Università della Calabria

DIPARTIMENTO DI INGEGNERIA INFORMATICA MODELLISTICA ELETTRONICA E SISTEMISTICA

Corso di Laurea Magistrale in Ingegneria Informatica

Intelligenza Artificiale e Rappresenzazione Della Conoscenza

Relazione Progetto Murus Gallicus

Candidati:
Luca Cataldi 207148
Francesco Musmanno 216634
Ivan Scuderi 216635

Relatore:
Prof. Luigi Palopoli

Anno Accademico 2020-2021

Indice

1 Introduzione

Il giocatore automatico da noi sviluppato per il progetto di Intelligenza Artificiale e Rappresentazione della Conoscenza è stato implementato interamente in Java, facendo uso delle librerie standard del linguaggio. Di seguito è riportato il diagramma UML relativo alla struttura delle classi, il quale verrà opportunamente commentato nei paragrafi successivi.

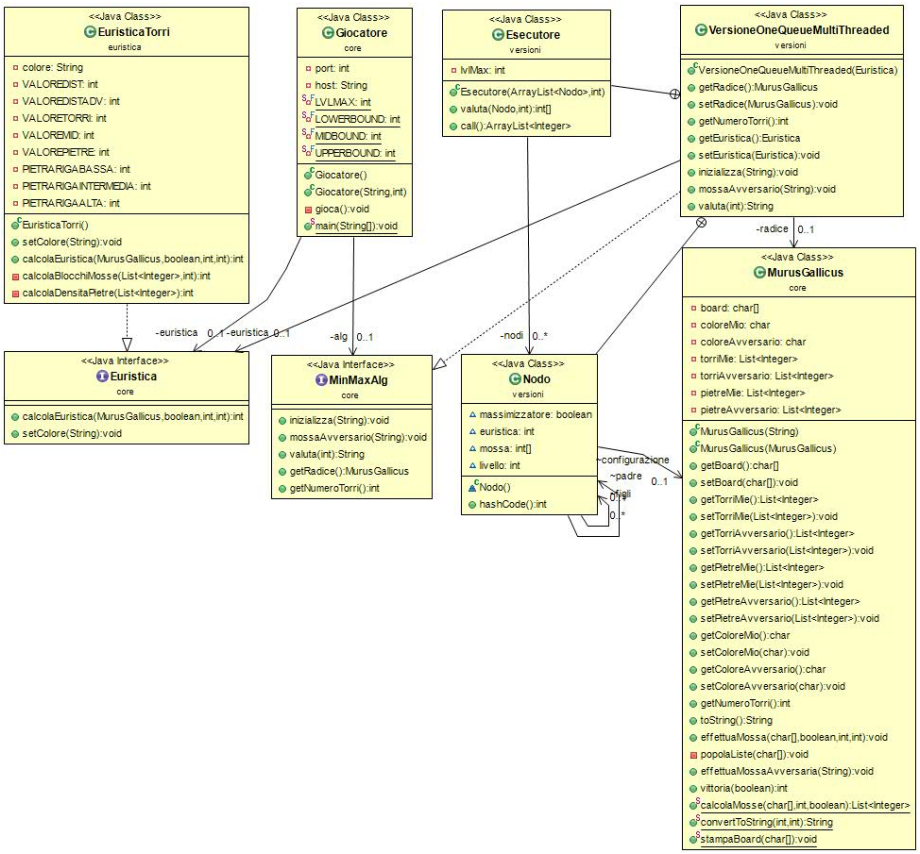

Figura 1: Diagramma UML generale

Come è possibile notare sono state utilizzate delle interfacce con il fine di rendere il codice il più modulare possibile e consentire inoltre, a valle dello sviluppo di classi migliorative, un testing maggiormente efficiente. Ispezionando il codice sorgente infatti, è possibile notare la presenza di classi che non vengono utilizzate all'interno della versione definitiva del progetto, ma che sono state di fondamentale importanza per raggiungere tale versione. Per scelta progettuale è stato deciso di non usufruire del tempo di *warm-up*, in quanto l'albero necessario alla scelta della successiva mossa viene generato interamente ad ogni turno. Nei paragrafi che seguono il tutto verrà trattato maggiormente in dettaglio.

