necessario per passare alfa e beta da una procedura ad un'altra. */

Ricerca con Avversari

Algoritmo Minimax con Potatura Alfa-Beta

```
function RICERCA-ALFA-BETA(s) returns un'azione
   v <- VALORE-MAX(s, -∞, +∞)
   return l'azione in AZIONI(s) con valore v
```

```
function VALORE-MAX(s) returns un valore di utilità
   if TEST-TERMINAZIONE(s) then return UTILITÀ(s)
   v <- -∞

   for each a in AZIONI(s) do
      v <- MAX(v, VALORE-MIN(RISULTATO(s, a), alfa, beta))
      if v >= beta then return v
      alfa <- MAX(alfa, v)

   return v
```

```
function VALO
   if TEST-TE
   v <- +∞

   for each a in ZIONI(s) do
      v <- MIN(v, VALORE-MAX(RISULTATO(s, a), alfa, beta))
      if v <= alfa then return v
      beta <- MIN(beta, v)

   return v
```

/* Lo stesso vale per la funzione VALORE-MIN. */

Ricerca con Avversari

Algoritmo Minimax con Potatura Alfa-Beta

```
function RICERCA-ALFA-BETA(s) returns un'azione
   v <- VALORE-MAX(s, -∞, +∞)
   return l'azione in AZIONI(s) con valore v
```

```
function VALORE-MAX(s) returns un valore di utilità
   if TEST-TERMINAZIONE(s) then return UTILITÀ(s)
   v <- -∞

   for each a in AZIONI(s) do
      v <- MAX(v, VALORE-MIN(RISULTATO(s, a), alfa, beta))
      if v >= beta then return v
      alfa <- MAX(alfa, v)

   return v
```

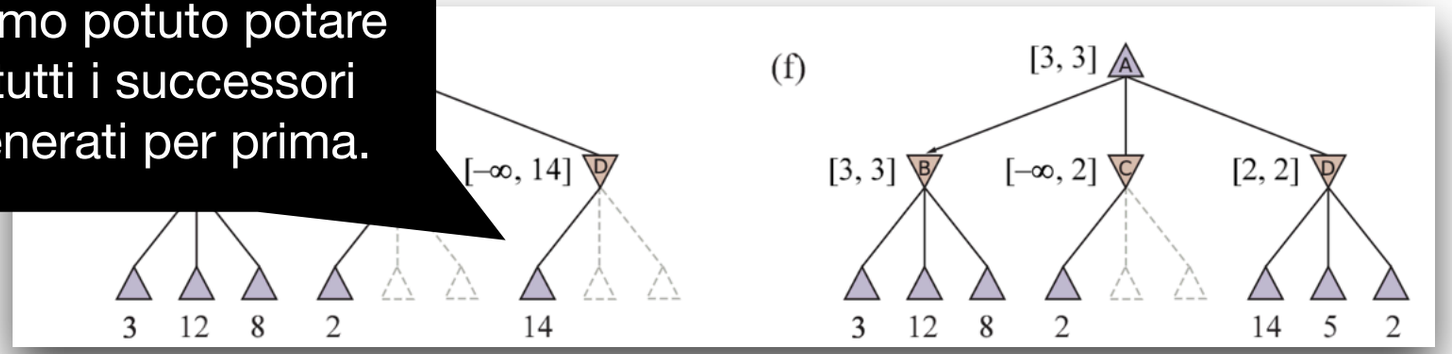
```
function VALORE-MIN(s) returns un va
   if TEST-TERMINAZIONE(s) then retu
   v <- +∞

   for each a in AZIONI(s) do
      v <- MIN(v, VALORE-MAX(RISULTATO(s, a), alfa, beta))
      if v <= alfa then return v
      beta <- MIN(beta, v)

   return v
```

/* Vale la pena notare che l'algoritmo, nel caso di MAX, stoppa l'esecuzione quando il valore dello stato corrente è superiore o uguale all'attuale beta —> abbiamo trovato uno stato migliore e pertanto non è conveniente proseguire l'esplorazione. */

Algoritmo Minimax con Potatura Alfa-Beta, ordinamento delle mosse

L'efficacia della potatura alfa-beta dipende fortemente dall'ordinamento in cui gli stati sono esaminati.

