DIMES - Dipartimento di INGEGNERIA INFORMATICA MODELLISTICA, ELETTRONICA E SISTEMISTICA

Corso di Intelligenza Artificiale e Rappresentazione della conoscenza

Anno Accademico 2019-2020

DIPOLE

GRUPPO 19

Docente

prof. Luigi Palopoli

Studenti

Giovanni Fioravanti Matr. 204888
Vincenzo Parrilla Matr. 204870
Vincenzo Saladino Matr. 207151

Indice

Introduzione

Obiettivo del progetto è mettere a punto una implementazione di un giocatore automatico di Dipolo, un gioco da scacchiera. Il codice è stato scritto interamente in linguaggio Java con l'ausilio di alcuni script Python e Shell. Il giocatore automatico è stato predisposto ad interagire con un Server, le cui informazioni di comunicazione sono state fornite nelle specifiche di progetto.
Scopo della relazione è quello di illustrare i punti cruciali nello sviluppo del giocatore automatico, mostrare le tecniche implementative nonché la logica di funzionamento dello stesso. Per cui, dopo un breve riassunto delle regole del gioco, verrà descritta la fase di progettazione e successivamente la fase di implementazione.

1. Specifiche

Dipole è un gioco da tavolo per il quale occorre una *scacchiera 8x8* e *24 pedine*, 12 nere e 12 bianche, che si muovono solo sulle caselle nere della scacchiera.

- All'inizio della partita le 12 pedine di ciascun giocatore sono disposte su due stack come mostrato in figura 1.
- La partita inizia con una mossa del giocatore bianco.
- Lo scopo del gioco è eliminare dalla scacchiera tutte le pedine dell'avversario.

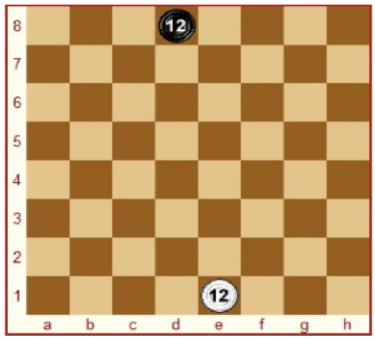

Figura 1: Configurazione iniziale

I giocatori, a turno, muovono uno dei loro stack. È possibile muovere l'intero stack o una sua porzione e ciò è vero sia per le mosse di tipo ***base*** che per quelle di tipo ***merge*** e ***capture*** (vedi sotto). Le mosse di tipo ***base*** possono essere effettuate solo in avanti, sia in verticale che lungo le due diagonali come mostrato in figura 2.

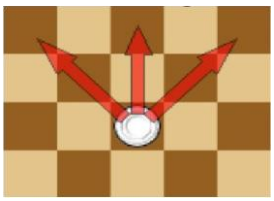

Figura 2: Direzioni di mossa

Il numero di caselle di cui si muove uno stack deve essere uguale al numero di pedine che lo compongono. Uno stack di **n** pedine deve muoversi di **n** posizioni, ma è consentito anche effettuare uno spostamento di **k** caselle, con **k < n**, muovendo solo k delle n pedine e lasciando le rimanenti nella casella di partenza; ciò vale anche per le mosse ***merge*** e ***capture***. Sebbene le sole caselle nere possano essere usate come destinazione, occorre includere nel conteggio anche quelle bianche.
In figura 3 il giocatore bianco dispone di uno stack di dimensione 2 che può muoversi su una delle tre celle indicate (o anche su una delle due celle diagonali vicine, spostando una sola pedina anziché l'intero stack).

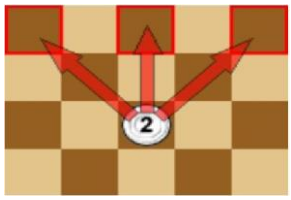

Figura 3: Esempio di mossa base

È possibile muovere uno stack *al di fuori della scacchiera* e ciò ne comporta la sua rimozione. Le mosse che comportano una rimozione di pedine sono assimilate a quelle di tipo ***base***, perciò possono essere eseguite solo in avanti, sia in verticale che lungo le diagonali. Uno stack (o parte di esso) può essere rimosso dalla scacchiera lungo una certa direzione se il numero di pedine che lo compongono è maggiore del numero di celle che può percorrere lungo tale direzione.

Qualora un giocatore non possa più eseguire alcuna mossa valida salta il turno finché non si crei una configurazione che gli consenta di effettuare una mossa.

MOSSE DI TIPO MERGE: È possibile muovere uno stack sovrapponendolo ad un altro del proprio colore. In figura 4 il bianco sposta 3 delle 4 pedine che compongono il suo stack su un altro stack di dimensione 2. Si noti che, per effettuare questa mossa, il bianco scavalca lo stack di pedine nere che incontra sul suo cammino: i movimenti degli stack non sono mai ostruiti dalla presenza di altri stack nella loro direzione di mossa, indipendentemente dalla loro dimensione e dal loro colore. Ciò vale anche per le mosse di tipo ***base*** e ***capture***.
Le mosse di tipo ***merge*** possono essere effettuate solo in avanti.

