Implemenazione del gioco del Bantumi

Corso Intelligenza Artificiale 2012

Nicola Febbrari
Università degli studi di Verona
Facoltà MM.FF.NN.
nicola.febbrari@studenti.univr.it

12 ottobre 2012

1 Introduzione

Con questo progetto vorrei mettere in pratica alcune tecniche che sono state spiegate durante il corso di intelligenza artificiale, pertanto ho deciso di realizzare il gioco del Bantumi implementando un algoritmo di intelligenza artificiale per poi confrontarlo con altri programmi analoghi.

2 Il Gioco

Il Bantumi è un antico gioco da tavolo di origine africana, composto da una tavola in legno con 12 buche disposte su due file parallele e due buche più grandi, 1 per giocatore, dette granai. Si gioca in due, in partenza ogni giocatore possiede 18 semi che dispone uniformemente nelle proprie buche (3 in ogni buca), lasciando i 2 granai vuoti. A turno, ogni giocatore può effettuare una mossa che consiste nello scegliere una propria buca non vuota, prendere tutti i semi che contiene e distribuirli uno in ogni buca successiva in senso antiorario utilizzando anche il granaio e le buche dell' avversario (granaio escluso).

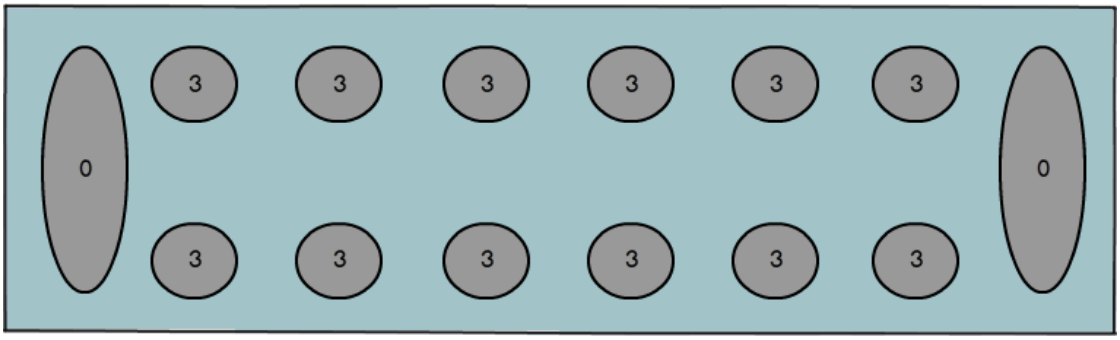

Figura 2.1: *Tavola.*

3 Regole particolari

1. Un giocatore non può mai depositare semi nel granaio dell'avversario
2. Se l'ultimo seme, tra quelli da distribuire del proprio turno, viene depositato nel granaio il giocatore ha diritto ad un ulteriore turno.

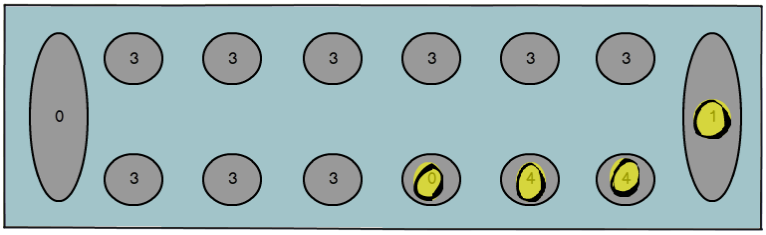

Figura 3.1: *Turno multipli.*

3. Se l'ultimo seme viene depositato in una propria buca senza semi, il giocatore prenderà questo seme e tutti i semi della buca di fronte dell'avversario e li metterà nel proprio granaio.

