Progetto di Intelligenza Artificiale I

TIC-TAC-TOE (TRIS)

Realizzato da:
Singh Taranjit - 858415

INDICE

INTRODUZIONE

INTRODUZIONE

L’agente intelligente si propone di giocare al gioco del tic-tac-toe (noto anche come tris). Esso si dimostra essere intelligente se riesce a far in modo di non perdere neanche ad una partita e, quando possibile, arrivare alla vittoria.

IL GIOCO DEL TRIS

Il gioco del tris

Il tris è un gioco a cui partecipano 2 giocatori. Il gioco si svolge su una grigli di 3x3 caselle. A turno, ciascun giocatore inserisce il proprio simbolo in una delle caselle libere. Vince il giocatore che, per primo, riesce a disporre tre dei suoi simboli in linea retta orizzontale, verticale o diagonale. Se tutte le posizioni vengono occupate e nessun giocatore ha vinto, allora si giunge ad un pareggio.

IL GIOCO DAL PUNTO DI VISTA MATEMATICO

Il gioco dal punto di vista matematico

Analizziamo la visione matematica del gioco. Più precisamente, ne diamo una definizione dal punto di vista della teoria dei giochi.

DEFINIZIONE:

Il tris è un gioco ad informazione perfetta a somma zero.

Il gioco dal punto di vista matematico

Affinchè un gioco sia ad **informazione completa** si richiede che ogni giocatore abbia tutte le informazioni sul contesto e sulle possibili strategie degli avversari, ma non necessariamente sulle loro azioni.

Quindi, che siano a conoscenza dello **spazio delle strategie di ogni altro giocatore** e dell'**utilità che ogni strategia ha per quel giocatore**

Un gioco è ad **informazione perfetta** se è ad informazione completa e ogni giocatore è anche a conoscenza di tutte le mosse eseguite dagli altri giocatori, fino ad quel punto del gioco.

Il gioco dal punto di vista matematico

Un gioco si dice a **somma zero** se descrive una situazione in cui il guadagno o la perdita di un partecipante è perfettamente bilanciato da una perdita o guadagno di un altro partecipante. Se alla somma totale dei guadagni dei partecipanti si sottrae la somma totale delle perdite, si ottiene zero.

Tale proprietà garantisce che ogni stato del gioco sia un ottimo paretiano: una situazione nella quale non è possibile apportare miglioramenti paretiani, cioè migliorare la situazione di uno, senza peggiorare quella di un altro.

PROPRIETA' DELL'AMBIENTE IN CUI LAVORA L'AGENTE

Proprietà dell'ambiente in cui lavora l'agente

Prima di realizzare un agente intelligente è opportuno studiare l'ambiente in cui esso andrà ad operare. Nel nostro caso, esso si ritrova a lavorare in una simulazione del gioco del tris.

L'ambiente è **completamente osservabile**, **deterministico** quando gioca l'agente e **nondeterministico** quando gioca l'avversario, **sequenziale**, **semidinamico** e **discreto**.

Proprietà dell'ambiente in cui lavora l'agente

Completamente osservabile: se l'agente è in grado di percepire lo stato completo dell'ambiente.

Deterministico: se il prossimo stato dell'ambiente è completamente determinabile dallo stato attuale e dalle azione degli agenti. L'agente è in grado di predire il prossimo stato quando è il suo turno.

Nondeterministico: se il prossimo stato dell'ambiente non è determinabile dallo stato attuale e dalle azioni degli agenti. Quando il turno è dell'avversario, l'agente sa che il prossimo stato sarà uno di quelli raggiungibili con una mossa dell'avversario, ma non è in grado di determinarlo con esattezza.

Proprietà dell'ambiente in cui lavora l'agente

Sequenziale: se ogni azione intrapresa nello stato attuale, avrà delle conseguenze negli stati prossimi. Altrimenti sarebbe episodico.

Semidinamico: se l'ambiente non cambia con il solo passare del tempo, ma le azioni degli agenti lo possono modificare.

Discreto: se il numero di percezioni e azioni distinte è finito in ogni stato dell'ambiente, altrimenti è continuo.

RICERCA DI UNA SOLUZIONE

Ricerca di una soluzione

Abbiamo visto fin qui che il tris è un gioco ad informazione perfetta e che l'ambiente è sequenziale, semidinamico e discreto. Questo facilita la rappresentazione dello sviluppo del gioco sotto forma di grafo ad albero finito. Il l'albero ha come radice la griglia vuota, mentre le foglie sono gli stati finali del gioco: griglie in cui tutte le posizioni sono state occupate, oppure situazioni in cui un giocatore ha vinto.

Ricerca di una soluzione

MINIMAX

Il minimax è un metodo per minimizzare la massima perdita possibile, che nel caso del tris è rappresentata dalla sconfitta. In ogni stato del gioco un giocatore cerca di aumentare le proprie possibilità di vittoria, mentre l'altro cercherà di diminuirle. Abbiamo visto che ogni stato del gioco è un ottimo paretiano, quindi il miglior modo di diminuire le possibilità di vittoria dell'avversario è quello di aumentare le proprie.

