Implementazione di un interprete PROLOG, interfacciabile con il C++, per lo sviluppo di applicazioni di IA

di

Michele Padovani
Andrea Vicentini

Introduzione.

I campi di interesse dell'Intelligenza Artificiale sono oramai molteplici, spaziando dall'analisi del linguaggio naturale, alla visione e riconoscimento automatico, all'apprendimento da esempi, alle prove automatiche di teoremi e analisi logica in generale, ai Sistemi Esperti (diagnosi medica), fino alla Teoria dei Giochi.

Esistono due approcci all'IA: quello dei filosofi, psicologi, linguisti, finalizzato alla modellazione del comportamento umano e ai suoi processi di informazione; quello ingegneristico, relativo alla emulazione selettiva dell'attività umana di ragionamento, e sicuramente più interessante nell'ambito del nostro corso di studi.

Con lo sviluppo di questa materia, e con l'ormai universale riconoscimento delle sue possibili applicazioni nei vari settori industriali e di ricerca, sono nati diversi ambienti e linguaggi di sviluppo a livello software, di natura diversa rispetto ai più tradizionali linguaggi imperativi (COBOL, BASIC, FORTRAN): è avuta infatti un'evoluzione degli stessi linguaggi imperativi verso tecniche object-oriented da un lato; dall'altro c'è stata la nascita vera e propria di linguaggi funzionali (LISP) e logici (PROLOG) di portata abbastanza vasta, e di ambienti di sviluppo (sistemi esperti) per problemi specifici (MYCIN, ...). Ma in che senso questi nuovi software sono strumenti dell'IA?

Un sistema di IA "classico" generalmente, per trovare una soluzione dinamica ad un dato problema, ha bisogno di una vasta base di conoscenza da utilizzare ed eventualmente modificare; non può inoltre prescindere dall'utilizzo di una sua logica di interpretazione e di derivazione di nuova conoscenza: un linguaggio dichiarativo come il PROLOG va incontro alle esigenze di rappresentazione della conoscenza, sotto forma di fatti e regole (relazioni); inoltre, pur con dei limiti che hanno richiamato notevoli critiche da più parti, è anche dotato di un'ottima capacità logica, basata sulle clausole del prim'ordine. Ha però anche diversi limiti, evidenziati ad esempio da alcune applicazioni di IA, che richiederebbero l'uso dei predicati del second'ordine, di capaci librerie matematiche, oppure in generale di un flusso di esecuzione sequenziale, tipico dei linguaggi imperativi; d'altro canto, questi stessi linguaggi non hanno le capacità di ricerca automatica di una soluzione.

Soluzione ideale sarebbe forse un'integrazione, così da renderli più duttili senza perdere alcuna delle caratteristiche di interesse.

È proprio da questo genere di considerazioni che è nata la nostra idea: partendo dal PROLOG abbiamo cercato di fornirlo di strumenti operativi potenti, da affiancare alla sua capacità logica: abbiamo voluto fondere insieme l'aspetto imperativo e dichiarativo della programmazione, in un unico

strumento in grado di coprire una più vasta gamma di applicazioni di IA, o comunque di aumentare l'efficienza dei meccanismi classici.

Seguirà quindi dapprima una descrizione tecnica e algoritmica di questo oggetto, con tanto di "istruzioni per l'uso"; poi passeremo ad analizzare tutte le nuove possibilità offerteci nel campo delle applicazioni di IA; quindi concluderemo con una di queste applicazioni, effettivamente realizzata da noi: il gioco del TRIS.

Interprete PROLOG interfacciato con C++.

Abbiamo sviluppato una libreria in C che implementa un interprete PROLOG; abbiamo quindi realizzato, in C++, un modulo di definizione dell'oggetto PROLOG, così da consentirne anche un'istanziazione multipla: in altre parole, all'interno di uno stesso sorgente C++, siamo in grado di dichiarare più oggetti distinti, interagenti tra loro, che fanno riferimento a sorgenti PROLOG diversi, ma allo stesso motore di inferenza e allo stesso tipo di gestione del sorgente PROLOG.

Oggetto PROLOG

Al momento dell'istanziazione di un nuovo oggetto, è necessario indicare il sorgente PROLOG corrispondente: volendo ad esempio riferire l'oggetto **prolog *p** al sorgente "prova.pro":

```
prolog *p = new prolog("prova.pro");
```

Come si può vedere dai listati allegati, l'oggetto mantiene tutte le informazioni relative all'evoluzione dello stato dell'interprete: sorgente PROLOG, variabili, stack degli stati, e consente di effettuare le query attraverso la funzione *solve()*.