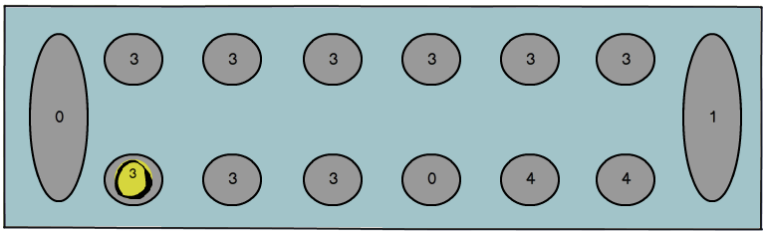

Figura 3.2: *Raccolta.*

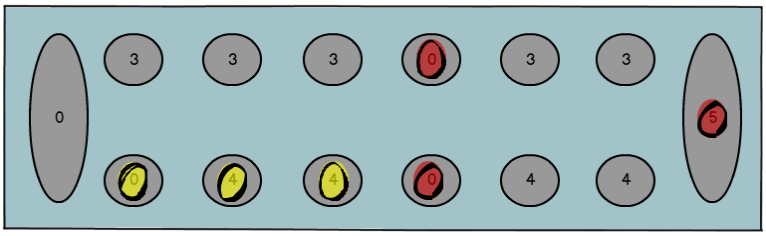

Figura 3.3: *Cattura.*

4. Il gioco termina quando, durante il turno di un giocatore, tutte le proprie buche sono vuote. In questo momento il giocatore con più semi nel granaio è il vincitore.

4 Implementazione del Gioco

4.1 I progetti.

Ho strutturato il codice dividendolo in 4 progetti.

- *Bantumi.entities.interface*: il progetto in cui sono contenute tutte le interfacce delle entità
- *Bantumi.entities*: l'implementazione di tutte le classi
- *Test*: il progetto di unit test per il testo in automatico durante il processo di TDD
- *UI Console*: il progetto per user interface

4.2 Le entità

Le principali entità per la realizzazione del gioco sono

- La classe *Tavola*, che descrive la tavola fisica, le buche e i semi contenuti
- Le classi *Player*, che descrivono le varie implementazioni dei giocatori
- La classe *Gioco*, che implementa le regole e i vincoli del Bantumi
- La classe *Bantumi*, che si occupa del set-up

5 Implementazione dell'algoritmo di IA

5.1 Implementazione MinMaxPlayer

Per implementare un giocatore con intelligenza artificiale ho utilizzato uno degli algoritmi per la *ricerca con avversari* visti a lezione, l'algoritmo *MinMax*.

Definizioni

- L'*albero* rappresenta la combinezione di tutte le possibili mosse eseguibili partendo da una situazione consolidata della tavola di partenza.
- Un *nodo* dell'albero rappresenta una possibile mossa fatta dal giocatore di turno.
- Il *livello* dell'albero rappresenta tutte lo possobili mosse (o sequenze di mosse nel caso di turno multiplo) che può eseguire il giocatore di turno

Algoritmo Ad ogni turno l'algoritmo decide quale sia la mossa migliore. Partendo da una situazione consolidata della Tavola, vengono elaborati i valori di ogni mossa possibile usando un calcolo ricorsivo sugli stati successivi generati da tutte le possibili combinazioni di mosse valide.
La ricorsione percorre l'albero fino alle foglie, valuta il valore della foglia assegnandogli un +1 se la foglia rappresenta uno stato di vittoria, 0 se di pareggio o -1 in caso di sconfitta. Dopo aver calcolato il valore di una foglia viene fatto *backtracking* sull'albero di gioco eventulmente aggiornando la scelta del nodo ottimo *MinMax* in base al livello dell'albero (nel caso di turno proprio turno Max nel caso di turtno dell'avversario Min).

Complessità Per eseguire la ricerca viene fatta una vista in profondità dell'albero di gioco. Nel Bantumi la dimensione dell'abero è risulta ingestibile, infatti se per ogni turno ci possono essere m mosse e una partita dura t turni, nello stato iniziale scorrere tutto l'albero ha una complessità in tempo di $O(m^t)$.

5.2 Alfa Beta pruning

La dimensione dell'albero di gioco del Bantumi nelle prime mosse è troppo grande e rendere impossibile l'esplorazione, questo è dovuto proprio all'esponenzialità rispetto al numero di turni. Ovviamente non è possibile ridurre il problema ad una complessità trattabile (polinomiale), tuttavia utilizzando un meccanismo di potatura possiamo evitare di valutare porzioni di albero che sicuramente non verranno prese in considerazione durante la fase di decisione della mossa migliore.
L'idea è quella di valutare ad ogni nodo, dato uno determinato stato possibile della tavola, se esiste uno stato predecessore che risulta essere migliore. Se questo fosse verificato ovviamente renderebbe inutile la valutazione di questa porzione di albero.

