Università degli Studi di Napoli Federico II

Corso di Laurea in Informatica

A.A. 2009/2010

Progetto di "Sistemi ad Intelligenza Distribuita"

“Connect4”

Docente:

Prof.ssa S.Rossi

Autori:

Davino Marcella N97/22

De Martino Pasquale N97/31

1. RICHIESTE DEL PROGETTO

Creare due sistemi di IA che siano in grado di giocare al gioco da tavolo noto come "*FORZA 4*". Il primo deve sviluppare l'albero di ricerca utilizzando un'implementazione dell'algoritmo Minimax, il secondo dell'algoritmo Alpha-Beta Pruning.

I due sistemi devono essere in grado di giocare l'uno contro l'altro e di sfidare un giocatore umano; a tal proposito è richiesta la creazione di un'interfaccia grafica che dovrà:

- Fornire un sistema che permetta di specificare la profondità dell'albero di ricerca per ogni sistema di IA
- Fornire un metodo di interazione tra il giocatore umano ed il sistema
- Fornire ad ogni istante la configurazione della scacchiera di gioco.

Alla fine di ogni partita deve essere visualizzato un resoconto per ogni sistema di IA:

- La vittoria o la sconfitta
- Il numero totale di nodi generati da ogni algoritmo
- Il tempo medio di calcolo per ogni turno di gioco

2. BACKGROUND

Il gioco "*FORZA 4*" rientra nella categoria dei giochi detti a somma zero:

- l'*ambiente* è rappresentato dalla scacchiera di gioco che è:
    - completamente osservabile (entrambi i giocatori conoscono in ogni momento la configurazione della scacchiera e lo stato del gioco)
    - strategico (l'ambiente è modificato solo dalle azioni dei due agenti, entrambi gli agenti hanno a disposizione lo stesso set di azioni e conoscono perfettamente l'esito di ciascuna azione)
    - statico (durante il turno di gioco di ciascun agente l'ambiente non muta mentre questo sta decidendo quale mossa eseguire)
- la *comunicazion*e tra i due agenti avviene solo attraverso il tavolo di gioco
- i valori di *funzione di utilità* alla fine del gioco per i due giocatori sono a somma zero quindi se uno vince l'altro necessariamente perde. Per questo gioco è prevista anche la configurazione di pareggio.

Ciascun agente gioca per massimizzare la propria utilità il che comporta che venga automaticamente minimizzata l'utilità dell'avversario proprio per le caratteristiche del gioco.

Per i giochi a somma zero, la scelta dell'azione da eseguire è dettata dal criterio *MINIMAX* : l'agente che gioca per primo sarà denominato MAX, mentre il suo avversario sarà denominato MIN; ad ogni passo MAX dovrà scegliere la mossa avente il valore *MINIMAX* più alto ossia la mossa migliore in risposta alla migliore mossa dell'avversario.

Il valore *MINIMAX* viene calcolato simulando l'intera partita con l'ausilio di un albero and/or che sarà così strutturato:

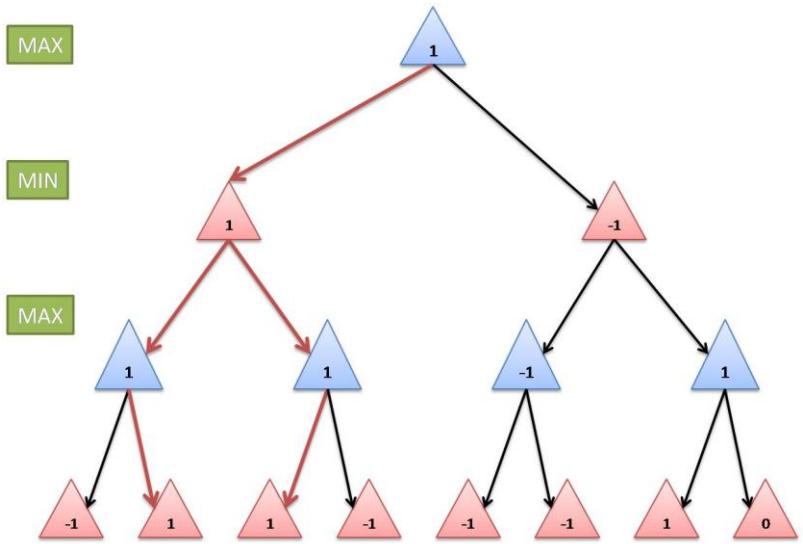
- Ogni nodo rappresenta una configurazione di gioco o stato.
- I nodi foglia rappresentano stati terminali di gioco ossia stati da cui è inutile proseguire il gioco oppure stati in cui è presente la vittoria di uno dei due giocatori.
- La radice dell'albero è la configurazione vuota.
- Ogni livello dell'albero rappresenta un turno di gioco, quindi, partendo dalla radice che viene assegnata al giocatore MAX, si alternano livelli di MAX con livelli di MIN.

Ai nodi foglia vengono assegnate delle valutazioni a seconda se questi rappresentino uno stato di vittoria per MAX (valore 1), uno stato di Vittoria per MIN (valore -1) o uno stato di pareggio (valore 0).

Tali valori risalgono a ritroso l'albero AND/OR, fino alla radice, secondo la seguente politica:

- Ciascun nodo MAX sceglierà il valore di funzione di utilità massimo tra i valori riportati dai figli (MAX gioca per massimizzare la propria utilità)
- Ciascun nodo MIN sceglierà il valore di funzione di utilità minimo tra i valori riportati dai figli (MIN gioca per minimizzare l'utilità di MAX)

Nell'immagine di seguito è dato un esempio di albero *MINIMAX.*



Esempio di albero minimax

Come si può vedere dall'immagine, simulando tutta la partita, il giocatore MAX può già definire la propria strategia di gioco. La mossa evidenziata in rosso, infatti, lo porterà a vittoria sicura dal momento che qualsiasi mossa di risposta sceglierà MIN, MAX potrà sempre raggiungere uno stato di vittoria.

Dal momento che l'esplorazione dell'albero avviene in DFS e non è necessario mantenere in memoria tutti i nodi, la complessità di spazio dell'algoritmo di *MINIMAX* è $O(mb)$, mentre, per quanto riguarda la complessità di tempo, dovendo generare tutti i nodi dell'albero è $O(b^m)$ dove b è il fattore di allargamento dell'albero ed m è la profondità massima dell'albero ed è quindi una complessità esponenziale sulla profondità dell'albero.

L'esempio della figura è, un esempio molto semplice in cui ogni giocatore ha sole 2 mosse a disposizione (fattore di allargamento) e l'intera partita si esaurisce nell'arco di 4 turni di gioco(max profondità dell'albero). Il numero di nodi generati, pari a 15, è molto esiguo e facile da calcolare.

Il caso del "*FORZA 4*" è ben diverso; il branching factor massimo, legato al numero massimo di mosse applicabili su un generico stato, è pari a 7 (una per ogni colonna del tabellone) e nel caso peggiore il numero dei turni di gioco è pari a (8x7) che corrisponde al numero di caselle del tabellone (di seguito poi si vedrà che non è proprio corretto dal momento che, per la natura intrinseca del gioco e per opportuni controlli implementati, non sarà mai possibile riempire completamente la scacchiera ma il gioco termina non appena non risulterà più possibile per entrambi i giocatori vincere).

Risulta evidente che per problemi caratterizzati da branching factor notevoli (come appunto il caso di "Forza 4") la simultazione dell'intera partita, quindi la conseguente generazione dell'albero di ricerca, richiederebbe un notevole sforzo computazionale rendendo inaccettabili i tempi di risposta dell'agente.

A tal proposito si fa uso di una funzione euristica che, dato in input uno stato, fornisce una stima di quanto questo sia vicino ad uno stato di goal. Essendo un gioco a somma 0, valori positivi indicano uno stato favorevole al giocatore MAX, i valori negativi sono associati a stati favorevoli per MIN, ∅ rappresenta una condizione di parità; ne va da se che +∞ è il valore ritornato in caso di vittoria di MAX mentre -∞ se a vincere è MIN.

Grazie all'applicazione di tale euristica, si sceglie di generare l'albero non completamente ma fino ad un limite prefissato detto *"orizzonte"* e calcolare la funzione euristica sugli stati associati ai nodi foglia di tale albero; a questo punto, seguendo le politiche di backup dei valori esposte precedentemente, è possibile scegliere l'azione migliore da intraprendere in tempo ridotto.

Risulta naturale notare che fondamentali per la buona riuscita di tale metodologia sono:

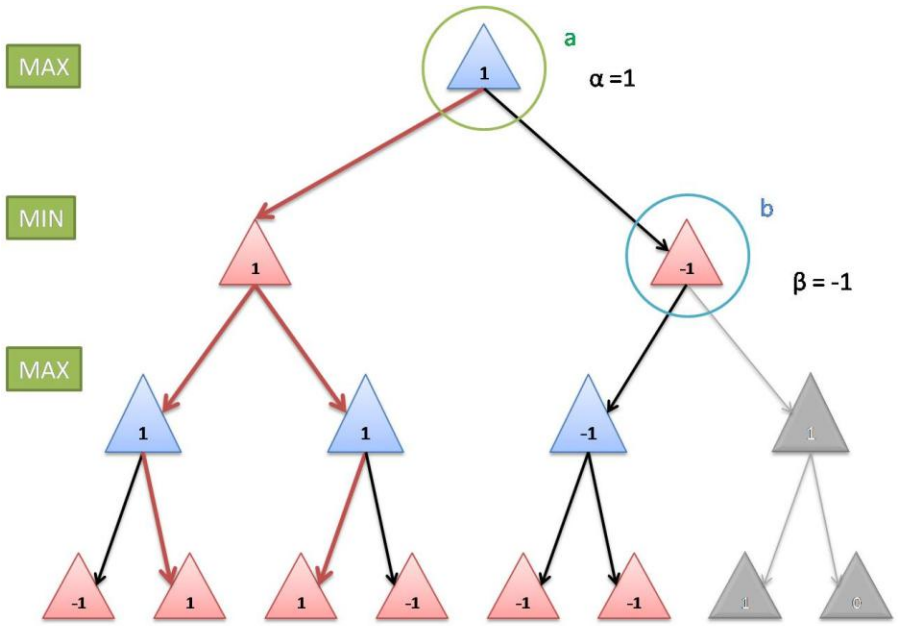
- la definizione di una buona funzione euristica che sia capace di stimare correttamente gli stati
- la scelta di un valore per l'orizzonte che dia la possibilità di guardare abbastanza avanti nel tempo ma che non comporti un carico di calcolo eccessivo.

