Alma Mater Studiorum · Università di Bologna

facoltà di scienze matematiche, fisiche e naturali
Corso di Laurea in Scienze di Internet

Programmare un sintetizzatore con Automi Cellulari

Tesi di Laurea in Programmazione Internet

Relatore:
Chiar.mo Prof.
Antonio Messina

Presentata da:
Armando Viliam Fortunato

Sessione III
Anno Accademico 2010/2011

Indice

1. Introduzione

Lo scopo di questa tesi è descrivere gli Automi Cellulari, un sistema che combina in sequenza un certo numero di celle , strutturate in binario, in base a semplici regole che ne determinano uno sviluppo "casuale".

Dimostrare come la sua applicazione sia estendibile a diversi argomenti sia matematici che "artistici".

In particolar modo si vuol mostrare il suo impiego nell'ambito musicale, sia come generatore di melodia sia come generatore di ritmica. Verrà mostrata la realizzazione di un progetto software in Java in grado di riprodurre un elementare melodia generata mediante Cellular Automata. Ovviamente ciò costituirà soltanto da esempio.

Le applicazioni e le implicazioni dei CA sono notevoli e il campo è tutt'oggi in aperta esplorazione.

La concezione degli Automi Cellulari è molto vicina a quella di cibernetica, tuttavia quest'ultima si è focalizzata più su aspetti ingegneristici, mentre gli automi hanno messo l'accento sul digitale e sulla logica dei moderni computers. I due campi hanno permesso verso la metà del secolo scorso l'inizio dell'era digitale.

1.1 Precedenti

Si presume che già in epoche molto antiche fossero stati inventati automi meccanici in grado di fare azioni anche non del tutto scontate. Ci sono svariati esempi nella letteratura che riportano ritrovamenti di automi, dagli antichi greci (120-150 a.C.) al basso medioevo (1100-1200 d.C.), e dal rinascimento sino ai giorni d'oggi dove l'automazione ha raggiunto un grado elevato di studio. E' riportato che nel mondo ellenistico gli automi erano utilizzati per impressionare i fedeli nei riti religiosi o come giocattoli o per dimostrare teorie e principi scientifici e il libro di Erone ci mostra già nel III secolo a.C. la conoscenza di diversi principi meccanici e idraulici e come questi fossero utilizzati sia a scopo civile sia come parti nella realizzazione di automazioni più complesse. Nell'VIII secolo l'alchimista islamico Giabir ibn Hayyan scrisse un trattato nel quale inseriva ricette per costruire serpenti, scorpioni ed esseri umani artificiali che fossero soggetti al controllo del loro creatore e nel 827 il califfo Al-mam'un si diceva avesse un albero d'argento automatizzato con uccelli che cantavano, sorprendente è poi la colomba di Archita menzionata da Aulo Gellio, un automa capace di volare[**RUS01**]. Altri esempi di persone che hanno dedicato la loro vita a questi studi sono Al-Jaziri, uno scienziato, artista e inventore mediorientale (1200 d.C.), che per intrattenere gli ospiti alle serate, si dice avesse realizzato una nave sulla quale vi erano quattro automi musicisti in grado di suonare e muoversi realisticamente, o Jaques Vocanson, inventore del telaio automatico, che esplorò il mondo delle automazioni ampiamente, riuscendo a riprodurre meccanismi che potremmo definire robot veri e propri (Es. l'anatra meccanica, o il suonatore di flauto). Non mancano poi esempi di automazioni a scopo bellico, marittimo, etc. Recentemente lo sviluppo si è accelerato vorticosamente, e molti sono i centri oggi che studiano la realizzazione di veri e propri androidi in grado di riprodurre fedelmente anche le espressioni umane. Nonostante ciò nel corso della storia sicuramente abbiamo perso molte conoscenze in molti ambiti, nuove ne abbiamo acquisite ma molte sono andate perdute, uno dei campi in cui questa conoscenza è venuta a mancare è stata proprio quella sugli automi e le automazioni. Non si intende negare che la storia manchi di esempi di studio come precedentemente detto, ma sicuramente lo sviluppo dell'automazione è stata in qualche modo frenata in quanto la sostituzione dell'uomo nel lavoro, anche nei campi definiti artistici, non è mai stata ben vista e compresa nel modo adeguato. Lo studio approfondito di tutte le applicazioni degli automi trascina con sé uno studio su ciò che è il lavoro umano e su ciò che realmente fa l'uomo nel dettaglio del suo cosi detto "lavoro", facendo emergere la verità sconcertante che la maggior parte dei lavori e delle azioni dell'uomo sono di tipo meccanico e quindi riproducibili, eseguibili anche da macchine e computer

progettati per farlo. Non bisogna sorprendersi del fatto che l'uomo sia quasi nella totalità delle sue manifestazioni odierne meccanico e quindi riproducibile, sostituibile e conseguentemente soggetto ad obsolescenza. Gli automi ci mostrano questa nostra tendenza e offrono la possibilità di sgravarci dai lavori puramente meccanici per dedicarci allo sviluppo e alla crescita.

Automi Cellulari

2.1 Cosa sono?

Dal punto di vista teorico gli automi cellulari furono introdotti da John von Neumann e Stanislaw Ulam negli anni '40. Dal punto di vista "pratico" invece fu John Horton Conway a introdurli, con il suo "Game of Life", uscito negli anni '60. I Cellular Automata (CA) sono Sistemi Dinamici Discreti studiati in computazione, matematica, biologia e fisica, spesso definiti come equivalenti delle equazioni differenziali parziali; che hanno la capacità di descrivere sistemi dinamici continui. Sono definiti discreti perché spazio, tempo e proprietà dell'automa possono avere solo un numero finito e numerabile di stati. L'idea che sta alla base dei CA è quella della simulazione attraverso l'interazione delle celle seguendo semplici regole. I CA non servono per descrivere un sistema complesso con equazioni complesse ma per lasciare che la complessità emerga dall'interazione di semplici elementi che seguono semplici regole. Le principali proprietà dei CA sono:

- Un reticolo n-dimensionale dove ogni cella della matrice ha uno stato discreto.
- Un comportamento dinamico descritto dalle regole definite. Queste regole descrivono lo stato della cella per il prossimo time-step dipendente dallo stato delle celle nell'intorno della cella stessa.

Il primo sistema ampiamente esplorato per i computer è il Game of Life (gioco della vita) che venne inizialmente usato per provare alcune teorie come la possibilità di prevedere lo stato futuro degli avvenimenti in un sistema. Si è poi evoluto in una teoria scientifica di sistemi complessi sino ad arrivare ai confini della teoria del caos (:come emerge l'ordine dal caos?). Ad oggi si è esteso l'utilizzo grazie alla potenza di calcolo che è divenuta sufficiente per farlo girare anche sui normali computer di casa.

2.2 Come funzionano?

Come è stato precedentemente accennato un CA è composto da pochi semplici elementi:

- □ la cella
- □ il reticolo
- □ il vicinato
- □ le regole

La cella è l'elemento base del CA. Rappresenta l'elemento di memoria degli stati. Ogni cella può rappresentare diversi stati, nei casi più semplici come quello binario 2 (0 e 1). In sistemi molto complessi ogni cella può avere delle proprietà ed ogni proprietà dei definiti stati.

Il reticolo è l'elemento che contiene le celle. Può avere più dimensioni, i più semplici CA sono unidimensionali, chiamati collana di perle, rappresentati da una stringa di valori; i più utilizzati sono invece quelli bi-dimensionali: matrici contenenti una o più celle. Quelli unidimensionali sono definiti più semplici in merito non solo ai problemi implementativi ma anche analitici dal punto di vista del risultato; sistemi a matrice n= 2 dimensioni evidenziano dinamiche notevolmente più complesse.

In parole semplici un CA può essere definito una griglia contenente celle che si comportano e evolvono in base a regole definite in base al vicinato. Il vicinato è dunque un qualsiasi tipo di intorno delle celle che viene analizzato e trattato secondo le regole che vengono imposte al sistema.

Abbiamo vari tipi noti di "neighbourhood" (vicinato).

- □ Von Neumann Neighbourhood:

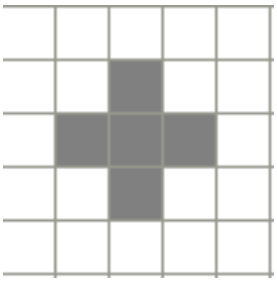


Fig.1: Vicinato di VonNeumann.

- □ Moore Neighbourhood:

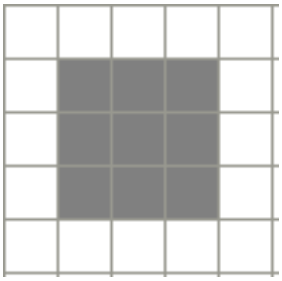


Fig.2: Vicinato di Moore.

Il vicinato di Moore può variare in base ad un raggio d'azione, ad esempio con r = 2 abbiamo non 8 celle bensì 24.

□ Margolus Neighbourhood:

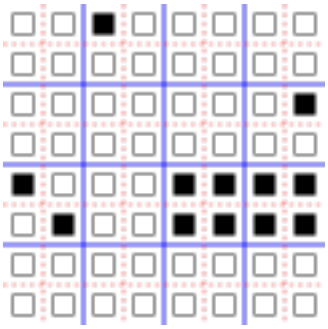


Fig.3: Vicinato di Margolus[**TOFMAR87**].

consiste nel dividere ogni spazio della griglia in gruppi di 4 celle, le regole vengono applicate internamente ad ogni porzione di griglia. La sua implementazione è più complessa.

Qualsiasi sia la griglia da costruire e il vicinato da regolare emerge sempre un problema comune a tutti i CA, cosa fare delle celle ai bordi? Le soluzioni sono due, o considerare la griglia come unita, e quindi un unica linea circolare o matrice, o considerare i bordi come specchi.

Le regole da applicare sono poche e semplici e si basano sul principio che "interazioni locali portano a dinamiche globali". Possiamo schematizzarle in tre classi:

1. Ogni gruppo di stati delle celle vicine è connesso allo stato della cella centrale. Esempio: se consideriamo un CA unidimensionale composto di valori binari ("...110...") possiamo stabile regole in base a quali valori, la cella x ad esempio, ha nelle posizioni x-1 e x+1, o qualsiasi altro "vicinato" si voglia stabilire, e dare una regola per stabilire il successivo nodo centrale.

2. Regole "totalistiche", ossia il valore della cella centrale dipende esclusivamente dalla somma delle celle vicine. Es. In base al risultato della somma possiamo stabilire se la cella sarà vera o falsa, o 0 e 1, etc...

3. Regole “lecite”, un tipo di regole “totalistiche” che sono un sottoinsieme di tutte le possibili regole. Fu Wolfram il primo a schematizzare queste regole. L'idea prende in considerazione che da uno stato-zero (stato in cui tutte le celle hanno valore 0) non può emergere alcuno sviluppo.

Studiando i CA ci si rende subito conto che anche con pochi semplici regole e valori è possibile avere un altissimo numero di combinazioni. Per questo motivo gli automi cellulari sono così noti negli studi teorici. Ad esempio già nel caso 1, un CA unidimensionale binario con r=1 (raggio del vicinato) composto da tre valori, ha 2 alla terza, ossia 8 possibili stati di 1 e 0, e quindi 2^8=256 possibili regole. In generale è possibile calcolare il numero delle regole possibili con s^(s^n), dove s=numero di stati e n=numero di vicini (o meglio delle celle prese in considerazione, vicini più nodo centrale). Partendo da ciò, se osserviamo sistemi lievemente più complessi come un CA bi-dimensionale a 2 stati con “Moore Neighbourhood” e quindi n=9, abbiamo 2^(2^9) ossia 10.154 regole possibili; ed è per questo che non è possibile studiare analiticamente tutte le possibili regole del sistema. Grazie a questa varietà di possibili combinazioni i CA sono stati testati per vari usi dando spesso risultati positivi, ad esempio uno studio condotto nel 2006 mostra che le 256 regole dell'automa unidimensionale danno ottimi risultati nella generazione di numeri random e dal confronto con il GSL Random Number Generator è emerso che i risultati sono statisticamente comparabili a quest'ultimo in termini di casualità; inoltre gli automi si sono rivelati tre volte più veloci dei Random Generator. Un'altra varietà di CA sono quelli “reversibili”, sistemi che possono tornare indietro nel tempo tenendo traccia di ciò che è avvenuto precedentemente. In qualsiasi momento è possibile ritornare a un “timestep” precedente o successivo senza che le informazioni vadano perse. Proprio questa tipologia è attualmente oggetto di studi nel campo della crittografia, sebbene non abbia ancora mostrato risultati del tutto positivi; difatti le regole dei CA generano quasi sempre processi ed evoluzioni reversibili, quindi, facilmente decodificabili.

2.3 Sistemi Dinamici e classi di Automi Cellulari

Possiamo definire sistema dinamico un sistema che evolve nel tempo. Quando sia l'input che l'output si sviluppano nel corso di vari time-step. In generale l'output di un sistema al tempo t, non dipende soltanto dall'input, ma anche da una grandezza definita "stato", che conserva la storia degli input nei vari istanti di tempo. Lo studio dei sistemi dinamici ci porta alla definizione di spazio delle fasi (o spazio-fase). Lo spazio delle fasi è lo spazio di tutti i possibili valori che le variabili del sistema possono assumere. Possiamo quindi definire "stato del sistema" ogni punto nello spazio delle fasi che rappresenta dei particolari valori per tutte le variabili possibili. Quindi lo spazio delle fasi di un sistema (o spazio degli stati) copre tutti i possibili stati in cui potrebbe essere il sistema. Prendendo i punti dello stato di sistema al variare delle variabili nel tempo si nota che assumono una traiettoria nello spazio[**SCH08**]. Questa traiettoria può essere interpretata come il comportamento del sistema, che possiamo schematizzare in tre tendenze principali, definite attrazioni per il fatto che sembrano seguire una forza di attrazione verso una traiettoria:

- attrazione ad un punto fissato: il comportamento si stabilizza ad un singolo punto dello spazio-fase che resta inalterato nel tempo. Si è osservato, considerando lunghi periodi di tempo, che le dinamiche tendono a essere simili per simili stati iniziali.
- attrazione periodica: il comportamento si stabilizza in un percorso chiuso. Dinamiche di lungo periodo mostrano che da simili stati iniziali si hanno simili traiettorie di percorsi di stato.
- attrazione caotica: il comportamento non si stabilizza mai, in questo caso però lo spazio di azione è delimitato da "bordi". In questo caso simili stati iniziali non portano ad evoluzioni e dinamiche simili, questa proprietà è definita "sensibilità alle condizioni iniziali".