Listing 1: Implementazione Max con taglio

```
if (!gioco.Finito)
{
    if (!cambioTurno)
        valoreProssimaMossa = MaxValue(gioco, depth, alfa, beta);
    else
        valoreProssimaMossa = MinValue(gioco, depth + 1, alfa, int.
            MaxValue);
}
ValoreMossa valore = new ValoreMossa(valoreProssimaMossa, i, ValoreAttuale
    (gioco));
if (null == resultMove)
  resultMove = valore;
else
  resultMove = AggiornoValore(resultMove, valore, latoStep);
if (resultMove.Valore > beta)
    return resultMove;
alfa = resultMove.Valore > alfa ? resultMove.Valore : alfa;
```

Un ruolo fondamentale per la qualità dei tagli eseguiti durante l'algoritmo è dato dalla funzione euristica. Scegliendo una buona funzione è possibile dare dei valori alti ad alfa beta durante i primi cammini, così da riuscire a tagliare grosse porzioni di albero e risparmiare cicli di comutazione. Nel caso del Bantumi ho implementato questa funzione basandomi sul risultato della differenza fra il valore dei due granai calcolato per ogni mossa possibile. Le mosse con una differenza maggiore vengono valutate prima.

5.3 Limitazione della profondità

La potatura alfa beta non è stata sufficente per ridurre l'elaborazione a tempi accettabili, quindi ho deciso di introdurre nell'algoritmo un parametro che limitasse la dimensione dell'albero. Ho deciso di parametrizzare quante mani espandere l'albero (una mano comprende i 2 turni dei giocatori).

5.4 Studio dell' apertura

Un' ultimo perfezionamento che ho fatto è stato quello di studiare le mosse di apertura. Ho fatto giocare contro se stesso il mio Bot impostando livelli di esplorazione elevati che hanno richiesto molte ore di computazione. Una volta analizzate le prime due mosse sono riuscito a aumentare notevolmente la capacità di vittoria anche utilizzando Bot con livelli di esplorazione bassi.

5.5 Conteggio operazioni

Completate tutte le implemetnazioni ho cercato di fare delle valutzioni empiriche cercando di confrontare gli algoritmi e le varie euristiche che ho implementato per capire se effettivamente i miglioramenti teorici che avevo ipotizzato hanno avuto degli effetti. Per la valutazione ho eseguito un conteggio del numero dei nodi esaminati partendo da una situazione neutra rappresentata dalla tavola iniziale. Come prima cosa ho eseguito il test sull'algoritmo MinMax variando la limitazione del numero di turni.

Algoritmo	Senza Hueristic	Heuristc Desc[1]	Heuristc Val[2]
MinMax(1)	129		
MinMax(2)	8727		
MinMax(3)	882215		
MinMax(4)	99569593		
MinMaxAlfaBeta(1)	112	83	97
MinMaxAlfaBeta(2)	3057	2209	2405
MinMaxAlfaBeta(3)	84577	56770	50387
MinMaxAlfaBeta(4)	2676841	1320574	1148562
MinMaxAlfaBeta(5)	81995855	25298451	23123673

[2]I nodi vengono scelti per la valutazione in modo decrescente

[3]I nodi vengono scelti con una funzione euristica che valuta il valore in base alla differenza fra i due granai piú 0,5 se la scelta da diritto ad una seconda mossa

5.6 Osservazioni

Confrontando i risultati ottenuti con le implementazioni realizzate si può immediatamente osservare il grande vantaggio computazionale dato dall'introduzione dell'alfabeta pruning. Con 4 mani di gioco (8 turni) senza nessuna potatura i nodi valutati sono quasi 100M, mentre a parità di profondita ma utilizzando una funzione euristica di ordinamento nella scelta della potatura i nodi analizzati sono poco piu di 1M. Questo risultato così evidente mi fa ipotizzare che con uno studio approfondito delle tecniche di gioco sia possibile implementare una funzione euristica migliore che ottimizzi i tagli per poter estendere ulteriormente la profondità dell'albero.