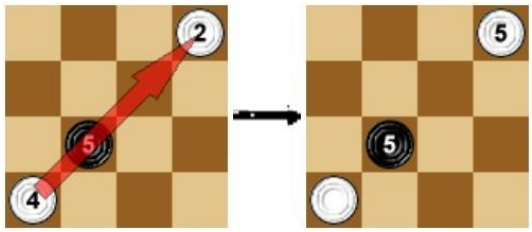

Figura 4: Mosse di tipo Merge

MOSSE DI TIPO CAPTURE: Le mosse di tipo ***capture*** possono essere eseguite in ciascuna delle 8 direzioni. Uno stack può catturare solo un intero stack nemico avente dimensione minore o uguale alla sua. In figura 5, il bianco cattura 1 pedina nera prelevando 2 pedine dal proprio stack di dimensione 3.

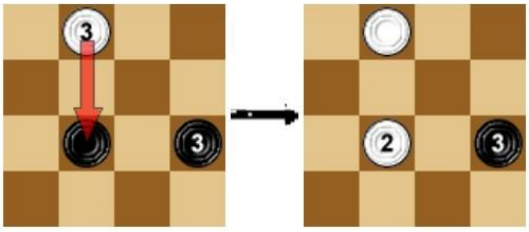

Figura 5: Mosse di tipo Capture

Il riferimento alle regole ufficiali del gioco è disponibile al seguente link:
http://www.marksteeregames.com/Dipole_rules.pdf

2. Progettazione

Il giocatore automatico è stato implementato utilizzando il linguaggio di programmazione Java in ambiente Eclipse. Inoltre, sono stati utilizzati degli script Python per la generazione delle matrici relative a tutte le possibili mosse, i cui risultati sono stati inseriti all'interno del codice Java come variabili statiche, e con l'ausilio di alcuni script Shell è stato possibile testare il giocatore automatico in modo da settare al meglio i valori delle variabili utilizzate nell'euristica. Per la condivisione del codice e il controllo delle versioni è stata utilizzata la piattaforma GitHub.

In fase di progettazione sono stati previsti i seguenti package:

- *communication*: in questo package sono state definite le classi che si occupano di comunicare con il Server, in particolare vengono implementate la classe ***Protocol*** ed una classe che estende *Thread*, ***Listener***. La prima implementa le operazioni che mettono in atto la comunicazione con il Server (e.g. *send()* e *receive()*) e definisce alcune variabili statiche utilizzate per riconoscere e gestire velocemente i messaggi ricevuti dallo stesso; la seconda utilizza la prima per rimanere sempre in ascolto sul canale di comunicazione aperto con il Server e si occupa di notificare al giocatore quando è il proprio turno e quale sia stata la mossa dell'avversario, oltre che stampare a video il contenuto dei messaggi.

- *player*: in questo package si è previsto di implementare la classe ***Player***, ovvero il giocatore, ed eventuali classi di supporto da lui utilizzate (nella versione finale del codice nessuna). Anche la classe Player estende la classe *Thread*: essa implementa un loop al cui interno vengono eseguite le operazioni svolte dal giocatore durante la partita, ovvero ricerca, invio della mossa ed attesa della risposta del giocatore avversario.

- *representation*: in questo package sono state definite due interfacce fondamentali. La prima è ***GameEngine***, la quale calcola e restituisce tutte le mosse valide a partire dalla configurazione passata come parametro al metodo *validActions()*; la seconda è ***RepresentationNode***, la quale rappresenta il singolo nodo dell'albero dello spazio di ricerca. La logica e l'implementazione delle classi concrete è discussa nella sezione relativa alla rappresentazione.

- *strategies*: in questo package viene definita l'interfaccia ***IHeuristic***, una semplice interfaccia utilizzata per distinguere le classi che implementano una funzione euristica *h()* che valuta il valore euristico del nodo passato in input.

- *searching*: in questo package viene definita una classe astratta ***SearchAlgrithm***, la quale rappresenta un algoritmo di ricerca sull'albero degli stati e fa uso di una implementazione concreta dell'interfaccia *IHeuristic* per conoscere la valutazione euristica dei nodi foglia.

Altri due packages di minore importanza sono i packages *util* e *main*. Nel primo, come suggerito dal nome, vengono inserite classi di utilità che servono a mantenere dei riferimenti statici a variabili che contengono informazioni utilizzate in diversi punti del codice, come ad esempio un riferimento ad una classe che implementi *GameEngine* o il numero di mosse totali eseguite dai due giocatori. Nel secondo si inserisce una classe *Main* che contiene il metodo *main()* del programma; questa classe si occupa di controllare che i parametri passati all'applicazione da riga di comando siano validi e, nel caso lo fossero, di avviare l'intero sistema, istanziando le classi fondamentali ed invocando il metodo *start()* dell'istanza del thread *Player*.
Nelle seguenti figure sono mostrati alcuni diagrammi UML dei packages e delle classi, i quale mostrano graficamente quanto descritto finora.

