ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI SCIENZE

CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

Evoluzione Artificiale di Automi Cellulari

Relazione finale in
Programmazione ad oggetti

Relatore
Andrea Roli

Correlatore
Alessandro Ricci

Presentato da
Francesco Del Duchetto

Sessione I

Anno accademico 2014 - 2015

"Evolution brings human beings.
Human beings, through a long
and painful process, bring humanity."
— Dan Simmons, Hyperion

"C'era un signore che stava lì tranquillo
Leggeva le istruzioni con la luce accesa
La natura di soppiatto
Senza fare alcun rumore
Arrivò all'interruttore
Fine."
— Dente, Lo scherzo della natura

Indice

Introduzione

L'uomo da sempre ha trovato nella natura un ottimo modello d'ispirazione per le proprie creazioni. Ciò che osserviamo in essa è infatti quello che 4.6 miliardi di anni di prove, fallimenti e miglioramenti hanno trasformato le cellule primordiali negli esseri viventi esistenti oggi sul nostro pianeta. Chiaramente nel progettare qualcosa è più facile copiare la struttura o il comportamento di un modello simile già funzionante, perché perfezionata nel tempo dall'evoluzione, che inventarne di nuovi.

I primi esempi, di cui l'uomo ha memoria, dell'utilizzo di tale approccio risalgono alla fine del XV secolo quando Leonardo da Vinci osservava l'anatomia ed i movimenti degli uccelli per trovare il modo di far volare gli esseri umani. I primi che riuscirono in questa impresa, circa 400 anni dopo, furono i fratelli Wright i quali costruirono un velivolo prendendo ispirazione dal volo dei piccioni. Innumerevoli sono state nell'arco della nostra storia le "invenzioni" prese in prestito dalla natura, giusto per citarne alcune: il velcro imita i piccoli uncini presenti in alcune piante, il meccanismo di visualizzazione dei colori nei nuovi schermi imita la colorazione strutturale presente nelle ali delle farfalle, i sistemi ad ultrasuoni imitano il sistema di localizzazione dei pipistrelli, e così via. Altri esempi possono essere trovati sul sito AskNature.org [1], una libreria di oltre 1800 fenomeni naturali che possono essere, o che sono già stati, modello d'ispirazione per progetti creati dall'uomo.

Il meccanismo che sta alla base di ogni processo evolutivo nella natura e che ha portato allo sviluppo di sistemi talmente sofisticati è la *selezione naturale*. Un processo che, tramite piccole e graduali modifiche, riesce nel tempo ad introdurre adattamento negli individui di una popolazione. Anche questo meccanismo è stato copiato dagli esseri umani per i propri scopi, questa volta però non come un "manuale" da seguire per progettare qualcosa, ma come "generatore" di possibili soluzioni per uno specifico problema.

Una classe di tali metodi che imitano l'evoluzione naturale, gli **algoritmi genetici**, verranno illustrati nel capitolo 3, nel quale vedremo anche come questi algorit-

mi risultino efficaci nella risoluzione di alcuni problemi difficilmente approcciabili da un essere umano.

La mente umana spesso fallisce nel risolvere, ed a volte addirittura nel capire la natura di problemi che implicano il dover considerare un vasto insieme di entità che lavorano collaborativamente per svolgere uno compito. Nel loro insieme questi gruppi di entità vengono chiamati *sistemi complessi* perché appunto sono molto complessi da studiare. La maggior parte di questi sistemi, inoltre, nel tempo mostra delle modifiche al proprio comportamento ed alla propria struttura in risposta ai cambiamenti esterni. Tali modifiche sono talmente coerenti che in passato le persone hanno ritenuto che l'esistenza di un'entità intelligente che li gestisse fosse l'unica spiegazione possibile della loro manifestazione.

I sistemi complessi che mostrano adattamento all'ambiente esterno vengono chiamati **sistemi adattativi complessi** e di questi parleremo nel capitolo 1.

Il grande interesse nello studio dei sistemi complessi ha portato alla nascita di un modello matematico che permette di studiare tali sistemi astraendoli dal loro contesto. Tale modello è conosciuto con il nome di **automa cellulare** e nel capitolo 2 vedremo come grazie ad esso riusciamo a simulare diversi sistemi.

Nel capitolo 4 cercheremo infine di mettere insieme le cose viste precedentemente per mostrare come si possa generare adattamento negli automi cellulari utilizzando gli algoritmi genetici.

Gli automi cellulari possono sì modellare un gran numero di sistemi, però spesso non è immediato capire la corrispondenza che li lega al sistema modellato. Un modo immediato per vedere i risultati dell'evoluzione è quello di considerare gli automi cellulari come delle piccole immagini binarie. Il capitolo 5 mostra le prime prove sperimentali effettuate e i risultati ottenuti nell'applicare questa metodologia.

In fine, nel capitolo 6, costruiamo sui risultati ottenuti precedentemente un sistema che permette di evolvere una qualunque immagine binaria, utilizzando un sistema di automi cellulari. Vedremo come questi si organizzano e si sincronizzano tra di loro, in modo sorprendentemente simile ad una popolazione di organismi viventi, per raggiungere il loro obiettivo.

1 Sistemi adattativi complessi

Quello dei sistemi complessi è un campo relativamente nuovo che iniziò nella metà del 1980 al Sante Fè Institute nel Nuovo Messico.

Con sistema complesso si intende un qualunque sistema che presenta una grande quantità di componenti che interagiscono tra di loro e la quale attività complessiva risultante è non-lineare, cioè non derivabile dalla somma delle attività delle singole componenti. Tipicamente le componenti si auto-organizzano sotto la spinta di una selezione, naturale o artificiale, producendo un comportamento globale molto complesso e spesso inaspettato.

Facciamo un esempio considerando il sistema immunitario. Il sistema immunitario umano è una comunità formata da un vasto numero di unità chiamate *anticorpi* che continuamente respingono o distruggono un insieme sempre mutevole di invasori chiamati *antigeni*. Gli invasori – principalmente sostanze biochimiche, batteri e virus – arrivano in un infinità di varietà distinte. A causa di questa varietà, e siccome questi invasori arrivano continuamente, il sistema immunitario non può semplicemente elencare tutti i possibili antigeni da respingere. Esso deve cambiare o adattare i suoi anticorpi all'arrivo dei nuovi invasori, senza mai stabilirsi in una configurazione fissata.

Nonostante la sua mutevolezza, il sistema immunitario mantiene un'impressionante coerenza. Potremmo infatti derivare da esso una definizione scientifica della nostra identità. È così efficace nel distinguere noi dal resto del mondo al punto da rigettare cellule provenienti da altri esseri umani.

Come il sistema immunitario esistono svariati esempi di sistemi che mostrano le caratteristiche simili:

- gli ecosistemi
- il sistema nervoso
- le relazioni sociali
- i sistemi economici
- gli stormi di uccelli

Anche se questi sistemi differiscono nei dettagli notiamo che ognuno di essi mostra coerenza di fronte al cambiamento. Possiamo cioè dire che si adattano alle nuove situazioni che possono verificarsi. Da qui il nome Sistema *Adattativo* Complesso (SAC).

Osservandoli attentamente possiamo estrarre diverse caratteristiche o proprietà comuni. John H. Holland, pioniere nel campo dei SAC e considerato il padre degli algoritmi genetici, nel suo libro "Hidden order: how adaptation build complexity" [6] elenca quattro principali proprietà e tre meccanismi che sono comuni a tutti i SAC. Di seguito riporteremo le suddette proprietà e meccanismi, che a mio avviso sono molto interessanti e determinanti per capire come tali sistemi funzionino.

1.1 Aggregazione - *proprietà*

L'aggregazione rientra nello studio dei SAC in due sensi. Il primo significato si riferisce al modo con cui siamo soliti semplificare cose complicate. Aggreghiamo cose simili in categorie – alberi, auto, edifici – per poterli trattare come equivalenti. Nel nostro caso è utile per creare gruppi, decidendo quali dettagli degli elementi possiamo trascurare e quali invece caratterizzano gli elementi del nostro aggregato. Gli anticorpi del nostro sistema immunitario non sono tutti uguali, però trascurando alcuni dettagli possiamo raggrupparli tutti nello stesso insieme che è il sistema immunitario.

Il secondo senso di aggregazione si riferisce a cosa un SAC *può fare* più che a *cosa è*. Riguarda quindi il comportamento che emerge dalle interazioni tra i componenti dell'aggregato. Il comportamento emergente nei sistemi complessi è solitamente una manifestazione imprevedibile che non può essere dedotta esaminando il comportamento delle singole componenti. Pensiamo all'identità fornita dagli anticorpi del sistema immunitario, o all'intelligenza generata da miliardi di neuroni interconnessi nel cervello. Queste proprietà aggregate emergenti appartengono al sistema considerato come entità unica, e non alle componenti che invece svolgono azioni semplici ed apparentemente insignificanti.

1.2 Etichettatura - *meccanismo*

È un meccanismo che facilita la formazione di aggregati, consentendo agli elementi del sistema di effettuare una interazione selettiva. Ogni componente può avere un'etichetta che mostra agli altri alcune, o tutte, le sue caratteristiche permettendo loro di ottenere più informazioni su di se. A fronte della conoscenza di tale informazione i vicini possono effettuare delle scelte – come interagire con lui oppure no. Grazie a questo meccanismo le componenti possono raggrupparsi e formare gerarchie migliorando la cooperazione.

1.3 Non linearità - *proprietà*

I sistemi lineari godono della proprietà secondo la quale l'insieme è uguale alla somma delle parti che lo compongono. Al contrario, i sistemi non lineari sono uguali a più della somma delle loro parti. Invece di essere una semplice sommatoria delle componenti, applica dei fattori moltiplicativi alle interazioni delle componenti. Intuitivamente possiamo pensare che se ad un sistema $S = \{s_1, s_2, \ldots, s_n\}$ composto da n componenti ognuna delle quali interagisce con una percentuale μ delle restanti, dove I è l'insieme delle interazioni totali del sistema, otteniamo che

$$|I| = n \cdot \mu \cdot (n-1)$$

In questa formula osserviamo che non abbiamo linearità perché se considerassimo un unico componente, fissiamo quindi $S' = \{s_k\}$ ed I_k il numero di interazioni di s_k, con $1 \leq k \leq n$, otteniamo

$$|I_k| = 1 \cdot \mu \cdot 0 = 0$$

e banalmente

$$|I| \neq \sum_{i=1}^{n} |I_i| = 0$$

Questa proprietà ci dà una prima intuizione del motivo per cui i sistemi complessi, se aggregati, producono un comportamento inspiegabile e molto più complicato di quello che ci saremmo aspettati osservandone le componenti.

1.4 Flusso - *proprietà*

Per flusso in un SAC si intende un collegamento tra due delle sue componenti. La presenza di un collegamento indica la possibilità di interazione tra le due parti, un collegamento mancante invece indica l'impossibilità di interagire.

Ora possiamo figurarci il nostro sistema come un grafo dove ogni vertice – o componente – è collegata tramite un arco – o flusso – ai vertici con cui può interagire. Il grafo però non è statico nel tempo, ma possono scomparire archi presenti e nascerne di nuovi. Tale operazione avviene grazie al meccanismo di etichettatura il quale permette ad un componente di scegliere con chi interagire, magari prediligendo quelli con etichette "più affidabili" e limitando le interazioni con i meno interessanti.

I flussi, a seconda della loro configurazione, possono avere un effetto moltiplicativo o ciclico. Nel primo caso l'interazione viene amplificata – o attenuata – permettendo l'aumento – o la diminuzione – delle interazioni nella zona interessata. L'effetto ciclico si ha quando nel grafo otteniamo un ciclo e ciò comporta l'accumulo di risorse derivate dalle interazioni.

1.5 Diversità - *proprietà*

Ogni SAC è composto da componenti diversi tra loro. La diversità sta nella varietà delle funzioni che svolgono all'interno dell'aggregato e le funzioni sono determinate dalle interazioni che questo ha con i suoi vicini. Ogni componente insomma occupa un posto specifico nel sistema che è definito dai flussi che esso può creare con i vicini. Se rimuoviamo un tipo di componente, tipicamente il sistema risponde con una serie di adattamenti che ne generano una nuova, la quale andrà a "riempire" lo spazio mancante. Questo processo può essere visto come una convergenza verso un determinato stato del sistema. Ma, come detto precedentemente, il sistema ha bisogno di adattarsi per poter sopravvivere alle interferenze esterne, quindi come può adeguarsi alle nuove esigenze se è spinto ad assumere una determinata configurazione? Grazie al sistema di etichettatura. Infatti quando una nuova componente va ad occupare un posto rimasto scoperto, esso avrà sì un comportamento adeguato alle richieste dei vicini, ma allo stesso tempo può avere degli ulteriori tratti, rimasti "nascosti" grazie alla sua etichetta, che sono diversi dai tratti che aveva il

suo predecessore. Il sistema quindi è in grado di modificarsi ed adattarsi, senza però perdere le sue caratteristiche originarie.

1.6 Modello interno - *meccanismo*

Un modello interno può essere visto come uno schema, che permette di mappare valori presi in input e filtrarli in pattern che il SAC può usare per modificare se stesso. Il sistema è capace di memorizzare questi pattern e li riusa in futuro per anticipare il risultato delle future configurazioni di input rilevate.

Quindi il modello interno permette al sistema sia di decidere cosa fare a fronte di uno stimolo, sia di anticipare gli avvenimenti in base alle esperienze pregresse.

Il nostro cervello, ad esempio, è capace di memorizzare gli avvenimenti durante l'arco della nostra vita, percependoli tramite i sensi, e riutilizzandoli per anticipare il risultato di scene simili a quelle già vissute.

1.7 Elementi costitutivi - *meccanismo*

Se consideriamo gli stimoli che il nostro cervello acquisisce dalla vista ci rendiamo conto che, secondo ciò che abbiamo detto finora, sarebbe impossibile riuscire a riconoscere nella visuale un oggetto che avevamo già incontrato in passato. Infatti è praticamente impossibile che i nostri occhi percepiscano due volte le stesse identiche intensità di luce nell'intera visuale – cioè che si presentino per due volte gli stessi valori di input.

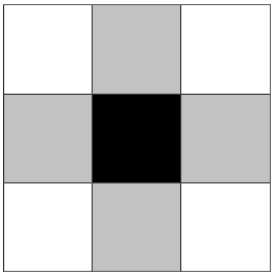
Qui ci viene in aiuto l'abilità di decomporre uno schema in più elementi costitutivi. Questi "pezzi" di schema possono essere riusati e ricombinati in una grande varietà di modi e ci permettono di riconoscere scene familiari considerando solamente le parti che ricordiamo.

Quando guardiamo in faccia una persona che non abbiamo mai incontrato, riusciamo a capire che è un altro uomo basandoci solamente su alcuni dettagli – come il fatto che ha gli occhi, le orecchie e la bocca disposti in un certo modo, che non ha la pelle ricoperta da pelliccia, e così via.

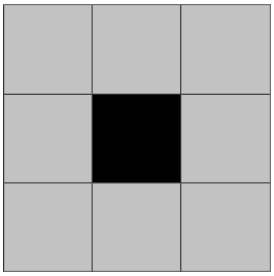
2 Automi cellulari

Gli automi cellulari sono dei modelli matematici discreti studiati in teoria della computabilità, matematica, fisica e biologia per descrivere l'evoluzione nel tempo di sistemi complessi.[19]

Un automa cellulare consiste di una griglia regolare di celle, ognuna delle quali può trovarsi in un numero finito di stati $H = \{h_1, \dots h_n\}$, come ad esempio *on* e *off*. La griglia può essere di un qualunque numero finito di dimensioni. Per ogni cella, si definisce un suo **vicinato** come un insieme di celle appartenenti alla griglia e situati ad una specifica posizione rispetto ad essa. I tipi di vicinato usati più di frequente sono quello di von Neumann e quello di Moore.



(a) Vicinato di von Neumann



(b) Vicinato di Moore

Con il vicinato di *von Neumann* per ogni cella consideriamo vicine le celle situate a nord, sud, est e ovest, rispetto ad essa, e la cella stessa dato che il suo stato futuro dipende anche dal suo stato attuale. Se i possibili stati sono due – ad esempio 0 e 1 – allora il numero di configurazioni possibili del vicinato sono $2^5 = 32$. Ognuna di queste configurazioni può portare la cella nello stato 0 oppure 1, quindi in totale il numero di possibili configurazioni di vicinato con il rispettivo stato risultante sono $2^{2^5} = 2^{32} = 4\,294\,967\,296$.

Utilizzando invece il vicinato di *Moore* il numero di celle vicine sale a 9 perché, oltre a quelle considerate nel vicinato di von Neumann, aggiungiamo le celle adiacenti posizionate in diagonale rispetto a quella considerata. Utilizzando ancora due possibili stati – 0 e 1 – otteniamo $2^9 = 512$ possibili configurazioni del vicinato e quindi $2^{2^9} = 2^{512} = 1.340\,781 \times 10^{154}$ possibili risultati della cella centrale.

Uno stato iniziale, al tempo $t = 0$, viene definito assegnando ad ogni cella uno dei possibili stati h_k, con $1 \leq k \leq n$, dell'insieme H. Una nuova generazione viene creata, ad ogni passo $t = t+1$, seguendo un insieme di regole fissate che determina il nuovo stato per ogni cella a seconda dello stato attuale e dello stato del suo *vicinato*. Tipicamente l'insieme di regole per aggiornare lo stato di una cella è lo stesso per tutte le celle della griglia, non cambia nel tempo ed è applicata all'intera griglia simultaneamente.

Gli automi cellulari sono stati inizialmente concepiti da John von Neumann, mentre lavorava nei Los Alamos National Laboratory nel 1940 circa, come struttura ideale per "macchine" auto-replicanti. Nello stesso periodo Stanislaw Ulam, collega di von Neuman, stava usando lo stesso approccio per studiare la crescita dei cristalli.

2.1 Macchina di Turing

La macchina di Turing è un ipotetico dispositivo capace di manipolare simboli stampati su un nastro seguendo una tabella di regole. Nonostante la sua semplicità, una macchina di Turing può essere adattata per simulare la logica di ogni algoritmo.

Macchina di Turing universale

Tale macchina si dice universale quando è capace di simulare una qualunque macchina di Turing su un input arbitrario. In parole povere, quando può elaborare ogni possibile sequenza elaborabile.

Turing completezza

Un sistema di regole di manipolazione di dati si dice Turing completo se ha la stessa potenza computazionale di una macchina di Turing universale.

2.2 Autoreplicazione

L'approccio di von Neumann al problema dell'auto-replicazione era prettamente logico-matematico: se l'auto-replicazione *è* prodotta da una sistema adattativo

complesso naturale - *macchina biologica* -, allora il comportamento di questa macchina è descrivibile come una sequenza di passi logici, ad esempio un algoritmo. Ora, se un algoritmo può essere eseguito da una macchina qualsiasi, allora esiste una *macchina di Turing* che può eseguire lo stesso algoritmo. Per tale motivo von Neumann cercò di dimostrare l'esistenza di una macchina di Turing effettivamente capace di auto-riprodursi. Se tale macchina di Turing esiste, è plausibile che il processo per cui gli organismi viventi si riproducono, e per implicazione, gli altri processi su cui si basa la vita stessa, siano algoritmicamente descrivibili e, pertanto, la vita stessa sia riproducibile dalle macchine.

Von Neumann riuscì a mostrare una macchina di Turing universale in un automa cellulare usando un insieme di 29 stati per cella e un vicinato di 5 celle. La macchina è composta da un input, rappresentato da una sequenza di bit, una parte che esegue le operazioni logiche e una parte che produce l'output. La parte che esegue la computazione è progettata in modo da produrre in output la rappresentazione di una qualunque configurazione di bit che può essere letta in input. Ora, se noi gli passiamo in input la configurazione di bit necessaria per produrre se stesso in output e passargli una copia dei valori di input, abbiamo ottenuto una macchina che si auto-replicherà indefinitamente. Una macchina capace di costruire qualunque pattern è chiamata **costruttore universale**. [11]

Col tempo sono stati trovati altri costruttori universali come il *costruttore di Codd*, quello di *Devore*. I *Cicli di Langton* invece non sono delle macchina universali, sono solo capaci di replicarsi. [9]

2.3 Gioco della vita

Il Gioco della vita anche conosciuto col nome **Game of Life** è un automa cellulare ideato dal matematico John Horton Conway. È un gioco con zero giocatori, quindi la sua evoluzione è determinata solamente dal suo stato iniziale. Il gioco è nato dal tentativo riuscito di Conway di semplificare il costruttore universale di von Neumann. Pertanto, anch'esso è una macchina di Turing universale.