Ricerca di una soluzione

MINIMAX

```
minimax(nodo, profondità, turno)
        SE (profondità = 0 OPPURE nodo è un nodo terminale)
                restituisci il valore euristico del nodo.
        SE (turno = nostro)
                migliorValore = −∞
                PER OGNI figlio di nodo
                        v = minimax(figlio, profondità − 1, avversario)
                        migliorValore = max(migliorValore, v)
                restituisci migliorValore
        ALTRIMENTI  //è il turno dell'avversario
                migliorValore = +∞
                PER OGNI figlio di nodo
                        v = minimax(figlio, profondità − 1, nostro)
                        migliorValore = min(migliorValore, v)
                restituisci migliorValore
```

Ricerca di una soluzione

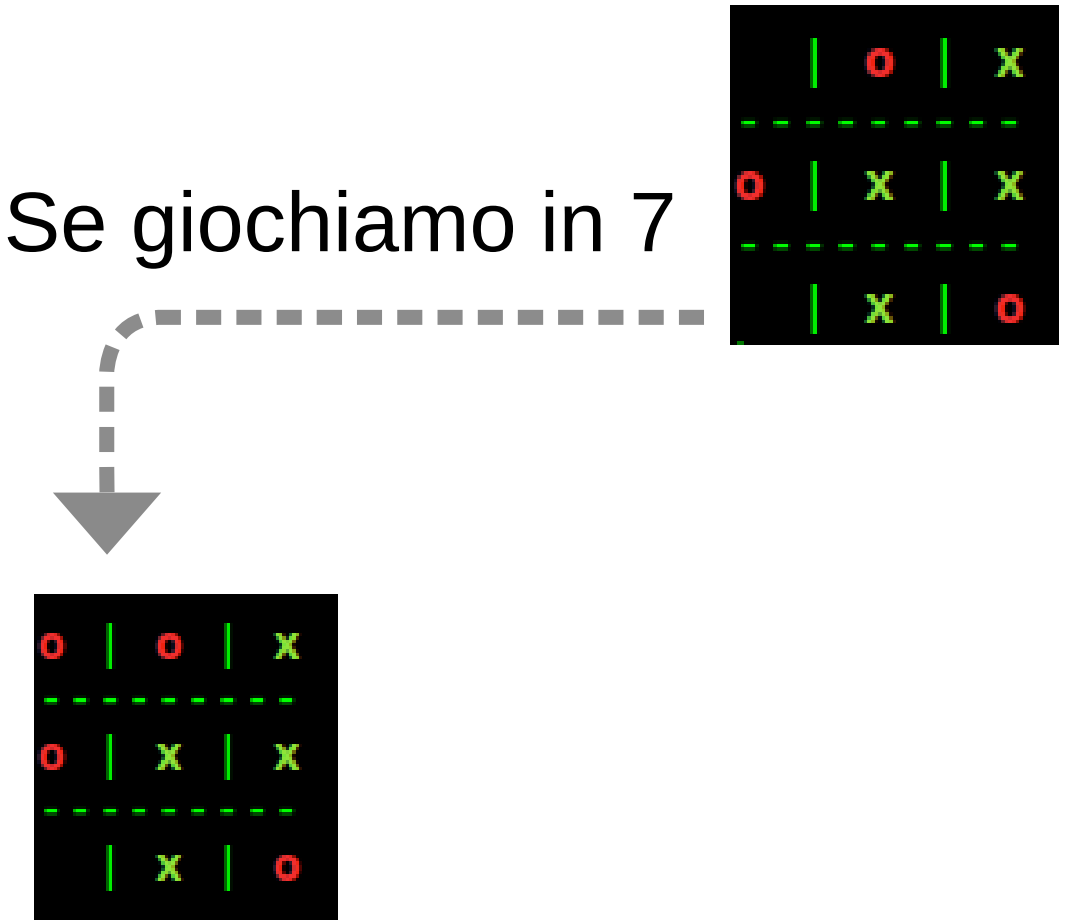
MINIMAX

Vediamo come funziona l'algoritmo minimax. Assumiamo che il nostro simbolo sia 'o' e che lo stato in cui ci troviamo sia il seguente:

Com'è possibile notare, è il nostro turno. Abbiamo 2 scelte: possiamo giocare nella posizione 7 oppure nella posizione 1. Vediamo come si evolve la partita in entrambi i casi.