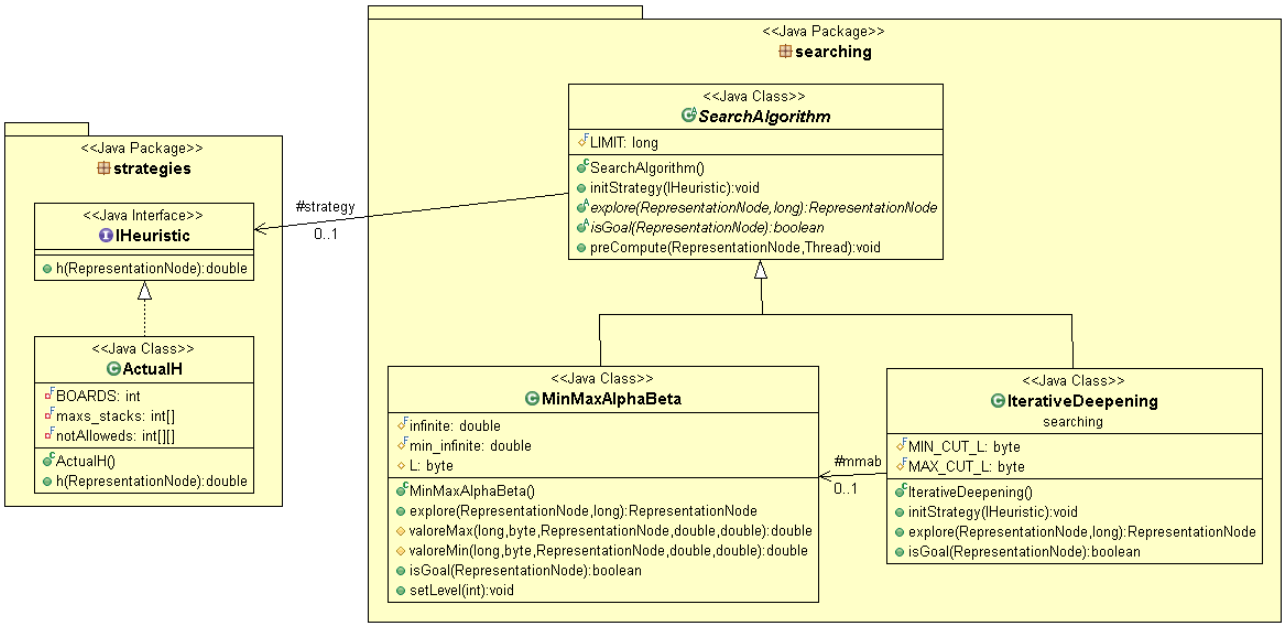

Figura 6: Diagramma UML dei packages 'searching' e 'strategies'

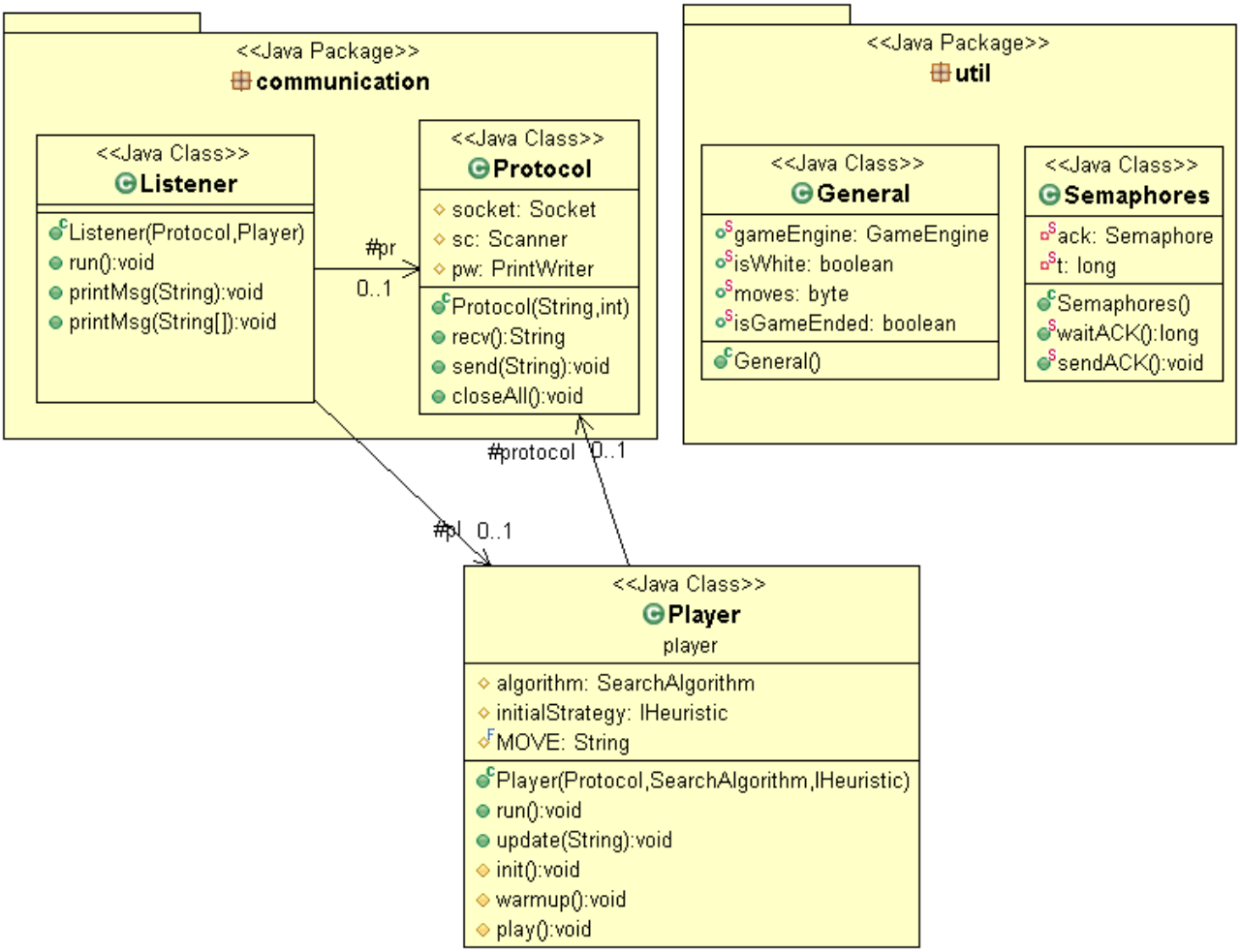

Figura 7: Diagramma UML dei packages 'communication' e 'util'

