IMPLEMENTAZIONE DI UN GIOCATORE AUTOMATICO PER IL GIOCO UNSTABLE BASATO SULL'ALGORITMO DI RICERCA NEGAMAX

Introduzione

Il progetto svolto prevede la realizzazione di un giocatore automatico per il gioco Unstable (anche noto come Chain Reaction) utilizzando gli strumenti dell'intelligenza artificiale. Esistono piú varianti del gioco in questione, ma noi ci interesseremo di quella che prevede due soli giocatori.

Progettazione

Nella prima fase del progetto é stato studiato il gioco cercandone caratteristiche utili nelle scelte degli algoritmi da utilizzare. Il gioco rientra nella categoria di quelli a due giocatori con informazione perfetta. Si é scelto di strutturare le configurazioni possibili della board come nodi di un albero dove si alternano, per livelli, le configurazioni dei due giocatori. Il numero di nodi massimo di un albero b-ario (b = "branching factor", d = "depth") é pari a:

$$N = \sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1}$$

che nel nostro caso diventa, con una sovrastima, (b = 60, d = 148):

$$N = \sum_{i=0}^{148} 60^i \cong 1.4917e263$$

Dato l'elevato numero di nodi, risulta ovvia l'impossibilitá di esplorare l'albero nella sua interezza. Si é dunque optato per l'adozione dell'algoritmo di ricerca **Iterative Deepening**, che sfrutta la visita in profonditá iterando su un numero di livelli sempre crescente, alla ricerca della configurazione vincente, o goal. Il solo uso di questo algoritmo non é sufficiente ad effettuare una buona ricerca nell'albero, poiché come appena visto il numero di nodi é troppo elevato, andando a limitare il numero di livelli esplorati. Una soluzione é quella di applicare algoritmi di ricerca che approssimano, grazie ad una euristica della board, la migliore mossa da effettuare per arrivare ad un nodo vantaggioso. Si é studiato ed applicato l'algoritmo **Minimax** con **Alpha-Beta Pruning**. Successivamente si é scelto di utilizzare una sua variante: **Negamax**. Un'altra caratteristica del nostro giocatore automatico prevede la possibilitá di scegliere una strategia da adottare in base alle mosse avversarie. Questo permette alla nostra IA di adattarsi all'avversario, studiandone poche mosse iniziali.

Implementazione

Classi

Board

Contiene tutte le informazioni utili a descrivere la configurazione attuale. Si é scelto di rappresentare la board come array bidimensionale di interi. Ogni cella puó contenere valori nel range [-3;+3]. Le celle vuote sono rappresentate da valori 0 mentre le altre hanno valori positivi per un player e negativi per l'altro. Altre strutture dati usate per memorizzare informazioni di gioco sono *ownedCells* e *unstableCells* che tengono traccia, rispettivamente, del numero di celle possedute e di quelle instabili per ognuno dei due giocatori. Board contiene inoltre il metodo *play* che aggiorna la configurazione attuale e le varie strutture di supporto a seguito di una mossa di uno dei due giocatori, ed il metodo *heuristic()* che calcola il valore euristico della configurazione.

Configuration

Possiede al suo interno un'istanza di Board, con rispettivo valore euristico, e la mossa che l'ha generata. La classe implementa l'interfaccia Comparable poiché é necessario stabilire un ordinamento in base al valore euristico.

Move

Rappresenta semplicemente una mossa di un giocatore. Memorizza le coordinate della cella che subisce la mossa ed il giocatore che l'ha giocata.

Utility

Racchiude al suo interno tutte le costanti del gioco, le strutture ed i metodi di supporto. Per ridurre la complessitá spaziale e temporale, é stata fatto largo uso di strutture statiche.

COUNT_NEARS è un array bidimensionale che, per ogni cella della board, contiene il numero di celle vicine. Analogamente UNSTABILITY contiene il numero di atomi necessari a rendere instabile la cella. NEARS é invece un array tridimensionale nel quale per ogni cella vengono salvate le celle vicine in un array. Da precisare che in questa struttura le celle vengono identificate da un solo valore che va da 0 a 59.

