Umano vs Computer Un esempio pratico (Parte 2)

Una delle tecniche utilizzate nei giochi a due avversari in cui uno è una macchina, è quella del min-max. Esaminiamola e costruiamone una implementazione object oriented in C++

di Francesco Sblendorio e Giovanni Vitiello

Il mese scorso abbiamo affrontato gli aspetti teorici riguardanti l'algoritmo *min-max*, una delle tecniche utilizzate per "istruire un computer" per farlo giocare contro un umano.
Ora mostreremo come implementare quanto detto in un framework di classi C++ ed applicare il tutto al famoso gioco del *tris*. Infine daremo al tutto una gradevole interfaccia grafica che fa utilizzo della libreria QT molto nota sotto Linux (vedi **Figura 1**). Rimandiamo al codice, ampiamente commentato, per ulteriori dettagli (esso è reperibile presso ftp://ftp.infomedia.it/pub/DEV./Listati)

GRAMMATICA UTILIZZATA

La grammatica utilizzata dal sistema è rappresentata nel **Riquadro 1**. Eccola dettagliata:

- **<*STATUS*>** Rappresenta uno stato del gioco; per il gioco del *tris* decidiamo di adottare una stringa in cui il primo carattere indica il turno di gioco ('x' oppure 'o') mentre i successivi nove la scacchiera linearizzata. Le costanti utilizzate sono: 'x' per il computer, 'o' per il giocatore, '_' (*underscore*) per la cella libera. Faccio un esempio: se il turno è del computer ('x') ed il giocatore umano ha posto al centro della scacchiera inizialmente vuota una 'o' lo stato sarà *"x----o----"*.

- **<*PATTERN*>** Rappresenta, appunto, un *pattern* (classe di situazioni possibili, vedi [1]) in cui le lettere maiuscole indicano *variabili* sulle quali effettuare sostituzioni tramite l'operatore di *pattern-matching*. Le minuscole e tutti gli altri caratteri indicano *costanti*. Un esempio per il gioco del *tris*: *"x-ABCDEFGH"*.
  Questo *pattern* indica che il turno è del computer ('x') e che la cella in alto a sinistra è vuota. Le altre celle possono assumere qualsiasi stato.

- **<RULE>** Rappresenta una *regola*: è formata da una parte sinistra (<*lhs*>) che specifica le condizioni di applicabilità e da una parte destra (<*rhs*>) che specifica l'azione. Esempio per il gioco del *tris*:
  x-ABCDEFGH -> oxABCDEFGH. Questa regola dice che se la prima casella è vuota ed il turno è del computer allora esso può porre il segno 'x' nella prima cella in alto a sinistra e che il turno passa all'umano ('o')

IMPLEMENTAZIONE: CLASSE *STATUS*

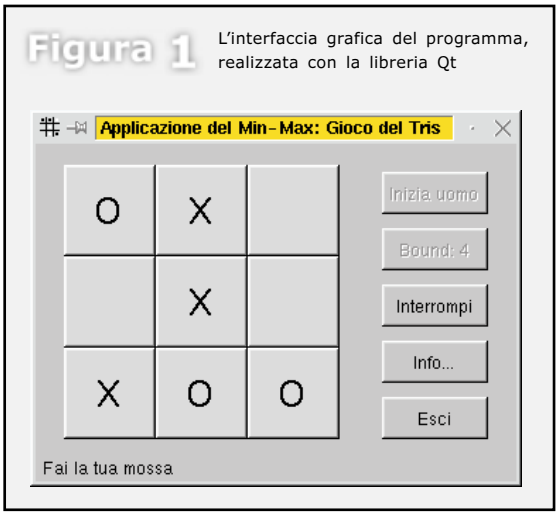
Analizziamo ora le classi di **Figura 2** esaminando per ogni classe solo i metodi più significativi (escludiamo quindi costruttori, distruttori, metodi *set* e metodi *get*). Cominciamo dalla classe *status* che rappresenta appunto uno stato del gioco: questo è memorizzato nell'attributo di tipo stringa "*situation*". Il metodo *char operator*[]*(int)* serve ad estrarne i singoli caratteri.

IMPLEMENTAZIONE DEL PATTERN MATCHING: CLASSI *PATTERN* E *VARSPACE*

Il pattern matching è realizzato tramite le due classi *pattern* e *varspace*. Quest'ultima rappresenta uno *spazio di variabili* ovvero associa un valore ad un simbolo (o variabile) ed un'istanza di questa classe è il risultato di un'operazione di *pattern matching*.