L'universo del gioco è un'infinita griglia bidimensionale di celle quadrate, ognuna delle quali si può trovare in due possibili stati, *vivo* o *morto*. Ogni cella può interagire con un vicinato composto dalle sue otto celle adiacenti. Ad ogni step temporale le celle vengono aggiornate secondo le seguenti regole:

1. Ogni cella viva con meno di due vicini vivi muore, a causa di isolamento.
2. Ogni cella viva con due o tre vicini vivi sopravvive.
3. Ogni cella viva con più di tre vicini vivi muore, a causa di sovrappopolazione.
4. Ogni cella morta con esattamente tre vicini vivi diventa una cella viva.

Il gioco è diventato subito molto famoso ed ha portato alla nascita di un nuovo campo di ricerca matematica, che è appunto quello sugli automi cellulari. Il motivo per cui questo automa ha attratto grande interesse è a causa del sorprendente modo con cui i pattern di celle possono evolvere. Nel gioco della vita si possono osservare esempi di auto-organizzazione e la nascita di comportamenti emergenti – caratteristiche che abbiamo visto essere tipiche dei SAC presenti in natura.

2.4 Classificazione

Stephen Wolfram, fisico ed informatico britannico, ha speso buona parte della sua vita investigando gli automi cellulari, spinto dalla facilità con cui questi sistemi permettessero di simulare fenomeni complessi. Nel suo libro *A New Kind of Science* esso presenta uno studio empirico sui sistemi computazionali basati su regole semplici – come gli automi cellulari – e sostiene che questi tipi di sistemi, al posto della matematica classica, siano necessari per modellare e capire la complessità in natura. Il suo lavoro lo porta ad affermare che l'universo è per sua natura digitale e che le sue regole fondamentali possono essere descritte con dei semplici programmi.[20]

Nel libro definisce quattro classi in cui gli automi cellulari, e molti altri modelli computazionali, possono essere divisi a seconda del loro comportamento. Wolfram ha cercato di classificare le regole di evoluzione di questi sistemi notando che, indipendentemente dalle condizioni iniziali, essi esibivano un comportamento con caratteristiche comuni. In ordine di complessità le classi sono:

Classe 1

I pattern iniziali evolvono rapidamente in uno stato stabile e omogeneo. Ogni casualità nel pattern iniziale scompare.

Classe 2

I pattern iniziali evolvono rapidamente in una struttura stabile o oscillante. Una parte della casualità nel pattern iniziale viene filtrata via, ma una parte rimane. Cambiamenti locali nei pattern iniziali tendono a rimanere locali.

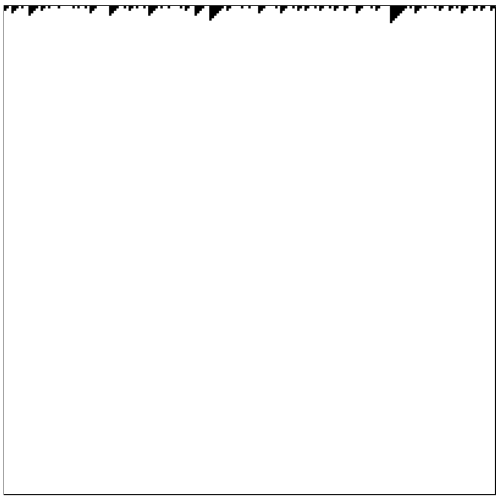
Classe 3

I pattern iniziali evolvono in maniera pseudo-random o caotica. Ogni struttura stabile che compare è distrutta dal rumore circostante. Cambiamenti locali nei pattern iniziali tendono ad espandersi indefinitamente.

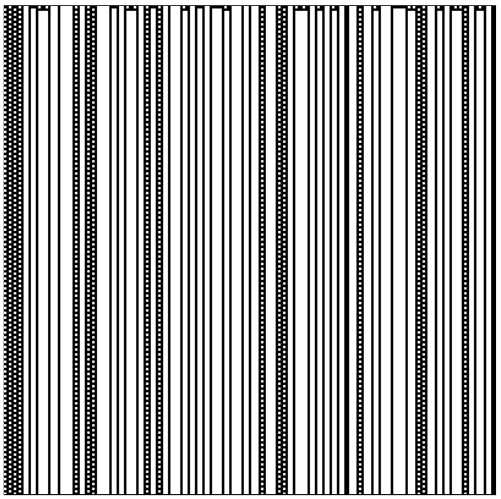
Classe 4

I pattern iniziali evolvono in strutture che interagiscono in modi complessi ed interessanti, formando strutture locali che riescono a sopravvivere per lunghi periodi di tempo. Può convergere verso una struttura stabile o oscillante – classe 2 – ma è necessario un gran numero di passi affinchè ciò accada. Wolfram suppone che molte, se non tutte, le regole di classe 4 siano delle *macchine di Turing universali.*

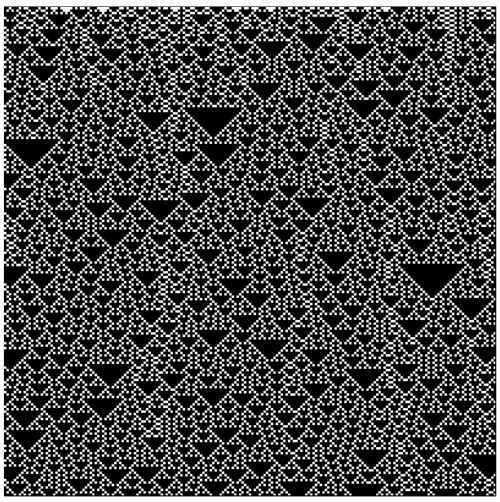
Figura 2.2: Esempi di classi di automi cellulari elementari



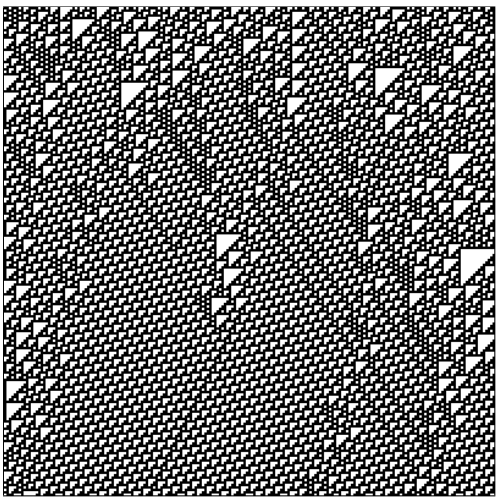
(a) *Classe 1 - regola 136*



(b) Classe 2 - regola 108



(c) Classe 3 - regola 182



(d) Classe 4 - regola 110

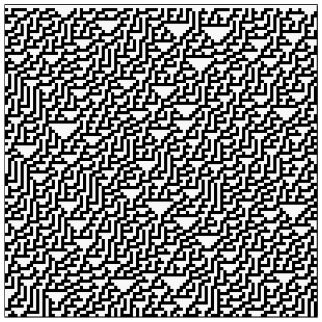
2.5 Biologia

Alcuni processi biologici possono essere simulati con gli automi cellulari. Di seguito alcuni esempi.

I Pattern di alcune conchiglie di mare sono generati da automi cellulari naturali. Ad esempio nel caso del *Conus textile*, le cellule dei pigmenti risiedono su una stretta striscia lungo il labbro del guscio. Ognuna di esse secerne i pigmenti a seconda dell'attività attivante o inibente delle sue cellule vicine, obbedendo quindi ad una versione naturale di una regola matematica. La striscia di cellule lascia il guscio colorato durante la crescita della conchiglia. È impressionante vedere come il guscio di questa conchiglia abbia dei motivi molto simili a quelli che genera l'automa cellulare con regola 30 – utilizzando la denominazione suggerita da Wolfram. [3]

Figura 2.3: Somiglianza tra i pattern del *Conus Textile* e quelli generati dalla regola 30

(a) *Conus Textile*



(b) Automa cellulare - regola 30

Le piante regolano la quantità d'aria assorbita e di gas rilasciati tramite un meccanismo ad automi cellulari. Ogni stoma sulla foglia agisce come una cella. [14]

Sono stati inventati automi cellulari che imitano i neuroni. Grazie ad essi si è riuscito a simulare comportamenti complessi come il riconoscimento e l'apprendimento. [7]

2.6 Reversibilità

Un automa cellulare si dice reversibile se, per ogni configurazione attuale dell'automa, esiste una ed una sola configurazione precedente. Se pensiamo all'automa cellulare come una funzione che mappa ogni configurazione in un'altra, la reversibilità implica che questa funzione è *biettiva*. Se un automa cellulare è reversibile allora il suo comportamento, invertendo il tempo, può essere anch'esso descritto con un automa cellulare.

Per gli automi ad una dimensione si conoscono algoritmi per decidere se una regola è reversibile o irreversibile. Però, per automi con dimensioni maggiori a uno la reversibilità è un problema *indecidibile*, cioè non è possibile trovare un algoritmo che ci garantisca di trovare sempre un risultato negativo o positivo per quel problema.

Gli automi cellulari reversibili sono spesso usati per simulare fenomeni fisici come le dinamiche di gas e fluidi, poiché obbediscono alle leggi della termodinamica.

3 Algoritmi genetici

L'algoritmo genetico è un algoritmo di ricerca euristico che simula il processo della selezione naturale teorizzato da Charles Darwin nel 1859. L'idea che ha spinto la nascita di questi algoritmi è che se la natura, tramite la selezione e l'evoluzione, è riuscita a creare cose talmente complesse – come il cervello umano – allora adottando lo stesso approccio possiamo trovare soluzioni a problemi altrettanto complessi.

3.1 Esperimenti noti

Sono molti i casi in cui gli studiosi hanno messo in pratica dei veri e propri algoritmi genetici. Nel corso degli anni si è quindi potuto osservare come questo tipo di metodologia sia utile nelle applicazioni tecniche e allo stesso tempo essenziale per rispondere ad alcune delle domande ancora irrisolte sulla biologia.

3.1.1 E. coli long-term evolution experiment

Richard Lenski, noto biologo americano, porta avanti dal 24 febbraio 1988 il cosiddetto *E. coli long-term evolution experiment*: uno studio sperimentale sull'evoluzione di 12 popolazioni inizialmente identiche del batterio Escherichia coli. Ad oggi, superate le 60 000 generazioni, una di queste 12 popolazioni si è evoluta in modo da essere in grado di usare il citrato come sorgente di nutrimento.

Figura 3.1: Le 12 popolazioni di E. coli

3.1.2 Evolved antenna

A metà degli anni 90, la prima *antenna evoluta* fece la sua comparsa nel mondo delle radio comunicazioni. Un'antenna evoluta non viene progettata manualmente, bensì tramite l'utilizzo di un algoritmo genetico.

Figura 3.2: Un'antenna evoluta

L'algoritmo parte considerando delle forme semplici di antenna, alle quali prova a fare delle alterazioni e delle aggiunte in una maniera semicasuale in modo da generare una certa quantità di *candidati*. Questi vengono a loro volta valutati per determinare quanto sono conformi ai requisiti di progettazione.

Dopodiché, analogamente a ciò che accadrebbe in biologia a causa della selezione naturale, una parte di questi candidati — quella che viene valutata come la "peggiore" — viene scartata, lasciando intatta una popolazione ridotta ma composta solo dalle antenne con un design efficiente.

Le antenne che si ottengono grazie all'applicazione di un certo numero di iterazioni di questo algoritmo sono generalmente molto migliori delle migliori antenne progettate manualmente, perché le forme asimmetriche assunte dal design "evoluto" sono virtualmente impossibili da trovare con i classici metodi di progettazione manuale.

3.2 Algoritmi euristici

Un algoritmo euristico è una tecnica progettata per risolvere un problema rapidamente quando un metodo classico è troppo lento, o per trovare una soluzione approssimata quando un metodo classico fallisce nel trovare la soluzione ottima.

L'obbiettivo di questa categoria di algoritmi quindi non è tanto quello di trovare la soluzione ottima ad un problema, ma quello di restituire velocemente un risultato che si avvicini ad essa, se non è la soluzione ottima stessa. Spesso sono usati in congiunzione con altri algoritmi per velocizzare la ricerca di un risultato.

Per alcuni tipi di problemi – ad esempio i problemi *NP-completi* – gli algoritmi euristici sono l'unico approccio possibile.

3.3 Evoluzione darwiniana

In biologia con evoluzione si intende il processo che porta al manifestarsi di significativi cambiamenti morfologici, strutturali e funzionali negli organismi viventi. Il processo si manifesta col progressivo accumularsi di piccole modifiche nel patrimonio genetico che gli individui trasmettono alla propria progenie.

Negli organismi viventi i tratti che caratterizzano ogni individuo sono codificati nei *geni*. I geni sono raggruppati in lunghe catene, semplificando ed astraendo, chiamate *cromosomi*. Ogni gene contribuisce a rappresentare uno specifico tratto dell'organismo, come il colore degli occhi o attitudini comportamentali, e può assumere diversi valori. L'insieme dei geni e dei valori che assumono è generalmente riferito come *genotipo*. L'espressione fisica del genotipo, quindi l'organismo stesso con i suoi comportamenti, si chiama *fenotipo*.

L'evoluzione naturale viene attuata tramite tre elementi fondamentali:

Ereditarietà
: Processo per il quale i figli ricevono le caratteristiche dei propri genitori. Se le creature vivono abbastanza a lungo per riprodursi, allora i propri tratti vengono trasmessi ai figli nella generazione successiva.

Variazione
: Deve essere presente nella popolazione una varietà di tratti o un meccanismo che introduca variabilità. Senza nessuna varietà nella popolazione, i figli saranno sempre identici ai genitori e tra di loro. Nuove combinazioni di tratti non possono presentarsi e non c'è evoluzione.

Selezione
: Meccanismo per il quale alcuni membri della popolazione hanno l'opportunità di essere genitori, e passare i propri tratti ai figli, ed altri no. È solitamente chiamata "sopravvivenza del più forte". I geni che determinano maggior probabilità di riprodursi sono quelli che determinano un miglior adattamento all'ambiente e hanno maggior probabilità di sopravvivenza.

3.4 Funzionamento

Avendo acquisito un po' di background dalla biologia, possiamo ora definire il modo con cui gli algoritmi genetici sfruttano l'evoluzione naturale per risolvere problemi, anche distanti dal contesto della natura. Non è immediato capire come poter applicare il metodo della selezione naturale per problemi che risolviamo tipicamente con un computer. In genere i problemi risolvibili con un calcolatore sono del tipo "dato un insieme di valori in input, trovare un valore in output che sia il migliore tra tutti i possibili risultati". In natura invece il dominio dei valori, di input e di output, non è ben definito e quindi facciamo difficoltà a fare analogie.

Torniamo un attimo nel mondo naturale e consideriamo qual è in questo caso il problema da risolvere e quali sono le possibili soluzioni al problema. Assodato che il processo che cerca di risolvere il problema è la *selezione naturale*, il problema può essere definito come massimizzare il numero di figli, o meglio "massimizzare la diffusione dei propri geni". Appare ovvio ora che le soluzioni al problema sono i figli, quindi gli organismi stessi. La soluzione sarà tanto migliore quanto migliore sarà la capacità dei figli di sopravvivere e riprodursi, quindi di trasmettere i propri geni.

Nel suo libro *The selfish gene* Richard Dawkins spiega appunto la sua visione della teoria evoluzionistica secondo la quale i geni "egoisti" sono i soggetti principali della selezione naturale, e non gli individui. [5]

3.4.1 Rappresentazione del fenotipo

Abbiamo appena definito gli organismi come le soluzioni al nostro problema di ottimizzazione. Ricordiamo inoltre che un organismo è il fenotipo, essendo quest'ultimo la rappresentazione fisica del genotipo. Possiamo affermare quindi che, nel contesto del nostro algoritmo genetico, il fenotipo rappresenta una possibile soluzione al problema.

Il fenotipo funge anche da input del nostro algoritmo. Ad ogni step infatti, il risultato del passo precedente rappresenta l'input per l'attuale passo di evoluzione.

Dobbiamo tener conto ora di un importante elemento dell'evoluzione: la variazione. Come nel caso dell'evoluzione naturale, abbiamo bisogno di assicurare una buona variazione tra gli individui della popolazione. Se facessimo partire l'algo-

ritmo genetico su una popolazione composta da un solo individuo, non avremmo alcuna evoluzione. Perciò solitamente sceglieremo una popolazione iniziale abbastanza numerosa. Più è numerosa più assicuriamo differenziazione tra gli individui – fenotipi.

3.4.2 Rappresentazione del genotipo

Il genotipo abbiamo detto essere ciò che internamente rappresenta ogni individuo della popolazione. Chiaramente il suo dominio può variare molto a seconda del tipo di problema e di cosa sono i fenotipi. In generale però possiamo dire che, dato che su un computer tutto viene rappresentato con una sequenza di bit, possiamo definire il genotipo come una stringa di bit.

0	0	0	0	1	1	1	1	0	1	1	0

Figura 3.3: Esempio di rappresentazione del genotipo

Dato un genotipo – stringa di bit – dobbiamo poter estrarre le regole per rappresentare il fenotipo. Quindi la stringa deve essere strutturata, a seconda del contesto, in modo da permettere un mapping tra genotipo e fenotipo.

3.4.3 Ereditarietà

L'ereditarietà è il meccanismo che permette di trasmettere il proprio genotipo, o parte di esso, ai figli.

I modi per passare la propria stringa di bit nella generazione successiva possono essere molti, come ci insegna la natura. Possiamo infatti simulare la riproduzione asessuata – lasciando che il genotipo di un individuo sia ereditato in parte da un solo genitore. Oppure possiamo simulare la riproduzione tra due individui – generando, ad ogni passo, dei figli che ereditano una parte del genotipo da un genitore e l'altra parte da un altro.

Solitamente si adotta quest'ultimo approccio perché permette una maggiore variazione nei risultati, dato che mescola i tratti degli individui.

Per unire due genotipi – ricordiamo che sono stringhe di bit – possiamo scegliere un indice k, compreso tra 1 e la lunghezza della stringa, e nella stringa del figlio ricopiare la prima parte di stringa, compresa tra 1 e k, con il corrispondente pezzo

di stringa del primo genitore e il pezzo del secondo genitore. Altrimenti possiamo, per ogni casella della stringa, ricopiare il valore corrispondete alla stringa di uno dei due genitori scelto casualmente.

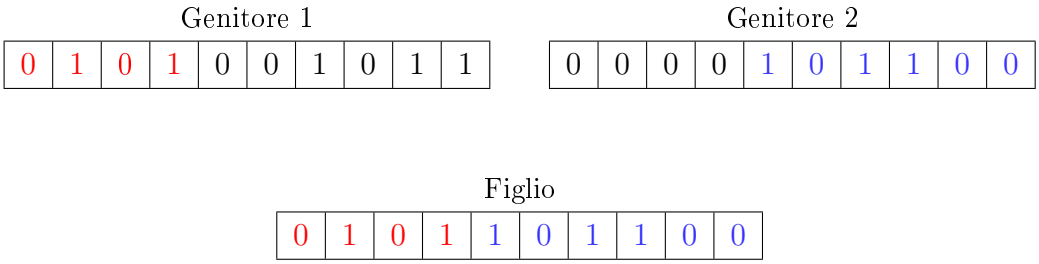


Figura 3.4: Esempio di meccanismo di ereditarietà, $k = 4$

3.4.4 Mutazione

Spesso il semplice fatto di creare una popolazione numerosa non basta per assicurare un'adeguata variazione tra gli individui.

Consideriamo, ad esempio, il genotipo in figura 3.3 come la stringa che rappresenta la soluzione ottima al nostro problema. Se la popolazione non contiene nessun un individuo il cui genotipo ha il primo bit a zero, per quanto mischiamo le stringhe di bit tra di loro, non otterremo mai la soluzione desiderata. La soluzione migliore che potremmo trovare sarebbe un *massimo locale* tra tutte possibili soluzioni al problema.

Per evitare di incappare in situazioni del genere dobbiamo utilizzare un meccanismo che introduce degli elementi nuovi, non provenienti dalla popolazione stessa. La mutazione appunto consiste nel cambiare casualmente il valore di una cella scelta a caso nella stringa di un individuo. In genere si imposta una frequenza di mutazione con cui è possibile variare la probabilità con cui ogni cella venga cambiata di valore.

3.4.5 Selezione

La selezione è un meccanismo fondamentale per l'evoluzione di una popolazione. Infatti, essa permette di selezionare, ad ogni passo evolutivo, gli individui "migliori" e scartare quelli "peggiori".