Altra osservazione da fare è che l'albero di *MINIMAX* completo rappresenta una simulazione di *tutte* le possibili partite, e, di conseguenza, consente di scegliere una strategia a priori e di applicarla durante il gioco senza dover rieseguire delle elaborazioni.

Nel momento in cui l'albero di *MINIMAX* non è completo e i valori su cui ci si basa sono delle stime, ciò non è più possibile. Ad ogni turno di gioco la radice dell'albero sarà lo stato corrente e verrà sviluppato l'albero fino alla profondità prestabilita. La scelta non potrà essere dell'intera strategia ma della singola azione in base alle informazioni parziali che si possiedono.

L'algoritmo MINIMAX non è da considerarsi una strategia propriamente applicabile, bensì un metro di valutazione per il contronto di metodi più ottimizzati.

Una versione più intelligente del metodo MINIMAX è data dall'algoritmo *ALPHABETA PRUNING* che Seguendo gli stessi concetti del MINIMAX naive, genera l'albero di ricerca evitando, però di sviluppare rami inutili.



Esempio di operazione di pruning

Come si può vedere nell'immagine precedente, il ramo destro del nodo *b* è stato "potato" poiché non avrebbe in alcun modo influenzato la valutazione dell'albero. Dalla visita del primo sottoalbero, il nodo *a* ha ottenuto un 1. Nel secondo sottoalbero di a, *b* ha già visitato il suo ramo sinistro ottenendo un -1. La visita del ramo destro risulta inutile perché se da tale ramo si ottenesse un valore maggiore di -1 questo valore non risalirebbe nell'albero perché figlio di un nodo MIN che sicuramente preferirebbe il -1; se si ottenesse un valore minore di -1 comunque non interesserebbe la radice dell'albero che è MAX ed ha già a disposizione un 1.

Vengono quindi introdotte due variabili:

- **α**: Indica, per un nodo *MAX*, il massimo valore risalito dai figli *MIN* fino a quel momento e non può mai diminuire.
- **β**: Indica, per un nodo *MIN*, il minimo valore risalito dai figli *MAX* fino a quel momento e non può mai aumentare.

I valori di α/β di un nodo (a seconda se è un nodo di *MAX* o di *MIN*) vengono aggiornati man mano che vengono completate le espansioni dei figli.

La ricerca viene interrotta provocando la "potatura" dei rami figli di un nodo in due casi:

- Il nodo è *MIN* ed ha un β ≤ dell' α di un nodo *MAX* suo antenato.
- In nodo è *MAX* ed ha un α ≥ del β di un nodo *MIN* suo antenato.

Con l' *ALPHABETA PRUNING* i tempi di esecuzione migliorano nel seguente modo:

- Nel caso migliore (nodi *MIN* ordinati in ordine descescente, nodi *MAX* ordinati in ordine crescente) il tempo di esecuzione è $O\left(b^{\frac{h}{2}}\right)$
- Nel caso peggiore il tempo di esecuzione è uguale al *MINIMAX.*
- Nel caso medio $O\left(b^{\frac{3h}{4}}\right)$

Inquadrato il problema e il contesto in cui ci si trova si rendono necessari:

a) Una rappresentazione compatta e pratica dello stato del gioco.

b) Una funzione successore che dato uno stato in input, in base alle azioni su esso applicabili, genera gli stati immediatamente raggiungibili

c) Una funzione euristica che valuti le foglie dell'albero incompleto.

d) Un goal test che individui se un nodo è terminale o meno

3. DESCRIZIONE DELLA SOLUZIONE PROPOSTA

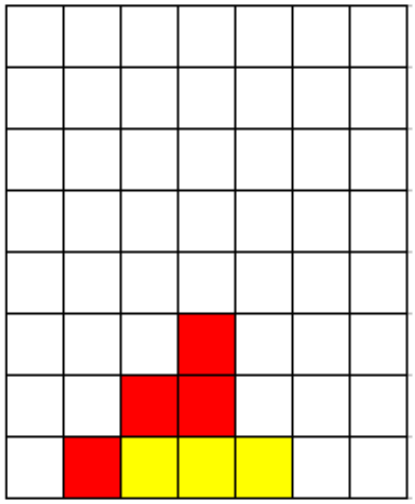
3.1. DEFINIZIONE DELLO STATO

Lo stato deve essere una rappresentazione compatta e facilmente manipolabile della configurazione del gioco in ogni momento.

La scacchiera di gioco è una griglia 8x7 (8 righe e 7 colonne) le cui caselle possono essere in ogni istante vuote, occupate da una pedina del giocatore rosso oppure occupate da una pedina del giocatore giallo; i due giocatori si alternato nell'inserimento delle pedine.

Viene naturale quindi definire lo stato del gioco mediante una matrice 8x7 alla quale è affiancato un intero che sta ad indicare il turno.
Ogni cella della matrice può avere uno dei seguenti valori interi:

- **0:** casella vuota;
- **1:** casella contenente una pedina del giocatore rosso;
- **-1:** casella contenente una pedina del giocatore giallo.



0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	1	0	0	0
0	1	-1	-1	-1	0	0

Rappresentazione di uno stato

3.2. DEFINIZIONE ACTIONSET

Ad ogni turno di gioco ciascun giocatore inserisce una pedina nel tabellone. Scelta una delle 7 colonne, la pedina verrà inserita dal top della scacchiera e fatta cadere nella colonna specificata in modo da occupare la prima posizione libera.

Le pedine, quindi, andranno gradualmente ad occupare le colonne specificate dal basso verso l'alto e la loro posizione sarà determinata dalla configurazione della scacchiera.

I giocatori dovranno, perciò, specificare solo la colonna in cui desiderano inserire la propria pedina.

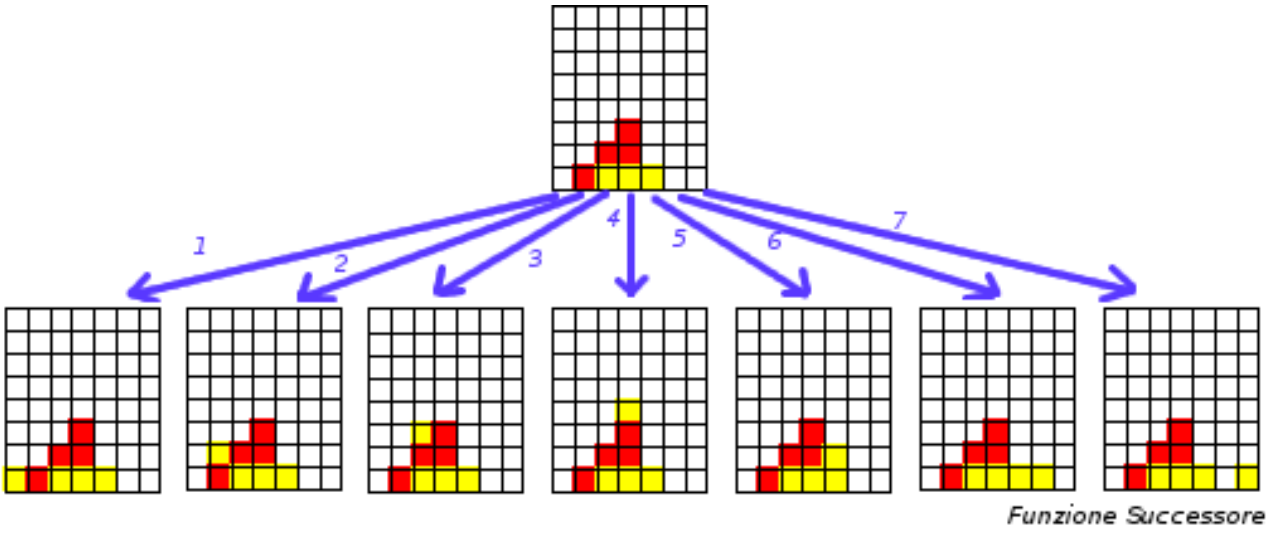
A tal proposito ogni giocatore avranno a disposizione le azioni dell'insieme

$$ActionSet = \{1,2,3,4,5,6,7\}$$

Le azioni dell' insieme ActionSet non saranno applicabili a tutti gli stati, come esposto nel paragrafo seguente

3.3. DEFINIZIONE DELLA FUNZIONE SUCCESSORE

La funzione successore genera gli stati legali ottenibili applicando le azioni dell'insieme $ActionSet$ sullo stato attuale.



Funzione Successore

Se una delle azioni non è applicabile sullo stato corrente (ad esempio perchè una delle colonne è totalmente riempita) la funzione successore restituisce in output lo stato attuale.

3.4. DEFINIZIONE DELLA FUNZIONE EURISTICA

Dal momento che i due algoritmi utilizzati non esplorano tutto l'albero di ricerca simulando tutte le possibili partite, ma si limitano a sviluppare di volta in volta solo porzioni dell'albero, si rende necessario l'utilizzo di una funzione euristica che permetta di stabilire quanto un generico stato si avvicini al goal.

