Un programma che gioca a Mancala

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

Parole chiave: Mancala, spazio degli stati, gioco strettamente determinato a somma zero, 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 diffuse e semplici tra le oltre duecento varianti di un antichissimo gioco da tavolo africano: il Mancala. Le regole per lo svolgimento del gioco si possono reperire con facilità; per il finale, si adottò questa regola: se un giocatore rimane senza semi nel proprio campo, l'avversario non è obbligato a "dargli da mangiare", potendo, e ciò può determinare la fine della partita; così, se al suo turno egli non può più muovere per mancanza di semi, l'avversario trasferisce nel proprio granaio quelli che gli sono rimasti, e vince chi ne ha di più.
Dal 1991 alcuni informatici, segnatamente olandesi, si occupano di un'altra variante di questa famiglia, nota come Awari, con regole un po' più complicate: c'è l'obbligo di "dar da mangiare", il che può portare a situazioni di *loop*, risolte comunque dopo la terza ripetizione; e quando si finisce per ripetizione di mosse, i semi rimasti sono divisi esattamente a metà tra i due giocatori. 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 per quasi 900 miliardi di posizioni (che ne costituiscono lo *spazio degli stati*) e concludendo che, quando entrambi i giocatori 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, insieme con le possibilità di ottenere varie statistiche e di competere con un programma "infallibile", all'indirizzo http://awari.cs.vu.nl/ (dove si trovano anche le regole complete).
Ferrando ha allora aggiunto al nostro programma l'opzione per giocare secondo tali regole, e così abbiamo potuto provarlo contro l'olandese: il nostro è riuscito a vincere sui primi livelli di gioco, ma alla fine è stato costretto a soccombere ai più alti livelli, pur riuscendo a protrarre la partita per un considerevole numero di mosse! In ultimo, l'ha pure dotato della possibilità di giocare contro sé stesso, per poter confrontare le prestazioni di diversi algoritmi, eventualmente spinti a diverse profondità di analisi (ossia a diversi livelli di gioco).
Quali sono gli algoritmi da noi specializzati per questo progetto? Il classico *minimax* (von Neumann, 1928) e il suo miglioramento con la *potatura alfa-beta* (McCarthy, 1956), trattati a lezione per tutti gli allievi di quarta, e infine l'ulteriore raffinamento noto come *negascout* o ricerca a variante principale (Reinefeld, 1989), di assai semplice realizzazione rispetto a procedimenti più moderni ma più sofisticati, che impiegano strutture di dati ausiliarie piuttosto complesse.

Il nostro rientra nella categoria dei giochi *strettamente determinati*, per i quali:
- i giocatori sono due e muovono a turno, *alternandosi*;
- il gioco è *finito*: ossia, 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);
- il gioco è *a informazione perfetta*: ossia, in ogni momento, in particolare prima della scelta della mossa, entrambi i giocatori conoscono lo stato completo del gioco – nulla è tenuto nascosto o lasciato al caso.

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.
Inoltre, il gioco è *a somma nulla*, vale a dire che uno svantaggio di uno dei due giocatori equivale a un vantaggio di pari entità dell'altro (poiché alla fine sarà determinante la differenza tra le quantità di semi accumulati nei rispettivi granai); quindi il minimo guadagno di uno corrisponde al massimo dell'altro, e ciascuno dei due cercherà di *minimizzare* la propria *massima* perdita possibile.
Questa ipotesi è utile per il nostro programma, che esplora 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 tocca 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: qui, banalmente, si può decidere che sia la differenza in semi tra il granaio del giocatore a cui toccherebbe muovere e quello dell'avversario.
Sviluppando il progetto in C++, dapprima abbiamo definito una classe `board`, un'istanza della quale contiene il tavoliere di gioco con la disposizione dei semi e l'informazione binaria che dice "a chi tocca muovere". In generale, la più semplice versione del *metodo* principale (ricorsivo all'interno di un ciclo) è basata sull'equivalenza: max (x, y) = – min (–x, –y), e può essere così codificata:

```
float minimax (int depth, int & move) const {
  float value, newvalue; int i, auxmove; board C;
  if (depth == 0 || isaleaf()) { move = 0; return payoff(); }
  value = -infinity; i = 1;
  do {
    C = *this;  // assegnamento tra board (overloaded)
    C.move(i);  // esegue la i-esima mossa legale sul board C
    newvalue = - C.minimax(depth - 1, auxmove);
    if (newvalue > value) { move = i; value = newvalue; }
    i++;
  } while (i <= moves());  // ci sono ancora mosse da analizzare
  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 un *board* 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 qualsiasi *board*. Le mosse legali eseguibili su un *board* sono enumerate progressivamente a partire da `1`, e poi generate man mano nel *board* ausiliario `C`: nel nostro caso, questo compito è assai facile.
Un esempio di applicazione del metodo illustrato è contenuto nella seguente istruzione: `value = B.minimax(13, 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 mossa da eseguire 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, `13`, stabilisce il livello di profondità massima da raggiungere nell'albero a partire dal nodo corrente, 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 (chiamati `alphabeta` e `negascout`) che qui non dettaglieremo, sebbene non comportino radicali modifiche nel codice di `minimax`.
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). E spesso, in tanti giochi, la valutazione di uno stato, pur quiescente, non è affatto banale…
Abbiamo ovviamente ricordato una *mossa migliore*, almeno quella (o una di quelle – nel codice riportato è la prima) che il programma dovrà eseguire nel nodo con lo stato attuale per assicurarsi almeno il punteggio ad esso attribuito al termine della ricerca. Sarebbe meglio, tuttavia, a parità di punteggio tra due o più nodi figli, 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* o in un'altra struttura adeguata – delle mosse migliori calcolate a turno per il programma e per l'avversario, fino allo 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 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, il tempo si moltiplica per un fattore circa uguale a 5, e ciò corrisponde proprio a quanto ci si poteva aspettare. Per dare un'idea, su un *personal computer* del nostro laboratorio, giocando al livello 13, si aspetta la mossa del programma per circa 6 minuti, mentre gli algoritmi che operano potature consentono di ottenere una risposta pressoché immediata.