UNIVERSITÀ DELLA CALABRIA
DIPARTIMENTO DI
INGEGNERIA INFORMATICA,
MODELLISTICA, ELETTRONICA
E SISTEMISTICA
DIMES

Intelligenza Artificiale e Rappresentazione della Conoscenza

Progetto Fission

Gruppo «EUREKA»

Anno Accademico 2021/2022

Michele Calabrò
Matricola 224521

Viviana Colantonio
Matricola 224473

Cristian Giuseppe Costa
Matricola 227507

Indice

Capitolo 1

Introduzione

Il gioco in analisi è Fission[1]. Non risulta letteratura sul gioco, tuttavia si può immaginare che si tratti di un gioco a somma zero, per il quale cioè valgono le assunzioni adottate dagli algoritmi MinMax classici.

Il gioco è composto da 2 giocatori, una scacchiera 8 x 8 e 24 pedine totali, di cui 12 per ogni giocatore.

L'obiettivo è eliminare le pedine dell'avversario, mantenendo almeno una delle proprie. Le pedine si possono muovere in tutte e 8 le direzioni cardinali, a patto che ci sia almeno una casella nel percorso che la porta a destinazione. Nel caso in cui la pedina dovesse toccare un'altra pedina, verranno eliminate tutte le pedine ad essa limitrofe (nel raggio di una casella).

Per affrontare il progetto, sono state effettuate delle partite preliminari tramite il software Zillions. In seguito, sono state effettuate delle analisi approssimate sulla struttura dell'albero di ricerca. Le analisi così fatte hanno portato a pensare che il gioco gode di una certa stabilità: una volta ottenuto un vantaggio, diventa difficile per l'altro giocatore recuperarlo.

[1] *Games of Soldiers - FISSION* (2022). URL: `https://www.di.fc.ul.pt/~jpn/gv/fission.htm` (visitato il 17/02/2022).

Capitolo 2

Analisi preliminari

2.1 Analisi dello spazio di ricerca del gioco

È stata svolta una breve analisi qualitativa dello spazio di ricerca del gioco, al fine di capire fino a quale profondità fosse consentito spingersi e se ciò fosse influenzato in qualche misura dallo stato della scacchiera. A tal fine, è stato esplorato l'albero in tutti i 100 livelli rilevanti ai fini del torneo (50 mosse per giocatore), con un vincolo importante: per ogni livello è stato esplorato un numero massimo di nodi pari a 10.000. Durante la visita di tali nodi, circa 1300 sono risultati di vittoria per il primo giocatore, circa 1000 per il secondo, il che rafforza la convinzione, ottenuta giocando contro il software Zillions, che un vantaggio maturato è difficile da ribaltare.

Per quanto riguarda il *branching factor*, esso risulta piuttosto elevato, specialmente durante le fasi di *opening* e di *mid game*. Il valore medio stimato, per ogni livello, è mostrato in figura 2.1.

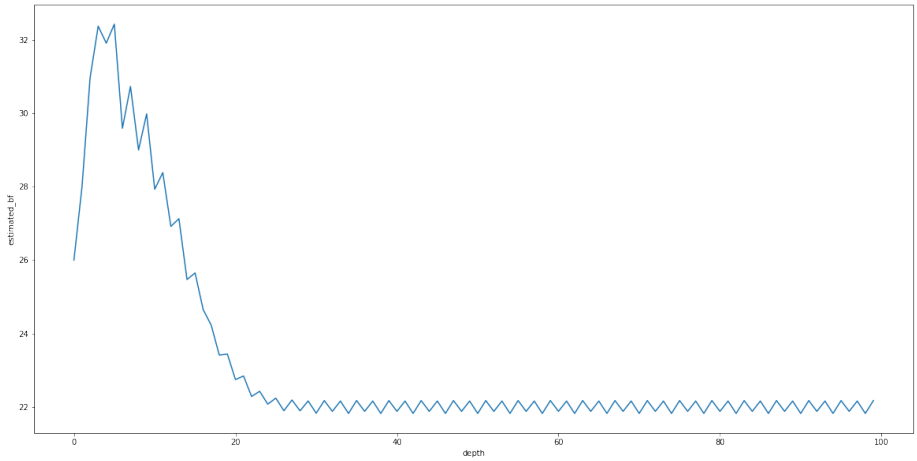


Figura 2.1: Andamento di una stima del *branching factor*

Si può notare il picco che si ha per profondità comprese tra 0 e 10, mentre nelle fasi 'finali' di gioco il *branching factor* non sembra subire cambiamenti significativi.

2.2 Algoritmi di ricerca

La letteratura per giochi a somma zero ha prodotto diverse soluzioni rispetto al problema della costruzione di un giocatore automatico. Gli alberi MinMax sono in genere presenti in molte di queste soluzioni.

Le soluzioni analizzate includono:

- algoritmo MinMax con Alfa-Beta pruning;
- algoritmo SCOUT (Pearl 1980);
- algoritmo SCOUT con Alfa-Beta (*null-window search*), detto anche NegaScout o Principal Variational Search (Reinefeld 1983);
- algoritmo Monte Carlo Tree Search (MCTS) (Chaslot et al. 2008) (Silver et al. 2017).

Ognuna di esse possiede vantaggi algoritmici rispetto alle altre, per cui la scelta di una rispetto a un'altra dovrà essere sufficientemente motivata.

In tutte queste soluzioni è presente un albero MinMax. Un albero MinMax è un albero il cui nodo radice è la configurazione di gioco corrente (per la quale,

cioè, andrebbe scelta la mossa ottimale): il giocatore a cui spetta la mossa in tale configurazione sarà detto «massimizzatore», mentre l'avversario sarà chiamato «minimizzatore». Il motivo di questa nomenclatura risiede nel fatto che l'albero tiene traccia del vantaggio del giocatore massimizzatore, per cui, essendo il gioco a somma zero, il giocatore avversario avrà premura di rendere minimo tale vantaggio (se possibile, negarlo e/o ribaltarlo). Ovviamente tale ragionamento non varrebbe per un gioco non a somma zero.

