UNIVERSITÀ DELLA CALABRIA

DIPARTIMENTO DI
INGEGNERIA INFORMATICA,
MODELLISTICA, ELETTRONICA
E SISTEMISTICA

DIMES

Progetto di Intelligenza Artificiale e Rappresentazione della Conoscenza

Gruppo: NotAnAi

Studenti:

Mattia Gatto 216649

Francesco Maria Granata 216648

Serafino Salatino 214455

Professore:

Luigi Palopoli

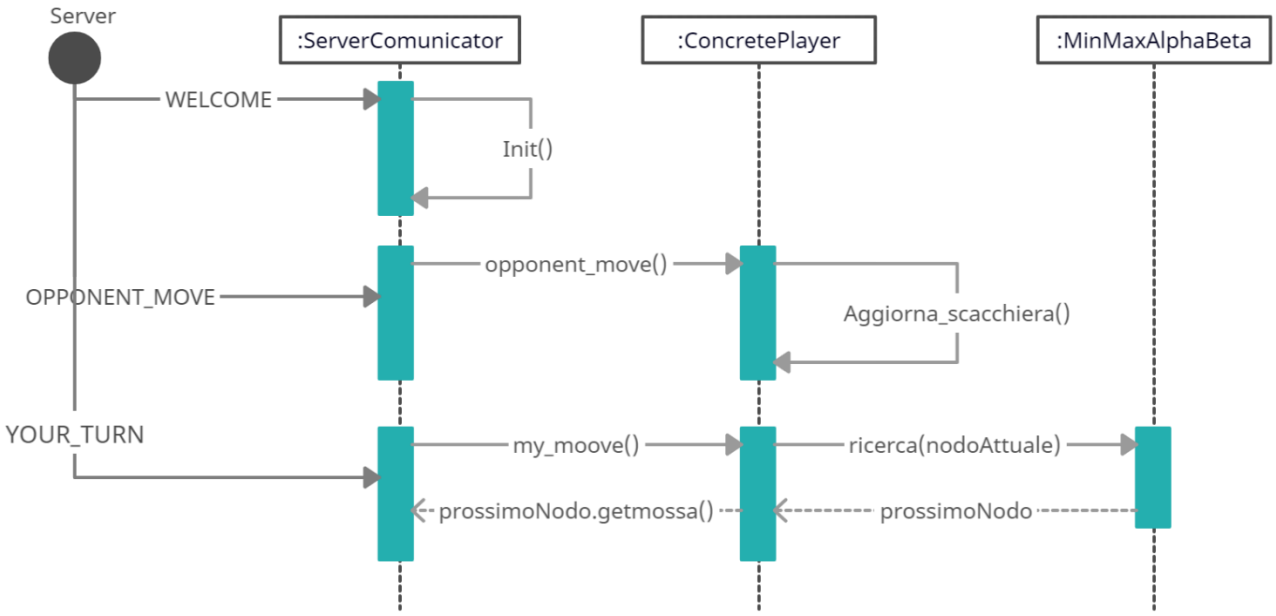
Sommario

Struttura del Progetto

Il progetto dispone di una struttura interna formata da tre package ognuno dei quali formato internamente da un certo numero di classi:

- Controller;
- Player;
- SpazioRicerca;

Ecco una figura che mostra come il flusso di informazioni, ricevuto un input dal server, passa attraverso le varie classi:



Al fine di rendere più efficiente, sia in termini di complessità spaziale che di complessità temporale (visto il limite ad 1 secondo per ogni mossa), si è deciso di effettuare alcune scelte implementative.