Una funzione di valutazione di uno stato può essere calcolata come somma pesata di diverse *features,* caratteristiche: $e(s) = \sum_{i=1}^{n} w_i f_i(s)$.

Nel nostro caso sono state individuate due diverse euristiche che identificano due diverse features di uno stato: la prima fornisce una misura della vicinanza dello stato dal goal; la seconda fornisce una stima di quanto lo stato possa essere favorevole per gli sviluppi futuri del gioco.

Le due euristiche sono state denominate *EURISTICA PRESENTE* ed *EURISTICA FUTURA*.

Ciò che deve essere ancora definito è il peso da attribuire a ciascuna euristica.

Durante l'analisi del problema è stato appurato che il fattore più rilevante nella scelta di uno stato obiettivo è la sua vicinanza ad uno stato di goal; a parità di tale fattore risulta più conveniente

scegliere lo stato che offre maggiori prospettive per il futuro di conseguenza il valore totale dell'euristica sarà: $g(S) = 10 \times EuristicaPresente(S) + EuristicaFutura(S)$.

Di seguito viene data una descrizione di entrambe le euristiche.

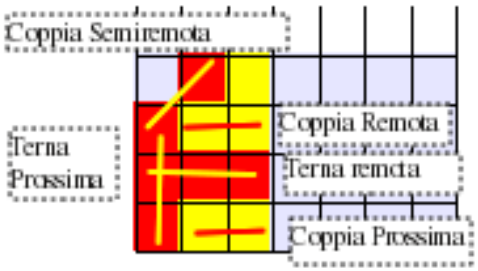
3.4.1. EURISTICA PRESENTE

La funzione di valutazione offre una stima della distanza dello stato attuale ad uno stato di goal. Per farlo, prende in considerazione le configurazioni parziali già formate, cioè per ogni giocatore conta le terne e le coppie di pedine già allineate, le quali incrementano la probabilità che il giocatore raggiunga il goal da lì a poco.

Per ogni npla si calcolano i seguenti parametri:

- Numero di pedine di cui è composta e quindi il numero di pedine necessarie per arrivare al goal
- Numero di pedine ausiliarie necessarie per completare la serie (ad es. in presenza di colonne vuote)

A tal proposito le serie parziali, presenti nello stato, vengono suddivise in 5 classi, ad ognuna delle quali è associato un punteggio:



Tipologia		**Punteggio**
Terna Prossima	(X X X _), (X X _ X)	4
Coppia Prossima	(X X _ _)	3
Terna Remota	(X X X), (X X X)	2
Coppia Remota	(X X)	1
Coppia Semiremota	(X X _), (X X _)	1.5

TABELLA 1: FATTORI MOLTIPLICATIVI DELLE CONFIGURAZIONI PARZIALI

Per ogni giocatore si calcola il numero totale di configurazioni parziali:

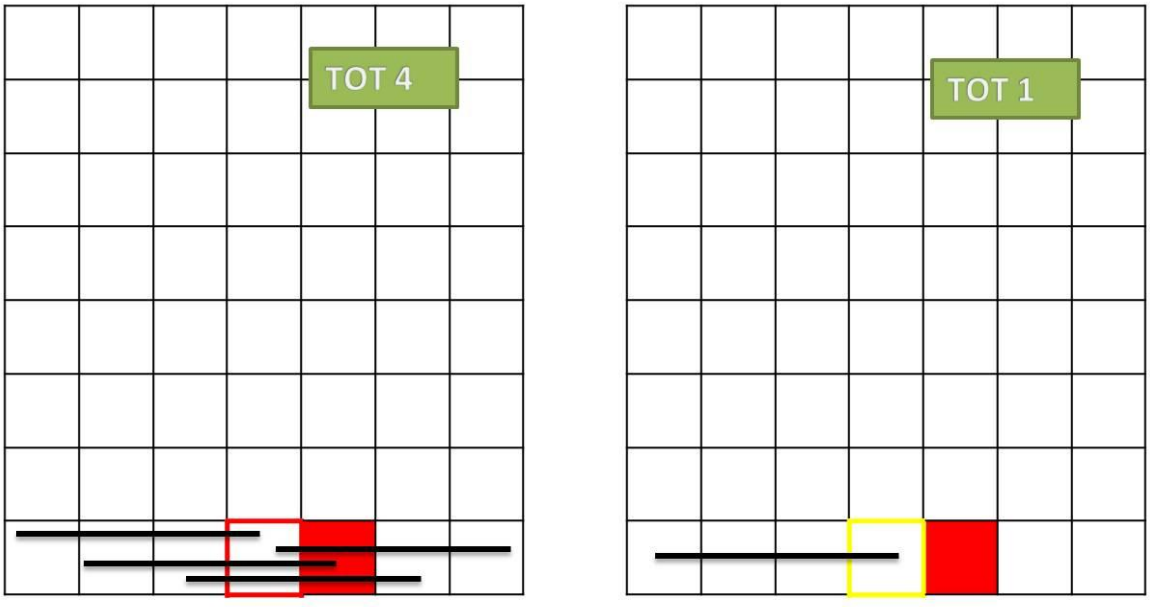
4x #(Terne Prossime) + 3x #(Coppie Prossime) + 2x #(Terne Remote) + 1x #(Coppie Remote) + Base(1,5x #(Coppie Semiremote))

La differenza tra il risultato ottenuto per il giocatore rosso e quello ottenuto per il giocatore giallo rappresenta una prima valutazione dello stato che risulta essere favorevole al giocatore giallo se negativa, in caso contrario risulta più conveniente per il giocatore rosso.

Se una delle serie è formata da almeno 4 pedine rosse la funzione restituisce +∞, -∞ se le pedine sono del giocatore giallo, segnalando, così, che lo stato è un goal.

3.4.2. EURISTICA FUTURA

Mentre l'euristica presente si preoccupa di valutare quanto uno stato sia "buono" in quel momento, rappresentando, quindi, una sorta di fotografia del gioco, l'euristica futura fornisce informazioni su quanto uno stato sia promettente nell'immediato futuro, ossia di quanti "*FORZA 4*"sono ancora potenzialmente realizzabili.
Quindi, mentre nell'euristica presente vengono analizzate le pedine già posizionate, nell'euristica futura vengono analizzate le prime posizioni libere per ogni colonna, quelle quindi immediatamente occupabili nelle mosse successive.
Per ogni "primo vuoto" viene analizzato l'intorno nelle quattro direzioni (verticale, orizzontale, diagonale nord-est/sud-ovest e diagonale nord-ovest/sud-est) dal punto di vista di entrambi i giocatori contando le caselle vuote immediatamente occupabili e le caselle occupate dal giocatore stesso. Per ogni direzione quindi viene calcolato il numero di "*FORZA 4*"realizzabili tenendo conto anche di tutte le possibili combinazioni così come chiarito dalla seguente figura.



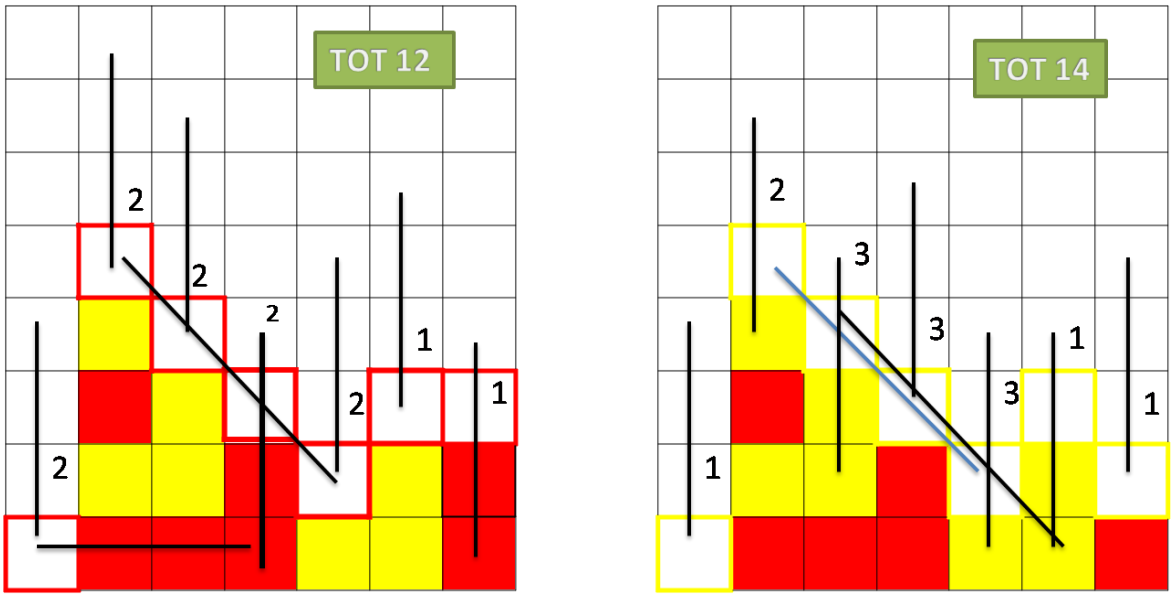
Esempio di conteggio dei forza 4 disponibili – Direzione Orizzontale

Nell'immagine precendente viene analizzata la direzione orizzontale della casella centrale dal punto di vista di entrambi i giocatori:

- Il giocatore rosso ha orizzontalmente 5 vuoti immediatamente occupabili ed una casella rossa totalizzando un totale di ben 6 caselle favorevoli, che, unite alla casella centrale, costituiscono un potenziale di ben 4 "*FORZA 4*"!
- Il giocatore giallo ha solo 3 caselle libere alla propria sinistra mentre alla propria destra la presenza della casella rossa blocca qualunque formazione favorevole totalizzando, quindi, un unico "*FORZA 4*"potenziale.