Nel caso della natura non c'è un vero e proprio meccanismo che permette di etichettare gli individui in base alle loro qualità, semplicemente è migliore chi riesce a riprodursi di più. Nel nostro caso invece gli individui non sono un'entità attiva, capaci di agire autonomamente, quindi siamo noi che dobbiamo agire da selettori decidendo chi "merita" di riprodursi e chi no. Dobbiamo a tal scopo definire, a seconda del contesto, un modo per assegnare un valore di *fitness* ad ogni individuo alla fine di ogni passo evolutivo. Questo valore deve poterci indicare, su una scala di valori, quanto un individuo è "buono". In questo modo, ad ogni passo possiamo selezionare gli individui migliori che potranno riprodursi e trasmettere i propri tratti alla generazione successiva.

Figura 3.5: Pseudo-codice algoritmo genetico

```
Crea una popolazione di individui generati casualmente
REPEAT
  Calcola il valore di fitness per ogni individuo
  REPEAT
    Seleziona alcuni individui in base al loro valore di fitness
    Genera dagli individui selezionati dei figli
    Inserisci i figli in una nuova popolazione
  UNTIL la nuova popolazione e' piena
  Applica la mutazione sugli individui della nuova popolazione
  Scarta la popolazione vecchia
UNTIL non e' presente un individuo ottimo
```

4 Visione unificata

Lo scopo di questo capitolo è quello di cercare di mettere insieme le parti di teoria finora analizzate, per poter arrivare ad affermare che gli automi cellulari possono essere visti come dei sistemi adattativi complessi. Vedremo che in determinate condizioni essi sono effettivamente dei SAC mostrando un comportamento emergente e riuscendo a produrre adattamento.

D'ora in poi, per evitare di fare confusione, utilizzeremo la parola *evoluzione* per indicare il processo di miglioramento della soluzione negli algoritmi genetici, mentre ci riferiremo al progresso degli automi cellulari nel tempo con il termine *sviluppo*.

4.1 Automi cellulari come sistemi complessi

I sistemi complessi, come abbiamo visto nel primo capitolo, sono tali per cui è impossibile riuscire a progettarne uno a partire dai requisiti che il sistema deve avere. Né tanto meno guardando i componenti riusciamo a inferire quale sarà il comportamento complessivo.

Gli automi cellulari possono essere considerati sistemi complessi, e ci permettono di simulare molti sistemi composti da tanti individui che seguono regole semplici. Sono sistemi complessi perché abbiamo visto che posseggono molte caratteristiche tipiche dei SAC.

Aggregazione Gli automi cellulari sono composti da un gran numero di elementi – celle –, ma possono essere considerati come un'unica entità. Anche loro, come i SAC, esibiscono un comportamento emergente – vedi figura 2.2 – totalmente inaspettato guardando il comportamento delle singole celle.

Non-linearità Sono chiaramente non-lineari dato che ogni cella si sviluppa considerando lo stato dei suoi vicini. Per rendersene conto basta osservare che utilizzando il vicinato di von Neumann – figura 2.1a – il numero di possibili stati di una cella sono dell'ordine di 4×10^9, mentre aggiungendo solamente quattro vicini – figura 2.1b – il numero aumenta di un fattore 10^{145}.

Flusso Il fatto che ogni cella si sviluppa a seconda del proprio stato attuale e dello stato dei suoi vicini, può esser visto come una forma di interazione tra le componenti dell'automa. Chiaramente in questo caso i flussi sono statici in quanto tipicamente vengono definiti all'inizio dello sviluppo e non variano nel tempo.

Diversità Ogni cella possiede uno stato, che può assumere diversi valori. Finora abbiamo analizzato solamente stati binari ma in generale il numero di stati può essere anche molto grande. Per via della non linearità, sappiamo che aumentando il numero di stati i possibili risultati aumentano con un fattore moltiplicativo.

Come visto per i SAC anche per gli automi cellulari si osserva una convergenza verso una determinata configurazione. Abbiamo visto – figura 2.2 – che gli automi ricadono in determinati modelli, indipendentemente dallo stato iniziale delle celle.

Modello interno Gli automi cellulari posseggono un modello interno che è rappresentato dalle loro regole di sviluppo. Grazie al modello interno ogni cella sa come deve svilupparsi nella prossima generazione a fronte degli stimoli in input, provenienti dai flussi che lo collegano ai suoi vicini.

Nonostante le similarità tra gli automi cellulari e i SAC vediamo che mancano ancora dei pezzi affinché possiamo effettivamente considerare gli automi come dei SAC. Gli automi, per come li abbiamo visti finora, non mostrano adattamento. Per quanto li facciamo sviluppare il loro comportamento, quindi le loro regole, non variano nel tempo, nè variano le interazioni tra una cella e l'altra.

Possiamo, per il momento, affermare che gli automi cellulari sono dei "semplici" sistemi complessi.

4.2 Automi cellulari come sistemi *adattativi* complessi

Nel capitolo 3 abbiamo visto che la natura riesce a generare adattamento utilizzando tre semplici meccanismi, *ereditarietà*, *mutazione* e *selezione*. Ed abbiamo anche

visto che esiste una classe di algoritmi – gli **algoritmi genetici** – che permettono di applicare lo stesso meccanismo per la risoluzione di svariati problemi. Pertanto, eseguendo un algoritmo genetico su una popolazione di automi cellulari dovrebbe avere l'effetto di evolvere questi ultimi.

Dobbiamo prima definire il modo con cui rappresentiamo il *fenotipo*, il *genotipo* ed in che modo attuiamo i meccanismi di *ereditarietà*, *mutazione* e *selezione*.

Rappresentazione del fenotipo In un algoritmo genetico il fenotipo rappresenta uno degli individui che vogliamo evolvere. Come detto precedentemente, ad ogni passo dell'algoritmo, l'insieme degli individui di output del passo precedente funge da input per il passo attuale.

Nel caso degli automi cellulari gli individui sono rappresentati dalla griglia composta dalle singole celle. D'ora in poi assumeremo che l'automa cellulare sia bidimensionale quindi la griglia è una matrice quadrata.

Rappresentazione del genotipo Il genotipo invece rappresenta internamente l'individuo, quindi è ciò che prima abbiamo definito essere il modello interno dell'automa cellulare: le regole di sviluppo. Queste regole ci dicono, ad ogni passo, come sviluppare le nuove celle. In generale considerando un vicinato grande k e un numero di stati n, abbiamo che il numero di possibili configurazioni sono n^k. Per ognuna di queste configurazioni la cella assumerà uno dei possibili valori degli stati. Quindi possiamo rappresentare il genotipo come una stringa lunga n^k dove ogni posizione della stringa rappresenta una possibile configurazione e il valore contenuto è quello che assumerà la cella.

Oltre alle regole c'è un altro elemento da cui un automa cellulare dipende, lo stato iniziale delle celle. Due automi con le stesse regole ma con griglie iniziali diverse, si sviluppano in modo diverso, anche se il comportamento generale può essere simile. Quindi fa parte del genotipo di un'automa cellulare anche la griglia contenente lo stato iniziale delle celle.

Ereditarietà Abbiamo definito il genotipo come composto da una stringa di bit – le regole di sviluppo – e una griglia – lo stato iniziale. Quindi quando abbiamo finito una fase evolutiva e ci troviamo a dover generare un nuovo figlio – assumiamo che un individuo viene generato da due genitori – dobbiamo fare in modo che questo erediti le componenti del genotipo da entrambi i genitori.

Per quanto riguarda la stringa con le regole possiamo direttamente utilizzare uno dei due metodi spiegato nella sezione 3.4.3. Anche la griglia iniziale – matrice quadrata – può essere vista come un vettore – scomponendo le righe e mettendole una di seguito all'altra –, e quindi possiamo utlizzare un metodo analogo.

Mutazione La mutazione può essere effettuata come spiegato nella sezione 3.4.4, scambiando quindi con una certa probabilità il valore di ogni cella del genotipo. Lo stesso metodo può essere applicato sia per quanto riguarda le regole che per la griglia iniziale.

Selezione Per decidere come applicare la selezione dobbiamo prima decidere quali sono i fattori che discriminano un individuo "buono" da uno "non buono". Questa scelta dipende da cosa vogliamo ottenere dall'evoluzione degli automi cellulari, da qual è il nostro obiettivo. Ad esempio possiamo pensare di far evolvere gli automi affinché lo stato delle celle nella loro griglia assuma una specifica configurazione. Oppure potremmo farli evolvere allo scopo di eseguire delle specifiche operazioni sullo stato iniziale delle celle.

Prendendo in considerazione il primo esempio, ad ogni passo dell'evoluzione dobbiamo calcolare un valore che chiamiamo di *fitness* che ci indica quanto la griglia dell'automa è uguale alla griglia obiettivo. Una volta calcolato il valore di fitness per ogni elemento della popolazione possiamo selezionare i migliori e da loro generare i figli.

Abbiamo definito una tecnica che permette agli automi cellulari di evolversi di generazione in generazione, fino alla comparsa delle caratteristiche che noi abbiamo definito essere l'obiettivo dell'evoluzione. Ora, gli automi non hanno più un set di regole statiche, nè una configurazione iniziale fissata durante le diverse generazioni.

5 Evolvere automi cellulari

In questo capitolo mostreremo i primi esperimenti effettuati per l'evoluzione di automi cellulari. L'evoluzione sarà guidata da una piccola immagine binaria target, che dovrà essere quindi lo stato finale dell'automa se l'evoluzione ha successo. Alla fine mostreremo i risultati ottenuti.

Lavori passati – [10] e [4] – hanno mostrato che usare gli algoritmi genetici sugli automi cellulari monodimensionale può risultare una combinazione molto efficiente. In un interessante articolo [2] viene mostrato come utilizzare lo stesso approccio su automi multidimensionali migliora notevolmente le prestazioni. Nello specifico viene utilizzato tale approccio per risolvere il *majority problem* e per evolvere piccole bitmap binarie.

5.1 Majority problem

Il *majority problem* è spesso usato per mostrare le potenzialità degli automi cellulari. È un semplice esempio di come regole locali possono compiere un compito globale. Tale problema può essere definito come segue:

Dato un insieme $A = \{a_1, \ldots, a_n\}$ *con* n *dispari e* $a_m \in \{0, 1\}$ *per ogni* $1 \leq m \leq n$*, rispondi alla domanda: "Ci sono più uno o più zero in A?".*

Nonostante, a prima vista, non sembri un problema molto difficile da risolvere, quando questo problema è portato nell'ambito degli automi cellulari diventa parecchio difficile. Ciò accade perché le regole in un automa sono locali, non permettono ad una cella di guardare al di fuori dei suoi vicini. Quindi le celle devono lavorare insieme e usare una sorta di comunicazione per trovare la soluzione.

Se, dopo un certo numero di passi, lo stato di tutte le celle dell'automa è uguale allo stato di maggioranza nella configurazione iniziale, allora l'automa ha risolto il problema.

Nel 1995 [8] è stato dimostrato che non esiste una regola con due stati che risolve tale problema su qualunque configurazione di partenza quando il numero di celle è abbastanza grande, e il numero di vicini è abbastanza piccolo. Rilasciando

però alcuni vincoli, per i quali possiamo dire che l'automa ha riconosciuto qual è lo stato di maggioranza, è possibile trovare una regola esatta per tutti gli stati iniziali.

5.2 Evolvere bitmap

Questo problema è un caso più generale e complesso del precedente, in quanto adesso lo stato dell'automa dopo un certo numero di passi deve essere uguale ad una configurazione fissata. Il problema può essere definito come:

Dato uno stato iniziale ed un desiderato stato finale: cerca una regola che sviluppa lo stato iniziale nello stato finale desiderato in meno di k passi.

Anche in questo caso non è assicurato che esista sempre una regola che ci permetta di finire nello stato desiderato. Bisogna sempre considerare la grandezza del vicinato – cioè con quanti altri una cella può comunicare – e la grandezza della griglia.

5.3 Prove effettuate

Di seguito mostreremo gli esperimenti effettuati per evolvere delle piccole immagini bitmap. Queste prime prove ci hanno permesso innanzitutto di dimostrare l'efficacia dell'approccio descritto finora per evolvere gli automi cellulari. In oltre abbiamo osservato come varia l'evoluzione, modificando i parametri dell'algoritmo genetico e la configurazione iniziale degli automi.

Di seguito verranno illustrati i due differenti approcci sperimentati. Per prima cosa però vediamo quali sono le caratteristiche comuni, tra tutti e due gli esperimenti, degli automi cellulari e dell'algoritmo genetico.

5.3.1 Automi cellulari

Nel primo esperimento sono stati utilizzati automi cellulari bidimensionali di ordine $n = 5$, composti quindi da una griglia 5×5. Gli stati possibili per ogni cella sono $H = \{0, 1\}$.

Il vicinato scelto è quello di von Neumann – figura 2.1a – quindi il set di regole è un vettore binario contenente 32 celle, una per ogni possibile configurazione dei

vicini. Le griglie sono toroidali, nel senso che i bordi sono collegati tra loro (il bordo superiore con il bordo inferiore e quello di destra con il bordo di sinistra).

Le regole iniziali di sviluppo sono scelte a caso, quindi ogni singolo automa cellulare può avere una delle 2^{32} possibili configurazioni di regole. Ogni automa ha la possibilità di svilupparsi fino alla generazione $g = 50$ – generazioni degli stati dell'automa, non dell'algoritmo genetico.

5.3.2 Algoritmo genetico

L'algoritmo genetico ha quindi lo scopo di evolvere le regole degli automi cellulari affinché l'automa generi lo schema voluto.

È stata creata una popolazione di automi cellulari composta da 200 individui. Per ogni passo dell'algoritmo genetico gli automi vengono fatti sviluppare fino g volte e per ogni passo di sviluppo viene controllato se lo schema corrisponde all'obiettivo. In caso affermativo l'automa viene "bloccato", quindi non si sviluppa più.

L'algoritmo è stato ripetuto più volte, su popolazioni ogni volta nuove, utilizzando diversi valori di mutazione $\mu \in \{0.01, 0.06, 0.1, 0.15, 0.2\}$ per vedere come tale fattore influisse sulle prestazioni dell'algoritmo. Per ogni valore di μ sono stati effettuati 50 esperimenti in modo da poter avere una stima abbastanza accurata sui risultati ottenuti.

Per ogni nuovo tentativo l'algoritmo è stato fatto girare fino a 50 generazioni – dell'algoritmo genetico questa volta – e poi, anche in caso di fallimento, il programma è stato fermato.

Le bitmap obiettivo su cui è stato testato l'algoritmo sono le seguenti:

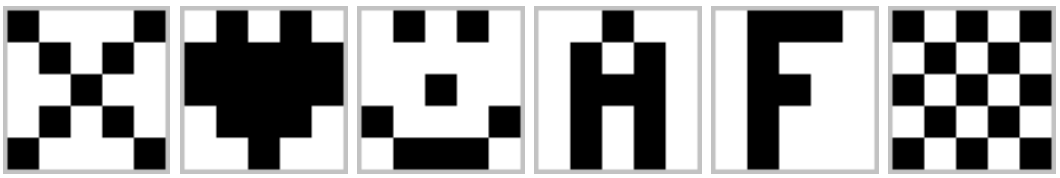


Figura 5.1: Bitmap obiettivo

Il meccanismo di ereditarietà, alla fine di ogni passo evolutivo, è stato applicato creando un insieme di automi cellulari – *mating pool* – presi dalla popolazione e "pescando" casualmente da lì gli individui da far riprodurre. In questo insieme ogni individuo è stato inserito un certo numero di volte α scelto in base al suo valore di fitness $f, f \in \{0, \dots 25\}$. In particolare $\alpha = \lfloor 2^f / 1 \times 10^5 \rceil$. In questo modo permettiamo solamente a quelli il cui valore f supera una certa soglia minima ϵ di riprodursi. Dalla formula di α ricaviamo che tale valore di soglia è $\epsilon = 15$, dato che $\alpha_{f=15} = \lfloor 2^{15} / 1 \times 10^5 \rceil = \lfloor 0.327\,68 \rceil = 0$ e $\alpha_{f=16} = \lfloor 2^{16} / 1 \times 10^5 \rceil = \lfloor 0.655\,36 \rceil = 1$. In conclusione, dato un automa e il suo valore di fitness f^* esso avrà una probabilità di riprodursi pari a zero se $f^* \leq 15$, ma la possibilità aumenta in modo esponenziale con l'aumentare di f^*. Se $f^* = 20 \Rightarrow \alpha = 10$, mentre se $f^* = 24 \Rightarrow \alpha = 168$.

Il meccanismo di trasmissione dei geni tra una generazione e l'altra viene applicato prendendo a caso due individui dal mating pool e producendo dei nuovi individui a partire da essi. Ad ogni generazione tutta la popolazione viene rimpiazzata dai figli, ad eccezione dei migliori dieci che invece sopravvivono.

5.3.3 Prova 1: configurazione iniziale fissata

In questo primo esperimento abbiamo scelto di utilizzare una configurazione fissa per la griglia iniziale di ogni automa cellulare – figura 5.2.

Figura 5.2: Configurazione iniziale

Il meccanismo di ereditarietà viene applicato producendo due figli da ogni coppia di genitori. Le regole di sviluppo dei due figli saranno una ricombinazione delle regole dei due genitori. Nello specifico preso uno dei figli, e presa una qualunque delle sue regole, questa proviene dal genitore 1 con una probabilità del 50% oppure dal due con la stessa probabilità. Presa la corrispondente regola del fratello, questa proverrà dal primo genitore se nel fratello proveniva dal secondo, dal genitore 2 altrimenti.

Nella tabella 5.1 vediamo riportati il numero di regole di successo trovate eseguendo l'algoritmo descritto precedentemente.

Osserviamo che il numero di regole trovate varia parecchio a seconda della configurazione delle celle. In particolare notiamo che le bitmap simmetriche, quindi la 5.1a e la 5.1f, sono molto più facili da ricercare rispetto a quelle non simmetriche.

D'istinto ci verrebbe da dire che ciò accade a causa della configurazione iniziale – che è stata fissata ad una cella nera al centro – la quale ci limita nell'esplorazione di tutte le possibili configurazioni.

Tabella 5.1: Regole ottime trovate con configurazione iniziale fissa

Bitmap	Regole trovate
(a)	60
(b)	11
(c)	5
(d)	6
(e)	1
(f)	75

Nella prossima pagina abbiamo rappresentato con degli istogrammi – figura 5.3 – il numero di regole ottime trovate per ogni valore di mutazione. Osserviamo che i valori per le figure 5.1a e 5.1f non sono molto significative a causa del fatto che è molto facile trovare delle regole che le generano. Considerando quindi le restanti possiamo affermare che in generale con l'aumento del valore di mutazione le prestazioni peggiorano. Infatti con un valore troppo elevato può accadere che l'adattamento acquisito finora venga perduto a causa della continua modifica casuale delle regole. D'altra parte però il valore di mutazione non può essere neanche troppo basso altrimenti rischieremmo di dover aspettare troppo tempo affinché alcuni dei tratti che ci interessano si manifestino.

Nella figura 5.4 invece abbiamo riportato il valore medio di fitness al variare del valore di mutazione. I risultati in questo caso non sono molto significativi, a causa del basso numero di prove effettuate – dato che effettuare un numero maggiore di prove avrebbe richiesto troppo tempo. Comunque osserviamo che in generale un valore troppo alto o troppo basso di mutazione non permette di ottenere degli ottimi risultati. Un buon valore di mutazione in generale si trova empiricamente dato che può variare molto a seconda dell'applicazione, dalla popolazione e dal metodo di ereditarietà.

In figura 5.5 sono rappresentate le stringhe binarie di alcune regole trovate con i relativi sviluppi della griglia. È molto interessante vedere come alcuni automi presentano un comportamento molto simile tra di loro, a volte speculare o altre volte addirittura identico, nonostante seguino regole di sviluppo differenti.

