Progetto di Esperienze di Programmazione

“Backgammon”

Filippo Fontanelli

Filippo Fontanelli Matricola: 422385; Laurea in Informatica - Classe 26; E-mail: fontanelli.filippo@gmail.com

Indice

Introduzione

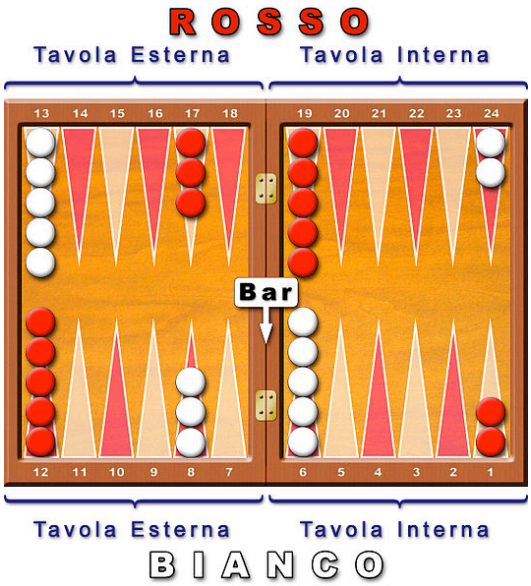
Lo scopo del progetto è lo sviluppo del gioco del Backgammon Turco “Tavla” utilizzando come linguaggio di programmazione Java.

In particolare, l’obiettivo è di implementare l’algoritmo MinMax con Chance.

Descrizione del gioco

Introduzione al gioco

Il Backgammon è un gioco per due persone. Si utilizza una tavola su cui sono disegnati ventiquattro triangoli, chiamati punte (points). I triangoli, a colori alterni, sono raggruppati in quattro quadranti, composti da sei punte ognuno. I quadranti sono chiamati tavola interna o casa (home board) e tavola esterna (outer board) del giocatore e del suo avversario. Le tavole interne e quelle esterne sono separate, al centro della tavola, da una striscia chiamata bar.



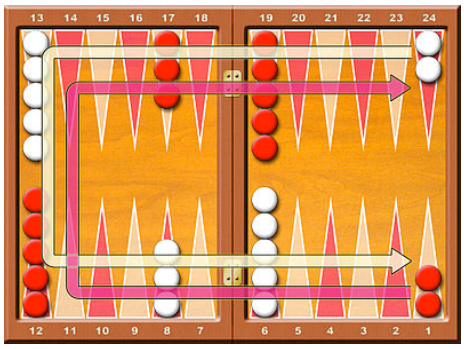
Una tavola da backgammon con le pedine nella loro disposizione iniziale. È indicata la numerazione delle punte per il solo giocatore Bianco.

Per ciascun giocatore le punte sono numerate a partire dalla propria tavola interna. Il punto più lontano per ogni giocatore è quindi il ventiquattro, che equivale alla punta uno dell'avversario. Ciascun giocatore possiede 15 pedine di un proprio colore. Il posizionamento iniziale delle pedine è: 2 pedine su ogni punto ventiquattro, 5 su ogni punto tredici, 3 su ogni punto otto e 5 su ogni punto sei.

Entrambi i giocatori possiedono una coppia di dadi e per tirarli utilizzano un bussolotto.

Lo scopo del gioco

Lo scopo del gioco, per ogni giocatore, è quello di portare tutte le proprie pedine nella propria casa e successivamente di portarle fuori (bear off). Il primo giocatore che le ha portate fuori tutte vince la partita.



Senso di marcia.

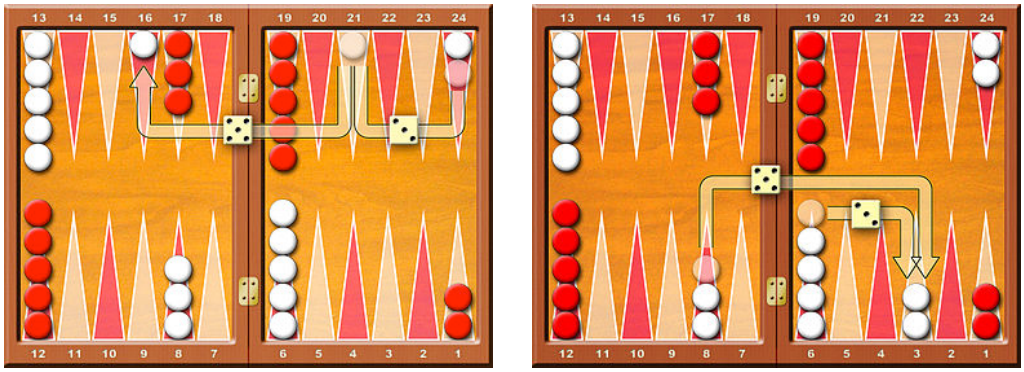
Il movimento delle pedine

Per iniziare la partita, ogni giocatore lancia un singolo dado. Se entrambi i giocatori hanno ottenuto lo stesso numero, allora i dadi devono essere tirati nuovamente fino a quando i numeri ottenuti saranno diversi. Dopo questo lancio di partenza, i giocatori tireranno la propria coppia di dadi a turni alterni. Per evitare contestazioni, un lancio è considerato valido solo se entrambi i dadi si fermano in piano sulla tavola interna.

Chi ha ottenuto il numero più alto, parte per primo ed i numeri da utilizzare per la prima mossa sono proprio quelli che si ottengono con un nuovo lancio di entrambi i due dadi.

I numeri ottenuti con i dadi indicano di quanti punti (pips), il giocatore deve muovere le proprie pedine. Le pedine devono sempre essere mosse in avanti, verso le punte corrispondenti a numeri più bassi. Le regole per il movimento delle pedine sono le seguenti:

- Una pedina può essere mossa solo su una punta aperta, ovvero una punta che non sia occupata da due o più pedine avversarie (che è detta anche "casa").
- I numeri sui due dadi costituiscono due mosse separate. Per esempio, se un giocatore ha ottenuto 5 e 3, può decidere se muovere una pedina di cinque spazi su una punta aperta ed un'altra pedina di tre spazi sempre su una punta aperta, oppure se muovere un'unica pedina di otto spazi su una punta aperta, ma solo se almeno uno dei punti intermedi (a distanza tre o cinque dal punto di partenza) è aperto.



Esempio di spostamento singolo o cumulativo.

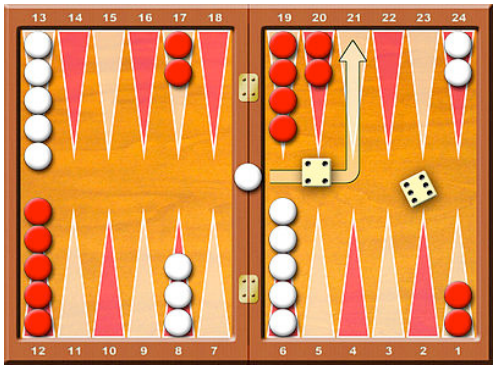
- Un giocatore che ottiene due numeri uguali, muove il doppio delle mosse. Per esempio, un giocatore che ottiene 6 e 6, ha quattro mosse da sei spazi a disposizione, che può completare con la combinazione di pedine che lui ritiene più appropriata.
- Un giocatore deve utilizzare entrambi i numeri che ottiene (o quattro se ha ottenuto un doppio), ammesso che questo sia possibile. Quando può essere giocato solo un numero, tale numero deve essere giocato. Se entrambi i numeri sono possibili, ma l'utilizzazione di uno di essi non rende più possibile l'utilizzazione del secondo, allora il giocatore deve muovere il numero più grande. Se nessun numero può essere utilizzato, il giocatore perde il proprio turno. Nel caso di numeri doppi, quando non è possibile usarli tutti e quattro, devono essere giocati più numeri possibili.

Mangiare e Rientrare

Una singola pedina, qualsiasi sia il colore, che occupi da sola un punto è detta "scoperta" (blot). Se una pedina finisce su una pedina scoperta dell'avversario, quest'ultima viene mangiata o colpita (hit) e posta sul bar.

