Fondamenti algebrici degli automi cellulari invertibili

Leo Liberti

Università degli Studi di Torino

Facoltà di Scienze M.F.N.

Tesi di Laurea in Matematica

Fondamenti Algebrici degli Automi Cellulari Invertibili

Leo Liberti

Novembre 1997

Relatore: Prof. Umberto Cerruti

Sommario

Vengono poste le basi per una trattazione algebrico-combinatorica degli automi cellulari monodimensionali. Si investiga la struttura del monoide delle evoluzioni (o trasformazioni cellulari) e la struttura del gruppo delle trasformazioni cellulari invertibili. Si analizzano classi di trasformazioni che mantengono l'invertibilità quando applicate ad automi di qualsiasi lunghezza. Viene effettuata un'analisi dei lavori di Amoroso, Patt, Richardson nel campo degli automi invertibili infiniti. Vengono dati cenni sugli automi cellulari a più dimensioni e vengono descritte delle simulazioni di automi pluridimensionali per mezzo degli automi ad una dimensione.

Ringraziamenti

Sebbene sia stato molto attento a indicare quali, dei concetti esposti in questa tesi, venissero da varie pubblicazioni e quali fossero invece miei, non mi è stato possibile citare nella bibliografia tutte le idee suggeritemi a voce da altre persone, che passo quindi a ringraziare.

Il relatore di questa tesi Prof. Umberto Cerruti, cui appartiene l'idea della "mappa diagonale descritta nell'ultimo capitolo.

Laura Basile, studentessa in questo ateneo, che mi ha suggerito il punto cruciale della dimostrazione del teorema (7.2) ed ha passato molte ore a correggere vari capitoli della tesi.

Mio padre, per le correzioni delle parti più discorsive e meno matematiche della tesi.

Il Prof. Jarkko J. Kari, dell'Università dell'Iowa, che ha risposto ai miei insistenti messaggi e che ha dimostrato per me il teorema (6.8), a cui io avevo lavorato infruttuosamente per circa quattro mesi e che lui ha liquidato nello spazio di ventiquattr'ore.

Indice

1 Introduzione **10**

1.1 Permutazioni come trasformazioni cellulari 13

1.2 Matrici come trasformazioni cellulari 14

1.3 Polinomi come trasformazioni cellulari 15

1.4 Critica al metodo 16

1.5 Esistenza di trasformazioni non lineari 17

1.6 Terminologia 18

2 Formalismi e Proprietà Fondamentali **19**

2.1 Fondamenti 19

2.2 Struttura moltiplicativa di $\mathcal{A}^n(R)$ 23

2.3 Trasformazioni con raggio d'azione $< n$ 28

2.4 Prodotto di trasformazioni in $\mathcal{A}^n(R)$ 32

3 Rappresentazioni Numeriche **34**

3.1 Rappresentazioni locali 34

3.2 Rappresentazioni decimali 36

3.3 Convenzioni numeriche per $\mathcal{B}^n_{r_0,r_1}(R)$ 38

3.4 Prodotto di trasformazioni 43

3.4.1 Divisione in orbite 43

3.4.2 Applicazione di σ ad un vettore di R^n 48

3.4.3 Calcolo del prodotto 50

3.5 Applicazioni dell'algoritmo di moltiplicazione 54

3.5.1 Tavole di moltiplicazione 54

3.5.2 Ottimizzazione degli algoritmi 55

4 Divisione in Orbite **57**

4.1 Fondamenti 57

4.2 Rappresentazioni orbitali 59

4.3 Schema orbitale 60

4.4 Prodotto simbolico di trasformazioni in $\mathcal{A}^n(R)$ 64

5 Trasformazioni Cellulari Invertibili **67**

5.1 Il gruppo $\mathcal{G}^n(R)$ 67

5.2 Caratterizzazione degli elementi di $\mathcal{G}^n(R)$ 68

5.3 La struttura di $\mathcal{G}^n(R)$ 70

5.4 Sottogruppi di G_i^n 79

5.5 $\mathcal{G}^n(R)$ come gruppo di permutazioni 80

5.6 Altre applicazioni della rappresentazione globale 83

5.6.1 Prodotto di rappresentazioni globali 83

5.6.2 Conversione alla rappresentazione locale 84

5.6.3 Invertibilità e rappresentazioni globali 85

6 La Rappresentazione Locale **87**

6.1 Indipendenza dalla lunghezza dell'automa 89

6.2 Questione aperta 96

7 Automi di Lunghezza Infinita **101**

7.1 Giardini dell'Eden 104

7.2 Algoritmo di verifica dell'invertibilità 107

7.3 Il gruppo $\mathcal{G}^{\infty}(R)$ 115

8 Automi Cellulari d-Dimensionali **116**

8.1 L'intorno 116

8.2 Game of Life 118

8.3 Mappe transdimensionali 119

8.3.1 Mappa dell'intorno ipercubico 119

8.3.2 Mappa diagonale 121

8.4 Reversibilità degli automi bidimensionali 123

9 Conclusione **125**

A Appendice: Il Programma `CAInvPrm` **126**

B Appendice: Il Programma `CAInvInf` **154**

Fondamenti Algebrici degli Automi Cellulari Invertibili

Lo scopo della presente tesi è quello di fornire degli strumenti algebrici per la manipolazione degli automi cellulari, in particolare per investigare la loro reversibilità e trovare gli inversi.

Un automa cellulare, nella sua accezione più ampia, è un sistema dinamico discreto con una legge di evoluzione locale; ovvero è un reticolo n-dimensionale con una variabile in ciascun nodo (o cella) in cui il valore di qualsiasi variabile al tempo $t+1$ dipende dai valori delle variabili al tempo t nei nodi di un intorno prefissato. Come si può vedere la definizione è molto generica, in quanto non specifica nulla né della geometria del reticolo, né delle sue dimensioni, né delle condizioni al suo contorno, né della topologia dell'intorno di ogni nodo, né della funzione di evoluzione. Molto in generale, se $V(x)$ è l'intorno della cella x, si ha

$$x(t+1) = \phi(V(x(t)))$$

dove ϕ rappresenta la funzione di evoluzione.

In questa sede tratteremo soprattutto gli automi cellulari finiti, con condizioni al contorno periodiche (cioé "chiusi su sè stessi), unidimensionali e a stati discreti. Parleremo solo negli ultimi due capitoli di automi cellulari infiniti e multidimensionali.

Quando un teorema non è un contributo originale viene chiaramente indicato il nome dell'autore ed il relativo richiamo bibliografico.

Nel primo capitolo si introducono alcuni concetti di base e vengono discusse delle problematiche inerenti all'inadeguatezza delle tecniche più comuni usate per rappresentare una legge di evoluzione di automi cellulari.

Nel secondo capitolo viene analizzato l'insieme delle regole locali di evoluzione di un automa cellulare unidimensionale finito con condizioni al contorno periodiche e si prova che tale insieme è un monoide con prodotto definito dalla composizione di funzioni. Si dimostra che le regole locali commutano con lo *shift* (la trasformazione canonica che sposta l'automa di una cella verso sinistra).

Nel terzo capitolo vengono affrontate questioni di calcolo numerico. Vengono introdotte delle rappresentazioni di regole locali che facilitano la loro manipolazione per mezzo di un calcolatore. Viene discusso in dettaglio un algoritmo per effettuare il prodotto di regole locali.

Nel quarto capitolo vengono introdotte le tecniche fondamentali per affrontare lo studio della struttura del gruppo delle regole locali che hanno un'inversa; l'analisi dettagliata della struttura del gruppo viene effettuata nel quinto capitolo, in cui si propongono anche degli algoritmi per trovare l'inversa di una regola locale data.

Nel sesto capitolo si prova che esiste un sottogruppo di regole locali invertibili che hanno la proprietà di restare invertibili anche quando applicate ad automi di varia lunghezza. Viene poi posta una questione aperta riguardante la struttura di alcune regole locali che sono invertibili solo su automi di lunghezza dispari.

Nel settimo capitolo si analizzano alcuni aspetti degli automi di lunghezza infinita, in particolare si prova che una regola locale invertibile su automi di lunghezza infinita è invertibile su automi di qualsiasi lunghezza finita. Viene discusso un risultato fondamentale di Richardson e viene implementato un algoritmo proposto da Amoroso e Patt per decidere se una data regola locale ha un'inversa su automi di lunghezza infinita.

Nell'ottavo capitolo vengono fatti dei cenni sugli automi cellulari a più

dimensioni. Vengono proposti due metodi per mappare gli automi multi-dimensionali su automi ad una dimensione. Il primo ha valore generale (si applica a qualsiasi automa d-dimensionale finito con reticolo euclideo) e permette di simulare l'automa d-dimensionale per mezzo di un numero finito di automi unidimensionali finiti. La simulazione ha un costo computazionale elevato ma possiede l'indiscusso valore teorico di estendere i risultati dei capitoli precedenti anche agli automi a più dimensioni. Il secondo metodo è basato sull'applicazione dell'isomorfismo di gruppi

$$C_n \cong C_{p_1}^{e_1} \times \ldots \times C_{p_d}^{e_d}$$

(dove $p_1^{e_1} \cdots p_k^{e_d}$ è la fattorizzazione in primi di n) a reticoli d-dimensionali per ottenere una classe di automi d-dimensionali che possono essere simulati da un singolo automa ad una dimensione di lunghezza n.

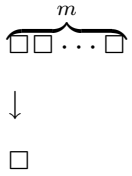
1 Introduzione

Possiamo rappresentare un automa cellulare unidimensionale finito con una stringa di valori $a_1, ..., a_n$ di un anello[1] finito R in cui porremo le condizioni al contorno in modo che $a_{n+1} = a_1$ e $a_0 = a_n$. Geometricamente parlando, curviamo questa stringa fino ad unire le due estremità. L'intorno di una cella a_i è il vettore di m celle $(a_i, a_{i+1}, \ldots, a_{i+m-1})$ dove ovviamente $m < n$. Se indichiamo con $\phi : R^m \to R$ la legge di evoluzione dell'automa cellulare, il valore a_i al tempo $t + 1$ sarà dato da

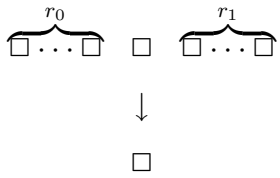
$$a_i(t + 1) = \phi(a_i(t), a_{i+1}(t), \ldots, a_{i+m-1}(t))$$

Chiameremo m il **raggio d'azione** di ϕ.

1.1 (Convenzione dell'intorno) Si noti che abbiamo fissato per convenzione l'intorno di a_i come a_i stessa e $m-1$ celle alla sua destra. Chiameremo questo tipo di intorno **intorno convenzionale**. Nell'intorno convenzionale si ha



È possibile fissare l'intorno in modi diversi: in generale con r_0 celle a sinistra di a_i, a_i stessa e r_1 celle alla sua destra[2], in modo che $r_0 + r_1 + 1 = m$. Si parla allora di **raggio sinistro** e **raggio destro** dell'intorno; graficamente si ha



[1] Viene dato all'insieme degli stati dell'automa la struttura di anello solo per poter descrivere le leggi di evoluzione degli automi con mezzi algebrici anziché soltanto combinatorici. In teoria non sarebbe strettamente necessario.

[2] Cfr. [Wol83, WMO84].

Chiameremo un intorno di questo genere un **(r_0, r_1)-intorno**. È evidente che un intorno convenzionale è un $(0, m-1)$-intorno, e che quindi gli (r_0, r_1)-intorni hanno valore più generico: cercheremo perciò di usare questi ultimi laddove possibile. Useremo gli intorni convenzionali soprattutto nel capitolo 2, dove mostreremo che la distinzione non è importante ai fini teorici. □

La funzione ϕ è una **regola locale** , nel senso che da un intorno di un punto al tempo t calcola il valore del punto stesso al passo temporale successivo $t+1$. Applicando ϕ agli intorni convenzionali di tutte le componenti di

$$\underline{a}(t) = (a_1(t), \ldots, a_n(t))$$

otteniamo il vettore

$$\underline{a}(t+1) = (a_1(t+1), \ldots, a_n(t+1))$$

e quindi possiamo estendere ϕ a una funzione $\sigma : R^n \to R^n$ tale che

$$\sigma(\underline{a}(t)) = \underline{a}(t+1)$$

Una estensione di una regola locale ϕ su (r_0, r_1)-intorni sarà indicata con σ_{r_0,r_1}.

Ricordiamo a questo punto che non tutte le funzioni $R^n \to R^n$ sono estensioni di una regola locale: per esempio, una funzione σ tale che

$$\sigma((1,1,1)) = (0,0,1)$$

non può avere estendere una regola locale. Supponiamo infatti che ϕ sia la regola locale estesa da σ con raggio d'azione uguale a 3; allora

$$\phi((a_1, a_2, a_3)) = \phi((1,1,1)) = 0$$

e

$$\phi((a_3, a_1, a_2)) = \phi((1,1,1)) = 1$$

e quindi ϕ non è ben definita. Una funzione $\sigma : R^n \to R^n$ che estende una regola locale può sempre essere vista come una legge di evoluzione di automi cellulari.

