

EDITOR PER AUTOMI CELLULARI

DOMENICO BRUZZESE
FABIO CAPIRCHIO
GABRIELE FRARACCI
LEONARDO EMILI

1. INTRODUZIONE

Il progetto consiste nella creazione di un editor per automi cellulari e di una GUI. L'intero progetto é fortemente *user-oriented*, l'editor lascia libero l'utente di personalizzarne ogni aspetto: dalla personalizzazione grafica fino a quella strutturale che gestisce il flusso della simulazione.

L'idea é quella di eseguire una simulazione che utilizzi i concetti di multithreading, concorrenza nonché i pilastri della *OOP*.

2. TECNOLOGIE UTILIZZATE

L'intero progetto é stato scritto in *Java 9*, assieme a: *JavaFx* utilizzato per creare la GUI, *FXML* che ha contribuito a formare la struttura del programma, *CSS* per le personalizzazioni grafiche, *LaTeX* per la stesura di questa relazione.

3. GUIDA ALL'USO

Il programma ha un'interfaccia semplice e intuitiva ed é organizzato in 3 stadi logici: un primo stadio dove l'utente sceglie in che modalit  eseguire il programma, un secondo stadio dove vengono recuperate le informazioni necessarie affinché la simulazione possa iniziare, ed infine un ultimo stadio di interazione con il simulatore.

3.1. Schermata di benvenuto.

3.1.1. *Crea nuovo profilo: l'utente viene guidato in una procedura che gli permette di personalizzare i colori associati agli stati e le direttive che regolano lo sviluppo dell'automa. Al termine viene chiesto all'utente se si desidera preservare queste configurazioni per poterle riutilizzare in futuro, in caso contrario verr  creata una "shallow copy" delle configurazioni che verr  poi distrutta al termine dell'esecuzione del programma.*

3.1.2. *Carica profilo: il programma carica le configurazioni utente gi  esistenti per essere poi fornite al simulatore.*

3.1.3. *Scegli automa: viene visualizzata una raccolta di automi cellulari con configurazioni e colori di default.*

3.2. **Schermata degli input.** In questa sezione l'utente é libero di scegliere gli stati delle cellule attraverso un'idea pi  intuitiva: la scelta dei colori che le cellule dovranno avere, in questo modo si interagisce direttamente con il loro ciclo di vita. É inoltre possibile personalizzare i principi che regolano le transizioni di stato. Queste idee verranno trattate in maniera pi  dettagliata nei paragrafi successivi.

3.3. **Schermata del simulatore.** Il protagonista, la tela del quadro é dove avviene la scena e dove é possibile assistere alla simulazione. Questa é circondata ai lati da pulsanti che regolano le configurazioni iniziali, mentre all'estremo inferiore sono localizzati i pulsanti di controllo con i quali é possibile interagire direttamente con la simulazione.

4. AUTOMI CELLULARI

Nel seguito assumiamo che ciascuna cellula appartenga ad un unico stato, i motivi legati a questo dettaglio sono molteplici e la ragione apparirà più chiara nel *paragrafo 6* dove la questione viene trattata più nel dettaglio.

Nel testo viene inoltre utilizzata frequentemente la nozione di *vicino* fornita da *Edward Forrest Moore*, pioniere della *teoria degli automi cellulari*: il vicinato di una cella C è composto da 9 celle, la cella centrale C e dalle 8 celle che lo circondano.

4.1. Game of Life. L'automa sviluppato dal matematico *John Conway* è uno dei pilastri della collezione, l'apparente semplicità con cui regole e stati vengono definiti porta all'occhio non pochi risvolti teorici. La scena si svolge su una griglia bidimensionale formata da cellule che possono assumere soltanto due stati: vivo o morto. Le regole di transizione adottate sono le stesse definite in origine dal matematico britannico:

- La cellula che si trova nello stato morto e conta esattamente 3 cellule vive nel suo vicinato transisce nello stato vivo.
- La cellula che si trova nello stato vivo e che ha o un numero di vicini maggiore strettamente di 3 oppure numero di vicini minore strettamente di 2 transisce nello stato morto.
- La cellula che si trova nello stato vivo e conta un numero di vicini pari a 2 o 3 sopravvive alla prossima generazione rimanendo nello stesso stato.