Ogni volta che un giocatore ha una o più pedine sul bar, è obbligato, come prima mossa, a far rientrare tutte le proprie pedine nella tavola interna dell'avversario. Ciò consiste nel prelevarla dal bar e porla su una punta aperta, il cui numero corrisponde ad uno dei due dadi lanciati.

Ad esempio, se un giocatore ottiene 4 e 6, può far rientrare una pedina sul punto 4 o sul 6 dell'avversario, a meno che uno o entrambi i punti siano già occupati da due o più pedine avversarie.



Esempio di rientro.

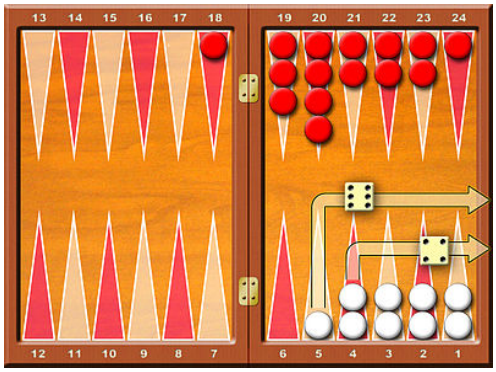
Se nessun punto è aperto, il giocatore perde il suo turno. Se un giocatore può sfruttare solo parzialmente i numeri che ha ottenuto per il rientro, è obbligato a far rientrare più pedine possibili e deve rinunciare alle mosse rimanenti.

Dopo che l'ultima pedina è rientrata dal bar, il giocatore deve usare gli eventuali numeri non utilizzati dei dadi muovendo quella stessa pedina oppure una qualsiasi altra.

Portare fuori le pedine

Una volta che un giocatore ha portato tutte le sue 15 pedine sulla propria casa, può iniziare a portarle fuori dalla tavola (bear off). Un giocatore porta fuori una pedina tirando un numero corrispondente al punto in cui si trova la pedina e rimuovendo tale pedina dalla tavola. Quindi, tirando un 6, il giocatore può portare fuori una pedina che si trova sulla punta sei.

Se non ci sono pedine su uno dei punti indicati dai dadi, il giocatore deve muovere in maniera legale una pedina che si trova su un punto corrispondente ad un numero più alto. Se non ci sono pedine neppure su un punto più alto, allora il giocatore deve rimuovere una pedina dal punto più alto che è ancora occupato da una qualche sua rimanente pedina. Il giocatore non è però obbligato a portare fuori una pedina nel caso ci siano altre mosse consentite.



Esempio di uscita.

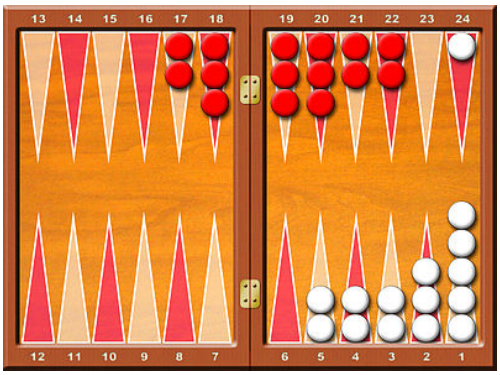
Per far uscire una qualsiasi pedina, un giocatore deve avere tutte le sue pedine ancora in gioco sulla sua tavola interna. Se una pedina viene mangiata mentre è in corso il processo per portarle fuori, prima di riprendere tale processo, la pedina mangiata deve essere riportata sulla propria tavola interna. Il primo giocatore che porta fuori tutte e 15 le pedine ha vinto la partita.

Informazioni aggiuntive

Movimento delle pedine

1.Che cosa é un blocco?

Un blocco, o sbarramento (prime) è una sequenza di 6 punti consecutivi ognuno dei quali occupato da almeno due pedine di uno stesso colore (cioè da 6 "case" consecutive). Quando si riesce a costruire un blocco, nessuna pedina avversaria potrà scavalcarlo fino a quando resta intatto. Infatti una pedina non può fermarsi su un punto bloccato e non può nemmeno muovere più di sei punti in un solo balzo.

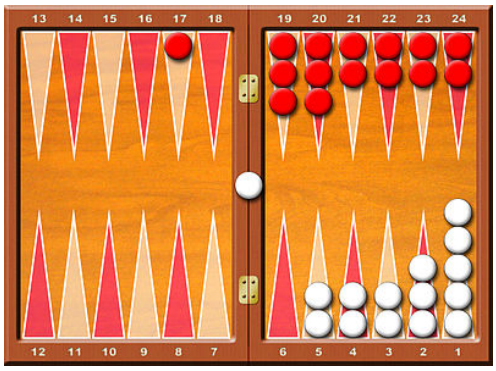


Esempio di blocco delle pedine.

Nel seguente esempio, il Rosso ha costruito un blocco ed intrappolato la pedina più lontana del Bianco rimasta sul ventiquattro. Il blocco è realizzato con una sequenza di 6 punte occupate da almeno due delle sue pedine (perciò 6 "case" consecutive).

2.Cosa significa “esser chiusi”?

Quando un giocatore ha costruito un blocco contenuto completamente sulla sua tavola interna ed ha mangiato uno o più pedine scoperte del suo avversario, quest'ultimo "è chiuso" (closed out). Le sue pedine mangiate rimarranno sul bar, quindi non potrà muovere fino a quando il suo avversario non aprirà uno dei punti della propria tavola interna.



Un esempio di posizione in cui il Bianco è stato chiuso. Il Bianco ha una pedina sul bar e non ci sono punti aperti sulla tavola interna del Rosso.

(1) Posso passare il turno se non mi piacciono i numeri che ho ottenuto con i dadi?

Se ci sono mosse legalmente consentite, è obbligatorio giocare i numeri ottenuti. Non esiste nel backgammon la possibilità di passare il turno oppure di ritirare i dadi.

(2) C'è un limite al numero di pedine che possono stare su un punto?

Non esiste un limite al numero di pedine che possono occupare un punto. Infatti, è possibile avere anche tutte e 15 le pedine su un unico punto (anche se è molto raro). Quando si hanno più di 5 pedine su un punto, di solito è una buona idea impilare qualche pedina sopra le altre.

Mangiare e Rientrare

1.C'è un numero massimo di pedine che possono stare contemporaneamente sul bar?

Non c'è nessun limite. Anche se raramente, può succedere di vedere più di 6 pedine contemporaneamente sul bar.

2.Posso muovere altre pedine durante un turno in cui devo far rientrare una pedina dal bar?

Se non si hanno altre pedine sul bar, allora è obbligatorio utilizzare il numero del dado non sfruttato per muovere o la stessa pedina che è rientrata dal bar oppure una diversa. Se invece non è possibile far rientrare tutte le tue pedine dal bar, allora è obbligatorio farne rientrare il maggior numero possibile e rinunciare al resto dei numeri ottenuti dai dadi.

3.È possibile raggiungere una posizione di stallo nel backgammon?

Con posizione di stallo si intende una posizione in cui nessuno dei due giocatori può più muovere. È possibile immaginare una situazione del genere se entrambi i giocatori hanno pedine nel bar e tavole interne chiuse. Nessun giocatore in questo caso può rientrare e quindi nessuno potrà più muovere ed il gioco sarebbe bloccato.