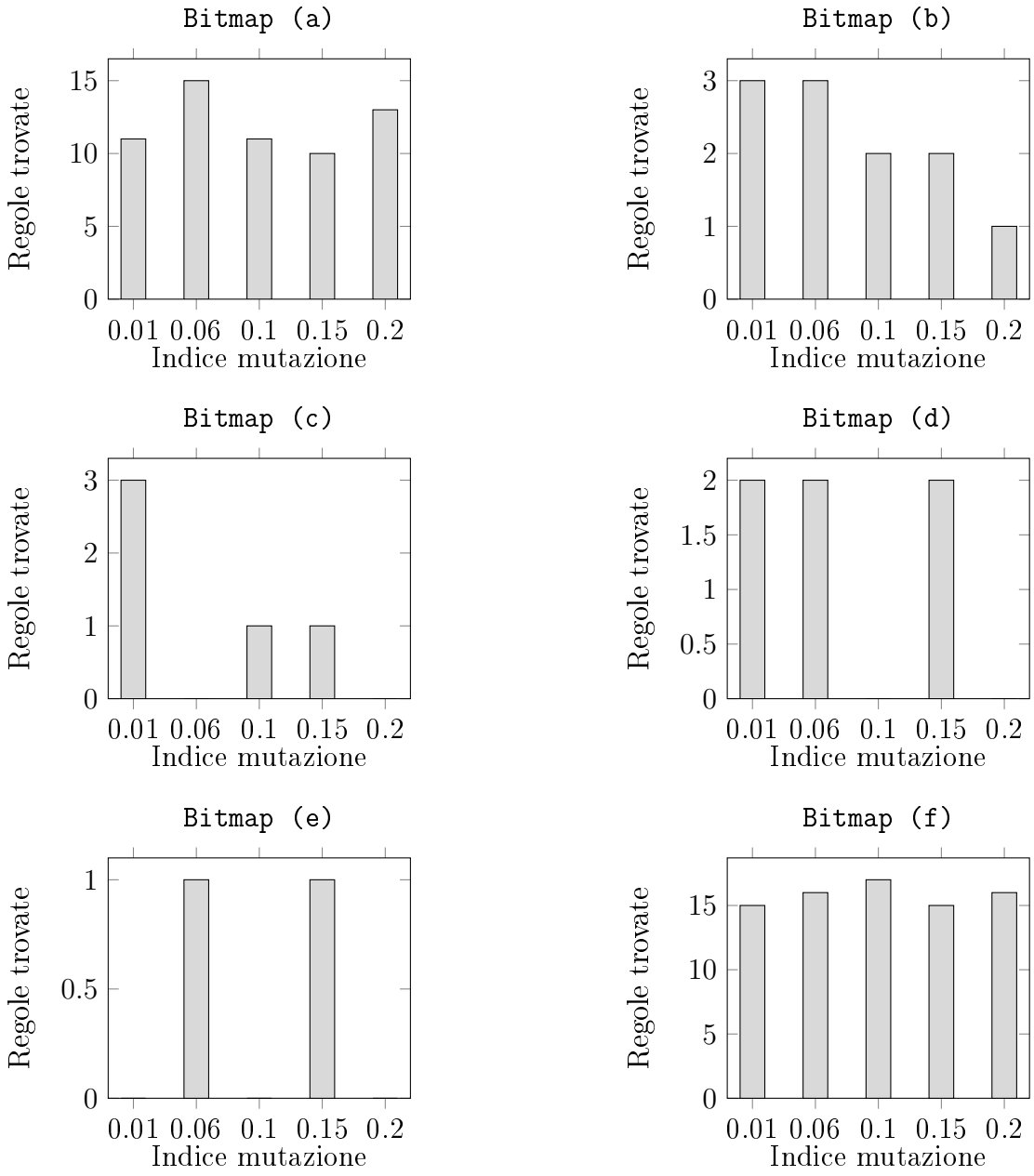


Figura 5.3: Regole trovate al variare della mutazione

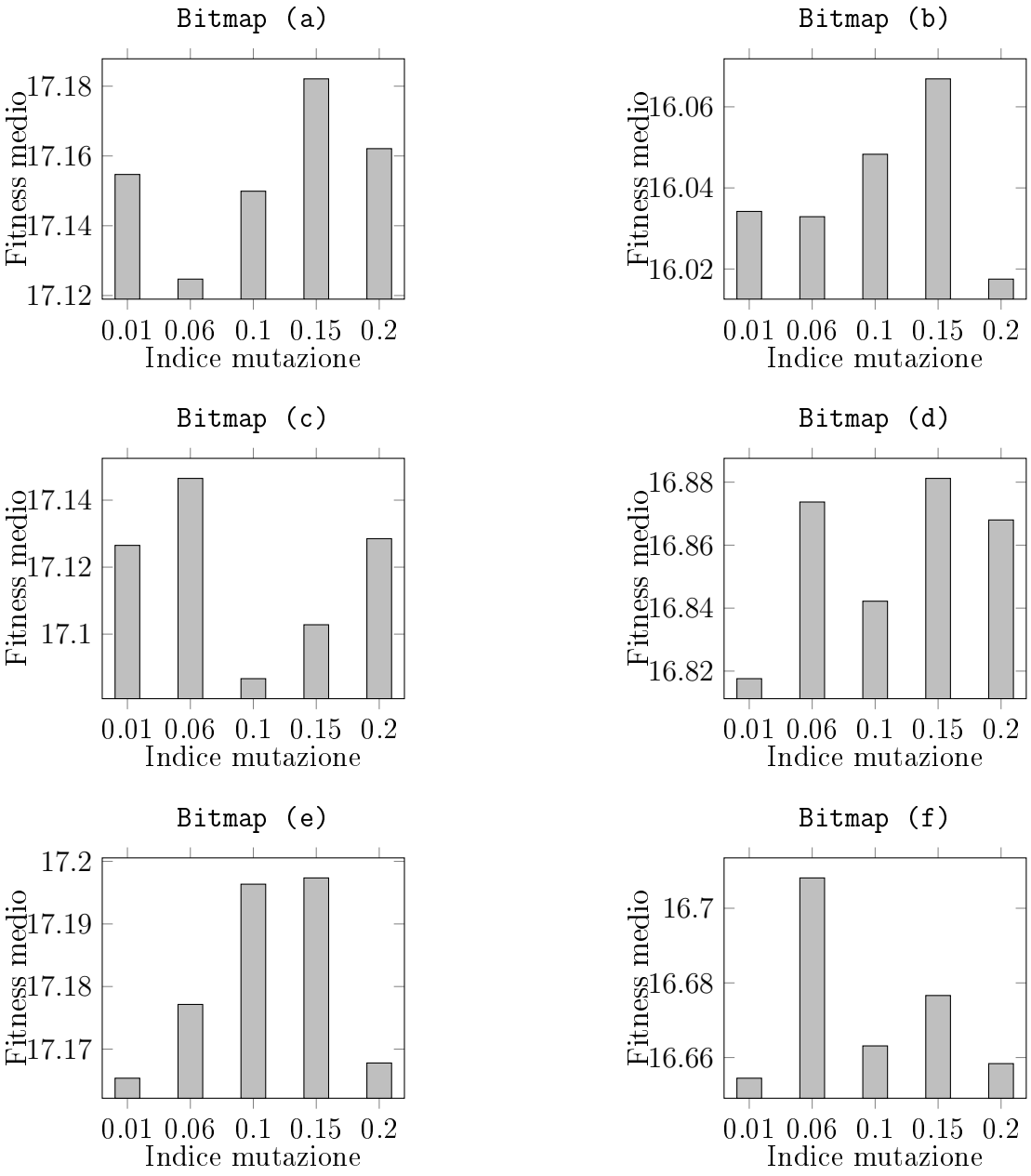
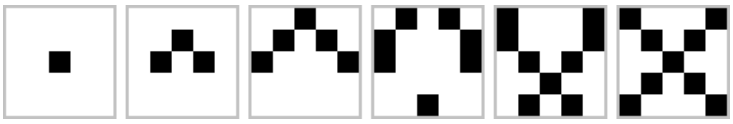


Figura 5.4: Fitness medio della popolazione al variare della mutazione

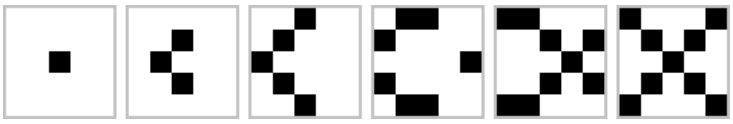
Figura 5.5: Sviluppo di alcune regole trovate

Bitmap (a)



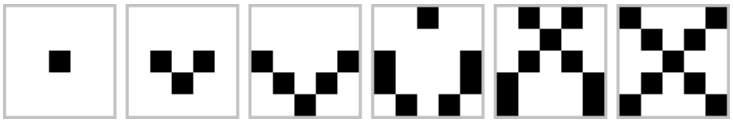
10000010100110100111111111001110

01001010100010000011111111001110



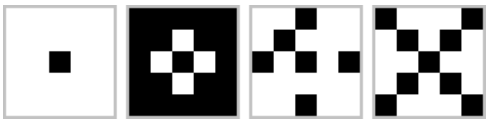
10001100111111010101110011101110

10100100011111110100100010101110

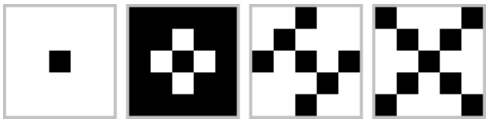


10010111111111110101000101100100

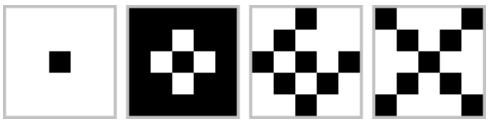
11110111111111111110101010100100



01110001100001001000101000010001



01110000101000101000001010010001



01110100100010101110010010010001

01001001110001111000001100001110

01111001000001011100011100001110

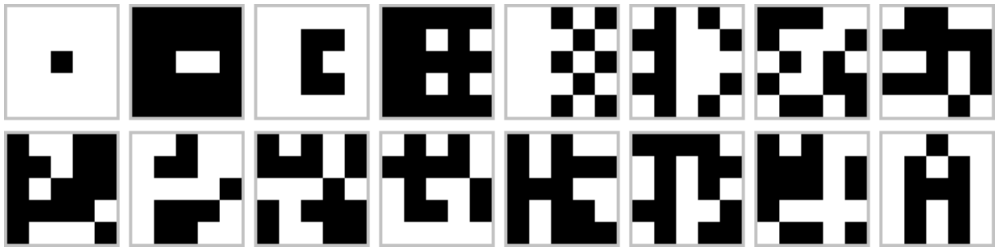
Bitmap (b)

00010100010000101110110110010110

Bitmap (c)

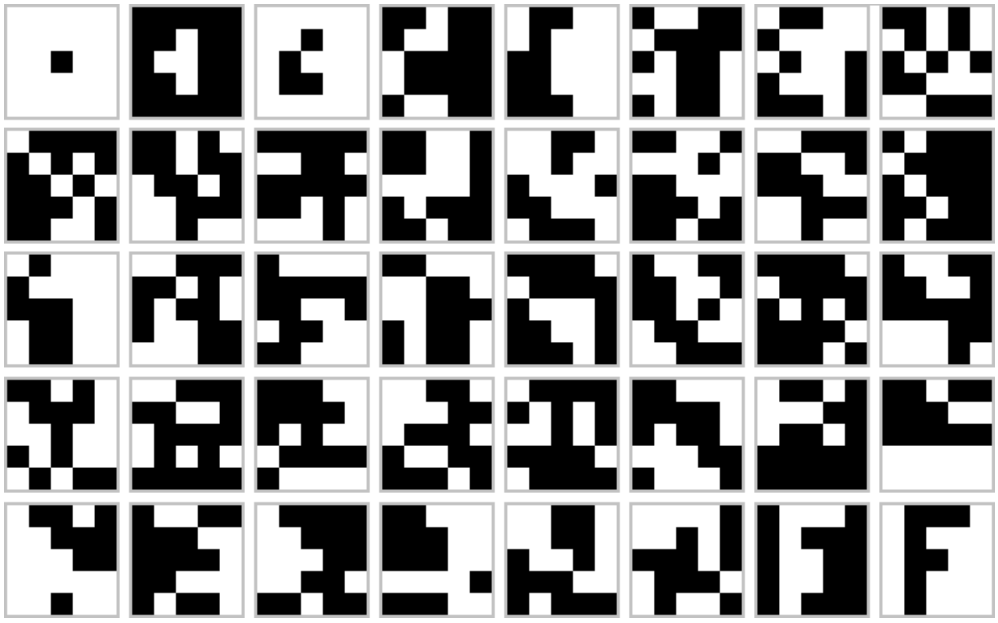
10001011000001011001100010100011

Bitmap (d)



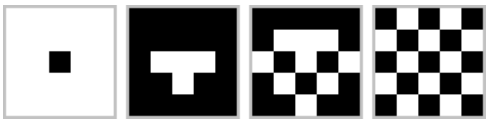
01010111001000011101110001001111

Bitmap (e)

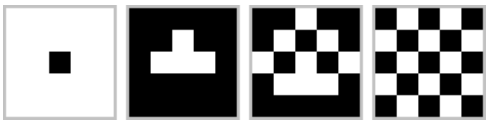


00001111011011100111111110100001

Bitmap (f)



10001011001011100101111001100011



10001111001010010100000000001001

01010001000101011001000011111011

00110001101101100111001101111101

00111111001011110000101110001110

10111111011011010110101100001110

10001110000010110000110110001111

10001111010010010001101100000111

5.3.4 Prova 2: configurazione iniziale variabile

Questa volta abbiamo invece deciso di togliere il vincolo della configurazione fissata per vedere se in questo modo riuscivamo ad ottenere un numero maggiore di regole per le configurazioni non simmetriche.

La configurazione iniziale di ogni automa viene generata casualmente al momento della creazione. Con l'avanzamento dell'algoritmo esso viene fatto evolvere, come accade per le regole de sviluppo. Quindi vengono applicati gli stessi meccanismi di mutazione ed ereditarietà utilizzati per loro. Per evitare che l'automa evolva in modo da "nascere" già con la griglia nella configurazione ottima, abbiamo deciso di limitare il numero di celle iniziali con lo stato 1 a n. Durante la mutazione ognuna di queste celle – con valore 1 – possono essere spostate in un'altra posizione della griglia con probabilità μ. Con la stessa probabilità può accadere che una di quelle celle venga portata a 0 oppure che ne nasca una nuova.

Durante la creazione della nuova generazione di individui, dobbiamo passare ai figli anche i punti iniziali dei genitori. Allora per generare ogni possibile combinazione di tratti derivanti da due genitori abbiamo deciso di far nascere, da ogni coppia, quattro figli. Vengono generati due figli con lo stesso metodo della prova precedente, poi essi vengono clonati e ad ogni coppia di figli con regole uguali, viene impostata al primo la configurazione iniziale del genitore 1 e al secondo quella del genitore 2. In questo modo copriamo tutti modi possibili di combinare il genotipo dei due genitori.

Osservando i risultati – in tabella 5.2 – ci accorgiamo sicuramente del fatto che il nuovo algoritmo è più efficace nel trovare le soluzioni ottime. Però in generale non abbiamo ottenuto un miglioramento nel far evolvere schemi non simmetrici – confrontare infatti i risultati nelle due tabelle utilizzando le bitmap 5.1c e 5.1b – ma solamente un generale miglioramento nella convergenza verso la soluzione.

Anche in questa prova abbiamo visualizzato tramite degli istogrammi – figure 5.6 e 5.7 – il numero delle regole trovate ed il valore medio di fitness al variare dell'indice di mutazione. Come detto per gli istogrammi precedenti i valori non sono molto significativi a causa del basso numero di prove effettuate. Infatti i valori in figura 5.6 non ci dicono molto riguardo a come scegliere un buon indice di mutazione. Se guardiamo invece come variano i valori medi di fitness notiamo che in quasi tutti i casi all'aumentare dell'indice di mutazione otteniamo un

peggioramento delle prestazioni. In questo caso la perdita di efficienza è più evidente a causa del fatto che l'indice di mutazione ha effetto non solo sulle regole dell'automa, ma anche sulla configurazione iniziale della griglia.

In figura 5.8 abbiamo invece riportato alcuni automi cellulari trovati, insieme le rispettive regole di sviluppo. Ancora una volta è interessante vedere come alcune regole molto simili producano un comportamento identico nell'automa cellulare. Ad esempio per lo sviluppo della bitmap (a) sono mostrate due regole che producono la stessa sequenza. Confrontando i bit vediamo che l'unica differenza tra le due regole è nel sesto bit, a partire da destra. Il valore di quel bit quindi, nell'insieme di regole di cui fa parte, non ha alcun effetto.