4.2. Circular Redistribution. Si presenta in aggiunta un nuovo automa a cui è stato dato il nome *Circular Redistribution*, essendo nato dall'unione di regole per l'eccitamento di una cella con un *blur* circolare basato su una semplificazione delle funzioni trigonometriche per selezionare uno dei vicini della cella sulla quale si eseguono le varie computazioni.

Lo stato iniziale assegnato ad una cella è un valore preso nell'intervallo $[0, \dots, 1]$ tramite il quale viene assegnato un colore in scala di grigi che riflette lo stato scelto. L'algoritmo applicato dall'automa è il seguente:

- (1) Per ogni cella, si calcola la somma S dello stato assegnato ad ognuno dei vicini di *rango 1* della cella corrente.
- (2) Due valori, *MinThreshold* e *MaxThreshold* vengono utilizzati per determinare se la cella è in stato di *eccitazione* o meno. Si ottiene una cella in stato di eccitazione se $\text{MinThreshold} < S < \text{MaxThreshold}$.
- (3) La somma S dei valori di una cella eccitata viene trasformata in un *angolo*, utilizzato per selezionare uno degli 8 vicini di *Moore* della cella eccitata. Una volta selezionato il vicino, si interpolerà il valore dello stato della cella eccitata con il valore dello stato della cella selezionata, tramite l'utilizzo di una variabile definita *interpolation*.
- (4) L'angolo utilizzato viene salvato all'interno della cella *eccitata*, di modo che nell'iterazione successiva l'angolo calcolato tramite la nuova somma dei valori verrà sommato all'angolo precedentemente trovato, creando appunto una *rotazione* delle celle scelte per il *blur* che ha dato il nome all'automa presentato.

Utilizzando una configurazione iniziale nella quale la griglia è composta di colori casuali esclusivamente neri o bianchi con una distribuzione di almeno il 50% di elementi bianchi si ottiene una struttura che a seguito delle prime 500 generazioni

tende a creare cluster simili alle simulazioni ottenute con gli algoritmi di *marching squares*.

Non è ancora chiaro se la struttura cellulare che si genera con questo tipo di configurazioni è destinata o meno a stabilizzarsi nel corso di un numero finito di generazioni, tramite sperimentazioni su un lasso di tempo prolungato le dimensioni dei cluster sembrano crescere fino a stabilizzarsi, ma l'evoluzione del loro movimento lungo la griglia è continuo.

4.3. Cyclic CA. Inizialmente sviluppati da *David Griffeath* all'università del Wisconsin, gli automi cellulari ciclici esibiscono un complesso sistema di auto - organizzazione iterando una regola estremamente semplice su di una matrice di stati. Un numero n di possibili stati viene scelto a priori, che verranno rappresentati tramite una scala cromatica di colori, uno per ogni stato. Ad ogni iterazione ciascuno *stato/colore* può avanzare esclusivamente al suo **stato successivo**, l'ultimo dei quali ritorna al primo, in modo ciclico.

Le regole di sviluppo dell'automa seguono la notazione **$R/T/C/N$** :

- **R** specifica il rango della ricerca dei vicini all'interno della griglia sul quale si simula l'automa, per $R=1$ saranno dunque visitate tutte le celle a distanza 1 dalla cella sulla quale si stanno applicando le regole.
- **T** specifica una soglia (*threshold*) che indica il minimo numero di celle adiacenti che hanno il *colore/stato* successivo necessario alla cella corrente per avanzare allo stato successivo.
- **C** specifica il numero di possibili stati n presenti nella griglia. Ad ognuno di questi sarà automaticamente assegnato un colore specifico della scala cromatica.
- **N** discrimina la regola utilizzata per selezionare i vicini, che può essere *extended Moore* oppure *von Neumann*. In questa si è deciso di utilizzare esclusivamente l'*Extended Moore* essendo questo quello più affine alla creazione di pattern a seguito del cambio delle tre regole precedenti.

La condizione iniziale da preferirsi per godere al meglio di questa simulazione è quella in cui viene assegnato un colore casuale ad ogni cella della griglia, sia questo una *tupla* (r,g,b) oppure un colore in scala grigi.