Ma una tale situazione non può essere raggiunta attraverso mosse consentite dal regolamento. Ecco il perché. Supponiamo una situazione con entrambe le tavole interne chiuse. Qualcuno tra i due avversari avrà chiuso la propria tavola interna per primo; supponiamo sia stato il Bianco:

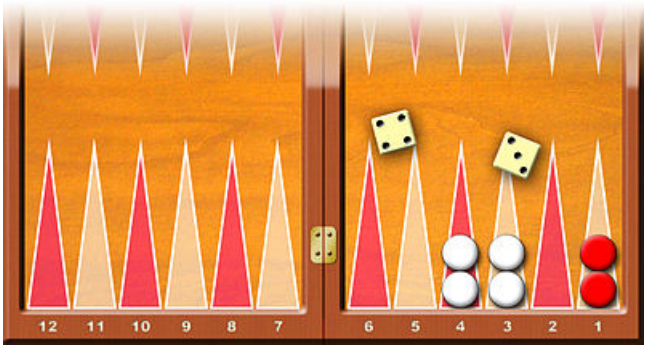
- il Bianco non può chiudere la sua tavola interna se ha una pedina sul bar. Quindi il suo avversario, il Rosso, deve mangiargli una pedina solo dopo che il Bianco sia riuscito a chiudere la propria tavola interna.
- Il Rosso non può mangiare, mentre resta chiusa la tavola interna del Bianco, se possiede ancora una pedina sul bar. Quindi, dopo che lui avrà mangiato una pedina al Bianco, quest'ultimo dovrà mangiarne una al Rosso.
- Ma il Bianco non può mangiare una pedina del suo avversario finché ha una pedina sul bar e la tavola interna del Rosso è chiusa. Quindi il Rosso ha presumibilmente chiuso la sua tavola interna dopo che il Bianco gli ha mangiato una pedina.
- Ma se il Bianco mangiasse, il suo avversario si ritroverebbe una pedina nel bar con la tavola interna del Bianco chiusa e non sarebbe quindi in grado di chiudere la propria tavola interna.

Quindi una situazione di doppia chiusura non potrà mai essere raggiunta.

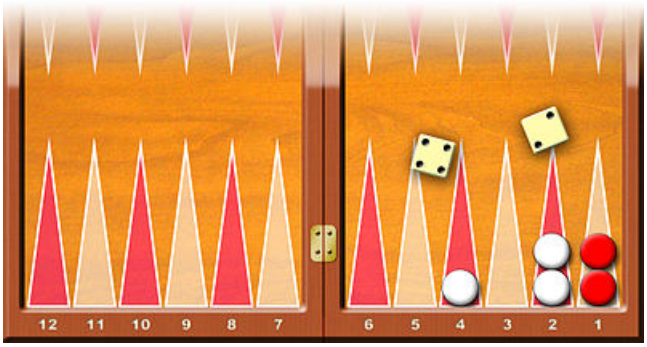
Portare fuori le pedine

1.Sono obbligato a portare fuori una pedina anche se non voglio?

Le regole richiedono che vengano utilizzati entrambi i numeri ottenuti dai dadi (quattro se esce un doppio) se ciò è possibile. Se si possono giocare delle mosse che non coinvolgano l'uscita delle pedine, è lecito giocarle. Se invece l'unica mossa consentita è quella di portare fuori una pedina, allora tale mossa deve essere giocata.



Qui è rappresentato un esempio in cui è obbligatorio portare fuori le pedine. Il Bianco ha ottenuto 4 e 3. In questa posizione, è obbligato a portare fuori una pedina dal punto 3 ed un'altra dal punto 4 - è infatti l'unica mossa legale che consente al Bianco di sfruttare entrambi i numeri.

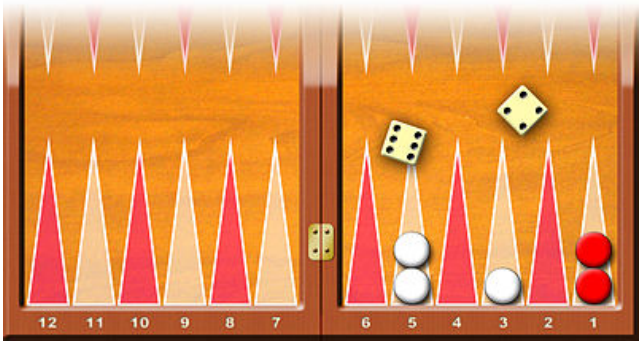


Qui c'è un esempio in cui il Bianco può portare fuori una oppure due pedine. Se il bianco porta fuori due pedine, lascia una pedina scoperta. Ma ciò non è necessario. Il Bianco può utilizzare il 2 per muovere una pedina dal punto quattro al punto due e poi può usare il 4 per portare fuori una pedina dal punto due. Il risultato è di aver portato fuori una pedina senza lasciarne una scoperta.

2.Posso portare fuori una pedina da una punta il cui numero è minore di quelli ottenuti con i dadi?

L'unico caso in cui si può portare fuori una pedina da una punta più bassa è quando non ci sono altre pedine sulle punte più alte. Si può utilizzare un 5 per portare fuori una pedina dal punto tre solo se non hai altre pedine sul quattro, sul cinque e sul sei (e naturalmente non si hanno pedine fuori dalla casa).

Nella seguente posizione, il Bianco può (e deve) utilizzare il 6 per portare fuori una pedina dal punto cinque, ma non ha modo di giocare il 4. Non è consentito utilizzare il 4 per portare fuori una pedina dal punto tre in quanto vi sono ancora pedine bianche sul punto cinque. Ed il Bianco non può neppure muovere una pedina dal punto cinque in quanto la mossa è bloccata dalle due pedine del Rosso nel punto uno.



Principio di funzionamento

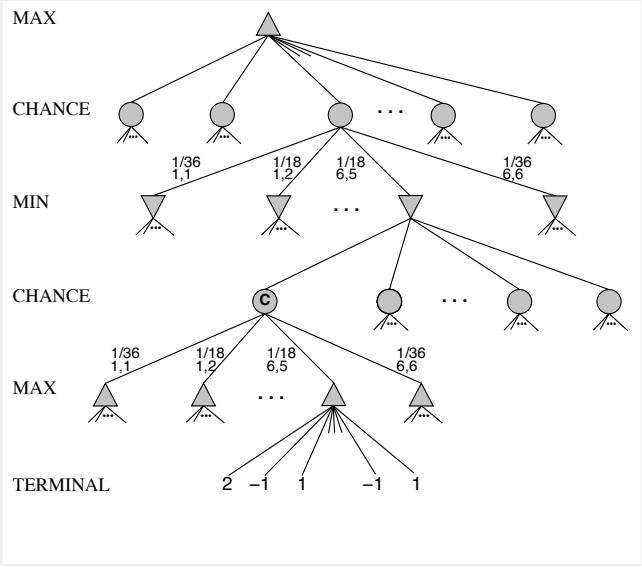
Software

Il software realizzato prevede essenzialmente 2 modalità di gioco: PC vs. HUMAN oppure PC vs. PC.

Inoltre per il giocatore PC é possibile scegliere quale funzione di valutazione dovrà esser utilizzata dall'algoritmo Expected MinMax.

Algoritmo Expected MinMax

Il Backgammon e' un gioco ad ***informazione completa***, a ***somma zero*** e ***stocastico*** ed il modello matematico discreto con cui è possibile rappresentare i problemi decisionali posti da questo tipo di giochi è ***l'albero di gioco***.



Questa struttura è costituita da nodi, che rappresentano i possibili stati del gioco intercalati da nodi che rappresentano le chance, e da archi, che rappresentano le mosse. Gli archi uscenti da un nodo corrispondono alle mosse legali nello stato del gioco associato a quel particolare nodo. La radice dell'albero è lo stato iniziale del gioco. I nodi foglia dell'albero sono gli stati finali del gioco, in cui non esistono mosse legali, a questi nodi è associato un valore che indica il risultato della partita.

Nei giochi molto semplici (per es. il Tic-tac-toe) l'albero di gioco ha una dimensione piuttosto ridotta, ma nei giochi più complessi assume dimensioni enormi (per es. l'albero di gioco degli Scacchi, Backgammon ha alcune migliaia di livelli e in media ogni nodo ha una trentina di figli).

Per costruire un ***giocatore artificiale*** occorre una strategia che sia in grado di decidere, dato uno stato del gioco (quindi un nodo dell'albero di gioco), quale sia la migliore mossa da giocare.

Prima di tutto occorre considerare che *la grande dimensione* dell'albero di gioco del Backgammon rende impensabile una sua analisi esaustiva per decidere la mossa da giocare. Perciò l'analisi deve necessariamente essere limitata ad un albero di gioco parziale. I nodi foglia di questo albero parziale possono non essere stati finali del gioco e quindi non avere associata alcuna informazione sul risultato della partita, Quindi una qualche valutazione di questi nodi è necessaria per decidere la mossa da giocare.

