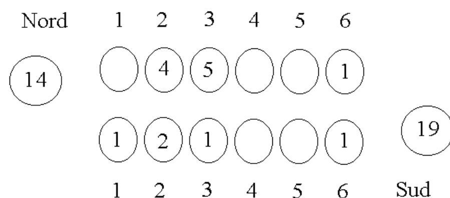
Un programma che gioca a Mancala

Angelo Ferrando, Alessandro Bonanno, Lorenzo Repetto
Istituto Tecnico Industriale Statale "Italo Calvino" – Genova
Via Borzoli 21, 16153, {alex.bonanno,repetto}@calvino.ge.it

Parole chiave: Mancala o Awele, spazio degli stati legali, gioco strettamente determinato, albero di gioco, algoritmo *minimax*, potature.
Classi destinatarie dell'iniziativa didattica: quarte dell'indirizzo informatico.

Lo scorso anno, uno di noi (Ferrando, all'epoca studente di quarta dell'indirizzo sperimentale "Abacus") realizzò un programma in grado di giocare a una delle più semplici tra le oltre duecento varianti di un antichissimo gioco da tavolo africano, oggi diffuso in tutto il mondo: il Mancala o Awele.
Un bel libro che ne parla è *Giochi africani*, di Carlo Zampolini (Sansoni Editore, Firenze 1984, pp. 23-78). Si gioca su un tavoliere che ha, per ciascuna delle due parti, una fila di 6 buche (contenenti 4 semi ciascuna) e un "granaio" (che conterrà i semi catturati). Comune a tutte le varianti di questa famiglia di giochi, di forte valenza rituale, è il ciclo *antiorario*, forse legato a quello celeste (a nord dell'equatore) o al terreno. Il giocatore che deve muovere sceglie una delle sue buche, non vuota; preleva tutti i semi che vi sono contenuti e li semina a uno a uno nelle buche successive, anche nel campo opposto, saltando poi la buca originaria se sono più di 11. Se l'ultimo seme cade in una buca avversaria che contiene 2 o 3 semi (compreso quello appena seminato) questi sono catturati, e così di seguito, a ritroso, finché si incontrano buche nel campo avversario con 2 o 3 semi. Se una posizione si ripete tre volte, la partita termina: i semi rimasti sono divisi esattamente a metà tra i due giocatori, e aggiunti ai rispettivi granai.
Le principali varianti del gioco sono legate alla situazione di blocco (e poi alla conduzione di una *manche*). Originariamente adottammo la regola più semplice per il finale: se tutte le buche nel campo del giocatore di turno sono vuote, l'avversario trasferisce nel proprio granaio i semi che gli sono eventualmente rimasti, e la partita finisce. Il punteggio è determinato dalla differenza tra i contenuti dei rispettivi granai.
Successivamente creammo un'altra versione del programma, secondo la regola scelta nel 1991 da alcuni ricercatori olandesi per l'analisi al computer, e ormai ben nota come Awari anche nella comunità informatica: per prolungare il gioco, non sono consentite mosse che lasciano l'avversario senza semi, eccetto nel caso in cui tutte portino inevitabilmente a questa situazione. (Non è permesso comunque saltare un turno, ciò che invece è previsto in altre varianti.)
Ad esempio, nello stato di figura con tratto a Sud, se questi muove dalla buca 6 allora deve mangiare e guadagna 2 semi, ma poi Nord vince 27 a 21 perché entrambe le mosse che può fare lasciano Sud senza semi. Se invece la buca 6 di Nord fosse vuota, a seguito della stessa mossa di Sud, adesso Nord sarebbe obbligato a muovere lo stesso unico seme dalla propria buca 6 nella 5: infatti, ora le altre due mosse sono illegali poiché lasciano Sud senza semi.