1.1 Scelta dei pesi da assegnare alle euristiche

Nel momento in cui viene instaurata la connessione tra il socket e il ServerComunicator, che corrisponderà al master della comunicazione, si è deciso che, al fine di rendere migliore il giocatore in termini di scelte da effettuare per poter raggiungere il prima possibile una soluzione goal, vengono assegnati differenti pesi alle singole euristiche definite nel paragrafo successivo in base al tipo di colore che verrà assegnato dal Server all'intelligenza artificiale.
Infatti, tramite l'utilizzo di un algoritmo genetico, discusso nei capitoli successivi, sono stati creati due array di pesi da associare al giocatore bianco e al giocatore nero poiché empiricamente ci si è resi conto come essi possano differenziare e determinare un notevole vantaggio nell'utilizzare tale approccio.

1.2 Struttura dati utilizzata

Per quanto riguarda la struttura dati, rappresentazione della nostra scacchiera, si è preferito l'utilizzo di un array di byte con il quale si indicano le pedine e le torri, nostre e dell'avversario. L'array pesa 56 byte, ognuno dei quali corrisponde ad una cella della scacchiera. Nello specifico si assegna, per ogni byte, un numero compreso tra 0 e 4, per definire i possibili contenuti all'interno della singola cella della scacchiera:

- “0” indicherà lo spazio vuoto;
- “1” indicherà la pedina bianca;
- “2” indicherà la torre bianca;
- “3” indicherà la pedina nera;
- “4” indicherà la torre nera;

Questa scelta nell’utilizzare un array di byte è sembrata essere una ottima scelta progettuale al fine di riuscire a ridurre la complessità spaziale che esploderà via via usando l’algoritmo di ricerca visto che dovrà andare a generare un numero di figli esponenziale, ognuno dei quali corrisponderà ad una configurazione della scacchiera subito dopo l’aggiornamento della mossa che la identifica. Inoltre, l’utilizzo della rappresentazione di una singola cella attraverso il tipo byte risulta essere migliore rispetto alla definizione di un intero, poiché l’intero tende ad occupare 4 byte.

1.3. Spazio di Ricerca

Per quanto riguarda le scelte relative allo sviluppo della definizione dello spazio di ricerca e l’algoritmo che andrà a valutare tale spazio, è stato sfruttato l’oggetto nodo.
Il nodo viene definito attraverso queste caratteristiche:

- “color” rappresenta il colore relativo al giocatore che definirà la mossa per il nodo corrente,
- “stato” è un array di byte che va a definire la configurazione della scacchiera in quel dato instante,
- ”Predecessore” definisce il nodo padre,
- “mossa” è un array di 2 byte, il primo indica la posizione nella scacchiera della torre che ha modificato lo stato corrente, il secondo byte indica la direzione (0 nord, 1 nord-est, etc..),
- “livello” definisce la distanza di questo nodo dalla radice,
- “hval” è il valore della funzione euristica associata a questo nodo (di default impostato al valore di -100).

1.4. Algoritmo di ricerca

L’algoritmo di ricerca utilizzato per muoversi sullo spazio di ricerca al fine di trovare la soluzione goal è l’algoritmo MinMax con tecnica di pruning AlphaBeta.
L’algoritmo è stato implementato in forma ricorsiva. Per quanto riguarda le condizioni di terminazione dell’algoritmo al fine di andare a valutare l’euristica per uno specifico nodo, si definiscono 3 situazioni:

- Nodo foglia
- Livello del nodo pari a 6
- Tempo >= 800 ms

La prima situazione indica un nodo goal (che sia di vittoria, sconfitta o patta).
La scelta di fermarci al livello 6 è stata presa dopo aver eseguito le prove sulla macchina dove si effettuerà il torneo. Si è notato che l’AI ha esplorato in tempo ragionevole 6 livelli, mentre con 7 livelli andava in errore.
La condizione sul tempo (benchè sperimentalmente non si è verificata) è stata inserita per evitare situazioni particolari dove troppi nodi vengono creati e il giocatore va in timeout.
La scelta di utilizzare l’algoritmo minmax con pruning è stata validata con il confronto con l’algoritmo senza pruning. Si è potuto notare che le mosse effettuate erano le stesse ma con tempo superiore, quindi si è optato per la variante con il pruning