2 MurusGallicus

All'interno della classe **MurusGallicus** è codificata l'intera logica sulla quale l'omonimo gioco si basa. Occorre sottolineare che, per scelta progettuale, è stato deciso di adottare una struttura dati basata su *array* di *char*, in quanto è stata ritenuta più opportuna rispetto a soluzioni differenti, quali le Bitboard, per semplicità di utilizzo e di gestione. É stato preferito questo approccio anche rispetto ad una soluzione basata su matrici, in quanto gli *array* vengono memorizzati sequenzialmente, diminuendo così il numero degli accessi in memoria. Di seguito è riportato il diagramma UML relativo alla classe.

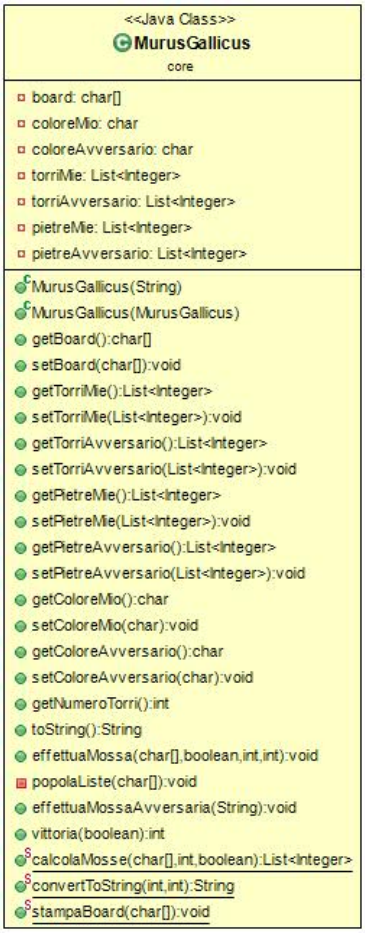

Figura 2: Diagramma UML relativo alla classe MurusGallicus

In particolare, per rappresentare la *board*, si utilizza un *array* di 56 elementi su cui si lavora in modulo 8, per differenziare opportunamente le righe. Gli elementi contenuti all'interno di questa struttura sono:

- "**P**" → rappresenta una casella in cui è collocata una pietra del massimizzatore;
- "**T**" → rappresenta una casella in cui è collocata una torre del massimizzatore;
- "**p**" → rappresenta una casella in cui è collocata una pietra del minimizzatore;
- "**t**" → rappresenta una casella in cui è collocata una torre del minimizzatore.

Ogni metodo all'interno di questa classe tiene conto del colore di entrambi i giocatori, salvato nel momento in cui l'oggetto viene creato in apposite variabili d'istanza. Inoltre, si tiene in considerazione anche il fatto che una mossa possa essere eseguita sia da parte del massimizzatore che del minimizzatore e si comporta di conseguenza nell'aggiornamento della *board*. All'interno delle variabili d'istanza sono presenti alcune strutture dati che sono necessarie al calcolo della funzione euristica; nello specifico, si memorizzano informazioni relative alla posizione sulla scacchiera delle torri e delle pietre di entrambi i giocatori, aggiornandole ogni qual volta si effettua una mossa. Queste informazioni consentono di ottimizzare la valutazione della funzione euristica non essendo necessario il calcolo per ogni nodo foglia.

3 Giocatore

La classe Giocatore si occupa della comunicazione con il server; il suo funzionamento si basa nel ricevere messaggi dal server. A seguito dell'interpretazione di quest'ultimi, la classe, mediante l'invocazione di opportuni metodi, calcolerà la miglior mossa da effettuare che sarà infine inviata al server. Si noti che il **Giocatore** incapsula un oggetto di tipo **MinMaxAlg**, il quale verrà trattato in maniera maggiormente dettagliata nei paragrafi successivi. Il Giocatore, nella sua accezione di base genera 4 livelli ($LVLMAX$) e al contempo, definisce al suo interno alcuni parametri che consentono una gestione dinamica della generazione dei livelli dell'albero. A seguito di un approfondito studio si è notato che, il numero di mosse totali disponibili diminuisce al diminuire delle torri per come il gioco stesso è strutturato; di conseguenza, man mano che la partita evolve, al diminuire delle torri sarà possibile generare più livelli rientrando in ogni caso all'interno delle specifiche di progetto. Più dettagliatamente, la generazione dinamica impiegata lavora come segue:

- $LOWERBOUND = 4$ → nel momento in cui il numero di torri è minore di 4 si genereranno ben 7 livelli;
- $MIDBOUND = 5$ → quando il numero di torri è compreso tra *MIDBOUND* ed *LOWERBOUND* si genereranno 6 livelli;
- $UPPERBOUND = 8$ → quando il numero di torri è compreso tra *UPPERBOUND* ed *MIDBOUND* si genereranno 5 livelli.