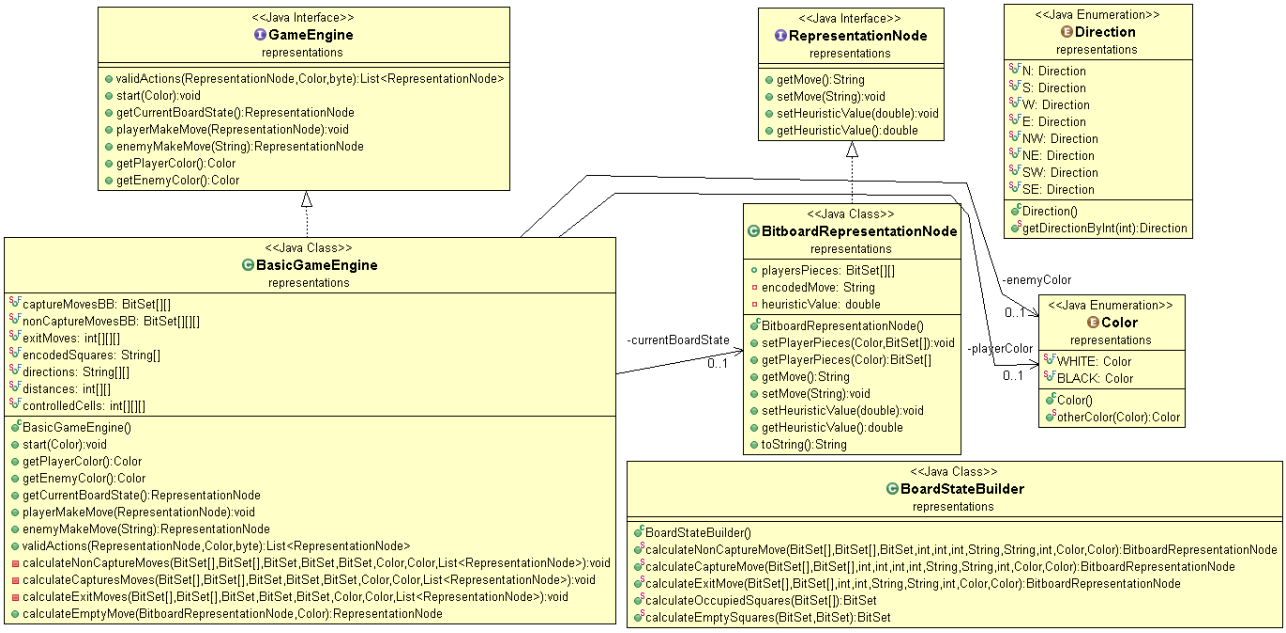

Figura 8: Diagramma UML del package 'representations'

