Umano vs Computer La sfida
(Parte 1)

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

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*

Chissà quante volte siamo stati battuti dal "computer" in una partita a scacchi o giochi simili e ci siamo chiesti come esso è in grado di prendere delle decisioni. In questa serie di articoli rispondiamo a questa domanda. Prima però diamo uno sguardo a delle basi teoriche che ci permetteranno di capire a fondo questi meccanismi.

SISTEMI GUIDATI DA PATTERN

Molte applicazioni dell'intelligenza artificiale, a differenza delle applicazioni tradizionali, non seguono uno schema procedurale: in esse non c'è bisogno di specificare passo dopo passo le operazioni da compiere, ma piuttosto seguire una logica cosiddetta *pattern-directed*.

Tale logica consente, in base ad una situazione corrente ed alle regole conosciute, di generare nuove situazioni finché non viene raggiunta una condizione di terminazione. Il termine *pattern* indica una classe di situazioni possibili che hanno qualcosa in comune.

Un sistema *pattern-directed* è formato da:

- una serie di moduli (chiamati *pdm*) che possono essere attivati dai *pattern;*
- lo stato (o situazione) corrente del sistema (chiamato *database globale*);
- una strategia di controllo (l'algoritmo) che gestisce l'attivazione e la selezione dei *pdm.*

Chissà quante volte siamo stati battuti dal "computer" in una partita a scacchi o giochi simili

IL PROBLEMA DEL PATTERN MATCHING

L'operazione fondamentale per la risoluzione di un problema di intelligenza artificiale è quella denominata *pattern matching*. Questa, dati in input una situazione ed un pattern, fornisce in output:

1. una risposta booleana che dice se la situazione data fa parte della classe di situazione rappresentata dal pattern;
2. una serie di sostituzioni da effettuare sul pattern per renderlo uguale alla situazione data.

Chiariamo il tutto con un esempio (usiamo una rappresentazione a stringhe delle situazioni e dei *pattern* e indichiamo con lettere maiuscole le variabili e con le minuscole le costanti):

Situazione:	"abbcdebb"
Pattern 1:	"XZYcdeYY"
Pattern 2:	"aXXYdeXZ"
Pattern 3:	"XYZcZebb"

In questo esempio i pattern 1 e 2 "*matchano*" (ovvero sono confrontabili) con la situazione, mentre il pattern 3 no.

Infatti applicando l'operatore di *pattern matching* alla situazione ed al pattern 1 ricaviamo la sostituzione "X=a, Y=b, Z=b", per il pattern 2 ho "X=b, Y=c, Z=b", mentre per il pattern 3 avrei "Z=b" e contemporaneamente "Z=d", che è contraddittorio.

SISTEMI A PRODUZIONI

Un sistema a produzioni è un particolare tipo di sistema *pattern-directed* in cui i *pdm* sono detti *regole* e sono costituiti da coppie *antecedente-conseguente* ove l'antecedente è un *pattern*. Nei sistemi a produzioni separiamo la fase di esame dei dati da quella di modifica. La prima fase ha luogo nell'*antecedente* (detto anche LHS ovvero **L**eft **H**and **S**ide, parte sinistra della regola) mentre la seconda ha luogo nel *conseguente* (RHS, **R**ight **H**and **S**ide, parte destra delle regola). Il funzionamento di questi sistemi si basa su un ciclo denominato *recognize-act* che consiste in:

1. ricercare l'insieme delle regole applicabili allo stato corrente;
2. scegliere quale applicare tramite la *strategia di controllo;*
3. eseguire le azioni che la regola prevede nella parte destra.

Linux

Facciamo un esempio. Supponiamo di avere le seguenti regole (*regole di produzione*):

```
ABBAd -> BBAAh
ABBCd -> CBBBh
ArtBC -> chABC
```

Supponiamo anche che lo stato corrente sia rappresentato dalla stringa "ottod". L'insieme delle regole applicabili è costituito dalle prime due e supponiamo che la strategia di controllo selezioni la prima. Un'azione consiste nei seguenti passi:

1. pattern matching dello stato corrente con la parte sinistra della regola;
2. applicazione della sostituzione ottenuta nel passo precedente alla parte destra;
3. il nuovo stato corrente diventa la parte destra con le sostituzioni appena effettuate.

Nel nostro caso eseguendo l'azione specificata dalla regola selezionata (la prima) ottengo la stringa "ttooh".

Scopo dell'algoritmo min-max è quello di fornire la "migliore prima mossa successiva", avendo in input una certa configurazione di gioco

STRATEGIA DI CONTROLLO: ALGORITMO DEL MIN-MAX

Nei giochi a due avversari che utilizzano sistemi *pattern-directed* la strategia di controllo principale è chiamata *min-max*; Questa strategia fa uso di un altro concetto, quello di *funzione euristica*. Tale funzione ha come argomento uno stato del sistema e restituisce un valore numerico che rappresenta la "bontà" dello stato relativa al giocatore.

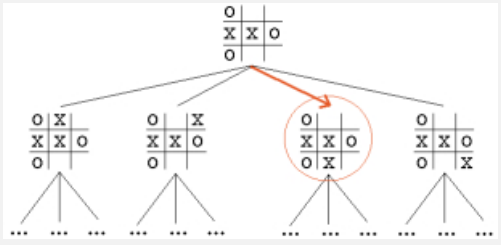
Per convenzione, tanto più grande è questo valore, tanto più la situazione è vantaggiosa per il "giocatore computer" (chiamato convenzionalmente *max*). Al contrario più la situazione è vantaggiosa per il "giocatore umano" (chiamato *min*) tanto più piccolo è questo valore. Nel caso in cui la situazione sia vincente per *max*, la funzione varrà "+infinito"; in caso contrario, se la situazione è vincente per *min*, varrà "-infinito".