Di seguito è riportato il diagramma UML relativo alla suddetta classe.

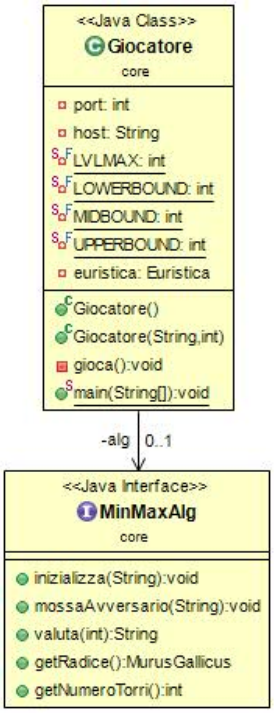

Figura 3: Diagramma UML relativo alla classe Giocatore

Per consentire una migliore comprensione di seguito è riportato il *sequence diagram UML* relativo alla classe.

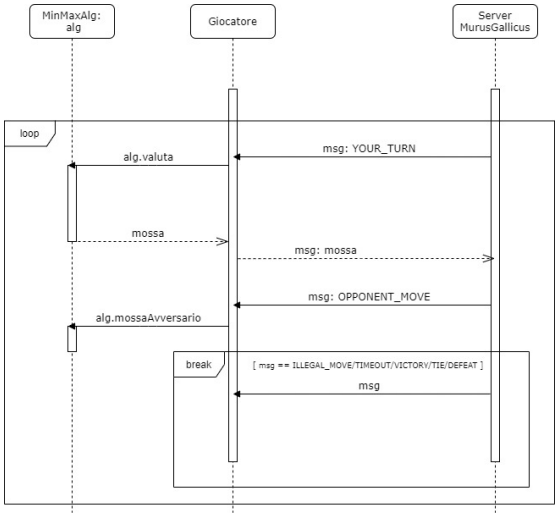

Figura 4: Diagramma di sequenza UML relativo alla classe Giocatore

4 MinMaxAlg

Questo blocco di classi è di vitale importanza per il funzionamento del giocatore automatico, poiché ingloba la logica dell'algoritmo *MinMax* impiegato. Di seguito è riportato il diagramma UML relativo a tale gruppo di classi.