Euristiche

Nell' ottica di un problema di intelligenza artificiale, modellato come problema di ricerca, una funzione euristica è una funzione che prende in input una configurazione del problema e restituisce un valore che mira a dare un'informazione qualitativa sulla configurazione presa in esame. Funzioni euristiche valide hanno il vantaggio di poter guidare l'algoritmo di ricerca verso la soluzione ottima "indicando" i path più promettenti che vale la pena esplorare e i path che invece conviene trascurare. Funzioni di questo tipo possono essere realizzate attraverso una buona conoscenza del problema, inoltre non devono avere un costo computazionale troppo elevato in modo tale da non perdere i vantaggi in termini di efficienza che si hanno nell'utilizzarle.

2.1 Obiettivi da perseguire

Nella progettazione delle varie euristiche è stato fondamentale stabilire quali sono gli obiettivi che se perseguiti da un giocatore all'interno di una partita di "Murus Gallicus" lo avvicinano alla vittoria.

1)Bilanciamento delle pedine: In Murus Gallicus disponiamo di due tipi di pedine, le torri e i muri, le torri sono le uniche in grado di muoversi ed hanno un alto potenziale offensivo, i muri se piazzati nei punti giusti sono in grado di bloccare gli attacchi dell'avversario e permettere il movimento delle torri. Avere poche torri riduce la mobilità del nostro giocatore, mentre avere pochi muri vuol dire lasciare tanta libertà di movimento all'avversario.

2)Controllo del centro della scacchiera: Una torre che si trova in prossimità del centro della scacchiera ha una buona probabilità di avere un range di movimento più ampio di una torre che si trova in posizioni più marginali, quindi cercare di ottenere un buon numero di torri verso il centro può portare ad avere più strategie disponibili.

3)Limitare il numero di mosse dell'avversario: Il numero di torri possedute da un giocatore fornisce un'approssimazione del numero di mosse a sua disposizione, quindi cercare di forzare l'avversario a perdere le sue torri può portare ad un notevole vantaggio.

4)Avvicinarsi alla base dell'avversario: Una delle condizioni di vittoria consiste nell'andare a piazzare un muro nella riga base dell'avversario, a tal fine, minimizzare la distanza delle nostre pedine dalla base avversaria è un obiettivo da perseguire.

5)Allontanare l'avversario dalla propria base: Vogliamo massimizzare la distanza delle pedine avversarie dalla nostra base al fine di evitare che l'avversario arrivi a posizionarvi un muro.

Per ognuno di questi obiettivi è stata realizzata una funzione euristica indipendente.

Vittoria e Sconfitta

Sono state realizzate due ulteriori euristiche che valutano rispettivamente se la configurazione in input è di vittoria o di sconfitta. Queste due euristiche però non tengono in considerazione la situazione in cui a uno dei due giocatori sono rimaste solo una o più torri che non si possono muovere, situazione che in effetti sarebbe di vittoria o di sconfitta. Si è scelto di non considerare tale evenienza per tre motivi:

1)Si tratta di un caso raro che capita con una frequenza molto bassa.

2)L'efficienza delle due euristiche appena presentate si ridurrebbe.

3)L'euristica tra quelle presentate nel paragrafo precedente che fa riferimento al ridurre la mobilità dell'avversario copre l'evenienza seppur in modo approssimativo.

2.2 Euristica complessiva

Una volta definite le funzioni euristiche indipendenti l'euristica complessiva non è altro che la somma pesata di tali funzioni.