(Faccio notare come per situazione vantaggiosa *non* si intende che il giocatore abbia vinto. In tal caso, si dice appunto che la situazione è *vincente*.)

Lo scopo dell'algoritmo di *min-max* è quello di fornire la migliore prima mossa successiva avendo in input una certa configurazione di gioco. (vedi **Figura 1**).

Questo algoritmo utilizza tutti i concetti precedentemente esposti: il *database globale*, che rappresenta la situazione del gioco in un determinato istante e le *regole*, che rappresentano

Figura 1 La prima mossa migliore, selezionata nell'albero di tutte quelle "possibili"



le mosse e la *funzione euristica*.

La strategia min-max, partendo da uno stato iniziale, attiva tutte le regole ad esso applicabili generando nuovi stati-figli, per poi ripetere il procedimento ricorsivamente su tutti questi stati così generati. Procedendo in questo modo, viene prodotto un albero su cui viene effettuata la ricerca della migliore prima mossa. La crescita di quest'albero è esponenziale; se non si ponesse un limite di profondità i costi della generazione, sia in termini di spazio che in termini di tempo, sarebbero insostenibili.

Per ovviare a questo problema si impone un limite di profondità, detto *depth bound*.

Notiamo che ai livelli pari dell'albero corrispondono turni di gioco per *max*, mentre ai livelli dispari turni di gioco per *min* (la radice è compresa, in quanto è al livello 0).

Conseguentemente, i successori di uno stato in cui il turno è di *max* saranno ordinati in modo decrescente in base alla funzione euristica, in modo tale che siano esaminati per primi gli stati migliori; lo stesso avverrà per i turni di gioco

Figura 2 Ordinamento dei nodi in base al livello (pari o dispari)

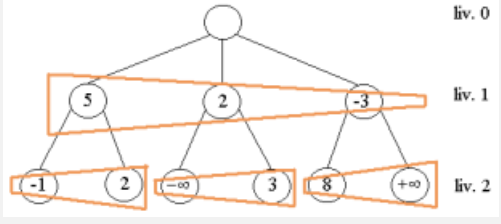
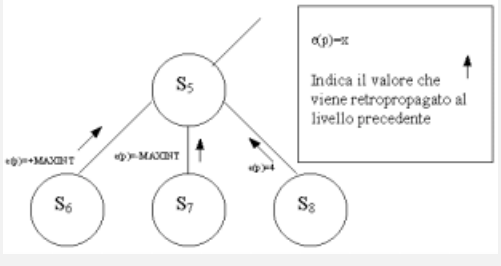


Figura 3 Procedere del valore retropropagato durante l'esecuzione dell'algoritmo min-max



di *min*, ovvero i livelli dispari dell'albero (vedi **Figura 2**) in cui l'ordinamento viene effettuato in modo crescente.

È fondamentale la definizione della "funzione euristica"

Non abbiamo però ancora spiegato come faccia l'algoritmo a scegliere la migliore prima mossa; viene effettuata un post-visita **[1]** dell'albero in questo modo (vedi **Figura 3**):

1. Se il nodo esaminato è foglia gli viene assegnato un valore pari alla *funzione euristica* qui calcolata; altrimenti...
2. ...se il nodo esaminato è un nodo *max*, gli viene assegnato il valore massimo tra quelli assegnati ai suoi figli; lo stesso vale per *min* (il valore minimo)
3. Tutto ciò va ripetuto ricorsivamente finché non viene assegnato un valore alla radice. A questo punto la migliore prima mossa è quella che collega la radice con lo stato che ha "retropropagato" il suo valore alla radice

La massimizzazione del secondo punto si spiega col fatto che *max* deve massimizzare l'euristica. Attenzione però: non si tratta di un'euristica *locale* come quella utilizzata durante la generazione dell'albero (un'euristica *locale* è relativa solo allo stato attuale, non tenendo conto di nessun altro fattore); si tratta invece di un'euristica che tiene conto di tutta una serie di mosse che vengono simulate da un intero percorso nell'albero (si noti che la *funzione euristica* viene calcolata solo sulle foglie).

CONCLUSIONI

Questa tecnica è abbastanza generale per i giochi a due avversari, ma soffre di alcune limitazioni: ad esempio in giochi come gli scacchi è impensabile esplodere tutte le configurazioni di gioco possibili dato il loro numero elevato (dell'ordine di 35^{100}). In questi casi si utilizzano delle varianti dell'algoritmo che operano una potatura euristica dell'albero di ricerca; una di queste è *alpha-beta potato* **[2]**.

Questa tecnica è abbastanza generale per i giochi a due avversari, ma soffre di alcune limitazioni

Dopo avervi annoiato con tutta questa teoria nel prossimo articolo vedremo dettagliatamente una realizzazione pratica di quanto detto, applicando il tutto al famoso gioco del *tris*.

BIBLIOGRAFIA

[1] A.A.Bertossi, "Strutture, Algoritmi, Complessità", ECIG, ISBN 88-7545-370-5

[2] http://yoda.cis.temple.edu:8080/UGAIWWW/lectures96/search/minimax/alpha-beta.html