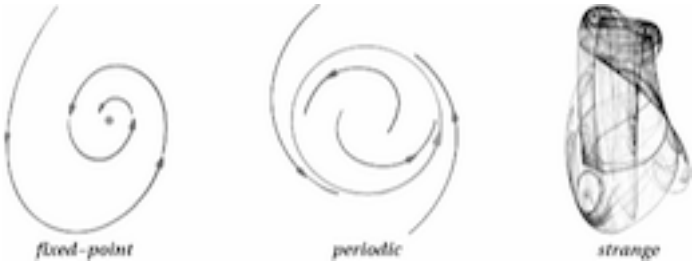


Fig.4: La figura mostra i differenti tipi di attrazione[**SCH08**]

Questa teoria sta alla base dello studio della classificazione dei CA; così come si può tracciare una traiettoria di punti nello spazio-fase rappresentante i valori delle variabili, è possibile studiare lo stato globale di un CA dalla configurazione totale corrente degli stati della cella. Come Wolfram afferma possiamo far ricadere i comportamenti di tutti (o quasi) i CA in quattro classi principali[**WOL83**]:

- Classe 1: CA che convergono velocemente verso uno stato uniforme dopo un numero definito di step.
- Classe 2: CA che velocemente convergono verso stati stabili o ripetitivi. Le strutture sono di breve periodo.
- Classe 3: CA che evolvono in stati che sembrano essere del tutto casuali. Partendo da schemi iniziali diversi creano generazioni statisticamente interessanti, paragonabili alle curve frattali. In questa classe si percepisce il maggior disordine sia a livello globale che locale.
- Classe 4: CA che formano non solo stati ripetitivi o stabili, ma anche strutture che interagiscono tra loro in maniera complicata. A livello locale si osserva un determinato ordine.

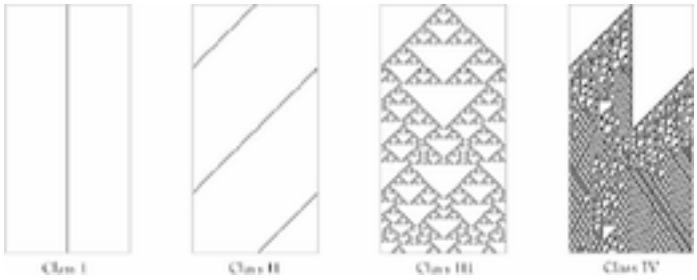


Fig.5: Ogni immagine è stata realizzata con un CA a 2 stati e con vicinato 3[**WOL02**].

Le classi si comportano differentemente a seconda degli input iniziali, la classe 1 mostra un comportamento analogo per tutte le configurazioni dopo un certo fluire di tempo, la classe 2 tende periodicamente a determinare lo stato della cella dai valori iniziali di una regione vicina ad essa e assume un comportamento prevedibile che si manifesta con strutture isolate tra loro. Da stati iniziali particolari formano strutture semplici, per questo sono utilizzati nell'editing delle immagini per la costruzione di filtri. La classe 3 evolve in configurazioni caotiche e difficilmente prevedibili a patto della conoscenza di tutti i valori della configurazione iniziale; dopo lunghi periodi di tempo mantengono le stesse proprietà statistiche. La classe 4 di CA è costituita da poche regole di transizione che generano strutture complesse nello spazio e nel tempo. Nella maggior parte dei casi osserviamo le celle cambiare stato dopo un numero finito di passi, in altri casi si osservano strutture stabili o periodiche che persistono per un numero infinito di passi o strutture che si propagano.

Il suo andamento è impredicibile per cui il miglior studio di questa classe di CA è costituito dall'osservazione delle sue evoluzioni nelle strutture. Come cambiano le strutture delle varie classi modificando il valore iniziale di una sola cella? Negli automi di classe 1 non si percepisce alcuna variazione, di fatto tutti conducono ad uno stesso stato finale dopo un certo numero di passi. Nella classe 2 si nota un lieve cambiamento in prossimità di piccole aree delimitate. La classe 3 mostra cambiamenti che poi si propagano per tutto il proseguire della struttura. Infine la classe 4 mostra un comportamento del tutto unico e imprevedibile perché al cambiare del valore anche di una sola cella si modifica l'andamento di tutta la struttura. Inoltre gli Automi Cellulari sono stati definiti capaci di Computazioni Universali. In particolar modo quelli di classe 4, tra i quali un consistente e conosciuto esempio è il "Gioco della Vita".

2.4 Computazione Universale e Automi Cellulari

Descrizione di Turing della composizione della sua macchina:

> "... Una capacità di memoria infinita ottenuto sotto forma di un nastro infinito tracciato in quadrati, su ciascuno dei quali un simbolo può essere stampato. In ogni momento c'è un simbolo nella macchina: si chiama il simbolo acquisito. La macchina può alterare il simbolo di scansione e il suo comportamento è in parte determinata da quel simbolo, ma i simboli sul nastro altrove non influiscono sul comportamento della macchina. Tuttavia, il nastro può essere spostato avanti e indietro attraverso la macchina, questo è uno delle operazioni elementari della macchina. Qualsiasi simbolo sul nastro può quindi finalmente avere un inning (il suo turno)."[**TUR36**]

In informatica una Macchina di Turing Universale è una Macchina di Turing in grado di simulare una Macchina di Turing arbitraria su valori di input arbitrari. Teoria che ha poi portato J. VonNeumann alla creazione degli automi cellulari, come componente principale di macchine in grado di auto-riprodursi. Per sostenere l'auto-riproducibilità delle macchine VonNeumann riusci a dimostrare la possibilità della creazione di un computer universale con un CA bi-dimensionale a 29 stati e (ovviamente) 5 vicini. Da ciò deriva che automi con minor stati supportano la computazione universale. Ma cosa serve a un sistema per essere capace di computazioni universali? In particolare sono tre le condizioni necessarie e sufficienti affinché un tale sistema sia costruito:

(1) Le dinamiche devono supportare l'archiviazione dell'informazione. Stati di informazione locali vengono preservati arbitrariamente per lungo tempo.
(2) Le dinamiche devono supportare la trasmissione dell'informazione. Piccoli stati di informazione locali devono potersi propagare per distanze arbitrariamente lunghe.
(3) Le informazioni archiviate e trasmesse sono abilitate ad interagire, con il possibile cambiamento di una delle due.

Da ciò evinciamo che un sistema dinamico deve poter far interagire archiviazione e trasmissione e deve poter prendere in considerazione dinamiche arbitrariamente lunghe. Difatti è stato teorizzato (da Langton) che è l'associazione tra CA di classe 4 e transienti molto lunghi a spiegare la capacità computazionale degli automi e la loro imprevedibilità nel comportamento.

2.5 Game of Life (GoL)

Il gioco della via fu inventato dal matematico John Conway nel 1970, Scelse le regole accuratamente dopo aver tentato molte possibilità. Alcune regole portavano ad una morte veloce di tutte le cellule (o celle), altre portavano a sovrappopolazione. Egli tentò di creare delle regole che bilanciassero la popolazione delle celle e che determinassero configurazioni di crescita infinita o stabilizzazione. Inizialmente questo automa cellulare era suonato a manovella e veniva impostato per ogni generazione, successivamente l'avvento dei computer portò ad una più vasta esplorazione delle sue applicazioni. Ovviamente il "Gioco delle Vita" è solo uno degli esempi di possibili automi cellulari realizzabili, tuttavia il suo studio ha destato non pochi interessi perché, come vedremo, pur partendo da semplici concetti, si è dimostrato in grado di evolvere in maniera complessa. GoL è uno dei giochi, e sistemi, più riprodotti che siano mai esistiti ed è stato rielaborato in una vasta varietà di griglie (quadrate, rettangolari, esagonali, pattern misti,..) e regole. Ovviamente non è un gioco nel senso convenzionale del termine; non ci sono giocatori e quindi vincitori o perdenti; è sufficiente piazzare le celle iniziali nella griglia e vedere cosa accade dopo. Le evoluzioni sono imprevedibili, come ci si aspetta da un automa di classe 4, e le predizioni anche se riferite a minimi cambiamenti dello stato iniziale sono tutt'altro che banali da calcolare. Può essere definito un automa cellulare di tipo totalistico e può essere implementato in vari modi[**BCG04**].

Come funziona

GoL è elaborato in una griglia di quadrati contenenti celle, questa griglia può estendersi all'infinito, o avere bordi di vario genere. Le celle al suo interno possono essere "vive" o "morte". Considerando un vicinato del tipo "Moore Neighbourhood", con 8 celle attorno alla cellula presa in considerazione vengono applicati le seguenti regole:

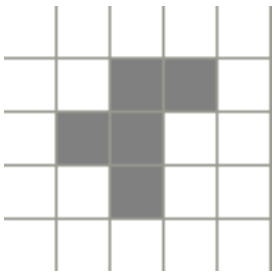
- La cella MUORE se ha nel vicinato meno di 2 o più di 3 celle vive.
- La cella VIVE se ha nel vicinato 2 o 3 celle vive.
- La cella NASCE se ha nel vicinato 3 celle vive.

Naturalmente il calcolo delle regole inizia dopo una precedente disposizione delle cellule vive.

> (In altre parole, stabilito lo stato iniziale del sistema, possiamo dedurre che una cella morta con esattamente 3 vicini nasce, una cella resta viva se ha 2 o 3 vicini, e che in tutti gli altri casi resta una cella morta.)

Pattern di Stati

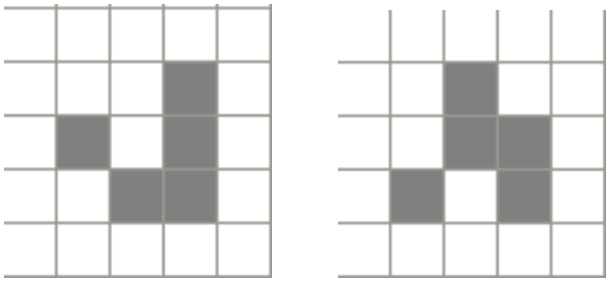
Un buon modo per studiare le regole del gioco della vita è in veste grafica con una griglia sulla quale inserire le celle iniziali, in questo modo, grazie ai moderni computer, è possibile notare in breve tempo diversi andamenti. Ad esempio con una sola cella o con due notiamo che le celle muoiono alla prima generazione, con 3 celle in riga abbiamo una rotazione ciclica delle 3 celle che può essere definita un primo oscillatore; quattro celle a blocco formano un altro blocco con un buco stabile, cinque celle producono 4 oscillatori del tipo a 3 celle, sei si sviluppano in forme geometriche per poi sparire, e così via... Negli anni vari pattern sono stati sperimentati, alcuni per rispondere a specifiche domande, altri per lanciare delle sfide. Una delle prime domande fu proprio risolta da Conway. "Può un pattern continuare a generare senza fine?" Nel 1969 egli scoprì un pattern di 5 celle (l'R-pentomino) che fu in grado di scoraggiarlo dal predire l'evoluzione finale.



.

Fig.6: R-pentomino.

Difatti questo pattern sembra avere dinamiche stabili per la prima migliaia di generazioni, ma dopo comincia ad assumere un andamento del tutto imprevedibile. Ovviamente con i moderni calcolatori questo stato si risolve molto velocemente ma all'epoca era di difficile dimostrazione. In realtà vennero scoperti molti pattern a 5 celle che avevano comportamenti interessanti, questi vengono definiti "Gilder" e paiono comportarsi come una particella base che può essere usata per costruire altre forme.



.

FIgg.7,8: Gilders.

Questi stati iniziali conducono a un cammino nello spazio della forma Gilder, fino a quando scontra con altre celle. Vennero usati all'incirca come un acceleratore di particelle, il pattern veniva lanciato contro altre forme per vedere cosa accadeva. Inizialmente persino Conway non credeva che potessero esistere pattern in grado di continuare generazioni all'infinito, tant'è che lanciò una scommesso di 50 dollari a chiunque riuscisse a fare un pattern che continuasse a crescere all'infinito. Bill Gosper, matematico e programmatore americano, riuscì a vincere la scommessa realizzando il "Gilder Gun". Ovvero un pattern che ogni 30 generazioni crea un gilder di 5 celle e lo spara verso l'infinito della griglia. Questo stato iniziale porta ad un flusso continuo di celle che se considerate come corrente elettrica, associate ad un sistema di memorizzazione degli stati e a dei vincoli logici, possono riprodurre una Macchina di Turing Universale. In parole povere ogni gilder può reagire in un determinato modo per compiere varie operazioni logiche. Da queste deduzioni susseguì poi l'era dei calcolatori digitali.

