Un *puzzle* con le pedine della dama

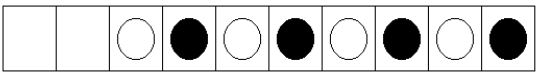
Lorenzo Repetto
Istituto Tecnico Industriale Statale "Italo Calvino" – Genova
Via Borzoli 21, 16153, repetto@calvino.ge.it

Parole chiave: *puzzle*, algoritmo di esaurimento, grafo orientato, ricerca in profondità, *backtracking*.
Classi destinatarie dell'iniziativa didattica: quarte dell'indirizzo informatico.

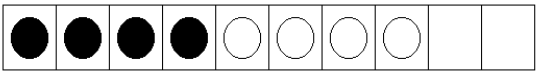
L'esperienza qui presentata vuole mostrare, attraverso lo sviluppo di un caso di studio, come si affronta più in generale la progettazione di un programma che risolve un *puzzle*, cioè un "rompicapo", che preveda una sequenza di passi o mosse per giungere alla sua soluzione. Da molti anni propongo esempi e giochi diversi ai miei studenti del quarto anno dell'indirizzo sperimentale "Abacus" – dai classici problemi di percorso del cavallo e di posizionamento delle regine su una scacchiera, o di ricerca di un'uscita da un labirinto, al gioco dell'otto (semplificazione di quello, famoso, del quindici) o al completamento di uno schema di Sudoku – con l'obiettivo di rendere loro familiare la nozione di *algoritmo di esaurimento*, che esplori "a tappeto" tutte le possibili sequenze di mosse, fino a trovarne una risolutiva se c'è – se non ce n'è alcuna, lo segnali con un opportuno messaggio, e si fermi comunque. Una tal procedura può fare una ricerca in profondità (*depth-first*) sul grafo orientato delle possibili transizioni da uno stato del gioco a un altro, impiegando la tecnica del *backtracking*: così sfrutta lo *stack* del sistema, e il codice sorgente (ricorsivo) è assai conciso.

L'idea del *backtracking* è semplice: durante l'esplorazione, se giungo in un vicolo cieco, torno indietro al primo bivio e tento un'altra strada. L'unico pericolo consiste nei "circoli viziosi": se esiste la possibilità di cadervi, allora per uscirne devo ricordare gli scenari già visti – la favola di Pollicino insegna!

Il *puzzle* che prendiamo in esame risale almeno agli anni '80 dell'Ottocento; il problema, riportato da P. G. Tait della Edinburgh Mathematical Society, fu poi generalizzato e studiato da H. Delannoy della Société Mathématique de France – come ricorda Edouard Lucas in una delle sue "Récréations mathématiques" pubblicata nel novembre del 1892, un mese dopo la sua morte.



Ad ogni mossa si spostano nelle due case libere due pedine qualsiasi, purché contigue, lasciandole nello stesso ordine. Qual è la sequenza di mosse più breve che permette di arrivare alla configurazione qui sotto riportata?



Qui c'è il serio rischio di ritornare in uno stato già visitato: banalmente, fatta una mossa, esiste subito la possibilità di tornare nello stesso stato con la mossa successiva! In termini tecnici: il grafo del gioco è *ciclico*, e quindi il suo *unfolding* è un albero con rami infiniti. In questo caso, il provvedimento più immediato e semplice da adottare è porre un limite a priori (`M`) al numero di mosse da fare, ossia alla profondità massima da raggiungere nell'esplorazione di tale albero.

