

Università degli Studi di Modena e Reggio Emilia
Facoltà di Scienze della Comunicazione e dell'Economia

Votazioni in Latticeworld

*Un automa cellulare per lo studio dell'influenza
dei legami sociali sulle scelte elettorali*

Informatica I
Anno Accademico 2005/06
prof. Marco Villani

Relazione a cura di:

Fabio Ruini - matricola nr. 7496 (LS EGRI)
Gaia Guerzoni – matricola nr. 11810 (LS EGRI)

Latticeworld: il funzionamento

Il modello che verrà descritto nelle prossime pagine, denominato “Latticeworld”, è un automa cellulare “di stampo politico”. In questo modello, il “mondo” è rappresentato da una matrice di N righe ed M colonne, ciascuna delle quali occupata da un individuo. Ognuno di questi individui ha, nel momento in cui il sistema inizia ad evolvere, un proprio credo politico, scelto tra due ideologie esistenti e rappresentato dal valore 0 od 1. L’evoluzione avviene a tempi discreti (da $t=0$ a $t+1$, $t+2$, ... $t+nr_passi$) e, ad ogni passo, ciascun individuo aggiorna la propria convinzione politica. Tutti lo fanno in maniera perfettamente sincrona, basandosi sulle convinzioni dei propri vicini, ossia gli individui con i quali essi hanno presumibilmente una più frequente interazione sociale. Se tra i vicini di un individuo (ossia tra coloro che costituiscono il suo “intorno”) ve n’è almeno un certo numero avente orientamento 1, allora anche l’individuo diventerà (o continuerà ad essere) un sostenitore dell’idea 1. Al contrario, se il valore di soglia non dovesse venire raggiunto, l’individuo diventerà (o rimarrà) sostenitore dell’idea 0. Il mondo considerato è toroidale, in maniera tale che ogni individuo abbia lo stesso numero di individui (ossia la stessa dimensione dell’intorno).

Un modello concettualmente semplice come questo, permette comunque di osservare molti fenomeni interessanti e comportamenti “globali” che emergono spontaneamente grazie alla sinergia “locale” dei singoli individui.

Il workflow del programma

Rispetto alla versione “liscia” del programma, sono state apportate tutte le modifiche suggerite nella sezione “successive complicazioni” del testo d’esame. Il funzionamento del programma che ne deriva è schematizzato nel diagramma di flusso allegato in appendice.

Le strutture dati utilizzate

Essendo il programma piuttosto lineare e semplice in quanto a struttura, non è stato necessario ricorrere a strutture dati particolarmente complesse¹. L’intero software ruota attorno a due variabili principali, che sono le matrici A e B. In linguaggio C, queste due matrici sono state definite con le istruzioni:

- `int matrice_A[nr_righe][nr_colonne];`
- `int matrice_B[nr_righe][nr_colonne];`

Come è possibile osservare da questo brevissimo frammento di codice, le due matrici vengono definite all’interno del programma con dimensioni non-fisse. Questa soluzione, sebbene un po’ laboriosa, permette all’utente di utilizzare matrici di qualsiasi dimensione² (facendole creare in maniera random al programma, oppure inserendo manualmente ogni valore all’interno del file “input.dat”) e di non essere vincolato ad un particolare ordine matriciale.

Come abbiamo appena accennato, il programma si appoggia anche ad alcuni files (tutti semplicissimi plain-text) esterni. Questi sono nel dettaglio:

¹ Ciò non toglie, ovviamente, che al programma che abbiamo scritto sia potenzialmente possibile apportare innumerevoli miglioramenti, alcuni dei quali sicuramente implementabili attraverso il ricorso a strutture dati maggiormente complesse.

² L’unico limite che è stato imposto sull’ordine delle matrici è che il numero di righe e di colonne sia compreso nell’intervallo [1,50]. Il tetto massimo di 50 righe/colonne è stato fissato esclusivamente per essere certi che la visualizzazione a video fosse sempre agevole.