Dato il numero di caselle favorevoli, il numero di "*FORZA 4*" è ottenuto dalla seguente espressione $nForza4 = [(nCaselle - 4) + 1]$.
Da notare che la casella in esame è essa stessa vuota e favorevole, quindi deve essere inclusa nel conteggio.

Nella seguente immagine è fornito un esempio più completo di calcolo dell'euristica.



Esempio di valutazione di uno stato dal punto di vista di entrambi i giocatori

Come si può vedere nell'immagine, a differenza dell' "*euristica presente*" in cui ciascuna formazione viene conteggiata una e una sola volta, per questa euristica ogni casella indica il numero esatto dei potenziali "*FORZA 4*"in cui è coinvolta nonostante questo significhi di fatto contarli più volte.

Altra osservazione da fare riguarda i vuoti. I vuoti presi in esame sono solo quelli immediatamente occupabili mentre i vuoti senza caselle sottostanti sono considerati come sfavorevoli al pari delle caselle avversarie.

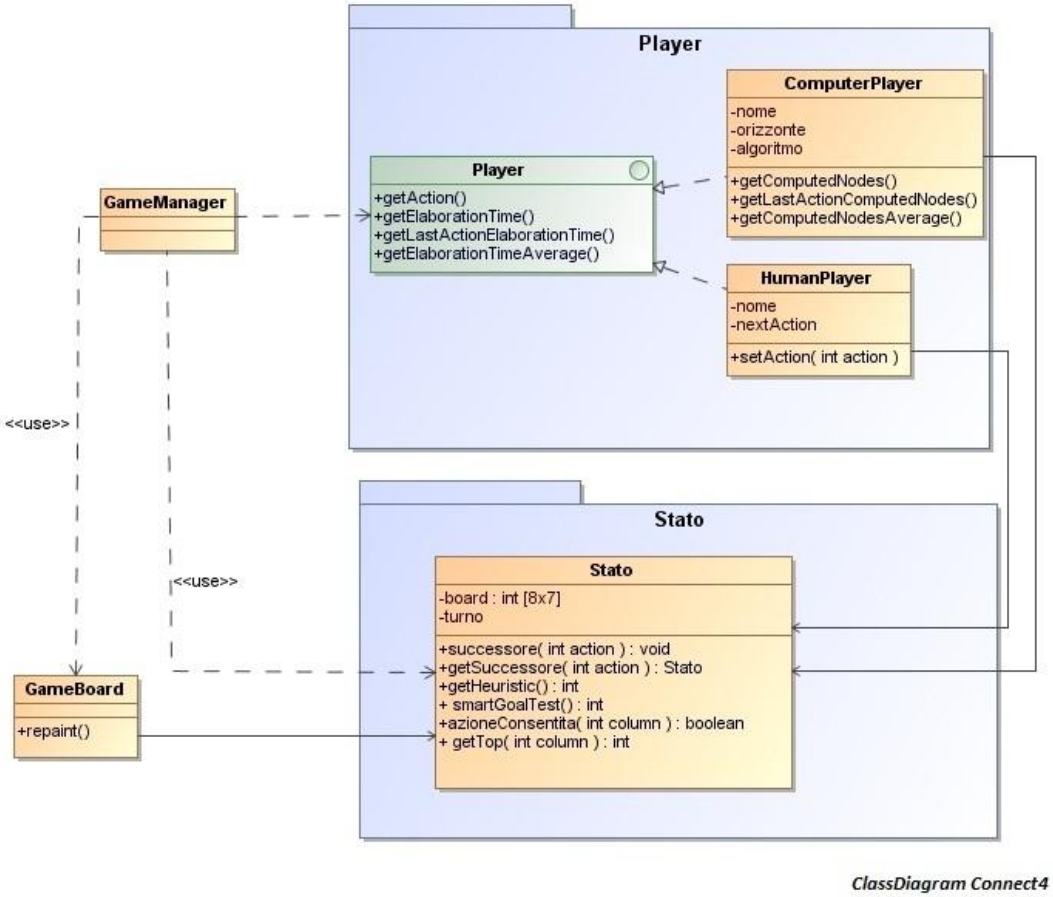
Si è scelta tale strategia per due motivi:

1. Il primo motivo è che già nell' "*euristica presente*" viene fatta una valutazione pesata rispetto ai vuoti ed alle tipologie di vuoti, giustificata dal fatto che tale euristica dovesse essere una precisa rappresentazione dello stato e, di conseguenza, la differenza tra una configurazione remota ed una prossima doveva essere immediatamente percepibile. Rieseguire tale controllo anche in questa euristica ci è sembrato quindi ridondante.
2. Il secondo motivo è di natura semantica. L'euristica futura deve dare informazioni di quanto uno stato è buono nel senso di quante caselle sono immediatamente coinvolgibili in "*FORZA 4*" ed è quindi una rappresentazione del futuro immediato. Il futuro più lontano viene dato dall'albero di minimax/alphabeta-pruning che, simulando le mosse successive, può portare alla scelta di riempire un vuoto che, sebbene non porti alcun punteggio, apre per le mosse successive nuove configurazioni favorevoli.

Il valore totale dell'euristica è dato dalla differenza del numero totali di "*FORZA 4*" contati per i due giocatori. Nell'esempio precedente, ipotizzando che lo stato sia la foglia dell'albero costruito dal giocatore rosso, l'euristica totale ha valore $euristicaFutura = 12 - 14 = -2$.

4. PROGETTAZIONE ED IMPLEMENTAZIONE

4.1. DIAGRAMMA DELLE CLASSI



4.2. IMPLEMENTAZIONE DELLE CLASSI

Il progetto è stato implementato come applicazione stand-alone utilizzando il linguaggio di programmazione object-oriented JAVA sviluppato da SUN.

Le classi implementate sono state suddivise in package in base alle funzionalità (vd. Fig. “Class Diagram Connect4)

4.2.1. CLASSE STATO.STATO

La classe permette di rappresentare un determinato stato del gioco e fornisce metodi per la sua gestione e per la generazione degli stati successori.

- **Attributi**
  - **board[][]**: Matrice di interi di dimensioni 9x7 che memorizza il contenuto della scacchiera di gioco. Sebbene la scaccheria sia di dimensioni 8x7, la matrice è stata dotata di una riga supplementare che consente di registrare, per ogni colonna, l'indice di riga dell'ultimo elemento inserito (-1 se la colonna è vuota); questa scelta permette, nel momendo delll'inserimento di una nuova pedina, di evitare di analizzare l'intera colonna alla ricerca della prima cella libera, redendo costante la complessità del metodo per la generazione di uno stato successore.
  - **turno**: intero con valore ∈ {1,-1} che indica il turno di gioco; il giocatore indicato da tale attributo sarà il prossimo a dover inserire la pedina. L'aggiornamento di tale attributo è affidato ai metodi che generano gli stati successori.

- **Metodi**
  - **successore(int action):** il metodo *successore* trasforma l'attuale istanza di Stato un uno stato successore; prendendo in input un intero rappresentante l'azione da intraprendere (la colonna in cui inserire la pedina), controlla se l'azione è possibile e se lo stato attuale non rappresenta uno stato di goal ed, in caso affermativo, inserisce la nuova pedina e provvede a fare lo switch del turno gioco. Tale metodo ha complessità costante e fa uso dei seguenti metodi supplementari:
  - **azioneConsentita(int)**: controlla, con complessità costante grazie alla riga supplementare introdotta precedentemente, se la colonna specificata è piena
  - **getSuccessore(int action):** tale metodo ha funzionalità e caratteristiche analoghe al metodo *successore* ma, a differenza di quast'ultimo, non modifica l'istanza attuale ma ne genera un'altra contenete lo stato successore. La complessità è naturalmente maggiore rispetto al metodo *successore* in quanto prevede la copia del contenuto dell'attributo *board*. Il metodo *getSuccessore* viene utilizzato dagli algoritmi quali *Alpha-Beta Pruning* e *Minimax* per generare i nodi dell'albero di ricerca.
  - **smartGoalTest():** effettua un goal test ottimizzato analizzando solo le caselle adiacenti all'ultima pedina inserita bassandosi sul concetto che, se lo stato attuale è uno stato di goal e lo stato precedente non lo era, la serie vincente è stata formata solo in conseguenza dell'ultima pedina inserita. Il metodo controlla progressivamente le 4 direzioni (verticale, orizzontale, diagonale principale e diagonale secondaria) alla ricerca di almeno quattro pedine, di segno uguale

all'ultima inserita, adiacentri tra loro; per ogni direzione vengono esaminate al massimo 6 caselle e, non appena viene rilevata una serie vincente, il metodo termina restituendo il relativo ed eventuale valore di vittoria.
    - **azioneConsentita(int action):** controlla se l'azione passata come parametro è applicabile allo stato attuale verificando se la relativa colonna ha ancora celle vuote; il metodo ha complessità costante dal momento che analizza l'indice presente nell'ultima riga dell'attriuto board e ne confronta il contenuto con le dimensioni di riga della scacchiera di gioco.
    - **getTop(int):** restituisce l'indice di riga dell'ultimo elemento inserito nella colonna specificata come parametro.
    - **getHeuristic():** tale metodo implementa la funzione fornendo una valutazione dello stato auttuale. Il metodo verrà analizzato nel dettaglio successivamente.

4.2.2. INTERFACCIA PLAYER.PLAYER

L'interfaccia rappresenta descrive i metodi che un generico giocatore, sia esso umano o artificiale, deve implementare per essere utilizzato.