MinMax con Alfa-Beta pruning L'algoritmo (o meglio, la famiglia di algoritmi) consiste in una visita in profondità (fino a un certo livello) in cui:

- ad un nodo foglia (non espandibile, o perchè terminale o perchè troppo profondo) viene associato un valore di euristica;
- ad un nodo non foglia e del **massimizzatore** viene associato il massimo tra i valori di euristica dei figli;
- ad un nodo non foglia e del **minimizzatore** viene associato il minimo tra i valori di euristica dei figli.

I termini «alfa» e «beta» si riferiscono ai valori ottimi fino a quel momento maturati, durante la ricerca, rispettivamente da massimizzatore e minimizzatore. Se cioè un massimizzatore, durante la sua visita, dovesse incontrare tra i suoi figli uno con valore di euristica superiore al minimo garantito al minimizzatore che lo ha generato (il suo *parent*), egli può evitare di esplorare (*pruning*) i rimanenti figli e restituire immediatamente tale valore di euristica. Una considerazione duale vale per il minimizzatore. Il *pruning* permette pertanto di ridurre artificialmente il *branching factor* dell'albero di ricerca del gioco.

SCOUT L'algoritmo procede in maniera simile al classico *minmax*, cambia unicamente il meccanismo di *pruning*: in SCOUT, non si tiene propriamente traccia dei valori «alfa» e «beta», bensì si hanno procedure di test in questa forma:

```
test(node, value, '>')
```

Il simbolo «>» indica la tipologia di test da effettuare (di maggioranza, di minoranza, etc), il nodo «node» è un nodo ancora non esplorato in una data iterazione mentre «value» è un valore di euristica. Obiettivo del metodo è

stabilire se il criterio scelto, ad esempio «>», potrà essere rispettato dal nodo «node», ossia se vale la pena visitare quel nodo (altrimenti *pruning*).

Si può intuire dunque quali saranno i criteri scelti da minimizzatore e da massimizzatore rispettivamente per decidere se un nodo figlio debba essere esplorato o meno. In particolare, il valore «value», ad ogni visita dei figli di un nodo, è inizialmente impostato al valore ottenuto dalla ricerca (sempre con SCOUT) sul figlio più a sinistra (poi aggiornato a seconda del tipo di nodo, come ottimo corrente). A differenza di minmax con alfa-beta, non si mantiene pertanto un'informazione 'globale' sul valore garantito all'avversario al livello successivo nell'albero («value» non viene propagato ai figli).

L'ordinamento delle mosse è pertanto cruciale.

Alfa-Beta Scout L'algoritmo implementa le procedure di test di SCOUT attraverso visite MinMax con alfa-beta pruning e *null-window search*, ossia 'finestre' di valori a intervallo nullo (alfa = beta).

In particolare, una volta visitato il figlio più a sinistra di un nodo e prelevato pertanto il valore di euristica «value» ad esso associato, gli altri figli saranno 'testati' attraverso una finestra in cui sia alfa sia beta saranno pari a «value» (che verrà opportunamente aggiornato durante la visita). Una visita di questo tipo terminerà non appena sarà stato trovato, ad un certo punto nel sotto albero, un valore di euristica diverso da «value». Se tale valore è minore/maggiore (a seconda del tipo di nodo, massimizzatore o minimizzatore) di «value», tale figlio sarà scartato (pruning), altrimenti sarà necessario rilanciare una visita alfa-beta (con una finestra più lasca, in genere simile a quella di minmax con alfabeta) in modo da prelevare il valore 'corretto' e non un lower/upper bound (a seconda che il nodo esplorato sia, rispettivamente, un massimizzatore o un minimizzatore).

MCTS Gli algoritmi MCTS sono algoritmi che permettono di inferire euristiche opportune sulla base di visite random nell'albero MinMax di gioco. Tali algoritmi tendono a convergere a una soluzione esatta di MinMax. Un vantaggio di MCTS è che, essendo un algoritmo iterativo a convergenza, può essere interrotto in qualsiasi momento e restituire una soluzione approssimata più o meno soddisfacente.

2.3 Scelta dell'algoritmo

Dopo un'analisi degli algoritmi sopra elencati, si è infine optato per algoritmi di tipo *minmax* con alfa-beta pruning, in quanto più facilmente adattabili in strategie personalizzate.

Capitolo 3

Architettura di massima

Il sistema è composto da diverse classi, alcune delle quali (le più importanti) sono modellate in figura 3.1.

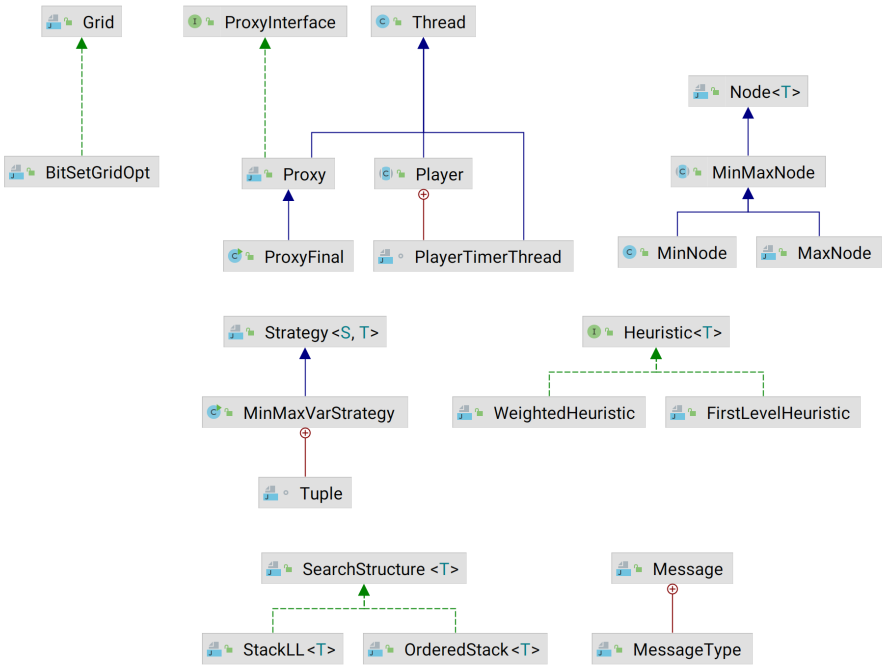


Figura 3.1: Diagramma UML contenente alcune delle classi del sistema

Come è possibile osservare dal diagramma, alcune classi sono *thread*.