- “*parametri.dat*”: file contenente un particolare valore su ogni riga. Nello specifico:
 - riga 1: numero di righe della matrice da creare in maniera random;
 - riga 2: numero di colonne della stessa matrice;
 - riga 3: valore di soglia da utilizzare durante l’aggiornamento della matrice;
 - riga 4: tempo di attesa tra la visualizzazione di una matrice e la successiva;
 - riga 5: numero di step per i quali far evolvere il sistema;
 - riga 6: comportamento da adottare per l’aggiornamento, nel caso di uguaglianza tra il numero di vicini con valore 1 ed il valore di soglia. Se il valore contenuto in riga 6 è 0, allora la cella manterrà il valore corrente; se il valore del file è 1, la cella assumerà con probabilità 0.5 il valore 0 o il valore 1.
- “*input.dat*”: file contenente la matrice di partenza, da utilizzarsi nel caso in cui l’utente decida di non crearla in maniera casuale;
- “*output.dat*”: file contenente la matrice A al termine dell’evoluzione. Nel caso in cui il programma termini prima del numero di passi prefissati, all’interno del file output.dat verrà memorizzato l’ultimo stato evolutivo raggiunto dalla matrice;
- “*storia.dat*”: file “riepilogativo”, contenente al suo interno, per ogni step evolutivo, il numero di celle con valore 1 presenti nel sistema.

Le funzioni utilizzate

Al fine di rendere più chiara la struttura del programma, nonché per evitare inutili e dispendiose (in termini di occupazione di memoria) ripetizioni di codice, le istruzioni richiamate più frequentemente sono state scritte sotto forma di funzioni. Queste funzioni sono nel dettaglio:

- `int` `calcola_valore_random(float prob)`

La funzione “`calcola_valore_random`”, ricevuto in input un valore reale “`prob`”, restituisce in output il valore intero 0 con probabilità uguale a `prob`, mentre restituisce il valore intero 1 con probabilità `1-prob`. Se `prob` è uguale a 0.5, la funzione restituisce con la medesima probabilità il valore 0 o 1.

- `void` `attendi(long int tempo)` {

Ricevuto in input un valore intero “`tempo`”, la funzione “`attendi`” esegue un semplice calcolo (nel caso specifico, questo calcolo è $j=j+1$, dove j è una variabile intera inizializzata al valore 0) per `tempo`-volte. Scopo di questa funzione è quello di rallentare (ma non bloccare) l’esecuzione del programma, in maniera tale da consentire all’utente di prendere consapevolmente visione, ad ogni step temporale, dell’evoluzione del mondo matriciale³.

- `int` `visualizza_matrice(int nr_righe, int nr_colonne, int A[][nr_colonne], char tipo, int step, long attesa)`

³ Il valore di “`tempo`” ricevuto in input dipende fortemente dal calcolatore su cui è in esecuzione la simulazione. Durante le nostre prove, ci siamo accorti che un valore uguale a 30000000 è più che sufficiente su un iBook con sistema operativo Mac OS X 10.4.4, processore PowerPC G4 ad 1.33 GHz ed un GB di memoria RAM. Su un calcolatore con installato Windows XP, processore Pentium 4 a 3 GHz ed un GB di memoria RAM, un tempo accettabile è 60000000.

Questa funzione si occupa della visualizzazione a video della matrice, di dimensioni $nr_righe \times nr_colonne$, ricevuta in input. Ogni valore 1 contenuto all'interno della matrice è visualizzato a video come un asterisco, mentre ogni 0 è sostituito da uno spazio bianco. Oltre a visualizzare la matrice, questa funzione fornisce all'utente alcune informazioni aggiuntive sullo stato del sistema, quali: il tipo di matrice che si sta visualizzando (se A o B), lo step evolutivo corrente, quante celle sono presenti nel sistema e come si suddividono tra 0 ed 1. La funzione "visualizza_matrice" richiama infine la funzione "attendi", in maniera tale da fissare sullo schermo per alcuni secondi il proprio output.

La funzione restituisce in output una variabile intera, che corrisponde al numero di celle con valore 1 presenti nel sistema in quel determinato step.