Tabella 5.2: Regole ottime trovate con configurazione iniziale variabile

Bitmap	**Regole trovate**
(a)	134
(b)	10
(c)	3
(d)	10
(e)	9
(f)	281

Figura 5.6: Regole trovate al variare della mutazione

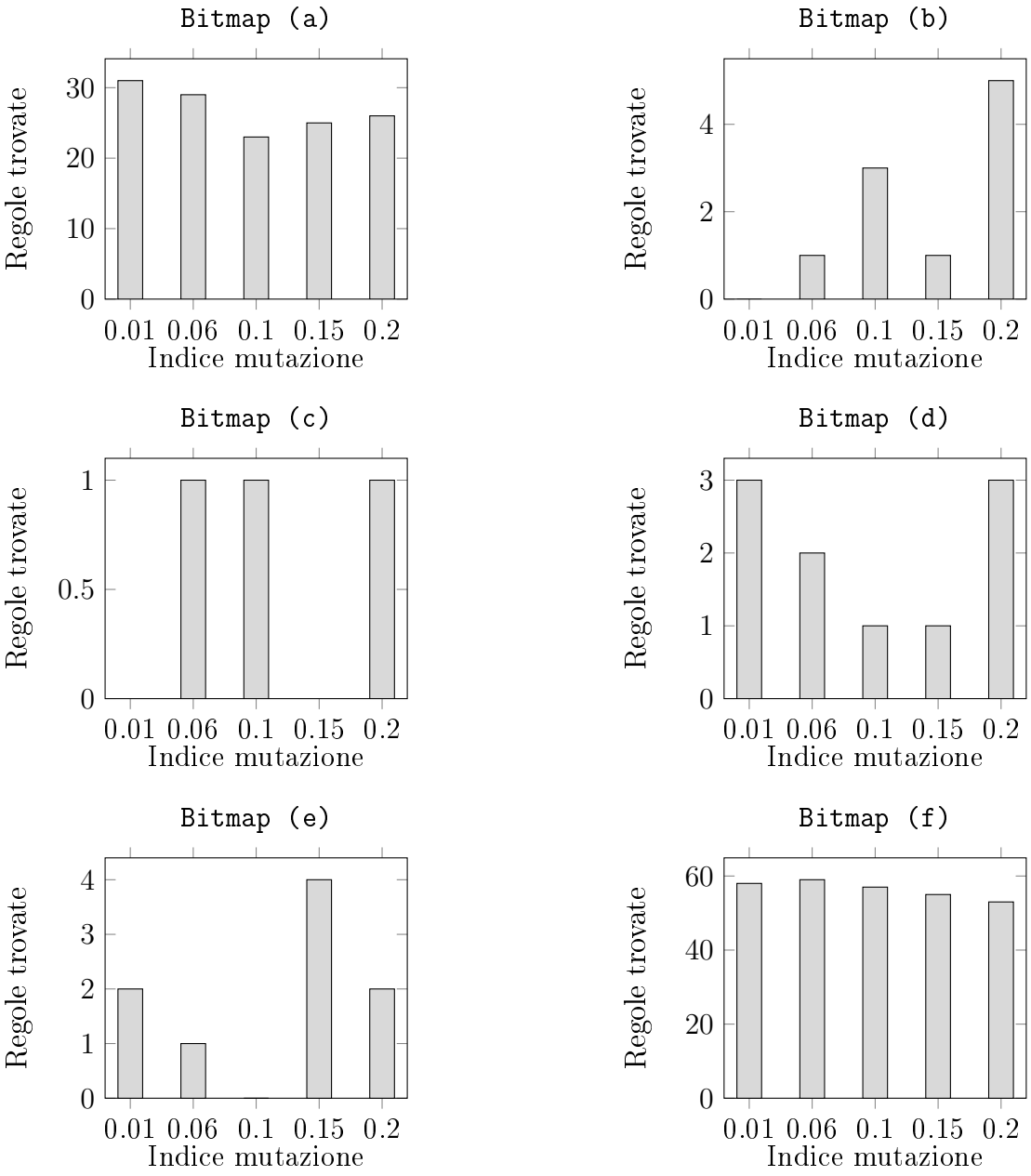


Figura 5.7: Fitness medio della popolazione al variare della mutazione

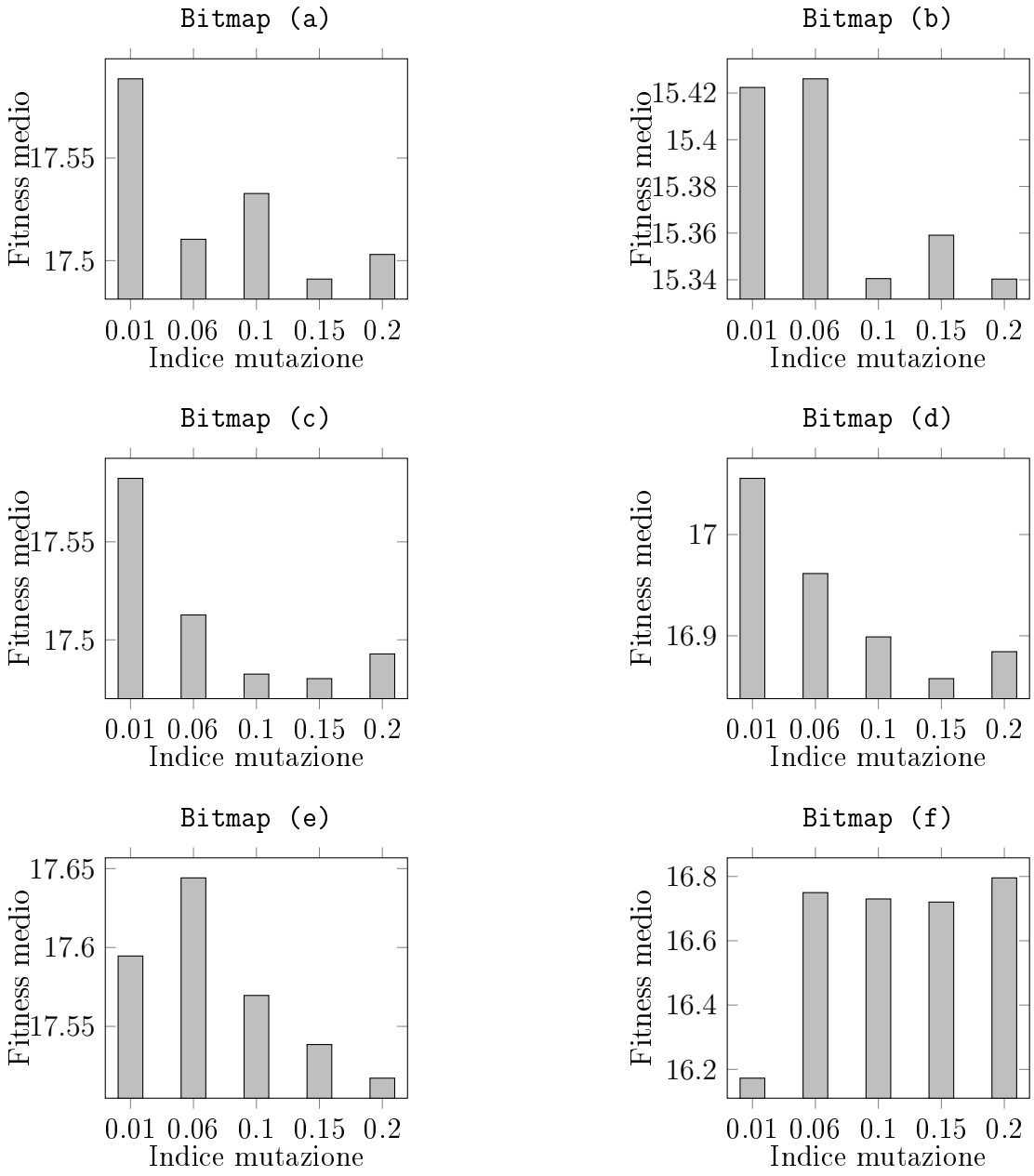
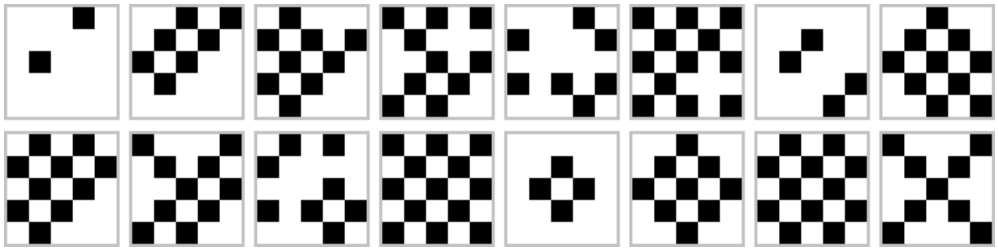


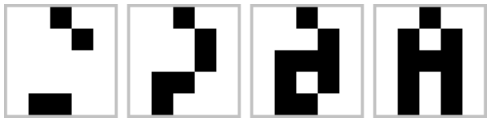
Figura 5.8: Sviluppo di alcune regole trovate

Bitmap (a)



11101001000101110001011100001110
11101001000101110001011100101110

Bitmap (d)



10110101011100111100010011111010

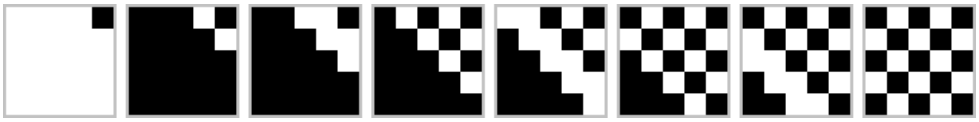
01000100001111010100101011110010

Bitmap (e)

10011001000101101110100000110111

10000000011010010010000111011010

Bitmap (f)



11110000111100101100011110011011

01111001000001011100011100001110

6 Progetto finale: `ImageEvolver`

Con il nostro progetto abbiamo voluto mettere in pratica quello finora visto teoricamente. In un certo senso il progetto è stato una sperimentazione visto che il campo dell'evoluzione degli automi cellulari, soprattutto utilizzando algoritmi genetici, sembra non essere ancora molto ben definito e consolidato. Inoltre delle sperimentazioni simili a quelle che stiamo per affrontare nell'esplorazione del nostro progetto sembra non siano mai state effettuate prima d'ora.

L'obbiettivo di questo progetto è quello di mostrare come sia possibile generare un immagine binaria scelta a piacere utilizzando un insieme di automi cellulari bidimensionali. A differenza degli esperimenti effettuati in precedenza, l'immagine che vogliamo evolvere può avere anche dimensioni molto grandi – superiore alle centinaia di pixel per lato. Risulterebbe pertanto impossibile far evolvere automi cellulari della grandezza dell'immagine, considerando l'enorme insieme delle possibili soluzioni tra le quali il programma dovrebbe cercare quella ottima.

Figura 6.1: Esempi di immagini obiettivo

È chiaro quindi che dobbiamo utilizzare un approccio differente da quello usato finora, un approccio che consenta di ridurre la complessità del problema.

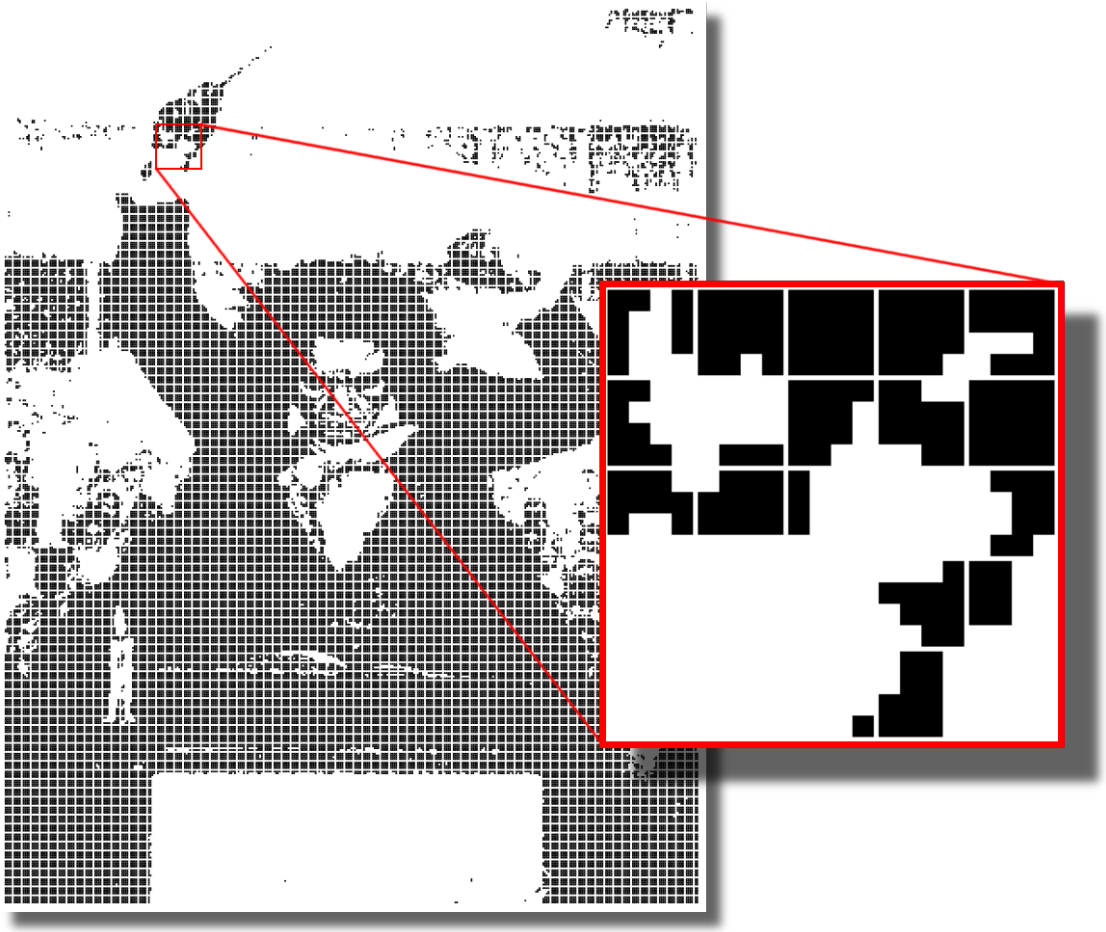


Figura 6.2: Immagine obiettivo suddivisa

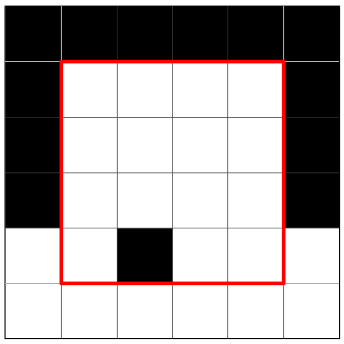
Suddividiamo l'immagine obiettivo in diverse immagini più piccole. Ora possiamo considerare ogni singola sotto-immagine separatamente e farla evolvere parallelamente alle altre. Ognuna di esse quindi avrà il suo set di regole ed una sua popolazione.

A livello concettuale possiamo pensare all'intera immagine come ad un sistema composto da tanti piccoli sotto-sistemi. Prendiamo, come esempio il sistema *corpo umano* composto dal cervello, del sistema immunitario, dal sistema nervoso e così via. Ognuno di questi sotto-sistemi agisce autonomamente, svolgendo i suoi compiti – il cervello elabora le informazioni, il sistema immunitario distrugge i batteri dannosi per l'organismo, ecc. – però se consideriamo l'intero sistema *corpo umano* essi, collaborando tra di loro, svolgono il compito globale di far sopravvivere

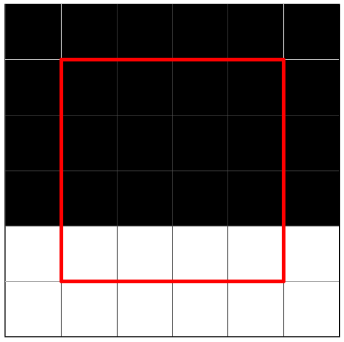
l'individuo.

Torniamo alla nostro sistema *immagine* ed immaginiamocelo come composto da tanti piccoli sotto-sistemi – immagini anche loro – i quali devono sì evolvere per trovare la soluzione al loro problema locale, ma hanno bisogno di comunicare per risolvere il problema di evolvere l'intera immagine. Ogni sotto-immagine quindi, utilizzando il vicinato di von Neumann, comunica con le sotto-immagini con cui condivide i bordi – a nord, sud, est e ovest. Ora gli automi cellulari non sono più toroidali, ma possono aggiornare il loro stato considerando lo stato degli automi che hanno affianco.

Figura 6.3



(a) Condizione iniziale



(b) Obiettivo

Consideriamo la figura 6.3a in cui l'automa evidenziato si trova nel suo stato iniziale, mentre i suoi vicini hanno già finito di evolvere. La figura 6.3b mostra quale deve essere la configurazione finale. Se l'automa centrale può utilizzare lo stato dei suoi vicini per svilupparsi, ad intuito diremmo che troverà molto velocemente la configurazione finale. Mentre se lo isoliamo dal resto ci vien da dire che impiegherebbe molto più tempo. Vedremo più avanti che effettivamente la possibilità di esplorare lo stato dei propri vicini ci porta ad un notevole aumento in termini di prestazioni del programma.

Abbiamo però bisogno di definire una qualche sorta di sincronizzazione affinché si riesca ad avere una convergenza verso la soluzione, dato che durante lo sviluppo degli automi cellulari lo stato della sua griglia cambia continuamente. Quindi le celle di bordo di ogni sotto-immagine, quando devono aggiornare il proprio stato, si troveranno a dover valutare stati di vicini che sono sempre diversi. Alla generazione successiva, una popolazione adattata all'"ambiente" definito dalla generazione precedente dei suoi vicini, potrebbe ritrovarsi in un ambiente nuovo in cui ciò che ha imparato finora non è più valido.

Perciò, per permettere un'evoluzione stabile, partiamo con una configurazione in cui ogni sotto-immagine è sola, ed evolve considerando che intorno a lui non c'è nessuno – quindi i suoi bordi sono circondati da bit settati a 0. Quando una

popolazione trova la soluzione desiderata per la sua porzione d'immagine, comunica ai suoi quattro vicini il fatto che possono utilizzare il suo stato per evolversi a loro volta. Si viene a creare quindi una sorta di gerarchia in cui le popolazioni che in un certo momento della loro evoluzione non sono riuscite a trovare la soluzione ottima, si appoggiano ai vicini che invece l'hanno già trovata.

Facciamo ora un passo indietro e torniamo al capitolo 1, nel quale abbiamo detto che un SAC generalmente può essere descritto da alcune proprietà – *aggregazione*, *non-linearità*, *flusso*, *diversità* – e meccanismi – *etichettatura*, *modello interno*, *elementi costitutivi*. Nel paragrafo 4.2 abbiamo visto come queste peculiarità dei SAC potessero essere riscontrate anche negli automi cellulari, una volta inserito un meccanismo che permette l'adattamento – algoritmo genetico. Però mancavano ancora alcuni pezzi affinché potessimo affermare che tali sistemi mostrassero effettivamente adattamento. Mancavano alcune parti che adesso, considerando un sistema composto da altri sistemi, siamo riusciti a ricollegare.

Dinamicità dei flussi I collegamenti tra una popolazione e l'altra non sono fissati, ma variano nel tempo. Nello specifico all'inizio le popolazioni sono totalmente isolate, ma con l'avanzare delle generazioni sono loro stesse che si mettono in comunicazione con le popolazioni vicine.

Suddivisione del modello interno Il modello interno dell'intera immagine è dato dal totale delle regole locali trovate da ogni singola popolazione. Quindi esso è in realtà composto da tante parti, ognuna dedicata al suo specifico compito locale, le quali possono essere riutilizzate ove necessario. Infatti, nel caso in cui ci siano due o più sotto-immagini uguali possiamo evitare di far evolvere più popolazioni verso lo stesso obiettivo. Ne evolviamo solamente una e riutilizziamo il suo risultato per le parti uguali.

Ora possiamo finalmente dire che il nostro sistema è un *sistema adattativo complesso*. Esso è composto da un insieme di elementi – popolazioni di automi cellulari, che a loro volta sono formate da un vasto numero di automi cellulari – i quali hanno un obiettivo locale e collaborando, sincronizzandosi e auto-organizzandosi fanno emergere un comportamento globale che è la formazione dell'immagine binaria scelta.

6.1 Ambiente di lavoro

6.1.1 Java e programmazione ad oggetti

Il linguaggio di programmazione usato è stato **Java** [17]. Java è un linguaggio di programmazione orientato agli oggetti [18] e specificatamente progettato per essere il più possibile indipendente dalla piattaforma di esecuzione.

La scelta è caduta su questo linguaggio perché sfrutta appieno le potenzialità della *Programmazione ad Oggetti*(PO). Nel contesto del nostro progetto l'utilizzo della *PO* si è resa fondamentale in quanto ci ha permesso di modellare in modo naturale le componenti del programma. Infatti grazie alla possibilità di definire *oggetti software* e ai meccanismi di *incapsulamento*, *ereditarietà* e *polimorfismo* i linguaggi ad oggetti permettono di definire classi che rappresentano oggetti reali e che interagiscono tra di loro tramite scambio di messaggi. Quando inoltre bisogna lavorare su progetti di una grandezza consistente la *PO* ci dà un grande supporto, sia perché permette di tenere pulito ed organizzato il codice, sia perché limita la possibilità di fare errori.

6.1.2 Processing

Data la natura del progetto era molto importante per noi la possibilità di mostrare, tramite un interfaccia grafica, l'avanzamento del programma. A tal scopo ci siamo serviti della libreria **Processing** [15], basata interamente su **Java** e sviluppata per applicazioni come giochi, animazioni e contenuti interattivi.

La libreria permette di creare velocemente degli "schizzi" di un'applicazione semplicemente estendendo la classe `PApplet` ed implementando i metodi `setup()` e `draw()`. Il primo metodo viene chiamato, una sola volta, all'avvio dell'applicazione e contiene le istruzioni necessarie per preparare le strutture di cui il programma ha bisogno durante l'esecuzione. Mentre il metodo `draw()` è la funzione di aggiornamento dell'interfaccia grafica che deve perciò contenere tutte le istruzioni per modificare l'interfaccia a seconda dello stato interno dell'applicazione.

La libreria inoltre fornisce diverse primitive per la modellazione geometrica di oggetti 2D e 3D.

6.2 Formato PBM

Il formato scelto per memorizzare le immagini binarie necessari all'applicazione è stato il PBM (Portable Bit Map) [16]. È stato creato con lo scopo di facilitare lo sviluppo di applicazioni che lavorano con immagini, senza badare all'efficienza. Il formato è stato sviluppato all'interno del progetto NETPBM insieme ai formati PGM (Portable Gray Map) e PPM (Portable Pix Map), i quali rispettivamente permettono di memorizzare immagini grayscale ed RGB.

6.2.1 Descrizione formato

Ogni file inizia con due byte – codificati in ASCII – che identificano il tipo di file e la codifica.

Tabella 6.1: Formati Netpbm

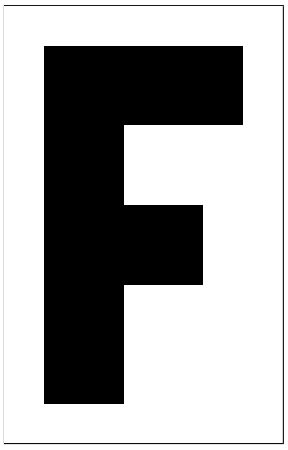
Tipo	**Formato**	**Codifica**	**Colori**
PBM	P1	ASCII	$0-1$ (bianco e nero)
PGM	P2	ASCII	$0-255$ (scala di grigi)
PPM	P3	ASCII	$0-255$ (RGB)
PBM	P4	binaria	$0-1$ (bianco e nero)
PGM	P5	binaria	$0-255$ (scala di grigi)
PPM	P6	binaria	$0-255$ (RGB)

Utilizzando il formato ASCII il file è leggibile dall'uomo e facile da trasferire tra le diverse piattaforme. Il formato binario invece permette maggiore efficienza in termini di dimensioni dei file e nella decodifica del file.

Un file pbm contiene i due byte del formato nella prima riga, la lunghezza e l'altezza dell'immagine in termini di pixel nella seconda e poi, dalla terza in poi, abbiamo tutti i valori dei pixel dell'immagine – figura 6.4.

