Ricerca in uno spazio di stati

Il testo che segue è in parte rielaborato da "Informatica Concetti e Sperimentazioni" © Apogeo 2003.

Tra i risultati ottenuti nelle ricerche di intelligenza artificiale si annoverano quelli che permettono al computer di "giocare" ai classici giochi della nostra società, sia solitari che a più giocatori. Applicazioni di questo tipo non rivestono un'utilità immediata, ma sono molto interessanti dal punto di vista speculativo, perché permettono di saggiare i limiti delle possibilità del computer.

Se per "giocare" intendiamo la capacità di effettuare le azioni più vantaggiose, far giocare un computer non è cosa semplice. Normalmente un giocatore umano sfrutta nel giusto modo i propri punti di forza e, studiando le mosse dell'avversario, impara a conoscerne i punti deboli in modo da arrivare alla vittoria. Un computer che gioca, invece, non ha (per ora) questa capacità, ma si limita a provare un gran numero di mosse fino a trovare quella migliore, secondo una "funzione di valutazione" (cioè un criterio per valutare quanto una situazione di gioco sia vantaggiosa o svantaggiosa) impostata dal giudizio di chi lo ha programmato.

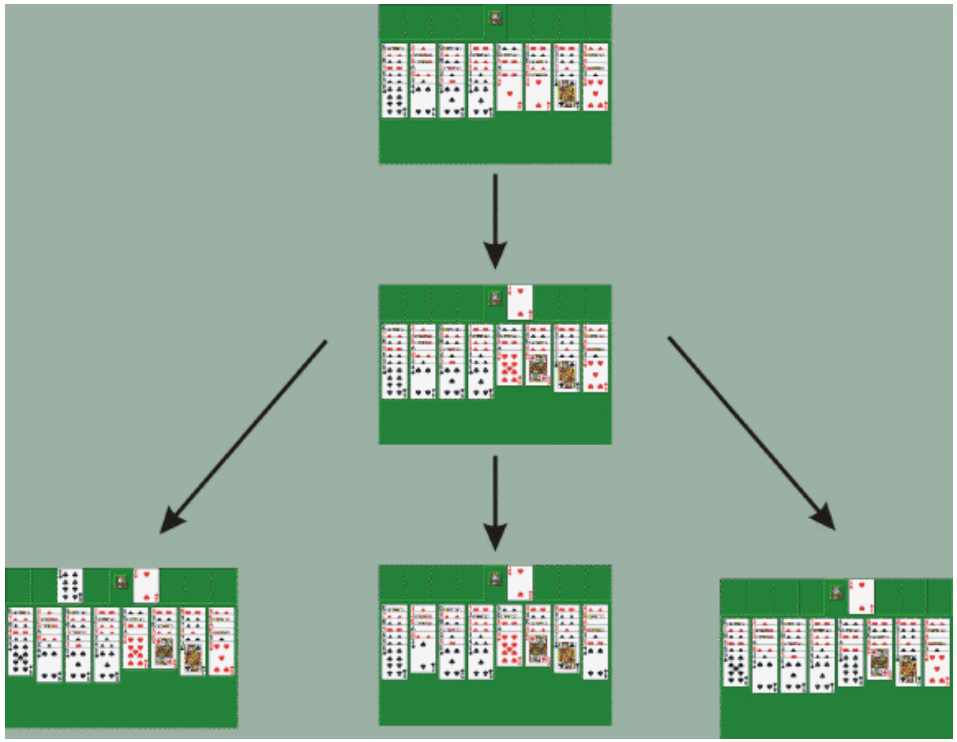
Per permettere ai computer di essere competitivi ai massimi livelli, sono usate sia particolari strategie di programmazione sia potenti architetture hardware realizzate ad hoc. Il primo computer capace di battere il campione mondiale di scacchi è stato Deep Blue, un supercalcolatore multiprocessore con hardware specializzato nel calcolare le mosse dei pezzi.