Nel caso precedente, non abbiamo potuto potare alcun successore di D perché tutti i successori peggiori (per MIN) sono stati generati per prima.



Quindi, potrebbe essere una buona idea quella di esaminare per prima i successori *più promettenti* — ovvero, usando una strategia *best-first*. Se questo può essere fatto, la ricerca alfa-beta dovrà esaminare "solo" $O(b^{m/2})$ nodi - piuttosto che $O(b^m)$.

Una modifica di questo tipo consente alla ricerca alfa-beta di risolvere un albero profondo il doppio di quello che può risolvere minimax *nello stesso lasso di tempo*.

Talvolta, decidere un ordinamento è "semplice". Negli scacchi, questa potrebbe essere una funziona tale che la ricerca abbia come obiettivo quello di catturare pezzi, poi di minacciarli, poi le mosse in avanti ed infine quelle all'indietro.

Ricerca con Avversari

Algoritmo Minimax con Potatura Alfa-Beta, ordinamento dinamico delle mosse

Aggiungere schemi dinamici di ordinamento delle mosse, come provare per prima cosa le mosse che si sono rivelate le migliori in passato, ci porta vicino al limite teorico.

Un modo per ottenere informazioni dalla mossa corrente è quello di utilizzare una **ricerca ad approfondimento iterativo**.

> **Ricerca ad approfondimento iterativo:** Con questa strategia, il limite viene incrementato progressivamente, finché un nodo obiettivo non è identificato.

Con questa strategia, si cerca innanzitutto uno strato in profondità e si registra il miglior cammino di mosse. Poi si cerca uno strato ancora in profondità, ma *si utilizza il cammino registrato allo scopo di fornire informazioni per l'ordinamento delle mosse*.

Le mosse migliori sono spesso chiamate *mosse killer* e si parla di *euristica della mossa killer* quando queste vengono provate per prime.

Adesso, però, la domanda vera è: *come utilizzare la ricerca ad approfondimento iterativo per l'ordinamento delle mosse?*

Ricordate che la ricerca ad approfondimento iterativo è **analoga a quella in ampiezza**, poiché ad ogni iterazione esplora completamente un livello di nodi prima di prendere in considerazione il successivo. Quindi, possiamo **esplorare i nodi in ampiezza e decidere come variare l'ordinamento in base ai valori minimax!**

Algoritmo Minimax con Potatura Alfa-Beta, ordinamento dinamico delle mosse

Un altro modo è quello di considerare le **trasposizioni**, che portano l'albero di gioco a contenere **cammini ridondanti** — stati ripetuti più volte nello spazio degli stati che possono determinare un aumento esponenziale del costo di ricerca.

Nella dama, ad esempio, il bianco può fare la mossa a_1 a cui il nero risponderà con b_1. Allo stesso tempo, una mossa indipendente a_2 porterà alla mossa b_2. Le sequenze [a_1, b_1, a_2, b_2] e [a_1, b_2, a_2, b_1] si concluderanno con la stessa configurazione!

Quindi, vale la pena *memorizzare la valutazione di una configurazione* in una tabella hash la prima volta che questa viene incontrata.

La tabella viene chiamata **tabella delle trasposizioni**: in essenza, questa è identica alla lista `esplorati` che abbiamo visto con gli algoritmi di ricerca non informata.

Proprio perché identica alla lista `esplorati`, possiamo decidere di ordinare le mosse successive sulla base della **conoscenza già acquisita esplorando una trasposizione**.

Dall'altra parte, se l'albero di gioco avesse milioni di stati, si arriverà necessariamente a dover mantenere in tabella solo alcune delle trasposizioni. Diverse strategie sono disponibili per selezionare le trasposizioni da mantenere in memoria.

Una strategia potrebbe essere derivata tramite la definizione di una funzione euristica.

Ricerca con Avversari

Decisioni imperfette in tempo reale

L'algoritmo minimax genera l'intero spazio di ricerca, mentre quello alfa-beta ci permette di potarne buona parte. Tuttavia, anche alfa-beta deve condurre la ricerca fino agli stati terminali - almeno per una porzione dello spazio di ricerca.

Questa profondità **risulta normalmente non gestibile**, perché le mosse devono essere calcolate in un tempo ragionevole.