Ricerca di una soluzione

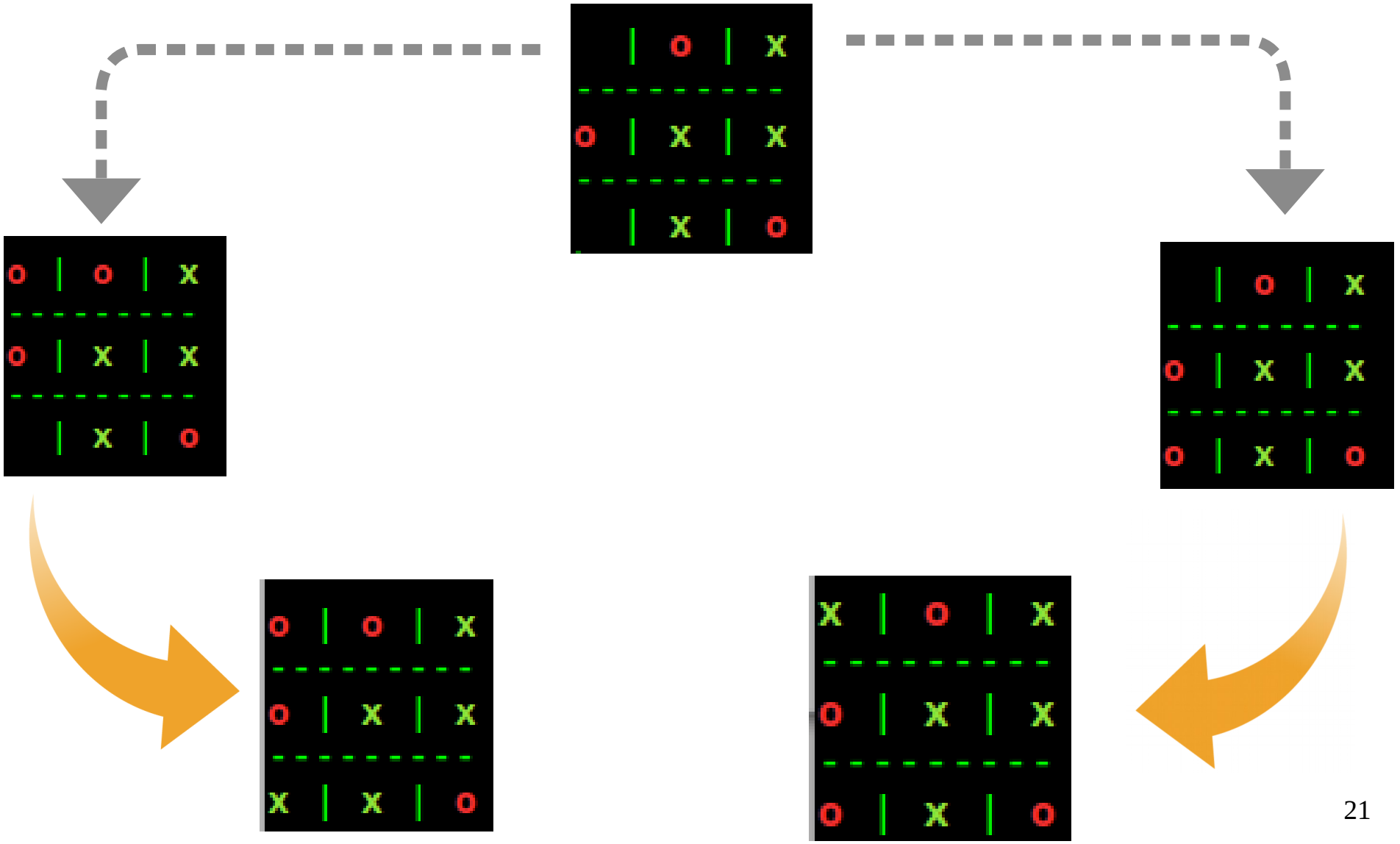
MINIMAX



Ora è il turno dell'avversario.

Ricerca di una soluzione

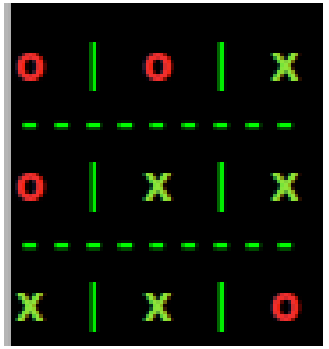
MINIMAX



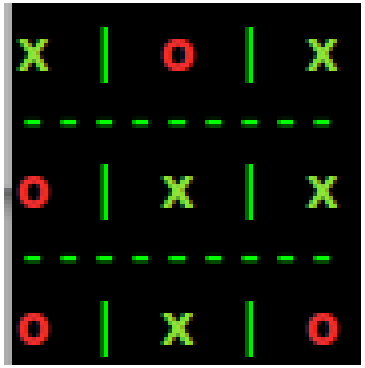
Ricerca di una soluzione

MINIMAX

Abbiamo raggiunto i nodi foglia. Dobbiamo valutare questi nodi.



In questo stato l'avversario vince, quindi valutiamo questo stato in modo negativo. Per esempio -1.

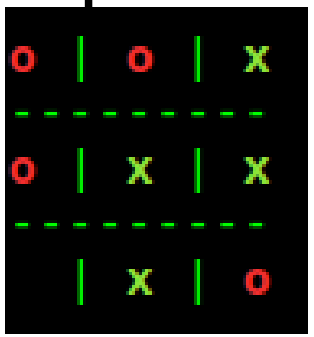


In questo stato invece la partita è giunta alla fine con un pareggio. Valutiamo questo stato con punteggio più alto del precedente: 0 per esempio.

Ricerca di una soluzione

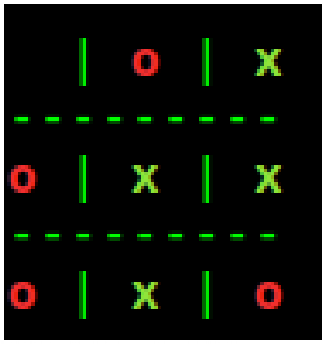
MINIMAX

L’avversario sceglierà di fare la mossa che diminuisce le nostre probabilità di vittoria, quindi quelle con un valore euristico basso:

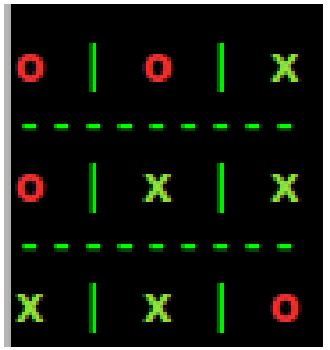


min(-1)= -1

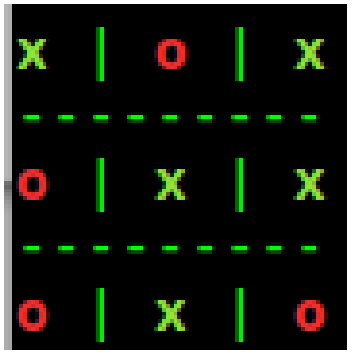
min(0)=0



-1



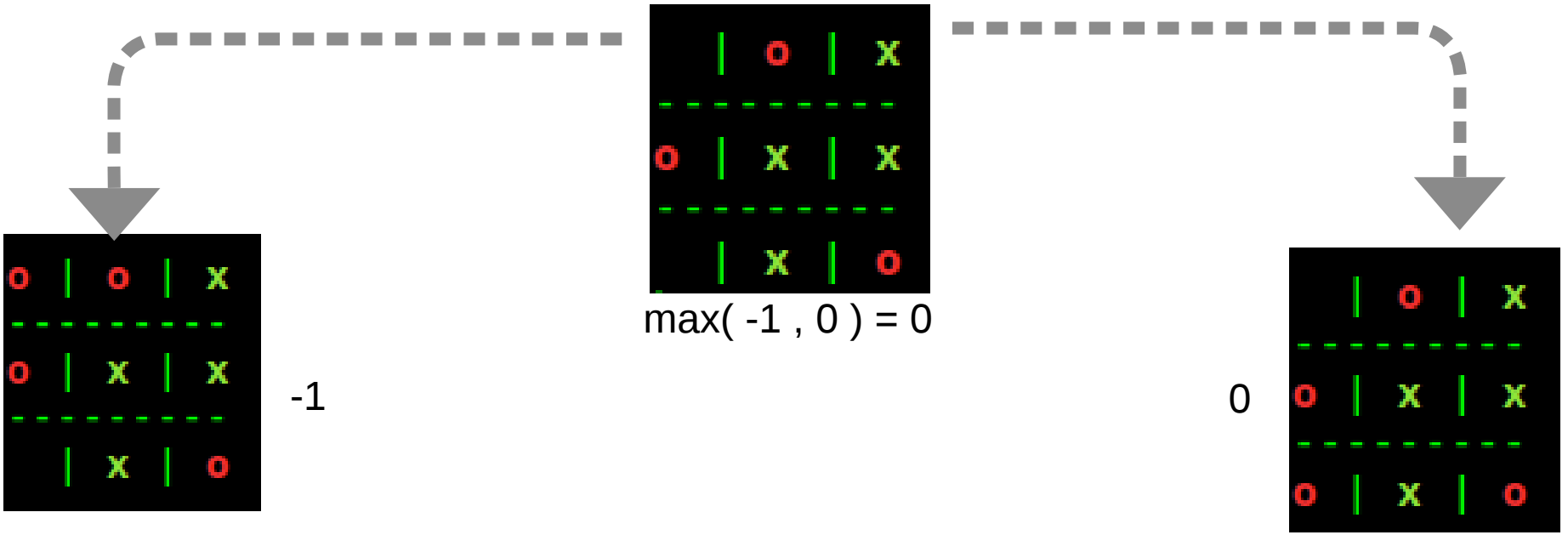
0



Ricerca di una soluzione

MINIMAX

Noi, invece, scegliamo di fare la mossa che ci porta ad un guadagno maggiore o ad una perdita minore, quindi con un valore euristico alto.



Ne segue che, per noi, la miglior mossa da fare è la seconda.

Ricerca di una soluzione

MINIMAX – potatura alpha-beta

L’algoritmo minimax, come abbiamo visto, ci fornisce la mossa migliore da effettuare. Ma per riuscire a trovarla, richiede di percorrere tutto l’albero degli stati. Nel tris ci sono meno di 9! (362,880) stati, per cui una ricerca non troppo profonda porta a buoni risultati in termini di qualità e tempi di calcolo. Per gli scacchi, invece, la questione è più complessa. Il fattore di diramazione del tris è di 4, mentre quello degli scacchi è di 35 circa. Una ricerca con profondità 12 richiederebbe troppo tempo.

Ricerca di una soluzione

MINIMAX – potatura alpha-beta

Una soluzione è data dal miglioramento dell'algoritmo minimax, noto come alpha-beta pruning (potatura alpha-beta). L'algoritmo interrompe la valutazione di una mossa se viene dimostrato che è comunque peggiore di un altra già valutata. In questo modo il numero di nodi da valutare vengono drasticamente diminuiti. Nel caso di un ordinamento pessimo dei nodi il numero di mosse valutate è uguale a quello del minimax (caso peggiore). Nel caso di ordinamento perfetto il numero di nodi valutati è la radice quadrata del numero dei nodi valutati usando il minimax (caso migliore). Negli scacchi il rapporto tra caso migliore e peggiore è circa 40^6, cioè 4 miliardi di volte più veloce.

Ricerca di una soluzione

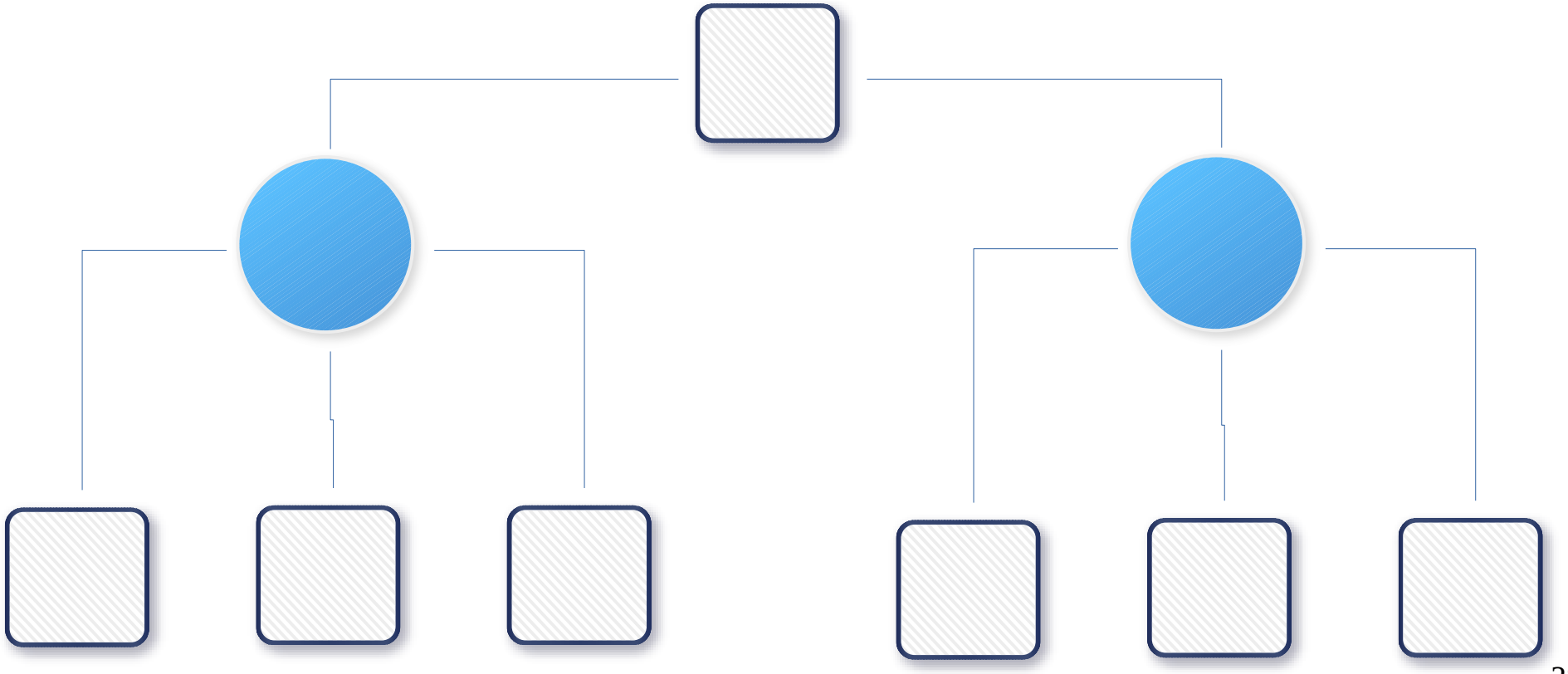
MINIMAX – potatura alpha-beta

```
alphabeta(nodo, profondità, α, β, turno)
    SE profondità = 0 O nodo è un nodo terminale
        return valore euristico del nodo
    SE turno = nostro
        v = -∞
        PER OGNI figlio del nodo
            v = max(v, alphabeta(figlio, profondità-1, α, β, avversario))
            α = max(α, v)
            SE β ≤ α
                break //taglio di β
        return v
    ALTRIMENTI
        v = ∞
        PER OGNI figlio del nodo
            v = min(v, alphabeta(figlio, profondità - 1, α, β, nostro)
                β = min(β, v)
            SE β ≤ α
                break //taglio di α
        return v
```

