DIPARTIMENTO DI
INGEGNERIA INFORMATICA,
MODELLISTICA, ELETTRONICA
E SISTEMISTICA
DIMES

Corso di Intelligenza Artificiale

Relazione del progetto "Dipole"

Gruppo RIP BLUE

Studenti:

Morelli Lorenzo 207038

Piluso Antonio 204865

Rosano' Antonio 204967

Professore: Luigi Palopoli

Anno accademico 2019/2020

Introduzione

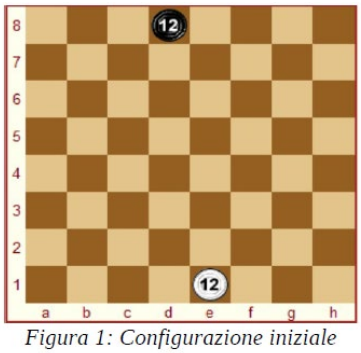


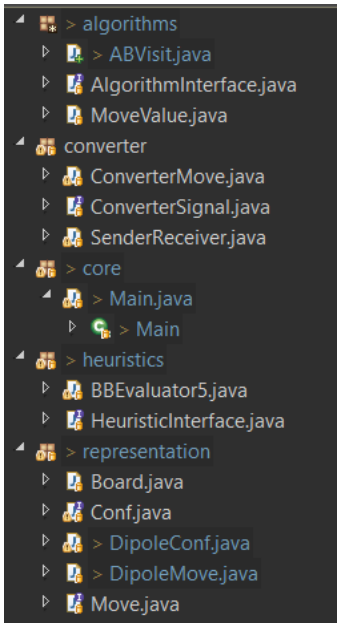
Figura 1: Configurazione iniziale

Il progetto consiste nell'implementazione di una AI in grado di poter giocare al gioco da tavola "Dipole". Lo sviluppo di tale AI è stato intrapreso utilizzando Java come linguaggio di riferimento e la sua implementazione è stata suddivisa in 3 macroaree:

- Rappresentazione dello stato del gioco.
- Euristica.
- Algoritmo di visita.

Per facilitarne lo studio e lo sviluppo, il team ha deciso di dividersi le macroaree di competenza in modo da poter procedere in parallelo con lo studio e l'approfondimento in un determinato campo, utilizzando un approccio brain storming per poter condividere le conoscenze apprese e poter prendere delle decisioni progettuali in maniera democratica. Ogni membro del team si è dunque specializzato in una determinata macroarea, ma la totalità del progetto è comune, conosciuta e condivisa dall'intero team. Lo sviluppo del progetto ha delineato diversi componenti:

- **Algorithms**: tale package contiene la logica di visita del grafo.
- **Converter**: contiene le classi di utilità che sono necessarie per catturare, inviare e convertire i pacchetti in mosse e viceversa.
- **Core**: contiene il main e vari metodi di utilità per dialogare con il server.
- **Heuristics**: contiene la classe designata per calcolare lo score della configurazione.
- **Representation**: contiene diverse classi per gestire la rappresentazione dello stato, la generazione di mosse e la loro applicazione.



Successivamente sono state descritte nel dettaglio le scelte progettuali in maniera più dettagliata per ogni macroarea.

Rappresentazione

Per realizzare una AI per giochi da tavola bisogna innanzitutto andare ad effettuare una corretta rappresentazione interna della scacchiera in modo tale da mantenere le posizioni dei pezzi e ottenere le informazioni relative alle possibili mosse. Tale rappresentazione deve essere realizzata in modo tale da ridurre il carico computazionale e l'utilizzo della memoria. La scelta è dunque ricaduta nell'utilizzo delle bitboard. Una bitboard consiste in un array di bit (64 bit o 32 bit) dove ogni bit va a rappresentare uno spazio o un pezzo della scacchiera. Nonostante nel gioco del Dipole si vanno ad utilizzare solo 32 delle 64 caselle, per semplicità di rappresentazione si è comunque scelto l'utilizzo di bitboard a 64 bit rappresentate attraverso l'utilizzo dei numeri Long di Java, dove il bit meno significativo corrisponde alla cella **H8** mentre il bit più significativo corrisponde alla cella **A1** (rappresentazione little endian rank-file). Si è scelto questo modello di rappresentazione sia perché è possibile ottimizzare l'utilizzo della memoria tramite i Long, riducendo il numero delle strutture dati presenti all'interno dell'applicazione, e sia perché le bitboard vengono manipolate dalle operazioni binarie, che corrispondo alle operazioni più veloci che una macchina può eseguire. La logica binaria risulta effettivamente molto utile per effettuare interrogazioni alla struttura dati della scacchiera: gli operatori **and**, **or**, **not**, **xor** e **shift** possono essere utilizzati, ad esempio, per verificare quali siano le caselle vuote, quali minacciate da altre pedine, quali occupate da pedine nemiche e via dicendo. Processori anche non potentissimi riescono comunque ad effettuare un numero elevato di