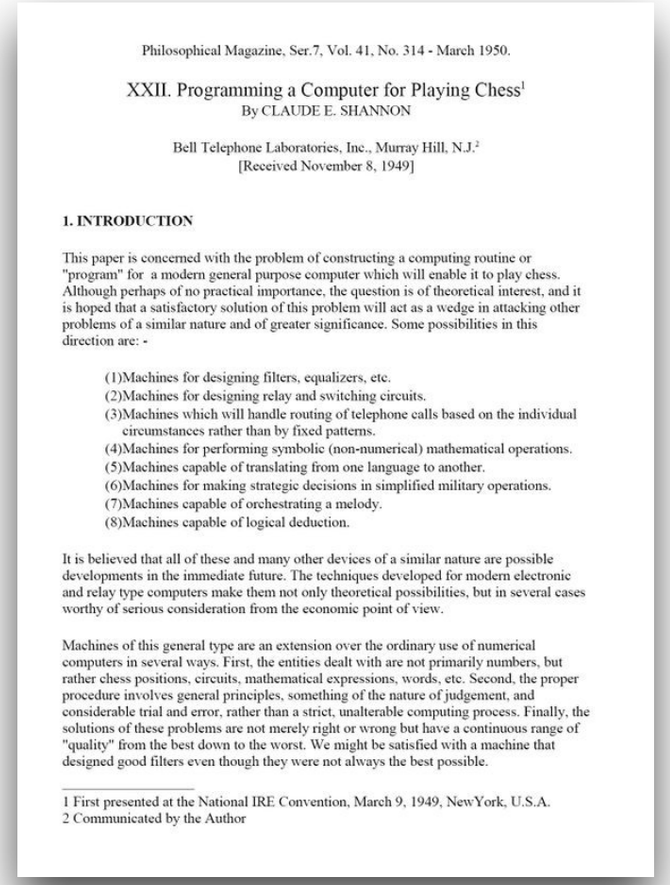
Supponiamo di avere a disposizione 100 secondi per mossa e poter esplorare 10^4 nodi per secondo —> si possono esplorare circa 10^6 nodi per mossa, il ché non è ragionevole!

Claude Shannon (lo stesso Shannon della *formulazione di entropia*) nel 1950 propose che i programmi "tagliassero" la ricerca **prima di raggiungere le foglie applicando una funzione di valutazione euristica agli stati**.

In questa formulazione del problema, i nodi non terminali *diventano* foglie. Il minimax viene modificato in due punti:

- La funzione di utilità viene sostituita da EVAL, che fornisce una stima dell'utilità della posizione raggiunta.
- Il test di terminazione viene sostituito con un test di taglio (*cutoff test*), che decide quando applicare EVAL.

Ridefiniamo quindi il problema in questi termini.



Philosophical Magazine, Ser.7, Vol. 41, No. 314 - March 1950.

XXII. Programming a Computer for Playing Chess[1]
By CLAUDE E. SHANNON

Bell Telephone Laboratories, Inc., Murray Hill, N.J.[2]
[Received November 8, 1949]

1. INTRODUCTION

This paper is concerned with the problem of constructing a computing routine or "program" for a modern general purpose computer which will enable it to play chess. Although perhaps of no practical importance, the question is of theoretical interest, and it is hoped that a satisfactory solution of this problem will act as a wedge in attacking other problems of a similar nature and of greater significance. Some possibilities in this direction are: -

(1)Machines for designing filters, equalizers, etc.
(2)Machines for designing relay and switching circuits.
(3)Machines which will handle routing of telephone calls based on the individual circumstances rather than by fixed patterns.
(4)Machines for performing symbolic (non-numerical) mathematical operations.
(5)Machines capable of translating from one language to another.
(6)Machines for making strategic decisions in simplified military operations.
(7)Machines capable of orchestrating a melody.
(8)Machines capable of logical deduction.

It is believed that all of these and many other devices of a similar nature are possible developments in the immediate future. The techniques developed for modern electronic and relay type computers make them not only theoretical possibilities, but in several cases worthy of serious consideration from the economic point of view.