Ha un solo attributo, *vars*, che è formato da 26 elementi ovvero le lettere dell'alfabeto. Analizziamo i suoi metodi:

Figura 1 L'interfaccia grafica del programma, realizzata con la libreria Qt



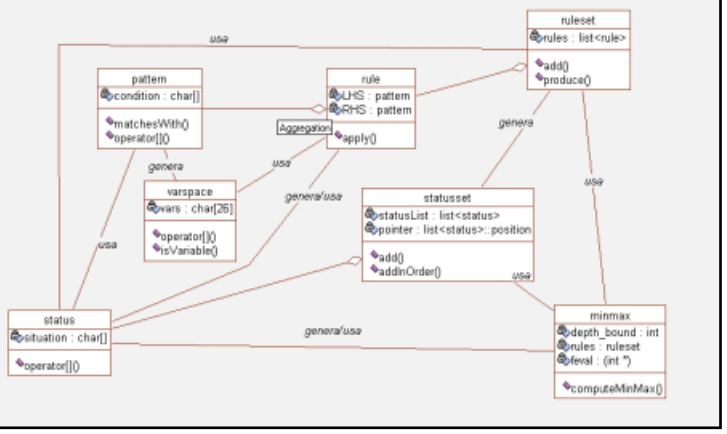
Francesco Sblendorio *è studente di Informatica presso l'Università degli Studi di Bari. Si occupa di programmazione in C/C++, Visual Basic e di problematiche legate al Web. Può essere contattato tramite e-mail all'indirizzo sblendorio@infomedia.it*

Giovanni Vitiello *è studente di Informatica presso l'Università degli Studi di Bari. I suoi interessi principali sono le tecnologie Object-Oriented e lo sviluppo di applicazioni web-based. Può essere contattato tramite e-mail all'indirizzo vitiello@infomedia.it*

Linux

Nel pattern le lettere maiuscole indicano variabili, *le minuscole e tutti gli altri caratteri indicano* costanti

Figura 2 Diagramma delle classi (semplificato) del sistema



- *bool isVariable(char c)*
  È questo metodo che stabilisce quali caratteri rappresentano le *variabili* e quali no (le *costanti*). Per come è scritto, le variabili sono le lettere maiuscole.
- *char operator* [ ]*(char c)*
  Quest'operatore restituisce il valore della variabile data in input. Ad esempio se *v* è un oggetto di classe *varspace,* allora *v*['A'] restituisce il valore della variabile 'A' dello spazio 'v'. È inoltre utilizzato per assegnare valori alle variabili (ad esempio: *v{'A'} = 'x'*). Nel caso volessimo definire diversamente le variabili dovremmo modificare, oltre all'operatore *isVariable*, anche questo.
- *void clear* ()
  Questo metodo annulla lo spazio delle variabili che risultano così non assegnate. Ciò viene realizzato ponendo a zero tutti gli elementi del vettore *vars*.

La classe *pattern* ha il solo attributo *condition* di tipo stringa sul quale vengono eseguite le operazioni.

Il metodo principale è *bool matchesWith(status st, varspace &vars)*, che è l'operatore di pattern matching: riceve in input uno stato *st* e restituisce *true* se il matching è avvenuto, *false* altrimenti. Nel caso restituisca *true* scrive nel parametro *vars* tutte assegnazioni effettuate (vedi **Riquadro 2**)

IMPLEMENTAZIONE DI UNA REGOLA: CLASSE *RULE*

La classe *rule* ha come attributi due *pattern*: *LHS* (parte sinistra) e *RHS* (parte destra). Il metodo principale è *bool apply(status initial, status &final)*. Esso riceve in input lo stato "initial", applica a questa l'operatore di pattern matching con la parte sinistra e se l'operazione è andata a buon termine, esegue l'azione specificata nella parte destra. L'operazione consiste nei tre passi menzionati nell'articolo precedente ([1]). Per i dettagli implementativi si veda il **Riquadro 3**.

IMPLEMENTAZIONE: FUNZIONE EURISTICA

La funzione euristica è implementata mediante un *puntatore a funzione*. Questo significa che l'euristica deve essere una normale funzione che abbia esattamente la seguente interfaccia:

```
int nomeFunzione(int depth, const status &actual)
```

In essa il parametro *depth* indica il livello del nodo nell'albero, mentre lo stato *actual* rappresenta lo stato a cui applicare la funzione euristica. Il valore calcolato sarà restituito come valore di ritorno.