$e_1 = Bilanciamento\ delle\ pedine$

$e_2 = Controllo\ del\ centro$

$e_3 = Limitare\ le\ mosse\ dell'avversario$

$e_4 = Avvicinarsi\ alla\ base\ dell'avversario$

$e_5 = Allontanare\ l'avversario\ dalla\ propria\ base$

$e_6 = Vittoria$

$e_7 = Sconfitta$

$pesi = \langle \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha_7 \rangle$

$x = configurazione\ da\ valutare$

$$h(x) = \alpha_1 e_1(x) + \alpha_2 e_2(x) + \alpha_3 e_3(x) + \alpha_4 e_4(x) + \alpha_5 e_5(x) + \alpha_6 e_6(x) + \alpha_7 e_7(x)$$

Al variare dei pesi associati alle euristiche indipendenti il giocatore automatico adotterà strategie differenti dando più importanza ad alcuni obiettivi piuttosto che ad altri, rimane quindi la domanda relativa a come tali pesi vengano scelti, obiettivo della prossima sezione sarà rispondere a questa domanda.

Algoritmo Genetico

L'algoritmo genetico è un algoritmo euristico che si pone l'obiettivo di risolvere problemi di ottimizzazione in un tempo ragionevole. L'utilizzo di questo algoritmo nel progetto è dovuto all'incertezza dei valori restituiti dalle euristiche. Come descritto nei paragrafi precedente, ogni euristica restituisce un valore che viene moltiplicato per il peso assegnato a quell'euristica. Le varie strategie, implementate sotto forma di euristiche, permettono di valutare la configurazione attuale della "board". Diverse sono le strategie implementate, diverso è quindi l'impatto di ognuna. Ad esempio, se dassimo un peso molto basso all'euristica che valuta la vittoria, ed un peso molto alto all'euristica che valuta la sconfitta, l'intelligenza artificiale potrebbe adottare una strategia difensiva piuttosto che una strategia offensiva. A causa delle nostre poche conoscenze su questo gioco, si è preferito generare tramite l'algoritmo genetico dei pesi per le varie euristiche così da poter avere una buona intelligenza artificale. L'algoritmo genetico non è un algoritmo esatto, quindi i pesi che ci vengono restituiti sono dei pesi che concorreranno alla formazione di un'intelligenza artificiale subottima che, come vedremo successivamente, non è perfetta.

3.1 Descrizione

L'obiettivo dell'algoritmo genetico è la generazione dei pesi ottimali.
L'insieme dei pesi è chiamato ***cromosoma*** e ogni peso è un ***gene***. Ogni giocatore sarà creato con 7 pesi, corrispondenti alle 7 euristiche definite nel paragrafo precedente. L'insieme dei giocatori è chiamato ***popolazione***.
Le fasi dell'algoritmo sono le seguenti:

1) Generazione della popolazione
2) Calcolo della funzione di fitness
3) Selezione dei migliori in base alla funzione di fitness
4) Fase di crossover
5) Mutazione

Esclusa la prima fase, le altre verranno eseguite ripetutamente al fine di convergere ad una soluzione subottima.