MOVES contiene tutte le mosse possibili di entrambi i giocatori su ognuna delle celle della board. Cosi facendo nessuna nuova mossa verrá creata o distrutta durante il gioco ma tutte le mosse eseguibili saranno reperite da questa struttura. Questo meccanismo garantisce un notevole risparmio di risorse computazionali, sia spaziali che temporali.

Un altro metodo di utilità è otherPlayer() che, ricevendo un intero corrispondente ad un giocatore, ne restituisce l'avversario. É stato fatto uso delle funzioni che lavorano direttamente sui bit di Java 8 per aumentarne l'efficienza.

```
public static int otherPlayer(int player) {
        return ~player & 3;
}
```

Unstable

Unstable è una vera e propria istanza del gioco e possiede la board corrente che provvede a tenere aggiornata dopo ogni mossa.

calculateInitialMove() oltre che restituire le mosse nella fase iniziale di gioco, permette di decidere la strategia che verrà successivamente adottata per la generazione delle mosse. L'idea di questo

metodo è quella di portare il gioco ad una configurazione vantaggiosa distinguendo i casi in cui si gioca come black e come white.

iterativeDeepening() implementa l'algoritmo di ricerca richiamando, per ogni mossa disponibile, il metodo *negaMax()* passando una copia della board corrente. Cosí facendo é possibile modificare la copia della board poiché non in aliasing con la board originale, evitando di dover progettare un metodo che annulli la mossa appena eseguita. I valori delle varie Configuration vengono infine ordinati prima di eseguire la prossima iterazione con profondità maggiore. Questo permette di esplorare in anticipo le configurazioni piu promettenti del passo precedente. Inoltre questo metodo é capace di restituire la mossa che porta al goal ottimo (configurazione vincente piú vicina al nodo radice) nel caso in cui vengano individuati piú di un goal.

Il metodo negaMax() avrà come valore di ritorno un intero corrispondente al valore del nodo appena analizzato così da non dover tenere traccia delle mosse che hanno portato a quella data configurazione. Sarà invece compito del metodo *iterativeDeepening*() quello di mantenere il vettore delle mosse disponibili come oggetto *Move* e restituire la migliore corrente, facendo sì che si abbia sempre una mossa valida da inviare al server.

Euristica

Per valutare la bontá di una generica configurazione della board, si é scelto di implementare un'euristica non onerosa in termini di prestazioni, poiché questa operazione é una delle piú frequenti nella fase di ricerca all'interno dell'albero. L'euristica ideata consiste nella differenza tra le celle possedute sommata alla differenza tra quelle instabili dei i due giocatori. Avendo previsto delle strutture dati specifiche a questo scopo, il metodo non fa altro che eseguire, semplicemente, una somma algebrica, poiché questi valori sono mantenuti aggiornati dinamicamente:

```
int heuristic(int player) {
	int otherPlayer = Utility.otherPlayer(player);
	if (ownedCells[otherPlayer] == 0) {
		return Utility.MAX_VALUE;
	}
	return ownedCells[player] - ownedCells[otherPlayer] +
unstableCells[player] - unstableCells[otherPlayer];
```

Se la configurazione da valutare é terminale, ovvero il player che ha eseguito la mossa ha catturato tutte le celle avversario, la funzione euristica restituira il valore massimo Utility.MAX_VALUE. Questo é indispensabile per individuare la mossa vincente ed evitare quindi che la ricerca continui ulteriormente.

Strategy

Come accennato in precedenza il giocatore ha bisogno di una strategia con la quale generare le mosse. Si è quindi fatto uso del ***pattern design Strategy*** attraverso il quale è stato possibile implementare un meccanismo che sceglie ed utilizza dinamicamente una strategia tra le varianti disponibili. Si é arrivati a considerare di fondamentale importanza l'uso delle strategie poiché si é visto che, cambiando punto iniziale a direzione in cui giocare le mosse, puó portare ad un notevole vantaggio.