Machines of this general type are an extension over the ordinary use of numerical computers in several ways. First, the entities dealt with are not primarily numbers, but rather chess positions, circuits, mathematical expressions, words, etc. Second, the proper procedure involves general principles, something of the nature of judgement, and considerable trial and error, rather than a strict, unalterable computing process. Finally, the solutions of these problems are not merely right or wrong but have a continuous range of "quality" from the best down to the worst. We might be satisfied with a machine that designed good filters even though they were not always the best possible.

1 First presented at the National IRE Convention, March 9, 1949, NewYork, U.S.A.
2 Communicated by the Author

Ricerca con Avversari

Decisioni imperfette in tempo reale, ridefinizione del problema

La nuova formulazione porta il problema da:

$$\text{MINIMAX}(n) = \begin{cases} \text{UTILITÀ}(s, \text{MAX}) & \text{se TEST-TERMINALE}(s); \\ \max_{a \in \text{AZIONI}(s)} \text{VALORE-MINIMAX-RISULTATO}(s, a) & \text{se GIOCATORE}(s) = \text{MAX}; \\ \min_{a \in \text{AZIONI}(s)} \text{VALORE-MINIMAX-RISULTATO}(s, a) & \text{se GIOCATORE}(s) = \text{MIN}; \end{cases}$$

alla seguente:

$$\text{H-MINIMAX}(n) = \begin{cases} \text{EVAL}(s) & \text{se TEST-TAGLIO}(s, d); \\ \max_{a \in \text{AZIONI}(s)} \text{H-MINIMAX}(\text{RISULTATO}(s, a), d+1) & \text{se GIOCATORE}(s) = \text{MAX}; \\ \min_{a \in \text{AZIONI}(s)} \text{H-MINIMAX}(\text{RISULTATO}(s, a), d+1) & \text{se GIOCATORE}(s) = \text{MIN}; \end{cases}$$

Una funzione di valutazione restituisce una stima del guadagno atteso in una determinata posizione, **esattamente come le funzioni euristiche.**

Pertanto, la difficoltà di definizione di una funzione di valutazione è identica. D'altro canto, ogni giocatore di scacchi implementerà una propria strategia!

Decisioni imperfette in tempo reale, funzioni di valutazione

La funzione di valutazione dovrebbe in primo luogo *ordinare* gli stati terminali nello stesso modo della funzione di utilità. Gli stati che sono vittorie devono avere una valutazione migliore dei pareggi, che a loro volta devono essere migliori delle sconfitte.

La funzione di valutazione dovrebbe poi essere *veloce* da calcolare (altrimenti, non avrebbe senso sostituire la funzione di utilità!)

Terzo, per gli stati non terminali la funzione di valutazione dovrebbe avere una *forte correlazione con la probabilità reale di vincere la partita*.

La maggior parte delle funzioni di valutazione si basano sulle *caratteristiche* di uno stato: ad esempio, negli scacchi avremo come caratteristiche il numero di pedoni bianchi, il numero di pedoni neri, di regine bianche, di regine nere, e così via.

Le caratteristiche definiscono le cosiddette *classi di equivalenza*, che rappresenta un insieme di stati aventi lo stesso valore per tutte le caratteristiche. Ad esempio, tutti gli stati finali con due pedoni contro un pedone rappresentano una classe.

Ogni classe di equivalenza può contenere stati che portano alla vittoria, al pareggio e alla sconfitta. La funzione di valutazione non potrà sapere quali sono, ma potrà restituire un singolo valore che riflette la *proporzione di stati* che portano ad ogni risultato.

Portiamo avanti l'esempio degli scacchi.

Ricerca con Avversari

Decisioni imperfette in tempo reale, funzioni di valutazione - l'esempio degli scacchi

Supponiamo che la nostra esperienza suggerisca che il 72% degli stati nella classe di equivalenza di due pedoni contro un pedone conducano ad una vittoria (utilità=+1), il 20% ad una sconfitta (0) e l'8% al pareggio (1/2).

In questo caso, una valutazione ragionevole per gli stati di questa classe di equivalenza potrebbe essere quella del *valore atteso*: (0,72 x 1) + (0,20 x 0) + (0,8 x 1/2) = 0,76.

Si può, quasi sempre, determinare il valore atteso di ogni categoria, fornendo così una funzione di valutazione applicabile ad ogni stato.

Qualcuno potrebbe notare che una funzione di questo tipo richiede *un'eccessiva quantità di esperienza* in fase di definizione.