Perciò i nodi foglia dell'albero parziale vengono valutati da una apposita funzione, detta di valutazione statica, che cerca di approssimare il risultato finale della partita sulla base dello stato attuale del gioco. Un albero parziale si può ottenere principalmente in due modi:

• limitando l'estensione dell'albero, dal nodo attuale, ad una certa profondità

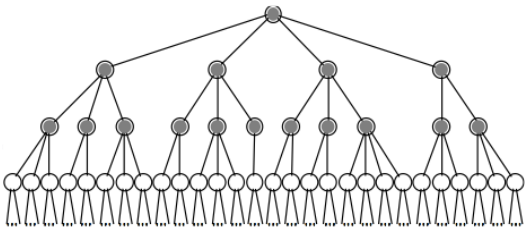


Figura 2.2. Un albero di gioco visitato con la strategia A fino al secondo livello di profondità. I nodi grigi sono quelli visitati.

• considerando non tutte le mosse legali, ma solo quelle che risultano più interessanti secondo dei criteri euristici

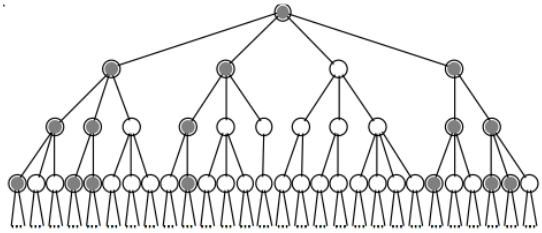


Figura 2.3. Un albero di gioco visitato con la strategia B.
I nodi grigi sono quelli visitati.

Gli attuali programmi di gioco adottano in genere la prima strategia, detta anche di "espansione cieca", e la seconda strategia, o di "espansione euristica", solo in casi particolari (ad es. per la ricerca quiescente). L'insuccesso dei programmi che adottano solo la seconda strategia è dovuto sostanzialmente alla difficoltà di stabilire a priori quali siano le mosse interessanti e quali quelle da scartare. Ragion per cui ho scelto di utilizzare la prima strategia per il mio progetto.

L'algoritmo ExpectedMinMax viene utilizzato per i giochi a due giocatori a somma zero, in cui l'esito dipende da una combinazione di abilità del giocatore e da elementi casuali, come il lancio dei dadi. Oltre ai nodi "MIN" e "MAX" della struttura tradizionale dell'albero di gioco *MiniMax*, questa variante introduce il nodo "chance", il cui valore è la probabilità che un evento casuale si verifichi.

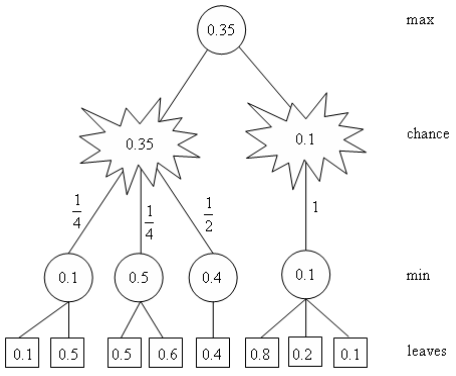


Figure 2.2: A sample expectimax game tree

Nel tradizionale algoritmo *minimax*, i livelli dell'albero si alternano da MAX a MIN fino a che il limite di profondità dell'albero non è stato raggiunto. In un *albero di gioco ExpectedMinMax,* i nodi **"*chance*"** sono intercalati con i nodi MAX e MIN.

Invece di prendere il massimo o minimo della valutazione dei loro figli, i nodi chance prendono ***la media ponderata***, con la probabilità con cui quel figlio è stato raggiunto. Esempio di chance:

$$\text{TheValue (C)} = (\text{probability}_1 \cdot \text{Max Value (position}_1\text{)} + \cdots + \text{probability}_b \cdot \text{MaxValue(position}_b\text{)}) \,/\, \#\text{chance}$$

L'interleaving dipende dal gioco. Ogni "turno" del gioco viene valutato come un nodo "MAX" (che rappresenta il turno del giocatore IA), un nodo "MIN" (che rappresenta il turno di un avversario potenzialmente non ottimale), o un nodo "CHANCE" (che rappresenta il tiro dei dadi ed il calcolo del successivo turno).

Codice algoritmo ExpectedMinMax:

```
/**Metodo principale dell'algoritmo ExpectedMinMax
* @param bb stato del gioco
* @param depth profondità
*  @return nuovo stato del gioco, dopo aver eseguito la mossa migliore
*/
public static BackgammonBoard MinMax(BackgammonBoard bb, int depth, Evalutation eval){

       .....
       ArrayList<BackgammonBoard> children = null;

        /* Genero i figli dello stato autuale*/
       children = EngineMinMax.getChildren2(bb);

       .....
     for (int i = 0; i < children.size(); i++) {

              tempmin = ExpectedMinMax(children.get(i), depth-1, CHANCEAFTERMAX, 1,eval);
              children.get(i).setValue((float) tempmin);

              /* Calcolo il massimo valore ritornato da ExpectedMinMax*/
              if (tempmin > maxsofar || calculated == false) {
                     bestboard = i;
              maxsofar = tempmin;
              calculated = true;
              }
       }

       return children.get(bestboard);
        ....
    }

/**Algoritmo ExpectedMinMax
* @param bb stato del gioco
* @param depth profonditá
* @param state player relativo allo stato del gioco
* @param level livello di profonditá
* @return Exptected Value dello stato
*/
private static double ExpectedMinMax(BackgammonBoard bb, int depth, int state, int level, Evalutation eval) {

       int nextstate = (state + 1) % 4;
       double value = 0.0;

       if ((bb.isTerminal()) || (depth == 0))
       return eval.boardscore(bb, bb.getHasToMove());

       switch (state) {
              case MIN: {
              value = Integer.MAX_VALUE;

                     for (BackgammonBoard child : EngineMinMax.getChildren2(bb))
                        value = Math.min(value, ExpectedMinMax(child, depth - 1,
```

```
                                nextstate, level + 1,eval));
          }
          break;
          case MAX: {
               value = Integer.MIN_VALUE;

             for (BackgammonBoard child : EngineMinMax.getChildren2(bb))
                     value = Math.max(value, ExpectedMinMax(child, depth - 1,
                                nextstate, level + 1,eval));
        }
        break;
        case CHANCEAFTERMAX:
        case CHANCEAFTERMIN: {
            .......
            for (int i = 0; i < 21; i++) {
                dc1 = dierolls[i][0];
                dc2 = dierolls[i][1];
                prob = Dice.dicePorobability(dc1,dc2);
                bb.setDice1(dc1);
                bb.setDice2(dc2);

                for (BackgammonBoard child : EngineMinMax.getChildren2(bb))
                        value += prob * ExpectedMinMax(child, depth - 1,
                                                                nextstate, level + 1,eval);
            }
            value /= 21.0;
        }
        break;
    }
    return value;
}
```

Funzione di Valutazione

In un giocatore artificiale la funzione di *valutazione statica* rappresenta una delle parti fondamentali e più complesse. La qualità di un programma di gioco è in buona parte determinata dalla capacità della propria funzione di valutazione di stimare con precisione le posizioni nei nodi foglia dell'albero di gioco. Questa valutazione viene fatta attraverso l'osservazione di caratteristiche notevoli della posizione in esame. La posizione viene esaminata sotto diversi aspetti tramite l'uso di diversi termini di conoscenza (vicinanza alla tavola interna, numero di posizioni chiuse, numero di posizioni aperte, chiusura della tavola interna ecc.) che poi vengono sintetizzate in un unico valore tramite un'apposita funzione. Tipicamente una funzione di valutazione ha la seguente forma:

$$f(p) = \sum_{i=1}^{N} weight_i \cdot A_i(p)$$

dove con Ai(p) si indicano i valori dei termini di conoscenza applicati alla posizione p e con weight il peso relativo di ciascun termine di conoscenza.