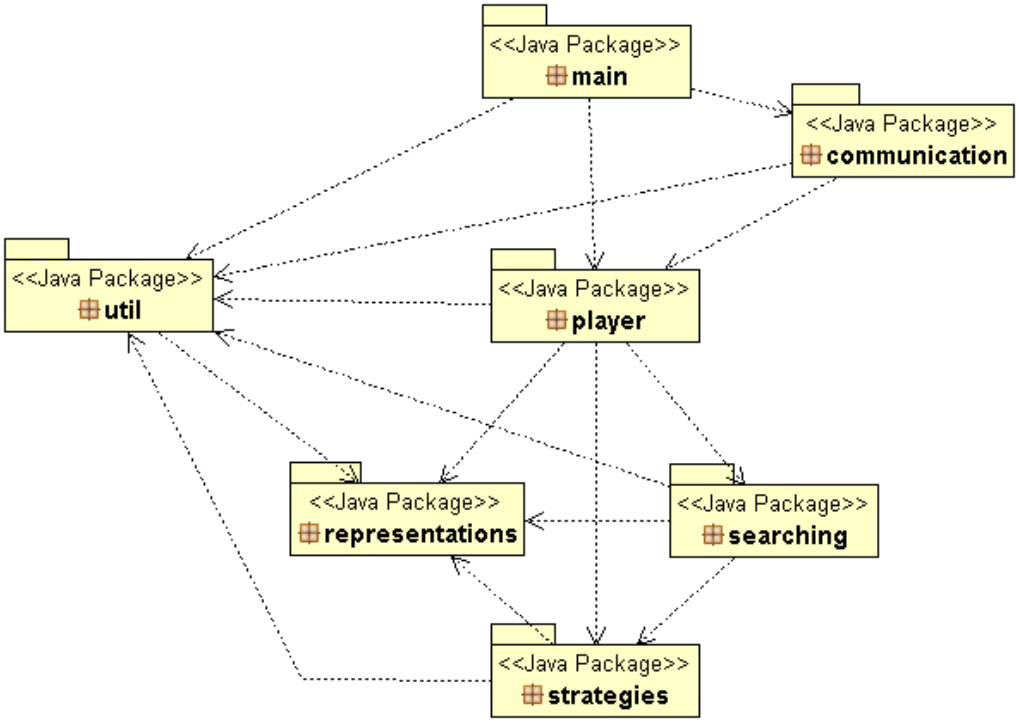

Figura 9: Diagramma UML delle dipendenze fra i packages del progetto

3. Rappresentazione

La prima difficoltà che si incontra nella creazione di un giocatore artificiale per questa tipologia di gioco è sicuramente la vasta superficie di gioco. Per implementare un programma ad alte prestazioni e ridurre notevolmente il carico di ogni singola rappresentazione dello stato attuale e successivo di gioco, è stata adottata come struttura dati una bitboard. Una bitboard è comunemente una rappresentazione a 32 o 64 bit della scacchiera, utilizzata largamente nella creazione di AIs per il gioco degli scacchi data la compattezza della scacchiera (64 caselle).
Molti computer moderni funzionano in modo nativo o almeno molto velocemente con numeri interi a 64 bit e la bitboard è semplicemente un numero intero a 64 bit (senza segno) che contiene informazioni booleane sullo stato della scacchiera. Il motivo di tale scelta è dovuto alla velocità con cui è possibile effettuare delle operazioni complesse con semplici operazioni logiche come l'AND o l'OR. Le bitboard non sono importanti solo perché contengono molte informazioni in un piccolo spazio, ma anche perché la maggior parte dei computer supporta molte operazioni utili sulle bitboard.
L'alternativa alla rappresentazione bitboard era quella che faceva uso di array di *char*. Quest'ultima rappresentazione, tuttavia, è risultata essere molto dispendiosa in termini di verifica del numero di pedine presenti sulla scacchiera, poiché bisognava iterare ogni volta sull'array per contare il numero di pedine presenti. Inoltre, questa operazione doveva essere ripetuta anche per verificare dove si trovava un determinato stack sulla scacchiera.

Dopo un'analisi delle modalità di gioco si è deciso di rappresentare solo le celle nere della scacchiera, poiché quelle bianche non influiscono sulle mosse del gioco (nonostante si tenga conto della loro presenza per spostamenti verticali e orizzontali). Inoltre, si è deciso di considerare uno stack come un "*pezzo*" di dimensione pari al numero di pedine che lo compone, ottenendo, quindi, un totale di 12 pezzi diversi posizionabili sulla scacchiera.
Dunque, si rappresenta una configurazione di gioco, quindi un nodo dell'albero, memorizzando una bitboard da 32 elementi (bit) per ciascun pezzo possibile, per un totale di 24 bitboards diverse (12 per il giocatore bianco e 12 per il giocatore nero). In questo scenario, le operazioni sopra citate possono essere effettuate in maniera immediata, utilizzando oggetti di tipo *java.util.BitSet* ed invocando semplicemente i metodi *cardinality()* e *nextSetBit()*, i quali restituiscono, rispettivamente, il numero di bit posti ad 1 sulla bitboard (quindi il numero di stack di una certa dimensione presenti sulla scacchiera) e la posizione del primo bit posto ad 1 a partire da un dato indice (quindi la posizione sulla scacchiera di uno stack di una certa dimensione).

La struttura dati che memorizza le bitboards è una matrice di dimensione 2x12 di oggetti BitSet (12 BitSet per ciascun giocatore), dove ciascun BitSet, come già detto, è una bitboard a 32 bit. Di seguito è rappresentata una "riga" della matrice:

Posizione sulla scacchiera →	0	1	2	3	4	…	26	27	28	29	30	31
0	A2	A4	A6	A8	B1	…	G6	G8	H1	H3	H5	H7
1	A2	A4	A6	A8	B1	…	G6	G8	H1	H3	H5	H7
2	A2	A4	A6	A8	B1	…	G6	G8	H1	H3	H5	H7
3	A2	A4	A6	A8	B1	…	G6	G8	H1	H3	H5	H7
..	…	…	…	…	…	…	…	…	…	…	…	…
7	A2	A4	A6	A8	B1	…	G6	G8	H1	H3	H5	H7
8	A2	A4	A6	A8	B1	…	G6	G8	H1	H3	H5	H7
9	A2	A4	A6	A8	B1	…	G6	G8	H1	H3	H5	H7
10	A2	A4	A6	A8	B1	…	G6	G8	H1	H3	H5	H7
11	A2	A4	A6	A8	B1	…	G6	G8	H1	H3	H5	H7

↑ Stack di dimensione i+1

Se, ad esempio, il bit della cella evidenziata è settato ad 1, allora vuol dire che in posizione H5 è posizionato uno stack composto da 4 pedine. La classe concreta che implementa tale rappresentazione è chiamata ***BitboardRepresentationNode*** che implementa *RepresentationNode*.