I metodi che fanno uso di questa funzione la ricevono in input come parametro (per esempio vedi il metodo *ruleset::produce()*).

Nel caso del gioco del tris, l'euristica usata è la seguente: numero di righe, colonne, diagonali libere per la X meno il numero di righe, colonne, diagonali libere per la O; si prescinde dal parametro *depth*.

IMPLEMENTAZIONE: SET DI REGOLE DEL SISTEMA A PRODUZIONI (CLASSE *RULESET*)

Questa classe è un aggregato della classe *rule.* Viene usata come contenitore di tutte le regole del sistema a produzioni che si vuole modella-

Riquadro 1 Grammatica utilizzata dal sistema

```
<PATTERN>     ::= <sym>+          /* uno o piu' <sym> */
<sym>         ::= <var> | <const>
<var>         ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<const>       ::= /* tutti i caratteri che non sono <var> */

<STATUS>      ::= <const>+        /* una o piu' costanti */

<RULE>        ::= <lhs> <spaces> '->' <spaces> <rhs>
<lhs>         ::= <PATTERN>
<rhs>         ::= <PATTERN>
<spaces>      ::= ' '+            /* uno o piu' spazi */
```

Riquadro 2 Metodo pattern::matchesWith()

```
/** Controlla che il pattern (*this) coincida (match) con lo stato "st". In caso affermativo
                        restituisce "true" ed in "vars" pone le sostituzioni da effettuare */
bool pattern::matchesWith(const status &st, varspace &vars) const
{
   // azzera lo spazio delle variabili "vars"
   vars.clear();

   // se la lunghezza del pattern e' diversa da quella dello stato non ho match
   if (length() != st.length())
      return false;
   for (int i=0; i<length(); i++)
   {
      char c = condition[i];
      if (!vars.isVariable(c))   // se "c" non e' una variabile...
      {
         if (c != st[i])         // dev'essere presente all'i-esima pos. dello stato
            return false;        // altrimenti non ho match
      }
      else // gestisco le variabili
      {
         if (vars[c] == 0)       // se la variabile non e' ancora usata...
            vars[c] = st[i];     // ...la assegno...
         else                    // ...altrimenti controllo che sia OK
         {
            if (st[i] != vars[c])   return false;
         }
      }
   }
   return true;
}
```

re. Ha come attributo una lista di regole (la struttura dati *lista* è implementata nel file "list.h", prelevabile presso ftp.infomedia.it/pub/DEV./Listati); i metodi principali sono due ed hanno lo stesso nome, ma diverse interfacce:

```
bool produce(status origin, statusset
                                    &result);
```

```
bool produce(status origin, statusset
                    &result, int (*feval)(int, const
                                status &), int depth)
```

La prima, dato lo stato "origin" trova tutte le regole con parte sinistra che coincide (*match*), le applica e produce il set di stati "result". Restituisce "false" se non c'è alcuna regola che coincide (*match*); "true", altrimenti.

Riquadro 3 Metodo rule::apply()

```
/** Applica la regola allo stato "initial" per produrre lo stato "final";
    restituisce "true" se la parte sinistra coincide (match) con lo stato "initial", false
                                                                    altrimenti */
bool rule::apply(const status &initial, status &final)
{
   bool result = false;
   varspace variables;
   char temp[PATTERN_LEN]; // qui sara' contenuto il risultato finale
   int i = 0;

   // applico il pattern matching alla parte sinistra ed allo stato "initial"
   result = LHS.matchesWith(initial, variables);
   while ( (i < RHS.length()) && result ) // Eseguito SOLO se result=TRUE
   {
      char c = RHS[i];
      if (!variables.isVariable(c))  // se "c" e' una costante...
         temp[i] = c;                // ...la sostituisco tale e quale...
      else                           // ...altrimenti...
         temp[i] = variables[c];     // ...la sostituisco col suo valore
      i++;
   }
   temp[i] = '\0';
   final.setStatus(temp); // restituisco il nuovo stato
   return result;
}
```

La seconda opera allo stesso modo ma produce un set di stati ordinato in base alla funzione euristica. L'ordinamento è effettuato in ordine crescente se il parametro *depth* è dispari, in ordine decrescente se *depth* è pari.

