Progetto *CellAuto*

Progetto per il corso di Programmazione 2005/06

1 Descrizione

Il progetto consiste nell'implementare in Java le classi necessarie a simulare il funzionamento di un particolare tipo di automi cellulari. Gli automi cellulari che considereremo qui sono macchine astratte che funzionano su una griglia, con r righe e c colonne; ogni cella della griglia è individuata da una coppia di coordinate (i, j) dove $i = 0, \ldots, r-1$ indica la riga e $j = 0, \ldots, c-1$ indica la colonna. La cella $(0, 0)$ è quella in alto a sinistra, la cella $(r-1, c-1)$ è quella in basso a destra.

Ogni cella della griglia può essere, in ciascun istante, *occupata* oppure *libera*; indicheremo in seguito con 1 lo stato occupato, e con 0 lo stato libero.

Un automa cellulare è descritto da un insieme di regole di evoluzione, che chiameremo anche *programma*, che prescrivono come evolve il sistema. Ogni programma, applicato a una certa griglia di partenza G^0, evolve per un certo numero di passi (eventualmente infinito) e, ad ogni passo, determina una nuova griglia G^{t+1} a partire dalla griglia precedente G^t.

Nel seguito descriviamo i vari tipi di programmi. Per ogni tipo di programma, indicheremo in che modo fa evolvere la griglia e per quanti passi prosegue prima di terminare.

1.1 Programma semplice

Un programma semplice viene eseguito in un unico passo. In quest'unico passo di esecuzione, diciamo dalla griglia G^0 alla griglia G^1, lo stato (occupato o libero) di ciascuna cella di G^1 è determinato sulla base dello stato che aveva in G^0 e dallo stato che in G^0 avevano le celle adiacenti. Ogni cella (i, j) ha esattamente 8 celle adiacenti: 3 sopra, 3 sotto, e 2 ai lati; la griglia va pensata come un toro, quindi le celle che si trovano nella prima colonna hanno, alla loro sinistra, le celle che si trovano nell'ultima colonna, e le celle della prima riga hanno sopra le celle dell'ultima riga.

Lo stato complessivo di una cella e delle 8 celle adiacenti è chiamato *configurazione*; un esempio di configurazione è visibile in Fig. 1.

Poiché ognuna delle 9 celle di una configurazione può assumere solo 2 valori, l'insieme C delle configurazioni possibili ha $2^9 = 512$ elementi. In Fig. 2 mostriamo come sia possibile attribuire a ciascuna delle 9 celle di una configurazione una potenza di 2: in questo modo si determina una esplicita biiezione fra l'insieme C e l'insieme degli interi da 0 a 511. In pratica, per stabilire il valore di una specifica configurazione basta sommare le potenze di 2 nelle posizioni occupate.

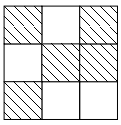

Figura 1: Una delle 512 possibili configurazioni.

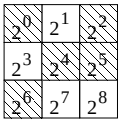

Figura 2: Come assegnare una codifica intera a ciascuna configurazione.

Nell'esempio di Fig. 2 la configurazione è la numero $2^0 + 2^2 + 2^4 + 2^5 + 2^6 = 1 + 4 + 16 + 32 + 64 = 117$. Analogamente, ad esempio, la configurazione 0 è quella tutta vuota e la configurazione 511 è quella con tutte le celle occupate.

D'ora in avanti, identificheremo l'insieme C delle configurazioni con l'insieme degli interi, secondo la codifica sopra indicata.

Ora, un programma semplice è una funzione $f : C \to \{0, 1\}$ che decide se una cella deve essere libera o occupata sapendo quale era la configurazione del suo intorno all'istante precedente. In pratica, l'esecuzione del programma sulla griglia G^0 avverrà come segue: per ogni (i, j), si determina la configurazione intorno alle cella (i, j) nella griglia G^0, si applica f alla configurazione e si assegna il valore (0 o 1) così ottenuto alla cella (i, j) di G^1.