La classe concreta, invece, che implementa l'interfaccia *GameEngine* è ***BasicGameEngine***. Quest'ultima svolge diversi compiti:

- incapsula dei riferimenti statici alle matrici relative a tutte le possibili mosse;
- incapsula delle informazioni importanti, come ad esempio una matrice che indica la distanza fra le diverse celle della scacchiera (o '-1' se non è possibile raggiungere una cella a partire da un'altra) ed il numero di celle potenzialmente controllate da ciascuno stack in ogni cella;
- incapsula la rappresentazione della configurazione iniziale e la aggiorna ogni volta che un giocatore effettua una mossa (quindi incapsula la configurazione 'corrente');
- genera tutte le configurazioni raggiungibili (con una mossa) a partire dalla configurazione passata come parametro al metodo *validActions()*; per tale scopo vengono invocati tre metodi che distinguono le varie tipologie di mosse, ciascuno dei quali, a sua volta, si serve dei metodi della classe *BoardStateBuilder* per generare le singole configurazioni (e.g. il metodo *calculateCaptureMoves()* itera sulle strutture dati e ogni volta che una configurazione può essere generata l'istanza concreta viene restituita a seguito dell'invocazione del metodo *calculateCaptureMove()* della classe *BoardStateBuilder*).

Poiché durante una singola iterazione dell'algoritmo *MiniMax* il numero di oggetti *RepresentationNode* potrebbe essere molto grande e, di conseguenza, questi potrebbe occupare grandi quantità di memoria, ogni volta che il *BasicGameEngine* genera una nuova configurazione crea degli *aliasing* strategici con il nodo padre. In pratica, viene allocato spazio soltanto per quelle bitboards che vengono modificate in seguito all'applicazione dell'azione che genera la nuova configurazione, mentre per tutte le altre bitboard (immutate) viene creato un riferimento a quelle già allocate dal nodo padre. In figura 10 viene mostrato un esempio di questo processo.

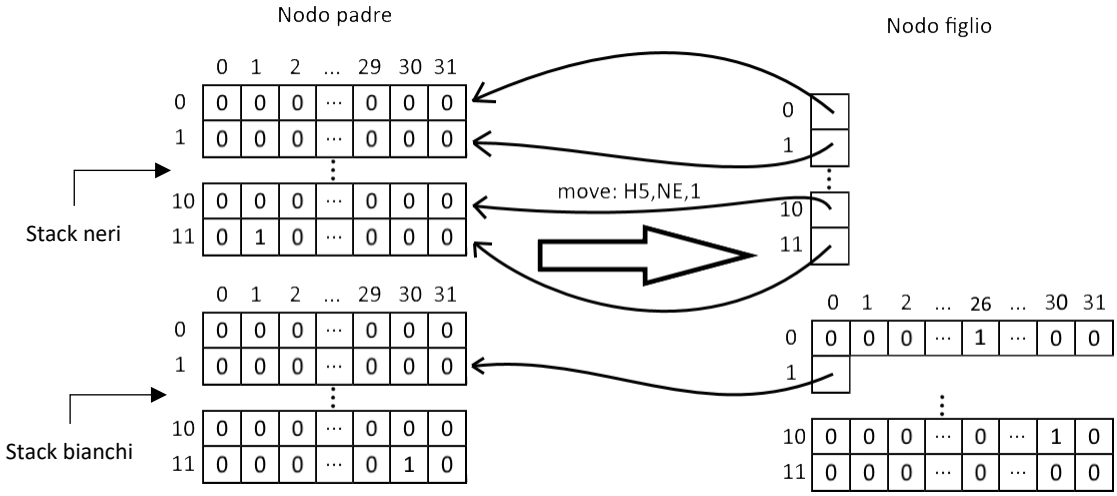

Figura 10: Esempio di aliasing alla generazione di una nuova configurazione

Nell'esempio il nodo padre è la configurazione iniziale, mentre il nodo figlio è ottenuto generando la configurazione risultante dall'azione "H5,NE,1"; come si può notare il nuovo spazio viene allocato solo per gli stack di dimensione 12, 11 ed 1 del giocatore bianco, poiché sono gli unici ad essere variati rispetto alla configurazione del nodo padre.

4. Algoritmo di ricerca

Con la rappresentazione a bit si riesce a guadagnare dal punto di vista prestazionale rispetto ad altre alternative come matrici di interi bidimensionali o monodimensionali o array di *Stringhe* o di *char*. Esse risultano molto leggere e compatte in memoria e le operazioni bit a bit come AND, OR o SHIFT sono fortemente ottimizzate sul calcolatore.
Per esplorare le varie configurazioni di tale rappresentazione viene utilizzato come algoritmo di ricerca **Iterative Deepening**, il quale esegue iterativamente l'algoritmo di ricerca **MiniMax**, il quale a sua volta fa uso della tecnica di *pruning alpha-beta*.
Per l'esecuzione di tale algoritmo viene stato fissato un tempo limite di 950 millisecondi, trascorso il quale bisognerà necessariamente restituire la prossima mossa da effettuare; il *cutting level* massimo è fissato a 15, mentre si utilizza 5 come cutting level di partenza. Durante la generica iterazione i-esima viene eseguito l'algoritmo *MiniMax*, il quale esplora l'albero fino al massimo al livello di taglio C_i, considerando i nodi su quel livello come foglie. Durante il periodo di tempo fissato, fino a quando non si raggiunge il limite, *Iterative Deepening* incrementa il cutting level di uno ed avvia una nuova ricerca *MiniMax*.