6 Bot a confronto

Per una valutazione finale dell'implementazione ho confrontato l'algoritmo con alcuni giochi del Bantumi presenti in commercio. Questi sono i risultati:

Player1	Player2	Vincitore
MinMaxAlphaBetaWithOpen(1)	kalahandroid 1.2	kalahandroid
MinMaxAlphaBetaWithOpen(2)	kalahandroid 1.2	kalahandroid
MinMaxAlphaBetaWithOpen(3)	kalahandroid 1.2	MinMax
kalahandroid 1.2	MinMaxAlphaBetaWithOpen(3)	PARI
kalahandroid 1.2	MinMaxAlphaBetaWithOpen(4)	MinMax
MinMaxAlphaBetaWithOpen(3)	Mancala-Gold	MinMax
MinMaxAlphaBetaWithOpen(3)	Bantumi Atomic Pineapple	MinMax
MinMaxAlphaBetaWithOpen(3)	Bantumi FREE Paul Production	MinMax
MinMaxAlphaBetaWithOpen(3)	Nicola	MinMax
Nicola	MinMaxAlphaBetaWithOpen(3)	MinMax
Nicola	MinMaxAlphaBetaWithOpen(1)	MinMax

7 Risultati e sviluppi futuri

Il progetto è stato strutturato cercando di mantenere un forte indipendenza fra l'infrastruttura del gioco e l'implementazione dei vari algoritmi di IA. Questa scelta mi ha permesso di procedere con un miglioramento graduale degli algoritmi scritti, di sperimentare diverse strategie, ma soprattutto di poter consolidare in modo semplice e immediato alcuni concetti teorici spiegati durante il corso.
Il progetto potrebbe essere completato con la realizzazione di una UI web/mobile, una possibile evlouzione del core, invece, potrebbe essere quella di sperimentare qualche tecnica di calcolo distribuito per sfruttare la potenza computazionale che mette a disposizione una architettura di cloud computing.