Anziché rappresentare un programma semplice mediante una funzione f, lo rappresentiamo mediante un sottoinsieme F di C, definito da $f^{-1}(1)$. In altre parole, un programma semplice è univocamente identificato dall'insieme delle configurazioni a fronte delle quali una cella viene occupata.

1.2 Programmi composti

Un programma composto è ottenuto a partire da altri programmi, semplici o composti, nei seguenti modi:

- *sequenza*: una sequenza è una successione finita di programmi; la sua esecuzione comporta l'esecuzione dei programmi uno dopo l'altro;
- *for*: un for è definito da un programma e da un intero; la sua esecuzione comporta l'esecuzione del programma per il numero di volte specificato;
- *forever*: un forever è definito da un programma; la sua esecuzione comporta l'esecuzione del programma un numero infinito di volte;
- *while*: un while è definito da un programma e da una condizione; la sua esecuzione comporta l'esecuzione del programma finché la condizione indicata non diventa falsa.

2 La gerarchia `Set`

Per la realizzazione del progetto, sarà conveniente l'uso della gerarchia `Set` (nel pacchetto `java.util`). Se ne fornisce qui una breve descrizione (relativa alla versione Java 1.5) per rendere comprensibile il testo del progetto. `Set` è un'interfaccia generica le cui istanze rappresentano degli insiemi; più precisamente, `Set<String>` rappresenta un insieme di stringhe, `Set<Integer>` un insieme di interi ecc. Per gli scopi di questo progetto converrà usare dei `Set<Integer>`.

`Set` ha un metodo `add`, che consente di aggiungere un elemento all'insieme; questo metodo non ha alcun effetto se l'insieme contiene già l'elemento indicato. Inoltre, ha un metodo `contains` che restituisce true o false a seconda che l'elemento indicato stia nell'insieme oppure no.

`HashSet` è un'implementazione concreta (che si trova sempre in `java.util`) di `Set`.

Il seguente frammento di codice dichiara e crea un insieme di interi, e lo riempie con i primi 10 numeri pari:

```
Set<Integer> x = new HashSet<Integer>();
for ( int i = 0; i < 10; i++ )
 x.add( 2 * i );
for ( int i = 0; i < 20; i++ )
 if ( x.contains( i ) )
  System.out.println( "L'insieme␣contiene␣" + i );
```

3 Le classi da realizzare

È richiesto di risolvere il progetto descritto nella sezione precedente realizzando in Java le seguenti classi e interfacce:

- `Griglia`, che descrive una griglia; ha i seguenti metodi e costruttori:
  - `Griglia(int r,int c)`: crea una griglia con r righe e c colonne, vuota;
  - `Griglia(Griglia g)`: crea una griglia identica a g (stesso numero di righe, di colonne e stesso contenuto); notate che le due griglie dovranno essere indipendenti (cioè, cambiando lo stato dell'una non dovrà essere mutato lo stato dell'altra);
  - `int getR()`: restituisce il numero di righe;
  - `int getC()`: restituisce il numero di colonne;
  - `int conf(int i,int j)`: restituisce il codice della configurazione intorno alla cella (i, j), come descritto in precedenza (un numero fra 0 e 511);
  - `boolean occupato(int i,int j)`: restituisce true se e solo se la cella (i, j) è occupata;
  - `void set(int i,int j,boolean v)`: rende la cella (i, j) occupata o libera, a seconda che v sia true o false;
  - `void flip(int i,int j)`: cambia lo stato della cella (i, j) (da libera a occupata o viceversa);
  - `int occupati()`: restituisce il numero di celle occupate.