Un requisito fondamentale per ottimizzare la potatura dei nodi dell'albero è la buona gestione della generazione delle mosse. Infatti, quando viene invocato il metodo che genera tutte le possibili mosse a partire da una determinata configurazione, si è fatto in modo che queste vengano generate nel seguente ordine:

1. mosse di cattura
2. mosse di spostamento
3. mosse di uscita

Poiché l'esplorazione dell'albero avviene in *depth first*, verranno prima valutati i nodi che permettono la cattura di uno stack avversario, poi quelli di spostamento e soltanto alla fine verranno valutate le mosse di uscita dalla scacchiera. Quello che ci si aspetta, in generale, è che il numero di mosse di cattura sia minore del numero di mosse di spostamento ed uscita e, di conseguenza, la totalità di queste mosse possa essere valutata più velocemente. Inoltre, ci si aspetta nella maggior parte dei casi che le mosse più promettenti siano di cattura o di spostamento; permettendo all'algoritmo di visitare prima queste mosse e, quindi, di valutarne prima il valore euristico, si cerca di favorire una potatura più efficiente.
Per cercare di migliorare la ricerca sono stati implementati altri tipi di algoritmi; tra questi alcuni prevedevano l'ausilio di strutture dati di supporto per memorizzare i nodi già visitati, altri basati sulla randomizzazione delle scelte per nodi aventi valori euristici uguali. Tuttavia, i tentativi di ottimizzazione dell'algoritmo di ricerca non hanno prodotto i risultati sperati.

5. Euristica

Il valore euristico per la valutazione di un nodo, quindi di una configurazione della scacchiera in un preciso istante, è una quantità che tiene conto di diversi fattori. In particolare, è la somma di cinque diversi contributi, analizzati nel dettaglio di seguito:

1. *Differenza di pedine*: questo contributo è il valore risultante dalla semplice differenza di pedine possedute sulla scacchiera da ogni singolo giocatore;

2. *Mosse restanti*: questo contributo è un valore che valuta il numero di mosse necessarie a far uscire tutte le pedine di ciascun giocatore spostandole una per volta, senza considerare se gli spostamenti sono in pratica eseguibili o meno. Ad esempio, nel caso mostrato nella figura 11, il valore risultante dalla configurazione per il giocatore bianco è pari a 7x4=28. Questo calcolo viene eseguito per ogni stack presente sulla scacchiera e viene calcolata la differenza fra le mosse del giocatore e quelle dell'avversario;

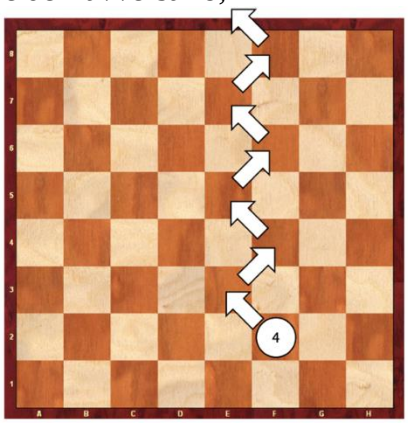

Figura 11: Esempio di calcolo delle mosse restanti

3. *Celle controllate*: questo contributo valuta il numero di celle in avanti che ogni singolo stack controlla, ovvero su quali caselle può essere spostato (per intero e non), senza considerare se gli spostamenti sono in pratica eseguibili o meno. Nelle due scacchiere rappresentate in figura 12, lo stack bianco da 4 pedine può muoversi e su tutte le celle evidenziate, per un totale di undici celle, e lo stesso vale per lo stack nero da 4 pedine. Bisogna sottolineare che gli spostamenti laterali sono solo "ideali", poiché entrambi gli stack non possono muoversi lungo le direzioni Est ed Ovest a meno di una mossa di cattura; tuttavia, si è ritenuto opportuno considerare anche queste celle per tenere conto della posizione orizzontale della pedina. Infine, viene calcolata la differenza fra le celle controllate dal giocatore e quelle controllate dall'avversario;

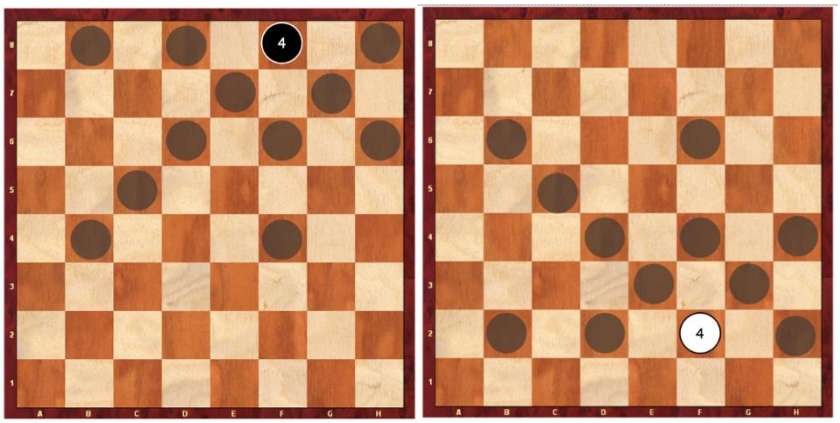

Figura 12: Esempio di calcolo delle celle controllate

4. *Numero di stack*: questo contributo è un valore che tiene conto del numero di stack del giocatore. In particolare, è stato allocato il seguente array di valori:

0	1	2	3	4	5	6	7	8	9	10	11
2	4	4	2	1	1	1	1	1	1	1	1

L'indice i-esimo dell'array (più uno) indica la dimensione dello stack i-esimo, mentre il valore al suo interno è un limite al numero di stack che si possono avere sulla scacchiera.
Cioè si sta imponendo che sarebbe preferibile avere:
⇒ massimo 2 stack di dimensione 1 e 4
⇒ massimo 4 stack di dimensione 2 e 3
⇒ massimo 1 stack di dimensione 4, 5, 6, 7, 8, 9, 10, 11 e 12
Questo contributo permette di tenere sotto controllo gli split di uno stack, in modo da evitare di avere troppi stack di dimensione ridotta (i.e. dimensione 1).