Ci proponiamo di trovare dei modi di rappresentare σ che facilitino la comprensione del comportamento di σ sui vettori di R^n. In particolare ci proponiamo di investigare il problema della reversibilità: data un'estensione di una regola locale (che chiameremo anche "trasformazione cellulare) determinare se è invertibile e quale è l'inversa.

Cominceremo con il dimostrare l'inadeguatezza in generale delle più comuni funzioni $R^n \to R^n$ come rappresentazioni di trasformazioni cellulari: le permutazioni del gruppo S_n e le matrici $n \times n$. Ne seguirà il bisogno dell'introduzione di nuove tecniche di calcolo. A questo fine prenderemo "in prestito dal capitolo 2 un teorema fondamentale, il teorema (2.3). Sia α la permutazione $(12 \dots n)$ intesa come funzione $R^n \to R^n$: il teorema afferma che una funzione $\tau : R^n \to R^n$ è una trasformazione cellulare se e solo se per ogni vettore $\underline{v} \in R^n$ la seguente condizione è valida:

$$\alpha\tau(\underline{v}) = \tau(\alpha\underline{v})$$

In altre parole, τ ha una rappresentazione locale se e solo se τ commuta con α. In questo caso si dice anche che τ ammette una localizzazione. Fissiamo a questo punto per convenzione che

$$\alpha(v_1, v_2, \dots, v_n) = (v_{\alpha(1)}, v_{\alpha(2)}, \dots, v_{\alpha(n)}) = (v_2, v_3, \dots, v_n, v_1)$$

Si osserva che ovviamente α commuta con se stesso e quindi ammette una localizzazione; la regola locale derivata da α viene comunemente chiamata **shift**.

1.1 Permutazioni come trasformazioni cellulari

Sia g un elemento del gruppo simmetrico S_n. Allora g è una trasformazione cellulare se e solo se

$$g\alpha = \alpha g$$

Vogliamo trovare il sottogruppo massimale $G \leq S_n$ tale che per ogni $g \in G$ si abbia $g\alpha = \alpha g$.

Sia

$$g = \begin{pmatrix} 1 & 2 & \dots & n \\ g_1 & g_2 & \dots & g_n \end{pmatrix}$$

e

$$\alpha = \begin{pmatrix} 1 & 2 & \dots & n \\ 2 & 3 & \dots & 1 \end{pmatrix}$$

si ottiene

$$g\alpha = \begin{pmatrix} 1 & 2 & \dots & n \\ g_2 & g_3 & \dots & g_1 \end{pmatrix}$$

$$\alpha g = \begin{pmatrix} 1 & & 2 & & \dots & n & \\ g_1+1 & \mod n & g_2+1 & \mod n & \dots & g_n+1 & \mod n \end{pmatrix}$$

e quindi per ogni $1 \leq i < n$ si ha $g_{i+1} = g_i + 1 \mod n$ e $g_1 = g_n + 1 \mod n$. Da queste condizioni si ha che

$$\begin{aligned} g_2 &= g_1 + 1 \mod n \\ g_3 &= g_1 + 2 \mod n \\ &\vdots \\ g_n &= g_1 + n - 1 \mod n \end{aligned}$$

e quindi

$$g = \begin{pmatrix} 1 & & 2 & & \dots & n & \\ g_1 & g_1+1 & \mod n & \dots & g_1+n-1 & \mod n & \end{pmatrix}$$

da cui $g = \alpha^{g_1}$. Dunque $G = \langle\alpha\rangle \cong C_n$.

Questo significa che solo le potenze di α sono trasformazioni cellulari.

1.2 Matrici come trasformazioni cellulari

Sia $A = (a_{ij})$ una matrice $n \times n$. A è una trasformazione cellulare se e solo se, per ogni $\underline{v}^\top = (v_1, \ldots, v_n) \in R^n$,

$$A(\alpha\underline{v}) = \alpha A(\underline{v})$$

ovvero, per ogni $i \leq n$,

$$\sum_{j=1}^{n} a_{ij}\ v_{j+1} = \sum_{j=1}^{n} a_{i+1,j}\ v_j$$

dove gli indici hanno la proprietà che se appare un numero $k > n$, k viene rimpiazzato da $k \mod n$. Ricordiamo che la condizione sopra deve valere per ogni $\underline{v} \in R^n$, cioè per v_j arbitrari. Questo ci permette di concludere che A è una trasformazione cellulare se e solo se, per ogni $i, j \leq n$,

$$a_{ij} = a_{i+1,j+1}$$

Una matrice con questa proprietà si chiama **matrice circolante**. Un discorso completo sulle matrici circolanti sarebbe qui fuori luogo, data la vastità dell'argomento; è tuttavia importante ricordare che:

- L'insieme di tutte le matrici circolanti $n \times n$ definite su un anello R è chiuso rispetto alla moltiplicazione di matrici.
- Per ogni $\underline{v} \in R^n$ si ha

$$\alpha\underline{v} = \begin{pmatrix} 0 & 1 & 0 & \ldots & 0 \\ \vdots & \vdots & \ddots & & \vdots \\ 0 & 0 & 0 & \ldots & 1 \\ 1 & 0 & 0 & \ldots & 0 \end{pmatrix} \underline{v}^\top$$

e quindi si possono rappresentare tutte le permutazioni del gruppo ciclico C_n (e cioè, per quanto detto sulle permutazioni come trasformazioni cellulari, tutte e sole le permutazioni che rappresentano delle trasformazioni cellulari) per mezzo delle matrici circolanti.

- Le matrici sono rappresentazioni di mappe lineari, e non vi sono mappe lineari $R^n \to R^n$ che non siano rappresentabili con le matrici; perciò le matrici circolanti sono tutte e sole le trasformazioni cellulari lineari.

Vediamo ora un altro metodo di rappresentare le trasformazioni cellulari. Il metodo consiste nell'usare i polinomi anziché i vettori, ed è sicuramente il metodo più diffuso[3] di trattare gli automi cellulari a causa della facilità computazionale che presenta.

1.3 Polinomi come trasformazioni cellulari

In questa rappresentazione la distinzione di cui al punto (1.1) riveste una certa importanza. Anziché vedere l'automa cellulare come un vettore, lo raffiguriamo come un polinomio. Per ogni vettore $\underline{v} = (v_0, v_1, \ldots, v_{n-1})$ introduciamo il polinomio

$$A^t(x) = \sum_{i=0}^{n-1} v_i \; x^i = v_0 + v_1 x + v_2 x^2 + \cdots + v_{n-1} x^{n-1}$$

che rappresenta l'automa cellulare al tempo t. Le trasformazioni cellulari vengono invece rappresentate da una particolare classe di polinomi, i dipolinomi, in cui si ammettono anche esponenti negativi della x. Per esempio, una trasformazione cellulare in cui al tempo $t+1$ la cella i-esima contiene la somma modulo 2 delle celle $(i-1)$- e $(i+1)$-esime al tempo t si indica con il dipolinomio

$$T(x) = x^{-1} + x$$

[3] Cfr. [WMO84].

Una trasformazione cellulare su (r_0, r_1)-intorni avrà una rappresentazione polinomiale

$$T(x) = a_{-r_0}x^{-r_0} + \cdots + a_{-1}x^{-1} + a_0 + a_1x + \cdots + a_{r_1}x^{r_1}$$

In questo modo possiamo scrivere

$$A^{t+1}(x) = T(x)A(x) \mod (x^n - 1) \tag{1}$$

dove la riduzione modulo $x^n - 1$ impone le condizioni circolari al contorno.

È evidente che l'applicazione della trasformazione cellulare $T(x)$ all'automa $A(x)$ mediante la regola (1) è tale che la trasformazione è una mappa lineare. Poiché abbiamo visto in precedenza che le trasformazioni cellulari lineari potevano essere tutte rappresentate mediante matrici circolanti, questo metodo può solo rappresentare una sottoclasse delle trasformazioni date dalle matrici circolanti. Poiché si può dimostrare che esiste un isomorfismo fra l'algebra data dai polinomi con il prodotto (1) e l'algebra delle matrici circolanti, è chiaro che le trasformazioni ottenute con questo metodo sono esattamente tutte le trasformazioni cellulari lineari.

1.4 Critica al metodo

Abbiamo presentato tre diversi approcci al problema di trattare algebricamente l'evoluzione di un automa cellulare. Il primo, basato sulle permutazioni, non ha dato grandi frutti: abbiamo visto infatti che potevamo includerlo come sottoclasse del secondo metodo, basato sulle matrici. Il metodo basato sui polinomi è essenzialmente lo stesso di quello basato sulle matrici, ma il prodotto è più semplice. Il problema di fondo di tutti i metodi precedenti è la necessità che la trasformazione sia lineare sulle celle dell'automa, condizione che in realtà è eccessivamente restrittiva: lo studio degli automi cellulari, infatti, è volto o alle simulazioni di sistemi complessi, o alla generazione di

numeri pseudo-casuali, oppure ancora alla creazione di chiavi crittografiche; e in tutti questi campi la non linearità della trasformazione considerata è molto importante. Abbiamo dunque una situazione in cui le uniche trasformazioni che è agevole studiare sono proprio quelle che non interessano ai fini delle applicazioni pratiche. Partendo da queste considerazioni presenteremo perciò una serie di metodi per trattare e investigare tutte le trasformazioni cellulari a prescindere dalla linearità e in particolare quelle invertibili, che sono largamente usate nelle simulazioni del comportamento dei gas e nella crittografia.

Proprio a causa della complessità degli automi cellulari, che è il nostro punto debole ma anche la ragione della loro utilità, uno studio algebrico in questo senso non può che essere, ovviamente, molto limitato e molto complicato. Numerose questioni rimangono aperte o indecidibili: ci limiteremo ad aggiungere alla conoscenza degli automi cellulari alcune proprietà che in molti casi possono essere di grande aiuto al loro studio.

1.5 Esistenza di trasformazioni non lineari

La prima domanda fondamentale che affronteremo è: esistono trasformazioni non lineari? La risposta è affermativa e possiamo dimostrarlo con semplici considerazioni di cardinalità. Sia $\mathcal{A}^n(R)$ l'insieme di tutte le trasformazioni cellulari con raggio d'azione m uguale alla lunghezza n degli automi, definiti su un anello R. Possiamo definire un elemento di $\mathcal{A}^n(R)$ dicendo qual è il valore della cella al tempo $t+1$ quando il suo intorno di n celle al tempo t è il vettore $\underline{v}$. Abbiamo dunque bisogno di specificare un valore in R per ogni vettore di R^n: in tutto otteniamo $|R|^{|R|^n}$ possibili successioni di $|R|^n$ valori, pertanto $|\mathcal{A}^n(R)| = |R|^{|R|^n}$. Le trasformazioni cellulari lineari, invece, sono date da tutte le matrici circolanti su R: poiché una matrice

circolante è completamente determinata da una sua qualsiasi riga o colonna, ne abbiamo in tutto $|R|^n$. L'unico caso in cui le cardinalità coincidono è quando $|R| = 1$, ovvero $R = \{0\}$, il caso banale. Risulta quindi evidente che se R non è banale, la classe delle trasformazioni non lineari è la quasi totalità delle trasformazioni cellulari.

1.6 Terminologia

Faremo spesso uso del termine "vettore senza riferirci necessariamente ad un elemento di uno spazio vettoriale bensì come sinonimo di "successione finita ed ordinata di elementi di un anello R. L'ordine delle componenti di un vettore potrà seguire l'ordine della posizione o l'ordine inverso — così potremo avere vettori come (v_1, v_2, v_3) oppure come

$$(v_7, v_6, v_5, v_4, v_3, v_2, v_1, v_0)$$

Le permutazioni del gruppo simmetrico S_n potranno, a seconda dei casi, agire sull'insieme $\{1, \ldots, n\}$ o sull'insieme $\{0, \ldots, n-1\}$.

Per indicare la i-esima componente del vettore $\underline{v} = (v_1, \ldots, v_n)$ definiamo una funzione λ con due argomenti tale che $\lambda(\underline{v}, i) = v_i$. In qualche caso, quando non possono sorgere dubbi, ove sia già stato introdotto un vettore $\underline{v}$ parleremo delle sue componenti v_i senza specificare che si tratta delle componenti di $\underline{v}$.