4.4. Diffusion Aggregation. Diffusion Aggregation è una simulazione semplificata di un processo conosciuto come *Diffusion-Limited Aggregation*¹. Processo tramite il quale particelle che percorrono un random walk secondo un movimento Browniano formano cluster aggregati di complesse ed ordinate strutture ad albero. Inizialmente una griglia vuota viene occupata da un numero casuale di celle il cui stato iniziale è *in movimento*, si assegna inoltre alla cella al centro della griglia lo stato *aggregata*.

Ad ogni cella *in movimento* viene assegnato un colore che descrive una delle 8 possibili direzioni (sempre secondo il *Moore neighborhood*) in cui la cella sta viaggiando. Nel caso in cui due celle *in movimento* dovessero scontrarsi, viene assegnata una direzione causale differente ad entrambe per permettergli di continuare il loro viaggio. Se invece una cella entra in contatto con altra il cui stato è *aggregata*, anche lei muterà in *aggregata*. Si stabilisce un contatto quando una cella in movimento

¹La teoria fu proposta inizialmente da T.A. Witten Jr. nel 1981 (Witten, T. A.; Sander, L. M. (1981). "Diffusion-Limited Aggregation, a Kinetic Critical Phenomenon". Physical Review Letters. 47 (19): 14001403. Bibcode:1981PhRvL..47.1400W. doi:10.1103/PhysRevLett.47.1400).

diventa una vicina di una cella aggregata. All'aumentare delle celle *aggregate* segue il diminuire delle celle in movimento rimaste, la simulazione terminerà con l'assegnazione dello stato *aggregata* al totale delle celle iniziali presenti all'interno della griglia.

4.5. H3 Rule². È la classe di automi più semplice nella quale ogni cella possiede uno dei due possibili valori, 0 o 1. L'evoluzione di una cella dell'automata alla generazione successiva è interamente specificata dai valori delle due celle adiacenti e dallo stato attuale della cella, ottenendo dunque un massimo di 256 possibili combinazioni. Si parte da una prima *riga* nella superiore della griglia in cui viene mostrata la prima generazione dell'automata, ad ogni iterazione la riga successiva calcolerà gli stati della nuova generazione, e così via.

La regola utilizzata è la n° 182, che presenta un'evoluzione che ricalca il famoso ***triangolo di Sierpinski***³. Per visualizzare questo risultato è necessario impostare la condizione iniziale della griglia che assegna ad una sola cella lo stato 1, che sarà la cella centrale della prima riga in alto della griglia.

²H3-Rule è una simulazione dell'automata cellulare elementare 1-Dimensionale riferito tramite il suo Wolfram code, una convenzione inventata da Wolfram tramite la quale sono presenti fino a 256 possibili regole (<http://mathworld.wolfram.com/ElementaryCellularAutomaton.html>)

³Wacław Franciszek Sierpiński (Varsavia, 14 marzo 1882 - Varsavia, 21 ottobre 1969) è stato un matematico polacco. Egli è ampiamente conosciuto per i suoi contributi nella teoria degli insiemi ed per le sue scoperte, tra cui: il numero e il triangolo di Sierpiński.

5. CODE SNIPPETS

Esempio 1. Il seguente metodo mostra un possibile uso della Reflection.

```
private void comeToMeCA(String className) {
    try {
        Class<?> C = CellularAutomataProgram.class.forName("simulatorWindow.programs."
            + className);
        CellularAutomataProgram currentProgram = CellularAutomataProgram.class.cast(C.
            getConstructor().newInstance());
        simulator.setProgram(currentProgram);
    } catch (ClassNotFoundException e) {
        System.out.println("Class " + className + " has not been defined.");
    } catch (NoSuchMethodException e) {
        System.out.println("Constructor not found");
    } catch (IllegalAccessException e) {
        System.out.println(e.getMessage());
    } catch (InstantiationException e) {
        System.out.println(e.getMessage());
    } catch (InvocationTargetException e) {
        System.out.println(e.getMessage());
    }
}
```