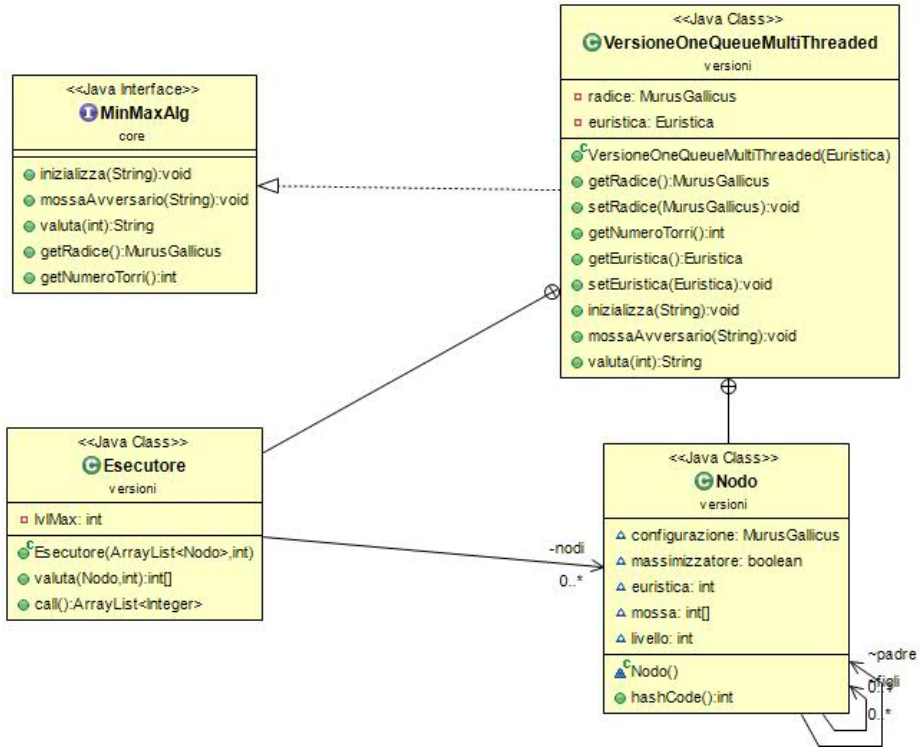

Figura 5: Diagramma UML relativo all'utilizzo dell'algoritmo MinMax

Dopo aver accuratamente vagliato diverse soluzioni, si è deciso di impiegare un algoritmo di visita dell'albero di tipo **Breadth First** iterativo. La struttura dati cardine impiegata è un'*ArrayList* di oggetti di tipo **Nodo**; è importante sottolineare che la coda in questione viene utilizzata sia in fase di discesa, durante la quale viene popolata, che in fase di risalita, in cui viene svuotata propagando i valori di euristica di ogni nodo sino alla radice. Il generico nodo incapsula al suo interno i seguenti elementi:

- un oggetto **MurusGallicus** che rappresenta la configurazione della *board* e tiene traccia dell'avanzamento dello stato del gioco;
- una variabile booleana per identificare se il nodo è massimizzatore o minimizzatore;
- una variabile intera per memorizzare il valore dell'euristica relativo al nodo;
- un *array* di interi di dimensione 2, che identifica la mossa in formato $[src][dest]$ che ha portato alla generazione del nodo;

- una variabile di tipo intero per tenere traccia del livello in cui si trova il nodo all'interno dell'albero.

L'ottimizzazione principale apportata alla generazione dell'albero è stata quella di suddividere il carico di lavoro su due *thread*, conformemente alle specifiche di progetto. In particolare, a seguito di tale modifica, è stato osservato computazionalmente uno *speed-up* di circa il 50% rispetto alla versione che non implementa tale *feature* (*VersioneMinMaxIterOneQueue*). Di seguito è riportato uno schema che rappresenta il funzionamento dell'algoritmo MinMax.

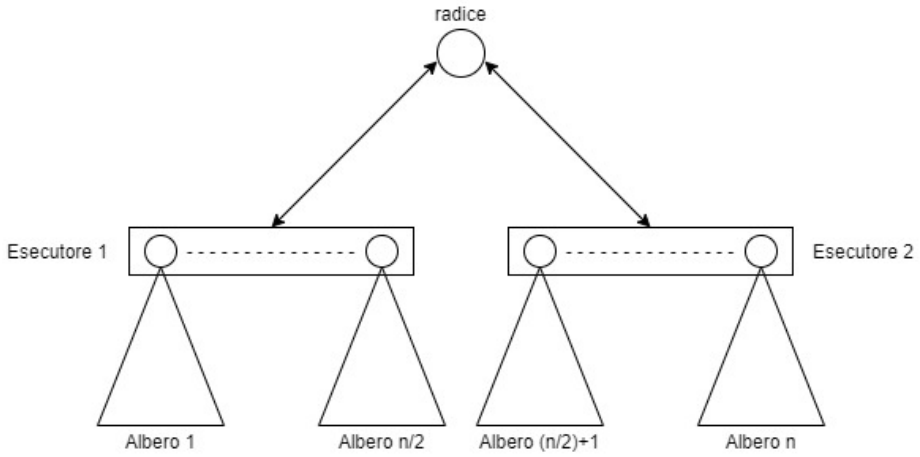

Figura 6: Schema funzionamento algoritmo MinMax

L'utilizzo del *multithreading* inizia a seguito della generazione sequenziale del primo livello; si suddividono i figli della radice assegnandoli a due esecutori, che provvederanno a svilupparne i sotto-alberi relativi. Ognuno di questi *thread* eseguirà l'algoritmo *MinMax* come precedentemente descritto e restituirà alla radice il valore di euristica migliore, tra i nodi a esso assegnati e la rispettiva mossa che ha generato tale nodo. Successivamente, la radice effettua un semplice controllo tra i due valori restituiti e ne sceglie il migliore. A questo punto, si conosce la mossa da effettuare ma nel formato [*src*][*dest*], quindi, si invoca l'apposito metodo dell'oggetto **MurusGallicus** incapsulato all'interno della radice, che ne consente la traduzione in stringa nel formato riconosciuto dal server. Infine, restituisce tale stringa all'oggetto **Giocatore** che a sua volta la invierà, mediante comunicazione *socket*, al server.