Abbiamo visto che una legge di evoluzione di un automa cellulare si può specificare con la sua regola locale o con la funzione $R^n \to R^n$ da essa derivata. Vedremo in seguito che esistono molte rappresentazioni di una legge di evoluzione; in generale indicheremo con ϕ e ψ le regole locali (altrimenti dette rappresentazioni locali) e con σ e τ le funzioni $R^n \to R^n$, che chiameremo trasformazioni cellulari o rappresentazioni globali.

2 Formalismi e Proprietà Fondamentali

2.1 Fondamenti

In questo capitolo rappresenteremo gli automi cellulari di lunghezza n su un anello R con dei vettori in R^n e investigheremo la struttura moltiplicativa dell'insieme di tutte le possibili leggi di evoluzione per mezzo di una particolare classe di funzioni $R^n \to R^n$.

Sia R un anello e n un intero positivo. Sia $T^n(R)$ l'insieme di tutte le funzioni da R^n a R^n, ovvero

$$T^n(R) = \{\xi \mid \xi : R^n \to R^n\}$$

Per $\sigma, \tau \in T^n(R)$ e per ogni elemento $\underline{v} \in R^n$ si definisca

$$(\sigma * \tau)(\underline{v}) = \sigma(\tau(\underline{v}))$$

Poiché $(\sigma * \tau) : R^n \to R^n$ è chiaro che $(\sigma * \tau) \in T^n(R)$, quindi $T^n(R)$ è chiuso sotto il prodotto $*$.

Sia $1 \in T^n(R)$ tale che per ogni $\underline{v} \in R^n$ si abbia $1(\underline{v}) = \underline{v}$. Allora per ogni $\tau \in T^n(R)$ e per ogni $\underline{v} \in R^n$ si ottiene

$$(\tau * 1)(\underline{v}) = \tau(1(\underline{v})) = \tau(\underline{v}) = 1(\tau(\underline{v})) = (1 * \tau)(\underline{v})$$

e quindi 1 è l'unità rispetto al prodotto $*$.

Siano ora $\sigma, \tau, \zeta \in T^n(R)$. Per ogni $\underline{v} \in R^n$ si ha che

$$\begin{aligned} ((\sigma * \tau) * \zeta)(\underline{v}) &= (\sigma * \tau)(\zeta(\underline{v})) \\ &= \sigma(\tau(\zeta(\underline{v}))) \\ &= \sigma((\tau * \zeta)(\underline{v})) \\ &= (\sigma * (\tau * \zeta))(\underline{v}) \end{aligned}$$

pertanto il prodotto $*$ è associativo e $\langle T^n(R), * \rangle$ è un monoide.

Sia $I^n(R)$ l'insieme di tutte le regole locali con raggio d'azione n, ovvero

$$I^n(R) = \{\phi \mid \phi : R^n \to R\}$$

e chiamiamo α la permutazione $(12 \dots n)$ vista come funzione $R^n \to R^n$ (lo *shift*[4]).

2.1 Lemma

Per ogni $\tau \in T^n(R)$ esistono (e sono uniche) $\phi_1, \dots, \phi_n \in I^n(R)$ tali che, per ogni $\underline{v} \in R^n$,

$$\tau(\underline{v}) = (\phi_1(\underline{v}), \phi_2(\alpha\underline{v}), \dots, \phi_n(\alpha^{n-1}\underline{v}))$$

Dimostrazione. Definiamo la funzione $\lambda : R^n \times \{1, \dots, n\} \to R$ in modo che per ogni $\underline{u} = (u_1, \dots, u_n) \in R^n$ e per ogni $j \leq n$ si abbia $\lambda(\underline{u}, j) = u_j$. Visto che per ogni $k \leq n$ si ha che $\lambda(\tau(\underline{v}), k) \in R$, la funzione $\lambda(\cdot, k)$ è un elemento di $I^n(R)$, e poiché $I^n(R)$ contiene tutte le possibili funzioni da R^n a R, esiste una funzione $\phi_k \in I^n(R)$ tale che, per ogni $\underline{v} \in R^n$,

$$\phi_k(\alpha^{k-1}\underline{v}) = \lambda(\tau(\underline{v}), k)$$

Supponiamo ora che ci sia una funzione ψ tale che per ogni $\underline{v} \in R^n$ si abbia

$$\psi(\alpha^{k-1}\underline{v}) = \lambda(\tau(\underline{v}), k)$$

Allora per ogni $\underline{v} \in R^n$ si ottiene $\psi(\alpha^{k-1}\underline{v}) = \phi_k(\alpha^{k-1}\underline{v})$ e quindi

$$\forall\ \underline{v} \in R^n\ (\psi(\underline{v}) = \phi_k(\underline{v}))$$

e pertanto $\psi = \phi_k$, che prova l'unicità di $\phi_1, \dots, \phi_n$. □

Con la notazione del lemma (2.1), per ogni $\tau \in T^n(R)$ definiamo la funzione $\Phi : T^n(R) \to (I^n(R))^n$ in modo che

$$\Phi(\tau) = (\phi_1, \dots, \phi_n)$$

[4]Cfr. pagina 12.

e per mezzo di Φ definiamo l'insieme $\mathcal{A}^n(R)$ delle funzioni $R^n \to R^n$ che possono essere descritte da una regola locale. Si ha che

$$\mathcal{A}^n(R) = \{\sigma \in T^n(R) \mid \exists \phi \in I^n(R)(\Phi(\sigma) = (\phi, \ldots, \phi))\}$$

e la biiezione $\theta : \mathcal{A}^n(R) \to I^n(R)$ data da $\theta(\sigma) = \phi$ che mette in relazione una funzione di $\mathcal{A}^n(R)$ con la regola locale che la descrive con intorni convenzionali. Vedremo nell'esempio (2.2) che l'estensione di una regola locale con intorni convenzionali è diversa dall'estensione della stessa regola con (r_0, r_1)-intorni. Diciamo perciò che se σ_{r_0,r_1} è l'estensione di ϕ con (r_0, r_1)-intorni si ha

$$\Phi_{r_0,r_1}(\sigma_{r_0,r_1}) = (\phi, \ldots, \phi)$$

e

$$\theta_{r_0,r_1}(\sigma_{r_0,r_1}) = \phi$$

Si noti che gli elementi τ di $\mathcal{A}^n(R)$ così definiti sono le trasformazioni dei vettori $\underline{v} = (v_1, \ldots, v_n)$ in cui la k-esima componente di $\tau(\underline{v})$ dipende da v_k e dalle $n-1$ componenti a destra di v_k, dove si considera per definizione $v_0 = v_n$ (ovvero il vettore $\underline{v}$ è arrangiato come un cerchio, con i due estremi v_1 e v_n contigui). In altre parole, le τ sono tutte le possibili evoluzioni successive con raggio d'azione n di un automa di lunghezza n.

2.2 Esempio

Consideriamo la famosa "regola 90 (nella notazione di Wolfram[5] si assegnano alle regole locali dei numeri interi che le identificano univocamente. Il numero associato ad una regola locale è anche chiamato il suo **numero di Wolfram**) applicata ad automi cellulari di lunghezza 3 definiti su $\mathbb{Z}_2$. Per questo esempio la distinzione di cui al punto (1.1) è importante: la regola 90 opera su $(1, 1)$-intorni ed è caratterizzata dal fatto che al tempo $t + 1$ il

[5] Cfr. [Wol83].

valore $a_i^{(t+1)}$ della i-esima cella dell'automa è uguale a

$$a_i^{(t+1)} = a_{i-1}^{(t)} + a_{i+1}^{(t)} \bmod 2$$

Chiamiamo $\sigma_{1,1}$ (gli indici denotano il raggio sinistro e destro dell'intorno considerato) la funzione $\mathbb{Z}_2^3 \to \mathbb{Z}_2^3$ che estende la regola 90. Si può anche descrivere ϕ come segue[6]:

$(1,1,1)$	$(1,1,0)$	$(1,0,1)$	$(1,0,0)$	$(0,1,1)$	$(0,1,0)$	$(0,0,1)$	$(0,0,0)$
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
0	1	0	1	1	0	1	0

Prendiamo ora un vettore in $\mathbb{Z}_2^3$, ad es. $(0,1,1)$. Volendo applicare $\sigma_{1,1}$ a $(0,1,1)$, bisogna innanzitutto notare che l'intorno che usiamo non è convenzionale: la sua forma infatti è la seguente

□ □ □

↓

□

e quindi

$$\sigma_{1,1}(\underline{v}) = (\phi(\alpha^2\underline{v}), \phi(\underline{v}), \phi(\alpha\underline{v}))$$

da cui

$$\begin{aligned}
\sigma_{1,1}(0,1,1) &= (\phi(\alpha^2(0,1,1)), \phi(\alpha^3(0,1,1)), \phi(\alpha(0,1,1))) = \\
&= (\phi((1\,3\,2)(0,1,1)), \phi((0,1,1)), \phi((1\,2\,3)(0,1,1))) = \\
&= (\phi(1,0,1), \phi(0,1,1), \phi(1,1,0)) = \\
&= (0,1,1)
\end{aligned}$$

Usiamo ora un intorno convenzionale di forma

□ □ □

↓

□

[6] Si noti che 01011010 è lo sviluppo binario di 90, che è poi la ragione del nome dato alla regola.

e sia σ l'estensione di ϕ con intorno convenzionale. Otteniamo in questo caso

$$\sigma(\underline{v}) = (\phi(\underline{v}), \phi(\alpha\underline{v}), \phi(\alpha^2\underline{v}))$$

che, applicata al vettore $(0,1,1)$, produce

$$\begin{aligned}
\sigma(0,1,1) &= (\phi(0,1,1), \phi(\alpha(0,1,1)), \phi(\alpha^2(0,1,1))) = \\
&= (\phi(0,1,1), \phi((1\,2\,3)(0,1,1)), \phi((1\,3\,2)(0,1,1))) = \\
&= (\phi(0,1,1), \phi(1,1,0), \phi(1,0,1)) = \\
&= (1,1,0)
\end{aligned}$$

Si osservi che

$$\sigma(0,1,1) = (1,1,0) = \alpha(0,1,1) = \alpha\sigma_{1,1}(0,1,1)$$

Questo fatto non è casuale, come si dimostrerà nel corollario (2.5). □

L'analisi fatta fino ad adesso non è sufficiente per applicare le regole per cui

raggio d'azione della regola di evoluzione $<$ lunghezza dell'automa

e quindi in particolare possiamo applicare la regola 90 solo ad automi di lunghezza 3. Tuttavia si proverà in seguito che le trasformazioni di automi cellulari in cui il raggio d'azione è minore della lunghezza dell'intorno formano un sottoinsieme di $\mathcal{A}^n(R)$.

2.2 Struttura moltiplicativa di $\mathcal{A}^n(R)$

Se restringiamo il prodotto $*$ a $\mathcal{A}^n(R)$ otteniamo un prodotto su $\mathcal{A}^n(R)$. Poiché $\mathcal{A}^n(R) \subset T^n(R)$ il prodotto $*$ è associativo su $\mathcal{A}^n(R)$. Inoltre la trasformazione unità $1 \in T^n(R)$ sta anche in $\mathcal{A}^n(R)$: sia $\lambda(\underline{v}, j)$ la funzione definita

nella dimostrazione del lemma (2.1) che ritorna la j-esima componente di $\underline{v}$. Si ha che

$$1(\underline{v}) = \underline{v} = (\lambda(\underline{v}, 1), \lambda(\alpha\underline{v}, 1), \ldots, \lambda(\alpha^{n-1}\underline{v}, 1))$$

e abbiamo visto[7] che $\lambda(\cdot, 1) \in I^n(R)$, dunque

$$\Phi(1) = (\lambda(\cdot, 1), \ldots, \lambda(\cdot, 1))$$

quindi $1 \in \mathcal{A}^n(R)$.

Per dotare $\langle \mathcal{A}^n(R), * \rangle$ di una struttura monoidale resta da mostrare che $\mathcal{A}^n(R)$ è chiuso sotto il prodotto $*$. La chiusura è un facile corollario del seguente teorema fondamentale, in cui si trovano condizioni necessarie e sufficienti all'appartenenza di una funzione generica all'insieme $\mathcal{A}^n(R)$.

2.3 Teorema

Per ogni intero positivo n e per ogni $\tau \in T^n(R)$ si ha che

$$(\forall\ \underline{v} \in R^n\ (\tau(\alpha\underline{v}) = \alpha\tau(\underline{v}))) \iff \tau \in \mathcal{A}^n(R)$$

Dimostrazione. Dimostriamo come premessa l'equivalenza dell'ipotesi con l'espressione $\forall\ h < n\ (\tau(\alpha^h\underline{v}) = \alpha^h\tau(\underline{v}))$. Visto che per ogni $\underline{v} \in R^n$ si ha $\tau(\alpha\underline{v}) = \alpha\tau(\underline{v})$,

$$\forall\ h < n\ (\tau(\alpha^h\underline{v}) = \tau(\alpha\alpha^{h-1}\underline{v}) = \alpha\tau(\alpha^{h-1}))$$

e quindi per induzione su k si ottiene

$$\forall\ h < n\ (\tau(\alpha^h\underline{v}) = \alpha^h\tau(\underline{v}))$$

Inoltre se $\forall\ h < n\ (\tau(\alpha^h\underline{v}) = \alpha^h\tau(\underline{v}))$ in particolare si ha $\tau(\alpha\underline{v}) = \alpha\tau(\underline{v})$.