Ricerca di una soluzione

MINIMAX – potatura alpha-beta

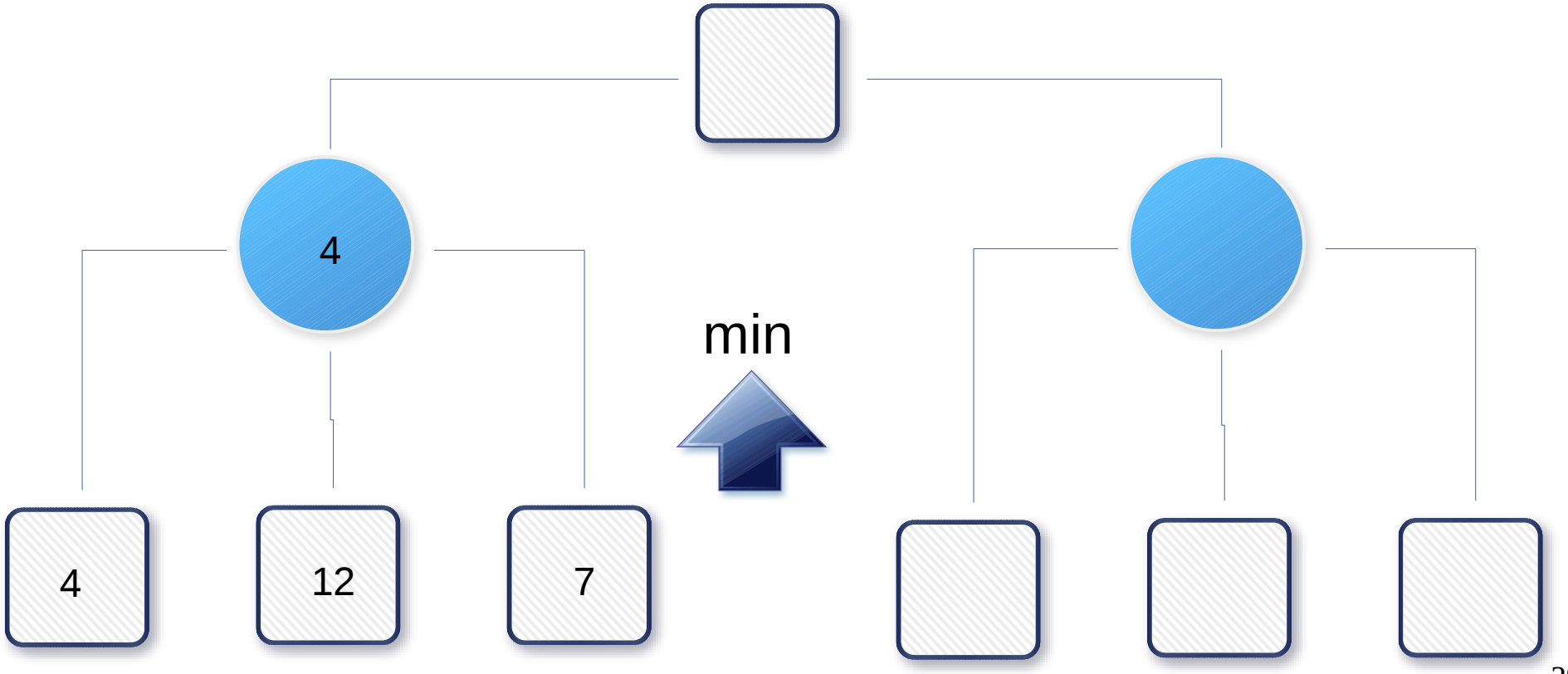
Supponiamo di trovarci in una situazione come la seguente. I rettangoli rappresentano le mosse eseguite da noi, mentre i cerchi quelle dall'avversario.