Alberi di ricerca

Per semplificare la trattazione, consideriamo intanto un gioco a completa informazione con un solo giocatore, per esempio il cubo di Rubik o il Sudoku oppure un solitario con tutte le carte scoperte sul tavolo. Una volta iniziata la partita, il giocatore umano deve scegliere una mossa tra le varie che gli si presentano. Egli guarda le carte e riflette: "Potrei spostare il fante da quella posizione a quell'altra, ma poi come posso proseguire?" Quando infine si decide e compie la mossa, la situazione sul tavolo cambia (in informatica si dice che il problema cambia "stato") e di nuovo il giocatore ha a disposizione un insieme di alternative. Il "gioco" continua fino a che non viene raggiunto l'obiettivo (in inglese **goal**), e il solitario termina con un successo, oppure non vi sono più mosse a disposizione e il solitario termina così con un fallimento.

L'evoluzione dei vari stati si può rappresentare con una struttura ad albero, in cui la sequenza delle mosse effettuate individua un percorso che dalla radice (la posizione iniziale) porta alla situazione finale (successo, fallimento o abbandono per impazienza). L'insieme degli stati connessi dalle varie mosse è detto **Albero di ricerca**.

Attenzione nei giochi in cui è possibile ritornare dopo una sequenza di mosse ad una posizione già visitata (per esempio il gioco del 15) la struttura formata dagli stati non è un **Albero** ma un **Grafo** ed esiste (anche per gli umani) la possibilità di *andare in loop*.



Albero del solitario

Il processo mentale del giocatore percorre, di mossa in mossa, un cammino nell'insieme dei possibili stati del problema. Per far giocare una macchina è necessario automatizzare e realizzare con un algoritmo questo processo mentale. Quando avremo scritto il programma, la macchina sarà in grado di giocare da sola. L'idea base consiste nell'introdurre una funzione di valutazione che assegna a ogni stato un punteggio per dare un'idea di quanto quello sia "promettente". Una buona funzione di valutazione dovrebbe agire come un oracolo permettendo alla macchina, di scelta in scelta, di trovare "magicamente" la strada giusta verso il *goal*.

L'algoritmo risultante è concettualmente molto semplice:

Ricerca nell'albero degli stati

1) **per ogni** stato che si può raggiungere con una mossa **calcola** il valore utilizzando la funzione di valutazione.
2) **scegli** una mossa che porti in uno degli stati con il valore più promettente;
3) **esegui** quella mossa;
4) **se** si è raggiunto il goal, **allora** stampa "Successo!";
5) **altrimenti se** non vi sono mosse disponibili, **allora** stampa "Fallimento!"
6) **altrimenti** (se vi sono mosse disponibili) si torna al punto 1.

Sorge spontanea la domanda: "Ottimo! Ma come si calcola la funzione di valutazione?". Alcuni degli stati del nostro gioco hanno un valore ovvio (altissimo quelli che rappresentano un successo, molto basso quelli che indicano un fallimento). Per valutare gli altri stati un giocatore umano "riflette" sul problema eseguendo a mente le mosse che "gli piacciono di più" e valutando ogni mossa in base al valore dello stato in cui essa porta. Ad esempio, se la mossa "sposta l'asso di cuori" può essere seguita da un'altra che porta al successo, allora anche "sposta l'asso di cuori" è una mossa con punteggio elevato.

Si potrebbe pensare che, una volta scritta la funzione di valutazione, per il computer sia sufficiente esplorare tutto l'albero per giocare splendidamente. Purtroppo però, per certi giochi l'albero può essere molto profondo, in alcuni casi addirittura infinito, e si rischia che l'algoritmo di valutazione

non termini in un tempo ragionevole (due milioni di anni sono decisamente eccessivi per decidere una mossa!).