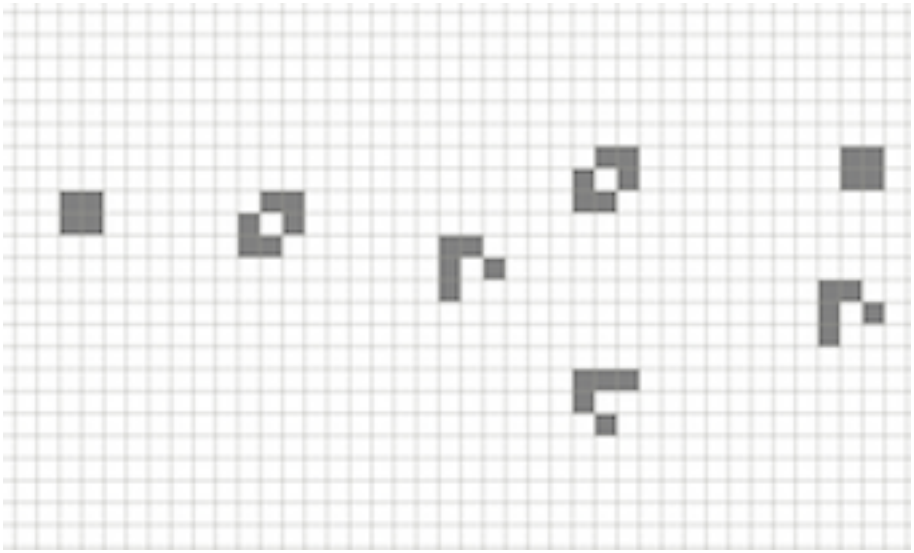
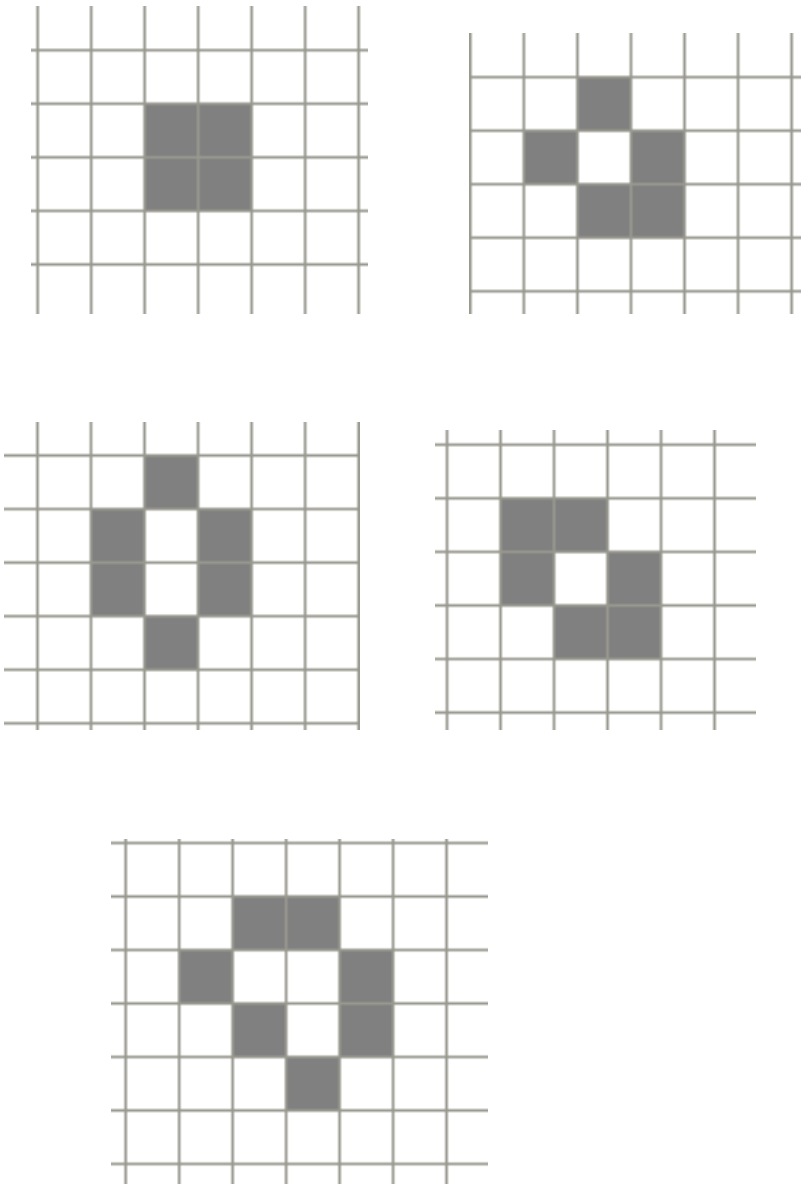


Fig.9: GilderGun.

Oggetti statici e oggetti ripetitivi

Partendo da diversi pattern iniziali possiamo osservare e distinguere almeno altri due tipi di comportamenti, uno statico e uno ripetitivo. In un comportamento statico non vengono create nuove celle vive ne quelle presenti muoiono. Questo si verifica quando ogni cella viva ha 2 o 3 vicini vivi e le celle morte non hanno mai 3 vicini vivi.

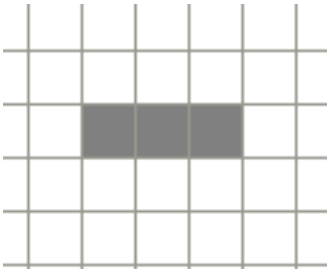


Figg.10,11,12,13,14: Oggetti statici (loop).

Come possiamo notare (dalle figure) ogni cella viva ha almeno 2 celle vicine vive e nessuna cella morta ha più di due vicini vivi. Contrariamente a queste forme statiche abbiamo delle semplici configurazioni iniziali che portano a comportamenti che cambiano si di generazione in generazione ma in modo ripetitivo. Questi comportamenti possono essere descritti come degli oscillatori a più fasi.

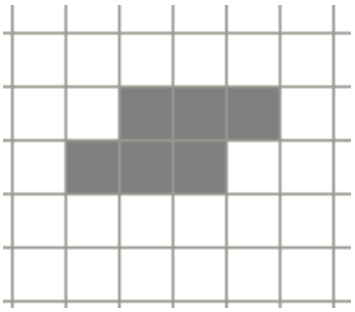
Ad esempio:

-: Il “blinker”:



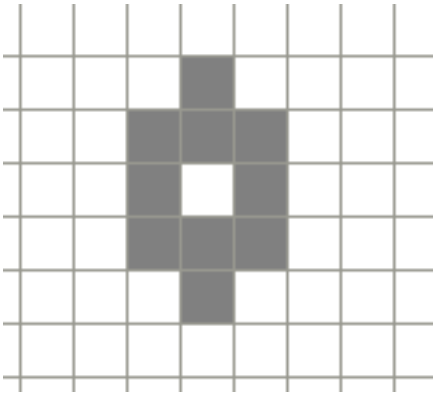
:

Fig.15: Un CA a 3 celle che può essere definito un oscillatore a 2 fasi.
-: Il “toad”:



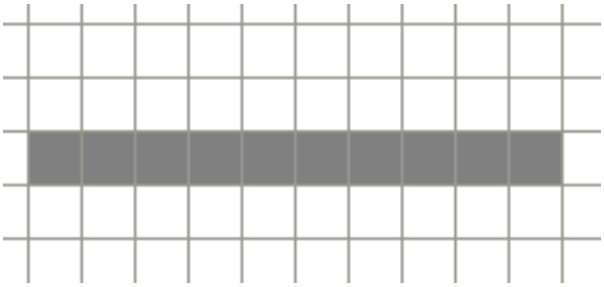
:

Fig.16: Altro oscillatore di periodo due.
-: Il “pulsar”:



:

Fig.17: Il Pulsar è un oscillatore di periodo 3.
-: Il “10 row”:



:

Fig.18: Oscillatore di periodo 15.

Escluse forme particolari, generalmente a stati iniziali casuali corrispondono stabilizzazioni nel breve o lungo periodo.

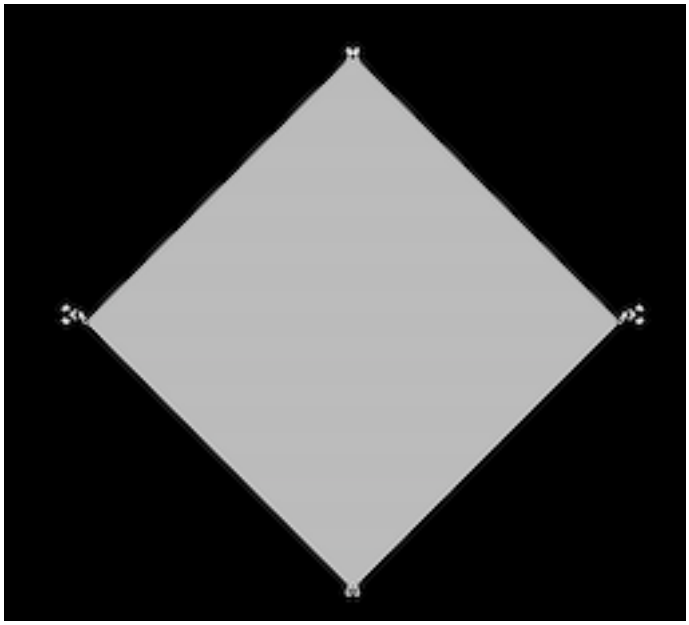
2.6 Spacefiller e Breeder

Nel Gioco della Vita un "breeder" è un pattern che mostra una crescita esponenziale manifestata con la generazione di molteplici copie di pattern secondari, terziari. I breeder possono essere classificati in base al modo in cui si muovono i pattern. Ne abbiamo quattro tipologie:

- "Gun": si comporta sia come un oscillatore che come un generatore di nuovi pattern. E' costituito da un pattern secondario stazionario e due pattern, primario e terziario, in movimento.
- "Puffer": può essere definito anche "puffer train", proprio per indicare il fatto che si muove lungo una linea lasciando una scia di pattern secondari. Vi sono due periodi, quello della figura principale e quello della "scia".
- "Rake": simile al puffer ma genera nel suo progressivo avanzare pattern nello stile "gun". Sono stati realizzati sia rake di puffer che rake di rake. Notevole è la crescita quadratica che si manifesta.

Fig.19: Esempio di "Breeder", il movimento in questo caso scorre da destra verso sinistra (Immagine realizzata con il software Golly 2.3).

Uno "Spcefiller", una delle scoperte più recenti nel campo degli automi cellulari (1993), può essere definito anch'esso un pattern a crescita esponenziale, ma viene considerato come una quinta classe di breeder. A differenza di quest'ultimo lo spacefiller riempie tutto lo spazio circostante in maniera omogenea, e gli oggetti creati appartengono ad un unico spazio, anziché essere indipendenti. Grazie a questa sua caratteristica ben si adatta a studi sull'immagine o sul popolamento dei pixel negli schermi.



.

Fig.20: Esempio di “Spacefiller” (Immagine realizzata con il software Golly 2.3).

Lo spacefiller è il pattern con la maggior crescita in assoluto. In oltre il riempimento è al massimo possibile, ossia 1 cella su 2 è viva. Prima della loro scoperta i breeder detenevano il primo posto come crescita, successivamente lo spacefiller con stati di partenza molto meno complessi ha preso il sopravvento.

2.7 Supercolliders

I Gilder nei CA unidimensionali sono gruppi compatti di pattern non-quiescienti e non-periodici che translano lungo una matrice. Questi elementi mostrano particolari intertazioni riconducibili a fenomeni di fusione, collisione elastica e collisione solitonica (il solitone è un onda solitaria auto-rinforzante). Per questo motivo sono oggetto di studio per la crezione di computer non convenzionali che meglio si adatterebbero allo studio dei fenomeni naturali e alla possibilità di calcolo parallelo.

Questa intuizione venne a Friedkin e Toffoli nel 1970, quando svilupparono un concetto di computazione generica basato sull'interazione ballistica di quanti di informazione rappresentati da particelle astratte. Queste furono chiamate supercolliders. Gli stati booleani delle variabili logiche sono rappresentati da palle o atomi, che preservano la loro identità durante la collisione tra loro. L'idea prende spunto da un ambiente simile al biliardo, palle che si scontrano e che rimbalzano su bordi mirror. Utilizzando CA partizionati su una griglia di vicinato Margolus si osservano comportamenti riconducibili a computazioni di tipo universale.

Anche un approfondito studio della "Unconventional Computing Centre" in Regno Unito nel 2011 ha dimostrato che è possibile creare computer non-convenzionali capaci di calcoli paralleli del tutto innovativi[**MASH11**].

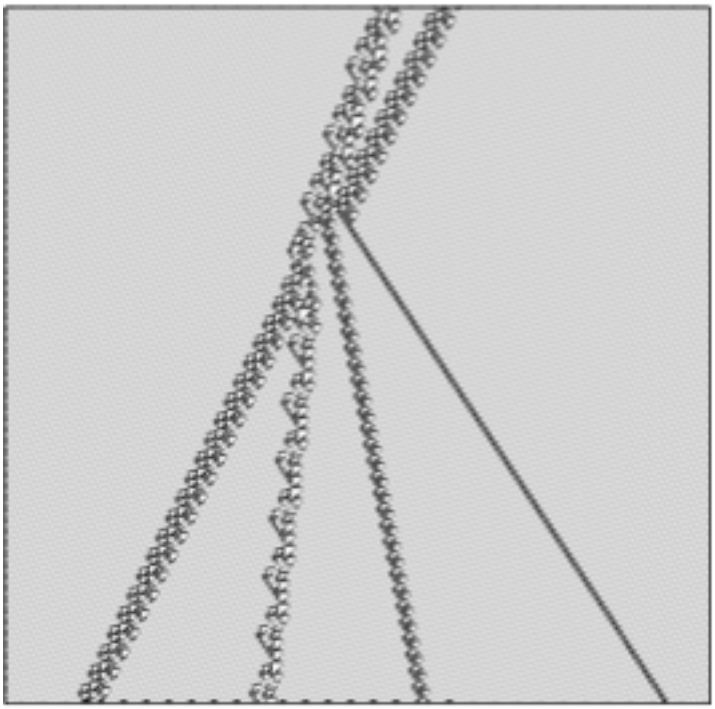


Fig.21: Supercolliders[**MASH11**].

2.8 Auto-riproducibilità

Uno dei primi problemi che espose Von Neumann, e che risolse, fu: che tipo di organizzazione logica è sufficiente ad un automa affinchè controlli se stesso in modo da potersi auto-riprodurre?

La questione fu posta pensando ai naturali sistemi di auto-riproduzione, ma la volontà non era quella di imitare questi sistemi dal punto di vista genetico o biochimico, piuttosto di astrarre da essi la logica per poter indirizzare le macchine.

Von Neumann approcciò il problema dal punto di vista logico matematico affermando che: se l'auto-riproduzione è il risultato di movimenti di complesse macchine biochimiche, allora il comportamento di queste macchine, seppur naturali, sarà descrivibile come sequenza logica di avvenimenti, e quindi esprimibile sotto forma di algoritmo.

Se è vero che gli algoritmi possono essere eseguiti da qualsiasi macchina, allora è anche vero che possono essere eseguiti da una macchina di Turing. Dunque è plausibile pensare a una macchina di Turing che esegue algoritmi determinati dalle naturali leggi di riproduzione degli organismi, e quindi a una macchina in grado di auto-riprodursi.

Von Neumann fu in grado di esibire le sue teorie su una macchina di Turing universale, adeguatamente modificata, in grado di costruire l'output derivante dall'input descritto dal nastro. Questa macchina venne definita "Costruttore Universale", in quanto era in grado di costruire qualsiasi altra macchina e di inserirle i dati input del nastro.