Una importante caratteristica che una buona funzione di valutazione deve avere è di non essere rigida, cioè di operare in maniera diversa a seconda della posizione a cui è applicata. Per esempio è bene che la funzione di valutazione adotti termini di conoscenza o pesi diversi per le varie fasi della partita (tipicamente divisa in apertura, mediogioco e finale) in cui è noto che sono necessarie metastrategie e quindi conoscenze diverse. È anche utile che il giocatore artificiale sappia riconoscere situazioni particolari in cui sono necessarie conoscenze molto specifiche: per esempio in un finale con pedine nel bar è inutile applicare una funzione di valutazione come quella per la fase di mediogioco, è necessario piuttosto applicare un insieme di regole strategiche appositamente studiate per questo tipo di finale.

Un problema ricorrente per la funzione di valutazione, chiamato "effetto orizzonte" , è valutare correttamente le posizioni in cui sono possibili molte mosse di "mangia la pedina / cattura" chiamate posizioni turbolente. In queste posizioni è difficile dare una valutazione statica a causa dei numerosi tatticismi che vi sono e che possono essere visti solo analizzando ulteriormente l'albero di gioco. Anche per questo motivo ho scelto di adottare una strategia di espansione cieca fino ad una certa profondità.

Sarebbe stato possibile usare una tecnica di ricerca, chiamata ricerca quiescente, che permette di espandere ulteriormente l'albero di gioco per le posizioni turbolente considerando solo le mosse di cattura. La funzione di valutazione viene poi applicata ai nodi foglia dell'albero di gioco così espanso, che non contengono posizioni turbolente.

Per la realizzazione di questo progetto, viste le finalità, ho realizzato 3 differenti funzioni di valutazione, cercando di applicare la teoria, appena citata, ma dopo qualche prova mi sono reso conto che ciò avrebbe richiesto molto tempo e una serie di conoscenze specifiche sul gioco. Quindi ho deciso di basarmi sulla mia poca esperienza di gioco, sicuramente ottenendo risultati inferiori.

Codice BackgammonEvalutationFirst:

```
/**
 *Classe di valutazione dello stato del gioco.
 * Prima funzione di valutazione realizzata. Non molto dettagliata, ma buona dal punto di    vista della valutazione.
 *
 * @author filippofontanelli
 */
public class BackgammonEvalutationFirst extends Evalutation{
    @Override
    public float boardscore(BackgammonBoard bb, int player) {
        float bsum = 0;
        float wsum = 0;
/*prima valutazione su quante posizioni non chiuse ci sono */
        for (int i = 1; i <= Board.NUM_BOARD; i++) {

/*
* Valutazione ponderata della posizione, se ho piu di una pedina, ovvero se la posizione é chiusa, allora incremento di 4 altri-
menti se la pozione
* non é chiusa, allora decremento di 3
*/
            if (bb.getPlayboard().getBoard()[i].getNumofcheckers() > 1 && bb.getPlayboard().getBoard()[i].getPlayer() ==
Board.BLACK) {
                bsum += 4;
            } else if (bb.getPlayboard().getBoard()[i].getNumofcheckers() == 1 && bb.getPlayboard().getBoard()[i].getPlayer() ==
Board.BLACK) {
                bsum -= 3;
            }
/*
* Faccio lo stesso per il white
*/
            if (bb.getPlayboard().getBoard()[i].getNumofcheckers() > 1 && bb.getPlayboard().getBoard()[i].getPlayer() ==
Board.WHITE) {
                wsum += 4;
            } else if (bb.getPlayboard().getBoard()[i].getNumofcheckers() == 1 && bb.getPlayboard().getBoard()[i].getPlayer() ==
Board.WHITE) {
                wsum -= 3;
            }
        }
/*
* Controllo quanti elementi hanno nella Home e nel Bar, incrementando il valore per le pedine nella Home e decrementandolo per
quelle del Bar
*/
        int bwhite = bb.getPlayboard().getOnBar(Board.WHITE).getNumofcheckers();
        if (bwhite > 0) {
            wsum -= bwhite * 4;
            bsum += bwhite * 4;
        }
        int ewhite = bb.getPlayboard().getInHome(Board.WHITE).getNumofcheckers();
        if (ewhite > 0) {
            wsum += ewhite * 5;
        }
        int bblack = bb.getPlayboard().getOnBar(Board.BLACK).getNumofcheckers();
        if (bblack > 0) {
            bsum -= bblack * 4;
            wsum += bblack * 4;
        }
        int eblack = bb.getPlayboard().getInHome(Board.BLACK).getNumofcheckers();
        if (eblack > 0) {
            bsum += eblack * 5;
        }

        bb.setValue((float) (bsum - wsum));
        return bsum - wsum;
    }
```

Codice BackgammonEvalutationSecond:

```
/**
 *Classe di valutazione dello stato del gioco.
 * Funzione di valutazione realizzata sulla base di alcuni principi di valutazione
 * noti nel gioco del backgammon.
 *
 * @author filippofontanelli
 */
public class BackgammonEvalutationSecond extends Evalutation {

    private static int BLOCKBONUS = 3;
    private static float BLOTBONUS = (float) .25;
    private static int ANCHORBONUS = 5;
    private static int DISTRIBUTIONBONUS = 2;
    private static float GETGOINGBONUS = (float) 0;
/**
* Costruttore
*/
public BackgammonEvalutationSecond()
......

@Override
public float boardscore(BackgammonBoard bb, int player) {

.......
Position[] board = bb.getPlayboard().getBoard();
/*
* Black case
*/
       if (player == Board.BLACK) {
              for (int i = 1; i <= Board.NUM_BOARD; i++) {

                if (board[i].getPlayer() == Board.BLACK) {
                    seenself = true;
                    if (owncount == 0) {
                        countopblots = true;
                    }
                    owncount += board[i].getNumofcheckers();
                    if (blockinrow < 0) {
                        blockinrow = 0;
                    }
/*
* Differenzio la valutazione in basa alla porsione di tavola in cui mi trovo
*/
                    if (i < 13) {
                        sum -= GETGOINGBONUS * board[i].getNumofcheckers() * (12 - i);
                    } else {
                        sum += GETGOINGBONUS * board[i].getNumofcheckers() * (i - 12);
                    }
/*
* Fino a che il gioco non é "finito"
*/
                    if (!endgame) {
                        if (board[i].getNumofcheckers() > 1) {
                            blockinrow++;
                            sum += blockinrow * BLOCKBONUS;
                            /* Nel caso in cui sia nella mia board interna*/
                            if (i > 18) {
                                anchorcount++;
                            }
                        } else {
/*se non ho la posizione chiusa oppure se la posizione é vuota*/
                            blockinrow = 0;
                            /*Calcolo un incremento relativo alla posizione*/
                            sum += (opponentcount - 15 - i) * BLOTBONUS;
                            if (i > 18) {
                                anchorblot++;
                            }
                        }
                    }
/*Nel caso in cui una posizione appartenga al mio avversario ( in questo caso White)*/
                } else if (board[i].getPlayer() == Board.WHITE) {
/*Incremento il contatore dei blocchi avversari*/
                    opponentcount += board[i].getNumofcheckers();
                    if (blockinrow > 0) {
                        blockinrow = 0;
                    }
                    if (!endgame) {
```

```
                        if (board[i].getNumofcheckers() > 1) {
                            blockinrow--;
                            sum += blockinrow * BLOCKBONUS;
/*Nel caso in cui sia nella tavola interna*/
                            if (i < 7) {
                                opponentanchorcount++;
                            }
                        } else {
                            if (countopblots) {
                                opponentblots++;
                            }
                            blockinrow = 0;
                            if (owncount + board[Board.BLACKBAR].getNumofcheckers() > 0) {
                                sum += ((owncount + (23 - i)) * BLOTBONUS);
                            }
                        }
                    }
                } else {
                    blockinrow = 0;
                }
            }
/*
* Calcolo bonus finale ottenuto dalla differenza delle posizioni ancorare, le pedine del bar e il numero di posizioni bloccate
dell'avversario*/
            float asdf = ((anchorcount - anchorblot) * (board[Board.WHITEBAR].getNumofcheckers() + opponentblots)) * ANCHORBONUS;
            sum += asdf;
        } else if (player == Board.WHITE) {

                                        .... Stessa logica utilizzata per il player White...
```

Conclusioni

Dopo una lunga ricerca di informazioni sulla realizzazione di *giocatori artificiali* tramite *l'algoritmo MinMax* mi sono reso conto della vera complessità del problema.