La ricerca della mossa deve essere quindi limitata secondo qualche criterio per terminare in un tempo accettabile. In pratica si usa una variante dell'algoritmo appena illustrato, progettata in modo che esso, dopo essere sceso di qualche livello, si fermi dando una valutazione dello stato basata su criteri euristici legati a regole empiriche del gioco. In noi umani questo controllo si attiva quando il giocatore non è più capace di tenere a mente tutta la complessità della situazione che sta esaminando e si passa a scegliere le configurazioni che "piacciono di più".

Questo limite imposto all'estensione dell'albero in cui si fa la ricerca provoca il cosiddetto "effetto orizzonte": stati vincenti posti oltre una certa profondità non possono essere presi in considerazione. Avere un orizzonte lontano è preferibile, poiché ciò permette valutazioni più precise, ma per ottenere questo occorrono hardware più potente e/o sofisticati accorgimenti di programmazione.

Esistono tecniche che permettono di cercare in profondità nell'albero "potando" i rami secchi (le situazioni particolarmente sfavorevoli che verosimilmente non portano al successo). Quando invece un algoritmo valuta tutte le possibili combinazioni, si parla di uso della "forza bruta": viene cioè sfruttata al massimo la potenza della macchina per analizzare il numero più alto possibile di nodi anziché fare speculazioni. Questo approccio, che ha il vantaggio di non richiedere particolari studi sullo natura del problema, è però applicabile solo in determinate situazioni e con computer particolarmente potenti.

Il fatto che molte delle decisioni a cui abbiamo accennato (gli stati empiricamente buoni, la scelta della profondità da esplorare, la potatura) siano intrinsecamente vaghe e sfuggenti fa sì che la scrittura di un buon programma per "giocare" sia un'arte da programmatori dotati di molta esperienza nel campo specifico e di un notevole intuito.

Un package Java per la ricerca in uno spazio di stati

L'idea è di scrivere codice che comunica con il particolare gioco con una interfaccia in modo che il motore di ricerca sia indipendente dal gioco stesso.

Il `package engine` comunica con il gioco attraverso l'interfaccia `State` che funge da "colla" tra le classi che implementano il gioco e il motore di ricerca.

Per gestire gli stati, il motore di risoluzione usa una coda a priorità, ovvero una coda in cui l'elemento estratto non è il primo ad essere stato inserito ma quello dotato di valore più basso. Java 1.5 fornisce un'efficiente implementazione nella classe `PriorityQueue<E>`.

Il motore di ricerca (la classe `Engine`) prende uno stato iniziale e lo inserisce in coda e poi continua ad espandere lo stato più promettente fino a trovare la soluzione (o esaurire le risorse).

```java
package engine;
import java.util.*;

public interface State extends Cloneable, Iterable<State>{

/**
 *@return the iterator for successors, ordered as increasing values
 */
   Iterator<State> iterator();

/**
 * Returns the value,
       a value of 0 means the goal has been reached
 * @return value.
 */
   double value();
}
```

```
package engine;
import java.util.*;

public class Engine implements Comparator<State> {
   private PriorityQueue<State> q;

   /**
    * Number of expanded nodes
    */
   public static int expandedNodes=0;


   /**
    * limit value for queue insertion
    */
   public static int limitValue = 1000;

   /**
    * limit size for queue
    */
   public static int limitSize = 100000;
```

```java
public Engine(State seed) {
   this.expandedNodes = 0;
   this.q = new PriorityQueue<State>(limitSize, this);
   this.q.add(seed);
}

/**
 * Expand a state putting its successors in the queue
 * @return the solution state if the goal has been reached
 * null otherwise
 */
public State expand() {
   expandedNodes++;
   if (q.isEmpty()) return null;
   State item = q.poll();
   if(item.value()==0)return item;
   for (State son : item) {
      if (son.value() == 0)  return son;
      if (son.value() < limitValue) q.add(son);
   }
   return null;
}
```

```
/**
 * Perform a complete search
 * @return the solution state if the goal has been reached
 *         null otherwise
 */
public State completeSearch() {
   State son = expand();
   while (!q.isEmpty() && q.size() < limitSize && son == null)
      son = expand();
   return son;
}
/**
 * @return a string with the queue size
 *    the minimal value and the number of expanded nodes.
 */
public String toString() {
   return
      "Queue size = " +
      q.size() +", min value " +
      q.peek().value()+", expanded nodes "+expandedNodes;
}
```

```
    /**
     * Comparator for the Priority Queue
     * @return the sign of the difference arg0.value()-arg1.value()
     */
    public int compare(State arg0, State arg1) {
       return (int) Math.signum(arg0.value()-arg1.value());
    }
}
```