In molti casi, la funzione calcola valori separati per ogni caratteristica e poi li *combina* insieme per formare il valore finale. Nel caso degli scacchi, esiste il concetto di *valore del materiale*: ogni pedone vale 1, un cavallo o un alfiere 3, e così via. Altre caratteristiche come la difesa del re o una "buona" disposizione dei pedoni potrebbero avere altri valori, che però potrebbero essere semplicemente sommati.

Matematicamente, questo significa avere una funzione lineare pesata, poiché può essere espressa come:

$$EVAL(s) = w_1f_1(s) + w_2f_2(s) + \ldots + w_nf_n(s)$$; dove w_i è il peso e f_i è una caratteristica.

Decisioni imperfette in tempo reale, funzioni di valutazione - l'esempio degli scacchi

Supponiamo che la nostra esperienza suggerisca che il 72% degli stati nella classe di equivalenza di due pedoni contro un pedone conducano ad una vittoria (utilità=+1), il 20% ad una sconfitta (0) e l'8% al pareggio (1/2).

In questo caso, una valutazione ragionevole per gli stati di questa classe di equivalenza potrebbe essere quella del *valore atteso*: (0,72 x 1) + (0,20 x 0) + (0,8 x 1/2) = 0,76.

Si può, quasi sempre, determinare il valore atteso di ogni categoria, fornendo così una funzione di valutazione applicabile ad ogni stato.

Qualcuno potrebbe notare che una funzione di questo tipo richiede *un'eccessiva quantità di esperienza* in fase di definizione.

In molti casi, la funzione calcola valori separati per ogni caratteristica e poi li *combina* insieme per formare il valore finale. Nel caso degli scacchi, esiste il concetto di *valore del materiale*: ogni pedone vale 1, un cavallo o un alfiere 3, e così via. Altre caratteristiche come la difesa del re o una "buona" disposizione dei pedoni potrebbero avere altri valori, che però potrebbero essere semplicemente sommati.

ché può

atteristica.

Questo è possibile se i contributi individuali delle caratteristiche sono **indipendenti** dagli altri! Nei programmi di scacchi più recenti, si usano combinazioni non lineari.

Ricerca con Avversari

Decisioni imperfette in tempo reale, terminare la ricerca

Come detto, la funzione di valutazione è una delle due modifiche necessarie al minimax. L'altra è quella relativa alla definizione del test di taglio.

Nella definizione del test di taglio, dobbiamo fare attenzione ai cosiddetti stati *non quiescenti* - stati che portano ad una variazione repentina del valore delle mosse.

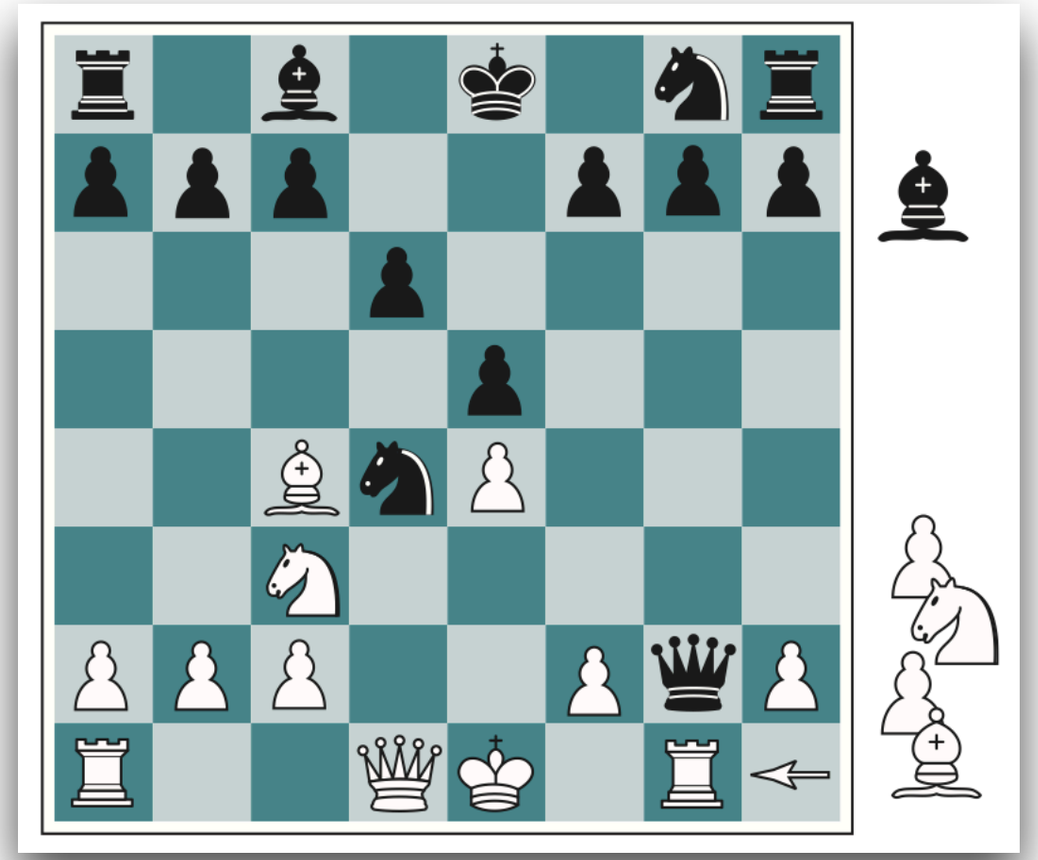
Supponiamo che, in una partita a scacchi, la funzione di valutazione sia basata sul vantaggio del materiale e che la ricerca sia arrivata al punto mostrato in figura.