Il giocatore analizza le mosse iniziali dell'avversario in modo da comprenderne la strategia e scegliere la direzione di gioco migliore per la partita. Una strategia si differenzia dalle altre per l'ordine con il quale vengono generate le mosse possibili a partire dalla configurazione corrente. Le strategie ideate sono in totale otto e coprono tutte le direzioni.

Scelte Progettuali

Gestione del tempo di mossa

Affinché venga restituita una mossa entro il tempo limite di risposta si è usato il meccanismo della TimeOutException che si adatta naturalmente a questo scopo. Viene sollevata un'eccezione dal metodo negaMax(), la quale verrà propagata tra le varie chiamate ricorsive fino a giungere al metodo iterativeDeepening, che la cattura e restituisce la mossa migliore calcolata fino a quel momento. É risultato ovvio, per il calcolo della mossa, sfruttare, oltre ai tre secondi concessi, anche il tempo che intercorre tra il momento in cui si riceve dal server la mossa avversaria e quello in cui si riceve il messaggio YOUR_TURN. Dopo vari test si é impostato come MAX_TIME il valore 4990.

Algoritmi di ricerca

Dopo aver implementato una prima versione dell'algoritmo MiniMax con Alpha-Beta pruning, ne é stata implementata una seconda versione utilizzando l'algoritmo NegaMax. Questo algoritmo é una versione migliorata del MiniMax, applicabile solo nei giochi a somma zero. É stata dunque rivista l'euristica in modo tale da soddisfare questa caratteristica. Altri algoritmi come il NegaScout ed il NegaMaxPV (Principal Variation) non si sono dimostrati utili al nostro scopo poiché risultati meno efficienti e dunque non sono stati oggetto di ulteriori analisi.

Hashing

Inizialmente era stata ipotizzata la possibilitá di salvare, per un certo numero di configurazioni, i corrispondenti valori euristici all'interno di un file. Si é cercato dunque di trovare una funzione di hash per la configurazione che fosse meno onerosa della funzione euristica stessa e che evitasse collisioni. Per distinguere una configurazione da un'altra sono necessari, con una stima pessimistica, $3 * 60bit = 180bit = 23byte$. Il numero 3 é dovuto al fatto che in ognuna delle 60 celle si puó trovare un numero compreso tra -3 e 3. Dunque i valori per ogni cella sono 7, rappresentabili con 3 bit. Per una serie di motivi si é scartata questa idea:

- Non si é riusciti a trovare per il calcolo dell'hash un modo piú efficiente rispetto al calcolo dell'euristica stessa, rendendo inutile, se non addirittura scoveniente, il suo utilizzo;
- L'hash generato é di dimensioni non trascurabili, andando ad impattare sulla dimensione del file che avrebbe dovuto memorizzare le coppie <hash_board, heuristic_value>;
- Il numero di configurazioni é cosí elevato che anche solo calcolarne l'1% equivale a calcolarne 1.4914e261.

Si é provato ad implementare la funzione di hash con l'algoritmo di Zobrist, ma si é notato che questo algoritmo non si presta bene a questo gioco. Infatti l'hash di Zobrist é stato pensato sui giochi da scacchiera dove una mossa genera pochi cambiamenti.

Speculazione

Una funzione che era stata implementata, ma poi rimossa perché risultata inefficace, era quella della speculazione sulla mossa avversaria. Si é cercato appunto di prevedere la prossima mossa avversaria basandoci sull'algoritmo di ricerca e sull'euristica. L'idea era quella di memorizzare la migliore mossa avversaria, secondo la nostra euristica, successiva alla giocata della nostra mossa migliore restituita dall'algoritmo Negamax. In caso di successo, questa funzione avrebbe permesso di continuare ad eseguire la ricerca raggiungendo livelli piú bassi nell'albero delle configurazioni. Questa funzione é risultata inefficace poiché dopo vari test ci si é accorti che la percentuale di successo sulla predizione é molto bassa.