Ricerca di una soluzione

MINIMAX – potatura alpha-beta

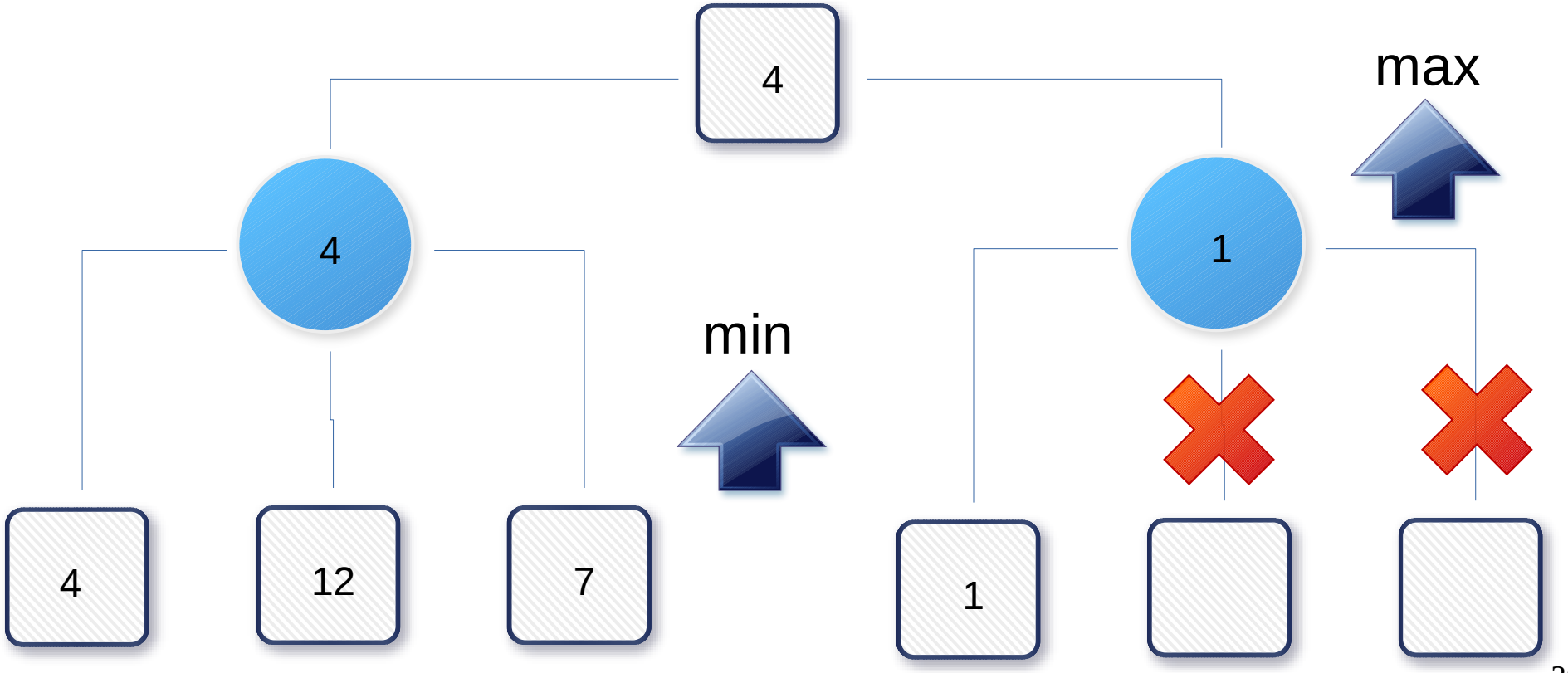
Cominciamo ad eseguire il minimax sul primo ramo e valutiamo i nodi foglia. L'avversario andrà a scegliere la mossa migliore per lui, quindi quella con il valore euristico più basso. Poi si passa al ramo successivo.



Ricerca di una soluzione

MINIMAX – potatura alpha-beta

Cominciata la valutazione del secondo ramo, vediamo che il valore euristico del primo figlio è 1. Sappiamo che andremo a scegliere il valore più basso, quindi il valore scelto sarà 1 o minore. Al passo successivo sceglieremo il valore più alto. 1 è minore di 4, quindi scarteremo quel ramo in ogni caso



Ricerca di una soluzione

MINIMAX – potatura alpha-beta

Nella pratica ci serviamo di due valori: alpha e beta. Nello specifico:

- Alpha è il limite inferiore del valore euristico di una possibile soluzione
- Beta è il limite superiore del valore euristico di una possibile soluzione.

Come la ricerca progredisce, questi valori vengono aggiornati, in modo da ridurre il più possibile l'intervallo compreso tra Alpha e Beta. Quando si valuta un nuovo nodo considerandolo parte del percorso ad una soluzione, esso è accettabile solo se:

Alpha <= N <= Beta

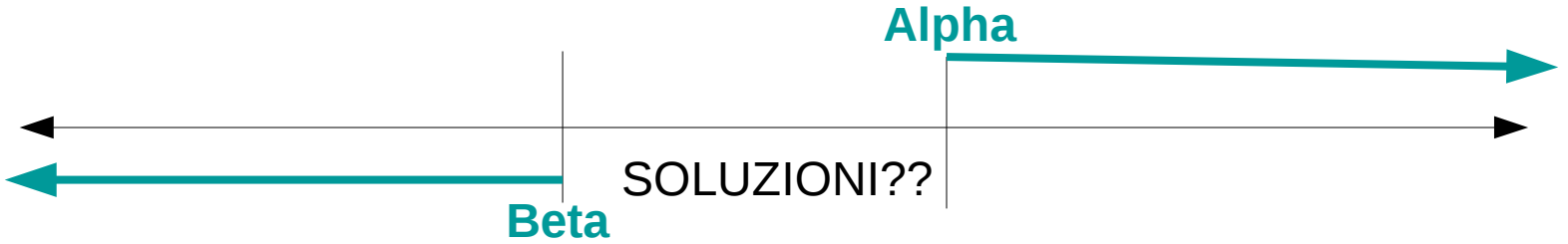
dove N è il valore euristico di tale nodo. In ogni istante, possiamo rappresentare Alpha, Beta e le soluzioni in questo modo:

Ricerca di una soluzione

MINIMAX – potatura alpha-beta

Man mano che la ricerca di una soluzione progredisce, l'intervallo tra Alpha e Beta diventerà sempre più piccolo.