In questo modo possiamo pensare all'auto-riproduzione come il caso in cui la macchina legge quella parte di nastro dove è descritto il "Costruttore Universale" stesso. Il risultato sarà la copia della macchina stessa[**LAN84**].

Ovviamente quando si parla di auto-riproduzione, inizialmente, non viene fatto riferimento propriamente agli automi cellulari, ma a degli automi che svolgono un limitato numero di funzioni dedicate. Difatti Von Neumann in principio riprodusse un meccanismo cinematico.

Partendo da nuclei considerati come elementi computanti, e quindi in grado di eseguire gli switch logici (and, or, not) e dei delays (ritardi), e simili a neuroni in quanto attivi se stimolati; egli aggiunse altri cinque tipi primitivi di elementi: un elemento cinematico (simile ad un muscolo) in grado di muovere gli elementi, un elemento separatore in grado di dividere due elementi, un elemento saldatore in grado di unire gli elementi, un elemento rigido in grado di creare strutture e un elemento sensibile capace di riconoscere e comunicare gli elementi.

Si può far funzionare questa macchina in un ambiente composto dello stesso tipo di elementi che fluttuano in movimenti casuali e osservare l'evoluzione del sistema, in un alternarsi di nuove concatenazioni e casi di auto-riproduzione. Questo, probabilmente, (similmente a ciò che accade nell'universo) evolverebbe in un sistema di sistemi cinematici automatizzati.

Tuttavia la creazione di un simile meccanismo implica non pochi problemi, come ad esempio la potenza che dovrebbe avere ogni elemento e la capacità di ognuno di essi a relazionarsi, in base a delle regole, con gli altri elementi del sistema. Inoltre abbiamo dinanzi un insieme di elementi che non possono essere definiti propriamente semplici o atomici.

Molti sostengono che per questi motivi Von Neumann si dedicò agli Automi Cellulari veri e propri, in realtà fu anche per vincere la sfida del calcolo parallelo. Comunque sia morì giovane e non potè continuare e pubblicare interamente i suoi studi, fortunatamente lo fece un suo caro amico (Arthur W.Burks).

Nonostante queste scoperte è strano notare come la storia recente manchi di esempi lampanti di meccanismi di auto-riproduzione cinematica, per lo meno a scopo civile. Forse perchè pare un principio anti-economico..(!?).

Un loop di self-reproduction

Inizialmente Von Neumann creò per la macchina un array cellulare con celle a 29 stati e vicinato 5. Effettivamente questo genere di array è capace di costruzioni universali ma è estremamente complesso da realizzare. Nel 1968 E.F.Codd mostrò la possibilità di realizzare un array capace di auto-riproduzione basato su celle a 8 stati e 5 vicini. La struttura sulla quale egli basò l'intera macchina è il data-path[**LAN84**].

Il data-path consiste in una stringa di celle in stato 1, le celle nucleo, circondate da celle di stato 2 , le celle guaina. Il data-path, come il nome suggerisce (cammino di dati), è capace di trasmettere dati in forma di "segnali". Un segnale è formato da due stati che viaggiano simultaneamente: un primo stato, che definisce poi il nome, rappresentato da un numero (stato 4,5,6,7) , seguito dallo stato 0. Il pachetto è così formato dallo stato X seguito dallo stato 0 .

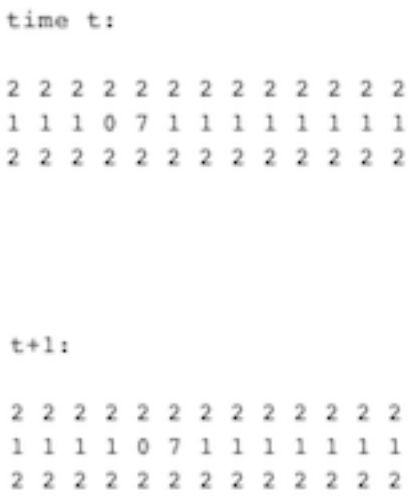


Fig.22: Data-path[**LAN84**].

Il data-path lungo il percorso può diramarsi o aprirsi a ventaglio, in entrambi i casi i segnali sono duplicati, e proseguono ognuno per la propria diramazione. I segnali sono raggruppati in sequenze trattate come istruzioni per effettuare determinate azioni; come ad esempio la ritrazione o l'allungamento del data-path verso precise direzioni.

Fig.23: Data-path[**LAN84**].

La macchina di Codd dovrebbe funzionare con un data-path usato come braccio per leggere lungo un nastro composto da una sequenza di 0 e 1. Il cambiamento 0 e

1 è percepito dal braccio ed è quindi possibile successivamente decodificare queste sequenze in sequenze di istruzioni.

Queste sequenze di istruzioni causano la formazione di un altro data-path: il "braccio costruttore", che può estendersi in un area vuota del nastro e muoversi avanti e indietro per settare le celle in modo da configurarle come la macchina descritta sul nastro. Successivamente un segnale di start viene inviato alla nuova macchina e il braccio viene rimosso.

Codd inoltre implementa un altro importante elemento, l"'emettitore periodico", che scandisce il tempo della macchina. L'emettitore non è altro che un data-path riprodotto a loop che genera un duplicato del segnale al giungere della giunzione a T. In questo modo un segnale procede lungo il nuovo datapth e un altro torna nel loop, scandendo ad ogni ciclo un timestep del sistema.

Fig.24: Emettitore periodico[**LAN84**].

Questa forma di data-path è molto importante nel sistema di Codd, difatti non è soltanto un modo per scandire il tempo ma può anche essere usato come zona di memoria, in quanto può contenere un ciclo all'infinito.

In questo modo è possibile conservare programmi in maniera dinamica, e non statica come in un nastro. Non avendo il problema di dover muovere il nastro avanti e indietro, la realizzazione della macchina risulta facilitata e con essa anche la sua riproducibilità.

Grazie a questa tecnica possiamo costruire facilmente molte macchine, anche complesse, in grado di eseguire le più svariate operazioni; basta trovare il loop della dimensione adeguata. Per sempio se inseriamo la sequenza di estensione del cammino in un loop, abbiamo una macchina che estende il suo data-path all'infinito.

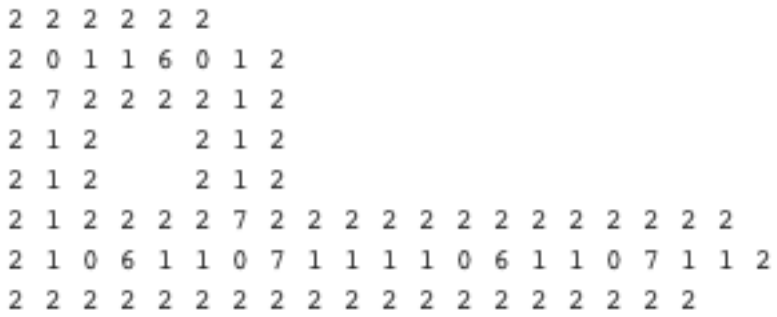


Fig.25: Data-path estendibile all'infinito[**LAN84**].

La macchina di Codd è effettivamente capace di costruzioni universali tuttavia fu successivamente modificata da Langton.

Langton alterò la capacità di costruzione universale della macchina per concentrare la sua attenzione alla creazione di loop capaci di maggior storage e quindi in generale, più potenti.

Quello che fece Langton fu modificare le regole di transizione che determinano il comportamento delle configurazioni dell'array, in modo da costituire strutture e giunzioni capaci di adattarsi meglio alle dimensioni del loop richiesto.

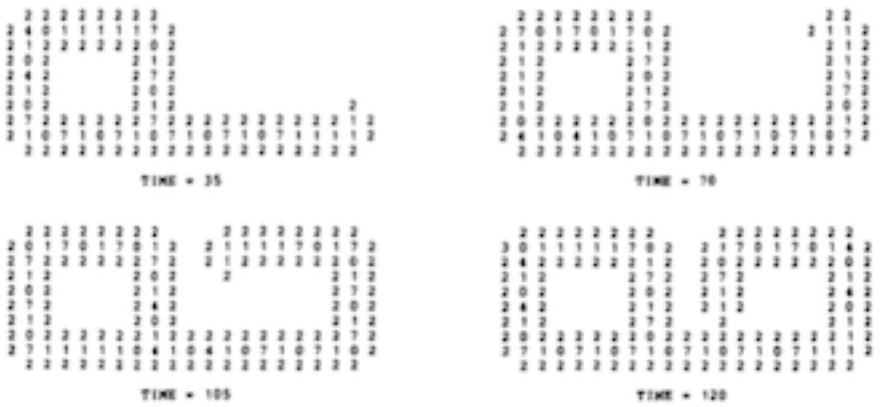


Fig.26: Data-path di Langton[**LAN84**].

2.9 Tipologie di Automi Cellulari

Analizzando un Automa Cellulare notiamo che è costituito da input, output e funzioni di transizione che ne determinano l'andamento. Ma formalizzando, possiamo definire un Automa Cellulare come una quintupla di valori (X, Y, Q, t , s) dove:

- X è l'insieme finito dei simboli di input
- Y è l'insieme finito dei simboli di output
- Q è l'insieme degli stati interni dell'automa
- t : Q x X Ⓡ Q è la funzione di transizione di stato che programma la trasformazione degli stati in funzione dell'input
- s : Q x X Ⓡ Y è la funzione di uscita che programma l'uscita in funzione dell'input e dello stato interno.

Da ciò possiamo evincere come: ogni AC sia completamente noto se si conosce la modalità con la quale reagisce agli input; e che la sua effettiva realizzabilità è legata alla possibilità di rappresentare un numero finito di stati.

Durante la storia sono stati fatti svariati tentativi di modifica ai valori della sua struttura, in particolar modo sono state espresse varianti alla funzione di transizione. Le più note varianti di Automi Cellulari sono:

- Automi cellulari non deterministici: sono caratterizzati da una funzione di transizione, che consente di scegliere uno stato in modo non deterministico, tra una più ampia scelta di stati possibili.
- Automi cellulari probabilistici: viene stabilito lo stato in cui si troverà una cella all'interazione successiva, assegnando un valore probabilistico ad ogni stato in base alla configurazione delle celle vicine. Vengono spesso usati nella simulazione di fenomeni naturali.
- Automi cellulari partizionati: negli AC partizionati troviamo una più semplice funzione di transizione in quanto non viene posta come necessaria la conoscenza di tutti gli stati delle celle vicine, come nel modello standard, per modificare lo stato della cella corrente.
- Automi cellulari asincroni: in questo genere di AC una cella può decidere ad ogni iterazione, in maniera non deterministica, se cambiare il proprio stato oppure mantenere lo stato corrente. In questo modo a differenza del modello standard non tutte le celle dovranno per forza agiornare il proprio stato ad ogni iterazione, questo verrà stabilito in maniera non-deterministica.
- Automi cellulari gerarchici: la loro caratteristica è il fatto di avere celle

con funzioni non atomiche; ossia ogni cella può avere più stati e quindi il suo stato finale dipende dallo stato delle sue parti. Questo metodo viene usato per simulare sistemi biologici multilivello.

- Automi cellulari inomogenei: sono costituiti da più funzioni di transizione, ogni cella può avere una sua funzione distinta, oppure possono distribuirsi le funzioni in aree separate nella griglia, o ancora una cella può cambiare funzioni col passare dei timestep. Sono stati impiegati nello studio di sorgenti di particelle o comportamenti dei vulcani.

2.10 Mobile Automata e Network Mobile Automata

Una delle caratteristiche degli Automi Celullari è che schemi contenenti evoluzioni basate sul colore sono processati con sistemi di calcolo parallelo ad ogni step della loro evoluzione. Ma è davvero importante simulare alcuni comportamenti con questo sistema o ci sono metodi più economici in termini computazionali? Una delle risposte potrebbero essere gli Automi Mobili.

Come per gli Automi Cellulari, gli Automi Mobili consistono di una stringa di celle ognuna delle quali ha 2 possibili colori. La differenza consiste nel fatto che il Mobile Automata ha una sola cella attiva per ogni step e delle regole che stabiliscono come questa cella deve muoversi da uno step all'altro.

Le regole del Mobile Automata specificano in che modo il colore di questa cella attiva deve essere aggiornato (solitamente viene considerato il colore della cella attiva e quella del suo immediato vicino) e se la cella andrà verso destra o sinistra. Le possibili regole, come per gli automi cellulari, possono essere formalizzate.

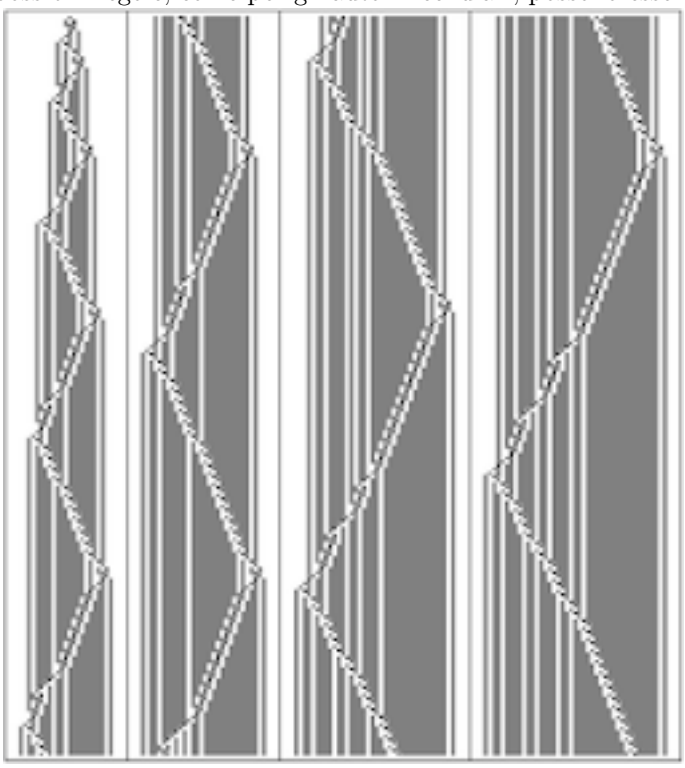


Fig.27: Mobile Automata[**WOL02**].