Nell'estate del 2002, John W. Romein e Henri E. Bal, della Vrije Universiteit di Amsterdam, hanno *risolto* questa variante del gioco, determinando il risultato finale e le mosse giuste per quasi 900 miliardi di posizioni (che ne costituiscono lo *spazio degli stati legali*, cioè raggiungibili in una partita reale) e concludendo che, quando entrambi i competitori giocano in modo ottimale, la partita termina in parità. I risultati di questa analisi sono stati raccolti in un database di 178 GB disponibile in rete, all'indirizzo http://awari.cs.vu.nl/, insieme con la possibilità di ottenere varie statistiche e di competere con un programma "infallibile".
Abbiamo provato il nostro programma contro l'olandese: il nostro è riuscito a vincere sui primi livelli di gioco (corrispondenti a basse profondità di analisi) dell'altro, ma è stato costretto a soccombere ai più alti livelli, pur riuscendo a protrarre la partita per un considerevole numero di mosse! In ultimo, Ferrando l'ha dotato della possibilità di giocare contro sé stesso, per poter confrontare le prestazioni di diversi algoritmi, magari spinti a diverse profondità di analisi.
Gli algoritmi da noi specializzati per questo lavoro sono il classico *minimax* (dal teorema di von Neumann del 1928) e il suo miglioramento con la *potatura alfa-beta* (McCarthy, 1956), trattati a lezione per tutti gli allievi della classe, e infine l'ulteriore raffinamento noto come *negascout* o "ricerca a variante principale" (Reinefeld, 1983), di semplice realizzazione rispetto a procedimenti più moderni ma più sofisticati, che impiegano strutture di dati ausiliarie piuttosto complesse.
Prima però è opportuna qualche considerazione sulle caratteristiche del gioco:
- è *finito*: l'*albero di gioco* è finito, sia in ampiezza (in ogni stato il numero di mosse lecite è limitato: qui, infatti, sono 6 al massimo), sia in altezza (prima o poi la partita termina), ed
- è *a informazione perfetta*: in ogni momento, in particolare prima della scelta della mossa, ciascuno dei competitori conosce completamente lo stato del gioco – nulla è tenuto nascosto, né lasciato al caso.
Per inciso, in queste ipotesi, il gioco ammette un equilibrio. Inoltre:
- è *a due giocatori*, che muovono a turno, alternandosi, ed
- è *a somma zero*: alla fine della partita sarà determinante la differenza tra le quantità di semi accumulati nei rispettivi granai; così se ad esempio Nord vince 27 a 21 si può affermare che guadagna 3 semi, mentre Sud ne perde 3, rispetto alla situazione di parità.
Il nostro gioco è dunque *strettamente determinato*: esiste almeno un *profilo di strategie pure* (consistente in una strategia per ciascun giocatore, che gli indichi come muovere in ognuno degli stati in cui potrà trovarsi durante una partita) che costituisce un equilibrio (di Nash). In effetti, i ricercatori olandesi hanno provato che qui, delle tre possibilità già previste da Zermelo nel 1912, accade che entrambi i giocatori abbiano almeno una strategia pura che assicura il pareggio.

Il fatto che il gioco sia a somma nulla comporta che uno svantaggio di uno dei due giocatori equivalga a un vantaggio di pari entità dell'altro, e quindi ciascuno dei due cercherà di *minimizzare* la propria *massima* perdita possibile. Ciò è utile per il nostro programma, sebbene esplori l'albero di gioco soltanto fino a una certa profondità, a partire dallo *stato attuale* (ossia lo stato iniziale della ricerca, radice del sottoalbero che sarà esplorato, in cui spetta muovere al programma stesso). Occorrerà poi calcolare in modo esplicito (ed euristico) il punteggio (o *payoff*) da attribuire agli stati in cui l'esplorazione non proseguirà: banalmente, qui si può decidere che sia la differenza in semi tra il granaio del giocatore a cui spetterebbe muovere e quello dell'avversario, quindi un intero certamente ≤ 48.
Sviluppando il progetto in C++, dapprima abbiamo definito una classe `board`, un'istanza della quale può rappresentare uno stato del gioco: il tavoliere con la disposizione dei semi più l'informazione binaria che dice "a chi spetta muovere". La più semplice versione del *metodo* di analisi, ricorsivo all'interno di un ciclo, si basa sull'equivalenza max (x, y) = – min (–x, –y), e può essere così codificata:

```
int minimax (int depth, int & move) const {
  int i, value, newmove, newvalue; board C;
  if (depth == 0 || isaleaf()) { move = 0; return payoff(); }
  value = -infinity;
  for (i = 1; i <= 6 && islegal(i); i++) {
    C = *this;        // assegnamento tra board (overloaded)
    C.makemove(i);    // esegue la mossa dalla buca i sul board C
    newvalue = - C.minimax(depth - 1, newmove);
    if (newvalue > value) { move = i; value = newvalue; }
  }
  return value;
}
```