5 Euristica

L'euristica adottata espone diversi parametri, come è possibile notare dal diagramma UML di seguito riportato.

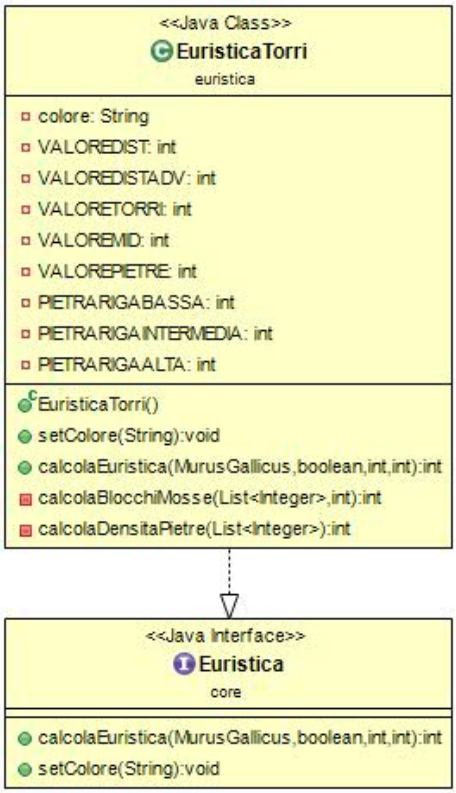

Figura 7: Diagramma UML relativo alla classe EuristicaTorri

I parametri utilizzati sono:

- *VALOREDIST*: tiene conto della distanza (in termini di caselle) della torre maggiormente vicina alla base avversaria;
- *VALOREDISTADV*: tiene conto della distanza (in termini di caselle) della torre avversaria maggiormente vicina alla base del giocatore corrente;
- *VALORETORRI*: tiene conto della differenza del numero di torri di entrambi i giocatori;
- *VALOREMID*: tiene conto della centralità delle torri sulla *board*;
- *VALOREPIETRE*: tiene conto del numero di mosse delle torri avversarie che vengono bloccate dalle pietre del giocatore corrente;
- *PIETRERIGABASSA*: tiene conto del numero di pietre avversarie che si trovano sulla scacchiera nella riga D;

- *PIETRERIGAINTERMEDIA*: tiene conto del numero di pietre avversarie che si trovano sulla scacchiera nella riga E, nel caso in cui il giocatore avversario è di colore nero, oppure C se è di colore bianco;

- *PIETRERIGAALTA*: tiene conto del numero di pietre avversarie che si trovano sulla scacchiera nella riga F, nel caso in cui il giocatore avversario è di colore nero, oppure B se è di colore bianco;

Tutti i parametri sopra descritti, eccetto *VALORETORRI*, vengono opportunamente sommati o sottratti per il calcolo del valore finale dell'euristica, in dipendenza dal trovarsi in corrispondenza di un nodo massimizzatore o minimizzatore. Calibrando opportunamente i parametri descritti è possibile adottare strategie maggiormente offensive o difensive. A seguito di valutazioni empiriche, si è deciso di adottare una distribuzione di valori differente in base al colore che viene assegnato al giocatore corrente. Inoltre, nel corso di vari test si è notato un comportamento anomalo dell'algoritmo, in particolare in presenza di *path* conducenti ad uno stato terminale del gioco (l'algoritmo non sempre sceglieva la soluzione ottima). Per ottenere il nodo *goal* più vicino alla radice è stata implementata una strategia che sfrutta la logica sulla quale si basa l'algoritmo A*, in particolare si somma al valore di euristica il livello del nodo nel caso in cui si è minimizzatori, si sottrae viceversa.