Figura 6.4: Esempio di file PBM

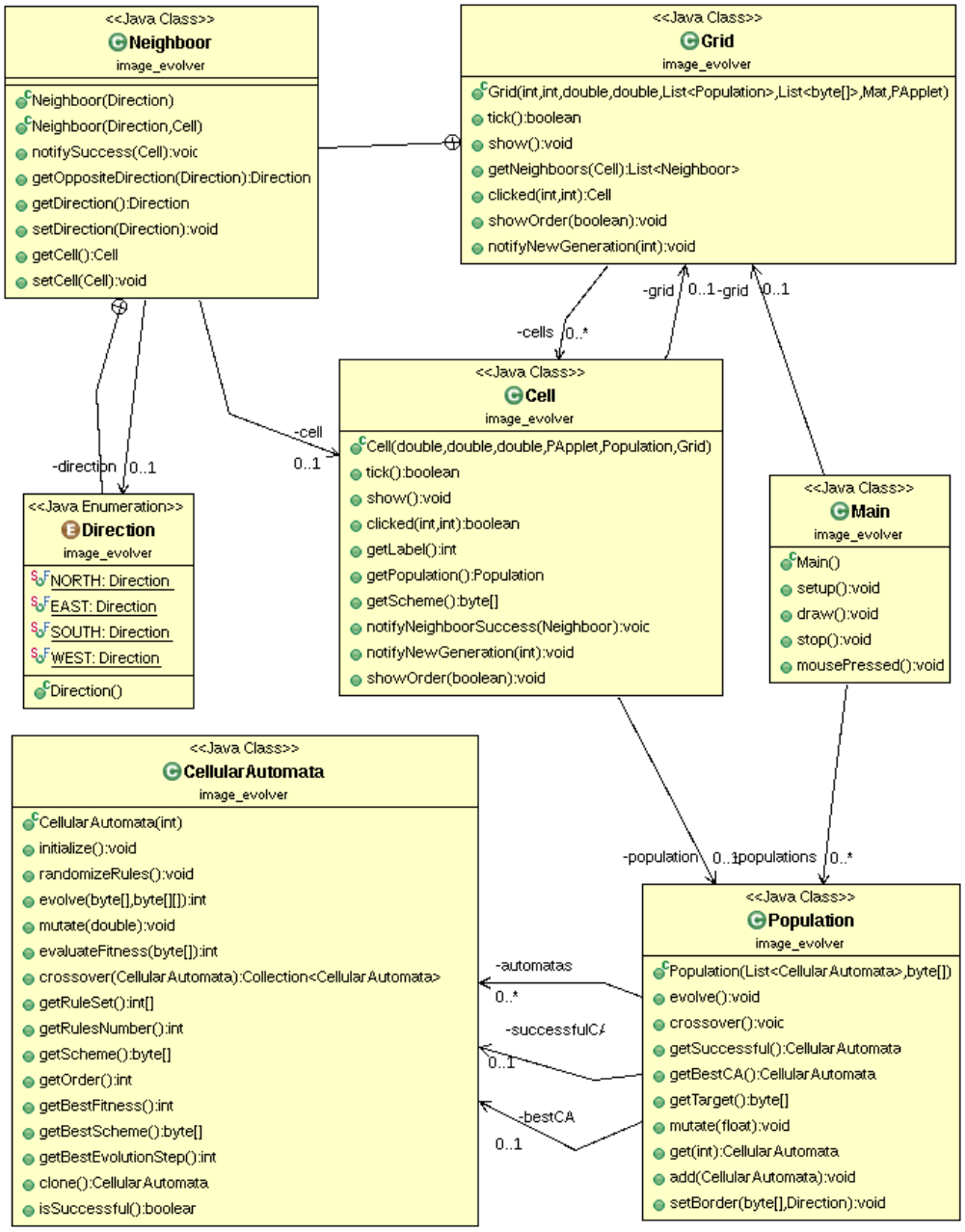
```
P1
# Bitmap esempio
7 11
0 0 0 0 0 0 0
0 1 1 1 1 1 0
0 1 1 1 1 1 0
0 1 1 0 0 0 0
0 1 1 0 0 0 0
0 1 1 1 1 0 0
0 1 1 1 1 0 0
0 1 1 0 0 0 0
0 1 1 0 0 0 0
0 1 1 0 0 0 0
0 0 0 0 0 0 0
```



6.3 Struttura del codice

Il codice è stato strutturato, grazie alla programmazione ad oggetti, in modo da rispettare il più fedelmente possibile la semantica dei componenti.

Per prima cosa, iniziamo ad esaminare le classi principali dell'applicazione `ImageEvolver`.



Main

È la classe principale dell'applicazione che gestisce l'avanzamento dell'evoluzione dell'immagine e mostra a all'utente l'avanzamento del lavoro.

Estende la classe `PApplet` della libreria Processing quindi implementa i metodi `setup()` e `draw()`. Nella funzione `setup()`, che viene richiamata una volta all'avvio, prepara tutte le strutture necessarie per far evolvere l'immagine. In `draw()`, che viene richiamata continuamente durante l'esecuzione del programma, ci sono le chiamate alle funzioni che aggiornano lo stato del sistema e che modificano l'interfaccia grafica.

Le funzioni `stop()` e `mousePressed()` sono anch'esse ereditate dalla classe `PApplet()` e rispettivamente permettono effettuare operazioni appena prima della chiusura del programma e di gestire a pressione del mouse sull'interfaccia grafica.

CellularAutomata

Questa classe rappresenta l'automa cellulare, quindi in un certo senso è il cuore della nostra applicazione.

Il costruttore `CellularAutomata(int)` crea un nuovo automa cellulare bidimensionale permettendo di specificare la grandezza del lato della griglia. Alla creazione l'automa ha un insieme di regole generate casualmente e lo schema iniziale è una griglia piena di zero con un uno al centro.

Il metodo `evolve(byte[], byte[][])` permette di sviluppare l'automa, facendo evolvere ogni casella della griglia seguendo l'insieme di regole e lo stato dei vicini. Il primo parametro richiesto è lo schema obiettivo – rappresentato come un vettore di `byte` – relativo alla popolazione in cui l'automa è posizionato. È necessaria alla funzione perché essa internamente richiama `evaluateFitness(byte[])` che calcola il valore di *fitness* dell'automa – il valore va da 0 al numero di caselle della griglia, ed è la somma del numero di caselle dell'automa che sono uguali a quelle dell'obiettivo. Il secondo parametro è una matrice le cui righe rappresentano i quattro bordi esterni adiacenti l'automa. I bordi che riceverà saranno nulli all'inizio, mentre man mano che le popolazioni vicine trovano la soluzione, assumeranno il valore dei bordi corrispondenti di quest'ultime.

Il metodo `mutate(double)` effettua una mutazione sul vettore delle regole. Il metodo scorre il vettore e per ogni casella inverte il valore presente con con la probabilità passata per parametro.

Il metodo `crossover(CellularAutomata)` permette di generare degli automi cellulari figli, a partire da due automi genitori, il primo è quello su cui viene chiamato il metodo mentre il secondo è l'automa passato per parametro. Il metodo restituisce una `Collection<CellularAutomata>` che contiene due figli generati. I figli sono generati mischiando i set di regole dei due genitori, utilizzando il metodo descritto nella sezione 3.4.3. Le regole dei due figli sono speculari, nel senso che presa una qualunque regola di un figlio, se il figlio l'ha ereditata dal primo genitore allora suo fratello l'ha ereditato dal secondo genitore e viceversa.

Population

La classe `Population` rappresenta una popolazione di automi cellulari e gestisce la loro evoluzione.

Il suo costruttore `Population(List<CellularAutomata>, byte[])` prende in input una lista di `CellularAutomata`, che rappresentano i componenti della popolazione, e lo schema obiettivo che guida l'evoluzione della popolazione.

Offre il metodo `evolve()` per far sviluppare gli schemi degli automi – quindi a sua volta richiama l'omonimo metodo su tutti gli automi cellulari della popolazione. Ed analogamente sono presenti i metodi `crossover()` e `mutate()` che invece attuano i corrispondenti meccanismi dell'algoritmo genetico.

La popolazione inoltre tiene traccia in ogni momento dell'automa migliore presente nella popolazione e permette di recuperarlo col metodo `getBestCA()`. Il metodo `getSuccessful()` invece restituisce l'automa che ha trovato la soluzione ottima se al momento della chiamata del metodo esiste, `null` altrimenti.

Grid

La classe `Grid` rappresenta l'immagine obiettivo suddivisa in tante celle. Essa infatti è composta da un gran numero di `Cell` una per ogni sotto-immagine dell'obiettivo. Nel costruttore prende la lista delle popolazioni `List<Population>` e la matrice `Mat` contenente l'immagine da trovare. Quindi per ogni `Population` crea una `Cell` e la assegna ad un pezzo del `Mat`.

Inoltre fornisce un metodo di supporto per le `Cell` permettendogli di scoprire quali sono le sue `Cell` vicine.

Cell

La classe `Cell`, come già detto poco fa, rappresenta una sotto-immagine della immagine obiettivo. Contiene al suo interno la popolazione che fa evolvere la sua immagine. Quando la sua popolazione ha trovato la soluzione ottima, tiene traccia del numero della generazione a cui è avvenuta la scoperta e notifica i vicini – permettendogli di aggiornare i bordi dei loro automi cellulari.

Neighboor

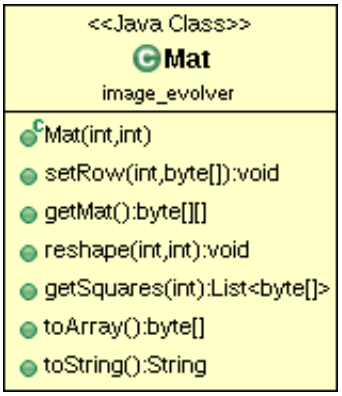
Rappresenta il vicino di una `Cell`. Contiene infatti una `Cell` e la relativa `Direction`. Quando una cella, la cui popolazione ha trovato la soluzione, notifica i vicini chiama il loro metodo `notifyNeighboorSuccess(Neighboor)` passandogli un `Neighboor` che contiene se stessa e la direzione in cui si trova rispetto alla cella che sta notificando.

ImageReader

Legge in input immagini con estensione `.pbm` – guardare nella sezione 6.2 per maggiori dettagli sui file PBM – e li salva in una matrice `Mat`. Il costruttore prende in input una stringa che indica il percorso del file.

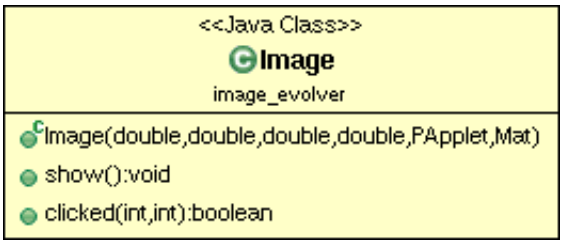
Mat

Rappresenta un matrice bidimensionale di `byte`. Ci permette di modificare le sue dimensioni con il metodo `reshape(int, int)` e di suddividere la matrice in quadrati più piccoli, grazie al metodo `getSqaures()` che ci ritorna una `List<byte[]>`. Quest'ultimo metodo ci risulta molto utile per estrapolare le sotto-immagini obiettivo dall'intera immagine.



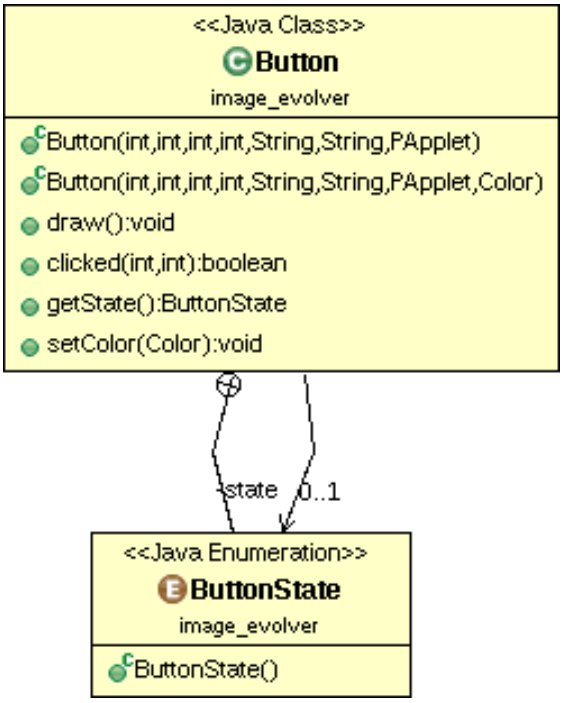
Image

È una classe che rappresenta un immagine da mostrare sull'interfaccia grafica. Il suo costuttore, oltre ai campi relativi alla visualizzazione – come dimensione e posizione –, prende un `Mat` che appunto contiene il valore dei pixel dell'immagine da mostrare.



Button

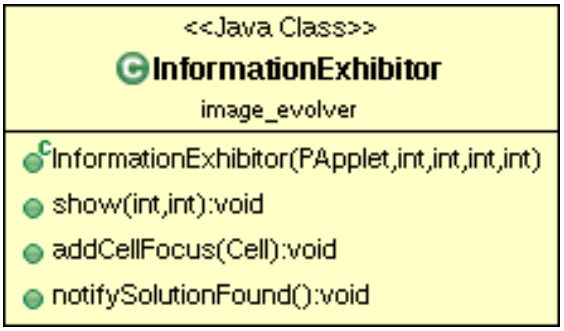
La classe `Button` rappresenta un pulsante da mostrare sull'interfaccia grafica. È stato necessario crearla perché la libreria `Processing` non fornisce oggetti visuali per il controllo nelle interfacce – come ad esempio pulsanti, slider e così via. Ogni pulsante può essere in due stati: `on` oppure `off`. Il metodo `getState()` ci permette di recuperare lo stato passandoci un `ButtonState`. Quest'ultimo è un `enum`, quindi a differenza delle classi non ha metodi né ha costruttori, ma permette di accedere ad una lista di proprietà – in questo caso `ON` e `OFF`. La funzione clicked ci ritorna un `boolean` che è vero se le coordinate passate per parametro sono comprese nello spazio da lui occupato nell'interfaccia.



InformationExhibitor

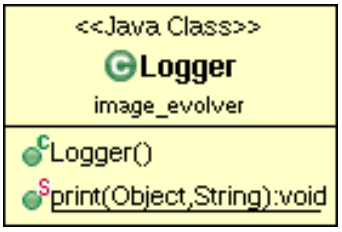
Questa classe rappresenta un pannello visibile sullo schermo che ci mostra informazioni circa lo stato dell'evoluzione delle popolazioni.

La funzione `addCellFocus(Cell)` permette di mostrare le informazioni relative ad una specifica `Cell`. Il metodo `notifySolutionFound()` invece lo utilizziamo per notificargli il fatto che l'algoritmo genetico ha terminato l'evoluzione.



Logger

Questa classe è stata implementata al solo scopo di migliorare la leggibilità dei messaggi di debug nel codice. Permette di stampare velocemente messaggi sulla console di output semplicemente chiamando il metodo statico `print(Object, Sring)`. Il primo parametro deve essere la classe dell'istanza che vuole stampare il messaggio, il secondo è il messaggio. La classe fornisce al `Logger` informazioni riguardo a chi vuole stampare in modo da poter aggiungere in testa al messaggio un suo identificativo. In questo modo la console di output risulta più ordinata e leggibile.



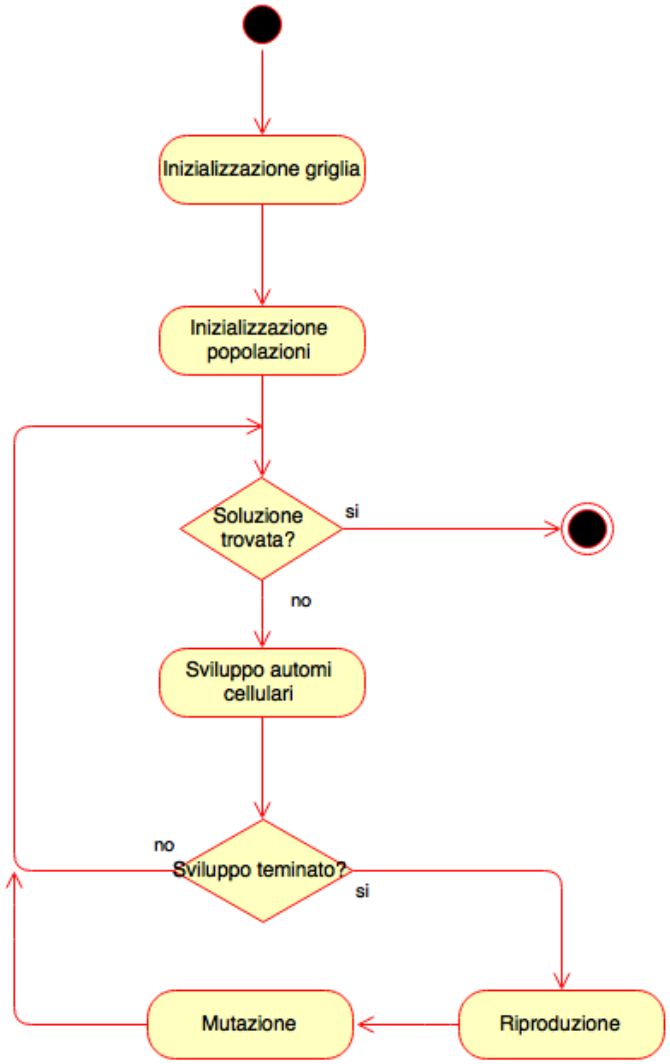
6.4 Implementazione

Precendemente abbiamo visto quali sono i componenti che compongono il nostro sistema. In questa sezione ci occuperemo di analizzare in che modo le varie entità agiscono e comunicano durante l'esecuzione del programma.

6.4.1 Sequenza delle attività

Il diagramma in figura 6.5 ci mostra la sequenza di esecuzione delle principali macro-attività.

Figura 6.5: Diagramma delle attività



Il programma è composto da una prima fase inizializzazione implementato nel metodo `setup()` della classe `Main`.

L'esecuzione vera e propria dell'algoritmo genetico è gestita da un ciclo implementato con nel metodo `draw()` del `Main` che esegue consecutivamente e dà il via alle operazioni per l'avanzamento del programma e per aggiornare l'interfaccia.

Il ciclo principale è quindi composto così:

ImageEvolver.Main

```
public void draw() {
  /* Soluzione trovata? */
  if (this.grid.tick()) {
    this.exhibitor.notifySolutionFound();
  } else {
    /* Sviluppo automi */
    for (Population p: this.populations) {
      p.evolve();
    }
    this.evolutionCount ++;
    /* Sviluppo teminato? */
    if (this.evolutionCount == this.maxEvolution) {
      /* Riproduzione */
      this.populations
        .parallelStream()
        .forEach(p -> p.crossover());
      /* Mutazione */
      this.populations
        .parallelStream()
        .forEach(p -> p.mutate(this.mutationRate));
      this.evolutionCount = 0;
      this.grid.notifyNewGeneration(++this.generationCount);
    }
  }
}
```

Il controllo relativo alla soluzione trovata è affidata alla `Grid` perché essa, es-

sendo l'entità che si occupa di gestire le celle di cui è composta l'immagine e le relative popolazioni, ad ogni chiamata della sua funzione di aggiornamento può controllare lo stato dell'evoluzione dell'immagine.

6.4.2 Implementazione delle attività

Scendiamo ora ad esaminare in dettaglio come sono state implementate le varie attività.

Inizializzazione griglia

Questa attività si occupa di leggere l'immagine in input e di preparare la griglia composta dalle varie sotto-immagini.

ImageEvolver.Main

```
1  /* Lettura immagine input */
2  ImageReader imageReader = new ImageReader("beethoven.pbm");
3  Mat mat = imageReader.readImage();
4
5  /* Estrazione delle sotto-immagini */
6  List<byte[]> targets = mat.getSquares(this.n);
7
8  /* Creazione della griglia */
9  Grid grid = new Grid(positionX, positionY, maxWidth, maxHeight,
10       this.populations, targets, mat, this);
```

`ImageReader` legge l'immagine `.pbm` di input specificato dalla stringa passata per parametro. Successivamente salviamo la matrice con i valori dei pixel dell'immagine in un `Mat`.

Fatto ciò estraiamo la lista di sotto-immagini obiettivo. Il metodo `getSquares (int)` divide l'immagine generando tale lista. Se le dimensioni dell'immagine non sono divisibili per le dimensioni specificate per le sotto-immagini, essa la ridimensiona aggiungendo pixel bianchi.

La griglia viene creata a partire dalla lista di sotto-immagini. La classe `Grid` incapsula sia la gestione logica delle interazioni tra le popolazioni che la gestione della visualizzazione delle immagini.

La lista di popolazioni passata alla griglia è vuota perché è la griglia stessa che si occupa di riempirla assegnando ad ogni popolazione la sua sotto-immagine obiettivo. L'inizializzazione delle popolazioni è stata affidata alla griglia perché essa può evitare di creare popolazioni duplicate, che cioè hanno la stessa immagine da evolvere. Infatti, crea parallelamente alle popolazioni le celle corrispondenti e quando si trova a dover creare una cella per un immagine obiettivo già assegnata, riusa la popolazione corrispondente.

Inizializzazione popolazioni

In questa fase riempiamo ogni popolazione con degli automi cellulari appena creati.

`ImageEvolver.Main`

```
/* Creazione delle popolazioni */
for (Population pop: this.populations) {
  for (int i=0; i<this.populationSize; i++) {
    pop.add(new CellularAutomata(this.n));
  }
}
```

Ogni `CellularAutomata` viene creato con regole inizializzate casualmente e con lo schema iniziale fissato – figura 5.2. Il tipo di vicinato utilizzato è quello di von Neumann – figura 2.1a.

`ImageEvolver.CellularAutomata`

```
/* Inizializzazione schema  */
public void initialize() {
  Arrays.fill(this.scheme, (byte)0);
  this.scheme[this.scheme.length / 2] = 1;
}
/* Inizializzazione regole random */
public void randomizeRules() {
  for (int i=0; i<this.ruleSet.length; i++) {
    this.ruleSet[i] = (int) Math.round(Math.random());
  }
}
```

Sviluppo automi cellulari

Lo sviluppo viene effettuato parallelamente per tutti i `CellularAutomata` aggiornando lo stato del loro schema. Anche l'aggiornamento dello stato dello schema viene effettuato parallelamente per ogni cella di uno schema. Chiaramente il parallelismo è simulato perché altrimenti avremmo bisogno di molte migliaia di core, uno per ogni cella di ogni automa.