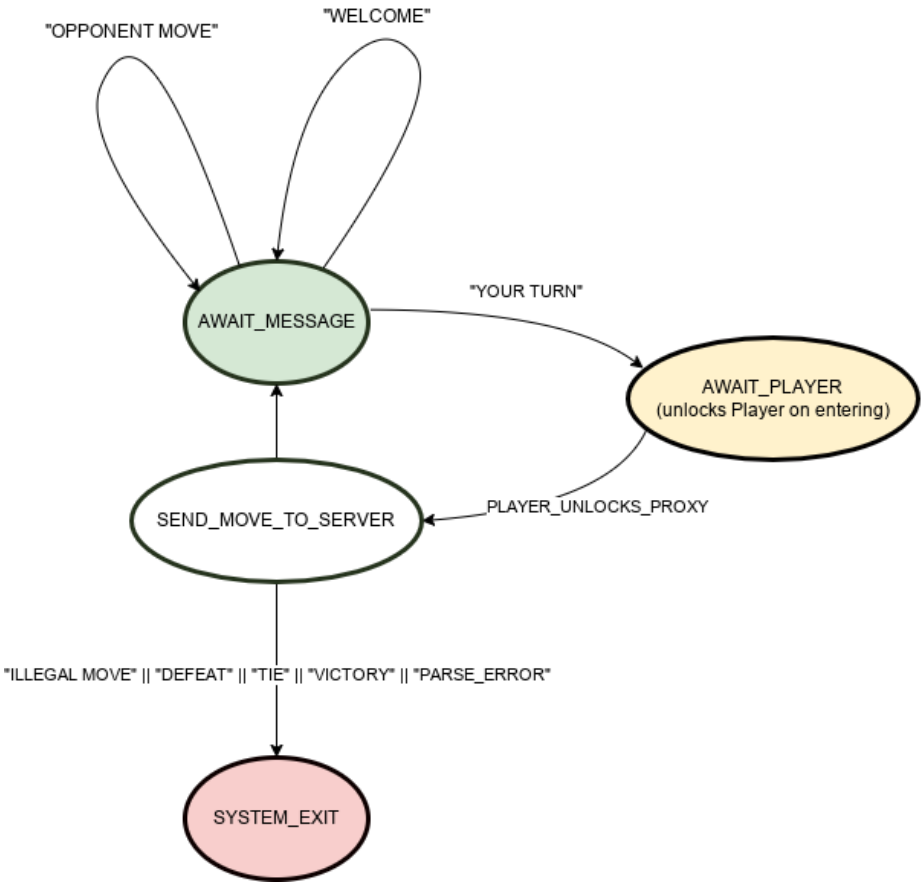


Figura 3.2: Proxy come ASF

In particolare, lo sono i *Proxy*, i *Player*, e alcune inner class dei *Player*, tra le quali *PlayerInitial* e *PlayerTimerThread*.

Ignorando per ora le inner class dei Player, la logica di interazione con il server di Fission e con gli algoritmi di ricerca è concentrata nelle classi Player e Proxy. Le classi interagiscono mediante un comportamento modellabile tramite automi a stati finiti.

In figura 3.2 è possibile osservare la modellazione del Proxy come automa.

La controparte Player, *tightly coupled* a Proxy, è invece modellata in figura 3.3.

Gli stati con colore giallo indicano sincronizzazione/attesa reciproca tra Player e Proxy. Lo stato colorato in rosso indica uno stato terminale, mentre gli stati colorati in verde sono gli stati iniziali per gli automi.

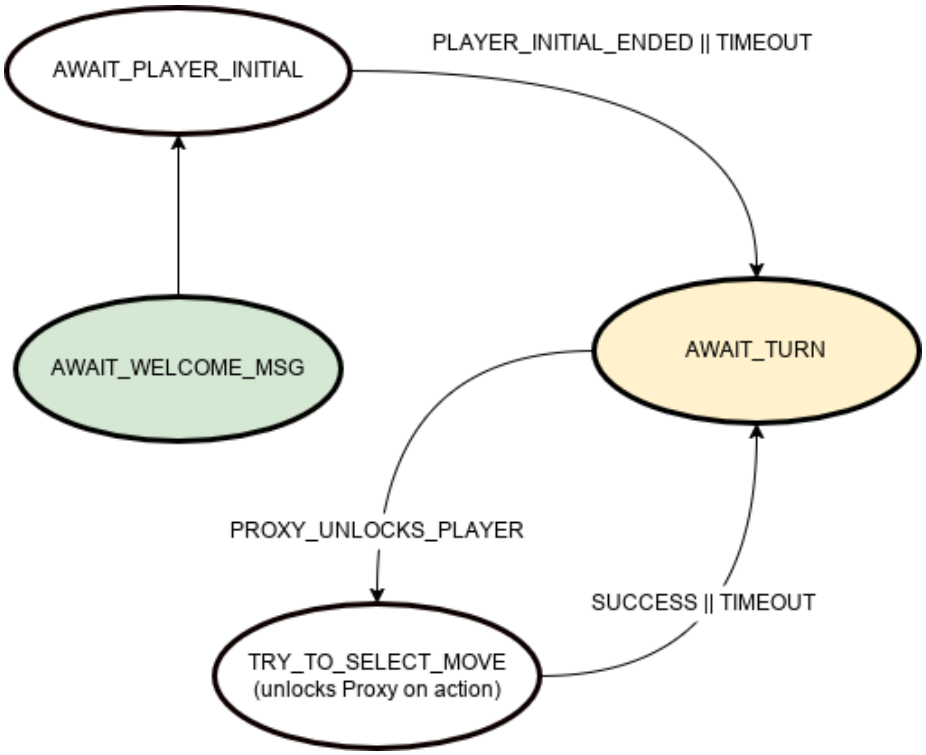


Figura 3.3: Player come ASF

Ovviamente, alle transizioni di stato sono associate azioni non riportate nelle figure di cui sopra. Ad esempio, la ricezione di un messaggio di *WELCOME* da parte del Proxy comporta il salvataggio del colore e l'istanziazione di un thread Player appropriato, con euristiche per il giocatore e algoritmo di ricerca opportuni (tramite costruttore).