Infatti problemi di questo tipo (senza auto-apprendimento) necessitano di molte funzioni/moduli/strutture dati ausiliarie per ottimizzare il calcolo dell'algoritmo e di conseguenza migliorare la qualità della scelta della mossa migliore a partire da un generico nodo dell'albero di gioco.

Infatti molto spesso vengono *combinati* più algoritmi, i più noti e utilizzati ad esempio sono:

• *L'algoritmo AlphaBeta* nasce dall'osservazione che molti nodi visitati dall'algoritmo MiniMax (o NegaMax) non contribuiscono alla valutazione della radice del sottoalbero di gioco che viene analizzato. L'idea di base dell'algoritmo è che quando si è scoperto che una variante non contribuirà alla valutazione del nodo radice, è inutile continuare ad analizzarla.

• *La variante Aspiration Search* differisce dall'algoritmo AlphaBeta per la finestra alpha-beta attribuita al nodo radice della ricerca. Anziché essere la canonica $\alpha = -\infty$, $\beta = +\infty$, la finestra iniziale viene calcolata sulla base di una valutazione provvisoria v del nodo radice e una costante K. Per cui si ha $\alpha = v - K$, e $\beta = v + K$. Lo scopo di questa nuova finestra è di generare un maggior numero di tagli.

• *La variante Fail Soft*, per migliorare le prestazioni dell'algoritmo Aspiration Search in caso di fallimento della prima ricerca è stato introdotta la variante Fail Soft. Questa variante consente di effettuare l'eventuale seconda ricerca con una finestra più piccola di quella usata dall'Aspiration Searc

• ecc..

Inoltre é noto che anche con gli algoritmi di analisi più sofisticati, la visita dell'albero di gioco è un'operazione dispendiosa, di complessità esponenziale rispetto alla profondità di ricerca, per cui diverse euristiche sono state introdotte per cercare di ottimizzarla.

• *Libro delle aperture* è un archivio in cui a delle posizioni sono associate mosse da giocare.

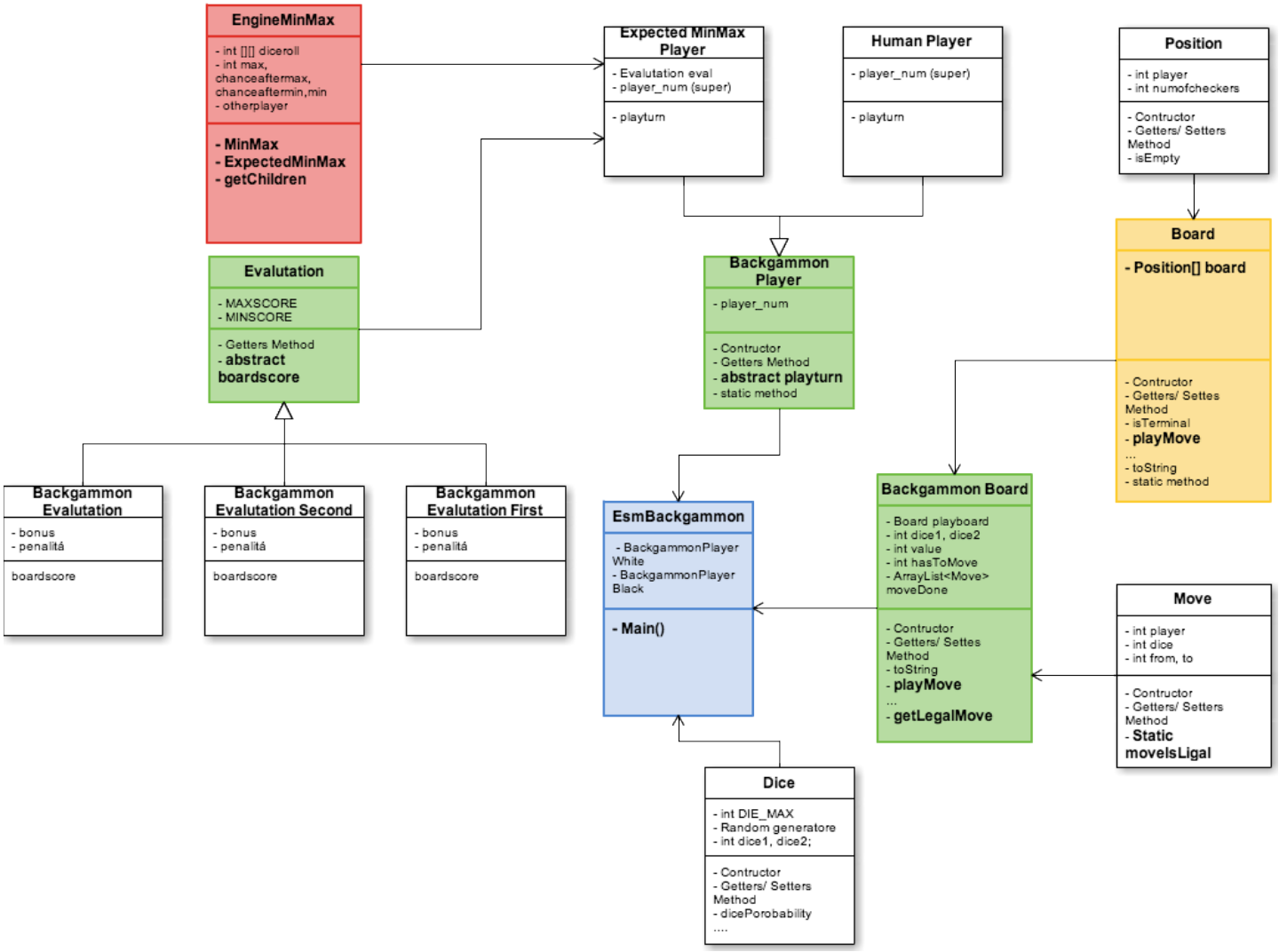
• *Tabella delle trasposizioni* utilizzata ormai da tutti i programmi di gioco, è una tabella hash in cui vengono memorizzate informazioni riguardanti le posizioni analizzate durante la ricerca.

Infine anche l'ordine in cui le mosse vengono analizzate dall'algoritmo AlphaBeta (o da sue varianti) è fondamentale ai fini della sua efficienza. Per questo sono state sviluppate diverse euristiche: come l'ordinamento a priori, mosse killer ecc.

Scelte progettuali

Diagramma delle classi

Il progetto ha una struttura estendibile, ho cercato di realizzarlo suddividendolo logicamente in base alle varie componenti che si possono riconoscere nel gioco del Backgammon. In particolare le componenti principali sono:



(1) ***Board***, che rappresenta la tavola da gioco in quanto tale, implementando i meccanismi “fisici” previsti dal gioco e la logica base dello spostamento delle pedine.

(2) ***BackgamonBoard***, rappresenta lo stato del gioco/il nodo radice dell’albero di gioco. Lo stato é composto da: la board, i due dadi, dal player attualmente di turno, le mosse effettuate nel turno precedente e dal valore associato allo stato. Inoltre essa contiene l’implementazione della logica relativa alle principali regole del Backgammon.

(3) ***BackgammonPlayer***, rappresenta la struttura e i comportamenti attesi da un player di Backgammon

(4) ***EngineMinMax***, rappresenta l’algoritmo Expected MinMax e tutte le utility utilizzate da esso.

La tavola da gioco

La tavola da gioco viene rappresentata tramite i seguenti oggetti:

1. ***Position***, che rappresenta un generico contenitore di pedine.