`ImageEvolver.CellularAutomata`

```
/* Trova il valore della cella nel nuovo schema */
private byte evolvePoint(int index, byte[][] borders) {
  int result;
  String configuration;
  int u, d, l, r, c = this.scheme[index];
  if (index < this.n)
    u = borders[0][index + this.n * (this.n - 1)];
  else u = this.scheme[index-this.n];
  if (index%this.n == this.n-1)
    r = borders[1][index - this.n + 1];
  else r = this.scheme[index+1];
  if (index >= this.n*(this.n-1))
    d = borders[2][index - this.n * (this.n - 1)];
  else d = this.scheme[index+this.n];
  if (index%this.n == 0)
    l = borders[3][index + (this.n-1)];
  else l = this.scheme[index-1];
  configuration = "" + u + l + c + r + d;
  result = Integer.parseInt(configuration, 2);
  return (byte)this.ruleSet[result];
}
```

Quella riportata è la funzione che fa evolvere il valore di ogni cella dello schema in base al valore del suo vicinato, che ricordiamo è quello di von Neumann. Nel caso che la cella da evolvere si trova sul bordo utilizza il vettore ricevuto come parametro `borders` che contiene il valore dei bordi dei suoi vicini, nel caso che questi abbiamo trovato le regole ottime.

Appena finito di sviluppare tutte le sue celle l'automa calcola il valore di fitness della nuova configurazione trovata.

Riproduzione

ImageEvolver.Population

```
public void crossover() {
  /* Ordinamento degli automi in base al valore di fitness */
  ...
  /* Salvataggio dei migliori */
  List<CellularAutomata> tempPopulation =
        new ArrayList<CellularAutomata>(this.automatas
        .subList(0, this.nSurvivors));
  /* Sostituzione degli automi con basso fitness */
  if (this.bestCA.getRulesNumber() == tempPopulation.get(0)
        .getRulesNumber()) {
    if (this.nSameBestCA++ == 15) {
      double meanFitness = this.automatas.stream()
          .mapToInt(CellularAutomata::getBestFitness)
          .average().getAsDouble();
      this.automatas.stream()
        .filter(ca -> ca.getBestFitness() < meanFitness)
        .forEach(ca -> ca.randomizeRules());
      this.nSameBestCA = 0;
    }
  } else this.nSameBestCA = 0;
  this.bestCA = tempPopulation.get(0);
  /* Creazione dei figli */
  while (tempPopulation.size() < this.automatas.size()) {
    tempPopulation.addAll(this.automatas.get(
          this.getRandomCAIndex()).crossover(
          this.automatas.get(this.getRandomCAIndex())));
  }
  this.automatas = tempPopulation;
}
```

La funzione che attua la generazione della nuova popolazione si compone di diversi passi. Per prima cosa vengono ordinati gli automi in base al loro fitness. Questo ci permette di salvare un certo numero di automi – 10 – che sopravvivono alla nuova generazione. Successivamente applichiamo un metodo che ci permette di introdurre degli elementi nuovi nella popolazione quando questa rimane bloccata per un certo numero di generazioni – 15 – con lo stesso automa migliore.

Successivamente passiamo alla creazione degli automi figli dalla popolazione precedente. La funzione `getRandomCAIndex()` ritorna l'indice di un elemento della popolazione in modo che la probabilità di ottenere un indice cresca linearmente in base al fitness dell'automa corrispondente. Una volta selezionati due automi da far "accoppiare" generiamo i figli.

`ImageEvolver.CellularAutomata`

```
public Collection<CellularAutomata> crossover(
        CellularAutomata partner) {
  Collection<CellularAutomata> prole = new ArrayList<>();
  CellularAutomata firstChild = new CellularAutomata(this.n);
  CellularAutomata secondChild = new CellularAutomata(this.n);
  /* Ricombinazione regole */
  int midPoint = (int) Math.random() * this.ruleSet.length;
  for (int i=0; i<this.ruleSet.length; i++) {
    if (i < midPoint) {
      firstChild.getRuleSet()[i] = this.ruleSet[i];
      secondChild.getRuleSet()[i] = partner.getRuleSet()[i];
    } else {
      secondChild.getRuleSet()[i] = this.ruleSet[i];
      firstChild.getRuleSet()[i] = partner.getRuleSet()[i];
    }
  }
  prole.add(firstChild);
  prole.add(secondChild);
  prole.stream().forEach(ca -> ca.initialize());
  return prole;
}
```

Vengono creati due figli con regole speculari tra di loro. La ricombinazione delle regole viene fatta scegliendo un indice intermedio e combinando le regole dei due genitori spezzandole secondo l'indice trovato.

Mutazione

La mutazione viene effettuata scambiando il valore di ogni singola regola dell'automa con una certa probabilità. Nella nostra applicazione è stato scelto un indice di mutazione $\mu = 0.05$.

`ImageEvolver.CellularAutomata`

```
public void mutate(double rate) {
  for (int i=0; i<this.ruleSet.length; i++) {
    if (Math.random() < rate) {
      this.ruleSet[i] = 1 - this.ruleSet[i];
    }
  }
}
```

6.4.3 Operazioni aggregate

In questo progetto ci siamo serviti di una nuova funzionalità della JDK 8 che ci permette di semplificare il codice e di renderlo più leggibile: le operazioni aggregate [12]. Le operazioni aggregate si possono utilizzare aprendo un *stream* su una collezione, un array, una funzione generatrice o un canale di input/output. A differenza degli iteratori non dobbiamo specificare come iterare gli elementi, ma la JDK determina automaticamente come farlo permettendo di sfruttare anche il *parallelismo* – sezione 6.4.4. Una volta aperto uno stream possiamo specificare una serie di operazioni aggregate che vogliamo vengano eseguite su di esso. Questa serie di operazioni è chiamata *pipeline*, perché le operazioni vengono eseguite in ordine di come sono scritte.

```
/* Conta il numero di popolazioni che  hanno trovato
 * la soluzione ottima, con operazioni aggregate
 */
this.populations.stream()
      .map(Population::getSuccessful)
      .filter(ca -> ca != null).count();
```

Nell'esempio lo stream sulle popolazioni viene mappato in uno stream contente gli automi ottimi per ogni popolazione, successivamente vengono filtrati via gli automi che non hanno ancora trovato la soluzione ed in fine vengono contati quanti elementi sono rimasti nello stream.

Nell'operazione di filtraggio abbiamo utilizzato una *lambda expression*, un'altra funzionalità nuova della JDK 8.

Senza l'utilizzo dello stream avremmo dovuto scrivere qualcosa del genere:

```
/* Conta il numero di popolazioni che hanno trovato
 * la soluzione ottima, con iteratore
 */
int count = 0;
for (Population p: this.populations) {
  if (p.getSuccessful != null) {
    count++;
  }
}
```

Lambda expression

Le lambda expression [13] sono delle funzioni anonime che non sono legate ad una classe. Possono essere usate per contenere funzionalità che non hanno bisogno di essere definite su uno specifico oggetto del sistema e che generalmente hanno un utilizzo circoscritto. Il loro utilizzo all'interno del codice non è essenziale dato che in ogni caso possono essere sostituite con delle classiche funzioni che compiono le stesse operazioni, però sono più pratiche ed eleganti nel caso in cui bisogna incapsulare un comportamento specifico e non riusabile.

6.4.4 Calcolo parallelo

Il calcolo parallelo è un meccanismo che consiste nell'eseguire simultaneamente il codice sorgente di un programma su più microprocessori o più core. Questo metodo permette di aumentare notevolmente le prestazioni del sistema di elaborazione.

Scrivere un programma parallelo – cioè che utilizza più core contemporaneamente per eseguire le proprie operazioni – è generalmente più difficile che scriverne uno sequenziale, a causa della *concorrenza* che si viene a formare tra i diversi flussi del processo in esecuzione. Le *corse critiche* sono uno dei tanti problemi che possono verificarsi e che vengono risolti tramite la **comunicazione** e la **sincronizzazione** tra i task del processo. Per scrivere un programma parallelo efficiente bisogna sfruttare adeguatamente questi due meccanismi, tenendo conto delle operazioni da eseguire e dalle disponibilità del sistema, altrimenti si può avere anche un peggioramento delle prestazioni – rispetto allo stesso programma scritto sequenzialmente.

Con la nuova versione della libreria Java – la JDK 8 – abbiamo la possibilità di sfruttare le potenzialità del calcolo parallelo in modo trasparente grazie alle *operazioni aggregate*. Abbiamo visto che le operazione aggregate possono essere attivate chiamando il metodo `stream()`, e successivamente specificare la pipeline di operazioni da eseguire. Queste operazioni però vengono eseguite sequenzialmente su ogni oggetto dello stream. Se vogliamo, invece, che le operazioni vengano eseguite in modo parallelo dobbiamo richiedere invece un `parallelStream` e su di lui specificare le operazioni da eseguire. Possiamo ad esempio specificare un'operazione da eseguire su ogni elemento dello stream usando l'operatore `forEach`, il quale prende come parametro una `lambda expression`.

Nel nostro caso, data la natura prettamente parallela degli algoritmi genetici e degli automi cellulari, la scelta se usare o meno i meccanismi di calcolo parallelo è stata immediata. Ricordiamo che abbiamo una vasto numero di popolazioni di automi cellulari i quali evolvono "parallelamente". In oltre, scendendo ad un livello più basso e considerando una singola popolazione, sappiamo che anche gli automi cellulari si sviluppano in modo "parallelo" ed indipendente gli uni dagli altri.

Ogni popolazione, quando viene richiamata il suo metodo `evolve()`, apre uno stream parallelo della sua popolazione di automi e chiama a sua volta il metodo per sviluppare gli automi.

ImageEvolver.Population

```
/* Sviluppo automi cellulari */
public void evolve() {
  if (this.keepLiving()) {
    this.automatas
      .parallelStream()
      .forEach(ca -> ca.evolve(this.target, this.borders));
  }
}
```

Per ogni passo dell'algoritmo genetico sfruttiamo il `parallelStream` per effettuare la riproduzione e la mutazione su tutte le popolazioni.

ImageEvolver.Main

```
/* Riproduzione */
this.populations
  .parallelStream()
  .forEach(p -> p.crossover());

/* Mutazione */
this.populations
  .parallelStream()
  .forEach(p -> p.mutate(this.mutationRate));
```

Vediamo che la **JDK** ci semplifica molto il lavoro, permettendoci di eseguire delle operazioni parallelamente con una singola linea di codice. Chiaramente la **Java Virtual Machine** cercherà di gestisce nel modo migliore possibile la parallelizzazione del lavoro sfruttando le conoscenze che possiede sul sistema – ad es. numero di core – e sulle operazioni da eseguire. Ad esempio potrebbe decidere di non sfruttare il parallelismo ed eseguire le istruzioni in modo sequenziale perché il numero di automi cellulari nella popolazione è piccolo – perderebbe più tempo a creare i thread che devono gestire i task, che non ad eseguirlo sequenzialmente. Oppure decidere di suddividere le operazioni in 8 task e creare un thread per

ognuno di essi – perché il processore su cui è in esecuzione il programma possiede 8 core.

In sostanza, demandando il compito di scegliere i parametri del parallelismo alla JVM siamo sicuri che il nostro codice eseguirà senza bug ed in modo efficiente su qualunque elaboratore.

6.5 Risultati

Ora esaminiamo i risultati ottenuti nell'evolvere immagini binarie con l'applicazione `ImageEvolver`.

Ciò che vogliamo osservare dal comportamento dell'applicazione è se effettivamente riesce in tempi accettabili ad evolvere una popolazione di popolazioni di automi cellulari affinché questi esibiscano un comportamento tale da riprodurre l'immagine voluta. Nel caso che il tempo per ottenere esattamente l'immagine desiderata sia troppo grande, potremmo esaminare il risultato ottenuto dopo alcune generazioni e vedere se questo comunque si avvicina al risultato ottenuto. Nel caso ciò avvenga possiamo affermare che l'"adattamento" effettivamente è in atto sul nostro sistema, cioè gli automi cellulari stanno "imparando" a generare l'immagine che vogliamo.

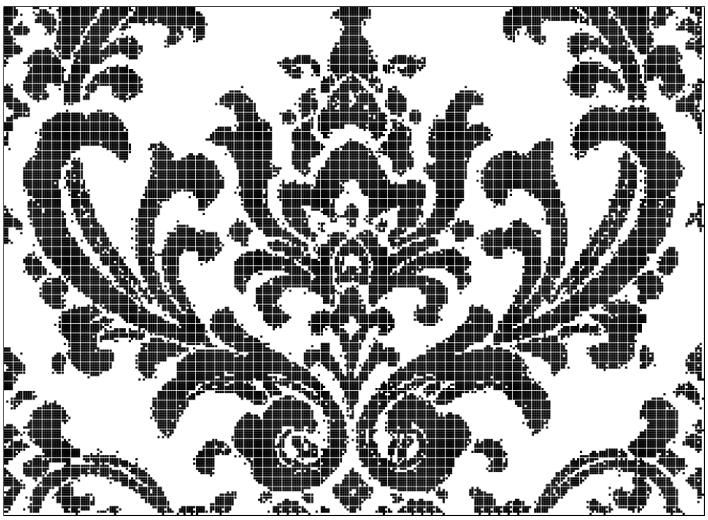
Quindi i nostri parametri di giudizio saranno:

1. Tempo necessario per evolvere l'immagine.
2. Qualità dell'immagine prodotta dopo un breve periodo di tempo, nel caso la convergenza sia troppo lenta.

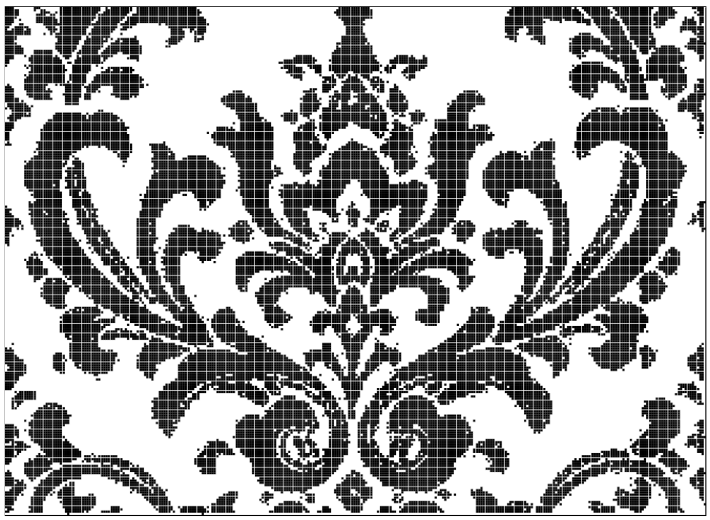
Figura 6.6: Immagine esempio da evolvere

Iniziamo con l'esaminare il suo comportamento con un'immagine di esempio – la figura 6.6. Nella pagina successiva vediamo alcuni passaggi dell'evoluzione. In quella ancora dopo abbiamo riportato le differenze tra la soluzione ottima e la soluzione attuale – in nero le sotto-immagini non ancora trovate.