Anche se alcuni metodi sono specifici del giocatore umano o di quello artificiale, sono stati comunque inseriti nell'interfaccia Player in modo da unificare la modalità di interazione con il Frame.

- **Metodi per la gestione delle azioni**
    - **getAction():** restituisce l'azione scelta dal giocatore per lo stato attuale; tale azione sarà usata per generare lo stato successore.
    - **getLastAction():** restituisce l'ultima azione intrapresa dal giocatore.
    - **public void setAction(int action):** specifica per il giocatore umano, le relative specifiche sono fornite nella descrizione della classe *HumanPlayer.*
- **Metodi per la gestione dei dati statistici**
    - **getActionNumber():** Restituisce il numero totale di azioni generate (il numero di volte che il metodo *getAction()* è stato chiamato).
    - **getElaborationTime():** fornisce il tempo totale in millisecondi di elaborazione finora occorso per il calcolo delle azioni da intraprendere.
    - **getLastActionElaborationTime():** restituisce il tempo in millisecond occorso per il calcolo dell'ultima azione intrapresa.
    - **getElaborationTimeAverage():** fornisce il tempo medio di elaborazione in millisecondi calcolato come *TempoTotale/NumeroAzioni.*

4.2.3. CLASSE PLAYER.COMPUTERPLAYER

Implementazione dell'interfaccia Player, questa classe rappresenta la generica entità di IA che giocherà contro un suo simile o contro un umano. E' all'interno di questa classe che sono implementati gli algoritmi *Minimax* ed *Alpha-Beta Pruning.*

- **Attributi identificazione giocatore e parametri di ricerca**
    - **nome**: intero con valore ∈ {1,-1} che rappresenta l'identità del giocatore (rosso o giallo); tale attributo viene impostato dal costruttore e non potrà più essere

modificato durante il cliclo di vita dell'oggetto, dovrà inoltre essere univoco per tutto il gioco.
    - **algoritmo**: stringa con valore ∈ {'alphabeta','minimax'} che indica l'algoritmo da utilizzare per la creazione dell'albero di ricerca; l'attributo viene settato al momento della costruzione dell'istanza e verrà utilizzato dal metodo getAction per decidere quale routine utilizzare per la ricerca.
    - **orizzonte**: intero che rappresenta la profondità massima degli algoritmi di ricerca; l'attributo viene settato al momento della costruzione ma può essere variato durante il ciclo di vita dell'istanza.
- **Attributi per la gestione delle statistiche**
    - **int lastAction:** inizializzata a -1 al momento dell'istanziazione, la variabile viene aggiornata ad ogni chiamata al metodo *getAction()* al fine di essere sincronizzata con l'ultima azione intrepresa.
    - **int actionNumber:** la variabile rappresenta un contatore sul numero di azioni richieste/eseguite.
    - **int computedNodes:** contatore del numero totale di nodi generati dagli algoritmi di ricerca.
    - **int lastActionComputedNodes:** numero di nodi generati dall'ultima esecuzione dell'algoritmo di ricerca.
    - **float elaborationTime:** timer contenente il tempo totale di elaborazione in millisecondi occorso per il calcolo delle azioni
    - **float lastActionElaborationTime:** variabile contenente il tempo di calcolo dell'ultima esecuzione dell'algoritmo di ricerca.

- **Metodi**

    - **ComputerPlayer(int,int,String,Stato)**: unico costruttore della classe; prende come parametri di input il nome del giocatore, l'orizzonte, il nome dell'algoritmo da usare e il puntatore all'istanza di Stato cui fare riferimento (l'ambiente in cui l'agente agisce)
    - **getAction()**: il metodo analizza lo stato attuale e fornisce in output l'azione che, secondo la funzione euristica, porterà a massimizzare l'utilità dell'agente; a seconda del valore dell'attributo *algoritmo*, viene utilizzata, per il calcolo dell'azione, la routine che implementa l'algoritmo *Minimax* o quella che implementa *Alpha-Beta Pruning* i cui dettagli saranno discussi successivamente. La chiamata a getAction() è bloccante ed il tempo che intercorre tra la sua invocazione e la restituzione dell'azione da intraprendere da parte di uno dei due algoritmi, è cronometrato al fine di concorrere alla statistiche finali richieste dal progetto.

4.2.4. CLASSE PLAYER.HUMANPLAYER

La classe rappresenta il giocatore umano e, più precisamente, fornisce le funzionalità che permettono l'interazione con il sistema.

- **Attributi**
  - **nome**
  - **int nextAction:** l'attributo ha scopo puramente funzionale e non è accessibile all'esterno direttamente e tantomeno mediante di opportuni metodi; il suo scopo è quello di memorizzare la richiesta dell'utente circa l'azione da intraprendere e di conservarla fino all'invocazione del metodo *getAction()*. Dal momento che il suo scopo è identificare la prossima azione, può assumere come valore uno qualsiasi degli elementi dell'ActionSet; è previsto però un valore flag (-1) utilizzato per indicare il caso in cui l'utente non abbia ancora specificato l'azione desiderata.
  - **int actionNumber:** il suo scopo è del tutto simile all'omonimo attributo definito per la classe *Player.ComputerPlayer*.
  - **int lastAction:** il suo scopo è del tutto simile all'omonimo attributo definito per la classe *Player.ComputerPlayer*.
  - **float elaborationTime:** il suo scopo è del tutto simile all'omonimo attributo definito per la classe *Player.ComputerPlayer*.
  - **float lastActionElaborationTime:** il suo valore rappresenterà l'intervallo di tempo, espresso in millisecondi, che occorre tra l'invocazione del metodo *getAction()* e la scelta dell'azione da parte dell'utente.
- **Metodi**
  - **HumanPlayer():** unico costruttore della classe
  - **getAction():** il metodo è una funzione che fornisce in output l'azione richiesta dall'utente; al momento della sua chiamata, il metodo controlla continuamente il contenuto dell'attributo *nextAction*, cronometrando il tempo durante il quale è pari a "-1" (valore flag che indica che l'utente non ha ancora scelto l'azione). Non appena il valore di *nextAction* varia, il cronometro viene stoppato e vengono opportunamente aggiornati gli attributi *lastActionElaborationTime* e d *elaborationTime*; in fine il valore contenuto in *nextAction* viene restituito come output e l'attributo riassume il valore simbolico "-1".
  - **setAction(int action):** il metodo permette all'utente di specificare quale azione intraprendere aggiornando il valore dell'attributo *nextAction*.

4.2.5. CLASSE GAMEBOARD

GameBoard implementa un'interfaccia grafica che permette di fornire all'utente la configurazione dello stato corrente. Estendendo la classe *JTable*, l'interfaccia fornita consiste in una tabella di dimensioni 8x7 (coerentemente con la scacchiera di gioco) a cui è possibile associare un'istanza della classe *Stato* permettendo un aggiornamento della GUI del tutto automatico. Al fine di migliorare l'esperienza dell'utente, le caselle vengono opportunamente colorate.

4.2.6. CLASSE GAMEMANAGER

GameManager è l'entità che si occupa di gestire i turni all'interno del gioco.

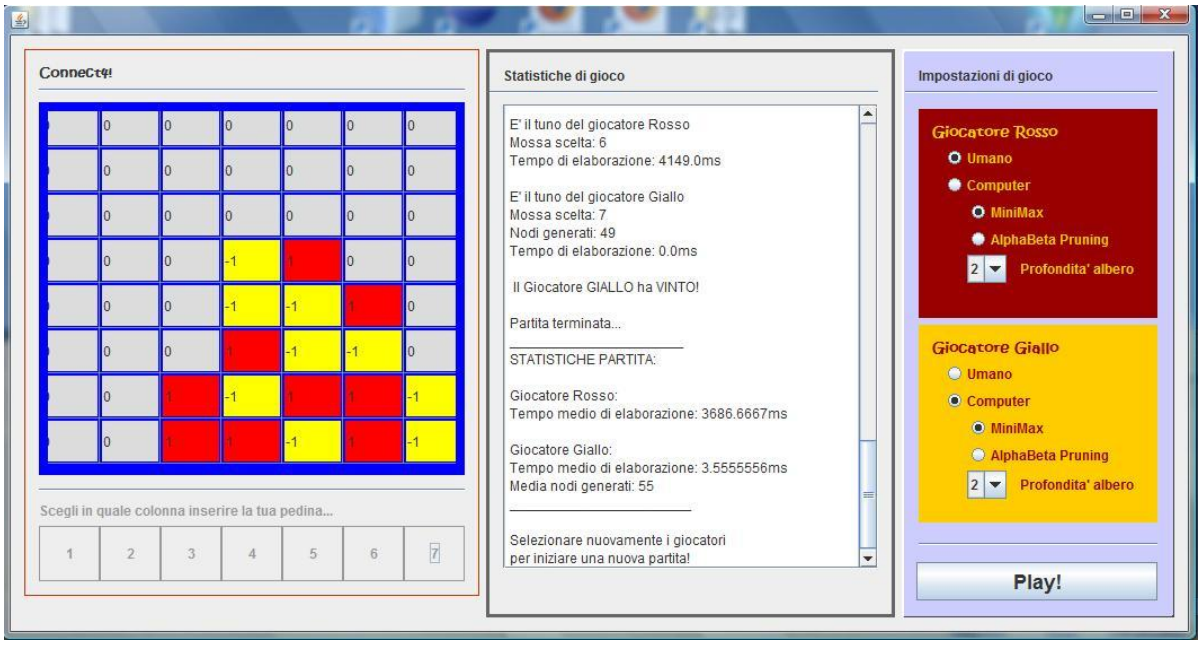
L'idea iniziale prevedeva di sviluppare *HumanPlayer* e *ComputerPlayer* come implementazioni dell'interfaccia JAVA *Runnable*, dando quindi la possibilità di eseguire le istanze dei giocatori come thread indipendenti con capacità di modifica diretta sullo stato. In fase di testing è stato però verificato che l'esecuzione dei thread concorreva con gli algoritmi di ricerca delle soluzioni, facendo aumentare il loro running time e rendendo gli agenti poco reattivi.