- `Programma`, che descrive un generico programma; è un'interfaccia con il solo metodo seguente:
  - `void esegui(Griglia g) throws InterruptedException`: esegue il programma sulla griglia g, modificandone il contenuto; alla fine dell'esecuzione g conterrà la griglia finale; se il programma è infinito, l'invocazione di questo metodo non ha termine. Si noti che il metodo deve poter sollevare una `InterruptedException`, come verrà spiegato in seguito[1].
- `ProgrammaSemplice`, un'implementazione di programma che rappresenta un programma semplice; deve avere i seguenti metodi e costruttori:
  - `ProgrammaSemplice()`: costruisce il programma $\emptyset$;
  - `boolean add(int x)`: aggiunge x al programma ammesso che non fosse già presente, e restituisce true se x non era presente, false altrimenti;
  - `void esegui(Griglia g) throws InterruptedException`: come richiesto dall'interfaccia (non solleva mai l'eccezione);
  - `static int densita(int conf)`: restituisce il numero di celle occupate nella configurazione data come argomento;
  - `static String convert(int conf)`: converte la configurazione data come argomento in una stringa di esattamente 9 caratteri, in cui l'i-esimo carattere è uno spazio o un asterisco a seconda che la posizione corrispondente alla potenza 2^i sia libera o occupata;
  - `static int convert(String conf)`: effettua la conversione inversa a quella appena descritta.

  Si noti che è possibile (e consigliabile) scrivere la classe in modo che estenda `Set<Integer>`.
- `Sequenza`, un'implementazione di programma che rappresenta una sequenza; deve avere i seguenti metodi e costruttori:
  - `Sequenza()`: costruisce la sequenza vuota;
  - `void add(Programma p)`: aggiunge p in fondo alla sequenza;
  - `void esegui(Griglia g) throws InterruptedException`: come richiesto dall'interfaccia; i programmi della sequenza vengono eseguiti uno dopo l'altro, nell'ordine, e fra l'esecuzione di ciascuno e del successivo viene invocato il metodo `Thread.sleep(500)` che fa una pausa di mezzo secondo; questo metodo può sollevare una `InterruptedException`.
- `For`, un'implementazione di programma che rappresenta un for; deve avere i seguenti metodi e costruttori:
  - `For(Programma p,int v)`: costruisce un for per eseguire v volte il programma p;

[1] Nota bene: per la realizzazione del progetto non è necessario che sappiate cos'è una `InterruptedException`; limitatevi a tal proposito a seguire le istruzioni indicate nel seguito.

  - `void esegui(Griglia g) throws InterruptedException`: come richiesto dall'interfaccia; fra un'esecuzione e la successiva viene invocato il metodo `Thread.sleep(500)` che fa una pausa di mezzo secondo; questo metodo può sollevare una `InterruptedException`.

- `Forever`, un'implementazione di programma che rappresenta un forever; deve avere i seguenti metodi e costruttori:
  - `Forever(Programma p)`: costruisce un ciclo per eseguire infinite volte il programma p;
  - `void esegui(Griglia g)`: come richiesto dall'interfaccia; fra un'esecuzione e la successiva viene invocato il metodo `Thread.sleep(500)` che fa una pausa di mezzo secondo; questo metodo può sollevare una `InterruptedException`.

- `Condizione`, un'interfaccia con il seguente solo metodo:
  - `boolean vale(Griglia g)`: restituisce true se e solo se la condizione vale sulla griglia g.

- `While`, un'implementazione di programma che rappresenta un while; deve avere i seguenti metodi e costruttori:
  - `While(Programma p,Condizione c)`: costruisce un ciclo while che esegue il programma p finché la condizione c diventa falsa (come in un normale ciclo while in Java o C);
  - `void esegui(Griglia g)`: come richiesto dall'interfaccia; fra un'esecuzione e la successiva viene invocato il metodo `Thread.sleep(500)` che fa una pausa di mezzo secondo; questo metodo può sollevare una `InterruptedException`.