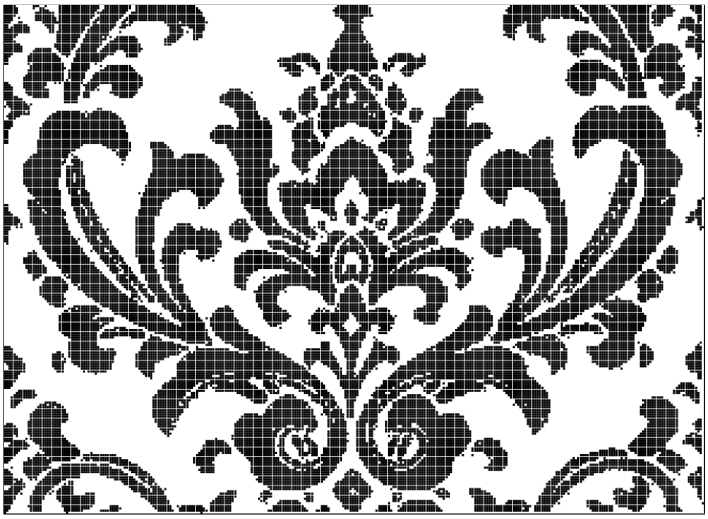
Figura 6.7: Evoluzione bitmap 6.6



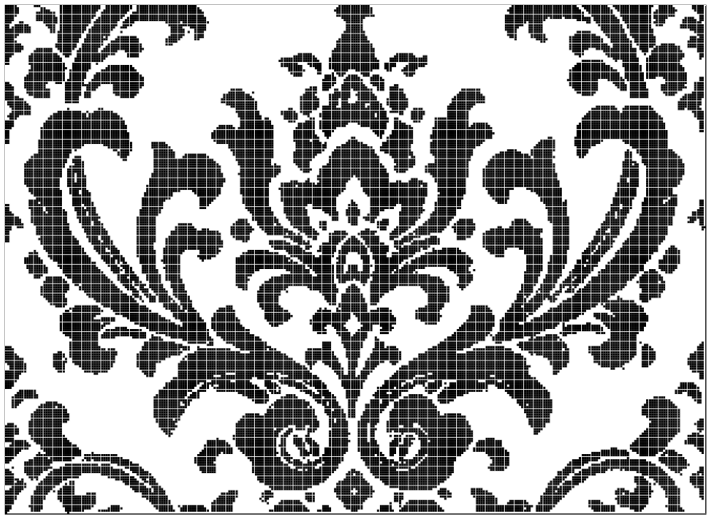
(a) generazione 1



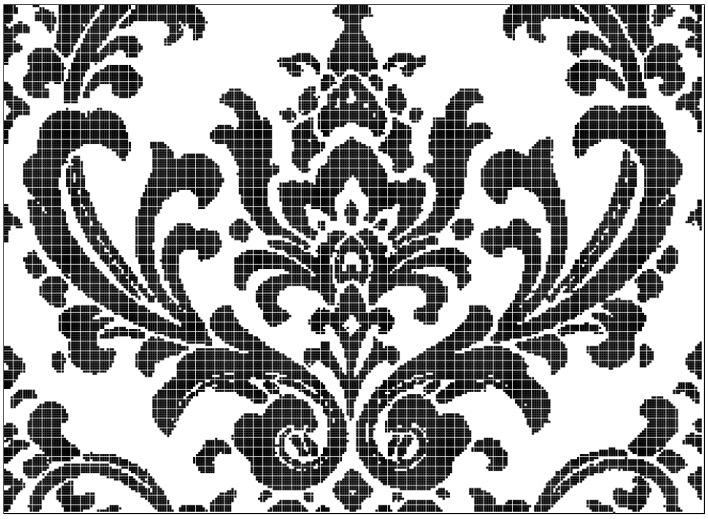
(b) generazione 2



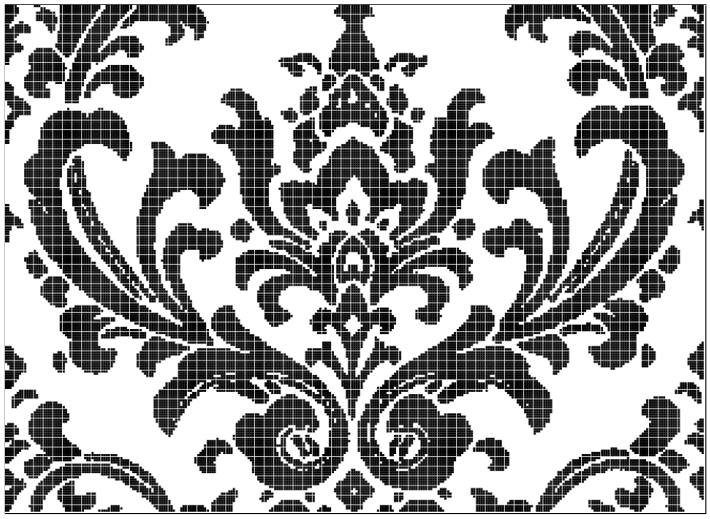
(c) generazione 5



(d) generazione 10

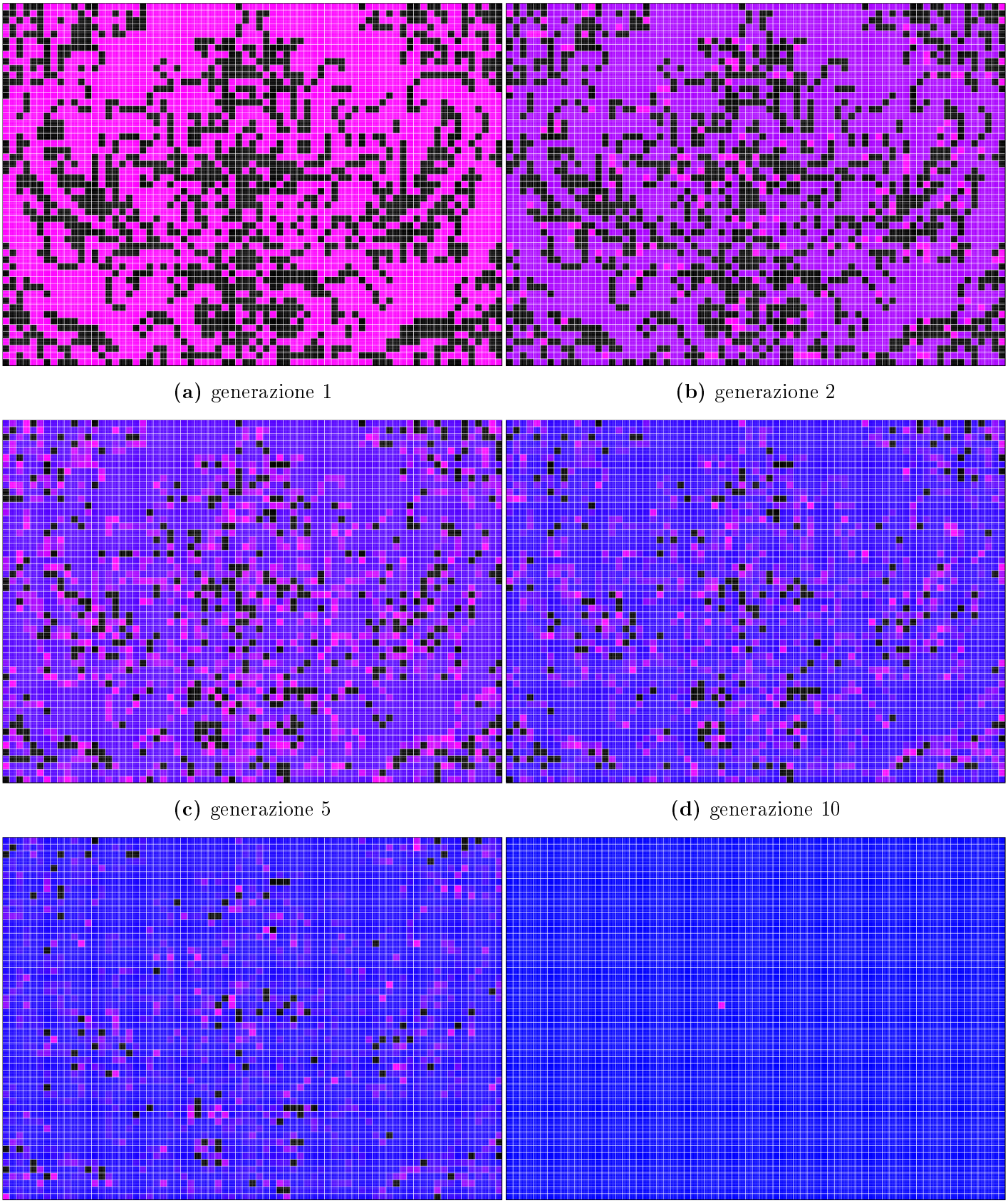


(e) generazione 20



(f) generazione 157 - soluzione ottima

Figura 6.8: Differenze dalla soluzione 6.6



(a) generazione 1

(b) generazione 2

(c) generazione 5

(d) generazione 10

(e) generazione 20

(f) generazione 157 - soluzione ottima

Osserviamo, nella figura 6.7, che già dalle prime generazioni di automi cellulari il risultato è sorprendentemente vicino alla soluzione che cerchiamo. Nella 10^a generazione facciamo già fatica a trovare differenze rispetto alla soluzione ottima raggiunta nella 157^a generazione.

In figura 6.8 riusciamo a notare meglio quali sono le differenze tra la soluzione, ad ogni specifica generazione, e la soluzione ottima. Le celle in nero sono quelle che non ancora ottime, le altre si. Più il colore si avvicina al fucsia più sono state trovate di recente.

Per renderci conto di come procede l'evoluzione dell'immagine nel tempo abbiamo raccolto alcuni dati durante l'esecuzione del programma. Nella figura 6.9 abbiamo riportato il valore medio di fitness dell'automa migliore di ogni popolazione per ogni generazione. In figura 6.10 invece mostriamo per ogni generazione il numero di popolazioni che sono riuscite a trovare la soluzione ottima per la loro sotto-immagine.

Figura 6.9: Fitness medio per generazione

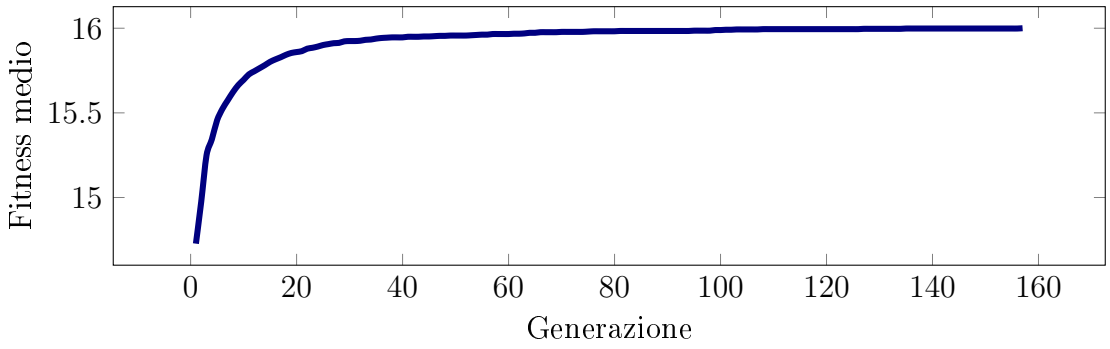
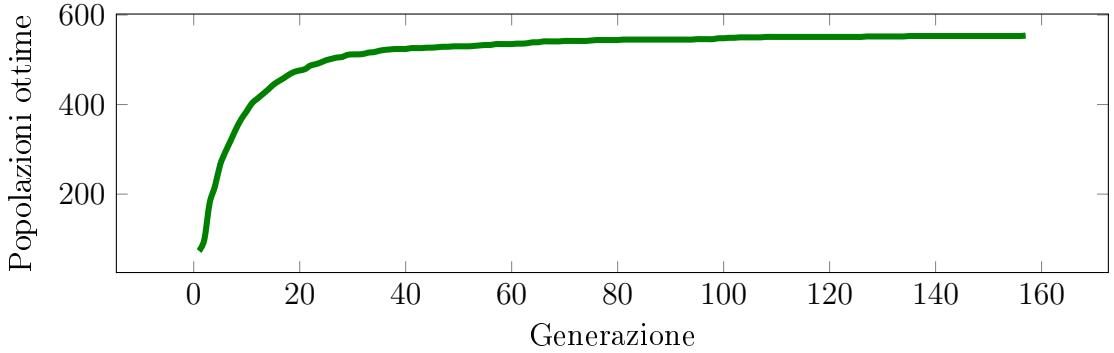


Figura 6.10: Popolazioni ottime per generazione



In entrambi i casi vediamo che la curva cresce velocemente fino alla ventesima generazione circa, e poi continua a crescere con un passo molto meno spedito. Il tempo impiegato per arrivare alla 157^a generazione è stato di 4 min e 11 s. Per raggiungere la 20^a invece ci sono voluti 48 s e per raggiungere la 10^a, 29 s. Bisogna notare che il tempo necessario per passare da una generazione all'altra diminuisce con l'avanzamento del programma perché, come detto precedentemente, quando una popolazione ha raggiunto la soluzione ottima notifica ai suoi vicini l'avvenimento e si ferma ad evolvere. Quindi con l'aumentare delle generazioni rimarranno sempre meno popolazioni da evolvere.

Le prime popolazioni a trovare la soluzione ottima saranno quelle che hanno una sotto-immagine molto semplice, ad esempio tutta nera o tutta bianca, oppure simmetrica – abbiamo visto nel capitolo 5 che le bitmap simmetriche sono molto facili da trovare. Osserviamo infatti nelle figure 6.7a e 6.8a che le sotto-immagini tutte bianche o nere sono state trovate già nella prima generazione. Quelle che invece contengono più dettagli, i bordi delle figure rappresentate ad esempio, richiedono più tempo per essere "scovate".

Differenze con popolazioni isolate

Come spiegato nelle sezioni precedenti, quando una popolazione arriva alla sua soluzione ottima essa notifica i suoi vicini dell'accaduto e gli permette di utilizzare i suoi bordi per aggiornare il loro stato della griglia.

Facciamo ora una valutazione di come questo meccanismo migliora la convergenza dell'evoluzione verso l'immagine obiettivo. Abbiamo provato ad eseguire di nuovo il programma `ImageEvolver` sull'immagine utilizzata finora – figura 6.6 – questa volta però lasciando ogni popolazione isolata durante l'esecuzione del programma.

Mentre nella situazione precedente bastavano un numero di generazioni che andava dalle 140 alle 210 circa, in questo caso ne occorrono più di 300.

Abbiamo quindi confermato la nostra intuizione sul fatto che la possibilità di comunicazione, durante l'evoluzione, tra le diverse popolazioni migliorasse la velocità di convergenza verso la soluzione.

6.5.1 Compressione delle immagini

Alla fine dell'esecuzione di `ImageEvolver` su una immagine binaria otteniamo un insieme di automi cellulari ognuno dei quali può essere descritto da:

- le sue regole di sviluppo
- per quanti passi deve svilupparsi
- il numero della generazione in cui ha raggiunto la soluzione
- la lista di indici delle celle che lo contengono

Possedendo queste informazioni possiamo rigenerare l'immagine obiettivo percorrendo i seguenti passi:

1. generare una griglia della grandezza dell'immagine obiettivo e portare ogni cella nella configurazione iniziale fissata.

2. assegnare ad ogni cella il suo automa corrispondente.

3. inizializzare un contatore a zero e per ogni suo incremento:

   (a) far sviluppare le celle i cui automi hanno il numero di generazione uguale al contatore. Lo sviluppo deve essere effettuato considerando i bordi dei vicini e per il numero di passi salvato nell'automa.

   (b) se non ci sono più automi il cui numero di generazione è maggiore o uguale al contatore fermarsi.

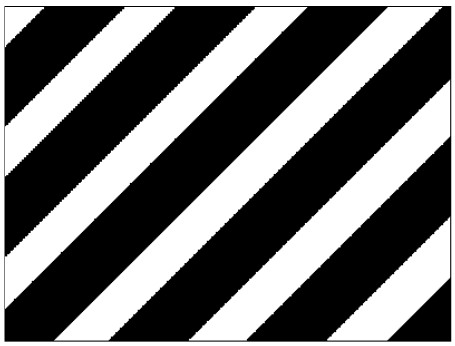
Seguendo l'algoritmo descritto generiamo esattamente l'immagine obiettivo con cui abbiamo evoluto gli automi cellulari. Le informazioni che abbiamo possiamo considerarle quindi come un insieme di regole che ci permettono letteralmente di generare un immagine. Non abbiamo bisogno di conoscere il valore di ogni singolo pixel, l'unico requisito di cui abbiamo bisogno è la conoscenza dello stato iniziale – uguale per tutti – con cui abbiamo generato gli automi.

A questo punto potremmo definire un metodo per memorizzare immagini che si basa non sulla descrizione di come appare l'immagine – quindi il valore di ogni pixel – ma sulla descrizione di come l'immagine può essere generata. Un po' come con gli esseri viventi in cui il DNA descrive in che modo le componenti dell'essere devono svilupparsi per generare l'individuo.

Il processo di generazione delle regole che descrivono l'immagine abbiamo però riscontrato non essere molto veloce. D'altra parte abbiamo osservato che anche solo evolvendo le popolazioni per alcune generazioni – cosa che richiede alcuni secondi – otteniamo un immagine molto simile a quella obiettivo, al punto che le differenze quasi non si notano. Quindi nel caso non ci preoccupi molto il fatto di avere alcuni pixel diversi da quelli originali, la strada proposta può essere percorribile.

Quello che otterremmo dalla "conversione" di un immagine binaria in un insieme di regole per generare l'immagine sarebbe una compressione dell'immagine. Consideriamo ad esempio l'immagine in figura 6.6 che ha delle dimensioni di 212x292 pixel. Utilizzando il formato PBM binario per memorizzarlo otterremo un file grande poco più di $212 \cdot 292 = 61\,904$ bit $= 7.738$ kB. Vediamo invece quanto occuperebbe se dovessimo salvare le regole degli automi cellulari che lo generano utilizzando automi grandi 4x4 di lato. L'immagine viene divisa in 3869 sotto-immagini delle quali 3315 sono uguali, quindi in totale creeremo $3869 - 3315 = 554$ popolazioni che genereranno altrettanti automi cellulari con le regole ottime. Utilizzando il vicinato di von Neumann sappiamo che le regole di sviluppo di ogni automa sono una stringa grande 32 bit. In totale perciò la somma delle regole occuperà $32 * 554 = 17\,728$ bit $= 2.216$ kB.

Figura 6.11



Chiaramente utilizzando un immagine molto meno complessa, come quella in figura 6.11 avremmo un risparmio in dimensioni molto maggiore. In questo caso l'immagine è grande 300x400 pixel e contiene 7500 sotto-immagini. Le sotto-immagini distinte sono solamente 30 quindi il totale delle regole sarebbe $30*32 = 960$ bit, invece dei 15 kB dell'immagine originale.

Nel calcolo delle dimensioni abbiamo però tralasciato alcuni degli attributi di cui abbiamo bisogno per rigenerare l'immagine, cioè il numero di passi di sviluppo, il numero di generazione e gli indici delle celle. Non continuiamo però a fare delle stime perchè queste sarebbero molto fantasiose, dato che non abbiamo fatto uno vero e proprio studio sulla codifica del nostro ipotetico formato.

Alla fine, osservando i risulati ottenuti con lo sviluppo di questo progetto possiamo dire che:

1. Siamo riusciti a trovare un modo abbastanza efficiente per far evolvere in modo automatico degli automi cellulari affinché questi, posizionati uno affianco all'altro, producano una qualunque immagine obiettivo binaria.

2. Il meccanismo può richiedere parecchio tempo per riprodurre esattamente l'immagine voluta, ma in breve tempo riesce ad approssimarla con poche differenze rispetto all'originale.

6.5.2 Sviluppi futuri

Dati i risultati molto soddisfacenti ottenuti ed il grande interesse da parte dell'autore riguardo gli argomenti affrontati nello sviluppo del progetto, possiamo affermare con una buona dose di certezza che questo progetto in futuro verrà migliorato ed ampliato.

Nello specifico si potrà valutare la possibilità di implementare un meccanismo che preveda fin dall'inizio dell'evoluzione la comunicazione tra le popolazioni. In questo modo si toglie il bisogno di sincronizzare l'evoluzione di ogni singola popolazione in base al successo dei propri vicini. In tal senso si potrebbe anche esplorare la possibilità di evolvere delle regole che permettano ad ogni automa di ogni sottoimmagine di arrivare alla configurazione finale nello stesso passo di sviluppo. Con quest'ultima miglioria non avremmo più nemmeno il bisogno di memorizzare fino a che passo ogni automa deve svilupparsi.

Se queste ultime due modifiche dovessero andare in porto si potrà definire un formato di memorizzazione di file che permetta di salvare l'insieme delle regole trovate.

Un'altro punto molto interessante che può essere esplorato è l'evoluzione di automi cellulari in cui le celle possono assumere più di due stati. Si potrebbe quindi cercare di evolvere immagini in scala di grigi o addirittura a colori. Sappiamo però che aumentando il numero di stati cresce di conseguenza lo spazio delle possibili soluzioni, bisognerà pertanto valutare attentamente l'efficienza dell'algoritmo di ricerca della soluzione voluta.

Conclusioni

Giunti alla fine di questa relazione possiamo concludere dicendo che siamo riusciti nel nostro tentativo di creare un sistema che evolva una popolazione di automi cellulari affinché generino una immagine binaria.

L'approccio iniziale è stato molto teorico e ci è risultato utile per definire in modo chiaro gli elementi fondamentali del sistema. Una volta fatto ciò abbiamo iniziato a fare i nostri primi esperimenti, basandoci sulla struttura teorica che avevamo costruito, per osservare se i risultati ottenuti rispettassero le nostre previsioni. Queste prime prove hanno avuto come obiettivo quello di evolvere delle piccole immagini binarie utilizzando degli automi cellulari. Abbiamo visto che i risultati sono stati abbastanza promettenti e conformi a ciò che ci eravamo figurati in precedenza, cioè: gli automi cellulari, sotto la spinta di un algoritmo genetico, riescono ad evolvere ed a mostrare il comportamento voluto.

In fine abbiamo provato a costruire un sistema che evolvesse, non singoli automi cellulari, ma un gruppo di automi in modo che se posizionati uno affianco all'altro in una griglia, generassero un'immagine binaria. L'immagine che ne viene fuori deve essere quella per cui gli automi sono stati fatti evolvere. In questo progetto abbiamo aggiunto la possibilità di comunicazione tra gli automi durante l'evoluzione e tale fattore si è dimostrato molto importante per la convergenza verso la soluzione.

I risultati del progetto finale sono molto promettenti in quanto, nonostante il tempo necessario per trovare la soluzione esatta non sia molto basso, osserviamo che già dopo qualche decina di generazioni gli automi prodotti approssimano molto bene l'immagine obiettivo. In oltre abbiamo visto come la nostra tecnica permette, in determinati casi, di ottenere una compressione dell'immagine.

Lo scopo di questa tesi e del progetto non è propriamente quello di costruire un metodo con lo scopo di poter essere applicato in qualche contesto reale, ma principalmente si pone come obbiettivo il mostrare come si può indurre artificialmente un adattamento in sistemi complessi simulati al computer, utilizzando degli algoritmi genetici. Nonostante ciò una possibile applicazione che può derivare dai

risultati ottenuti nello sviluppare il progetto, è un algoritmo di compressione lossy – cioè con perdita di qualità – per le immagini.

Bibliografia

[1] Institute Biomimicry. Ask nature. `http://www.asknature.org/`.

[2] Ron Breukelaar and Th Bäck. Using a genetic algorithm to evolve behavior in multi dimensional cellular automata: emergence of behavior. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 107–114. ACM, 2005.

[3] Stephen Coombes. The geometry and pigmentation of seashells. *Techn. Ber. Department of Mathematical Sciences*, 2009.

[4] James P Crutchfield and Melanie Mitchell. The evolution of emergent computation. *Proceedings of the National Academy of Sciences*, 92(23):10742–10746, 1995.

[5] Richard Dawkins. *The Selfish Gene*. Oxford University Press, Oxford, UK, 1976.

[6] John Henry Holland. *Hidden Order: How Adaptation Builds Complexity*. Perseus Books, 1995.

[7] Andrew Ilachinski. *Cellular Automata: A Discrete Universe*. World Scientific, 2001.

[8] Mark Land and Richard K Belew. No perfect two-state cellular automata for density classification exists. *Physical review letters*, 74(25):5148, 1995.

[9] Christopher G. Langton. Self-reproduction in cellular automata. *Physica D: Nonlinear Phenomena*, 10(1–2):135 – 144, 1984.

[10] Melanie Mitchell, James P Crutchfield, and Peter T Hraber. Evolving cellular automata to perform computations: Mechanisms and impediments. *Physica D: Nonlinear Phenomena*, 75(1):361–391, 1994.

[11] John Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966.

[12] Oracle. Aggregate operations. `https://docs.oracle.com/javase/tutorial/collections/streams/index.html`.

[13] Oracle. Lambda expressions. `https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`.

[14] David Peak, Jevin D. West, Susanna M. Messinger, and Keith A. Mott. Evidence for complex, collective dynamics and emergent, distributed computation in plants. *Proceedings of the National Academy of Sciences of the United States of America*, 101(4):918–922, January 2004.

[15] Foundation Processing. Processing. `https://processing.org/`.

[16] project Netpbm. Pbm format. `http://netpbm.sourceforge.net/doc/pbm.html`, November 2013.

[17] Wikipedia. Java (programming language). `https://en.wikipedia.org/wiki/Java_(programming_language)`, June 2015.

[18] Wikipedia. Object-oriented programming. `https://en.wikipedia.org/wiki/Object-oriented_programming`, June 2015.

[19] Stephen Wolfram. Statistical mechanics of cellular automata. *Reviews of modern physics*, 55(3):601, 1983.

[20] Stephen Wolfram. *A New Kind of Science*. Wolfram Media Inc., Champaign, Ilinois, US, United States, 2002.