- `void preparazione_matrice(int nr, int nc, int A[][nc], char tipo, long attesa)`

La funzione "preparazione_matrice" in funzione del valore della variabile "tipo" che riceve in input, va a costruire la matrice A di partenza. Nel caso in cui il valore di "tipo" sia uguale ad 'm', la creazione è da intendersi "manuale": verrà quindi chiesto all'utente di inserire le dimensioni volute della matrice ed essa sarà riempita in maniera casuale. Al contrario, se il valore di tipo è "f", come matrice di input verrà presa quella contenuta all'interno del file "input.dat".

- `int controlla_intorno(int n, int m, int A[][m], int i, int j)`

La funzione "controlla_intorno", ricevuta in input una matrice di ordine $n \times m$, analizza l'intorno della cella di coordinate (i,j), restituendo in output un valore intero, corrispondente al numero di celle con valore 1 presenti in questo intorno.

- `void avanza_matrice(int nr, int nc, int A[][nc], int B[][nc], int nome, int soglia, int cond_ug)`

Questa funzione, ricevute in input le matrici A e B, entrambe di ordine $nr \times nc$, si occupa di effettuare la transizione dall'una all'altra. Essendo ogni step caratterizzato da due aggiornamenti, la funzione "avanza_matrice" si occupa prima di tutto di aggiornare la matrice A, salvando il nuovo stato nella matrice B. Allo stesso modo, la matrice B verrà successivamente aggiornata dalla funzione ed i risultati saranno memorizzati all'interno della matrice A.

- `void stampa_matrice(int nr, int nc, int A[][nc])`

La funzione "stampa_matrice" richiamata al termine dell'evoluzione del sistema, si occupa di "stampare" all'interno del file "output.dat" la matrice finale.

Informazioni per l'utente finale

Il programma è stato "parametrizzato" ovunque ciò possibile. Pertanto, l'utente non ha la necessità di conoscere le strutture dati utilizzate per poter utilizzare questo software. In ogni caso, per un utilizzo più "avanzato" del programma, l'utente può modificare a suo piacimento il file "parametri.dat" (la sua struttura è descritta nel paragrafo "Le strutture dati utilizzate", così come può modificare la matrice di input contenuta all'interno del file "input.dat". In questa seconda

ipotesi, si tenga semplicemente in considerazione il fatto che ogni riga della matrice è considerata “terminata” non appena è presente il carattere di nuova riga (“\n”).

Gli esperimenti effettuati

Come detto in apertura di relazione, un modello pur strutturalmente semplice quale è il “Latticeworld”, può comunque essere un interessante oggetto di studio. Una volta creato e testato ripetutamente il codice che è stato scritto, si è dunque passati ad effettuare attraverso di esso alcuni esperimenti.

Esperimento 1

Nel primo esperimento è stata utilizzata una matrice di ordine 20 x 20, costruita per l’occasione all’interno del file “input.dat”, contenente inizialmente lo stesso numero di celle con valore 1 e di celle con valore 0. La condizione di aggiornamento è stata impostata per mantenere lo stato corrente della cella, nel caso in cui numero di celle con valore 1 presenti nell’intorno fosse pari al valore di soglia (in questo esperimento, pari a 4).

Osservando il file “storia.dat” risultante dalla prima esecuzione, notiamo che ha avuto luogo un’iniziale diminuzione delle celle con valore 1 a favore di quelle con valore 0. La situazione si è poi ristabilizzata negli steps seguenti, finendo anche per ribaltarsi leggermente a favore degli 1. All’ultimo passo abbiamo infatti riscontrato 205 uni (e di conseguenza 195 zeri, visto che la nostra matrice di 20 righe e 20 colonne dà origine a 400 valori). Sono stati effettuati altri due lanci per questo stesso esperimento ed il comportamento del sistema è stato leggermente diverso nella forma, ma non nella sostanza. Questi test hanno mostrato fin da subito una crescita del numero di celle con valore 1 (al contrario di quanto accaduto nel primo esperimento, quando esse erano inizialmente diminuite), per poi stabilizzarsi alla fine dell’evoluzione con una maggioranza di 220 a 180.