Una volta istanziato, il giocatore si preoccuperà di avviare un ulteriore thread (che potrà essere terminato in caso di timeout) che esegue eventuali operazioni utili durante il tempo di *warm-up*.

Appena sarà sbloccato dal Proxy per eseguire una mossa, il Player proverà, entro il tempo limite, a selezionare una mossa opportuna. In caso di timeout (inteso rispetto a un tempo inferiore a 1 secondo, ossia inteso come massimo rischio che si è pronti a correre), proverà a restituire la mossa migliore fin a quel momento calcolata.

Capitolo 4

Strutture dati

Nel capitolo seguente verranno elencate alcune delle tecniche utilizzate per rappresentare una configurazione di gioco e lo spazio di ricerca.

L'obiettivo è facilitare l'implementazione di algoritmi di ricerca oltre che di ridurre il tempo e lo spazio occupato rispetto ad approcci *naive* (lett. «ingenui»), sia per quanto riguarda la generazione delle mosse, sia per quanto riguarda la generazione di alberi *MinMax*.

4.1 La scacchiera di gioco

Come già detto in precedenza, la scacchiera di gioco è una scacchiera 8 x 8, contenente in totale 24 pedine, 12 per giocatore; le pedine sono di un tipo soltanto (non vi sono cioè pezzi specializzati, come negli scacchi) e possono muoversi nelle 8 direzioni cardinali.

La rappresentazione più naturale, ma non ottimale, sarebbe tramite matrice di caratteri, come riportato in figura 4.1.

In Java, una matrice 8 x 8 di *byte* occuperebbe più spazio del necessario, in quanto ogni singola posizione della griglia occuperebbe almeno un byte. Si può facilmente intuire come lo spazio minimo necessario a supportare il gioco, non solo nella sua rappresentazione, ma anche per quanto concerne la generazione delle mosse (il che è fondamentale), sia di 128 bit: ciò è dovuto al fatto che ogni giocatore ha potenzialmente 64 differenti caselle in cui piazzare le sue 12 pedine e verso cui muoversi, mentre la presenza o meno di una pedina in una casella è indicabile con un bit.

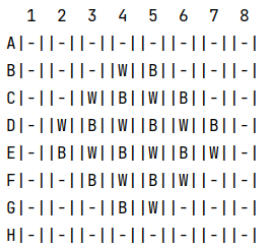
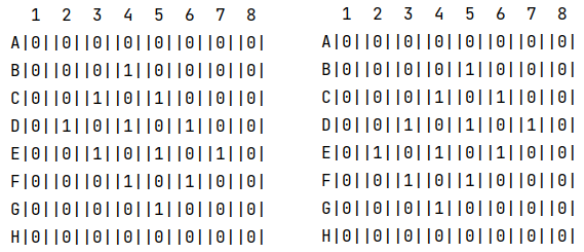


Figura 4.1: Matrice di byte/caratteri



(a) Parte alta (primi 64 bit) (b) Parte bassa (ultimi 64 bit)

Figura 4.2: Rappresentazione a bit (stampata sotto forma di matrice per migliore visibilità)

Java mette a disposizione una struttura dati, il BitSet, la quale permette di lavorare con array di bit (o meglio, array di long): ciò garantisce di ridurre la dimensione di una rappresentazione ad oggetto da 376 Byte a 102 Byte, con un guadagno pertanto del 369% in termini di spazio.

Nella nostra rappresentazione a 128 bit, la parte alta (i primi 64 bit) codifica le posizioni occupate dalle pedine del giocatore bianco, mentre la parte bassa (gli ultimi 64 bit) codifica quelle occupate dal giocatore nero. In figura 4.2 è possibile osservare la rappresentazione a bit della configurazione già mostrata in figura 4.1.

I vantaggi di una rappresentazione simile non si riducono in un ridotto spazio, ma hanno anche implicazioni sulle performance di algoritmi di generazione di mosse e altri, che possono ora sfruttare operazioni bit a bit, supportate anche da maschere. Ad esempio, si possono generare (e sono state generate) maschere a 8 bit come quella in figura 4.3, la quale facilita l'estrazione di informazioni sull'intorno di una posizione.

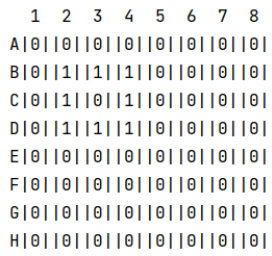


Figura 4.3: Maschera di 8 bit centrata in C3

4.2 Generazione delle mosse

La rappresentazione a vettore di bit permette di ottenere vantaggi non solo in termini di spazio, ma anche di tempo. Ciò è dovuto sia all'implementazione di BitSet, sia alle operazioni bit per bit, in genere più rapide da eseguire su calcolatore se in porzioni di memoria contigue.

Sfruttando maschere a 8 bit, come quella in figura 4.3 e altre maschere (che permettano di filtrare le sole pedine del giocatore bianco/nero) è stato possibile ridurre i tempi di esecuzione per la generazione delle mosse. In particolare, la generazione ripetuta delle mosse al primo livello (dalla prima configurazione di Fission) per 10'000'000 di volte richiedeva circa 18 secondi con rappresentazione a matrice, ma ne richiede circa 13 con la rappresentazione a bit, ottenendo quindi uno *speedup* pari al ~138%.