IMPLEMENTAZIONE: SET DI STATI (CLASSE STATUSSET)

L'algoritmo di min-max ha bisogno di tre parametri: le regole, l'euristica ed il depth-bound

Questa classe è uno aggregato della classe *status*. Viene utilizzata dalla classe *ruleset* per produrre l'output del metodo *ruleset::produce*. Possiede metodi di aggiunta ed estrazione degli elementi.

Analizziamo in particolare il metodo *statusset::addInOrder*; il suo scopo è di aggiungere uno stato all'insieme che è rappresentato dalla classe. *addInOrder* riceve in input lo stato da aggiungere, l'euristica da applicare e la profondità del nodo che contiene lo stato in questione. Lo stato viene aggiunto non in coda, ma in modo da mantenere l'ordine crescente (se la profondità è pari) o decrescente (profondità dispari) in base alla funzione euristica fornita.

IMPLEMENTAZIONE DELLA STATEGIA MIN-MAX (CLASSE MINMAX)

Attraverso questa classe viene implementato la strategia di controllo min-max. I parametri di input sono le sue variabili private: *rules* (le regole del gioco), *feval* (la funzione euristica) e *depth_bound* (il limite di profondità oltre il quale non esploro più in profondità, vedi articolo precedente).

Il suo metodo principale è *minmax::computeMinMax* (richiamato da *minmax::play* che riceve come parametri il solo stato iniziale). Esaminiamolo con attenzione (vedi **Riquadro** 4). Gli vengono forniti in input lo stato iniziale (*actual*) e la profondità alla quale ci si trova (*depth*); in output viene fornito lo stato finale e la variabile *euristic*. La funzione *computeMinMax* è ricorsiva ed i parametri *depth* ed *euristic* servono per sapere a che profondità siamo e per scegliere lo stato da passare al livello superiore nell'albero di ricerca.

Parlando di ricorsione esami-

Riquadro 4 Metodo minmax::computeMinMax()

```
/** Calcola la funzione MinMax sullo stato attuale (actual), la profondità attuale;
restituisce la funzione euristica (calcolata quando necessario) ed il nuovo stato "result" */
status minmax::computeMinMax(const status &actual, int depth, int &euristic)
{
   status result;
   int tempeur = (*feval)(depth, actual);
   // Se ho raggiunto una foglia...
   if ((tempeur == INFINITY) || (tempeur == -INFINITY) || (depth >= depth_bound))
   {
      euristic = tempeur;
      return result; // result in questo caso è vuoto
   }
   // ...altrimenti...
   else
   {
      int bestscore;
      statusset children;
      rules.produce(actual, children, feval, depth);
      children.reset();
      if (!children.eof())
         children.readNext(result);
      children.reset();
      // Se ci sono regole applicabili...
      if (!children.isEmpty())
      {
         if ((depth % 2) == 0)      // NODO CON PROFONDITA' PARI (MAX)
            bestscore = -INFINITY;
         else                       // NODO CON PROFONDITA' DISPARI (MIN)
            bestscore = +INFINITY;
         while (!children.eof())
         {
            status child;
            children.readNext(child);
            int fvalue;     // output della funzione di valutazione euristica
            computeMinMax(child, depth+1, fvalue);
            if ((depth % 2) == 0)  // NODO CON PROFONDITA' PARI (MAX)
               { if (fvalue > bestscore) {bestscore = fvalue; result = child;} }
            else                   // NODO CON PROFONDITA' DISPARI (MIN)
               { if (fvalue < bestscore) {bestscore = fvalue; result = child;} }
         }
      }
      else // altrimenti (se non ci sono regole applicabili...)
         euristic = tempeur;
         euristic = bestscore;
      return result;
   }
}
```

L'algoritmo di min-max è ricorsivo

niamo il livello assiomatico e il passo ricorsivo; il primo si verifica quando si raggiunge una foglia ovvero:

- quando raggiungo uno stato vincente per MAX
- quando raggiungo uno stato vincente per MIN
- quando raggiungo una profondità superiore al *depth bound*

A questo punto viene retro-propagato il valore della funzione euristica calcolata nel nodo-foglia in questione. Il passo ricorsivo invece espande (mediante *ruleset::produce*) lo stato corrente. Per ognuno dei nodi generati viene richiamata ricorsivamente *computeMinMax* (facendosi restituire i valori retro-propagati).

Tra i nodi generati si sceglie quello con valore retro-propagato maggiore (o minore, in base al livello) e viene passato al livello superiore. La funzione termina quando il livello 1 restituisce la migliore prima mossa alla "radice".