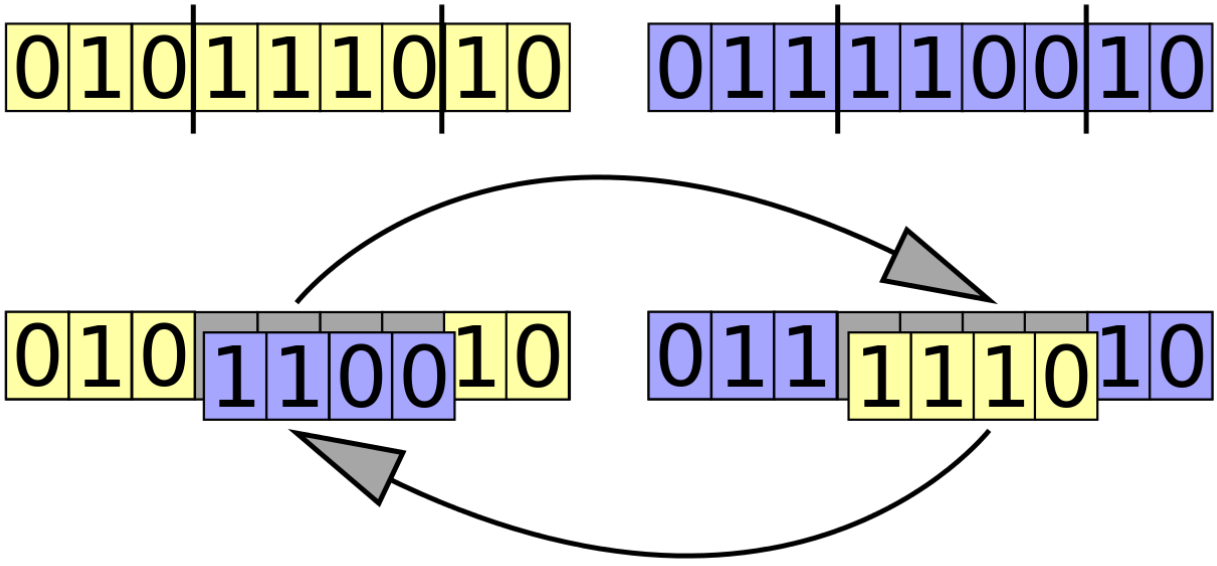
3.2 Implementazione

La prima fase dell'algoritmo si occupa di inizializzare i giocatori. Si fissa un numero di intelligenze artificiali da far sfidare e per ogni entità verrà assegnato l'algoritmo di ricerca corrispondente (in questo caso MinMaxAlphaBeta) e 7 pesi casuali relativi alle 7 euristiche definite.

Nella seconda fase bisognerà assegnare ad ogni giocatore un valore così da poter identificare se esso è migliore o peggiore di un altro. Nel nostro caso, un giocatore è tanto forte quante più partite vince. In questa fase verrà avviato un torneo all'italiana tra le varie AI della popolazione e per ogni partita verrà assegnato un punto al vincitore (indipendentemente se giochi da bianco o da nero).

Nella terza fase vengono scelti i giocatori migliori in base ai punti conquistati. Più precisamente verrà estratta dalla popolazione metà dei giocatori che concorreranno alla fase di crossover.

Nella fase di crossover si genereranno nuove possibili soluzioni (nuove entità) partendo da quelle scelte nella fase di selezione. La tecnica utilizzata è mostrata nella figura sottostante.



Questa tecnica è chiamata ***crossover a 2 punti*** e consiste nel prendere 2 soluzioni (nel nostro caso l'array dei pesi delle euristiche) e scegliere 2 punti così da dividere l'array in 3 parti: una testa, una coda e una parte centrale. Le 2 nuove soluzioni saranno formate come segue:

1) Testa e coda della prima soluzione e parte centrale della seconda soluzione
2) Testa e coda della seconda soluzione e parte centrale della prima soluzione

A differenza della figura, i geni corrispondenti ai pesi sono dei float e non dei bit.

L'ultima fase consiste nella mutazione. Questa fase non avviene sempre ed è per questo che viene fissata una frequenza tramite un valore generato da Math.random. L'obiettivo di questa fase è quello di scegliere casualmente un gene nella lista e modificarlo.

3.3 Sperimentazione

Una volta che l'algoritmo convergeva, esso veniva interrotto ed estratto il giocatore con più vittorie. Per la convergenza non è stato fissato un numero di iterazioni o un tempo specifico ma quando ci si accorgeva che ripetutamente il giocatore con più vittorie era sempre lo stesso allora si interrompeva l'algoritmo.
Esso è stato eseguito più volte e sono stati annotati i pesi relativi ai giocatori migliori. Arrivati ad un certo numero di entità, è stato organizzato un torneo all'italiana per vedere quale tra queste fosse la migliore. In questa fase ci si è resi conto che alcuni giocatori giocavano meglio da bianchi e altri meglio da neri. Come specificato nel paragrafo riguardante la struttura del progetto, i pesi diversi assegnati al ConcretePlayer derivano da ciò.