- `CondizioneOccupatiGeq`, un'implementazione dell'interfaccia `Condizione` che è vera se e solo se il numero di celle occupate è $\geq$ (Greater or EQual) a un numero specificato m; deve avere i seguenti metodi e costruttori:
  - `CondizioneOccupatiGeq(int m)`: crea la condizione;
  - `boolean vale(Griglia g)`: come richiesto dall'interfaccia.

- `CondizioneCella`, un'implementazione dell'interfaccia `Condizione` che è vera se e solo se una specifica cella (i, j) diventa libera o occupata; deve avere i seguenti metodi e costruttori:
  - `CondizioneCella(int i,int j,boolean v)`: crea la condizione che diventa vera non appena la cella (i, j) diventa occupata (se v è true) o libera (se v è false);
  - `boolean vale(Griglia g)`: come richiesto dall'interfaccia.

A parte quanto espressamente richiesto, è lasciata piena libertà sull'implementazione delle singole classi e sull'eventuale introduzione di classi aggiuntive, a patto di seguire le regole del paradigma ad oggetti ed i principi di buona programmazione. Si suggerisce di porre particolare attenzione alla scelta dei modificatori relativi a variabili d'istanza e metodi, nonché alla dichiarazione e

alla gestione delle eccezioni che possono venire lanciate dai vari metodi e alle relazioni di ereditarietà tra le varie classi. Per la realizzazione di `Sequenza` si consiglia di usare l'interfaccia `List<Programma>` (e la sua implementazione `ArrayList<Programma>`) di cui si consiglia di leggere la documentazione nelle API standard, pacchetto `java.util`; in alternativa, potete comunque usare un array opportunamente sovradimensionato.

4 Testing

Per facilitare il testing delle proprie classi si forniscono, sul sito web, tre sorgenti Java; per usarli, operate come segue:

- scaricate i tre sorgenti (`Driver.java`, `Griglie.java`, `Programmi.java`);
- modificate la prima riga dei sorgenti contenente la direttiva `package` indicando il nome del vostro pacchetto, oppure togliete la direttiva se le vostre classi si trovano nel pacchetto senza nome;
- compilate insieme le vostre classi e le tre classi fornite: notate che per farlo dovrete usare un compilatore Java 1.5; nel caso che stiate usando Eclipse, assicuratevi che fra le proprietà del progetto sotto *Java Compiler* sia indicato *Compiler compliance level: 5.0*, e che stiate usando la JRE 1.5 (*proprietà/Java Build Path/Libraries*).

Se avrete realizzato tutti i metodi sopra elencati dovreste riuscire a compilare. A questo punto eseguite il driver, con il comando:

```
java <nome pacchetto>.Driver
```

Nel menù *Griglie* potete scegliere fra vari modi di riempire la griglia, mentre nel menù *Programmi* potete mandare in esecuzione uno di vari programmi possibili. I modi di riempire la griglia e i programmi sono determinati nelle due classi *Griglie.java* e *Programmi.java*. Se volete, potete aggiungere vostri metodi a queste due classi *prestando attenzione che presentino la stessa segnatura esatta di quelli già presenti*: ricompilando, vedrete comparire nuove voci nei menù del Driver.

Non verranno presi in considerazione progetti le cui classi non permetteranno di compilare le classi presenti sul sito.

5 Demo

A titolo di esempio, forniamo un demo (ovviamente, solo in versione compilata). Per provarlo scaricate `demo.jar` dal sito, ed eseguitelo con il comando

```
java -cp demo.jar it.unimi.mat.esamegiu06.Driver
```

6 Valutazione

La valutazione del progetto sarà fatta in base alla

- presenza di funzionalità aggiuntive;

- conformità dell'implementazione scelta per risolvere il problema con il paradigma di programmazione a oggetti;
- conformità del codice presentato alle regole di buona programmazione;
- adeguatezza del manuale utente presentato a descrivere il modo in cui un utente può utilizzare il programma;
- adeguatezza della documentazione e dei commenti;
- assenza di errori nel programma.