Il Mobile Automata può essere considerato un ibrido tra gli automi cellulari elementari e la macchina di Turing.

Un analogo del Mobile Automata può essere definito per i networks settando un singolo nodo attivo e stabilendo regole che rimpiazzano i cluster di nodi attorno al nodo attivo muovendone la posizione. Questo sistema può essere usato, oltre che per simulare reti di automi mobili, per implementare dei veri e propri sistemi di rete mobili[**WOL02**].

Cresce sempre più il bisogno di reti altamente dinamiche e con essi la necessità di progettare algoritmi che constantemente ne cambino l'assetto, ma questa tipologia di algoritmi è tutt'altro che scontata ed è quindi sempre più necessario un meccanismo dinamico di aggiornamento dei sistemi di comunicazione. Recentemente, uno studio condotto dal Department of Computer Science di Israele e Cambridge, ha condotto alla scoperta di un metodo di astrazione per le reti mobili chiamato Timed Virtual Stationary Automata composto di reali clients mobili, un sistema di automi virtualizzato e tomporizzato chiamato Virtual Stationay Automata (VSAs) e un servizio di comunicazione per far interagire clients e VSAs. Il VSAs è un sistema di macchine virtuali temporizzate fisse, con un esplicita nozione di real-time, distribuite in punti conosciuti del piano ed emulate da reali nodi mobili nel sistema. Ogni VSA rappresenta un'area geografica predeterminata ed ha capacità di trasmissione simili ai nodi mobili, riuscendo quindi a comunicare con quest'ultimi. I VSAs sono collacati in regioni prefissate, come piastrellassero il piano, definendo un'infrastruttura virtuale statica. Questa infrastruttura può essere facilmente adattata a diverse tipologie di rete e può essere implementata con facili algoritmi.

Ad esempio sono stati associati ai nodi algoritmi di ricerca geografica per la localizzazione dei nodi vicini e sono state così create reti ad-hoc; solitamente usate con protocolli che associano ogni parte di dato a regioni prefissate della rete e preservano i dati in nodi specifici di quella regione.

2.11 Perché gli Automi Cellulari sono così interessanti

L'interesse principale per gli automi cellulari è dovuto al fatto che essi forniscono uno strumento matematico utile per la risoluzione di problemi fisici e naturali troppo complessi per essere affrontati tramite gli strumenti matematici tradizionali e quindi offrono la possibilità di esplorare nuove vie per la sperimentazione.

Molti studiosi sono dell'opinione che le applicazioni più significative degli automi cellulari si avranno nella produzione di modelli in grado di simulare il comportamento intrinseco distribuito e di auto-organizzazione. Già in passato infatti si era intuita la possibilità che alcuni meccanismi potessero svolgere funzioni tipicamente umane, come alcune elementari attività mentali.

Come recentemente dimostrato ogni cellula del nostro corpo contiene una struttra nota come citoscheletro, che ne forma la struttura portante. La forma di questa struttura tuttavia non è fissa, ma mobile, ed è composta da microtubuli che sembrano organizzare l'attività della cellula; le proteine molecolari, sotto unità dei microtubuli, sembrano organizzarsi in schemi riconducibili agli Automi Cellulari. Questo lascia pensare che, probabilmente, la vita si basa su forme organizzative di questo genere[**LUC03**].

Il comportamento delle cellule umane non è l'unico esempio che si riscontra in natura di automi cellulari; difatti sempre più si scopre che l'interazione tra molte materie dell'universo avviene con questi processi: la formazione delle rocce, della vegetazione, delle galassie, degli atomi, etc.

L'applicazione degli automi, quindi, offre la possibilità di analizzare e allo stesso tempo osservare e sperimentare l'evoluzione di molti sistemi; esempi concreti sono la modellazione dei gas e dei fluidi in genere, lo sviluppo urbanistico, o il processo delle immagini.

Il progetto

3.1 Premessa

Si è voluto realizzare un semplice programma in Java, che funge da arpeggiatore per un sintetizzatore midi. O meglio è stato realizzato un piccolo software in stile "Game of Life" per offrire la possibilità di realizzare musica in maniera "casuale" ma allo stesso tempo armonica. Lo scopo del programma è di dimostrare che anche in campo artistico le macchine se adeguatamente programmate possono esprimere strutture di dati che potremmo definire "creazioni", o più precisamente riproduzioni in stile frattale di pattern che trovano poi riscontro in natura, nelle scienze, o nel funzionamento dei centri neuronali. E si vuol quindi puntare l'attenzione su ciò che è veramente la creatività dell'uomo, se è un qualcosa che nasce senza alcuno schema o se è un qualcosa che emerge da iterazioni logiche che seguono delle regole. Ciò che si vuol insinuare è: le creazioni artistiche dell'uomo sono sempre una "pura creazione" o sono il risultato di riproduzioni casuali di pattern esistenti e che consciamente e inconsciamente, durante il corso della vita, sono stati "captati"?

Sicuramente la risposta a questa domanda è tutt'altro che scontata, e quasi certamente questa piccola dimostrazione lascierà delle incertezze. Tuttavia anche i piccoli risultati di questo semplice algoritmo lasciano intuire la possibilità di realizzazioni stupefacenti.

3.2 Synth Automaton

Questo software in Java altro non è che una rivisitazione musicale del Gioco della Vita di J.H.Conway. E vuol essere sia una dimostrazione come precedentemente detto, sia un tributo al primo automa cellulare che emetteva, come output delle generazioni create, dei segnali sonori. Partendo da una griglia rappresentata da una matrice bi-dimensionale sarà possibile posizionare a scelta le celle di partenza, queste, fornite come input al sistema, saranno il perno iniziale sul quale l'Automa Cellulare andrà a computare l'evoluzione. Come nel gioco della vita il gioco consiste nello stare ad ascoltare quello che succede. Anche se apparentemente privo di utilità aggiunta al progetto visivo l'evoluzione sonora, unita a quella grafica, porta l'utente interessato a scoprire nuovi pattern ad avere un nuovo punto di percezione, e quindi un gusto aggiunto che non si limita ad un evoluzione grafica, ma che da peso anche all'armonia sonora.

Possiamo definire Synth Automaton un arpeggiatore, in quanto il ruolo da esso svolto, è si quello di mostrare la realtà, il susseguirsi delle generazioni e delle regole del gioco, ma anche quello di fungere da esecutore di melodie definite o precedentemente determinate.

Difatti, grazie all'utilizzo di già note figure utilizzate in Automi Cellulari di questo tipo, e alle quali è stato già fatto riferimento, è possibile settare il programma in modo da eseguire loop statici o periodici, a seconda della configurazione iniziale. E' inoltre possibile utilizzare contemporaneamente più Synth Automaton, in modo da poter intersecare a proprio gusto i vari arpeggi ottenuti. Per mancanza di tempo e per semplicità implementativa è stato scelto un unico suono di base rappresentato dal suono di un pianoforte (lo standard per il midi); ovviamente con una piccola modifica è possibile aggiungere tutti gli strumenti delle soundbank del pacchetto java.sound.MIDI, in questo modo l'espandibilità del suo utilizzo cresce, e con essa la possibilità di creare veri e propri arrangiamenti.

Come mostrano anche semplici esempi del suo funzionamento, le melodie generate non hanno una logica di tipo musicale o armonico vera e propria, e non seguono degli schemi rigidi per l'evoluzione delle note, ma seguono semplicemente l'andamento del gioco della vita. Il motivo per il quale sono state scelte queste regole in particolare sono le stesse per le quali Conway le sclese: mostrano una vita medio-lunga dei pattern e non tendono generalmente nè a sovrapopolazione nè a cali drastici di quest'ultima.

Nel programma è possibile scegliere degli schemi prestabiliti sui quali le celle (o note in questo caso) andranno a popolarsi, oppure è possibile lasciare lo schema libero. Questi schemi altro non sono che i modi delle scale musicali applicati, con

delle limitazioni, alla matrice in modo da evitare quella fascia di note non presenti nel modo prestabilito.

E' possibile poi impostare una velocità d'esecuzione espressa in millisecondi che andrà a modificarsi in proporzione al valore iniziale immesso, in base alla posizione sulla matrice della cella. In questo modo avremo un alternanza di diversa durata delle note. Con l'applicazione dei modi e di un andamento non fisso dei timestep, si può già avere una prima impressione di improvvisazione da parte della macchina, in quanto già la semplice mancanza di dissonanze troppo evidenti e la concatenazione di note in scala porta ad un buon risultato uditivo (da non trascurare è poi il gusto personale o l'abitudine all'ascolto, che porta ad apprezzare a volte anche le melodie più "dissonanti"); ovviamente l'utilizzo di tecniche più raffinate e metodi che indirizzano anche i piccoli cambiamenti, o l'utilizzo di più tipologie di Automi Cellulari intersecati tra loro, possono portare a più precise armonie, o alla riproduzione di un vero e proprio gusto musicale da parte dell'automa.

Ad esempio, recentemente, è stato realizzato un software Java chiamato AutomatousMonk, in grado di generare improvvisazioni nello stile del pianista Theloniuos Monk, che utilizza frammenti di sequenze generate con CA o frammenti di pattern per eleborare melodia e ritmo; questo è un'altra buona dimostrazione del fatto che il gusto o le scelte "artistiche e creative" possono essere schematizzati, analizzati e riprodotti, e che sono quindi, in molti casi, di origine meccanica.

3.3 Composizione del software

Synth Automaton è costituito da due classi principali, la classe SynthAutomaton stessa, che estende l'utilizzo degli Applet e implementa la possibilità di Runnable, e la calsse CA che estende l'utilizzo del Canvas e contiene la collezione dei metod necessari al funzionamento sia dell'automa cellulare, sia del sintetizzatore.

Il componente Canvas rappresenta un'area rettangolare dello schermo entro il quale è possibile disegnare o inserire dei valori che andranno in input all'evento generato, o scelto, dall'utente; quindi la classe CA contiene anche l'inizializzazione grafica della matrice di grandezza 12x12 che andrà a costituire la griglia del "Gioco della Vita" e dalla quale saranno estrapolati i valori iniziali sui quali si baserà il calcolo della generazione successiva di celle.

Di seguito, il diagramma delle classi:

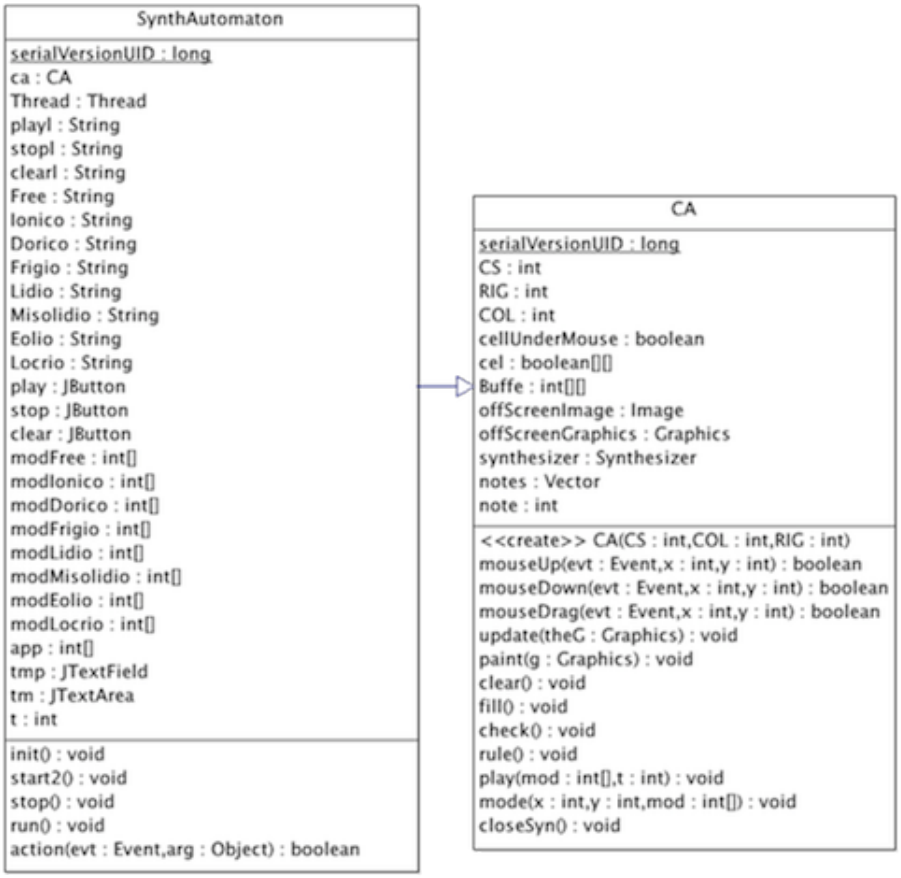


Fig.28: Diagramma delle Classi.

La classe SynthAutomaton

Nella classe SynthAutomaton vengono inizialmente instanziate tutte le variabili che andranno a costituire la parte grafica dell'Applet, il Thread che partirà nel run(), e degli array contenenti i valori delle ordinate che detemineranno il "modo" d'esecuzione. Scorrendo il codice troviamo:

Il metodo init(): Qui viene inizializzata la grafica dell'Applet; vengono impostati i valori della dimensione della matrice (pari a 12x12) e delle celle e vengono creati due pannelli principali sui quali andranno a posizionarsi i tasti per i comandi e le sclete sul primo e la griglia sull'altro, entrambi vengono aggiunti al Layout dell'Applet.

In questo metodo sono contenuti anche gli ActionListener atti a interpretare la funzione dei tasti di comando principali (play, stop, clear) che fanno riferimento direttamente ai metodi start() e stop() del Thread.

I metodi start() e stop(): Come intuibile sono i metodi che gestiscono l'avvio e l'arresto del Thread, ovvero del metodo run(). Il funzionamento è semplicemente basato sull'assegnazione del Thread != o == a "null".

Il metodo run(): Il metodo run() rappresenta l'esecuzione vera e propria del programma in quanto contiene tutti i metodi necessari al funzionamento dell'Automa Cellulare e del player del sintetizzatore. Il metodo viene mantenuto attivo da un ciclo while che prosegue la sua esecuzione fino al cambiamento dello stato del Thread a "null" e scorre in sequenza i seguenti metodi:

- fill(): per riempire il Buffer con il vicinato della cella.
- check(): per controllare la presenza di celle ai bordi e correggerle.
- rule(): per dettare le regole del Gioco della Vita.
- play(): per eseguire il risultato sul sintetizzatore
- repaint(): per aggiornare l'evoluzione grafica.
- closaSyn(): per chiudere il sintetizzatore.