Gli oggetti che incapsulano le mosse tengono traccia di una serie di informazioni utili, tra cui la tipologia di mossa (se di collisione o meno), il numero di pedine eventualmente rimosse dalla griglia di ambo i giocatori, e altre informazioni di contorno: queste risulteranno utili per ridurre artificialmente il *branching factor* negli alberi di ricerca sfruttati dai nostri algoritmi di ricerca.

4.3 Strutture di ricerca

Gli algoritmi di ricerca utilizzati si basano su alberi *MinMax*. È utile, a questo scopo, introdurre oggetti che possano incapsulare una configurazione di gioco e contenere informazioni aggiuntive, tra cui valore di euristica, profondità nell'albero, eventuali figli. Sono state realizzate due tipologie concrete di nodi: i *MaxNode*, i cui figli, se esistono (configurazione/nodo non terminale) sono dei *MinNode*, e i *MinNode*, i cui figli saranno *MaxNode*.

La suddivisione semplifica la realizzazione degli algoritmi di ricerca, che potranno basarsi su una verifica della tipologia di nodo a runtime per selezionare correttamente la procedura da eseguire.

La generazione dei figli di un nodo si basa su:

- verifica che il nodo non sia terminale;
- generazione delle mosse della configurazione di gioco incapsulata nel nodo;
- filtri sulle mosse generate, per ridurre branching factor.

I filtri sulle mosse si basano sull'assunzione che ambo i giocatori siano razionali, ossia che non compiano scelte che li penalizzino certamente in tutta la partita.

Fatta tale assunzione, sono state individuate alcune tipologie di mosse ritenute non razionali; in particolare:

- mosse di collisione che portano ad un bilancio netto di pedine perse sfavorevole al giocatore che fa la mossa;
- mosse non di collisione che addensano il giocatore che fa la mossa, avvantaggiando l'avversario implicitamente.

Queste mosse saranno filtrate, riducendo artificialmente il branching factor. Qualora al giocatore non rimangano altre mosse, esiste un meccanismo di *fallback* che permette di rimuovere i filtri e ottenere anche mosse non razionali.

Capitolo 5

Algoritmi e strategie

Nel capitolo seguente verrà analizzata la strategia di gioco, variante dell'algoritmo MinMax con Alpha-beta pruning, responsabile della scelta della mossa ottima in corrispondenza di ogni turno del giocatore. Dopo l'analisi della configurazione di gioco il giocatore ha un insieme di mosse papabili di essere giocate, l'obiettivo è quello di scegliere tra queste la mossa più razionale.

5.1 Variante con uso di bucket

La scelta della mossa è sicuramente l'operazione più delicata per un giocatore automatico; ad ogni turno di gioco la mossa da effettuare deve essere razionale, e dunque, costituire una scelta valida che con buone possibilità conduca il gioco verso la vittoria per il giocatore. La scelta è condizionata da diversi fattori, dalla fase di gioco in cui ci troviamo, dalla possibilità di effettuare una mossa che porta ad immediato vantaggio oppure dalla necessità di procedere con cautela scegliendo mosse non aggressive. Per ciascuna di queste situazioni la strategia deve essere in grado di adeguarsi, valutare i rischi derivanti da ogni singola mossa disponibile, e scegliere tra queste quella che massimizza il proprio vantaggio. In una fase iniziale del gioco la mossa opportuna potrebbe essere pescata da un pool di mosse statiche, caricato durante la fase iniziale del gioco. Tali mosse, dopo attenti ragionamenti, sono considerate le più adatte quando sussiste l'equilibrio iniziale. Nel caso in cui non sia possibile effettuare tale scelta, la ricerca della mossa ottima ricade sull'algoritmo di ricerca MinMax, appositamente modificato per adattarsi al gioco in esame e alla strategia

che si vuole seguire. Filo conduttore nella scelta della mossa opportuna è la semplice assunzione che una mossa che porta ad eliminare un numero di pedine dell'avversario maggiore rispetto a quello proprio è sicuramente vantaggioso per via delle caratteristiche intrinseche del gioco. Una volta ottenuto tale vantaggio, diventa difficile per l'avversario recuperarlo. L'obiettivo è dunque quello di selezionare qualsiasi mossa che porta ad un vantaggio, con una eccezione, se tale mossa ci conduce in uno stato di gioco per cui è l'avversario ad avere la disponibilità di una mossa ancora più vantaggiosa, a tal punto da ribaltare il vantaggio generato, tale scelta viene immediatamente scartata. Ovviamente, tra le mosse vantaggiose, viene scelta quella per cui vi è una differenza più alta nel numero di pedine eliminate. In tutti gli altri casi, ci si affida all'algoritmo di ricerca su albero.

L'algoritmo realizzato può essere considerato come una variante di quello più noto MinMax. A differenza di quest'ultimo, l'algoritmo di ricerca su albero è preceduto da una preliminare fase di suddivisione delle mosse disponibili in bucket. Un bucket è un insieme di configurazioni di gioco, o meglio di nodi MinMax, figli della radice dell'albero, dove ciascuno è il frutto dell'esecuzione di una particolare mossa sulla configurazione di gioco corrente. Ogni bucket ha la proprietà di accomunare nodi MinMax a cui è associato un valore di euristica simile. Tale valore viene assegnato in conseguenza ad una prima visita MinMax a partire dalla radice e con profondità massima pari ad uno. Per valutare un nodo è stata usata una prima euristica, denominata "Euristica di primo livello".

Sono stati previsti tre bucket ciascuno con un fine ben preciso:

- il primo bucket è l'insieme dei nodi, e dunque di mosse, considerate vantaggiose per il giocatore perché si tratta di una mossa tale che il numero di pedine distrutte dell'avversario sia maggiore di quello del giocatore che compie la mossa;
- il secondo bucket contiene tutte quelle mosse, con o senza collisione, che non alterano la differenza in numero di pedine tra i giocatori;
- il terzo bucket, invece, contiene le mosse considerate svantaggiose; cioè tutte quelle che una volta eseguite concedono all'avversario la possibilità di effettuare una mossa a suo vantaggio.

Una prima fase della strategia consiste dunque nell'effettuare una prima visita dell'albero secondo l'algoritmo MinMax, con profondità esattamente pari ad uno ed euristica "FirstLeveHeuristic".