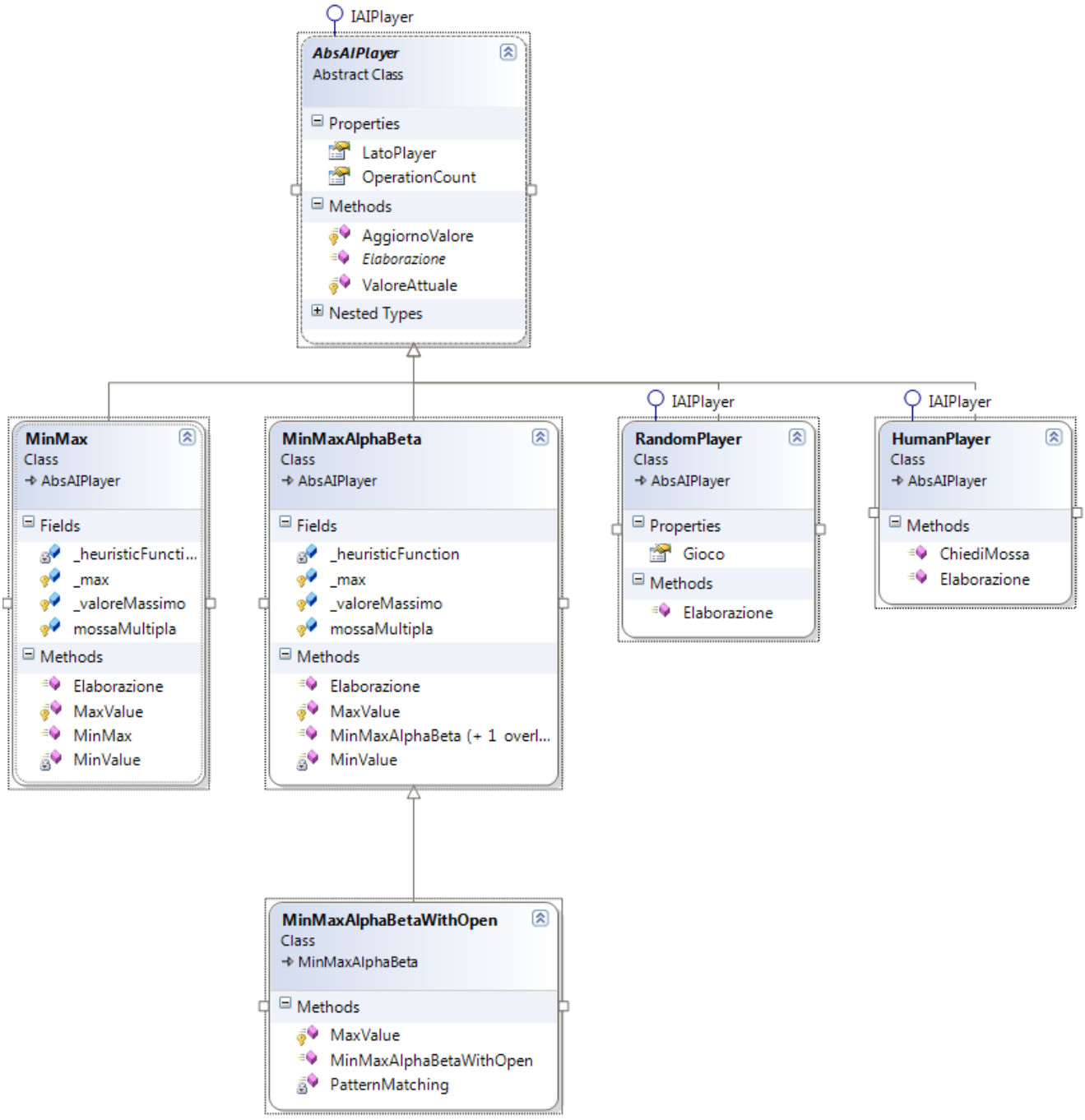

Figura 5.1: *Diagramma delle classi dei Players implementati.*

```
protected virtual ValoreMossa MaxValue(IGioco gioco, int depth, int alfa, int beta)
{
    ValoreMossa resultMove = null;
    if (depth <= _max && !gioco.Finito)
    {
        ITavola tavola = gioco.Tavola;
        Lato latoStep = gioco.ProssimoTurno;
        int[] heuristc = _heuristicFunction.HeuristicFunction(gioco.Clone());
        for (int j = 0; j < heuristc.Length; ++j)
        {
            int i = heuristc[j];
            OperationCount++;
            IGioco giocoClone = gioco.Clone();
            giocoClone.Muovi(i);
            bool cambioTurno = gioco.ProssimoTurno != giocoClone.ProssimoTurno;
            ValoreMossa valoreProssimaMossa = null;
            if (!giocoClone.Finito)
            {
                if (!cambioTurno)
                    valoreProssimaMossa = MaxValue(giocoClone, depth, alfa, beta);
                else
                    valoreProssimaMossa = MinValue(giocoClone, depth + 1, alfa, 1000);
            }
            ValoreMossa valore = new ValoreMossa(valoreProssimaMossa, i, ValoreAttuale(giocoClone));
            if (null == resultMove)//inizializzazione
            {
                resultMove = valore;
            }
            else
            {
                resultMove = AggiornoValore(resultMove, valore, latoStep);
            }
            if (resultMove.Valore > beta)
                return resultMove;
            alfa = resultMove.Valore > alfa ? resultMove.Valore : alfa;
        }
    }
    return resultMove;
}
private ValoreMossa MinValue(IGioco gioco, int depth, int alfa, int beta)
{
    ValoreMossa resultMove = null;
    if (depth <= _max && !gioco.Finito)
    {
        ITavola tavola = gioco.Tavola;
        Lato latoStep = gioco.ProssimoTurno;
        int[] heuristc = _heuristicFunction.HeuristicFunction(gioco.Clone());
        for (int j = 0; j < heuristc.Length; ++j)
        {
            int i = heuristc[j];
            OperationCount++;
            IGioco giocoClone = gioco.Clone();
            giocoClone.Muovi(i);
            bool cambioTurno = gioco.ProssimoTurno != giocoClone.ProssimoTurno;
            ValoreMossa valoreProssimaMossa = null;
            if (!giocoClone.Finito)
            {
                if (!cambioTurno)
                    valoreProssimaMossa = MinValue(giocoClone, depth, alfa, beta);
                else
                    valoreProssimaMossa = MaxValue(giocoClone, depth + 1, -1000, beta);
            }
            ValoreMossa valore = new ValoreMossa(valoreProssimaMossa, i, ValoreAttuale(giocoClone));
            if (null == resultMove)//inizializzazione
            {
                resultMove = valore;
            }
```

Figura 5.2: *Metodi che eseguono la ricorsione MinMax.*