In ultima analisi si è scelto di gestire i turni di gioco mediante un'entità esterna: l'istanza della classe *GameManager*, fino a quando il gioco non termina per vittoria o per assenza di mosse, controlla il turno di gioco ed interroga il relativo giocatore riguardo l'azione da intraprendere; in fine utilizza l'azione suggerita per generare lo stato successore. Altro compito di *GameManageri* è aggiornare continuamente la lavagna di gioco

4.2.7. INTERFACCIA GRAFICA

La completa interfaccia grafica è stata realizzata mediante la "creazione di una JFrame".

Come richiesto dal committente, l'interfaccia fornisce un pannello per impostare il tipo di giocatore e le sue caratteristiche (algoritmo da utilizzare e suo orizzonte); è presente anche una casella di testo scorrevole che tiene continuamente al corrente sulle azioni scelte e sui dati statistici.



4.3. IMPLEMENTAZIONE DEI METODI STRATEGICI

4.3.1. EURISTICA PRESENTE

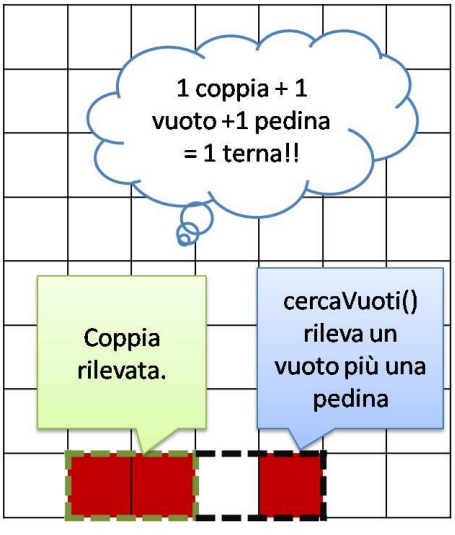
Come accennato precedentemente, l'euristica presente cerca di dare una valutazione dello stato in esame in termini di distanza dal goal.

L'euristica presente viene calcolata nella funzione privata **int euristicaPresente()** della classe Stato che, per ciascun giocatore, analizza le n-ple già posizionate ed assegna loro un punteggio in base alla loro conformazione. Il valore finale dell'euristica sarà la differenza tra la somma dei punti accumulati dal giocatore rosso e quelli del giocatore giallo. L'euristica è anche in grado di rilevare stati di goal e in tal caso restituirà valori $+\infty,-\infty$ (implementati con i valori +500 e -500).

La funzione utilizza a sua volta le seguenti funzioni private della classe Stato:

- **int cercaAdiacenze(Direzione direzione, int riga, int colonna):** conta le pedine adiacenti a quella specificata in input come segue. La riga e la colonna in input sono le coordinate di una pedina già conquistata da uno dei due giocatori. Il seme di tale pedina viene preso come riferimento. La direzione lungo la quale vengono contate le adiacenze è anch'essa specificata in input ed è un valore $\in \{N, S, E, W, NE, NW, SE, SW\}$. Se lungo la direzione indicata vengono rilevate pedine di seme concorde a quello di riferimento, viene aggiornato il contatore altrimenti il controllo termina.

- **double cercaVuoti(int giocatore, Direzione direzione, int riga, int colonna,int adiacenze,int lowerBound):** Conoscendo il numero di pedine già allineate e la direzione lungo cui controllare, analizza le caselle necessarie per poter individuare correttamente la natura della n-pla in esame e restituire il corrispettivo punteggio (specificato nella TABELLA1). In altri termini, vengono cercati i vuoti e individuata la loro tipologia in modo da poter riconoscere una npla remota o una coppia prossima. Tuttavia, per una corretta rilevazione delle configurazioni, oltre ai vuoti, è necessario rilevare eventuali pedine dello stesso seme presenti dopo le caselle vuote. Un tipico esempio è mostrato nella seguente figura.



Esempio di terna non consecutiva

Analizziamo nel dettaglio i parametri della funzione:

  - gli interi riga e colonna rappresentano le coordinate della casella a partire dalla quale contare i vuoti.

- La direzione è un valore $\in \{N, S, E, W, NE, NW, SE, SW\}$ e rappresenta appunto la direzione lungo la quale eseguire il controllo
- Giocatore rappresenta il seme delle caselle precedentemente rilevate necessario nel caso in cui di debba rilevare una pedina seguente il vuoto
- Adiacenze indica il numero delle pedine allineate già rilevate ed è utilizzato per calcolare il numero delle caselle che restano da analizzare con **cercaVuoti** utilizzando la formula : $4 - (\text{adiacenze} + 1)$
- L'intero lowerBound rappresenta un indice di controllo che permette di non contare più volte la stessa configurazione in modo da non sovrastimare lo stato

A partire dalla pedina specificata dalle coordinate, si segue la direzione data in input procedendo nel controllo fino a quando non risulta verificata una delle seguenti condizioni:

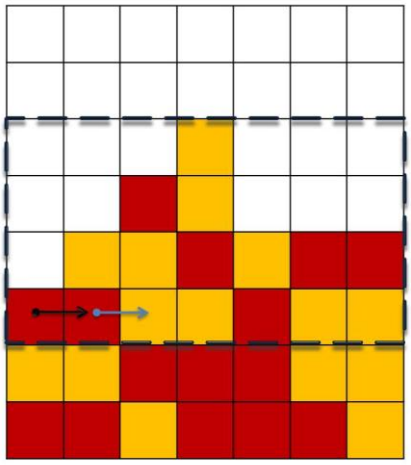
- termina la matrice di gioco,
- vengono rilevate pedine avversarie,
- viene raggiunto il lowerBound,
- viene raggiunto il numero di caselle da controllare (calcolato come specificato precedentemente).

Vengono quindi contati i vuoti immediatamente occupabili, i vuoti non immediatamente occupabili ed eventuali pedine del giocatore e memorizzati in 3 contatori.

In base ai valori di ciascun contatore la funzione restituirà il fattore moltiplicativo associato alla configurazione come indicato nella TABELLA 1.

La funzione **int euristicaPresente()**, quindi, utilizza le funzioni appena descritte per effettuare in maniera sequenziale 3 tipi di controlli:

1. *verticale*: per ogni colonna, partendo dall'ultima pedina inserita, verifica la presenza di eventuali pedine compatibili sottostanti e la presenza di un numero sufficiente di vuoti necessari per chiudere la quadrupla.
2. *Orizzontale*: Da un certo punto del gioco in poi, non tutta la matrice è interessante. Visto che quello che si vuole valutare sono le configurazioni aperte che possono evolvere in "*FORZA 4*", le righe di indice più basso della matrice completamente riempite contengono informazioni relative al passato e che non influenzano in alcun modo l'evolversi dello stato attuale. Il controllo orizzontale viene quindi eseguito solo sulle righe ancora attive costituite dalle righe comprese tra l'ultima pedina di colonna ad altezza più bassa e quella ad altezza più alta così come si può vedere nell'immagine.

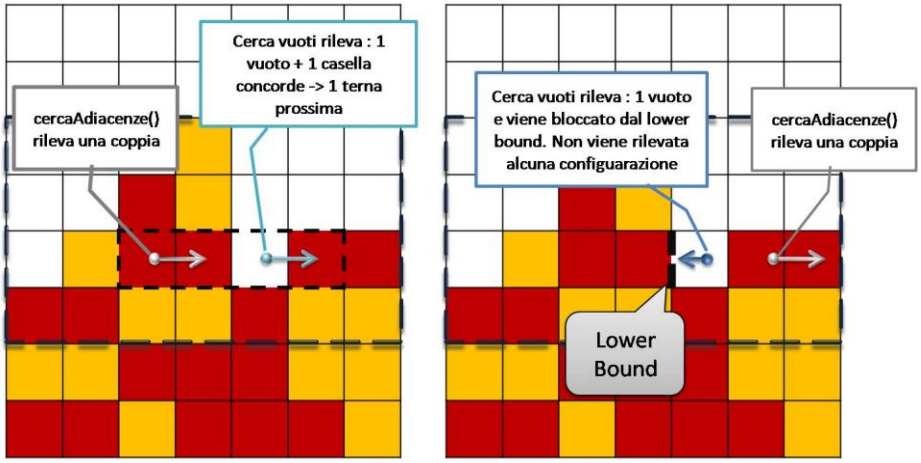


Esempio di controllo orizzontale

Per ogni riga attiva, procedendo da sinistra verso destra vengono quindi eseguite le seguenti operazioni:

1. Viene rilevato il seme della pedina (i vuoti non vengono considerati)
2. Vengono rilevate eventuali pedine concordi adiacenti con la funzione *cercaAdiacenze()* descritta precedentemente (se la pedina è sola, non è interessante).
3. Partendo dai due estremi della n-pla rilevata, viene richiamata la funzione *cercaVuoti()* per le due direzioni Est ed Ovest.
4. Viene aggiornato il valore del *lowerBound* per l'iterata successiva con l'indice dell'ultimo elemento della n-pla trovata da *cercaAdiacenze()*.
5. A senconda dei valori restituiti da *cercaVuoti()* viene data una valutazione della configurazione, le si assegna un punteggio e lo si somma ai punti totalizzati dal giocatore proprietario della n-pla.

Un esempio di quanto descritto è visibile nell'immagine seguente.