Inizialmente, tutti i nodi figli della radice corrispondono a mosse possibili e vengono raccolti nella collezione "bestAsOfNow" ordinati sulla base del valore di euristica, e rappresentano il meglio che la strategia può scegliere fino a quel momento.

Una volta che i figli della radice sono stati generati e popolati con il valore di euristica corrispondente, come già anticipato, vengono suddivisi nei diversi bucket. La strategia di massima consiste nel trattare in modo diverso i nodi in accordo alla suddivisione.

I nodi del primo bucket sono considerati vantaggiosi, e per tale motivo, su di essi non viene effettuata la normale ricerca su albero ma si è optato per una ricerca beam. Tale ricerca consiste nel recuperare per ogni nodo in questa lista, se esiste, il nodo figlio che rappresenta la best move per l'avversario e valutare il nodo corrispondente secondo l'euristica di primo livello. Per ogni nodo del primo bucket se non esiste un nodo figlio con valore di euristica più alto (vantaggio maggiore per l'avversario), viene salvata la corrispondente azione in una particolare lista denominata "BestFromBeam", a sua volta ordinata in modo opportuno. In caso contrario, tale nodo viene eliminato dalla collezione "bestAsOfNow" perché non più considerato vantaggioso.

Infine, se esiste almeno una mossa vantaggiosa in "BestFromBeam", essa viene restituita al chiamante come mossa ottima.

Per i nodi del primo bucket è riservato, dunque, un trattamento speciale, la strategia consiste nel recuperare da tale lista la mossa ottima risparmiando sul costo della ricerca e guadagnandone in tempo di risposta.

Non sempre esiste la mossa vantaggiosa, per tale motivo bisogna scegliere tra le mosse rimanenti quella più razionale. In ordine, si procede con l'analisi dei nodi del secondo bucket; essi corrispondono tutti a mosse che non alternano la differenza in numero di pedine tra i giocatori. Discernere quale tra queste effettuare è un compito che viene assegnato all'algoritmo di ricerca MinMax con pruning.

Prima di procedere viene creata una ulteriore collezione "VisitedNodes", contente i nodi correntemente visitati per il secondo bucket, con valore di euristica aggiornata sulla base di una seconda metrica denominata "WeightedHeuristic", la cui logica verrà discussa successivamente. Tale collezione, se non è vuota è usata al posto di "BestAsOfNow".

Uno dei parametri essenziali da configurare è la profondità massima di discesa dell'algoritmo di ricerca; esso si basa su una stima del branching factor, valutato sulla radice dell'albero e pari al numero di nodi figli. La profondità

minima di discesa è pari a quattro e cresce fino ad un tetto massimo di sette se il branching factor stimato è più piccolo di una data soglia.

Da questo momento in poi la scelta della mossa ottima farà affidamento unicamente sul valore di euristica associato ad ogni nodo come risultato di tante ricerche MinMax con pruning. Per migliorare i tempi di ricerca, la tecnica di pruning è trasversale a tutte le possibili esecuzioni dell'algoritmo sui diversi nodi del bucket.

Come risultato di questa fase della strategia, ogni nodo del secondo bucket avrà un nuovo valore di euristica associato; al chiamante verrà restituita l'intera collezione "VisitedNodes". Sarà cura del giocatore scegliere tra le mosse promettenti con stesso valore di euristica quale considerare come mossa ottima. Solitamente si procede con una scelta di tipo Round Robin.

In situazioni normali, la scelta della mossa si conclude con la visita del secondo bucket. Può tuttavia succedere che sia il primo bucket che il secondo siano vuoti; in tal caso al giocatore non rimane che pescare la mossa tra quelle considerate poco vantaggiose. Si procede, dunque, con la visita dei nodi del terzo e ultimo bucket e in modo analogo a quanto è stato fatto per il secondo, restituiamo al chiamante la mossa considerata meno svantaggiosa.

La strategia di gioco fa uso complessivamente di due euristiche per la valutazione dei nodi:

- **FirstLeveHeuristic**: utilizzata per attribuire un valore di euristica ai figli della radice, necessaria per il partizionamento dei nodi nei bucket;
- **WeightedHeuristic**: è quella che viene utilizzata per valutare un nodo ad una profondità qualsiasi e si basa su una funzione di valutazione.

5.2 Attacco e difesa

Al fine di migliorare la scelta della mossa in particolari situazioni di gioco sono state sviluppate due strategie mirate per due fasi delicate della partita. Una strategia di attacco è così definita perché è più aggressiva rispetto alla normale strategia di gioco; la scelta della mossa da effettuare potrebbe anche ricadere su una mossa che in realtà è considerata svantaggiosa poiché causerebbe un numero di pedine eliminate del giocatore maggiore rispetto a quelle dell'avversario. Tale situazione è però desiderabile se in una configurazione di gioco il vantaggio in numero di pedine è maggiore o uguale a due e l'avversario predispone di sole due pedine. In questo caso, prima di procedere con la strategia usuale, viene data la

precedenza a tali mosse con il vincolo che il vantaggio in numero di pedine sia comunque mantenuto. Una strategia di difesa, invece, si verifica allorquando il numero residuo di pedine del giocatore è minore o uguale a due e l'avversario ha un vantaggio di almeno una pedina. In questa situazione viene attivata una strategia conservativa, si tende a considerare "non razionali" anche le mosse che in realtà sono mosse positive. In soldoni, le mosse positive sono considerate alla pari delle altre mosse pur avendo un valore di euristica, associato dalla visita ad un livello, più alto rispetto alle altre mosse. Ciò comporta la vuotezza del primo bucket. Entrambe le strategie sono da considerare come strategie alternative ma non sostitutive.