Come detto, ad ogni valutazione di un nodo, i valori di Alpha e Beta si aggiornano. Quando per un nodo, la situazione diventa la seguente:



possiamo dedurre che da questo nodo in poi non ci sono soluzioni, perché non esiste un intervallo in grado di contenere tali soluzioni. Quindi, quando Alpha e Beta s'invertono, possiamo potare tutto il sotto-albero del nodo valutato, in quanto siamo certi che non possa contenere soluzioni.

TRIS IN PROLOG

Tris in Prolog

Vediamo un implementazione del tris in Prolog, che utilizza l'algoritmo minimax con potatura alpha-beta. La partita viene avviata con il il comando 'vai' che ha la seguente dichiarazione:

vai :-

retractall(won(_)),	falsifica ogni predicato della forma won(_) asserito
starting_state(S0),	S0 è lo stato iniziale → gliglia vuota
printScreen(S0),	stampa dello stato iniziale
step(S0).	Inizia il gioco.

Tris in Prolog

Lo stato iniziale è un lista di fluenti e contiene lo stato di ogni cella e il turno. ‘k’ indica la cella vuota. Viene scelto casualmente chi deve iniziare.

```
starting_state([played(1,k), played(2,k), played(3,k),
                played(4,k), played(5,k), played(6,k),
                played(7,k), played(8,k), played(9,k),
                aiTurn]) :-
                    Rand is random(2),
                    Rand==0,!.


starting_state([played(1,k), played(2,k), played(3,k),
                played(4,k), played(5,k), played(6,k),
                played(7,k), played(8,k), played(9,k),
                playerTurn]).
```

Tris in Prolog

Il predicato step fa compiere un azione all'interno del gioco.

```
step(S0):-
    trovato(S0),!,      Se S0 è uno stato finale,
    printScreen(S0).    Stampa lo stato e termina.
step(S0):-              altrimenti:
    play(S0, Fluent2Add, Fluent2Remove), si compie una mossa
    remove(Fluent2Remove,S0,Temp1), vengono tolti i fluenti non più veri
    append(Fluent2Add,Temp1,NewState), e aggiunti quelli aggiornati
    printScreen(NewState),          si stampa il nuovo stato
    step(NewState).         si procede a fare la prossima mossa
```

Tris in Prolog

Serve un predicato per valutare se uno stato è uno stato finale:

trovato(ST) :- Uno stato è finale se

player(X), un giocatore X

winningCheck(ST, X), !, ha vinto**,**

assert(won(X)). in tal caso si asserisce la sua vittoria

trovato(ST) :- Oppure se

already_played(ST,k,L), L è la lista delle posizioni ancora libere e

L=[ ], L è una lista vuota (non ci sono posizioni libere)

assert(won(no_one)). allora si può asserire che non ha vinto nessuno.

Nota: already_played(ST,k,[ ]) non avrebbe dato lo stesso risultato. Il terzo argomento deve essere una variabile (vedi specifica).

Tris in Prolog

Il predicato step(ST) chiama play(ST, Fluent2Add, Fluent2Remove).

Nel caso sia il turno del giocatore:

play(ST, [aiTurn, played(P,x)],[playerTurn, played(P,k)]) :-

member(playerTurn, ST), ci assicuriamo che sia il suo turno

ask(P), gli viene chiesto in quale posizione giocare

member(played(P, k), ST). e si controlla che tale posizione sia libera

Se tutto va a buon fine lo stato del gioco va modificato:

- Bisogna togliere dallo stato che è il turno del giocatore e che la posizione in cui ha giocato è libera (playerTurn e played(P,k))
- Bisogna aggiungere allo stato che è il turno della macchina e che la posizione in cui ha giocato il giocatore, ora è occupata dal suo simbolo (aiTurn e played(P,x)).

Tris in Prolog

Allo stesso modo funziona quando è il turno della macchina

play(ST, [playerTurn, played(Move,o)],[aiTurn, played(Move,k)]) :-

member(aiTurn, ST), ci assicuriamo che il turno sia della macchina

alphabeta(ST,_,-inf,+inf,o,5,[Move,_H]). e calcoliamo la mossa da fare

A questo punto, come prima, dobbiamo modificare lo stato del gioco:

- Non è più il turno della macchina e la posizione scelta non è più libera (aiTurn e played(Move,k))
- Il turno ora è del giocatore e la posizione scelta è occupata dal simbolo usato dalla macchina (playerTurn e played(Move,o)).

Tris in Prolog - MINIMAX

Come abbiamo visto, la scelta della mossa avviene tramite la chiamata del predicato alphabeta, vediamo come funziona.

alphabeta(Node,Move,_Alpha,_Beta,_Turn,Depth,[Move,H]):-

(Depth=0; se la profondità è zero, oppure il nodo che stiamo valutando è
winningCheck(Node,o); un nodo terminale, quindi ha vinto la macchina,
winningCheck(Node,x); oppure ha vinto il giocatore,
(already_played(Node,k,PosLibere), o non ci sono più posizioni libere
PosLibere = [ ])

)

-> (!,

winning_combinations(Comb), Allora valutiamo il nodo e ne
recursive_heuristic(Node,Comb,H) restituiamo il valore euristico

).

Il valore euristico H viene restituito in coppia con Move che contiene la mossa effettuata per raggiungere questo stato.