Esempio 2. Il seguente metodo mostra l'inputto della logica booleana nel progetto.

```
private void comeToMeCA(String className) {
    try {
        Class<?> C = CellularAutomataProgram.class.forName("simulatorWindow.programs."
            + className);
        CellularAutomataProgram currentProgram = CellularAutomataProgram.class.cast(C.
            getConstructor().newInstance());
        simulator.setProgram(currentProgram);
    } catch (ClassNotFoundException e) {
        System.out.println("Class " + className + " has not been defined.");
    } catch (NoSuchMethodException e) {
        System.out.println("Constructor not found");
    } catch (IllegalAccessException e) {
        System.out.println(e.getMessage());
    } catch (InstantiationException e) {
        System.out.println(e.getMessage());
    } catch (InvocationTargetException e) {
        System.out.println(e.getMessage());
    }
}
```

Esempio 3. Altro metodo figo che abbiamo fatto ? comeToMeCA serve solo per riempire lo spazio, va rimpiazzato col nuovo codice.

```
private void comeToMeCA(String className) {
    try {
        Class<?> C = CellularAutomataProgram.class.forName("simulatorWindow.programs."
            + className);
        CellularAutomataProgram currentProgram = CellularAutomataProgram.class.cast(C.
            getConstructor().newInstance());
        simulator.setProgram(currentProgram);
    } catch (ClassNotFoundException e) {
        System.out.println("Class " + className + " has not been defined.");
    } catch (NoSuchMethodException e) {
        System.out.println("Constructor not found");
    }
}
```

6. TEORIA DEGLI STATI E DELLE TRANSIZIONI

Lo scenario precedentemente descritto si basa su due concetti principali: stati e regole di transizione.

6.1. Gli stati. L'idea di base é che ciascuna cellula al momento della creazione viene associata uno stato che ne regola lo sviluppo. Si noti come in questo modello gli stati non sono solo semplici propriet  di cui ogni cellula dispone, bens  essi sono parte attiva dell'evoluzione dell'automa. Questo implica che in un dato momento τ uno stato S pu  avere nel proprio dominio D un numero indeterminato di cellule che oscilla tra lo 0 e l'intera popolazione di cellule. Nello scenario descritto vengono descritti i casi pi  estremi, e logico dunque assumere che in un momento τ lo stato S conterr  un numero medio di cellule che chiameremo m . In ogni istante ciascuno stato S interroga le proprie cellule e, in base alle regole di transizione definite, quest'ultime sono in grado di abbandonare S per essere integrate in un nuovo stato. Questo meccanismo viene eseguito contemporaneamente da parte di tutti gli stati che sono in grado di applicare le proprie regole senza interferire con cellule estranee al proprio dominio. Al termine della procedura tutte le possibili transizioni sono state eseguite e una volta eseguito il refresh globale della scena l'intero *algoritmo*   pronto a ripartire. Si noti come l'intero processo venga *virtualizzato* da parte degli stati: essi applicano regole su ciascuna delle proprie cellule che a loro volta possono mutare il loro stato interno, ma sar  solo al termine di ciascun colpo di clock che le cellule renderanno visibile la loro transizione agli altri stati. Questo meccanismo di mutua esclusione tra gli stati impedisce loro di interferire, rendendo cos  possibile la parallelizzazione dei processi su un'unica griglia.

6.2. Regole di transizione. Le regole di transizione regolano i processi di mutazione delle cellule in ogni momento. Esse possono essere pensate come particolari *funzioni matematiche biunivoche* definite su un dominio di stati D e su un codominio di stati C . A ciascuno stato   associato, in base regole, un proprio stato di arrivo che pu , eventualmente, coincidere con lo stato di partenza nel caso in cui le regole lo prevedano.

Le regole sono completamente personalizzabili e si basano sui pilastri della **logica booleana**. Ogni complessa equazione booleana pu  essere scomposta in proposizioni pi  semplici che prendono il nome di *proposizione atomiche*. Secondo il *principio del terzo escluso* della logica booleana ciascuna proposizione atomica ha un unico valore di verit : vero o falso. Il programma   in grado, attraverso una combinazione di molteplici proposizioni, di stabilire il valore di verit  di una generica regola o di una combinazione di regole ed di applicarle in tal senso. Alcuni dettagli implementativi a riguardo sono trattati nel *Paragrafo 5*.

7. PROBLEMI MATEMATICI COINVOLTI

8. CURIOSITÀ