5.3 Gestione timeout

In caso di timeout la strategia ha come obiettivo quelle di restituire la mossa migliore fino a quel momento calcolata. Per tale motivo, durante tutta l'implementazione della strategia, sono stati individuati punti strategici del codice in cui effettuare la gestione del timeout. Come supporto, sono state utilizzate diverse collezioni dati, tra queste "BestAsOfNow", contenente il meglio fino a quel momento. Tale struttura viene inizialmente riempita con tutti i nodi figli della radice, e dunque, con tutte le possibili mosse (tra quelle che hanno superato le diverse condizioni di filtro). Per tutta la durata di gestione dei nodi del primo bucket, in assenza di mosse vantaggiose, ad essere restituita è la collezione "BestAsOfNow". Ovviamente, se esiste una tale mossa, essa viene immediatamente restituita. Superata questa fase, ad essere visitati sono i nodi del secondo bucket e come collezione dati di supporto per la gestione del timeout abbiamo "VisitedNodes". Tale lista, se non vuota ha lo stesso ruolo di "BestAsOfNow", ma contiene i nodi correntemente visitati e con un valore di euristica aggiornati. La stessa collezione di supporto viene utilizzata nel momento in cui si procede con la visita dei nodi del terzo bucket (ciò vuol dire che i primi due bucket sono vuoti). Nel caso particolare in cui tutti i bucket non contengono nodi e il tempo di timeout è stato raggiunto, la collezione "BestAsOfNow" ci assicura che qualcosa venga comunque restituita.

Capitolo 6

Euristiche

In questo capitolo vengono esaminate nel dettaglio le due funzioni di valutazione euristiche applicate ai nodi durante la visita dell'albero, e necessarie al fine di sfruttare al meglio il tempo di calcolo a disposizione.

Esse sono le seguenti:

- **FirstLeveHeuristic**;
- **WeightedHeuristic**.

Entrambe si basano in qualche misura su una funzione lineare pesata, cioè sulla combinazione di valori calcolati in modo indipendente gli uni dagli altri, e poi aggregati insieme per formare il valore finale.

Può essere così espressa:

$$Value(s) = w_1 f_1(s) + w_2 f_2(s) + ... + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

dove s rappresenta lo stato del gioco.

Ogni f_i è il risultato di un ragionamento condotto sullo stato del gioco, mentre w_i è il peso associato che ne indica l'importanza.

Il valore finale è compreso nell'intervallo $[-1, 1]$, dove -1 indica una configurazione di assoluto svantaggio, mentre $+1$ indica una configurazione di assoluto vantaggio. I pesi delle due funzioni di valutazione sono stati ricavati sia sulla base di ragionamenti sul dominio del problema, e sia sulla base di ripetute simulazioni eseguite con diverse tarature dei pesi. A garantire l'interruzione della ricerca è un opportuno parametro che rappresenta la profondità massima

di discesa nell'albero. In caso di valutazione di un nodo terminale, cioè un nodo che rappresenta la fine della partita, l'euristica è in grado di dedurre dalla configurazione corrente del gioco la vincita, il pareggio o la sconfitta, e restituire di conseguenza al chiamante un opportuno valore.

Per semplificare l'argomentazione delle euristiche utilizzate è fondamentale conoscere la classificazione delle mosse utilizzate:

- **POS_MOVE**, mosse di collisione che portano ad un vantaggio immediato in termini di pedine eliminate tra quelle del giocatore che sta effettuando la mosse e quelle dell'avversario.
- **NEG_MOVE**, ricadono in questa classificazione due tipologie di mosse:
  - mosse di collisione con una pedina dello stesso colore del giocatore che sta effettuando la mossa.
  - mosse di collisione che portano ad uno svantaggio immediato in termini di pedine eliminate tra quelle del giocatore che sta effettuando la mossa e quelle dell'avversario.
- **EQ_MOVE**, mosse che lasciano invariato il vantaggio o lo svantaggio del giocatore che sta effettuando la mossa. Si hanno due classificazioni:
  - mosse senza collisione, che quindi non eliminano nessuna pedina dalla scacchiera.
  - mosse con collisione che portano il giocatore che effettua la mossa ad una perdita di pedine in numero pari a quelle dell'avversario.

6.1 Euristica di primo livello

Questa euristica viene utilizzata per la valutazione dei nodi figli della radice.

L'obiettivo è quello di associare a ciascuno di esso una stima indicativa dell'utilità attesa dello stato per il giocatore e permetterne, successivamente, la suddivisione in *bucket*, utile per la strategia di gioco. Un primo step consiste nella valutazione della mossa: essa corrisponde a una delle possibili scelte che il giocatore può effettuare a partire dalla configurazione corrente del gioco.

Il valore della mossa dipende dal suo tipo:

- in caso di **POS_MOVE** il valore associato alla mossa sarà commisurato al vantaggio immediato ottenuto a seguito della sua esecuzione, ovvero

alla differenza positiva di pedine eliminate tra le proprie e quelle dell'avversario. L'onere di confermare l'effettiva bontà della mossa è posticipato al momento di valutazione dei nodi nei diversi bucket.

- In caso di **EQ_MOVE** il valore associato è calcolato sulla base delle seguenti considerazioni:
  - se non è una mossa di collisione e la posizione finale coincide con uno dei quattro spigoli della griglia di gioco il valore sarà fissato ad un opportuno valore negativo;
  - altrimenti, il valore è la somma pesata di due contributi, ossia
    - un valore v_1 che è tanto più alto quanto più allontano da un intorno denso di pedine appartenenti allo stesso giocatore che sta effettuando la mossa.
    - un valore v_2 che è diverso da zero solo nel caso in cui la mossa non è di collisione, e la pedina spostata si è affiancata ad una pedina dell'avversario. Una pedina si considera affiancata ad un'altra se occupa una cella adiacente all'altra, nelle 8 direzioni cardinali.
- La mossa di tipo **NEG_MOVE**, invece, viene scartata automaticamente al momento di generazione delle mosse disponibili per mezzo dei meccanismi di filtro sulle mosse. Se per caso sfortunato dovesse essere generata (il che avviene solo se al giocatore non rimangono altre mosse), non si fa particolare distinzione.

Dunque, tra le mosse di tipo **EQ_MOVE** vengono preferite le mosse che spostano pedine da intorni densi di pedine dello stesso giocatore, che si affiancano all'avversario e che non vanno ad occupare gli spigoli.