Tocca al bianco muovere e il nero sembra essere in vantaggio, poiché ha un cavallo e due pedoni in più rispetto al bianco.

Il programma assocerebbe questo stato ad un valore euristico molto positivo, ritenendo che la situazione porterà ad una vittoria.

Ma la mossa successiva del bianco cattura la regina nera, portandolo in una situazione di vittoria quasi certa.

Questo è uno stato non quiescente, che porta ad un cambio netto dello stato di gioco.



Decisioni imperfette in tempo reale, terminare la ricerca

La funzione di valutazione dovrebbe perciò essere applicata solo a posizioni quiescenti, ovvero quelle per cui sia improbabile il verificarsi di grandi variazioni di valore nelle mosse immediatamente successive.

Talvolta, potrebbe essere inclusa nella definizione dell'algoritmo una *ricerca di quiescenza*, in cui stati non quiescenti vengono ulteriormente espansi fino a raggiungere posizioni quiescenti.

Da un punto di vista implementativo, le modifiche richieste all'algoritmo minimax con potatura alfa-beta sono minime.

```
function VALORE-MAX(s) returns un valore di utilità
   if TEST-TERMINAZIONE(s) then return UTILITÀ(s)
   if TEST-TAGLIO(s) then return EVAL(s)
   v <- -∞

   for each a in AZIONI(s) do
      v <- MAX(v, VALORE-MIN(RISULTATO(s, a), alfa, beta))
      if v >= beta then return v
      alfa <- MAX(alfa, v)

   return v
```

Decisioni imperfette in tempo reale, terminare la ricerca

La funzione di valutazione dovrebbe perciò essere applicata solo a posizioni quiescenti, ovvero quelle per cui sia improbabile il verificarsi di grandi variazioni di valore nelle mosse immediatamente successive.

Talvolta, potrebbe essere inclusa nella definizione dell'algoritmo una *ricerca di quiescenza*, in cui stati non quiescenti vengono ulteriormente espansi fino a raggiungere posizioni quiescenti.

Da un punto di vista implementativo, le modifiche richieste all'algoritmo minimax con potatura alfa-beta sono minime.

```
function VALORE-MAX(s) returns un valore di utilità
    if TEST-TERMINAZIONE(s) then return UTILITÀ(s)
    if TEST-TAGLIO(s) then return EVAL(s)
    v <- -∞

    for
        v                                          alfa, beta))
        i
        a

    return v
```

/* L'eventuale valutazione della quiescenza di s sarà a carico della funzione TEST-TAGLIO. */

Minimax con potatura alfa-beta: ulteriori miglioramenti

Potatura in avanti. L'idea di base è quella di esplorare solo alcune mosse ritenute particolarmente promettenti, tagliando le altre.