operazioni binarie al secondo e, nei processori con architettura a 64 bit, la rappresentazione tramite bitboard risulta essere ancora più efficiente. Una configurazione, che corrisponde ad un determinato stato della partita, viene rappresentata mediante l'uso di un oggetto della classe **DipoleConf**, dove al suo interno sono presenti due bitboards, che vanno a rappresentare la posizione e il numero di pedine del giocatore Nero e del giocatore bianco, e un array al cui interno sono presenti delle bitboards, ognuna delle quali va a codificare le possibili tipologie di pedine (da 1 a 12) che possono essere presenti all'interno dello stato di gioco. Da una configurazione è possibile ottenere tutte le possibili mosse del giocatore di turno, le quali potranno essere applicate a quel determinato stato per dare vita ad ulteriori configurazioni. Da questo si deduce che gli oggetti facente parte della classe **DipoleConf** andranno a formare i nodi del grafo che sarà visitato attraverso l'utilizzo dell'algoritmo scelto. Le possibili mosse sono divise in 5 categorie in maniera tale da aiutare l'euristica a dare un peso maggiore a determinate azioni piuttosto che ad altre. Tali categorie sono:

- **BACKATTACK**: Corrispondono agli attacchi in dietro;
- **FRONTATTACK**: Corrispondono agli attacchi in avanti;
- **MERGE**: Corrispondono alle mosse di merge tra due stack dello stesso giocatore;
- **QUIET MOVE**: sono le mosse che si effettuano quando si sposta una pedina su una posizione vuota;
- **DEATH**: Che corrispondo alle possibili morte che si possono effettuare;

Per ognuna delle pedine nella scacchiera verranno calcolate queste cinque categorie di mosse e rappresentate attraverso l'utilizzo di una bitboard (una per ogni categoria). All'inizio della realizzazione della rappresentazione le mosse venivano calcolate attraverso l'utilizzo delle operazioni di shift, ottenendo così inizialmente una rosa completa di movimento alla quale poi venivano effettuate operazioni logiche per ottenere le varie tipologie di mosse e rimuovere le mosse illegali. Successivamente però, per ottimizzarne lo sviluppo, si è scelto di precalcolare tutte le rose di movimento delle pedine attraverso l'utilizzo di maschere, inserite in un'apposita struttura dati all'interno della classe Board, alle quali vengono tolte le mosse non possibili, dovute alla grandezza dello stack, e suddivise nelle 5 categorie elencate precedentemente. Una mossa però non viene rappresentata attraverso l'utilizzo di una bitboard, bensì attraverso l'utilizzo di un oggetto **DipoleMove** che contiene al suo interno alcune informazioni riguardanti l'azione effettuata (cella di provenienza, cella di destinazione, cella di arrivo, ecc…). Una possibile ottimizzazione pensata durante la fase di progettazione consisteva nell'andare a codificare la mossa in 23 bit.

```
/* * encoding move in 23 bit: 6 fromSq + 6 toSq + 3 distance + 4 type + 1 color +
        * 3 move type
        *
        * @param fromSq
        * @param toSq
        * @param type
        * @param black
        * @param tp    0 backAttack; 1 frontattack; 2 quietmove; 3 merge; 4 death;
        * @return int
        */
```

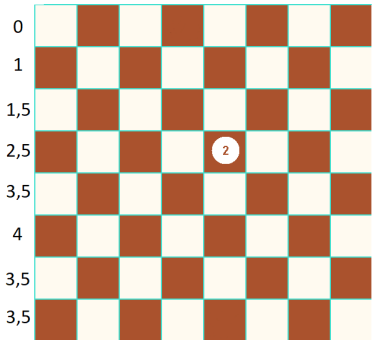
Questa scelta implementativa è stata però accantonata, in quanto effettuava numerosi calcoli per codifica e decodifica della mossa, nonostante riuscisse ad ottimizzare la memoria poiché si andava a lavorare con int e non più con oggetti Move. Una volta ottenute tutte le possibili mosse attraverso la funzione getActions presente all'interno della classe DipoloConf, esse vengono applicate nella configurazione attraverso il metodo applyTo presente nella classe DipoleMove.