($\Rightarrow$):Supponiamo adesso per assurdo che $\tau \notin \mathcal{A}^n(R)$ e proviamo che esistono un intero $h < n$ un vettore $\underline{v} \in R^n$ tali che

$$\tau(\alpha^h\underline{v}) \neq \alpha^h\tau(\underline{v})$$

[7] Cfr. dimostrazione del lemma (2.1) a pagina 20.

Sia dunque $\tau \notin \mathcal{A}^n(R)$. Per definizione non c'è alcuna funzione $\phi \in I^n(R)$ tale che $\Phi(\tau) = (\phi, \ldots, \phi)$, dunque esistono $\phi_1, \ldots, \phi_n \in I^n(R)$, esiste un vettore $\underline{v} \in R^n$ e esistono $j, k \leq n$ con $j \neq k$ tali che $\Phi(\tau) = (\phi_1, \ldots, \phi_n)$ e che $\phi_j \neq \phi_k$ su almeno un vettore nell'insieme $\{\alpha^{i-1}\underline{v} \mid 1 \leq i \leq n\}$; ovvero, se scriviamo $\tau(\underline{v})$ come

$$\tau(\underline{v}) = (\phi_1(\underline{v}), \phi_2(\alpha\underline{v}), \ldots, \phi_n(\alpha^{n-1}\underline{v}))$$

esistono $j \neq k$ entrambi minori o uguali a n tali che

$$\phi_j(\alpha^{k-1}\underline{v}) \neq \phi_k(\alpha^{k-1}\underline{v})$$

Ora si ha che $\phi_j(\alpha^{k-1}\underline{v})$ è la j-esima componente di $\tau(\alpha^{k-j}\underline{v})$ e $\phi_k(\alpha^{k-1}\underline{v})$ è la j-esima componente di $\alpha^{k-j}\tau(\underline{v})$, quindi $\tau(\alpha^{k-j}\underline{v}) \neq \alpha^{k-j}\tau(\underline{v})$, ma questo implica che

$$\exists\, h < n\ \exists\ \underline{v} \in R^n(\tau \notin \mathcal{A}^n(R) \rightarrow \tau(\alpha^h\underline{v}) \neq \alpha^h\tau(\underline{v}))$$

contro l'ipotesi che $\forall\ \underline{v} \in R^n\ (\alpha\tau(\underline{v}) = \tau(\alpha\underline{v}))$.

($\Leftarrow$): Se $\tau \in \mathcal{A}^n(R)$ esiste per definizione una funzione $\phi \in I^n(R)$ tale che $\Phi(\tau) = (\phi, \ldots, \phi)$. Se per ogni $\underline{v} \in R^n$ scriviamo

$$\tau(\underline{v}) = (\phi(\underline{v}), \phi(\alpha\underline{v}), \ldots, \phi(\alpha^{n-1}\underline{v}))$$

si ha che

$$\tau(\underline{v}) = (\phi(\alpha\underline{v}), \phi(\alpha^2\underline{v}), \ldots, \phi(\alpha^{n-1}\underline{v}), \phi(\underline{v})) = \alpha\tau(\underline{v})$$

□

Si noti che il teorema è analogo a dire che una funzione generica $R^n \rightarrow R^n$ è un'estensione di una regola locale se e solo se commuta con lo *shift*. In effetti sono in molti a usare questa condizione proprio come definizione[8] di automa cellulare.

[8]Cfr. fra gli altri [Kar96, Hed69]. Una fra le definizioni più "compatte di automa cellulare è *shift commuting discrete dynamical system*.

Possiamo ora completare la dimostrazione del fatto che $\langle \mathcal{A}^n(R), * \rangle$ è un monoide.

2.4 Corollario

Per ogni intero positivo n, $\mathcal{A}^n(R)$ è chiuso sotto il prodotto $*$.

Dimostrazione. Siano $\sigma, \tau \in \mathcal{A}^n(R)$ e mostriamo che $(\sigma * \tau) \in \mathcal{A}^n(R)$. In virtù del teorema (2.3) basta provare che per ogni vettore $\underline{v} \in R^n$ si ha $(\sigma * \tau)(\alpha \underline{v}) = \alpha(\sigma * \tau)(\underline{v})$. Ma per lo stesso teorema applicato prima a τ e poi a σ,

$$(\sigma * \tau)(\alpha \underline{v}) = \sigma(\tau(\alpha \underline{v})) = \sigma(\alpha \tau(\underline{v})) = \alpha\sigma(\tau(\underline{v})) = \alpha(\sigma * \tau)(\underline{v})$$

□

2.5 Corollario

Sia $\phi : R^n \to R$ una regola locale; sia $\sigma : R^n \to R^n$ la sua estensione con intorno convenzionale e sia $\sigma_{r_0,r_1} : R^n \to R^n$ la sua estensione su (r_0, r_1)-intorni con $r_0 + r_1 + 1 = n$. Allora si ha

$$\sigma = \alpha^{r_0} \sigma_{r_0,r_1}$$

Dimostrazione. Sia $\underline{v} = (v_1, \ldots, v_n) \in R^n$. La k-esima componente di $\sigma_{r_0,r_1}(\underline{v})$ è data da

$$\phi(v_{k-r_0}, \ldots, v_k, \ldots, v_{k+n-r_0-1})$$

(dove le operazioni sugli indici sono mod n) ovvero $\phi(\alpha^k \alpha^{n-r_0} \underline{v})$. Si ha quindi

$$\begin{aligned} \sigma_{r_0,r_1}(\underline{v}) &= (\phi(\alpha^{n-r_0} \underline{v}), \ldots, \phi(\alpha^{n-r_0} \alpha^k \underline{v})) \\ &= \sigma(\alpha^{n-r_0} \underline{v}) \\ &= \alpha^{n-r_0} \sigma(\underline{v}) \end{aligned}$$

per il teorema (2.3). Segue la tesi. □

L'implicazione più evidente di questo corollario è l'indipendenza di $\mathcal{A}^n(R)$ dalla convenzionalità o meno dell'intorno considerato: una funzione σ_{r_0,r_1}

che estende una regola locale ϕ con intorno non convenzionale può sempre essere scritta come composizione di α^{r_0} (che in quanto commutante con α è in $\mathcal{A}^n(R)$) e di σ, la funzione che estende ϕ con intorno convenzionale. Possiamo ora concludere l'esempio (2.2) dicendo che se σ e σ_{r_0,r_1} estendono la regola locale con numero di Wolfram 90 si ha

$$\sigma = \alpha\sigma_{r_0,r_1}$$

Dal corollario si evince anche che, se ϕ è una regola locale $R^n \to R$ e σ_{r_0,r_1} è la sua estensione su (r_0, r_1)-intorni con $r_0 + r_1 + 1 = n$, la sua applicazione ad un vettore $\underline{v}$ di R^n è data da

$$\sigma_{r_0,r_1}(\underline{v}) = (\phi(\beta\underline{v}), \phi(\alpha\beta\underline{v}), \ldots, \phi(\alpha^{n-1}\beta\underline{v}))$$

dove $\beta = \alpha^{n-r_0}$. Vista la "popolarità degli intorni con $r_0 = 1$ (causata in larga parte dagli articoli di Wolfram [Wol83], [WMO84]) useremo questo tipo di intorni più spesso degli intorni convenzionali.

Possiamo considerare risolto, almeno in teoria, il problema dell'evoluzione di un automa cellulare di lunghezza n in cui la regola locale abbia raggio d'azione n. Ad esempio, volendo calcolare l'evoluzione al tempo t ($t \in \mathbb{N}$) dell'automa 00100100 secondo una regola σ su $(1, 6)$-intorni, si possono compilare delle tavole di moltiplicazione[9] del monoide $\mathcal{A}_1^8(\mathbb{Z}_2)$ servendosi delle quali è immediato trovare τ tale che $\tau = \sigma^t$. Anche se nella maggior parte dei problemi pratici la computazione delle tavole è più dispendiosa dell'applicazione di σ per t volte, si possono ottenere dei vantaggi qualora fosse richiesto un numero sufficientemente elevato di evoluzioni di automi della stessa lunghezza.

[9] Vedi capitolo 3

2.3 Trasformazioni con raggio d'azione $< n$

Ci occupiamo ora delle trasformazioni di automi cellulari di lunghezza n in cui il raggio d'azione m è minore di n. In questo caso la convenzionalità o meno dell'intorno è importante, come dimostreremo in seguito. Usiamo quindi degli (r_0, r_1)-intorni tali che $r_0 + r_1 + 1 = m$, e consideriamo un intorno convenzionale come un $(0, m-1)$-intorno. La strategia di soluzione del problema sarà di prendere in considerazione delle sottostringhe di lunghezza m dell'automa. Dove in precedenza abbiamo usato l'intero vettore $\underline{v}$, "girandolo di un elemento alla volta per mezzo dello *shift* e trasformandolo poi nell' immagine di $\phi : R^n \to R$, sostituiremo adesso un vettore $\underline{u}$ formato dalle prime m componenti di $\underline{v}$. A questo scopo sia $\eta : R^n \to R^m$ la trasformazione lineare rappresentata dalla matrice di dimensioni $m \times n$

$$\eta = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 & 0 & \cdots & 0 \end{pmatrix}$$

che servirà ad "estrarre le prime m componenti di un vettore di lunghezza n. Si noti infatti che se $\underline{v} = (v_1, \ldots, v_n)$ allora $\eta(\underline{v}) = (v_1, \ldots, v_m)$. Si ricorda che per ogni $\sigma \in \mathcal{A}^n(R)$ e per ogni $\underline{v} \in R^n$ si ha che, usando (r_0, r_1)-intorni

$$\sigma(\underline{v}) = (\phi(\beta\underline{v}), \ldots, \phi(\beta\alpha^{n-1}\underline{v}))$$

dove $\phi = \theta_{0,m-1}(\sigma) \in I^n(R)$, α è la permutazione $(12\ldots n)$, cioè lo *shift*, e $\beta = \alpha^{n-r_0}$. L'insieme $\mathcal{B}^n_{r_0,r_1}(R)$ di tutte le trasformazioni cellulari con raggio d'azione m su (r_0, r_1)-intorni è definito da

$$\mathcal{B}^n_{r_0,r_1}(R) = \{\sigma \in \mathcal{A}^n(R) \mid \exists\ \psi \in I^m(R)(\theta_{r_0,r_1}(\sigma) = \psi\eta)\}$$

e quindi per ogni $\sigma \in \mathcal{B}^n_{r_0,r_1}(R)$ e per ogni $\underline{v} \in R^n$ si ha che

$$\sigma(\underline{v}) = (\psi\eta(\beta\underline{v}), \ldots, \psi\eta(\beta\alpha^{n-1}\underline{v}))$$

Si definiscano inoltre, analogamente a quanto fatto in precedenza, per ogni $\sigma \in \mathcal{B}^n_{r_0,r_1}(R)$, la funzione $\Phi_{\eta,r_0,r_1} : \mathcal{B}^n_{r_0,r_1}(R) \to (I^m(R))^n$ data da

$$\Phi_{\eta,r_0,r_1}(\sigma) = (\psi, \ldots, \psi)$$

e la biiezione $\theta_{\eta,r_0,r_1} : \mathcal{B}^n_{r_0,r_1}(R) \to I^m(R)$ data da

$$\theta_{\eta,r_0,r_1}(\sigma) = \psi$$

Il fatto che $\mathcal{B}^n_{r_0,r_1}(R) \subseteq \mathcal{A}^n(R)$ è evidente dalla definizione. In generale, tuttavia, $\mathcal{B}^n_{r_0,r_1}(R)$ non è un sottomonoide di $\mathcal{A}^n(R)$, come si dimostra nel seguente controesempio: si consideri la trasformazione $\sigma \in \mathcal{B}^5_{0,2}(\mathbb{Z}_2)$ con $m = 3$, operante su automi di lunghezza $n = 5$ definiti sull'anello $\mathbb{Z}_2$, tale che $\theta_\eta(\sigma) = \psi$ dove $\psi : (\mathbb{Z}_2)^3 \to \mathbb{Z}_2$ è data da

$$\psi(\underline{v}) = \begin{cases} 1 & \text{se } \underline{v} = (0,0,1) \\ 0 & \text{altrimenti} \end{cases}$$

Sia $\underline{u} = (0,0,0,0,1) \in (\mathbb{Z}_2)^5$. Si ha che $\sigma^2(\underline{u}) = (1,0,0,0,0)$, e chiaramente $\theta_{\eta,0,2}(\sigma^2)$ non è definita poiché dovrebbe mandare $(0,0,0)$ sia in 1 (nel caso di u_1, u_2, u_3) che in 0 (nel caso di u_2, u_3, u_4); di conseguenza $\sigma^2 \notin \mathcal{B}^5_{0,2}(\mathbb{Z}_2)$. Pertanto l'insieme considerato non è chiuso sotto il prodotto $*$ e dunque non è un monoide.