Gli ultimi tre metodi sono racchiusi in un try/catch per gestire le eccezioni che potrebbero verificarsi a livello del MIDI. Inoltre nel metodo è incluso il comando per prendere in input il valore del tempo espresso in millisecondi.

Il metodo boolean action(evt, arg): Questo metodo è costituito da una serie di "if else" che andranno a determinare la scelta dei "modi armonici" da applicare presenti sul menù a scelta dell'Applet.

La classe CA

La classe CA (Cellular Automaton), come precedentemente accennato, contiene tutti i metodi per il funzionamento del programma, l'inizializzazione grafica del Canvas, il suo aggiornamento, e la gestione delle azioni del mouse.

Inizialmente vengono instanziate le variabili necessarie al Canvas, il sintetizzatore, un array contenente i valori delle note per il sintetizzatore, e due array di matrici, uno destinato a contenere i valori booleani della griglia sulla quale verranno presi gli input e visualizzati gli output, e l'altro utilizzato come contenitore dei valori di vicinato della cella presa in considerazione.

Successivamente viene gestito con i metodi mouseUp(), mouseDown(), mouseDrag(), il posizionamento delle celle sulla griglia (cell[][]) contenente i valori booleani stabiliti in base alla scelta risultante dalla pressione o dal rilascio del mouse.

Scorrendo il codice troviamo i seguenti metodi:

paint(): Il metodo paint() come intuibile disegna la griglia sulla quale saranno disposte le celle. Partendo dai valori di grandezza della matrice e delle celle, traccia prima il fondo del Canvas e in seguito la griglia di partenza.

Successivamente scorrendo la matrice booleana cell[][] disegna sulla griglia le celle attive, ossia i valori rappresentati dal flag "true".

update(): Il metodo update semplicemente serve per aggiornare la grafica ad ogni iterazione.

clear(): Questo metodo ripulisce la griglia settando tutti i valori della matrice cell[][] a "false". Due cicli "for" provvedono a scorrere tutti i valori delle x e delle y della matrice.

fill(): Il metodo fill() lavora con la matrice Buffer[][], che serve a contenere tutti i valori dell'intorno della cella (in questo caso un valore di vicinato di tipo Moore Neighbourhood e quindi 8 celle: 9 meno quella centrale). Inizialmente, come nel metodo precedente, vengono settati tutti i valori del Buffer[][] a zero; in questo modo non si avranno accavallamenti delle analisi precedenti. Immediatamente dopo il Buffer viene riempito dei valori dell'intorno della cella presa di volta in volta in considerazione, ossia [x-1][y-1], [x][y-1], etc..„ i valori dell'intorno vengono aggiunti nel Buffer se trovano una corrispondenza true nella matrice cell[][].

check(): Questo metodo provvede al controllo delle celle inserite nel Buffer., o meglio verifica prima di inserirle che facciano parte della griglia vera e propria. Un ciclo while mantiene attivo il controllo finché il metodo è in esecuzione, e in presenza di una cella true iniziano i controlli realizzati con if clauses annidati. I primi due "if" controllano che x e y siano maggiori di 0 e quindi se è possibile aggiungere al Buffer il "vicino" in alto a sinistra, ossia posizione [x-1][y-1]; se oltre alle prime due condizioni la y è minore del numero delle righe-1 possono aggiungersi anche le celle in posizione [x-1][y+1] e [x-1][y], ossia la cella in basso a sinistra e quella immediatamente a sinistra, in quanto se il valore di y è compreso tra lo 0 e RIG-1 e il valore di x è maggiore di 0 allora sicuramente avrà nell'intorno celle che non sono valori di bordatura appartenenti al bordo verticale sinistro. Seguendo la medesima logica il controllo prosegue per controllare la presenza di valori di bordatura: se la x è minore del numero di colonne -1 e y e minore del valore delle righe -1 allora è possibile aggiungere la cella nell'angolo in basso a destra, se in aggiunta la y è maggiore di 0 allora anche le celle in posizione [x+1][y-1] e [x+1][y]; successivamente se y è maggiore di 0 allora la posizione [x][y-1] viene inserita e in fine se la y è minore del numero di righe -1 si aggiunge [x][y+1]. Dopo questo ciclo di controllo vengono inizializzate due variabili (dx, dy) utilizzate per reindirizzare le posizioni in caso di cella risultante al bordo. Gli if else, similmente alla logica precedente, servono a determinare di quale bordo si tratta e stabiliscono un valore di dx e dy che possa correggere la traiettorie reindirizzandola nella griglia; possiamo definirlo come uno specchio riflettente, è un sistema chiuso.

rule(): Come il nome stesso lascia intuire il metodo rule() determina le regole del CA, le regole espresse in questo metodo sono riconducibili alla regola 110 di Wolfram, meglio conosciuta come "Gioco della Vita" di J. Conway. Le regole sono state implementate con un semplice switch che in base al numero di celle presenti nel Buffer (ossia nel vicinato) stabilisce se la cella nasce, vive o resta morta. Come noto, e come precedentemente esposto, nel gioco della vita se una cella morta ha 3 celle vive nell'intorno allora vive, se ha 2 o 3 vicini vivi resta viva altrimenti

permane lo stato di morte. Come è possibile intuire dalla matrice cell[][] lo stato di morte o vita della cella è rappresentato dai valori booleani, quindi lo switch presenta tre casi, uno di default nel quale la cella viene impostata false, un case 2, nel quale nn svolge alcuna operazione, e un case 3 nel quale dichiara la cella true. Il case 2 rappresenta il caso in cui nel Buffer vi sono 2 celle dell'intorno e quindi non deve fare alcuna operazione, in quanto se la cella è morta resta morta, se è viva deve restare tale. Il case 3 invece dichiara la cella true nel caso in cui l'intorno (Buffer) contiene 3 celle vive. Questo metodo rappresenta il vero cuore dell'automa cellulare e può essere facilmente reimpostato per simulare altre regole.

play(): Il metodo play() permette la riproduzione sonora del risultato del Gioco della Vita, e rappresenta l'innovazione. Nel metodo vengono inizializzati il sintetizzatore della classe java.sound.midi, e il MidiChannel per permettere il passaggio e l'ascolto dei suoni uscenti dal synth.

Nel metodo viene fatta scorrere la matrice booleana cell[][] per captare la presenza di celle settate "true", successivamente viene invocato il metodo mode() della stessa classe, così il "Vector" notes, il vettore contenente i valori delle celle espresse in note (associate ad un numero intero rappresentate la nota della tastiera midi), può essere riempito di tutti i valori contenuti nella matrice.

Il vettore notes viene poi fatto scorrere da un "ciclo for" che, per ogni nota contenuta nel vettore, attiva sia due funzioni switch, sia il canale dal quale passeranno i suoni.

Il primo swtich gestisce e distribuisce le diverse variazioni di tempo lungo la griglia. Ad ogni gruppo di possibili valori della somma di x e y viene attribuita un diversa variazione di t. Ossia una riduzione (t/2) o un aumento (2*t) dei millisecondi che andranno nel Thread.sleep a scandire il tempo di esecuzione. La scelta dei tempi (t, t/2, t/3, t/4, 2*t, ...) e della loro posizione sulla griglia è dettata in maniera arbitraria per semplicità, ma può essere implementata con un differente automa cellulare che ne stabilisce l'evoluzione; tuttavia la coordinazione dei due automi, specialmente in caso di differenti evoluzioni, può far sorgere diverse problematiche. Il secondo switch con la medesima logica crea, in maniera arbitraria, delle variazioni alla velocity delle note; lo scopo è quello di aggiungere maggiore dinamica all'esecuzione.

In fine troviamo la chiusura del canale e del metodo stesso.

mode(): Il metodo mode() è il regolatore delle note che saranno date in "pasto" al metodo play(). Detta in che modo i "modi" armonici verranno applicati al gioco. Viene stabilita una griglia immaginaria sovrapposta a quella esistente in modo da evitare quei valori considerati fuori scala.

Inizialmente vengono istanziati cinque valori interi rappresentanti i valori della x in corrispondenza dei quali saranno situate le note da correggere. La logica applicata a questo schema è il seguente: se la matrice sulla quale si svolge il gioco viene creata di dimensione 12x12, allora ogni riga della matrice conterrà tutti e 12 i valori delle note musicali, in questo modo si verrà a costituire un ottava completa su ogni riga e tutte le note appartenenti allo stesso tipo saranno in colonna. Quindi

assegnando alle x i valori che corrisponderanno alla colonna sottostante, possiamo scegliere di escludere una determinata fascia di toni, e di riassegnare a quelle note una posizione che sia nel modo scelto. In questo modo, senza stravolgere completamente l'evoluzione e le dinamiche del gioco della vita, possiamo percepire una maggiore armonia nell'esecuzione.

Il metodo sostanzialmente contiene una serie di "if" ed "else if" che stabiliscono: se la cella è vera, ossia se è attiva sulla matrice booleana cell[][], e se si trova su una delle colonne in corrispondenza dei precedentemente detti valori di x, allora viene ricalcolata in modo da essere posizionata, ad una quinta sopra ma entro i valori della scala di riferimento; altrimenti, se in posizione "corretta" viene calcolata nella maniera standard (il valore delle ascisse per il numero massimo di righe sommato al valore delle ordinate). Un secondo schema "if clause" è applicato a tutti i conteggi delle note in modo da stabilire che: se la nota è eccessivamente grave verrà rispedita a delle ottave superiori e se troppo acuta a ottave inferiori.

Il calcolo delle note e la lettura della matrice in ordine verticale eseguono un arpeggio che non è, costantemente, un crescendo di toni ma un alternarsi di alti e bassi.

closeSyn(): E' il metodo che chiude il sintetizzatore. Per evitare che un synth rimasto precedentemente aperto si accavalli al successivo. La mancanza di un semplice metodo come questo aveva portato a grossi problemi nell'esecuzione, in quanto con l'avanzare e l'accavallarsi dei cicli il programma, oltre ad appesantire enormemente il calcolatore, sembra deviare la riproduzione dei valori imposti dall'automa, tendendo ad un arresto anticipato dell'esecuzione.

3.4 Illustrazione del software

L'Applet come precedentemente riassunto è composto da pochi semplici elementi:

□ Una matrice grafica di grandezza 12x12:

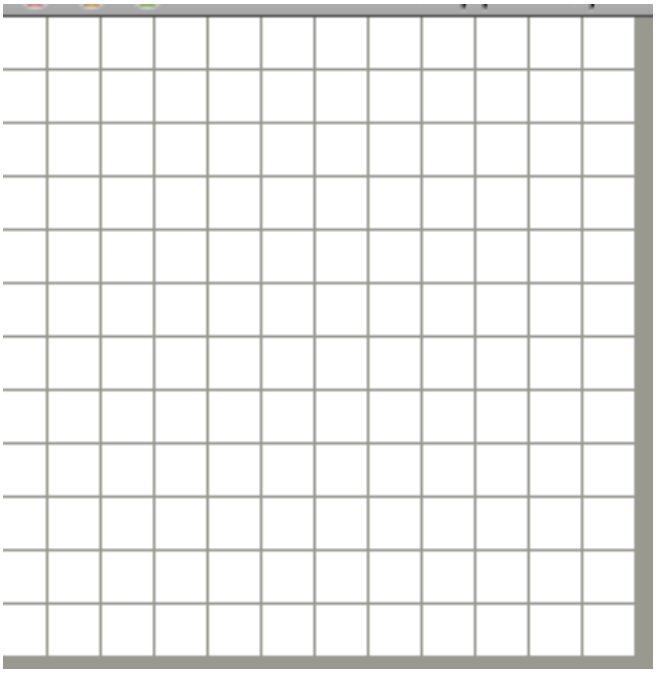


Fig.29: Matrice 12x12

□ Tre tasti di azione principale:

Fig.30: Tasti d'azione play/stop/clear.

□ Un riquadro per l'inserimento del valore del tempo espresso in millisecondi:

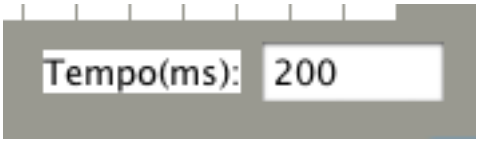


Fig.31: Tempo.

□ Un menù a scelta per selezionare il modo da applicare all'esecuzione:

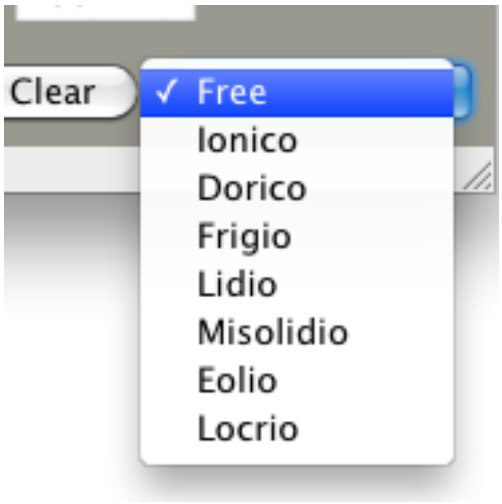


Fig.32: Scelta dei modi.

Visualizzazione della schermata principale: .