L'esplorazione lungo un ramo si ferma quando il parametro `depth` ha valore zero oppure quando il *board* corrente al quale il metodo è applicato (`*this`) è una *foglia*, ossia uno stato di fine del gioco (in entrambi i casi, il punteggio ritornato da `payoff` deve essere relativo al giocatore che dovrebbe muovere). La costante `infinity` deve essere maggiore del massimo punteggio attribuibile a un *board* (perciò > 48). Le mosse legali eseguibili nello stato corrente sono generate man mano nel *board* ausiliario `C`. Una volta valutati tutti gli stati "figli", il valore dello stato "padre" è il minimo dei valori dei figli cambiato di segno.
Un esempio di applicazione del metodo illustrato è contenuto nella seguente istruzione: `value = B.minimax(12, move);` dove il *board* `B` rappresenta l'attuale tavoliere di gioco, con il tratto al programma, e `move` è una variabile trasmessa per riferimento (è un parametro di puro *output*), il cui valore finale indicherà il numero d'ordine della buca da svuotare in `B`: `minimax` è dunque un metodo con un effetto collaterale, che si aggiunge al calcolo del punteggio associabile al *board* `B`, valore che è ritornato invece come risultato esplicito (e, nell'esempio, memorizzato nella variabile `value`). Il primo dei due argomenti espliciti, `12`, stabilisce la profondità massima da raggiungere nell'albero a partire dal nodo corrente, vale a dire il numero di semimosse di cui avanzare, a meno

che non siano trovate foglie, né intervengano *potature*, più in alto. In sintesi, una potatura evita di scendere lungo rami che non possono affinare la valutazione dello stato attuale finora fatta; miglioramenti di questo genere sono compiuti da due altri metodi, `alphabeta` e `negascout`, di cui per completezza riportiamo il codice alla pagina successiva, con qualche essenziale commento.
Si tenga comunque presente che di solito, nelle applicazioni concrete, la ricerca procede oltre la profondità massima stabilita qualora lo stato raggiunto non sia *quiescente* (ad esempio, nel gioco degli scacchi, uno stato può dirsi quiescente se non vi sono possibili catture, né minacce al Re; qui, se non vi sono buche con semi esposti a cattura). E spesso, in tanti giochi, la valutazione di uno stato, pur quiescente, non è affatto banale! In generale, inoltre, quando vi è una sola mossa sensata, è bene spingere l'analisi a una maggiore profondità – ma se nello stato attuale c'è una sola mossa legale, l'esplorazione diviene inutile!
Con la funzione delineata sopra, abbiamo ovviamente ricordato *una mossa migliore*, almeno quella (o una di quelle – nel codice riportato è la prima, e forse qui pare più opportuno valutare le mosse legali in senso orario, da 1 a 6 per Nord, ma da 6 a 1 per Sud) che il programma dovrà eseguire nello stato attuale per assicurarsi (entro l'orizzonte di semimosse fissato) almeno il punteggio ad esso attribuito al termine dell'esplorazione. Tuttavia, a parità di punteggio tra due o più nodi figli, sarebbe meglio scegliere quello che ha a sua volta il figlio con punteggio massimo per chi deve muovere; se poi persiste una situazione di parità, scegliere casualmente un'alternativa. Per procedere con un occhio attento all'efficienza, bisognerebbe in realtà tenere memoria – collezionandole in una *lista* – delle mosse migliori calcolate a turno per il programma e per l'avversario, fino all'ultimo stato al quale è giunta l'analisi del gioco, ossia ricordare il "ramo migliore" (che costituisce la cosiddetta *variante principale*) della parte di albero esplorata, o addirittura *i rami* più promettenti – nel qual caso una lista di mosse non basta e occorre una struttura di dati più raffinata.
Nelle prove fatte, abbiamo potuto constatare la *crescita esponenziale* del tempo di calcolo richiesto dal metodo `minimax`: quando la profondità di analisi aumenta di un'unità (cioè una semimossa), il tempo di attesa della prima mossa del programma si moltiplica per un fattore circa uguale a 5, e ciò corrisponde a quanto ci si poteva aspettare, poiché nella fase iniziale del gioco accade spesso che un giocatore abbia 5 mosse possibili (una delle sue buche è rimasta vuota al suo turno precedente). Per dare un'idea, la tabella sotto riporta (in secondi) i tempi di attesa della prima mossa del programma, eseguito da un *personal computer* con processore Intel Core 2 Quad Q8300, a frequenza 2.5 GHz.