2. ***Board***, che rappresenta la vera e propria tavola da gioco, realizzata tramite un Array di Position così strutturati:

```
||=13===14===15===16===17===18==|=BAR=|==19==20===21===22===23===24==|=HOME=||
||==v====v====v====v====v====v==|==25=|==v====v====v====v====v====v==|==27==||
||  0                   X       |     |  X                        0  |       ||
||  0                   X       |     |  X                        0  |       ||
||  0                   X       |     |  X                           |       ||
||  0                           |     |  X                           |       ||
||  0                           |     |  X                           |       ||
||                              |     |                              |       ||
||                              |     |                              |       ||
||                              |     |                              |       ||
||                              |     |                              |       ||
||  X                           |     |  0                           |       ||
||  X                           |     |  0                           |       ||
||  X                   0       |     |  0                           |       ||
||  X                   0       |     |  0                        X  |       ||
||  X                   0       |     |  0                        X  |       ||
||==A====A====A====A====A====A==|==0==|==A====A====A====A====A====A==|==26==||
||=12===11===10====9====8====7==|=BAR=|==6====5====4====3====2====1==|=HOME=||
```

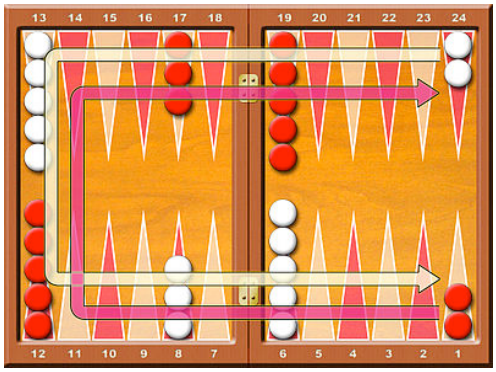
Le posizioni:

1. Con indice compreso tra ***1*** e ***24*** rappresentano ***le punte della tavola***.

2. Con indice ***0*** e ***25*** rappresentano rispettivamente ***il Bar del giocatore White e del giocatore Black***.

3. Con indice ***26*** e ***27*** rappresentano rispettivamente ***la Home del giocatore White e del giocatore Black***.

Andando a modificare il meno possibile il normale scenario di gioco. Per maggiori dettagli si veda la sezione relativa all'introduzione al gioco.



Le pedine **X (Rosse)** vengono mosse in senso antiorario dal player **Black** e le pedine **O (Bianche)** vengono mosse in senso orario dal **player White**.

Test

Ambiente di Test

I test sono stati effettuati con la seguente architettura:

MacBook Pro (Metà 2010)

Processore 2,4 GHz Intel Core 2 Duo

Memoria 4 GB 1067 MHz DDR3

Software OS X 10.8.2 (12C60)

Human vs ExpectedMinMax

Dopo aver effettuato alcune partite test utilizzando diverse profondità e diverse funzioni di valutazione i risultati più evidenti sono stati:

1. Per ottenere una buona velocità di gioco, il valore ***depth*** dell'algoritmo deve esser compreso tra 1 e 3.

2. L'algoritmo necessita di una quantità rilevante di memoria e di tempo in caso in cui si é fatto doppio.

3. Provando ad utilizzare le diverse funzioni di valutazione, il giocatore artificiale risulta perdente nella maggior parte dei casi.

ExpectedMinMax vs ExpectedMinMax

Dopo aver effettuato alcune partite test utilizzando diverse profondità e diverse funzioni di valutazione i risultati più evidenti sono stati:

1. Per ottenere una buona velocità di gioco, il valore *depth* dell'algoritmo deve esser compreso tra 1 e 2.

2. L'algoritmo necessita di una quantità rilevante di memoria e di tempo in caso in cui si é fatto doppio.

Vediamo alcuni risultati ottenuti, modificando la profondità e la funzione di valutazione:

Evalutation vs Evalutation

1. **Depth = 1**

N. Test	Execution Time Black Player	Execution Time White Player	N. Move (Black/White)	Who Win
1	199 ms	172 ms	28/28	White
2	145 ms	73 ms	25/26	White
3	279 ms	70 ms	33/32	Black
4	549 ms	76 ms	39/39	White
5	142 ms	289 ms	22/21	Black

2. **Depth = 2**

N. Test	Execution Time Black Player	Execution Time White Player	N. Move (Black/White)	Who Win
1	297175 ms	41921 ms	54/53	Black
2	282782 ms	441958 ms	44/43	Black

3. **Depth = 3**

Si nota un brusco rallentamento del calcolo, specialmente in caso di doppio, il quale fa si che la partita duri circa 5/7 minuti.

EvalutationFirst vs EvalutationFirst

1. **Depth = 1**

N. Test	Execution Time Black Player	Execution Time White Player	N. Move (Black/White)	Who Win
1	223 ms	56 ms	22/22	Black
2	69 ms	841 ms	34/33	Black
3	129 ms	217 ms	30/29	Black
4	33 ms	621 ms	28/28	Black
5	22 ms	835 ms	27/29	White

2. **Depth = 2**

N. Test	Execution Time Black Player	Execution Time White Player	N. Move (Black/White)	Who Win
1	5520896 ms	109274 ms	65/64	Black
2	4102235 ms	325268 ms	59/60	White

3. **Depth = 3**

Si nota un brusco rallentamento del calcolo, specialmente in caso di doppio, il quale fa si che la partita duri circa 5/7 minuti.

EvalutationSecond vs EvalutationSecond

1. **Depth = 1**

N. Test	Execution Time Black Player	Execution Time White Player	N. Move (Black/White)	Who Win
1	168 ms	45 ms	27/27	Black
2	119 ms	81 ms	29/30	White
3	403 ms	197 ms	32/32	Black
4	122 ms	192 ms	24/24	White
5	136 ms	70 ms	20/21	White

2. **Depth = 2**

N. Test	Execution Time Black Player	Execution Time White Player	N. Move (Black/White)	Who Win
1	21473 ms	10370 ms	24/23	Black
2	7492 ms	8328 ms	26/26	Black
3	390308 ms	287075 ms	32/32	Black

3. **Depth = 3**

Si nota un brusco rallentamento del calcolo, specialmente in caso di doppio, il quale fa si che la partita duri circa 5/7 minuti.

EvalutationFirst vs Evalutation

1. **Depth = 2**

N. Test	Execution Time Black Player	Execution Time White Player	N. Move (Black/White)	Who Win
1	4653 ms	3080 ms	27/27	Evalutation First (Black)
2	66618 ms	591950 ms	50/50	Evalutation First (Black)
3	14150 ms	12850 ms	24/25	Evalutation First Black)

EvalutationSecond vs Evalutation

1. **Depth = 2**

N. Test	Execution Time Black Player	Execution Time White Player	N. Move (Black/White)	Who Win
1	4653 ms	3080 ms	27/27	EvalutationSecond (Black)
2	66618 ms	591950 ms	50/50	EvalutationSecond (Black)
3	270682 ms	337801 ms	31/30	EvalutationSecond (Black)

EvalutationFirst vs EvalutationSecond

1. **Depth = 2**

N. Test	Execution Time Black Player	Execution Time White Player	N. Move (Black/White)	Who Win
1	44437 ms	59302 ms	27/27	EvalutationFirst (Black)
2	78308 ms	211863 ms	55/55	EvalutationSecond (White)
3	41242 ms	152327 ms	31/30	EvalutationFirst (Black)

Analisi delle classi

In questo capitolo perderemo in considerazione le singole classi, mettendo in risalto le funzionalità e i comportamenti più importanti. Si rimanda al ***JavaDoc*** e al ***codice*** per maggiori dettagli.

Position

Classe che rappresenta un contenitore di pedine come le punte della board, la home e il bar.

Attributi

La classe rende disponibili le seguenti variabili d'istanza:

```
private int player

private int numofcheckers
```

La prima é l'identificativo del giocatore associato alla position, la seconda identifica il numero di pedine posizionate nel contenitore.

Metodi

I comportamenti che la classe Position mette a disposizione sono quelli "classici": creazione, set e get degli attributi.

Board

Classe che rappresenta la tavola da gioco in quanto tale, implementando i meccanismi "fisici" previsti dal gioco e la logica base dello spostamento delle pedine.