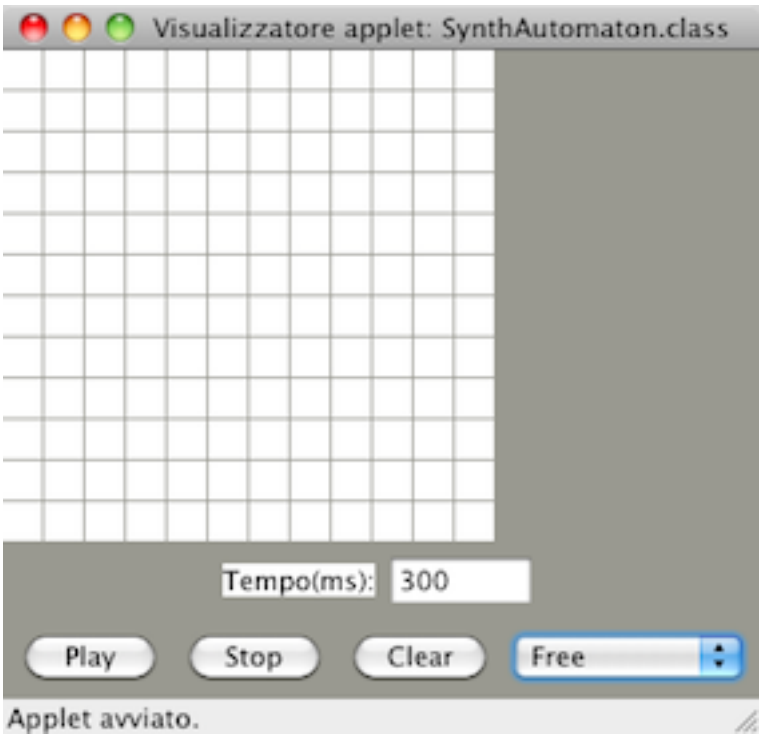


Fig.33: Schermata principale.

Esempio di un particolare loop di esecuzione: .

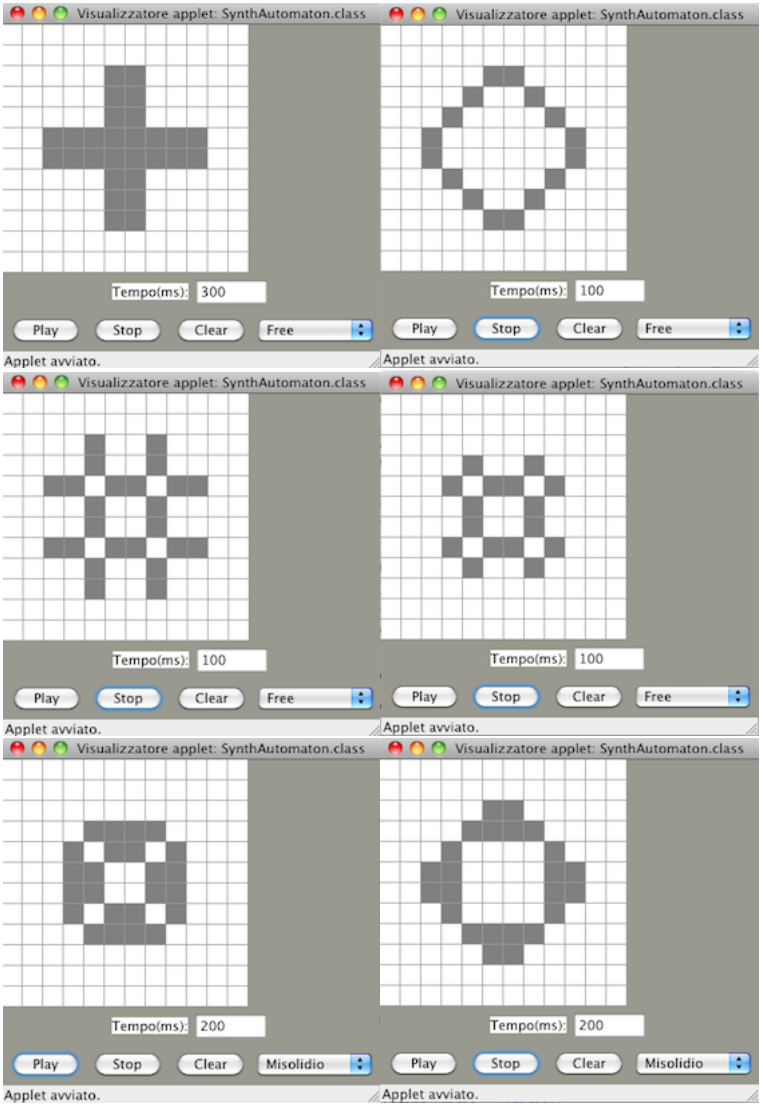


Fig.34,35,36,37,38,39: Le immagini mostrano un particolare oscillatore di periodo 5 che partendo da una figura iniziale a croce evolve in una sequenza costante (loop) dalla quale viene poi escluso il pattern di partenza.

Esempio di arpeggio costante: .

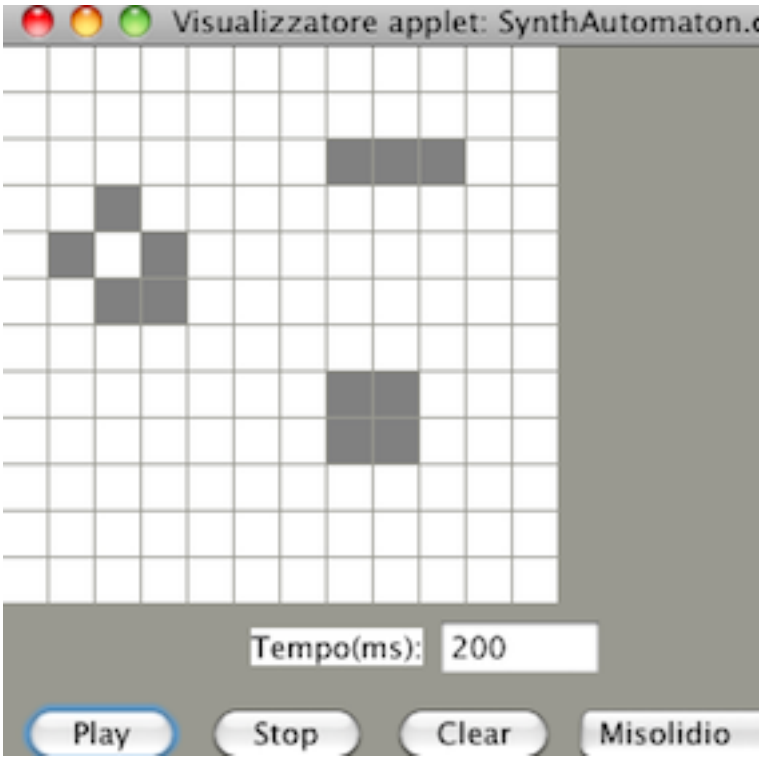


Fig.40: L'immagine mostra in che modo è possibile utilizzare le figure statiche del gioco della vita per realizzare dei loop.

Conclusioni

Il mondo degli automi cellulari esiste ormai da tempo, ma le sue applicazioni sono tutt'oggi limitate a sperimentazioni o simulazioni di ambienti macro o micro-cellulari. Tuttavia gli automi cellulari hanno dimostrato di essere un ottimo mezzo e sopratutto hanno dimostrato di avere una particolare logica che trova riscontro nella maggior parte dei fenomeni naturali. Includendo non solo quei fenomeni attribuiti agli organismi così detti viventi, ma anche l'evoluzione e la distribuzione di tutti i materiali dell'universo e quindi l'organizzazione dell'intero cosmo. Questo dovrebbe farci capire come la logica degli automi cellulari sia adattabile e quindi applicabile alla risoluzione di problemi e quesiti di vario genere. Con maggior riguardo l'utilizzo potrebbe essere esteso all'obiettivo di conoscere la natura umana, e sopratutto la natura del suo pensiero o di ciò che viene definito coscienza. Le reti di centri neurali (CNN) hanno dimostrato che molto probabilmente ciò che definiamo intelligenza attiva, artistica, o le nostre capacità creative derivano da processi logici simili a quelli degli automi cellulari, e che è quindi possibile riprodurre e sintetizzare tali meccanismi. Ciò che si propone questa tesi è proprio indirizzare la nostra curiosità verso la possibilità di creare meccanismi in grado di generare sequenze logiche interpretabili come pattern creativi. O meglio focalizzare l'attenzione, nell'ambito musicale, sulla possibilità di creare meccanismi in grado di generare, secondo gusti e modi definiti da schemi, sequenze melodiche e ritmiche riconducibili ad un' improvvisazione. Ossia quel momento musicale in cui gli artisti danno libero sfogo alla loro creatività mantenendo una certa congruenza logico-armonica nell'esecuzione. Anche una semplice implementazione come questa ha mostrato grandi capacità dal punto di vista creativo, e ha quindi lasciato larghi spiragli su vedute di questo genere. Intere parti musicali potrebbero essere prodotte da meccanismi simili, ognuno organizzato per gestire specifici strumenti riprodotti secondo diversi generi musicali. Si potrebbero creare software in grado di generare intere tracce o si potrebbero usare gli automi come stimolo per nuove direzioni alla creatività degli artisti. In conclusione si potrebbe affermare che una profonda conoscenza degli automi cellulari, e quindi di tutti i meccanismi necessari per implementarli, potrebbe condurci ad una più precisa definizione di ciò che è la natura umana, e ad una più netta distinzione di ciò che è in noi meccanico e automatico anziché puramente "umano".

Codice Java

Questo è il codice java del software realizzato:

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * SYNTHAUTOMATON - 2011/2012
 * REALIZZATO DA: ARMANDO VILIAM FORTUNATO   Mtr.: 0000217020
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.Canvas;
import java.awt.Choice;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Event;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.Panel;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Vector;
import javax.sound.midi.MidiChannel;
import javax.sound.midi.MidiSystem;
import javax.sound.midi.MidiUnavailableException;
import javax.sound.midi.Synthesizer;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.JTextField;

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * SynthAutomaton è un arpeggiatore per MIDI Synthesis in Java, consiste in una griglia dove
 * vengono inserite le celle rappresentati le note, che rielaborate da un Automa cellulare
 * evolvono in nuove strutture in base a delle regole prestabilite.
 * Queste celle vengono interpretate sotto forma di numero di tasto MIDI per il sintetizzatore,
 * che in base a una latenza assegnata ad ogni cella, suona (in questo caso con il suono di un
 * pianoforte) le celle riportate attive nella griglia, correggendo, a seconda della presenza o
 * meno di un "modo", le note fuori scala.
 * La latenza tra le note, ovvero il tempo dell'esecuzione, cambia a seconda della posizione
 * nella griglia della cella; e così per la durata di ogni nota, ossia la velocity.
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
public class SynthAutomaton extends Applet implements Runnable {

        // Istanzio tutte le varibili, Thread per l'esecuzione, String per i label
        // dei tasti
        // nella parte grafica, JButton e Jtext, e degli array di int contenenti le
        // stringhe
        // che verranno poi date in pasto al metodo mode() per la scelta del modo di
        // esecuzione;
        // in fine l'int t; rappresenta il tempo in millisecondi preso in input dal
        // JTextField tmp.
        private static final long serialVersionUID = 1L;
        private CA ca;
        private Thread Thread = null;
        private final String playl = "Play";
        private final String stopl = "Stop";
```

```
private final String clearl = "Clear";
private final String Free = "Free";
private final String Ionico = "Ionico";
private final String Dorico = "Dorico";
private final String Frigio = "Frigio";
private final String Lidio = "Lidio";
private final String Misolidio = "Misolidio";
private final String Eolio = "Eolio";
private final String Locrio = "Locrio";
final JButton play = new JButton(playl);
final JButton stop = new JButton(stopl);
final JButton clear = new JButton(clearl);
int[] modFree = new int[] { 12, 12, 12, 12, 12 };
int[] modIonico = new int[] { 1, 3, 6, 8, 10 };
int[] modDorico = new int[] { 1, 4, 6, 8, 11 };
int[] modFrigio = new int[] { 2, 4, 5, 7, 10 };
int[] modLidio = new int[] { 1, 3, 5, 8, 10 };
int[] modMisolidio = new int[] { 1, 3, 6, 8, 11 };
int[] modEolio = new int[] { 1, 4, 6, 9, 11 };
int[] modLocrio = new int[] { 2, 4, 7, 9, 11 };
int[] app = new int[] { 12, 12, 12, 12, 12 };
final JTextField tmp = new JTextField("300", 5);
final JTextArea tm = new JTextArea("Tempo(ms):");
int t;

// Il metodo init() inizializza l'Applet grafico, vengono posizionati due
// pannelli controls
// dove vengono aggiunti i pulsanti e quindi i comandi del progamma, e un
// oggetto di tipo CA
// rappresentante la griglia contenente le celle.
@SuppressWarnings("deprecation")
public void init() {
        int COL = 12;
        int RIG = 12;
        int CS = 20;

        setBackground(new Color(0x999990));

        final JPanel centerPanel = new JPanel();

        ca = new CA(CS, COL, RIG);

        Panel controls = new Panel();
        Panel controls1 = new Panel();
        controls.add("South", play);
        controls.add("South", stop);
        controls.add("South", clear);
        controls1.add("North", tm);
        controls1.add("North", tmp);
        centerPanel.add(ca);

        // Aggiungo ai bottoni play, stop e clear, un ActionListener, per
        // attribuire a questi
```

```
        // il comando di start() e stop() dell'esecuzione.
        play.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                        start2();
                }
        });
        stop.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                        stop();
                }
        });
        clear.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                        ca.clear();
                        ca.repaint();
                }
        });

        Choice ch = new Choice();
        ch.addItem(Free);
        ch.addItem(Ionico);
        ch.addItem(Dorico);
        ch.addItem(Frigio);
        ch.addItem(Lidio);
        ch.addItem(Misolidio);
        ch.addItem(Eolio);
        ch.addItem(Locrio);
        controls.add(ch);

        setLayout(new BorderLayout());
        add("North", ca);
        add(controls1);
        add("South", controls);

        show();
        resize(preferredSize());
        validate();

        setVisible(true);

}

// Gestisce l'avvio del Thread
public void start2() {
        if (Thread == null) {
                Thread = new Thread(this);
                Thread.start();
        }
}

// Gestisce lo stop del thread
@SuppressWarnings("deprecation")
public void stop() {
```

```
        if (Thread != null) {
                Thread.stop();
                Thread = null;
        }

}

// Metodo run() del thread, stabilisce l'ordine principale d'esecuzione,
// viene
// attivato e disattivato dai metodi start() e stop();
public synchronized void run() {
        Event evt = null;
        Object arg = null;
        action(evt, arg);

        while (Thread != null) {
                ca.fill();
                ca.check();
                ca.rule();

                try {
                        t = Integer.valueOf(tmp.getText());
                        ca.play(app, t);
                        ca.repaint();
                        ca.closeSyn();
                } catch (MidiUnavailableException e1) {
                        e1.printStackTrace();
                }
        }
}