Mostriamo ora che al variare di r_0, r_1 l'insieme $\mathcal{B}^n_{r_0,r_1}(R)$ varia: Si consideri la regola 90 applicata ad automi di lunghezza 4. Sia $\sigma_{1,1}$ la sua estensione su $(1,1)$-intorni, in modo che $\sigma_{1,1} \in \mathcal{B}^4_{1,1}(\mathbb{Z}_2)$: si ha che

$$\sigma_{1,1}((0,0,0,0)) = (0,0,0,0) \quad (2)$$

$$\sigma_{1,1}((0,0,0,1)) = (1,0,1,0) \quad (3)$$

È facile mostrare che $\sigma_{1,1} \notin \mathcal{B}^4_{0,2}(\mathbb{Z}_2)$: Se $\sigma_{1,1}$ estendesse una regola locale ϕ con intorno convenzionale (cioè avente $r_0 = 0, r_1 = 2$), si avrebbe $\phi(0,0,0) = 0$ per (2) e $\phi(0,0,0) = 1$ per (3), e quindi ϕ non sarebbe ben definita.

Per stabilire se una trasformazione $\sigma \in \mathcal{A}^n(R)$ appartiene a $\mathcal{B}^n_{r_0,r_1}(R)$ o meno, definiamo una funzione $\chi : R^n \to \mathcal{P}(R^n)$, dove $\mathcal{P}$ indica l'insieme potenza, data da

$$\chi(\underline{v}) = \{\underline{u} \in R^n \mid \eta(\underline{u}) = \eta(\underline{v})\}$$

In pratica χ riunisce tutti i vettori di R^n che hanno le prime m componenti uguali.

Per definizione si ha che σ sta in $\mathcal{B}^n_{r_0,r_1}(R)$ se e solo se esiste una funzione $\psi \in I^m(R)$ tale che per ogni $\underline{v} \in R^n$ valga la relazione

$$\phi(\underline{v}) = \psi(\eta(\underline{v}))$$

dove $\phi = \theta(\sigma)$. Servendoci della funzione χ definita poc'anzi possiamo fornire una condizione necessaria e sufficiente su ϕ perché questo accada: σ sta in $\mathcal{B}^n_{r_0,r_1}(R)$ se e solo se la sua regola locale ϕ è tale che per ogni coppia di vettori $\underline{u}$ e $\underline{v}$ con le prime d componenti uguali si abbia $\phi(\underline{u}) = \phi(\underline{v})$.

2.6 Proposizione

Data $\phi \in I^n(R)$, esiste $\psi \in I^m(R)$ tale che per ogni $\underline{v} \in R^n$ si abbia

$$\phi(\underline{v}) = \psi(\eta(\underline{v}))$$

se e solo se è verificata la seguente condizione

$$\forall\, \underline{v} \in R^n \;\forall\, \underline{u} \in \chi(\underline{v})\; (\phi(\underline{u}) = \phi(\underline{v}))$$

Dimostrazione. ($\Leftarrow$): Sia ξ la matrice $n \times m$ data da

$$\xi = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}$$

Per ogni $\underline{w} \in R^m$ si definisca

$$\psi(\underline{w}) = \phi(\xi(\underline{w}))$$

Che ψ sia ben definita è chiaro: se $\underline{w} = \underline{w'} \in R^d$ allora

$$\psi(\underline{w}) = \phi(\xi(\underline{w})) = \phi(\xi(\underline{w'})) = \psi(\underline{w'})$$

Rimane da mostrare che $\phi = \psi\eta$. Notiamo a questo punto che

$$\xi\eta = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \end{pmatrix}$$

dove il blocco $m \times m$ superiore sinistro è la matrice identità $m \times m$ e tutti gli altri valori sono zeri. Pertanto, se per ogni $\underline{v} \in R^n$ scriviamo $\underline{v} = (v_1, \ldots, v_n)$ abbiamo che

$$\xi\eta(\underline{v}) = (v_1, \ldots, v_m, 0, \ldots, 0)$$

Chiamiamo $\underline{u} = (v_1, \ldots, v_d, 0, \ldots, 0)$. Si ha che $\underline{u} \in \chi(\underline{v})$ e quindi per ipotesi $\phi(\underline{u}) = \phi(\underline{v})$ da cui

$$\psi(\eta(\underline{v})) = \phi(\xi\eta(\underline{v})) = \phi(\underline{u}) = \phi(\underline{v})$$

e quindi $\phi = \psi\eta$.

($\Rightarrow$): Se $\underline{u}, \underline{v} \in \chi(\underline{v})$ si ha che $\eta(\underline{u}) = \eta(\underline{v})$ e quindi per ogni $\underline{v} \in R^n$ e per ogni $\underline{u} \in \chi(\underline{v})$ si ottiene

$$\phi(\underline{v}) = \psi(\eta(\underline{v})) = \psi(\eta(\underline{u})) = \phi(\underline{u})$$

□

2.4 Prodotto di trasformazioni in $\mathcal{A}^n(R)$

Date due trasformazioni σ_1 e σ_2 in $\mathcal{A}^n(R)$ analizziamo ora il problema di calcolare la trasformazione τ in $\mathcal{A}^n(R)$ tale che $\tau = \sigma_1 * \sigma_2$. È evidente che τ è completamente definita quando si sa qual è il suo effetto su ognuno dei vettori $\underline{v} \in R^n$. Tuttavia abbiamo visto nel teorema (2.3) che τ sta in $\mathcal{A}^n(R)$ se e solo se per ogni $\underline{v} \in R^n$ si ha

$$\tau(\alpha\underline{v}) = \alpha\tau(\underline{v})$$

Ne segue che per determinare τ è sufficiente calcolare l'effetto di τ su ognuno dei rappresentanti delle orbite di R^n sotto α, dato che i vettori ottenuti applicando τ agli altri elementi delle orbite saranno permutazioni cicliche di τ applicato al rappresentante dell'orbita.

2.7 Esempio

Sia σ_1 la trasformazione che estende

$(1,1,1)$	$(1,1,0)$	$(1,0,1)$	$(1,0,0)$	$(0,1,1)$	$(0,1,0)$	$(0,0,1)$	$(0,0,0)$
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
0	1	0	1	1	0	1	0

e σ_2 la trasformazione che estende

$(1,1,1)$	$(1,1,0)$	$(1,0,1)$	$(1,0,0)$	$(0,1,1)$	$(0,1,0)$	$(0,0,1)$	$(0,0,0)$
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
0	0	0	0	0	0	1	0

con intorni convenzionali. Dividiamo ora $\mathbb{Z}_2^3$ in orbite sotto $\alpha = (1,2,3)$. È facile vedere che le orbite sono

$$\begin{aligned} O_1 &= \{(0,0,0)\} \\ O_2 &= \{(0,0,1),(1,0,0),(0,1,0)\} \\ O_3 &= \{(0,1,1),(1,0,1),(1,1,0)\} \\ O_4 &= \{(1,1,1)\} \end{aligned}$$

e scegliamo (arbitrariamente) i loro rappresentanti

$$\begin{aligned} \underline{v}_1 &= (0,0,0) \\ \underline{v}_2 &= (0,0,1) \\ \underline{v}_3 &= (0,1,1) \\ \underline{v}_4 &= (1,1,1) \end{aligned}$$

Calcoliamo ora l'effetto di $\tau = \sigma_1 * \sigma_2$ su ognuno dei rappresentanti $\underline{v}_i$.

$$\begin{aligned} \tau(\underline{v}_1) &= (0,0,0) \\ \tau(\underline{v}_2) &= (1,1,0) \\ \tau(\underline{v}_3) &= (0,0,0) \\ \tau(\underline{v}_4) &= (0,0,0) \end{aligned}$$

Da qui possiamo scrivere la localizzazione di τ (sempre con intorni convenzionali) come

$(1,1,1)$	$(1,1,0)$	$(1,0,1)$	$(1,0,0)$	$(0,1,1)$	$(0,1,0)$	$(0,0,1)$	$(0,0,0)$
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
0	0	0	0	0	1	1	0

□

Come si può arguire facilmente da questo esempio, calcolare il prodotto di elementi di $\mathcal{A}^n(R)$ non è affatto banale, e una gran parte della difficoltà sta nel determinare le orbite. Il problema verrà trattato in termini generali nel capitolo 4.

3 Rappresentazioni Numeriche

Solitamente gli algoritmi di manipolazione degli automi cellulari applicano ad una data stringa di cifre (l'"automa cellulare propriamente detto) una trasformazione basata su delle regole locali per un certo numero di volte. Il calcolo è molto veloce e le difficoltà teoriche degli algoritmi sono poche. In molti casi (quando le stringhe di partenza sono lunghe) questo approccio di "forza bruta è l'unico possibile. In questa sede, tuttavia, intendiamo sviluppare degli algoritmi basati sulla teoria introdotta nel capitolo precedente che in taluni casi presentano dei vantaggi sulla tecnica della "forza bruta.

I calcoli verranno effettuati sulla trasformazione anziché sull'automa cellulare: questo permetterà un risparmio computazionale nel caso in cui si debba applicare la medesima trasformazione a parecchie stringhe di partenza che siano tuttavia di lunghezza molto ridotta. Una situazione favorevole sarebbe per esempio il calcolo dell'evoluzione di un centinaio di stringhe di sette cifre binarie dopo cento passi.

Data una trasformazione σ che vogliamo applicare t volte, disegneremo un algoritmo che calcoli esplicitamente la trasformazione τ tale che $\tau = \sigma^t$. Abbiamo prima di tutto bisogno di descrivere le trasformazioni del monoide $\mathcal{A}^n(R)$ (vedi il capitolo 2) in modo che possano essere manipolate da un processore.

3.1 Rappresentazioni locali

Abbiamo visto che una trasformazione σ in $\mathcal{A}^n(R)$ agisce su una stringa di n elementi dell'anello finito R (ovvero su un vettore di R^n che chiameremo $\underline{v} = (v_1, \ldots, v_n)$) nel seguente modo: alla posizione i-esima del vettore $\sigma(\underline{v})$ c'è un elemento ϕ_i di R che dipende da un intorno di v_i, con la convenzione

che la 0-esima componente di $\underline{v}$ è uguale alla n-esima e la $(n+1)$-esima è uguale alla prima. Nel capitolo 2 abbiamo visto che per intorni convenzionali

$$\phi_i = \phi(\alpha^{i-1}\underline{v})$$

dove α è lo *shift* e abbiamo provato[10] che σ è completamente determinata dal tipo di intorno (e dai suoi raggi sinistro e destro se l'intorno non è convenzionale) e dalla funzione ϕ, che a sua volta è completamente determinata quando si conoscono i suoi valori su tutti i vettori di R^n. Come è già stato detto in (1.1), in questo capitolo i parametri r_0 e r_1 (raggi sinistro e destro di un intorno non convenzionale) sono importanti; a parte gli intorni convenzionali, per cui $r_0 = 0$ e $r_1 = n-1$, useremo spesso degli $(1, n-2)$-intorni, o se il raggio d'azione di ϕ è $m < n$, degli $(1, m-2)$-intorni.

Poiché R è un anello finito l'ordine lessicografico sui vettori di R^n è un buon ordine. Un'espressione che definisce l'effetto locale di σ si ottiene scrivendo tutti i vettori di R^n in ordine lessicografico decrescente e applicando ad ϕ ad ognuno di essi. Per esempio, se $R^n = \mathbb{Z}_2^3$ la lista dei vettori in ordine decrescente è

$$\{(1,1,1),(1,1,0),(1,0,1),(1,0,0),(0,1,1),(0,1,0),(0,0,1),(0,0,0)\} \quad (4)$$

Ora applichiamo ϕ ai vettori della lista e otteniamo un vettore

$$(\phi_7, \phi_6, \phi_5, \phi_4, \phi_3, \phi_2, \phi_1, \phi_0)$$

appartenente a $\mathbb{Z}_2^8$ che determina completamente σ. Chiameremo il vettore così costruito **rappresentazione locale** di σ. Si osservi che la convenzione di numerazione decrescente degli elementi del vettore è utile perché l'indice è la conversione decimale del numero in base $|R|$ ottenuto accostando gli elementi dei vettori nella lista (4). La lunghezza di una rappresentazione

[10] Cfr. lemma (2.1)

locale è evidentemente $|R|^{|R|^n}$ (in altre parole le rappresentazioni locali sono elementi dell'insieme $R^{|R|^n}$) e dunque si ha che

$$|\mathcal{A}^n(R)| = |R|^{|R|^n}$$

Abbiamo fatto uso delle rappresentazioni locali in modo implicito nel capitolo 2, negli esempi (2.2) e (2.7).