Un approccio è la *beam search*: di volta in volta, si considerano solo le prime *k* migliori mosse, definite sulla base della funzione di valutazione. Tuttavia, non garantisce la non potatura delle mosse migliori:

I *tagli probabilistici* sono invece basati sull'esperienza: questo utilizza statistiche ricavate dalle precedenti esperienze per ridurre la possibilità che mossa migliore venga potata.

Database di mosse di apertura e chiusura. L'idea di base è che all'inizio di una partita ci sono solitamente poche mosse sensate e ben studiate, per cui è inutile esplorarle tutte. In maniera simile, per le fasi finali potrebbe essere possibile già predisporre delle chiusure così da non avere alcuna necessità di esplorare lo spazio.

I giochi di scacchi utilizzano tipicamente un database di questo tipo. All'inizio della partita, utilizzeranno mosse che sono "storicamente buone", ovvero definite sulla base dell'osservazione dei più grandi scacchisti del mondo. Alla fine, avranno una maggior capacità di computazione rispetto all'uomo, essendo per questo capaci di definire delle strategie più efficaci/efficienti.

Ricerca con Avversari

Dai giochi deterministici ai giochi stocastici

Nella vita reale si verificano molti eventi esterni che ci mettono in situazioni impreviste. Molti giochi modellano questa imprevedibilità includendo elementi casuali, come ad esempio l'uso di un dado. Si parla dei giochi stocastici.

Nel gioco del backgammon un giocatore deve lanciare due volte un dado per decidere quali mosse sono lecite.

In questo caso, chiaramente, l'algoritmo minimax o la ricerca alfa-beta *non sarebbero utilizzabili* nella loro forma base.

Possiamo però complementare gli algoritmi visti finora in maniera da introdurre dei **nodi possibilità** accanto a quelli di scelta. Così facendo, nel calcolare i valori di MIN e MAX dovremo tener conto delle probabilità dell'esperimento casuale.

In altri termini, non andremo più a calcolare i valori di massimo e minimo, ma **i valori di massimo e minimo attesi**. Da un punto di vista pratico, i nodi assumeranno un valore corrispondente alla somma del valore su tutti i risultati pesati in base alla probabilità di ciascuna azione.

I nodi terminali, quelli per cui è già noto il risultato dell'evento stocastico, saranno invece calcolati **nello stesso modo** visto precedentemente.

Ricerca con Avversari

Dai giochi deterministici ai giochi stocastici

Un albero di gioco stocastico è rappresentato in figura. I nodi possibilità sono visualizzati come cerchi, quelli di scelta come triangoli.