Caratteristica comune ai tre “lanci” effettuati è la conformazione della matrice finale. In tutti questi casi, infatti, essa presenta una concentrazione di celle con valore 0 nella zona centrale, mentre l’agglomerato di celle con valore 1 va ad avere il proprio “centro” nella zona di confine tra gli angoli nord-ovest e sud-est della matrice. La conformazione di cui stiamo parlando è visibile nella figura qui sotto, dove è riportata una delle tre matrici di output ottenute.

```
11111111111111111111
11111111111111111111
11111111111111111111
11111111111111111111
11111111111111110110
11111111111101100000
11111110110000000000
11110110000000000000
01100000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000001001
00000000000010011111
00000001001111111111
00001001111111111111
10011111111111111111
11111111111111111111
11111111111111111111
```

Figura 1 – La conformazione tipica della matrice di output risultante dai tre lanci del primo esperimento

Non ci si dimentichi, se il risultato può apparire a prima vista inusuale, che il mondo considerato è toroidale e che quindi le celle con valore 1 appartengono ad un unico cluster e non a due separati.

Esperimento 2

Nel secondo esperimento, le condizioni di evoluzione del sistema sono le medesime utilizzate per l'esperimento precedente. L'unica differenza consiste nella matrice iniziale: in questo caso, infatti, si è voluto osservare il comportamento del sistema di fronte a matrici di partenza contenenti diverse proporzioni iniziali di celle con valore 0 o 1. Sono stati dunque effettuati due lanci con altrettante matrici di input diverse, una con maggioranza di 1 e l'altra con predominanza di 0.

Quello che è emerso in maniera chiara è che, dopo ogni lancio, l'elemento "dominante" all'inizio dell'evoluzione si è imposto su tutti i 400 valori finali della matrice, colonizzando in pochissimi step la "popolazione" degli individui. Con la matrice iniziale composta per la maggior parte di 1 abbiamo dunque ottenuto una matrice composta esclusivamente da 1; con maggioranza di 0, una matrice composta solo da 0.

Esperimento 3

Nel terzo esperimento si è osservata l'evoluzione del sistema di fronte a diversi valori di soglia. La matrice di input è quella già utilizzata per il primo esperimento, contenente un'identica proporzione di celle con valore 0 ed 1.

In primis abbiamo utilizzato una soglia pari a 3, dalla quale è rapidamente emersa come output evolutivo una matrice composta interamente da 1. Quando abbiamo invece utilizzato una soglia pari a 5, la situazione si è completamente invertita: le celle con valore 0 hanno velocemente colonizzato l'intera popolazione. Questo comportamento, d'altronde, era facilmente pronosticabile: con una soglia "bassa" (inferiore a 4, che nel primo esperimento ha dimostrato di essere un buon valore di equilibrio) le celle con valore 1 hanno più facilità di "riprodursi" e colonizzare il sistema. Al contrario, con un valore di soglia "alto" (superiore a 4), le celle con valore 1 possono conservarsi soltanto se "annidate" all'interno di isole di celle simili. Con soglia 5 o superiore, tuttavia, un agglomerato di 1 potrà "resistere" soltanto in casi estremamente rari e, quindi, con altissima probabilità il sistema tenderà comunque ad una colonizzazione da parte degli 0.

Esperimento 4

Il quarto ed ultimo esperimento è nuovamente basato sulla matrice iniziale contenente uguale proporzione di 0 e di 1. Il valore di soglia è quello "di equilibrio", ossia 4, ma varia in questo caso la condizione di aggiornamento della singola cella in caso di uguaglianza del numero di 1 vicini rispetto al valore di soglia. In una situazione del genere, dunque, la cella non rimane sulle proprie posizioni, ma sceglie in maniera casuale (con probabilità 0.5 per entrambi i contendenti) un nuovo valore.

I sei lanci effettuati hanno portato a risultati diversi: in due casi, la matrice ha mostrato una convergenza, seppur con tempistiche differenti, verso un ambiente esclusivamente caratterizzato da 0. In altri due casi, l'andamento è stato opposto e sono state le celle con valore 1 a colonizzare l'intero sistema. Infine, gli ultimi due lanci hanno evidenziato un comportamento meno "violento", ma che ha comunque portato alla supremazia piuttosto netta di uno dei due elementi.

Appendice – il workflow del programma