Motore di inferenza e gestione sorgente PROLOG

Il sorgente PROLOG, una volta letto da file, viene memorizzato in una struttura dinamica (tipo DB), realizzata come una coda, individuata univocamente dal puntatore al primo elemento, che vedremo essere una variabile **privata** dell'oggetto PROLOG.

Il motore di inferenza ha il compito, di fronte ad una query, di ricercare tra il sorgente PROLOG regole o fatti che consentano di dare le eventuali soluzioni alla stessa. Affiancato ad un algoritmo di esplorazione ricorsiva del sorgente, è presente un meccanismo di memorizzazione dello stato dinamico del motore di ricerca (modulo 'status'), che gestisce lo stack dei sottogoal, facendo uso della struttura dati STATUS. Inoltre fondamentali risultano l'algoritmo di unificazione e il modulo di gestione dinamica delle variabili.

Queste operazioni vengono effettuate con l'ausilio di strutture dati ad hoc (ADT): la più significativa è sicuramente la OBJ, che memorizza dinamicamente ogni termine o predicato, in maniera ricorsiva.

La prima innovazione da noi introdotta è la possibilità, da parte dell'utente, di realizzare predicati predefiniti personalizzati, scritti in linguaggio C (e quindi con tutte le potenzialità relative), trattati dall'interprete PROLOG alla stregua di quelli ordinari. Quest'ultimo viene istruito sulla loro esistenza tramite la macro *enable_predef*(), che abilita la lettura della tabella dei predicati personalizzati, rendendoli visibili a qualunque oggetto PROLOG istanziato. La tabella delle dichiarazioni associa al nome del predicato riconosciuto dall'interprete, una funzione C, con la seguente *signature*:

```
int <nome funzione>(OBJ* st);
```

dove st è un puntatore ad una struttura, che conterrà il nome del predicato e gli argomenti passati.

É importante notare come operazioni molto sofisticate di interazione con le strutture dati (p.e. la lettura e la modifica delle variabili PROLOG) richiedano una conoscenza approfondita del funzionamento dell'interprete; risultano invece decisamente semplici ed efficaci operazioni ad esempio di natura grafica, o che in generale non comportino uno scambio di informazioni dinamiche.

Per un esempio di utilizzo si veda l'ultima sezione.

Applicazioni possibili nel campo di IA.

É importante premettere che il nostro lavoro non è di per sé un'applicazione di IA, nel senso che non è specializzato alla risoluzione di un problema particolare, si propone invece come mezzo, a disposizione di chi desideri conciliare aspetti diversi in applicazioni di IA.

L'allargamento di prospettiva consentito da questo nuovo strumento ha ragion d'essere, nel momento in cui risulti tangibile, a livello applicativo, il miglioramento di progetti già esistenti, nati in ambienti più ristretti, quali il PROLOG o il C++. Si pensi solo al possibile aumento di efficienza e di funzionalità dell'ambiente PROLOG, una volta corredato da una libreria che gli consenta di accedere a funzioni C++: qualunque progetto di IA scritto in PROLOG può essere velocemente ottimizzato per quanto riguarda la parte di INPUT/OUTPUT e computazionale.

Lo sviluppo di un'applicazione di IA in PROLOG risente fortemente della mancanza di una modularità di questo ambiente, che é tipica del C++ ad esempio; ecco quindi che si delinea una nuova linea innovativa: la scrittura di sorgenti PROLOG distinti e completamente indipendenti tra loro, la cui interazione é gestita dai moduli C++, nei quali vengono istanziati, ed interrogati, tanti oggetti diversi quanti sono i sorgenti di interesse (si veda la

gestione di 2 giocatori diversi nella prossima sezione). Il risultato é che abbiamo affiancato alla modularità del C++ quella del PROLOG, seppur subordinata.

É opinione diffusa che la mancanza di un vero e proprio assegnamento sia al contempo vanto e mancanza di un linguaggio dichiarativo: il nostro interprete ha le potenzialità per manipolare arbitrariamente i valori di una variabile, anche se già istanziate, mediante l'implementazione di un opportuno predicato predefinito. É evidente che l'assegnamento é realizzabile a tutti gli effetti, anche se é richiesta molta attenzione nella gestione di simili procedimenti, che rischiano di rendere incontrollabile il motore di inferenza.

Si può inoltre pensare, sfruttando al solito il C++, di dotare il PROLOG di tipi di dati diversi: si pensi ad esempio alla gestione di file e/o di record.