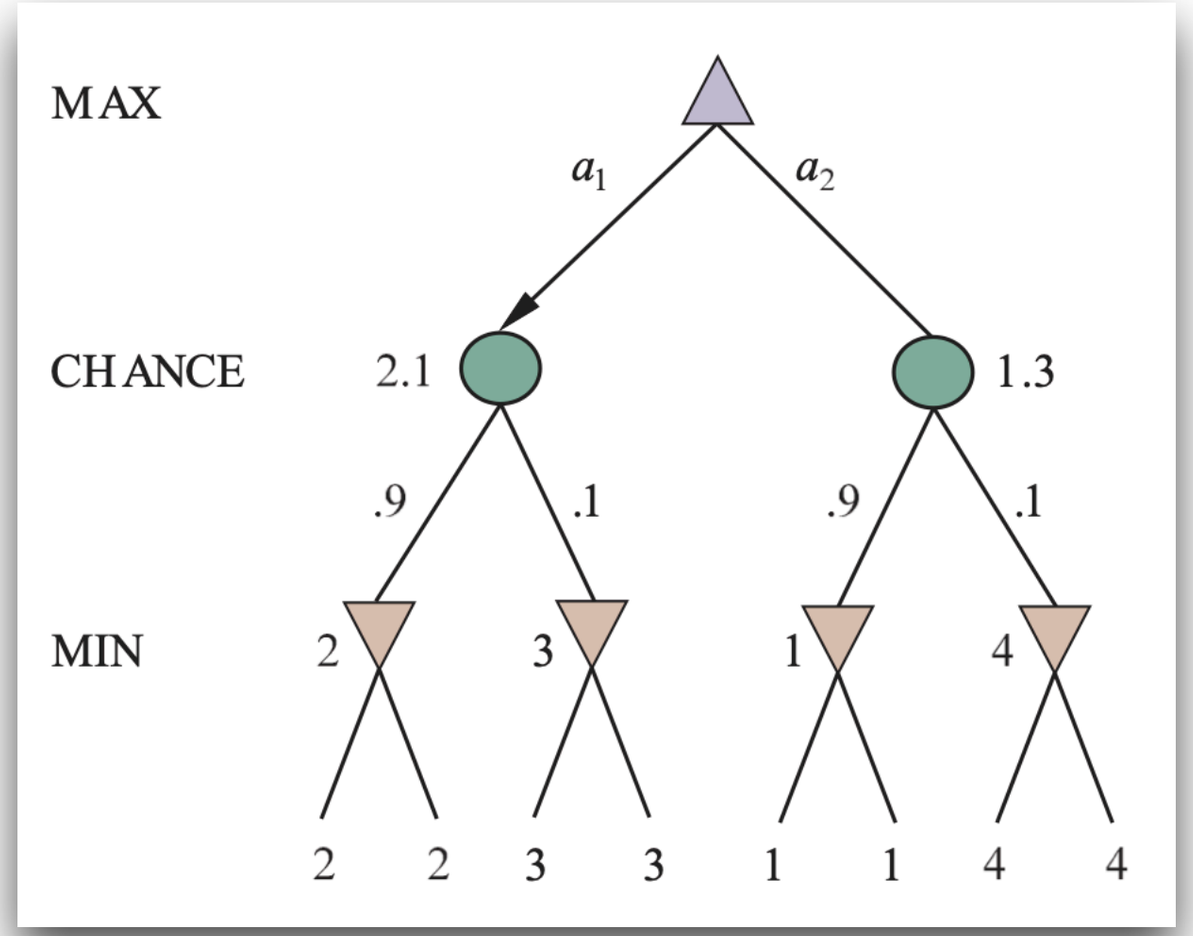
Come detto, i nodi terminali sono calcolati secondo la regola standard dell'algoritmo minimax.

Dopodiché, sappiamo che MIN con probabilità 0.9 farà 2 e con probabilità 0.1 farà 3. Dall'altro lato, con probabilità 0.9 farà 1 e con probabilità 0.1 farà 4. Quindi, deriviamo i valori attesi:

$0.9 \times 2 + 0.1 \times 3 = 2.1$

$0.9 \times 1 + 0.1 \times 4 = 1.3$

Di conseguenza, la mossa migliore per MAX è a_1.



Non avremo tempo di andare in dettaglio sulle caratteristiche di questi giochi, ma è fortemente consigliata la lettura del libro di testo, il quale propose una panoramica di come poter definire una funzione di valutazione per giochi con elementi casuali.

Ricerca con Avversari

Dai giochi deterministici ai giochi parzialmente osservabili

In questa categoria di giochi, l'incertezza sullo stato di gioco nasce interamente all'**impossibilità di accedere alle scelte fatte dall'avversario.** Ad esempio, i giochi di carte in cui le carte iniziali dell'avversario non sono note.

Intuitivamente, potremmo pensare a questi giochi come un caso particolare dei giochi stocastici: potremmo avere un dado con tantissime facce all'inizio della partita ed esprimere i nodi possibilità per ogni possibile azione dell'avversario.

Sebbene l'analogia non è corretta, può servire per definire un algoritmo efficace. Consideriamo *tutte le possibili distribuzioni di carte nascoste* e risolviamo *una per una* come se fossero un gioco completamente osservabile. A quel punto, basterà scegliere la mossa che ha il *miglior risultato calcolato in media su tutte le distribuzioni*.

Questo equivale a calcolare:

$$\operatorname*{argmax}_a \sum_s P(s)\,\textsc{Minimax}(\textsc{Result}(s,a))$$

Purtroppo però, in molti casi, il numero di possibili distribuzioni è troppo grande. Una soluzione consiste nell'applicazione di un'approssimazione Monte Carlo: invece di sommare tutte le distribuzioni, prendiamo un **campione casuale di N distribuzioni** in cui la probabilità di s di apparire nel campione è proporzionale alla probabilità P(s).

UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA

Laurea triennale in Informatica

Fondamenti di Intelligenza Artificiale

Lezione 11 - Ricerca con avversari (2)