depth	minimax	depth	alphabeta	negascout
8	2	13	2	1
9	4	14	3	2
10	21	15	9	6
11	105	16	19	12
12	480	17	45	29
13	2640	18	94	63

Il tempo di risposta degli algoritmi di potatura varia parecchio durante la partita.

Il metodo di "potatura alfa-beta" può essere definito nel seguente modo:

```
int alphabeta (int a, int b, int depth, int & move) const {
  // Versione originale ("fail-hard"). Precondizione: a < b
  int i, value, newmove; board C;
  if (depth == 0 || isaleaf()) { move = 0; return payoff(); }
  for (i = 1; i <= 6 && islegal(i); i++) {
    C = *this; C.makemove(i);
    value = - C.alphabeta(-b, -a, depth - 1, newmove);
    if (value > a) {
      move = i; a = value;
      if (a >= b) return a;  // potatura
    }
  }
  return a;
}
```

Esempio: `value = B.alphabeta(-infinity, +infinity, 12, move);`
L'intervallo [`a`, `b`] è detto *finestra di ricerca*: indicando con *v* l'opposto del valore attribuito a un nodo figlio del corrente, se *v* supera il corrente valore di `a` allora ne diventa il nuovo, e se raggiunge o supera anche quello di `b` allora diviene inutile esplorare i rimanenti figli (e i loro discendenti).
E questo è l'altro metodo, con stessa funzionalità e stesso impiego:

```
int negascout (int a, int b, int depth, int & move) const {
  int i, value, newmove, aa, bb; board C;
  if (depth == 0 || isaleaf()) { move = 0; return payoff(); }
  aa = -infinity; bb = b;
  for (i = 1; i <= 6 && islegal(i); i++) {
    C = *this; C.makemove(i);
    value = - C.negascout(-bb, -(a>aa?a:aa), depth - 1, newmove);
    if (value > aa) {  // la ricerca fallisce verso l'alto
      move = i;
      if (bb == b || depth <= 2) aa = value; else // nuova ricerca:
        aa = - C.negascout(-b, -value, depth - 1, newmove);
      if (aa >= b) return aa;  // normale potatura
    }
    bb = (a>aa?a:aa) + 1;  // imposta una nuova finestra nulla
  }
  return aa;
}
```

Per il primo nodo figlio usa una finestra di ricerca come il metodo `alphabeta`; per i successivi, dapprima usa una "finestra nulla" per vedere se `aa` è il risultato – il verso del fallimento indicherà come procedere…
L'efficienza di questi algoritmi si apprezza quanto più le mosse legali sono ordinate in una lista *best-first* – specialmente nel secondo caso, se queste sono almeno una ventina, e magari se si evita di analizzare più volte uno stesso stato. Comunque, il tempo di risposta del programma varia nella partita, spesso riducendosi drasticamente (specie all'inizio) rispetto all'applicazione di `minimax`.