3.2 Rappresentazioni decimali

La rappresentazione locale è ottima per la manipolazione dei dati da parte del processore. Una rappresentazione più adatta per facilitare l'*input* dei dati da parte dell'utente è la rappresentazione decimale. In pratica la rappresentazione locale può essere vista come un numero s di $|R|^n$ cifre in base $|R|$. La **rappresentazione decimale** è la conversione di s in base 10 (si veda la nota a piè di pagina 22).

Data la rappresentazione locale

$$\underline{\phi} = (\phi_{|R|^n-1}, \phi_{|R|^n-2}, \ldots, \phi_0)$$

della trasformazione σ, la corrispondente rappresentazione decimale è data da

$$s = \sum_{i=0}^{|R|^n-1} \phi_i |R|^i$$

Per l'applicazione al calcolatore useremo un convertitore generico fra sviluppi numerici dalla base `R` alla base 10.

```
unsigned long vett2dec(unsigned int R, unsigned int veclen, \
                       unsigned int *v)
{
        unsigned int i;
```

```
        unsigned long s, p;

        s = 0;
        p = 1;
        for(i = 0; i < veclen; i++)
                s = s + (*(v + i)) * power(R, i);

        return s;
}
```

dove `*v` è un puntatore a un vettore di `R^n` valori che contiene le componenti della rappresentazione locale $\underline{\phi}$, e `veclen` è la lunghezza del vettore in questione; in questo caso `veclen` = `R^n`. Per la definizione della funzione `power()` vedi la sezione 3.4.3.

La funzione `vett2dec` viene eseguita in $l =$ `veclen` passi, e per ogni passo viene chiamata `power`, che richiede $\log_2 i$ passi. In tutto, abbiamo

$$\sum_{i=1}^{l} \log_2 i = \log_2(\prod_{i=1}^{l} i) = \log_2(l!)$$

passi.

Viceversa, data una rappresentazione decimale s la i-esima componente della corrispondente rappresentazione locale è

$$\phi_i = \frac{s - (s \mod |R|^{|R|^n - i})}{|R|^{|R|^n - i}} \mod |R|$$

Anche in questo caso useremo un convertitore generico dalla base 10 alla base `R`.

```
unsigned int *dec2vett(unsigned int R, unsigned int veclen, \
                       unsigned long s)
```

```
{
        unsigned int i;
        unsigned int *v;

        v = (unsigned int *) malloc(veclen * sizeof(int));

        for(i = 1; i <= veclen; i++)
                *(v + i - 1) = (s / power(R, veclen - i)) % R;

        return v;
}
```

L'analisi dei passi richiesti a completare `dec2vett` è la stessa fatta più sopra per `vett2dec`.

3.3 Convenzioni numeriche per $\mathcal{B}^n_{r_0,r_1}(R)$

Le trasformazioni considerate fino ad ora appartengono al monoide $\mathcal{A}^n(R)$, che significa che lo sviluppo temporale di ogni singolo elemento dell'automa cellulare di partenza dipende da tutti gli elementi del vettore che lo rappresenta. Questa è in realtà una situazione insolita: capita molto più spesso che l'intorno di interesse sia più corto dell'automa stesso. Nel capitolo 2 abbiamo descritto algebricamente una trasformazione del genere per mezzo del sottoinsieme $\mathcal{B}^n_{r_0,r_1}(R)$ di $\mathcal{A}^n(R)$ (dove $r_0 + r_1 + 1 = m$, la lunghezza dell'intorno); tuttavia, mentre $\mathcal{A}^n(R)$ è una struttura chiusa rispetto al prodotto di trasformazioni, $\mathcal{B}^n_{r_0,r_1}(R)$ non lo è. Per questo motivo non è agevole fare i calcoli direttamente su elementi di $\mathcal{B}^n_{r_0,r_1}(R)$, ma è necessario prima esprimerli come elementi di $\mathcal{A}^n(R)$. In teoria, abbiamo visto nel capitolo 2 che il passaggio da $\mathcal{B}^n_{r_0,r_1}(R)$ ad $\mathcal{A}^n(R)$ si ottiene tramite il prodotto di matrici, che però

ha un costo computazionale eccessivo. Anche se non direttamente necessario ai fini di computare il prodotto di trasformazioni cellulari, svilupperemo un algoritmo per effettuare la trasformazione da una rappresentazione locale in $\mathcal{B}^n_{r_0,r_1}(R)$ ad una in $\mathcal{A}^n(R)$.

Notiamo a questo punto che una rappresentazione locale di trasformazioni in $\mathcal{B}^n_{r_0,r_1}(R)$ è, dal punto di vista della notazione, uguale ad una rappresentazione locale di una trasformazione in $\mathcal{A}^m(R)$. Ad esempio, la rappresentazione locale di una trasformazione $\sigma \in \mathcal{B}^5_{0,2}(\mathbb{Z}_2)$ avrà la forma

$$(\psi_7, \psi_6, \psi_5, \psi_4, \psi_3, \psi_2, \psi_1, \psi_0)$$

perché σ è completamente determinata dai valori che la sua localizzazione $\psi = \theta_{\eta,0,2}(\sigma)$ assume sui vettori di $\mathbb{Z}_2^m$, dove $m = 0 + 2 + 1 = 3$.

3.1 Esempio

Facciamo un esempio concreto per illustrare il principio che useremo per formulare le leggi di conversione fra rappresentazioni locali di $\mathcal{B}^n_{r_0,r_1}(R)$ e di $\mathcal{A}^n(R)$. Consideriamo la regola locale

$$\underline{\psi} = (0, 0, 0, 0, 0, 0, 1, 0) \tag{5}$$

e la sua estensione σ in $\mathcal{B}^4_{0,2}(\mathbb{Z}_2)$ su intorni convenzionali di lunghezza 3 (è la trasformazione che manda la tripla $(0, 0, 1)$ in 1 e tutte le altre in 0, applicata a stringhe di quattro cifre binarie). Vogliamo trovare la rappresentazione locale di σ calcolandola però su intorni convenzionali di lunghezza 4. Dividiamo $\mathbb{Z}_2^4$ in orbite sotto $\langle(1234)\rangle$ e calcoliamo $\sigma(\underline{v}_j)$ per ognuno dei rappresentanti

delle orbite $\underline{v}_j$.

$$\begin{aligned}
\sigma(1,1,1,1) &= (0,0,0,0)\\
\sigma(0,1,1,1) &= (0,0,0,0)\\
\sigma(0,0,1,1) &= (1,0,0,0)\\
\sigma(0,1,0,1) &= (0,0,0,0)\\
\sigma(0,0,0,1) &= (0,1,0,0)\\
\sigma(0,0,0,0) &= (0,0,0,0)
\end{aligned}$$

Ora facendo uso del teorema (2.3) possiamo calcolare σ su tutti i vettori di $\mathbb{Z}_2^4$, e ciò permette di esplicitare la rappresentazione locale della funzione $\phi = \theta(\sigma)$ associata a σ.

$$\begin{aligned}
\phi(1,1,1,1) &= 0\\
\phi(1,1,1,0) &= 0\\
\phi(1,1,0,1) &= 0\\
\phi(1,1,0,0) &= 0\\
\phi(1,0,1,1) &= 0\\
\phi(1,0,1,0) &= 0\\
\phi(1,0,0,1) &= 0\\
\phi(1,0,0,0) &= 0\\
\phi(0,1,1,1) &= 0\\
\phi(0,1,1,0) &= 0\\
\phi(0,1,0,1) &= 0\\
\phi(0,1,0,0) &= 0\\
\phi(0,0,1,1) &= 1\\
\phi(0,0,1,0) &= 1
\end{aligned}$$

$$\begin{aligned}\phi(0,0,0,1) &= 0\\ \phi(0,0,0,0) &= 0\end{aligned}$$

La rappresentazione locale di ϕ è

$$\underline{\phi} = (0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0) \tag{6}$$

Un confronto fra le espressioni (5) e (6) rivela che

$$\underline{\phi} = (\phi_7, \phi_7, \phi_6, \phi_6, \ldots, \phi_0, \phi_0)$$

□

In una situazione più generale, se conosciamo la rappresentazione locale $\psi = \theta_{\eta,r_0,r_1}(\sigma)$ dove $\sigma \in \mathcal{B}^n_{r_0,r_1}(R)$ e $\underline{\psi}$ è il vettore $(\psi_{|R|^m-1}, \ldots, \psi_0)$, sappiamo il valore che ψ assume su ogni vettore $(a_1, \ldots, a_m)$ in R^m dove $m = r_0+r_1+1$. Vogliamo conoscere il valore di $\phi = \theta_{r_0,r_1}(\sigma)$ su un vettore generico

$$\underline{v} = (a_1, \ldots, a_m, a_{m+1}, \ldots, a_n)$$

di R^n. Dalla sezione 2.3 sappiamo che se teniamo fissi $a_1, \ldots, a_m$ e facciamo variare le altre componenti di $\underline{v}$ abbiamo che per ognuna delle $|R|^{n-m}$ successioni di valori $(a_{m+1}, \ldots, a_n)$ si ottiene

$$\phi(a_1, \ldots, a_m, a_{m+1}, \ldots, a_n) = \psi(a_1, \ldots, a_m)$$

Considerando che nella lista dei vettori di R^n in ordine lessicografico decrescente tutti i vettori che cominciano con $a_1, \ldots, a_m$ sono contigui, risulta la seguente regola

$$\underline{\phi} = (\underbrace{\psi_{|R|^m-1}, \ldots, \psi_{|R|^m-1}}_{|R|^{n-m}}, \ldots, \underbrace{\psi_0, \ldots, \psi_0}_{|R|^{n-m}})$$

3.2 Esempio

La conversione ad $\mathcal{A}^3(\mathbb{Z}_3)$ della rappresentazione locale in $\mathcal{B}^3_{0,1}(\mathbb{Z}_3)$

$$(1,2,2,1,0,2,0,1,0)$$

è data dal vettore

$$(1, 1, 1, 2, 2, 2, 2, 2, 2, 1, 1, 1, 0, 0, 0, 2, 2, 2, 0, 0, 0, 1, 1, 1, 0, 0, 0)$$

□

L'algoritmo seguente esegue la conversione appena descritta.

```
unsigned int *BT2AT(unsigned int R, unsigned int n, \
                    unsigned int r0, unsigned int r1, \
                    unsigned int *psi)
{
        unsigned int m, i, j, l;
        unsigned int *phi;

        m = r0 + r1 + 1;
        phi = (unsigned int *) malloc(power(R, n) * \
              sizeof(int));
        for(i = 0; i < power(R, m); i++)
        {
                l = i * power(R, n - m);
                for(j = 0; j < power(R, n - m); j++)
                        *(phi + l + j) = *(psi + i);
        }

        return phi;
}
```

Come è già stato detto, l'algoritmo descritto in questa sezione non è strettamente necessario per calcolare il prodotto di due trasformazioni, che è

l'obbiettivo principale di questo capitolo; tuttavia i concetti appena esposti verranno usati per decidere se, data una rappresentazione di una trasformazione in $\mathcal{A}^n(R)$, questa appartenga a un sottoinsieme $\mathcal{B}^n_{r_0,r_1}(R)$ con $r_0 + r_1 + 1 = m < n$.

3.4 Prodotto di trasformazioni

Mediante il concetto di rappresentazione locale e decimale possiamo tentare di sviluppare un algoritmo per calcolare il prodotto di due trasformazioni. Abbiamo bisogno di una funzione che trovi i rappresentanti delle orbite di R^n sotto $\langle\alpha\rangle$, di una funzione che, dati un vettore $\underline{v} \in R^n$ e una trasformazione $\sigma \in \mathcal{A}^n(R)$, calcoli $\sigma(\underline{v})$ e infine di una che le coordini entrambe per produrre il risultato.

Gli algoritmi usati passeranno spesso da rappresentazioni locali a rappresentazioni decimali per tutti i tipi di vettori usati, a seconda della convenienza.