CENNI ALLA REALIZZAZIONE DELL'INTERFACCIA GRAFICA

L'interfaccia grafica è stata realizzata sotto Linux con la famosa libreria Qt; la scelta è dovuta alla notevole somiglianza di questa con la forse più nota *VCL* utilizzata in *Delphi* e *C++ Builder*, di casa Borland. La documentazione ufficiale è su **[2]**.

Lo schema utilizzato è il seguente: ogni oggetto (bottone, ecc.) viene definito, come variabile globale, prima dell'oggetto destinato a contenerlo (finestra, ecc.). Quest'ultimo viene a sua volta definito come variabile globale. L'approccio non è dei più puliti, in quanto Qt mette a disposizione un modo più pulito (*signal* & *slots*), ma abbiamo preferito agire in un certo modo per mantenere una certa somiglianza con gli ambienti Borland.

Il codice non viene qui riportato ma è comunque disponibile all'indirizzo FTP citato all'inizio di questo articolo.

Vediamo, passo dopo passo, come si definisce un oggetto dell'interfaccia grafica: come avviene in *C++ Builder* bisogna derivare dalla classe base che rappresenta l'oggetto desiderato. Ad esempio se si vuole costruire un bottone, bisogna creare una classe derivata da *QPushButton*, mentre per una etichetta di testo si deriva da *QLabel*. Le inizializzazioni vanno fatte all'interno del costruttore, che ha un'interfaccia standard per ogni oggetto:

```
nomeclasse(QWidget *parent, const char
                                    *name)
```

Il parametro *parent* indica l'oggetto contenitore, mentre *name* è il nome che l'oggetto avrà internamente, che per i nostri scopi può essere tranquillamente a *NULL*.

Le classi da noi utilizzate sono le seguenti:

- **QWidget**, che rappresenta un oggetto (widget) generico, ed è usato per creare la finestra principale dell'applicazione (chiamata *FormMain*)

- **QPushButton**, che rappresenta un bottone, ed è usato per creare i bottoni "Interrompi", "Esci" ecc., ed anche per creare la scacchiera di gioco (è un array di bottoni)
- **QLabel**, che rappresenta una etichetta di testo (usata per simulare la status bar)
- **QMessageBox**, che visualizza una message box

I metodi principali da ridefinire, oltre ai costruttori, sono i metodi di gestione degli eventi; in particolare noi gestiamo l'evento *mousePressEvent*, che ha la seguente interfaccia:

```
void mousePressEvent
                (QMouseEvent *event);
```

All'interno di questo metodo (virtuale) vanno definite tutte le azioni da eseguire alla pressione di un tasto dal mouse. Per determinare il tasto premuto si usa il parametro di input "*event*" (per l'elenco delle proprietà vedi [3]). Vediamo ora brevemente quali sono i principali metodi da invocare nel costruttore:

- **setText(const char *),** imposta la "caption" dell'oggetto con la stringa passata
- **setGeometry(int x, int y, int w, int h),** imposta la posizione dell'oggetto alle coordinate (x,y) e le dimensioni (w=larghezza, h=altezza)
- **setCaption(const char *),** imposta il titolo della finestra (usato nelle classi *QWidget* e *QMessageBox*)

Nel caso del gioco del tris, l'euristica usata è la seguente: numero di righe, colonne, diagonali libere per la X meno il numero di righe, colonne, diagonali libere per la O

Purtroppo lo spazio di un articolo è limitato per descrivere approfonditamente l'argomento, per cui vi rimandiamo alla bibliografia per ulteriori approfondimenti.

CONCLUSIONI

Il sistema qui presentato ha scopi didattici e soffre di alcune limitazioni: la rappresentazione degli stati è semplice, ma non permette di usare molte variabili, ad esempio per l'implementazione di un gioco come quello degli scacchi.

Questa semplicità gli consente comunque una certa efficienza computazionale in quanto l'operatore di matching, pesantemente utilizzato, ha una complessità lineare. Futuri sviluppi potrebbero permettere di superare questa limitazione: siete tutti invitati ad approfondire l'argomento.

BIBLIOGRAFIA

[1] Francesco Sblendorio e Giovanni Vitiello, "*Umano vs Computer: la sfida (Parte I)*", DEV n.85, maggio 2001

[2] Documentazione libreria Qt: http://doc.trolltech.com

[3] Elenco alfabetico classi Qt: http://doc.trolltech.com/classes.html

[4] Elenco per argomento delle classi Qt: http://doc.trolltech.com/topicals.html