// Aggiunge le funzioni ai restanti tasti dell'applet, come la scelta dei
// modi.
public boolean action(Event evt, Object arg) {
        if (Ionico.equals(arg)) {
                app = modIonico;
                return true;
        } else if (Dorico.equals(arg)) {
                app = modDorico;
                return true;
        } else if (Frigio.equals(arg)) {
                app = modFrigio;
                return true;
        } else if (Lidio.equals(arg)) {
                app = modLidio;
                return true;
        } else if (Misolidio.equals(arg)) {
                app = modMisolidio;
                return true;
        } else if (Eolio.equals(arg)) {
                app = modEolio;
                return true;
        } else if (Locrio.equals(arg)) {
```

```
                app = modLocrio;
                return true;
            } else if (Free.equals(arg)) {
                app = modFree;
                return true;
            }
            return false;
        }
}

// Questa classe rappresenta il cuore del progetto, viene instanziata la Grafica
// per la
// griglia e le celle, viene esplicato il funzionamento di posizionamento delle
// celle con
// il mouse, e contiene i metodi che attivano l'Automa Cellulare e il
// Sintetizzatore.

class CA extends Canvas {
        /**
         *
         */
        private static final long serialVersionUID = 1L;
        private int CS; // Grandezza delle celle
        private int RIG; // numero di righe
        private int COL; // numero di colonne
        private boolean cellUnderMouse;
        private boolean cell[][]; // matrice di valori booleani, la griglia del
                                                                // Gioco dell Vita
        private int Buffer[][]; // Buffer del CA
        private Image offScreenImage = null;
        private Graphics offScreenGraphics;
        Synthesizer synthesizer; // sintetizzatore
        public Vector<Integer> notes = new Vector<Integer>(); // vettore contenente

                        // le note
        int note;

        // costruttore della classe CA
        @SuppressWarnings("deprecation")
        public CA(int CS, int COL, int RIG) {
                cell = new boolean[COL][RIG];
                Buffer = new int[COL][RIG];
                this.CS = CS;
                this.COL = COL;
                this.RIG = RIG;
                reshape(0, 0, CS * COL - 1, CS * RIG - 1);
                clear();
        }

        // Metodi per gestire il posizonamento delle celle sulla griglia cell[][]
        public synchronized boolean mouseUp(java.awt.Event evt, int x, int y) {
                try {
                        cell[x / CS][y / CS] = !cellUnderMouse;
```

```
        } catch (java.lang.ArrayIndexOutOfBoundsException e) {
        }
        repaint();
        return true;
}

public synchronized boolean mouseDown(java.awt.Event evt, int x, int y) {
        try {
                cellUnderMouse = cell[x / CS][y / CS];
        } catch (java.lang.ArrayIndexOutOfBoundsException e) {
        }
        return true;
}

public synchronized boolean mouseDrag(java.awt.Event evt, int x, int y) {
        try {
                cell[x / CS][y / CS] = !cellUnderMouse;
        } catch (java.lang.ArrayIndexOutOfBoundsException e) {
        }
        repaint();
        return true;
}

// Metodo per l'aggiornamento della matrice grafica
public synchronized void update(Graphics theG) {
        @SuppressWarnings("deprecation")
        Dimension d = size();
        if ((offScreenImage == null)) {
                offScreenImage = createImage(d.width, d.height);
                offScreenGraphics = offScreenImage.getGraphics();
        }
        paint(offScreenGraphics);
        theG.drawImage(offScreenImage, 0, 0, null);
}

// Disegna la griglia
public void paint(Graphics g) {
        // traccia il fondo
        g.setColor(Color.white);
        g.fillRect(0, 0, CS * COL - 1, CS * RIG - 1);
        // traccia la griglia
        g.setColor(getBackground());
        for (int x = 1; x < COL; x++) {
                g.drawLine(x * CS - 1, 0, x * CS - 1, CS * RIG - 1);
        }
        for (int y = 1; y < RIG; y++) {
                g.drawLine(0, y * CS - 1, CS * COL - 1, y * CS - 1);
        }
        // Disegna le celle che si sono popolate
        g.setColor(Color.gray);
        for (int y = 0; y < RIG; y++) {
                for (int x = 0; x < COL; x++) {
                        if (cell[x][y]) {
```

```
                                g.fillRect(x * CS, y * CS, CS - 1, CS - 1);
                        }
                }
        }
}

// Ripulisce la matrice
public synchronized void clear() {

        for (int x = 0; x < COL; x++) {
                for (int y = 0; y < RIG; y++) {
                        cell[x][y] = false;
                }
        }
}

// Viene inizializzato un buffer con i valori vicini ad ogni cella
public synchronized void fill() {
        int x, y;
        // svuota il buffer
        for (x = 0; x < COL; x++) {
                for (y = 0; y < RIG; y++) {
                        Buffer[x][y] = 0;
                }
        }
        // Aggiunge nel Buffer tutti i punti dell'intorno della cella in
        // posizione
        // cell[x][y], considera un intorno di tipo Moore Neighbourhood
        for (x = 1; x < COL - 1; x++) {
                for (y = 1; y < RIG - 1; y++) {
                        if (cell[x][y]) {
                                Buffer[x - 1][y - 1]++;
                                Buffer[x][y - 1]++;
                                Buffer[x + 1][y - 1]++;
                                Buffer[x - 1][y]++;
                                Buffer[x + 1][y]++;
                                Buffer[x - 1][y + 1]++;
                                Buffer[x][y + 1]++;
                                Buffer[x + 1][y + 1]++;
                        }
                }
        }
}

public synchronized void check() {

        // Controlla che le celle true vicine alla casella nn siano bordi o ai
        // bordi
        int x = 1;
        int y = 0;
        int dx = 1;
        int dy = 0;
        while (true) {
```

```
                if (cell[x][y]) {
                        if (x > 0) {
                                if (y > 0)
                                        Buffer[x - 1][y - 1]++;
                                if (y < RIG - 1)
                                        Buffer[x - 1][y + 1]++;
                                Buffer[x - 1][y]++;
                        }
                        if (x < COL - 1) {
                                if (y < RIG - 1)
                                        Buffer[x + 1][y + 1]++;
                                if (y > 0)
                                        Buffer[x + 1][y - 1]++;
                                Buffer[x + 1][y]++;
                        }
                        if (y > 0)
                                Buffer[x][y - 1]++;
                        if (y < RIG - 1)
                                Buffer[x][y + 1]++;
                }

                // controlla in caso di cella al bordo, di che bordo si tatta e
                // corregge
                // la traiettoria facendole tornare indietro ai bordi
                if (x == COL - 1 && y == 0) {
                        dx = 0;
                        dy = 1;
                } else if (x == COL - 1 && y == RIG - 1) {
                        dx = -1;
                        dy = 0;
                } else if (x == 0 && y == RIG - 1) {
                        dx = 0;
                        dy = -1;
                } else if (x == 0 && y == 0) {
                        break;
                }
                x += dx;
                y += dy;
        }

}

// E' il vero Cellular Automata, rappresenta le regole del Gioco della vita.
// ogni cella morta con tre vicini vivi nasce, ogni cella viva con 2-3
// vicini vivi
// resta viva.
public synchronized void rule() {
        int x, y;
        // Se il cellsBuffer contiene 2 elementi non avviene nulla, se ne
        // contiene 3 la cells in quelle coordinate diventa true, altrimenti
        // diviene false.
        for (x = 0; x < COL; x++) {
                for (y = 0; y < RIG; y++) {
```

```
                        switch (Buffer[x][y]) {
                        case 2:
                                break;
                        case 3:
                                cell[x][y] = true;
                                break;
                        default:
                                cell[x][y] = false;
                                break;
                        }
                }
        }
}

// inizializza il sintetizatore, scorre la matrice e in base al mode assegna
// le note
// che verranno eseguite
public synchronized void play(int[] mod, int t)
                throws MidiUnavailableException {

        int x, y;

        synthesizer = MidiSystem.getSynthesizer();
        synthesizer.open(); // apro il synth
        MidiChannel channel = synthesizer.getChannels()[0];

        // scorro la matrice
        for (x = 0; x < RIG; x++) {
                for (y = 0; y < COL; y++) {
                        if (cell[x][y]) {

                                mode(x, y, mod);// metodo mode

                                // creo il vettore notes che verrà riempito di tutti i
                                // valori int delle note
                                notes = new Vector<Integer>();
                                notes.add(note);

                                // per ongi note del vettore notes : ...
                                for (int note : notes) {
                                        try {

                                        // questo swtich gestisce e distribuisce le diverse
                                        // variazioni di
                                        // tempo lungo la griglia. Ad ogni gruppo di
                                        // possibili valori
                                        // della somma di x e y viene attribuita un diversa
                                        // variazione di
                                        // t. Ossia una riduzione (t/2) o un aumento (2*t)
                                        // dei millisecondi
                                        // che andranno nel Thread.sleep a scandire
                                        // il tempo di esecuzione.
                                                int t1;
```

```
        switch (x + y) {
        case 0:
        case 1:
        case 2:
        case 3:
                t1 = t;
                break;
        case 4:
        case 5:
        case 6:
        case 7:
                t1 = t;
                break;
        case 8:
        case 9:
        case 10:
        case 11:
                t1 = 2 * t;
                break;
        case 12:
        case 13:
        case 14:
        case 15:
                t1 = t * (3 / 2);
                break;
        case 16:
        case 17:
        case 18:
        case 19:
                t1 = t;
                break;
        case 20:
        case 21:
        case 22:
                t1 = t / 2;
                break;
        default:
                t1 = t;
                break;
        }

// Con la medesima logica del tempo, creiamo delle
// variazioni, in
// modo arbitrario, nella velocity delle note. In
// modo da aumentare
// la dinamica della riproduzione musicale.
        int v = 50;
        int v1;
        switch (x + y) {
        case 0:
        case 1:
        case 2:
        case 3:
```

```
                                        v1 = v;
                                        break;
                                case 4:
                                case 5:
                                case 6:
                                case 7:
                                        v1 = v * (4 / 3);
                                        break;
                                case 8:
                                case 9:
                                case 10:
                                case 11:
                                        v1 = 2 * v;
                                        break;
                                case 12:
                                case 13:
                                case 14:
                                case 15:
                                        v1 = v * (3 / 2);
                                        break;
                                case 16:
                                case 17:
                                case 18:
                                case 19:
                                        v1 = v;
                                        break;
                                case 20:
                                case 21:
                                case 22:
                                        v1 = v * (5 / 3);
                                        break;
                                default:
                                        v1 = v;
                                        break;
                                }
                                // La nota viene passata al cnale
                                channel.noteOn(note, v1);

                                Thread.sleep(t1);
                        } catch (InterruptedException e) {
                        } finally {
                                channel.allNotesOff();
                        }
                }

        }
    }
}

// Il metodo mode stabilisce una griglia immaginaria sulla matrice
// esistente, in modo
// da "evitare" quelle colonne sulle quali sono posizionate note fuori dalla
```

```
	// da "evitare" quelle colonne sulle quali sono posizionate note fuori dalla
	// scala.
	public synchronized void mode(int x, int y, int[] mod) {

		// gli interi a,b,c,d,e rappresentano i valori della x presso i quali
		// sono situati i valori da risistemare.
		int a, b, c, d, e;

		a = mod[0];
		b = mod[1];
		c = mod[2];
		d = mod[3];
		e = mod[4];

		// se la cella è vera, ossia se è attiva sulla matrice booleana
		// cell[][],
		// e si trova su una delle colonne in corrispondenza dei suddetti valori
		// di x,
		// allora viene ricalcolata in modo da essere posizionata, ad una quinta
		// in sopra
		// ma entro i valori della scala di riferimento
		if (cell[x][y]) {
			if (x == a | x == b | x == c | x == d | x == e) {
				note = (((y) * RIG) + (x)) + 6;
				// se la nota è troppo alta o troppo basse viene spostata a
				// delle ottave
				// inferiori o superiori. In questo modo si evitano toni
				// ecessivamente e inutilmente
				// acuti o gravi.
				if (note < 24)
					note += 12;
				else if (note > 96)
					note -= 48;
			}
			// note = (((x)*RIG)+(COL-y))+1;

			// Qui il calcolo della nota è calcolato senza alcuna
			// aggiunta(correzione), se
			// il valore di x è diverso da quelli espressi.
			else if (x != a | x != b | x != c | x != d | x != e) {
				note = (((y) * RIG) + (x));
				if (note < 24)
					note += 12;
				else if (note > 96)
					note -= 48;
			}
		}
	}

	// metodo che chiude il sintetizzatore. Per evitare che un synth rimasto
	// aperto
	// precedentemente si accavalli al successivo.
	public void closeSyn() {
		synthesizer.close();
	}

}
```

Ringraziamenti:

Ringrazio tutta la mia famiglia, in particolar modo i miei genitori, le uniche persone che mi hanno sempre sostenuto.

Bibliografia

[LAN84] Christopher G. Langton, Self-Reproduction in Cellular Automata, Physica 10D, Amsterdam, 1984.

[TOFMAR87] T. Toffoli N. Margolus, Cellular Automata Machines: A New Enviroment for Modeling, Massachusetts Institute of Technlogy, US, 1987.

[WOL83] Stephen Wolfram, Theory and Apllication of Cellular Automata, World Scientific, 1983.

[WOL02] Stephen Wolfram, A New Kind of Science, Wolfram Media, University of Michigan, 2002.

[BCG04] Berlekamp E. R., Conway John Horton, Guy R.K., Winning Ways for your Mathematical Plays (2nd ed.), A K Peters Ltd, 2001-2004.

[RUS01] L. Russo, La Rivoluzione Dimenticata, il Pensiero Scientifico Greco e la Scienza Moderna, Feltrinelli, 2001.

[SCH08] Alexander Schatten, A new Science named Complexity, 2008, http://www.schatten.info/info/ca.html, 24 Febbraio 2012.

[LUC03] Chris Lucas, Automata: Agent of Life Within, http://www.calresco.org/automata.htm, 20 Febbraio 2012.

[TUR36] Alan Turing, On Computable Numbers, with an application to the Entscheidungsproblem, Proc. London Math. Soc., 1936.

[MASH11] Genaro J. Martìnez, Andrew Adamatzky, Christopher R. Stephens, Alejandro F. Hoeflich, Cellular automaton supercolliders, Bristol BS16 1QY, United Kingdom., 24 Maggio 2011.