Euristica

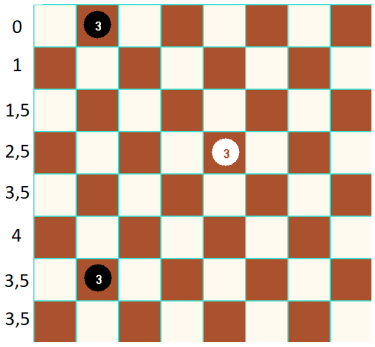
La valutazione dell'euristica si basa su 5 variabili principali:

1. **Numero di pedine**: corrisponde banalmente al numero di pedine che il giocatore ha sulla scacchiera.
2. **Material**: viene calcolato il valore di ogni stack di pedine in relazione alla sua posizione sulla scacchiera.
3. **Mobility**: rappresenta il "raggio d'azione" di ogni stack, ovvero il numero di mosse legali che può effettuare.
4. **Back attack**: calcolo del numero di attacchi all'indietro che il giocatore può effettuare.
5. **Front attack**: calcolo del numero di attacchi in avanti che il giocatore può effettuare.

Analizziamo ora un po' più nel dettaglio il calcolo di alcuni parametri. Com'è facile intuire, il conteggio delle pedine è un operazione banale ma necessaria. Lo scopo del gioco è proprio quello di eliminare tutte le pedine dell'avversario, quindi una configurazione in cui si ha un numero maggiore di pedine deve avere una valutazione maggiore. Per quanto riguarda il calcolo del **Material**, si scorre la scacchiera dal basso verso l'alto e per ogni stack presente, si moltiplica il numero di pedine che lo compongono per il valore della riga in cui si trova. Date le regole del gioco si è infatti deciso di dare maggiore importanza alle pedine posizionare nella zona medio-bassa della scacchiera, andando poi a decrementare tali valori man mano che le pedine si avvicinano al bordo superiore e quindi alla morte. I valori attribuiti alle righe della scacchiera sono rappresentati nella seguente immagine:

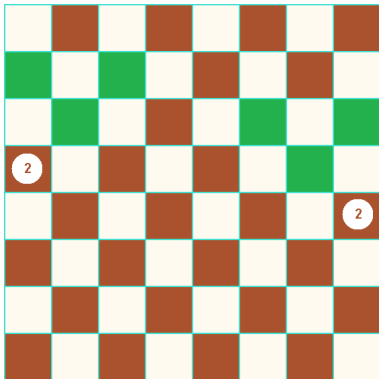


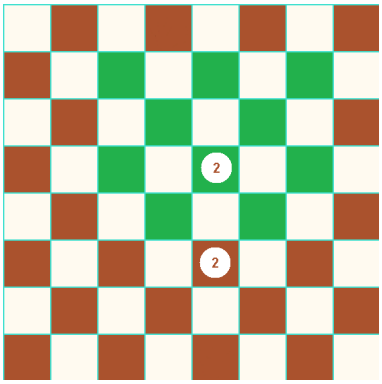
In tale esempio, il Material dello stack è dato dal numero di pedine (2) per il valore della riga (2,5), ottenendo quindi un risultato pari a 5. I valori dati alle righe, oltre a "scoraggiare" i movimenti troppo in avanti delle pedine, incoraggiano ad effettuare i back-attack. Analizziamo il seguente esempio:



In questo caso il giocatore bianco può scegliere se effettuare un attacco in avanti o indietro. Ovviamente la scelta migliore sarebbe la seconda, perché altrimenti lo stack si ritroverebbe sul bordo della scacchiera con la possibilità di effettuare solo mosse di death. Possiamo vedere come questa scelta è perfettamente codificata nel calcolo del Material, che risulterà essere pari a 0 nella configurazione in cui è stato effettuato l'attacco frontale, e pari a 10,5 nella configurazione con attacco all'indietro.

La **Mobility** rappresenta invece i possibili movimenti di ogni pedina. Essa si basa sull'idea che più scelte si hanno a disposizione, più forte è la propria posizione. Una configurazione in cui ogni stack di pedine è posto sui bordi della scacchiera è teoricamente meno vantaggiosa rispetto ad una configurazione in cui gli stack sono posti nella parte centrale della scacchiera, in quanto in quest'ultima il suo raggio d'azione è maggiore.