Nel campo dei giochi é ovvia la grande utilità di questo oggetto: abbiamo potuto notare l'estrema difficoltà incontrata da alcuni programmatori PROLOG per la realizzazione di giochi (come "Forza 4"), che richiedevano un interfacciamento con l'utente non banale. La stessa applicazione proposta da noi nella prossima sezione é un esempio di come invece il problema può essere risolto in maniera elegante ed efficace.

Realizzazione di giocatori di TRIS virtuali.

Premettiamo che quest'applicazione non ha pretese di ottimalità pratiche: non ci siamo interessati alle ottimizzazioni del tempo computazionale e dell'euristica; piuttosto é un esempio significativo di utilizzo del nostro sistema.

Abbiamo creato un piano virtuale di gioco, realizzato in C++, che gestisce le mosse di due giocatori, corrispondenti ad altrettanti oggetti PROLOG, introdotti dalle seguenti definizioni:

```
prolog *p1 = new prolog("tris0.pro");
prolog *p2 = new prolog("tris1.pro");
```

Si noti come i due oggetti facciano riferimento a due sorgenti prolog diversi, quindi a euristiche di gioco diverse, e di come in teoria non ci sia limite alla definizione di nuovi giocatori.

Abbiamo anche introdotto un oggetto arbitro, il cui ruolo é quello di verificare se uno dei due giocatori ha vinto (ma potrebbe essere esteso ad altri controlli):

```
prolog *referee = new prolog("rules.pro");
```

Creati questi tre oggetti, il compito dell'interfaccia C++ é semplicemente quello di interrogare i due giocatori, uno dopo l'altro, sulla mossa che intendono effettuare, istruendoli sullo stato attuale della scacchiera, e sulla pedina da muovere:

```
p1->solve("move([ e, e, e, e, e, e, e, e, e ], x, M).
");
```

Di fronte a questa query, p1 valuterà il posizionamento migliore della pedina 'x' sulla scacchiera vuota (rappresentata dalla lista di caselle vuote 'e'), e la sua variabile M verrà unificata alla nuova configurazione della scacchiera (in realtà avviene anche la stampa, ma rimandiamo al seguito questo discorso). A questo punto basterà interrogare p1 per conoscere la mossa già effettuata, con una semplice operazione:

```
Newboard = p1->sget_var("M");
```

Ora toccherà all'arbitro verificare se p1 ha vinto:

```
if (referee->solve("win(<Newboard>, x).") ) < ha
vinto p1 >;
```

Sarà quindi il turno di p2:

```
p2->solve( "move(<Newboard>, o, M)." );
```

E il gioco prosegue fino a quando uno dei due giocatori vince, oppure non muove più perché la scacchiera é piena.

Per ora non abbiamo ancora fatto riferimento al comportamento seguito dai giocatori e dall'arbitro: questo é frutto della base di conoscenza introdotta tramite i sorgenti PROLOG cui sono legati. Analizziamo prima quella dei giocatori (*TRIS0.PRO e TRIS1.PRO),* quindi quella dell'arbitro (*RULES.PRO).*

TRIS0.PRO e TRIS1.PRO

Sono entrambe casi particolari della più generale *TRISN.PRO*: si tratta di un'applicazione dell'algoritmo minimax, con profondità di esplorazione dell'albero *n*, valore passato come parametro al predicato *move* /4. Il PROLOG si presta ottimamente alla esplorazione ricorsiva dell'albero, ma quando si giunge alla valutazione numerica delle foglie, viene frenato dalla sua scarsa capacità computazionale. Abbiamo quindi introdotto un predicato predefinito da noi, *eval* /3, realizzato in C, in grado di analizzare ed etichettare, in maniera 'imperativa', lo stato della scacchiera in esame. Questo predicato *eval* /3, che richiede obbligatoriamente una variabile non istanziata come terzo parametro, restituisce tre tipi di valori, relativi a vittoria certa, sconfitta certa o indifferenza.

Abbiamo anche introdotto un secondo predicato predefinito, *o_tris* /1, realizzato molto semplicemente in C, che visualizza sul video lo stato della scacchiera passatogli come parametro (si noti come ottenere lo stesso risultato in PROLOG sarebbe stato più tortuoso).

RULES.PRO

C'é una sola regola, *win*(S,P), che controlla se S é uno stato vincente per il giocatore P, avvalendosi del predicato già discusso *eval* /3. Senza eccessiva difficoltà si potrebbero aggiungere controlli sulla legittimità di una mossa, e possibilità di dare giudizi.