Una volta valutata la mossa, se essa è effettivamente una **POS_MOVE**, il valore associato alla mossa corrisponde al valore associato al nodo e viene restituito il risultato al chiamante. Negli altri casi si procede con un secondo step di valutazione che ha come obiettivo la valutazione della configurazione di gioco, una volta eseguita la mossa, dal punto di vista dell'avversario.

Poiché si presuppone che il giocatore avversario sia razionale, tale valutazione coincide con la valutazione della migliore mossa di tipo **POS_MOVE** che l'avversario ha a disposizione. Se quest'ultima non esiste, la configurazione di gioco a seguito della mossa è considerata accettabile e il valore restituito corrisponderà al valore della mossa calcolato nella fase precedente.

Se invece, tale mossa esiste, il valore finale restituito consisterà nel bilancio negativo di pedine che verranno distrutte in seguito dall'avversario.

6.2 Euristica generale

A differenza della precedente euristica, quest'ultima viene utilizzata per la valutazione di un generico nodo e viene largamente invocata in tutte le ricerche MinMax effettuate. Si basa su una funzione di valutazione che ha la seguente forma:

$$Value(s) = w_1 f_1(s) + w_2 f_2(s) + w_3 f_3(s) + w_4 f_4(s) = \sum_{i=1}^{4} w_i f_i(s)$$

Prima di continuare con la discussione dell'euristica, è necessario specificare due concetti fondamentali nelle valutazioni effettuate:

- ***Configurazione del livello 1***: configurazione del nodo dell'albero Min max, figlio di primo livello della radice, ed antenato del nodo che si vuole valutare.
- ***Configurazione del livello orizzonte***: configurazione del nodo che si vuole valutare.

Il valore finale associato al nodo è la media pesata dei seguenti quattro contributi:

- Valutazione della mossa che ci ha portato in questo stato ($f_1(s)$);
- Profondità della mossa ($f_2(s)$);
- Valutazione della configurazione del livello 1 ($f_3(s)$);
- Valutazione della configurazione del livello orizzonte ($f_4(s)$).

Il valore del contribuito $f_1(s)$ viene calcolato allo stesso modo di come descritto nel paragrafo 6.1.

Il valore $f_2(s)$ tiene conto della profondità del nodo che stiamo valutando ed ha apporto negativo sul valore finale: l'effetto desiderato è preferire la mossa ad un livello più basso a parità di vantaggio derivato.

La valutazione del nodo tiene conto anche della *configurazione del livello 1*. Per comprendere tale scelta consideriamo la valutazione di due nodi n_1 e

n_2, posti ad una stessa profondità, caratterizzati dalla medesima configurazione di gioco (in termini di disposizione delle pedine sulla griglia), ma con antenati diversi.

Un modo per differenziare la valutazione dei due nodi è tenere conto di ciò che succede nell'immediato dopo aver effettuato la mossa "antenata" che li distingue. La valutazione di tale configurazione (il contributo $f_3(s)$) ci permette di comprendere quale potrebbe essere il vantaggio dell'avversario. Il valore è determinato dalla disponibilità per l'avversario di una **POS_MOVE** o, se non esiste, della migliore **EQ_MOVE**. In quest'ultimo caso, una mossa è tanto più svantaggiosa quanto più l'intorno da cui si muove la pedina dell'avversario è circondato da pedine del giocatore.

Il quarto e ultimo parametro ($f_4(s)$) valuta la configurazione di gioco del nodo corrente sulla base della differenza in numero di pedine tra il giocatore e l'avversario.

Capitolo 7

Alcuni confronti e conclusioni

In questo capitolo verrà illustrato il comportamento di 2 giocatori differenti in un mini-torneo.

I due giocatori sono:

- un giocatore con la strategia MinMaxVar, descritta nel capitolo 5;
- un giocatore che fa uso di NegaScout, con strategia denominata AlphaBetaScout.

I risultati delle partite sono illustrati in tabella 7.1,

Giocatore 1	Giocatore 2	Vincitore	MTPM[1] 1	MTPM 2
AlphaBetaScout	AlphaBetaScout	1	117 ms	128 ms
AlphaBetaScout	MinMaxVar	2	117 ms	174 ms
MinMaxVar	AlphaBetaScout	1	325 ms	165 ms
MinMaxVar	MinMaxVar	Pareggio	853 ms (!)[2]	852 ms (!)

Tabella 7.1: Risultati partite simulate

Come è possibile notare, in caso di identici giocatori, MinMaxVar porta a timeout, ossia porta lo stato del gioco verso configurazioni con un elevato branching factor, a differenza di Scout.

Tuttavia, per via di come vengono gestiti gli stati di attacco e difesa, oltre che le mosse 'positive', il giocatore MinMaxVar risulta più aggressivo e potenzialmente più efficace.

[1] Max Time Per Move, massimo tempo per mossa

[2] Indica timeout, siccome il massimo rischio pronti a correre era di 150 ms sul totale di 1000 ms

Bibliografia

Chaslot, Guillaume et al. (22 ott. 2008). "Monte-carlo tree search: a new framework for game AI". In: *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. AIIDE'08. Stanford, California: AAAI Press, pp. 216–217. (Visitato il 19/02/2022).

Games of Soldiers - FISSION (2022). URL: `https://www.di.fc.ul.pt/~jpn/gv/fission.htm` (visitato il 17/02/2022).

Pearl, Judea (18 ago. 1980). "Scout: a simple game-searching algorithm with proven optimal properties". In: *Proceedings of the First AAAI Conference on Artificial Intelligence*. AAAI'80. Stanford, California: AAAI Press, pp. 143–145. (Visitato il 19/02/2022).

Reinefeld, A. (1983). "An Improvement to the Scout Tree Search Algorithm". In: *J. Int. Comput. Games Assoc.* DOI: `10.3233/ICG-1983-6402`.

Silver, David et al. (5 dic. 2017). "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *arXiv:1712.01815 [cs]*. arXiv: `1712.01815`. URL: `http://arxiv.org/abs/1712.01815` (visitato il 19/02/2022).