Il calcolo in questo caso è abbastanza banale, in quanto consiste semplicemente nel conteggiare il numero di tutte le possibili mosse, ad esclusione delle mosse di attacco (frontale o indietro). Il numero dei possibili attacchi è invece conteggiato nei valori di back attack e front attack.

Per ottenere una distribuzione più uniforme dei valori, è stato inoltre moltiplicata ogni variabile per 5000 e divisa per il valore massimo che può avere. Il valore precedentemente analizzato nel Material, per esempio, è stato moltiplicato per 5000 e diviso per il massimo valore che la variabile Material può assumere, ovvero 48. Grazie a questa operazione, è stato quindi possibile ottenere una maggiore granularità per l'euristica, riuscendo così a distinguere un maggior numero di configurazioni.

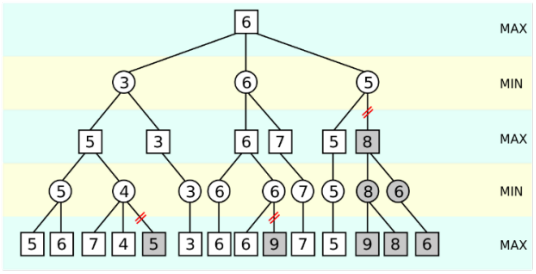
Le 5 variabili precedentemente descritte non influiscono in maniera equa nel calcolo dell'euristica. Proprio per questo motivo, si è scelto di moltiplicare ognuna di esse con delle costanti che ne caratterizzano l'importanza. La costante con valore maggiore (3,5) è stata moltiplicata alla variabile che rappresenta il numero di pedine, perché si è ritenuto essere quella con importanza maggiore. A seguire abbiamo la material moltiplicata per la costante 2,5 e la mobility per la costante 1,2. Per quanto concerne invece le costanti da moltiplicare a back attack e front attack, si è dovuto tener conto del turno. Più nello specifico, se per esempio è il turno del giocatore bianco, si moltiplica il numero dei back attack bianchi per 1,8 e dei front attack per 1,5, mentre si moltiplica il numero dei back attack neri per 1,3 e dei front attack per 1. Si è quindi data maggiore importanza ai possibili attacchi del giocatore che dovrà eseguire la prossima mossa. Com'è possibile notare dai valori, si è inoltre dato un peso maggiore ai back attack rispetto ai front attack in quanto implicano necessariamente un movimento all'indietro (o al massimo lungo la stessa riga) e, come spiegato precedentemente, questo è vantaggioso.

Importante è inoltre sottolineare che la costante 5000, i valori attribuiti alle righe della scacchiera e le costanti moltiplicate sono stati scelti dopo numerosi test.

Algoritmo di visita dello spazio di ricerca

Alpha–beta pruning

Class	Search algorithm
Worst-case performance	$O(b^d)$
Best-case performance	$O\left(\sqrt{b^d}\right)$



Come algoritmo di ricerca la scelta è ricaduta su AlphaBeta, in cui sono state effettuate alcune modifiche:

- La base è un MiniMax con alphabeta pruning in versione ricorsiva
- L'espansione delle configurazioni successive è dettata dalla generazione delle mosse valide, che sono generate partendo dalle mosse con distanza minima fino a quelle con distanza massima, arrivando poi alle mosse in cui vengono scartate le pedine.
- La generazione delle mosse di scarto pedina è stata limitata in quanto si generano solo le death per 1 sola pedina o per tutto lo stack. In accordo ai test si è notato che il tempo che si impiega per generare tutte le mosse death rispetto all'utilità delle stesse è sproporzionato e quasi sempre si riduce a fare una delle due scelte.
- La miglior mossa restituita è la prima Move con il valore di euristica più alto. Sono state testate varie alternative (lista di mosse ordinate, scelta di mossa random a parità di valore euristico) ma la soluzione che ha convinto di più il team è stata quella più semplice ed efficiente in quanto, dai vari test, non c'è stato un sostanziale miglioramento delle scelte effettuate, mentre si è evidenziato un leggero depotenziamento della ricerca (dovuto alla necessità di effettuare computazioni aggiuntive).
- Struttura iterative deepening: per sfruttare l'algoritmo si è deciso di inserirlo in una logica di iterative deepening. Tale scelta progettuale è giustificata dal fatto che, non volendo imporre una distanza fissa, l'iterative deepening permette di limitare la ricerca in contesti temporizzati, consentendo all'algoritmo di lavorare in piena potenza

andando successivamente a scartare l'ultima soluzione nel caso in cui il la ricerca del livello non sia completa. Questa scelta ha vari pro e contro:

- Il contro principale è che spesso il tempo di scelta risulta essere massimo (e quindi l'avversario ha la possibilità di sfruttarlo) in quanto si prova fino all'ultimo a scendere di livello e quasi sempre l'ultima scelta viene scartata (anche se spesso il livello ha pochi nodi visitati e quindi non è affidabile);
- Il vantaggio principale è che l'algoritmo riesce a sfruttare al massimo la situazione di gioco in quanto possiede un range di livelli che può visitare (5-15). Oltretutto in questo modo ci si assicura di riuscire a fermare la ricerca in tempo e restituire una soluzione affidabile.

```
public Move compute(Conf conf) {
    this.searchCutoff = System.currentTimeMillis() + MAX_RUN_TIME;
    this.ibreak = false;
    alpha = Integer.MIN_VALUE;
    beta = Integer.MAX_VALUE;
    MoveValue newBest = null;
    MoveValue oldBest = null;
    int d = startDepth;
    while (!timeUp() && d <= maxDepth) {
        oldBest = newBest;
        if (!this.blackPlayer)
            newBest = ABSearch_R(conf, null, alpha, beta, d, Ply.MAX);
        else
            newBest = ABSearch_B(conf, null, alpha, beta, d, Ply.MAX);
        d++;

    }

    if (oldBest.move == null) {
        return conf.nullMove();
    }

    if (this.ibreak) {
        return oldBest.move;
    } else {
        return newBest.move;
    }

}
```

Figura 1 struttura iterative deepening

La scelta dell'algoritmo è stata guidata dai vari test effettuati sulle altre implementazioni e versioni di algoritmi. L'approccio seguito è stato quello di utilizzare un range di tempo importante (20 giorni) per informarsi sulle varie caratteristiche degli algoritmi disponibili e per aggiornarsi sullo stato dell'arte.

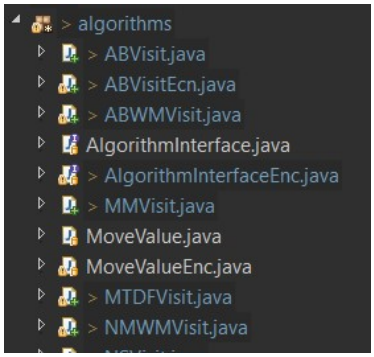


Figura 2 algoritmi sviluppati

Facendo ciò il team ha iniziato a comprendere le caratteristiche di ogni algoritmo, delle varie tecniche (transposition table, zobrist hash, zero-window search etc..) e quindi delle scelte dettate dal dominio che imponeva il progetto. L'idea di base era quella di implementare i più famosi algoritmi procedendo in maniera incrementale: implementare e testare la base degli algoritmi e successivamente realizzare le varianti logicamente più complesse (MiniMax -> AlphaBeta -> NegaMax ->AlphaBeta con memoria -> NegaScout -> MTDF).

Tale linea di sviluppo è stata seguita in maniera più o meno fedele. In particolare, l'obiettivo finale era quello di riuscire ad implementare una versione corretta e funzionante di MTDF che, dalle ricerche effettuate e dai lavori pubblicati da **Aske Plaat** (https://askeplaat.wordpress.com/534-2/mtdf-algorithm/), è uno degli algoritmi che meglio performa in questi ambiti in quanto basato su una logica completamente diversa dagli altri (zero-windows search) e che nei vari test presenti all'interno delle pubblicazioni che lo riguardano (nei quali è sempre presente **Aske plaat**) risulta il migliore in prestazioni e accuratezza. Effettuando però vari test, ci si è accorti che (probabilmente per il tempo limitato) l'utilizzo delle transposition table e dello zobrist hash, ad inizio partita, risultava negativo ai fini del gioco in quanto, partendo dalle stesse informazioni iniziali, le varianti degli algoritmi con memoria (anche l'MTDF) performavano peggio risultando così sempre vincitrice l'AI che utilizzava AlphaBeta. Tale fenomeno è stato attribuito al fatto che gli algoritmi con memoria, dovendo effettuare più operazioni soprattutto ad inizio partita, infatti, nella prima fase di gioco si ponevano in una condizione di svantaggio rispetto agli algoritmi senza memoria, condizione che non si riusciva poi a ribaltare nel corso