Una volta scritto il `package engine` si può facilmente usarlo per risolvere qualunque gioco che implementi l'interfaccia `State`. Nel seguito vediamo l'applicazione al **gioco del 15**.

Il gioco del 15

Il gioco del 15 è un vecchio passatempo molto diffuso adatto anche a bambini piccoli.

Si tratta di riordinare 15 numeri bloccati in una matrice 4×4 facendoli scorrere nella casella vuota, ne esistono anche versioni con immagini al posto dei numeri e in questo caso l'ordine è dato dalla posizione che ricompone la figura.

Per risolvere il gioco bisogna implementare l'intefaccia `State`. Noi preferiamo lavorare in due fasi. Costruire un `package game15` che implementa il gioco e poi scrivere una classe di collegamento esterna al `package` che implementa `State`.

Ecco una semplice implementazione del gioco: i numeri sono contenuti in un array di `int` con `0` che rappresenta il buco. Il valore di una configurazione è dato dal numero di elementi fuori posto.Si usa un `HashSet<State>` per memorizzare le configurazioni già toccate a cui viene dato un valore molto elevato (trasformando così il **grafo di ricerca** in un **albero**).

I costruttori permettono di creare

- la soluzione esatta se chiamati senza argomenti
- una configurazione qualsiasi se chiamati con un array di 16 interi che fornisce i numeri della tabella letti per righe
- una configurazione ottenuta da un'istanza di `Game15` applicando una mossa.

Le mosse sono rappresentate con l'intero `10*i + j` dove `(i,j)` sono le coordinate del numero da spostare.

```
package game15;
import java.util.*;

public class Game15 {

   public static final int DIM = 4;
   private int[][] T = new int[DIM][DIM];
   protected int val, i0, j0;
   private static Set<String> H = new HashSet<String> ();

   public Game15() {
      int k=1;
      for (int i = 0; i < DIM; i++)
         for (int j = 0; j < DIM; j++)
            T[i][j] = k++;
      T[DIM-1][DIM-1] = 0;
      i0 = DIM-1;
      j0 = DIM-1;
      val = 0;
   }
```

```
public Game15(int[] start) {
   int k=0;
   for (int i = 0; i < DIM; i++)
      for (int j = 0; j < DIM; j++) {
         T[i][j] = start[k++];
         if(T[i][j] == 0){i0=i; j0=j;}
   eval();
}

public Game15(Game15 G1, int move) {
   if (G1.T[G1.i0][G1.j0]!=0) throw new IllegalStateException();
   int i1 = move/10, j1 = move%10;
   if(Math.abs(G1.i0-i1)+Math.abs(G1.j0-j1)!=1)
      throw new IllegalArgumentException(" illegal move ");
   for (int i = 0; i < DIM; i++)
      for (int j = 0; j < DIM; j++) T[i][j] = G1.T[i][j];
   T[G1.i0][G1.j0]=G1.T[i1][j1];
   T[i1][j1]=0; i0=i1; j0=j1;
   eval();
}
```

```
private void eval() {
   if(H.contains(this.toLine())){val=10000000; return;};
   val=-1;
   int k = 1;
   for (int i = 0; i < DIM; i++)
      for (int j = 0; j < DIM; j++)
         if(T[i][j]!=k++)val++;
   H.add(this.toLine());
}

public double value(){
   return val;
}

public int getTable(int i, int j){
   return T[i][j];
}
```

```
public Vector<Integer> getMoves(){
     if (T[i0][j0]!=0) throw new IllegalStateException();
     Vector ls = new Vector<Integer> ();
     if (i0>0)       ls.addElement(i0-1)*10+j0);
     if (i0<DIM-1)   ls.addElement(i0+1)*10+j0);
     if (j0>0)       ls.addElement(i0*10+j0-1);
     if (j0<DIM-1)   ls.addElement(i0*10+j0+1);
     return ls;
    }

  public String toString(){
      String s="\n-----------------";
      for(int i = 0; i < DIM; i++) {
        s+="\n";
       for(int j = 0; j < DIM; j++){
           if(T[i][j]==0)  s+="    ";
           else if(T[i][j]<10)  s+="  "+T[i][j]+" ";
           else        s+=" "+T[i][j]+" ";
       }
     }
     return s+"\n-----------------";
  }
```

```
   public String toLine(){
        String s="{";
        for(int i = 0; i < DIM; i++)
         for(int j = 0; j < DIM; j++){
            s+=T[i][j];
            if(i==DIM-1 && j==DIM-1) s+="}";
            else                     s+=", ";
      }
      return s;
   }
}
```