Tris in Prolog - MINIMAX

Se la profondità raggiunta non è zero o il nodo che stiamo valutando non è un nodo finale, allora procediamo come segue:

alphabeta(Node, _Move, Alpha, Beta, Turn, Depth, Heur):-!,

already_played(Node,k,Moves), Moves = posizioni ancora libere

(Turn=o ->ActualV=[[invalidMove],-inf] ; In base al turno impostiamo
ActualV=[[invalidMove],+inf]), ActualV con il valore più basso per quel giocatore: -inf per macchina e +inf per l'avverario

iterator(Node,Moves,Alpha,Beta,Turn,Depth,Heur,ActualV).

Chiamiamo iterator che si occuperà di iterare la lista delle posizioni libere (Moves) e sviluppare l'albero di gioco facendo giocare nello stato presente (Node) il giocatore (Turn) in ciascuna delle posizioni nella lista.

Tris in Prolog - MINIMAX

Iterator deve fermarsi quando la lista delle mosse possibili sarà vuota:

iterator(_Parent,[ ],_Alpha,_Beta,_Turn,_Depth,H,H):-!.

Il valore euristico da ritornare è esattamente il valore che abbiamo trovato finora (H).

Altrimenti, iterator ha il compito di decidere se continuare a valutare i nodi figli, oppure è il caso di troncare il sotto-albero in quanto è dimostrabile che in esso non possiamo trovare la soluzione.

iterator(_Parent, _ListMoves, Alpha, Beta, _Turn, _Depth, H, H):-

minnn(Beta,Alpha, Min), Beta=Min, !.

Se beta è diventato minore o uguale ad alpha, è inutile proseguire a valutare il sotto-albero. Come prima ritorniamo Il valore euristico trovato finora (H).

Tris in Prolog - MINIMAX

Se la lista delle mosse possibili non è ancora vuota e non è il caso di troncare il sotto-albero del nodo, allora generiamo un nodo figlio eseguendo nel nodo attuale una delle mosse disponibili e proseguiamo a valutare tale nodo. Se è il turno della macchina:

```
iterator(Parent, [Head|Tail],Alpha, Beta, o, Depth,Heur, ActualV):- !,
    new_state(Parent,Head, o,Node)     Node = nodo figlio
    NewDepth is Depth - 1,
    alphabeta(Node,Head, Alpha, Beta, x, NewDepth, [_ExMove,Temp]),
    maxcc(ActualV,[Head,Temp],NewV),
    maxnc(Alpha, NewV, NewAlpha),
    iterator(Parent,Tail, NewAlpha, Beta, o, Depth, Heur, NewV).
```

Eseguito alphabeta sul nodo figlio, abbiamo ottenuto un valore euristico(Temp) associato ad una mossa ExMove. Il valore euristico del nodo è il massimo tra il valore calcolato finora e Temp. Il nuovo Alpha è il massimo tra il vecchio Alpha e il nuovo valore del nodo.

Tris in Prolog - MINIMAX

Nel caso tocchi al giocatore a giocare, la sequenza delle azioni è la stessa

```
iterator(Parent, [Head|Tail],Alpha, Beta, x, Depth,Heur, ActualV):-  !,
    new_state(Parent,Head, x,Node),
    NewDepth is Depth - 1,
    alphabeta(Node, Head, Alpha, Beta, o, NewDepth, [_ExMove,Temp]),
    mincc(ActualV,[Head,Temp],NewV),
    minnc(Beta, NewV, NewBeta),
    iterator(Parent,Tail, Alpha, NewBeta, x, Depth, Heur, NewV).
```

Le differenze sostanziali sono:

- A cambiare valore non è alpha ma beta
- Il nuovo valore euristico del nodo e quello di beta sono il minimo tra i valori presenti e non il massimo come nel caso precedente.

Tris in Prolog - Euristica

Vediamo come viene valutato un nodo.

```
recursive_heuristic(St,[Comb],H):-!,
    heuristic(St,Comb,H).


recursive_heuristic(St,[Comb1|Comb2],H3):-
    heuristic(St,Comb1,H1),
    recursive_heuristic(St,Comb2,H2),
    sum(H1,H2,H3).
```

Il valore euristico di un nodo è la somma dei valori euristici generati da ciascuna combinazione di 3 posizioni, giocando nelle quali, si può vincere una partita a tris. recursive_heuristic invoca ricorsivamente heuristic sul nodo e una delle combinazioni presenti. La somma dei valori restituiti è il valore euristico del nodo.

Tris in Prolog - Euristica

Data una combinazione di posizioni [E0,E1,E2], contiamo in quante di queste posizioni ha giocato la macchina (My) e in quante il giocatore (Enemy). In base a questi valori, viene calcolato il valore euristico della combinazione, come segue:

		M Y			
		0	1	2	3
E N E M Y	0	0	99	9999	999999
	1	-99	0	0	0
	2	-9999	0	0	0
	3	-999999	0	0	0

Tris in Prolog - Euristica

Ciò in prolog può essere tradotto nel seguente modo:

```
heuristic(St,[E0,E1,E2],H):-
   (member(played(E0,o),St) -> V1=1;V1=0),
   (member(played(E0,x),St) -> EV1=1;EV1=0),
   (member(played(E1,o),St) -> V2 is V1+1 ; V2=V1),
   (member(played(E1,x),St) -> EV2 is EV1+1 ; EV2=EV1),
   (member(played(E2,o),St) -> My is V2+1 ; My=V2),
   (member(played(E2,x),St) -> Enemy is EV2+1 ; Enemy=EV2),
   Temp1 is 100 ^ My,
   Temp2 is -(100 ^ Enemy),
   (((My=0,Enemy=0);(My>0,Enemy>0)) -> (H=0);true),
   (((My=0;Enemy=0), var(H)) -> H is Temp1 + Temp2;true).
```

calcolo delle posizioni occupate da ciascun giocatore

Calcolo del valore euristico finale