della partita, terminando quindi con una sconfitta per gli algoritmi con memoria. Tale fenomeno è stato appurato dalle statistiche che sono state implementate negli algoritmi (nodi visitati, nodi valutati, valori presi dalle TT, depth massima). Anche in questo caso si è notato che gli algoritmi con memoria, avendo 1 secondo a disposizione, nella prima fase di gioco hanno delle statistiche leggermente inferiori rispetto ai corrispettivi senza memoria, cosa che li portava in uno stato sfavorevole di gioco. Tale statistica veniva successivamente ribaltata in quanto, una volta popolate le TT, performavano meglio soprattutto grazie al minor numero di nodi valutati e ad un incremento dei valori presi dalle TT. Da vari test effettuati per delineare una strategia che potesse sfruttare l'andamento sopra citato sono uscite fuori due considerazioni che ci hanno portato a scegliere il noto AlphaBeta.

- Per mitigare la lentezza nella prima fase si è provato a sfruttare i 15 secondi iniziali per popolare le TT in modo da iniziare la partita con un backgroud solido di informazioni. Tale soluzione in pratica ha migliorato le performance degli algoritmi con memoria che però non riescono ad essere paragonabili a quelli senza memoria. Il motivo è da attribuire probabilmente all'utilizzo dell'hash che, pur essendo molto leggero nella versione finale implementata, obbliga a fare il check nella tabella e aggiunge computazioni che lo rendono instabile come algoritmo (per quanto riguarda i limiti di tempo non è rarissimo perdere la partita per timeout perché l'algoritmo non riesce a chiudere i frame aperti in quanto rimane qualche check sulle TT in esecuzione).
- Altre considerazioni riguardano il tempo di risposta: ci si è accorti che aumentando il tempo a disposizione per scegliere la mossa, gli algoritmi con memoria performano meglio di quelli senza (in particolare MTDF) in quanto riescono ad alleviare il fenomeno descritto precedentemente e quindi a partire già con delle buone statistiche; tutto ciò permette di reggere il confronto iniziale e successivamente prendere vantaggio. In generale il test ha evidenziato che con 2 secondi per mossa, gli algoritmi con memoria risultato più performanti dei corrispettivi senza memoria (non vi è uno stacco netto ma quanto basta per vincere). Tali test sono stati effettuati sulla macchina presa in considerazione dal team per lo sviluppo di tutto il progetto e quindi tali considerazioni sono da considerarsi valide solo in parte, ma in generale il fenomeno che si è palesato è stato che: *avendo un sufficiente ammontare di tempo e risorse computazionali, gli algoritmi con memoria (in particolare MTDF) rispondono mediamente in maniera migliore, mentre avendo delle limitazioni importanti in tempo e risorse (caso del progetto e macchina virtuale) risultano più stabili, e spesso anche più performati, gli algoritmi senza memoria che rendono quindi più affidabile il sistema.*

```
Evaluate: -12769
EvaluatedNodes: 1281384
SearchedNodes :2107032
depth :10
depth avg :7.1923076923076925
MOVE A6,NE,1
class algorithms.ABVisit
VALID_MOVE
OPPONENT_MOVE F5,SW,1
YOUR_TURN
```

Dopo i vari test effettuati, la scelta è quindi ricaduta su sull'algoritmo Alphabeta. In media si scende a livello 7 (su un computer performante) e a livello 6 (sulle specifiche della VM). Nei primi turni di gioco, avendo l'albero di ricerca praticamente completo, l'algoritmo raramente scende sotto il 7° livello. Nelle fasi centrali invece, quando i rami del grafo sono particolarmente sbilanciati (in quanto la lunghezza dei path risulta essere molto diversa da ramo a ramo per la presenza delle partite concluse), si toccano picchi di 12 livelli.

Conclusione

Lo sviluppo del progetto ha permesso al team di conoscere diverse sfaccettature del mondo dell'AI. La prima fase di sviluppo, in cui tutto il team ha scelto ed analizzato a fondo un determinato componente, ha permesso al team di sviluppare i concetti appresi al corso per poi metterli in pratica utilizzando un approccio allo sviluppo che spesso è diverso da quello standard, cioè basato sui test. Fin da subito il team ha appreso le criticità del testare il più possibile in maniera formale i meccanismi implementati che teoricamente potevano performare meglio ma che calati nel contesto del progetto risultavano macchinosi e poco adattabili alla causa. L'intelligenza artificiale Rip blue è pronta ad entrare in gioco e che vinca il migliore.