La classe `Game15State` implementa `State` fornendo il legame con i metodi di `Game15`.

La funzione di valutazione è composta di due termini: il numero di elementi fuori posto (fornito da `Game15`) e il numero di mosse dall'inizio del gioco. Questo secondo termine, diviso per `1000` e sommato al precedente serve a privilegiare la ricerca nei livelli più alti dell'albero evitando di "infognarsi" troppo. Ovviamente nel caso di soluzione esatta il secondo termine non deve essere aggiunto.

Il metodo `iterator` trasforma il vettore delle mosse (fornito da `Game15`) in un vettore di stati ignorando gli stati già toccati (quelli per cui il valore è maggiore di `1000`).

```
import util.*;
import game15.*;
import engine.*;

public class Game15State implements State {
   Game15 G;
   int level = 0;

   public Game15State(Game15 G) {this.G=G;}

   private Game15State(Game15 G, int level) {
      this.G=G;
      this.level=level;
   }

   public Iterator<State> iterator(){
      Vector<State>  VS = new Vector<State> ();
      for(int move : G.moves()) {
         Game15 g1 = new Game15(G, move);
         if(g1.value()<1000)
            VS.addElement(new Game15State(g1, level+1));
      }
      return VS.iterator();
   }
```

```
   public double value(){
      if(G.value()==0) return 0;
      return G.value()+this.level/1000.;
   }
   public String toString(){return G.toString();}
}
```

Il programma principale si limita a creare uno stato a partire da una configurazione con tutti i numeri fuori posto e a chiamare il motore di ricerca

```
import java.util.*;
import game15.*;
import engine.*;

public class SearchGame15 {
   public static void main(String args[]) {

      State T = new Game15State(new Game15(s));
      System.out.println(T);

      Engine E = new Engine(T);
      T = E.completeSearch();

      System.out.println("======== SOLUZIONE ======");
      System.out.println(T);
      System.out.println(" expanded nodes "+Engine.expandedNodes);
   }

   final static int[]
      s ={9, 6, 1, 3, 13, 10, 5, 4, 2, 0, 8, 11, 14, 15, 12, 7};
}
```

Ecco il risultato della ricerca, la soluzione viene trovata espandendo 4167 stati.

```
----------------
  9    6    1    3
 13   10    5    4
  2         8   11
 14   15   12    7
----------------
======== SOLUZIONE ======

----------------
  1    2    3    4
  5    6    7    8
  9   10   11   12
 13   14   15
----------------
 expanded nodes 4167
```