3.4.1 Divisione in orbite

L'idea guida della funzione è la seguente:

- Sia `orbit` un puntatore ad una array di strutture in cui verranno posti i rappresentanti delle orbite (in realtà lo sviluppo decimale dei vettori) e i loro periodi. In pratica si ha

```
struct orb
{
  unsigned long rep;
  /* in cui c'e' la rappresentazione
```

```
        decimale del rappresentante
        dell'orbita */
    unsigned int period;
    /* in cui c'e' il periodo */
};
struct orb *orbit;
```

- Sia `l` un puntatore ad un vettore di lunghezza $|R|^n$ inizializzato a $(0, \ldots, 0)$
- Inizializza i contatori `c` e `cv` a 0
- Ripeti ...
  - Se `l[c] == 0` allora
    - Sia `*(orbit + cv).rep = c`
    - Sia `c1 = c`
    - Sia `l[c1] = l[c1] + 1`
    - Sia `c1` lo sviluppo decimale dell'applicazione di α al vettore che rappresenta lo sviluppo in base $|R|$ di `c`
    - Inizializza il contatore `i` a 1
    - Ripeti ...
      - Sia `l[c1] = l[c1] + 1`
      - `c1` lo sviluppo decimale dell'applicazione di α al vettore che rappresenta lo sviluppo in base $|R|$ di `c`
      - Sia `i = i + 1`
    - ... fintantoché `c` è diverso da `c1`
    - Sia `*(orbit + cv).period = i`
    - Sia `cv = cv + 1`

  - Sia `c = c + 1`

- ... fintantoché `c` $<$ `R^n`
- `cv` contiene il numero di orbite

Introduciamo a questo punto una funzione `shift_vector` che applichi la permutazione α al vettore `v`. L'argomento `unsigned int times` contiene il numero di volte per cui si deve applicare α.

```
unsigned long shift_vector(unsigned int R, unsigned int n, \
                           unsigned int *v, unsigned int times)
{
        unsigned long c1;
        unsigned int i, m;

        times = times % n;
        c1 = 0;
        for(i = 0; i < n; i++)
        {
                m = (n + times - i - 1) % n;
                c1 = c1 + (*(v + m)) * power(R, i);
        }

        return c1;
}
```

La funzione `shift_vector` viene eseguita in un numero di passi dell'ordine di $\log_2(n!)$ (vedi l'analisi fatta più sopra per `vett2dec`).

Proponiamo ora la funzione `find_orbits` che implementa l'algoritmo discusso in questo paragrafo.

```
struct orb
{
        unsigned long rep;
        unsigned int period;
};

struct orb *find_orbits(unsigned int R, unsigned int n, \
                        unsigned int *num_orbits)
{
        unsigned long c, c1, cv, t, t1;
        unsigned int i;
        struct orb *orbit;
        unsigned int *l, *v;

        c = 0;
        c1 = 0;
        cv = 0;
        t = power(R, n);
        orbit = (struct orb *) malloc(t * sizeof(struct orb));
        l = (unsigned int *) malloc(t * sizeof(int));
        for(i = 0; i < t; i++)
                *(l + i) = 0;
        v = (unsigned int *) malloc(t * sizeof(int));

        while(c < t)
        {
                if(*(l + c) == 0)
                {
                        (*(orbit + cv)).rep = c;
```

```
                        c1 = c;
                        (*(l + c1))++;
                        v = dec2vett(R, n, c1);
                        c1 = shift_vector(R, n, v, 1);
                        i = 1;
                        while(c1 != c)
                        {
                                (*(l + c1))++;
                                i++;
                                c1 = shift_vector(R, n, v, i);
                        }
                        (*(orbit + cv)).period = i;
                        cv++;
                }
                c++;
        }
        *num_orbits = cv;

        return orbit;
}
```

È possibile contare il numero di passi p richiesto da `find_orbits` notando che `shift_vector` viene chiamata un numero di volte pari al numero di vettori in R^n e `dec2vett` viene chiamata tante volte quante sono le orbite di R^n. Si può stimare il numero di orbite di R^n in $\frac{|R|^n}{\bar{n}}$ (dove $\bar{n} = \frac{n+1}{2}$) è la media aritmetica dei primi n numeri), dato che per la proposizione (4.1) le dimensioni delle orbite dividono n. Pertanto p sarà dell'ordine di $|R|^n(\log_2(n!) + \frac{2}{n+1})$.

3.4.2 Applicazione di σ ad un vettore di R^n

La funzione `apply_rule` ha come argomenti le dimensioni dell'anello R, la lunghezza dell'automa cellulare n, dei raggi sinistro e destro r_0 e r_1, una rappresentazione decimale della trasformazione σ (vista come elemento di $\mathcal{B}^n_{r_0,r_1}(R)$) e una rappresentazione decimale del vettore $\underline{v}$ che descrive l'automa cellulare da trasformare, e ritorna una rappresentazione decimale dell'automa cellulare trasformato. L'algoritmo usato è una trasposizione in linguaggio C dei metodi teorici usati nella sezione 2.3 per definire $\mathcal{B}^n_{r_0,r_1}(R)$.

```
unsigned long apply_rule(unsigned int R, unsigned int n, \
                         unsigned int r0, unsigned int r1, \
                         unsigned long rule, \
                         unsigned long vector)
{
        unsigned int i, j, l, m, md;
        unsigned int *v;
        unsigned long t, c, c1, p;

        i = 0;
        t = power(R, n);
        d = r0 + r1 + 1;
        v = dec2vett(R, n, vector);
        c = 0;
        for(i = 0; i < n; i++)
        {
                l = n - i - 1;
                c1 = 0;
                for(j = 0; j < m; j++)
```

```
                {
                        md = mod(l - r0 + j, n);
                        /* Per la definizione di mod()
                           vedi piu' sotto */
                        c1 = c1 + (*(v + md)) * \
                             power(R, m - 1 - j);   //(1)
                }
                p = (rule / power(R, c1)) % R;      //(2)
                c = c + p * power(R, i);            //(3)
        }

        return c;
}
```

Il numero di passi dovuto a (1) è $n \log_2(m!)$, quello dovuto a (2) si può stimare nell'ordine di $n \log_2(|R|^n)$ e quello dovuto a (3) è $\log_2(n!)$. In tutto l'algoritmo richiede $\log_2(|R|^{(n^2)}(n!)(m!)^n)$ passi per la sua esecuzione.

La seguente è un'implementazione leggermente modificata della funzione matematica $a \mod n$ che ritorna sempre un intero positivo.

```
unsigned int mod(int a, unsigned int n)
{
        unsigned int a3;
        long a1, a2;

        a1 = (long) a % n;
        if(a1 < 0)
                a2 = n + a1;
        else
```

```
            a2 = a1;
        a3 = (unsigned int) a2;

        return a3;
}
```

3.4.3 Calcolo del prodotto

Date due trasformazioni $\sigma_1, \sigma_2 \in \mathcal{B}^n_{r_0,r_1}(R)$ vogliamo calcolare $\tau = \sigma_2 * \sigma_1$ Le difficoltà concettuali sono scarse: si tratta di applicare prima σ_1 e poi σ_2 a tutti i rappresentanti delle orbite e poi dal risultato costruire una rappresentazione di τ. Ben più consistenti sono le difficoltà tecniche, dovute al gran numero di permutazioni e manipolazioni delle rappresentazioni decimali usate. Presentiamo quindi uno schema dell'algoritmo prima di descriverlo in dettaglio.

Si ricorda che `orbit` è un puntatore a una array di strutture che contengono due elementi: `unsigned long orbit.rep` che è una rappresentazione decimale del rappresentante dell'orbita, e `unsigned int orbit.period`, il periodo di `orbit.rep`. L'algoritmo fa anche uso dell'intero `num_orbits`, che contiene il numero di orbite di R^n.

- Trova orbite e periodi utilizzando `find_orbits`
- Inizializza il contatore `k` a 0
- Ripeti ...
  - Sia $\underline{v}$ = `*(orbit + k).rep`
  - Sia `et` la rappresentazione decimale di $\underline{v}$
  - Sia `e` la rappresentazione decimale di $\sigma_2(\sigma_1(\underline{v}))$

- Sia `f` la $(r_0 + 1)$-esima componente della rappresentazione locale di `e`
- Sia `result = result + f * R^et`
- Inizializza il contatore `p` a 0
- Ripeti ...
  - Sia $\underline{u} = \alpha^p \underline{v}$
  - Sia `e1` la rappresentazione decimale di $\underline{u}$
  - Sia `q = ` $r_0 + 1$ `+ p` mod `n`
  - Se `q = 0` sia `q = n`
  - Sia `f` la `q`-esima componente della rappresentazione locale di `e` (nota bene: NON di `e1`!)
  - Sia `result = result + f * R ^e1`
- ... fintantoché `p` è minore di `*(orbit + cv).period`

• ... fintantoché `k` è minore di `num_orbits`

Segue il programma in C.

```
unsigned long multiply_rules(unsigned int R, unsigned int n, \
                             unsigned int r0, \
                             unsigned int r1, \
                             unsigned long sigma1, \
                             unsigned long sigma2)
{
        unsigned int i, p, k, f, q;
        unsigned long result, e, et, e1, t;
        unsigned int *v, *num_orbits;
        struct orb *orbit;
```

```
        num_orbits = (unsigned int *) malloc(sizeof(int));
        orbit = find_orbits(R, n, num_orbits);

        v = (unsigned int *) malloc(n * sizeof(int));
        result = 0;
        for(k = 0; k < *num_orbits; k++)
        {
                et = (*(orbit + k)).rep;
                e = apply_rule(R, n, r0, r1, sigma1, et);
                e = apply_rule(R, n, r0, r1, sigma2, e);
                f = (e / power(R, n - r0 - 1)) % R;
                result = result + f * power(R, et);
                v = dec2vett(R, n, et);
                for(m = 1; p < (*(orbit + k)).period; m++)
                {
                        e1 = shift_vector(R, n, v, p);
                        q = mod(r0 + 1 + p, n);
                        if(q == 0)
                                q = n;
                        f = (e / power(R, n - q)) % R;
                        result = result + f * power(R, e1);
                }
        }

        return result;
}
```

Assumendo che il ciclo `for` più esterno compia $\frac{|R|^n}{\bar{n}}$ passi e quello più

interno ne compia $\bar{n}$, facendo la somma di tutti i termini si ottiene un ordine globale di passi

$$|R|^n \log_2(2^{\frac{2}{n+1}}(n!)^{\frac{6}{n+1}+2}(n+r_0+1)^{\frac{2}{n+1}}n|R|^{n(\frac{4n+2}{n+1}+1)})$$

che introducendo le semplificazioni date dai termini con lo stesso ordine e omettendo i termini trascurabili, si riduce a

$$|R|^n \log_2(2^{\frac{1}{n}}|R|^n(n!))$$

Tutti gli algoritmi fanno uso di interi `unsigned long` per fare riferimento alle rappresentazioni decimali delle trasformazioni. In ANSI C questo tipo di variabile è implementato da una stringa binaria di 32 bit, quindi se l'anello R è $\mathbb{Z}_2$ (come spesso succede) questo permette di fare i calcoli in $\mathcal{A}^n_{r_0}(\mathbb{Z}_2)$ dove $n \leq 5$. Per $n > 5$ gli algoritmi richiedono delle modifiche per sostituire `unsigned long` con un tipo di variabile più adatto allo scopo.

È noto che il linguaggio C non dispone di un operatore di elevazione a potenza. Nella *standard library* si trova una funzione `pow()` che però è disegnata per lavorare su variabili *floating point*. Data la enorme quantità di operazioni coinvolte nel calcolo di una moltiplicazione tra trasformazioni cellulari (soprattutto quando la lunghezza degli automi non è trascurabile) la velocità degli algoritmi è cruciale, e abbiamo quindi preferito usare un algoritmo di esponenziazione rapida esclusivamente intera, che riportiamo di seguito.

```
unsigned long power(unsigned long a, unsigned long n)
{
        unsigned long u, v, t;

        u = n;
```

```
        v = 1;
        t = a;
        while(u != 0)
        {
                if(u % 2 == 1)
                {
                    v = v * t;
                }
                t = t * t;
                u = u / 2;
        }
        return v;
}
```

3.5 Applicazioni dell'algoritmo di moltiplicazione

3.5.1 Tavole di moltiplicazione

Il materiale presentato in questo capitolo è servito per implementare dei programmi di manipolazione delle trasformazioni cellulari. Sono state compilate delle tavole di moltiplicazione di $\mathcal{A}^3(\mathbb{Z}_2)$ e delle "tavole dei cicli che per ogni elemento di $\mathcal{A}^3(\mathbb{Z}_2)$ riportano gli elementi da esso generato. Queste ultime permettono di trovare una lista di generatori di $\mathcal{A}^n(R)$ non molto più estesa delle dimensioni stimate di una lista minimale possibile[11]. Per esempio, nel

[11]Si può stimare la cardinalità di un insieme minimale possibile di generatori di un monoide M trovando il numero minimo l di elementi del monoide con periodi $p_1, \dots p_l$ tali che $\prod_{i=1}^{l} p_i \geq |M|$: infatti se si avesse $\prod_{i=1}^{l} p_i < |M|$ sicuramente per motivi di cardinalità l'insieme considerato non potrebbe essere un insieme di generatori.

caso di $\mathcal{A}^3(\mathbb{Z}_2)$ i generatori trovati dal programma sono

$$\{3, 11, 14, 15, 25, 26, 27, 41, 43\}$$

Dove la rappresentazione decimale è computata su $(1, 1)$-intorni. Poiché dalle tavole dei cicli risulta evidente che la dimensione massima di ogni ciclo è 7, la dimensione minima l che una lista di generatori può avere è l'intero minimo tale che $7^l \geq 2^{2^3}$, e quindi $l = 3$.