Dal controllo della riga viene rilevata una terna prossima

Il controllo di riga prosegue ma non viene rilevata altra configurazione

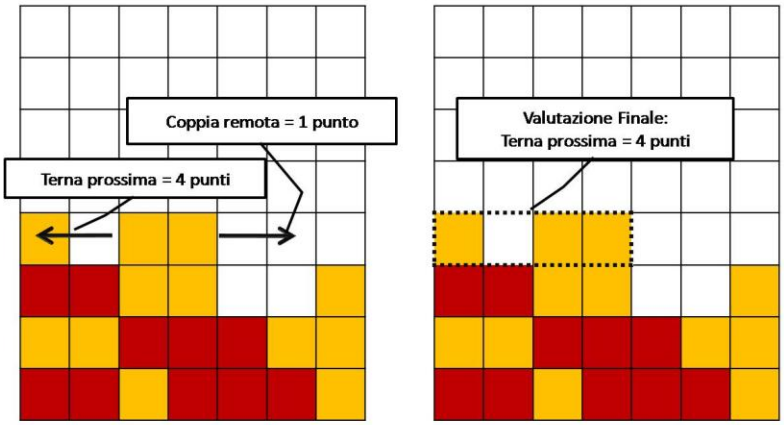
L'immagine rappresenta il processo di analisi della seconda riga attiva della matrice.

Come si può vedere nella scacchiera alla sinistra, partendo dalla colonna 3, la funzione *cercaAdiacenze()* rileva una coppia. Viene quindi richiamata *cercaVuoti(),* che, nella direzione ovest non rileva nulla per la presenza di una pedina gialla, mentre nella direzione est trova un vuoto e una pedina rossa per una valutazione complessiva di terna prossima. A questo punto il *lowerBound* viene aggiornato con l'indice della colonna 4 (posizione dell'ultima pedina della coppia), così come si può vedere nella scacchiera sulla destra, e *cercaAdiacenze()* ricomincia il suo controllo dalla colonna 6 (i vuoti non vengono considerati) rilevando un'altra coppia. Come prima viene richiamato cercaVuoti(), che, nella direzione est viene bloccato dalla fine della matrice di gioco, mentre nella direzione ovest viene bloccato dal lowerBound dopo aver trovato un vuoto. La valutazione termina quindi con un niente di fatto.

Senza il lowerBound si sarebbero rilevate erroneamente 2 terne prossime laddove, di fatto, la configurazione utile sarebbe stata una sola.

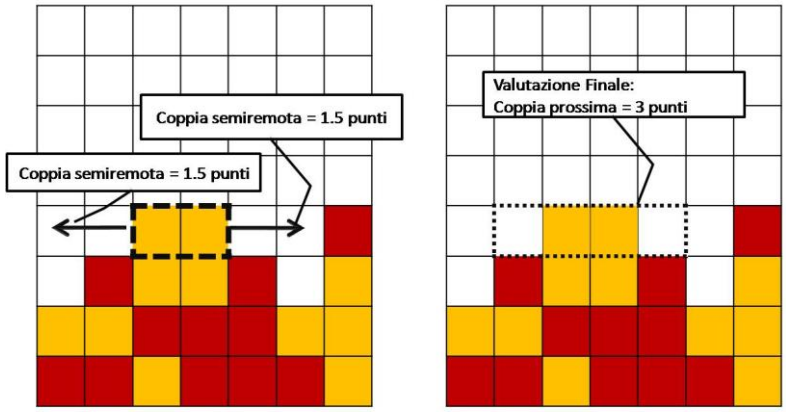
Per quanto riguarda il passo 5, sempre per evitare di sovrastimare lo stato, si deve fare una corretta elaborazione dei valori restituiti da *cercaVuoti()*.Di seguito sono riportati degli esempi:

- Caso base. La valutazione finale della n-pla è il valore massimo tra il risultato della scansione di *cercaVuoti()* in direzione Est e il risultato della scansione in direzione Ovest.



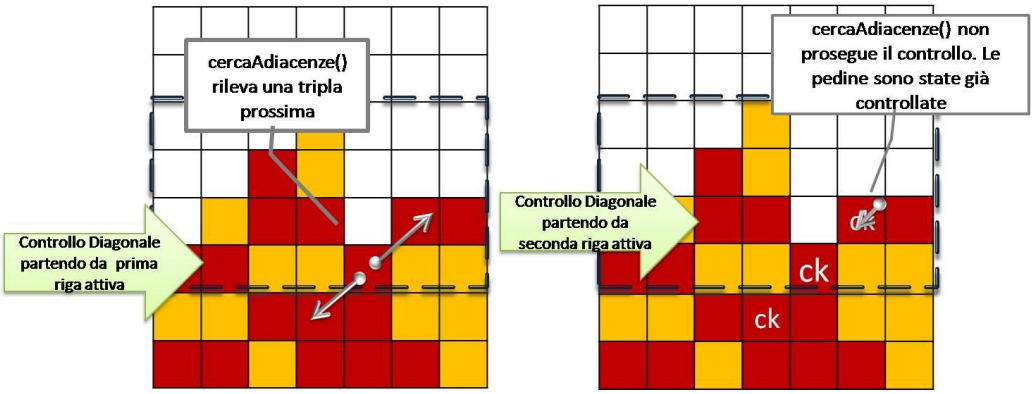
Caso Base. La valutazione finale è quella di valore massimo

- Caso particolare Coppia semiremota. In questo caso la valutazione finale della n-pla è dato dalla somma delle due scansioni.



Caso delle coppie semiremote. La valutazione è la somma

3. *Diagonale*: Per quanto riguarda il controllo diagonale, valgono le stesse cose dette per il controllo orizzontale con l'unica differenza che oltre al lowerBound è necessaria un'altra struttura di controllo per evitare di conteggiare più volte la stessa n-pla.



Dal controllo diagonale della prima riga attiva partendo dalla colonna 5, viene rilevata una tripla prossima

Il controllo diagonale della seconda riga partendo dalla colonna 6 non viene eseguito perché la pedina fa parte di una tripla già contata.

Il controllo diagonale consiste in due fasi sequenziali nelle quali si analizzano rispettivamente la diagonale NE-SW e quella NW-SE; a tal proposito è risultato naturale utilizzare, come struttura di controllo su citata, una matrice di dimensioni pari a quelle della matrice attiva (area tratteggiata) che tiene traccia delle celle che già fanno parte di una npla precedentemente considerata.

L'andamento dell'algoritmo di controllo segue lo schema colonna per riga: muovendosi lungo la colonna, per ogni cella vengono considerate le diagonali da essa sorgenti; terminate le celle, si ripete l'operazione partendo dalla prima

4.3.2. EURISTICA FUTURA

Come descritto precedentemente, il valore dell'euristica futura è la differenza tra il numero totale di configurazioni vincenti ancora disponibili per il giocatore rosso, e quelle per il giocatore giallo.

L'euristica futura viene calcolata nella funzione privata **int euristicaFutura()** della classe Stato che per ogni colonna, analizza il primo vuoto disponibile dal punto di vista di entrambi i giocatori e in tutte le 4 direzioni.

Per fare ciò richiama le seguenti funzioni private sempre della classe Stato:

- **int verticalHeuristic(int giocatore, int colonna):** Conta il numero di "*FORZA 4*" disponibili *VERTICALMENTE* per il giocatore dato in input in corrispondenza della colonna specificata.

  Per la direzione verticale, il numero di "*FORZA 4*" realizzabili è al più uno e solo qualora vi siano caselle vuote sufficienti a chiudere la quadrupla. Questo vuol dire che oltre a contare le caselle vuote ancora presenti nella colonna, si tiene conto anche delle pedine già posizionate.

- **int horizontalHeuristic(int giocatore, int colonna):** Conta il numero di "*FORZA 4*" disponibili *ORIZZONTALMENTE* per il giocatore dato in input in corrispondenza della colonna specificata.

  Il controllo orizzontale si svolge in 2 fasi. Nella prima vengono analizzate le 3 caselle a destra, nella seconda le 3 caselle a sinistra. Per entrambe le fasi le operazioni eseguite sono le seguenti:

  - Fino a che non si incontrano pedine avversarie, la fine della matrice di gioco, o caselle vuote ma non immediatamente occupabili, ci si sposta nelle 3 pedine a destra(sinistra) contando il numero di caselle già occupate dal giocatore specificato e il numero di vuoti immediatamente occupabili.

  - Se il numero di caselle contate è strettamente maggiore di 3 viene restituito il numero di possibili combinazioni di "*FORZA 4*" realizzabili con le caselle a disposizione pari a $(Count-4)+1$.

  - Se il numero di caselle contante è inferiore a 4, allora la colonna specificata non è coinvolta in nessun "*FORZA 4*" del giocatore e viene restituito 0.

- **int firstDiagonalHeuristic(int giocatore, int colonna):** Conta il numero di "*FORZA 4*" disponibili *PER LA DIAGONALE NORD-OVEST/ SUD-EST*(diagonale principale) per il giocatore dato in input in corrispondenza della colonna specificata.

  Il controllo della prima diagonale avviene in maniera del tutto analoga al controllo orizzontale. E' sempre suddiviso in due fasi, la prima che controlla le 3 pedine in basso a destra, la seconda che controlla le 3 pedine in alto a sinistra.

  Le operazioni eseguite ed i valori ritornati sono del tutto analoghi a quelli descritti per il controllo orizzontale.

- **int secondDiagonalHeuristic(int giocatore, int colonna):** Conta il numero di "*FORZA 4*" disponibili *PER LA DIAGONALE NORD-EST/ SUD-OVEST*(diagonale secondaria) per il giocatore dato in input in corrispondenza della colonna specificata.

Il controllo della seconda diagonale avviene in maniera del tutto analoga ai due controlli già descritti. E' sempre suddiviso in due fasi, la prima che controlla le 3 pedine in basso a sinistra, la seconda che controlla le 3 pedine in alto a destra.