Il programma in C++ delineato qui sotto procede per tentativi, secondo l'idea illustrata, fino a trovare una sequenza di al più `M` mosse risolutiva, se c'è. Il tavoliere di gioco è rappresentato da un *array* di interi (`0` = casa libera, `1` o `2` = casa occupata da una pedina bianca o nera). Se un'esecuzione di `SearchOne` arriva allo stato di successo ritorna "vero"; l'esecuzione ricorsiva chiamante stampa la mossa che aveva portato a quello stato e ritorna "vero" a sua volta… Così le mosse sono stampate in ordine inverso, ma basterà leggerle a ritroso! La mossa `i` significa che le due pedine di posti `i` e `i+1` devono essere spostate nelle due case libere; i posti sono numerati a partire da `0`, da sinistra a destra.

```
bool SearchOne (int T[], int m) {
  // T = tavoliere di gioco nel suo stato corrente
  // m = quante mosse può ancora fare
  if (success(T)) return true;       // ha trovato una soluzione!
  if (m == 0) return false;   // ha raggiunto il limite di mosse!
  int Tn[size]; // size = 2N+2, N = numero di pedine di un colore
  for (int i = 0; i < size-1; i++) // enumera le mosse da tentare
    if (T[i] > 0 && T[i+1] > 0) {          // la mossa i è lecita
      copy(T, Tn);                              // copia T in Tn
      move(Tn, i); // scambia Tn[i] e Tn[i+1] con le due case a 0
      if (SearchOne(Tn, m-1)) { // se da Tn arriva al successo...
        cout << i << " ";   // entro m-1 mosse, stampa la mossa i
        return true;   // ... e comunica a sua volta il successo!
      }
    }
  return false;  // uscita canonica dal ciclo for: a partire
}                // da T, nessuna mossa ha portato al successo!

void main () {
  int T[size];            // size costante, nel nostro esempio 10
  init(T);       // inizializza T con lo stato iniziale del gioco
  if (SearchOne(T, M)) // M costante, oppure variabile da leggere
    cout << "Leggere a ritroso la sequenza di mosse!";
  else cout << "Nessuna soluzione in " << M << " mosse o meno.";
}
```

Per avere un programma completo, rimangono soltanto da dettagliare le altre funzioni usate in questo codice.

Per alcuni *puzzle*, come il Sudoku, sarà sufficiente stampare lo schema finale, all'inizio sconosciuto. Per altri, ad esempio il giro del cavallo, l'operazione di *backtracking* è tanto semplice che diventa inutile predisporre un nuovo tavoliere in cui, ad ogni iterazione del ciclo **`for`**, copiare l'attuale e tentarvi una mossa…

Per trovare *tutte* le soluzioni in al più `M` mosse, si deve forzare il *backtracking* anche quando si trova uno stato di successo – attenzione alla complessità, che di solito è esponenziale! Ma in quel momento occorrerà stampare tutto ciò che determina la soluzione trovata: qui è la sequenza di mosse, che dovrà essere aggiornata e trasmessa a ciascuna esecuzione della procedura `SearchAll`, in un apposito parametro (di tipo "lista di interi").

```
void SearchAll (int T[], int m, IntList L) {
  // L = lista delle mosse che portano dallo stato iniziale a T
  if (success(T)) L.print(); // stampa le mosse fatte e va a capo
  else
    if (m > 0) {    // non ha ancora raggiunto il limite di mosse
      int Tn[size];           // esattamente come in SearchOne...
      for (int i = 0; i < size-1; i++)
        if (T[i] > 0 && T[i+1] > 0) {
          copy(T, Tn);
          move(Tn, i);                          // ... fino a qui!
          L.insert(i);                 // aggiunge i in testa a L
          SearchAll(Tn, m-1, L);
          L.remove();   // backtracking: toglie la testa (i) da L
        }
    }
}

void main () {
  int T[size];
  init(T);
  IntList L;   // il costruttore inizializza L come "lista vuota"
  SearchAll(T, M, L);
}
```

Se non vi sono soluzioni, questo programma termina senza stampare nulla. Se il metodo `print()` della classe `IntList` stampasse la lista rovesciata (magari ricorsivamente), non ci sarebbe più bisogno di leggere i risultati a ritroso!

Un'ultima osservazione, per mantenere lo stile di programmazione a un buon livello: qui il tavoliere di gioco è semplicemente un *array* di interi… Tuttavia, in generale, una scelta più elegante consiste nel definire una *classe* opportuna, il cui costruttore prepari una rappresentazione dello stato iniziale del gioco: lasciamo a voi lettori il compito di modificare e completare in tal senso i programmi, raccomandando particolare attenzione alle componenti (campi e metodi) pubbliche.

Eseguendo il secondo programma per vari valori di `M` e di `N` (e quindi di `size`), abbiamo scoperto che per `N` < 3 non ci sono soluzioni, per `N` = 3 ci sono due soluzioni in 4 mosse, per 4 <= `N` <= 6 c'è una sola soluzione in `N` mosse, per `N` > 6 il numero di soluzioni in `N` mosse aumenta piuttosto rapidamente… Ci siamo pure divertiti ad analizzarne alcune varianti, modificando la funzione `success`: alla fine devono esserci prima le pedine bianche e poi le nere e/o le due case libere a sinistra, come all'inizio del gioco.