Attributi

La classe rende disponibili le seguenti variabili d'istanza:

```
private Position[] board
```

La quale rappresenta la tavola da gioco composta dalle 24 punte, il bar e la home per entrambi i giocatori.

Metodi

I comportamenti che la classe Board mette a disposizione sono quelli "classici": creazione, set e get degli attributi ed una serie di utility per il suo utilizzo. Vediamo in dettaglio i più significativi:

Board

Il costruttore oltre ad inizializzare le strutture dati della classe, provvede a posizionare il giusto numero di pedine nelle posizioni di partenza previste dal gioco.

```
public Board() {

   board = new Position[NUM_POSITION];
   for (int i = 0; i < NUM_POSITION; i++) {
       board[i] = new Position();
   }
```

```
        //Black Player
        board[1].setPosition(BLACK, 2);
        board[12].setPosition(BLACK, 5);
        board[17].setPosition(BLACK, 3);
        board[19].setPosition(BLACK, 5);
        board[BLACKBAR].setPosition(BLACK, 0);
        board[BLACKEXIT].setPosition(BLACK, 0);

        //White Player
        board[24].setPosition(WHITE, 2);
        board[13].setPosition(WHITE, 5);
        board[8].setPosition(WHITE, 3);
        board[6].setPosition(WHITE, 5);
        board[WHITEBAR].setPosition(WHITE, 0);
        board[WHITEEXIT].setPosition(WHITE, 0);

    }
```

isTerminal

Controlla se a board attuale identifica una board terminale, ovvero se uno dei due giocatori ha un numero di pedine nella propria home pari a 15.

```
    public boolean isTerminal() {
        return (hasWon(BLACK) || hasWon(WHITE));

    }
```

PlayMove

Metodo che si occupa di prelevare la pedina dalla punta indicata dalla *Move m.from* e successivamente di inserire la pedina nella punta indicata dalla *Move m.to*, applicando i meccanismi di spostamento nel bar e di exit.

```
    public boolean playMove(Move m) {
          return pickCheckers(m.getPlayer(), m.getFrom()) && putCheckers(m.getPlayer(),
    m.getTo());
    }
```

Dice

Classe che rappresenta i due dadi utilizzati nel gioco.

Attributi

La classe rende disponibili le seguenti variabili d'istanza:

```
    public final static int DIE_MAX = 6;

    private Random generatore;

    private int dc1, dc2;
```

le quali rappresentano rispettivamente il massimo valore di un dado, un oggetto della classe random per il lancio dei dadi ed il valore corrente dei due dadi.

Metodi

I comportamenti che la classe Dice mette a disposizione sono quelli “classici”: creazione, set e get degli attributi ed il metodo *toss* per il lancio dei dadi.

Move

Classe che definisce una mossa di un determinato player, relativa all’utilizzo di un determinato dado.

Attributi

La classe rende disponibili le seguenti variabili d’istanza:

```
private int player;

private int dice;

private int from;

private int to;
```

le quali rappresentano rispettivamente il player a cui la mossa é associata, il relativo dado, la posizione di partenza e la posizione di arrivo della pedina da spostare.

Metodi

I comportamenti che la classe Move mette a disposizione sono quelli “classici”: creazione, set e get degli attributi. Inoltre vediamo in dettaglio il metodo più significativo:

MoveIsLegal

Metodo che controlla se la *Move m* é legale per lo stato attuale rappresentato dalla *BackgammonBoard bb*.

```
public static boolean moveIsLegal(Move m, BackgammonBoard bb) {

  ......

   ret = ret & checkPickCheckers(bb.getPlayboard(),bb.getHasToMove(),m.getFrom());
   ret = ret & checkPutCheckers(bb.getPlayboard(),bb.getHasToMove(),m.getTo());

 .......
}
```

Backgammon Board

Classe che rappresenta lo stato del gioco/il nodo radice dell’albero di gioco. Inoltre essa contiene l’implementazione della logica relativa alle principali regole del Backgammon.

Attributi

La classe rende disponibili le seguenti variabili d’istanza:

```
private Board playboard

private int dice1;

private int dice2;
```

```
    private float value;

    private int hasToMove;

    private ArrayList<Move> moveDone;
```

le quali rappresentano rispettivamente la tavola di gioco attuale, il valore dei due dati appena tirati,il valore associato allo stato, l'identificativo del giocatore che é di mano e le ultime mosse effettuate per raggiungere lo stato attuale.

Metodi

I comportamenti che la classe BackgammonBoard mette a disposizione sono quelli "classici": creazione, set e get degli attributi. Vediamo in dettaglio i più significativi:

PlayMove

Metodo che implementa la giocata della mossa ricevuta come parametro e la sua memorizzazione nelle variabili d'istanza. Si assume che la mossa sia già stata controllata e quindi "etichettata" come legale.

```
    public boolean playMove(Move m) {

       boolean ret = true;
       /*
        * Effettuo la mossa e aggiorno le variabili d'istanza
        */
       try {
           ret = playboard.playMove(m);
       } catch (Exception e) {
           System.out.println("playMove -> "+ m.toString() + ", "+e.toString());
           ret = false;
       }

       if (ret) moveDone.add(m);

       return ret;
   }
```

GetLegalMove

Metodo che identifica le mosse legali a partire stato attuale.

```
    public ArrayList<Move> getLegalMove(boolean used1, boolean used2) {

       ArrayList<Move> moves = new ArrayList<Move>();

       checkMoveFrom(moves, used1, used2);

       moves = checkMoveTo(moves);

       return moves;

   }
```

Procedo andando a generare tutte le mosse per ogni posizione di partenza valida tramite il metodo *checkMoveFrom*, e successivamente tramite il metodo *checkMoveTo* verifico e correggo la posizione di arrivo delle mosse individuate al passo precedente.

Backgammon Player

Classe astratta che identifica un generico giocatore.

Attributi

La classe rende disponibili la seguente variabile d'istanza:

```
private int player_num;
```

la quale rappresenta il numero identificativo del player (Es: 0 oppure 1).

Metodi

I comportamenti che la classe *BackgammonPlayer* mette a disposizione sono quelli "classici": creazione, set e get degli attributi ed alcune utility. Vediamo in dettaglio il più significativo:

PlayTurn

Metodo astratto che identifica il comportamento di un determinato player quando é il suo turno.

```
public abstract BackgammonBoard playturn(BackgammonBoard bb);
```

Human Player

Classe che estende la classe astratta BackgammonPlayer. La classe HumanPlayer identifica il giocatore umano del Backgammon.

Attributi

La classe eredita le variabili d'istanza dalla classe BackgammonPlayer.

Metodi

La classe *HumanPlayer* implementa il metodo astratto *PlayTurn*, il quale gestisce la logica e le modalità di interazione tra l'utente e il gioco, in particolare questo metodo permetterà all'utente di indicare le mosse da effettuare.

Si rimanda all'appendice A per maggiori dettagli.

ExpectedMinMax Player

Classe che estende la classe astratta BackgammonPlayer. La classe identifica il giocatore artificiale.

Attributi

La classe eredita le variabili d'istanza dalla classe BackgammonPlayer.

Metodi

La classe *ExpectedMinMaxPlayer* implementa il metodo astratto *PlayTurn*, il quale tramite l'algoritmo ExpectedMinMax determina la mossa migliore da eseguire e la esegue.

Evalutation

Classe astratta che identifica una generica funzione di valutazione di un nodo dell'albero di gioco.

Attributi

La classe rende disponibili le seguenti variabili d'istanza:

```
protected float MAXSCORE;
```

```
protected float MINSCORE;
```

le quali rappresentano rispettivamente il punteggio massimo e il punteggio minimo attribuito da una particolare funzione di valutazione ad uno stato della board.

Metodi

I comportamenti che la classe *Evalutation* mette a disposizione sono quelli “classici”: set e get degli attributi. Vediamo in dettaglio il più significatio:

BoardScore

Metodo astratto per la valutazione dello stato della board.

```
public abstract float boardscore(BackgammonBoard bb, int player);
```