Le operazioni eseguite ed i valori ritornati sono del tutto analoghi a quelli già descritti per il controllo orizzontale e della prima diagonale.

Nella funzione **int euristicaFutura()** per ogni colonna, vengono richiamate le funzioni descritte per entrambi i giocatori e sommati i valori restituiti in due contatori, uno per ciascun giocatore.

Se alla fine entrambi i contatori avranno valore nullo vuol dire che da quello stato entrambi i giocatori non sono in grado di realizzare un "*FORZA 4*" e che quello stato rappresenta una condizione di stallo da cui è inutile proseguire il gioco.

In caso contrario sarà restituita la differenza tra il valore del contatore associato al giocatore rosso e quello associato al giocatore giallo.

4.3.3. UNIONE DELLE EURISTICHE

L'unione delle informazioni fornite dalle due euristiche è eseguita nella funzione **int getHeuristic()** della classe Stato che non fa altro che applicare la formula già analizzata in precedenza.

Le due euristiche, però, forniscono singolarmente anche delle informazioni rilevanti. L'euristica presente individua anche uno stato di goal, mentre l'euristica futura rileva se uno stato è di parità.

Di conseguenza, se entrambe le euristiche non hanno un valore significativo, esse vengono combinate insieme come già detto, in caso contrario viene restituito il valore significativo.

4.3.4. MINIMAX

L'algoritmo di *MINIMAX* è eseguito dalla funzione privata **int getMossaMiniMax()** della classe *ComputerPlayer* che fa utilizzo a sua volta della funzione privata **int miniMax(Stato nodo, int h)** sempre della classe *ComputerPlayer.*

La funzione **int miniMax(Stato nodo, int h)** è una funzione ricorsiva utilizzata per costruire l'albero e che rappresenta il fulcro dell'intero procedimento. In particolare partendo dal valore dell'intero h, che indica l'attuale livello dell'albero, e che viene incrementato ad ogni chiamata ricorsiva, vengono eseguite le seguenti operazioni:

- Se h==orizzonte ossia se si è raggiunto il livello specificato dall'utente in input, vuol dire che il nodo dato in input è una foglia dell'albero, pertanto si dovrà semplicemente calcolare il valore dell'euristica e restituirla in output.

  A tale proposito è necessaria un'osservazione. Dal momento che abbiamo considerato l'euristica una qualità intrinseca dello stato che prescindesse dal giocatore che la volesse calcolare, per come è stata costruita essa restituisce valori positivi se è "avvantaggiato" il giocatore rosso, e valori negativi se è "avvantaggiato" il giocatore giallo. Di conseguenza, se è il giocatore giallo a costruire l'albero, dal momento che è lui a rappresentare il MAX in questo caso, bisognerà stare attenti a invertire i segni del valore dell'euristica in modo da

garantire il funzionamento dell'algoritmo altrimenti falsato da valori di euristica non coerenti.

- Se h<orizzonte, allora :
    - Se il nodo in input rappresenta uno stato di goal, nonostante non sia una foglia non viene espanso e viene restituito in output valore associato alla vittoria (+∞,-∞).
    - Se il nodo in input non rappresenta uno stato di goal e non è una foglia, allora rappresenta la radice di un sottoalbero, radice che dovrà scegliere il valore più alto tra quelli ritornati dai suoi figli se è un nodo MAX, oppure il più basso se è un nodo MIN. Per ogni azione ancora disponibile, quindi, viene generato un nodo e quindi richiamata ricorsivamente la funzione a cui viene passato il nodo appena generato e un valore incrementato di h.

Dal momento che, oltre agli stati in cui uno dei due giocatori vince, uno stato finale può essere anche costituito da una condizione di stallo nella quale entrambi i giocatori non hanno più mosse a disposizione, è previsto un valore flag anche per questo caso.

A livello implementativo, il valore flag è stato codificato con la cifra -200; per ovvi motivi di coerenza con i controlli eseguiti da MINIMAX ed AlphaBeta, tale valore oscillerà nell'intervallo {200,-200} in modo da essere correttamente interpretato sia dal giocatore MIN che da MAX.

Tale condizione viene, naturalmente, considerata comunque favorevole rispetto ad una sconfitta.

Esegue la prima iterazione, generando i figli della radice e fungendo da frontend con la ricorsione della funzione miniMax(Stato nodo,int h), restituendo in output l'azione che porterà ad uno stato favorevole (invece che il valore euristico come miniMax).

Durante le operazioni di testing abbiamo creato artificialmente uno stato che portasse alla vittoria sicura di uno dei due giocatori e che consentisse, tramite l'applicazione di una specifica azione, di ottenere immediatamente la vittoria; abbiamo lasciato che il sistema continuasse a giocare da solo. Quello che succedeva era che il giocatore che aveva sicuramente vinto, non eseguiva la mossa di chiusura ma eseguiva l'ultima azione analizzata. Il motivo di questo comportamento è semplice : data la presenza di molteplici strategie a vittoria sicura, dall'albero *MINIMAX* erano risaliti tutti valori pari a +∞, quindi, per il giocatore, una mossa valeva l'altra non rilevando il fatto che c'erano mosse che lo avrebbero portato alla vittoria più velocemente e in particolare una che consentisse il raggiungimento immediato di uno stato di goal.

Sebbene questo comportamento non rappresenti un problema in quanto non inficia sulla razionalità dell'agente (l'agente gioca comunque massimizzando la sua utilità), ci è sembrato doveroso ottimizzare il suo comportamento fornendo un controllo nella funzione **int getMossaMiniMax()**, che, per ogni azione disponibile, prima di eseguire la chiamata alla funzione **int miniMax(Stato nodo, int h)** con il nodo appena generato, controlla se il nodo rappresenta uno stato di goal e, in caso affermativo, termina restituendo l'indice dell'azione che lo ha generato. In questo modo, se l'azione che porta al goal è a profondità 1 nell'albero di ricerca, essa viene immediatamente individuata.

4.3.5. ALPHABETA PRUNING

Per quanto riguarda l'*ALPHABETA PRUNING,* esso viene eseguito nella funzione **int getMossaAlfaBetaPruning()** della classe *ComputerPlayer* che esegue le stesse operazioni della funzione **getMossaMiniMax()** precedentemente analizzata, di conseguenza, valgono per questa funzione le considerazione espresse precedentemente. Unica differenza sono le funzioni ricorsive che in questo caso sono due:

- **int alfaBetaMaxValue(Stato nodo, int alfa, int beta, int h)**
- **int alfaBetaMinValue(Stato nodo, int alfa, int beta, int h)**

Entrambe le funzioni svolgono le stesse operazioni della funzione *int minimax()* vista precedentemete con la differenza che mentre la funzione *int minimax()* gestisce entrambi i casi dei nodi *MAX* e *MIN,* in questo caso sono distribuiti tra le due funzioni.

Altra differenza è la presenza nelle due funzioni degli interi alfa e beta necessari per eseguire la potatura.

5. TESTING E STATISTICHE

5.1. RISULTATI DEI TESTING

Uno degli scopi del progetto è evidenziare i diversi comportamenti dei due algoritmi utilizzati in merito ai nodi generati ed al tempo di elaborazione; ciò che ci aspettiamo è una performance migliore dell'algoritmo AlphaBeta Pruning, come conseguenza delle scelte di ottimizzazione esposte precedentemente che consentono una enumerazione implicita dei possibili stati.

A tale scopo sono stati effettuati dei testing facendo eseguire entrambi gli algoritmi con diverse profondità dell'albero.

I risultati sono riportati nella seguente tabella (i tempi si intendono in millisecondi).

	Minimax		AlphaBeta Pruning		Confronto algoritmi	
Orizzonte	**Media Nodi**	**Tempo Medio**	**Media Nodi**	**Tempo Medio**	**Differenza Nodi**	**Differenza Tempo**
2	39	2,09	39	2,09	0	0
3	342	6,71	231	4,49	111	2,22
4	2559	50,14	1216	26,78	1343	23,36
5	16747	261,33	5433	92,33	11314	169
6	85376	1550	11730	198,25	73646	1351,75
7	570324	9670	44464	711,35	525860	8958,65
8	4609498	80470	200015	3172	4409483	77298
9	//	//	615360	7837	//	//

Per quanto riguarda il minimax a profondità 9, non sono state riportate statistiche poiché i tempi di calcolo per ciascuna mossa sono superiori a 500 secondi pertanto, non rappresentando un tempo di feedback accettabile, ne è sconsigliato l'utilizzo.

Come evince dalla tabella, i risultati confermano le ipotesi esposte in precedenza.

5.2. CONCLUSIONI E POSSIBILI SVILUPPI

Dal momento che l'obiettivo principale del progetto era fornire una formalizzazione corretta e consistente del problema e quindi fornire una definizione di stato, di funzione di successione, e di funzione euristica, non si sono prese in considerazione ulteriori ottimizzazioni; pertanto queste saranno presentate come possibili sviluppi futuri.

Proprio perchè non era un'esigenza primaria del progetto, ad ogni richiesta di azione, gli algoritmi di ricerca rigenerano l'albero AND/OR a partire dallo stato attuale, sebbene questo sia di fatto un sottoalbero di quello generato dalla precedente invocazione del metodo Player.getAction().

Una prima ottimizzazione potrebbe essere, quindi, quella di memorizzare dinamicamente gli stati generati in modo che ad ogni passo non debba essere generato tutto l'albero ex novo.

Tale soluzione, ovviamente, comporterebbe un aumento della complessità di spazio, pertanto altro accorgimento che si potrebbe adottare sarebbe quello di fare sì che entrambi i giocatori consultino il medesimo albero.