5. *Vantaggio*: questo contributo è un valore che viene calcolato in base alla posizione di ogni singolo stack e valuta se quella configurazione è vantaggiosa o meno in termini di cattura. Se lo stack preso in considerazione si trova in una posizione in cui può catturare uno stack avversario, allora viene sommato 1 al vantaggio; al contrario, se lo stack in questione può essere catturato, viene sottratto 1.

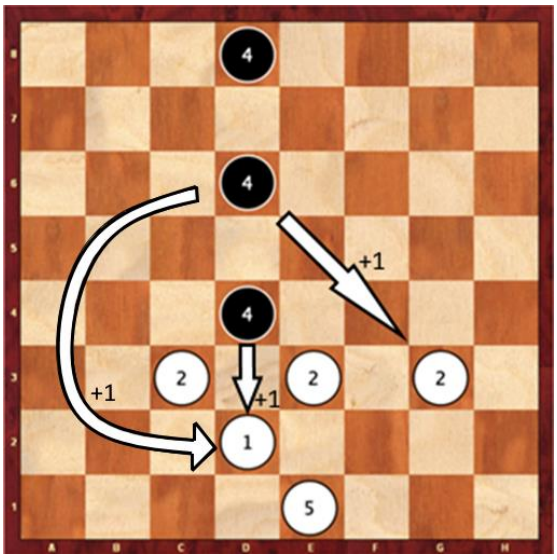

Figura 13: Esempio di vantaggio per il giocatore nero

Dunque, il vantaggio è calcolato in relazione alla distanza tra stack e alla dimensione e si ottiene sommando il vantaggio di ogni singolo stack. In figura 13 è illustrata una possibile configurazione della scacchiera; volendo calcolare il vantaggio del giocatore nero, si deve considerare ogni singolo stack.
⇒ Stack in posizione D8: per questo pezzo il vantaggio è nullo, poiché negli spostamenti disponibili dello stack non si incrociano stack avversari.
⇒ Stack in posizione D6: per questo pezzo il vantaggio è pari a +2, poiché può catturare lo stack posizionato in G3 e lo stack posizionato in D2;
⇒ Stack in posizione D4: per questo pezzo il vantaggio è pari a +1, poiché anche questo stack può catturare lo stack posizionato in G3;
Per il giocatore bianco il vantaggio è nullo, quindi in questo esempio non incide negativamente sul vantaggio del giocatore nero.

Il valore euristico viene calcolato nel seguente modo:

```
1.  int totpp = 0, totep = 0, //numero di pedine possedute da ciascun giocatore
2.  totpm = 0, totem = 0, //numero di mosse restanti a ciascun giocatore
3.  totpc = 0, totec = 0, //numero di celle controllate da ogni stack
4.  tots = 0, //numero di stack totale
5.  advantage = 0;
6.
7.  double value = (totpm-totem) + advantage + (totpp-totep) + tots + (totpc-totec)*0.4;
```

Dunque, come si può notare, il valore euristico è ottenuto dalla somma dei contributi appena analizzati. L'ultimo valore viene ridotto del 60% rispetto agli altri, poiché dopo una fase di testing si è rilevato che il giocatore risulta troppo "*aggressivo*" in diverse fasi di gioco, in particolare l'euristica che considera questo valore al 100% portava gli stack formati da più di 4 pedine a spostarsi in avanti verso il bordo della scacchiera, con un forte calo delle prestazioni del giocatore.
La somma di tutti questi contributi cabla al suo interno delle mosse strategiche che costringe il giocatore avversario a disporre gli stack in modo da poter essere catturato dal giocatore automatico. Inoltre, dopo diverse partite (anche con giocatori umani) è stato notato che il giocatore implementato favorisce le *catture all'indietro* rispetto a quelle in avanti, cioè, quando si trova in una condizione in cui ci sono due possibilità di cattura di uno stesso stack avversario ma una di queste porterebbe lo stack del giocatore in una posizione più arretrata (in riferimento al colore del giocatore), allora viene favorita quest'ultima.

6. Considerazioni finali

In generale, le performance ottenute sono in linea con quelli che sono gli obiettivi del corso, ovvero ottenere un giocatore artificiale capace di confrontarsi con un giocatore a livello intermedio. È importante sottolineare che le capacità di calcolo del giocatore permettono di raggiungere una profondità di ricerca media di 7-8 livelli nel tempo fissato (950 millisecondi), e nelle fasi finali di gioco, quando il numero di pedine è molto basso, si raggiunge il cutting level massimo (15). La rappresentazione a Bitboard permette di non gravare eccessivamente sulla memoria, che nei vari test effettuati è stata perfettamente gestita dalla JVM.