3.5.2 Ottimizzazione degli algoritmi

Gli algoritmi sono stati presentati in modo da chiarificare il più possibile la loro lettura, il che significa il più delle volte non sono affatto ottimizzati. Senza voler entrare nel merito delle architetture dei calcolatori, ci sono dei particolari accorgimenti che sono stati usati sui programmi prima della compilazione:

1. controllare che nelle clausole dei cicli appaiano funzioni solo quando non è possibile fare altrimenti. Per esempio, in una situazione come la seguente

```
unsigned long i;
unsigned int R, n;


R = 2;
n = 7;


for(i = 0; i < power(R, n); i++)
```

   ogni volta che si incrementa `i` viene ricalcolata `power(R, n)`. Una soluzione più economica, in termini di tempo, sarebbe

```
unsigned long i, tmp;
unsigned int R, n;

R = 2;
n = 7;
tmp = power(R, n);

for(i = 0; i < tmp; i++)
```

2. Attenzione all'uso della funzione `power` quando è interna al corpo di un ciclo. Negli algoritmi di cui sopra, la seguente situazione capita spesso (`v` è un puntatore ad una *array* di `unsigned int` di dimensione `veclen`, `s, i` sono `unsigned long` e `R` è un `unsigned int`):

```
for(i = 0; i < veclen; i++)
        s = s + (*(v + i)) * power(R, i);
```

Molto meglio sarebbe calcolare `power(R, i)` progressivamente, nel seguente modo:

```
unsigned long tmp;

tmp = 1;
for(i = 0; i < veclen; i++)
{
       s = s + (*(v + i)) * tmp;
       tmp = tmp * R;
}
```

4 Divisione in Orbite

Abbiamo visto nella sezione 2.4 che per determinare completamente una trasformazione $\tau \in \mathcal{A}^n(R)$ è sufficiente conoscere l'effetto di τ sui rappresentanti delle orbite di R^n sotto il gruppo ciclico C_n generato da α, dove $\alpha = (12 \dots n)$. Nel corso del seguente capitolo svilupperemo delle tecniche per esprimere delle informazioni sulle orbite in funzione di n. Introdurremo un nuovo modo per identificare gli elementi di $\mathcal{A}^n(R)$, la rappresentazione orbitale, che verrà usata estensivamente nel capitolo 5 come tecnica ausiliaria per calcolare la struttura del gruppo massimale contenuto in $\mathcal{A}^n(R)$.

4.1 Fondamenti

Se G è un gruppo, l'insieme X si dice **G-insieme** se è definito un prodotto fra elementi di G e elementi di X tale che essendo 1 l'unità di G si abbia $1x = x$ per ogni $x \in X$ e tale che se g e h sono elementi di G si abbia $(gh)x = g(hx)$. Per ogni elemento $x \in X$ si può definire l'orbita di x come $Gx = \{gx \mid g \in G\}$. Chiamiamo $|Gx|$ la lunghezza o il periodo dell'orbita. Si ricorda che le orbite di X sotto G formano una partizione di X.

Siano $x_1, \dots x_k$ i rappresentanti delle k orbite di X. Allora si ha

$$|X| = \sum_{i=1}^{k} |Gx_i|$$

Sia ora $C_n = \{\alpha^i \mid i < n\}$ il gruppo ciclico di ordine n generato da α, e consideriamo l'azione di C_n su R^n.

4.1 Proposizione

Per ogni $\underline{v} \in R^n$ si ha che $|C_n\underline{v}|$ divide n.

Dimostrazione. Definiamo un prodotto $\times$ sull'orbita $C_n\underline{v}$ come

$$\alpha^i\underline{v} \times \alpha^j\underline{v} = \alpha^{i+j}\underline{v}$$

Il prodotto $\times$ è evidentemente chiuso, $1\underline{v}$ è l'identità e per ogni i si ha

$$\alpha^i\underline{v} \times \alpha^{n-i}\underline{v} = 1\underline{v}$$

perciò $(C_n\underline{v}, \times)$ è un gruppo. Si definisca ora la mappa $\phi : C_n \to C_n\underline{v}$ data da $\phi(\alpha^i) = \alpha^i\underline{v}$. ϕ è chiaramente un omomorfismo di gruppi ed è suriettiva, perciò $\mathrm{Im}\phi = C_n\underline{v} \leq C_n$. Per il teorema di Lagrange si ha poi che $|C_n\underline{v}|$ divide $|C_n|$ da cui

$$\forall \underline{v} \in R^n \ (|C_n\underline{v}| \mid n)$$

□

4.2 Proposizione

Per ogni $\tau \in \mathcal{A}^n(R)$ e per ogni $\underline{v} \in R^n$ la lunghezza dell'orbita $C_n\tau(\underline{v})$ divide la lunghezza dell'orbita $C_n\underline{v}$.

Dimostrazione. Usiamo il prodotto $\times$ definito su $C_n\underline{v}$ dato da

$$\alpha^i\underline{v} \times \alpha^j\underline{v} = \alpha^{i+j}\underline{v}$$

È già stato provato nella dimostrazione della proposizione (4.1) che $(C_n\underline{v}, \times)$ è un gruppo. Sia $\theta : C_n\underline{v} \to C_n\tau(\underline{v})$ la funzione data da $\theta(\alpha^i\underline{v}) = \tau(\alpha^i\underline{v})$. θ è ben definita in virtù del fatto che τ è ben definita. θ è un omomorfismo di gruppi: dati $\alpha^i\underline{v}$ e $\alpha^j\underline{v}$ in $C_n\underline{v}$ si ha

$$\theta(\alpha^i\underline{v} \times \alpha^j\underline{v}) = \theta(\alpha^{i+j}\underline{v}) = \tau(\alpha^{i+j}\underline{v})$$

Poiché $\tau \in \mathcal{A}^n(R)$, per il teorema (2.3) τ commuta con le potenze di α, perciò

$$\tau(\alpha^{i+j}\underline{v}) = \alpha^{i+j}\tau(\underline{v}) = \alpha^i\tau(\underline{v}) \times \alpha^j\tau(\underline{v})$$

Ancora per il teorema (2.3) questo è uguale a

$$\tau(\alpha^i \underline{v}) \times \tau(\alpha^j \underline{v}) = \theta(\alpha^i \underline{v}) \times \theta(\alpha^j \underline{v})$$

Per quanto riguarda gli inversi, per ogni i

$$\theta(\alpha^{-i} \underline{v}) = \tau(\alpha^{-i} \underline{v}) = \alpha^{-i} \tau(\underline{v}) = (\alpha^i \tau(\underline{v}))^{-1} = (\tau(\alpha^i \underline{v}))^{-1} = \theta^{-1}(\alpha^i \underline{v})$$

Inoltre θ è suriettivo: sia $\underline{u} \in C_n \tau(\underline{v})$; allora per qualche i

$$\underline{u} = \alpha^i \tau(\underline{v}) = \tau(\alpha^i \underline{v}) = \theta(\alpha^i \underline{v})$$

Dunque l'immagine di θ è $C_n \tau(\underline{v})$, che implica che $C_n \tau(\underline{v})$ è un sottogruppo di $C_n \underline{v}$. Per il teorema di Lagrange segue il risultato. □

4.2 Rappresentazioni orbitali

Data una trasformazione τ di $\mathcal{A}^n(R)$, la sua **rappresentazione orbitale** $\mathcal{R}(\tau)$ non è altro che la sua restrizione ad un insieme di rappresentanti delle orbite di R^n sotto C_n. Sia

$$S = \{\underline{v}_1, \ldots, \underline{v}_k\}$$

un insieme di rappresentanti delle orbite $\underline{v}_i$. Allora $\mathcal{R}(\tau) = \tau|_S$.

Per le considerazioni fatte nella sezione 2.4 (cioè che $\tau \in \mathcal{A}^n(R)$ è determinata quando si conosce l'effetto su ognuno dei rappresentanti delle orbite di R^n sotto C_n), c'è un isomorfismo fra $\mathcal{A}^n(R)$ e l'insieme delle rappresentazioni orbitali di $\mathcal{A}^n(R)$ dato da $\tau \to \mathcal{R}(\tau)$. In breve, la rappresentazione orbitale è il modo simbolico più compatto per descrivere un elemento di $\mathcal{A}^n(R)$. Abbiamo già fatto uso delle rappresentazioni orbitali nell'esempio della sezione 2.4.

4.3 Esempio

La rappresentazione orbitale della "regola 90 estesa a $\mathcal{A}^3(\mathbb{Z}_2)$ con intorno convenzionale è data da

$$\begin{aligned}\mathcal{R}(\tau)(0,0,0) &= (0,0,0)\\ \mathcal{R}(\tau)(0,0,1) &= (1,0,1)\\ \mathcal{R}(\tau)(0,1,1) &= (1,1,0)\\ \mathcal{R}(\tau)(1,1,1) &= (0,0,0)\end{aligned}$$

Oppure, usando una notazione più stringata,

$$\begin{aligned}\mathcal{R}(\tau) = \{&((0,0,0),(0,0,0)),((0,0,1),(1,0,1)),((0,1,1),(1,1,0)),\\ &((1,1,1),(0,0,0))\}\end{aligned}$$

□

4.3 Schema orbitale

Siano $r_1,\ldots,r_l$ tutte le lunghezze distinte delle orbite di R^n sotto C_n. Lo **schema orbitale** di R^n sotto C_n è l'insieme

$$S(R,n) = \{(r_1,t_1),\ldots,(r_l,t_l)\}$$

dove per ogni $i \leq l$ ci sono t_i orbite di lunghezza r_i. L'obbiettivo di questo capitolo è quello di trovare lo schema orbitale dati R e n, ovvero di trovare, per ogni $i < l$, t_l in funzione di r_l. Definiamo a tale proposito le funzioni

$$\begin{aligned}\Omega_R(m) &= \text{numero di vettori in } R^n \text{ con periodo } m \text{ sotto } \alpha\\ \omega_R(m) &= \text{numero di orbite di lunghezza } m \text{ in } R^n\end{aligned}$$

Notiamo subito che in R^n ci sono $\Omega_R(m)$ vettori con periodo m divisi in orbite disgiunte di lunghezza m, e quindi

$$\omega_R(m) = \frac{\Omega_R(m)}{m}$$

Pertanto è sufficiente trovare $\Omega_R(m)$. Si procede per induzione: è chiaro che $\Omega_R(1) = |R|$: infatti per qualsiasi n i vettori di periodo 1 sono quelli della forma

$$\underbrace{(r, r, \ldots, r)}_{n} \quad \text{dove} \quad r \in R$$

Ora il passo induttivo: per $m > 1$, il numero dei vettori di R^m con periodo m è il numero totale dei vettori in R^m meno il numero dei vettori con periodo minore di m. Abbiamo dunque

$$\Omega_R(m) = |R|^m - \sum_{j=1}^{m-1} c_j \Omega_R(j)$$

dove c_j vale 1 se esistono vettori di periodo j in R^m e 0 altrimenti. A questo proposito, ricordiamo che per la proposizione (4.1) il periodo di un vettore di R^n deve dividere m. Viceversa, per ogni divisore di m esiste almeno un vettore di periodo d, come si prova nella seguente

4.4 Proposizione

Se $|R| \geq 2$, per ogni divisore d di n c'è un'orbita di lunghezza d.

Dimostrazione. Sia 0 l'elemento neutro del gruppo $(R, +)$ e 1 l'elemento neutro del monoide $(R, \cdot)$. Sia

$$\underline{v}_0 = (\underbrace{1, 0, \ldots, 0}_{d}, \ldots, \underbrace{1, 0, \ldots, 0}_{d})$$

Chiaramente si ha $\alpha^d(\underline{v}_0) = \underline{v}_0$ e $\alpha^i(\underline{v}_0) \neq \underline{v}_0$ per ogni $i < d, i > 0$. □

Pertanto abbiamo che, in R^m,

$$\Omega_R(m) = |R|^m - \sum_{\substack{d|m \\ d \neq m}} \Omega_R(d) \tag{7}$$

Si sottolinea il fatto che fra i divisori d di m va inclusa l'unità.

Dimostriamo ora che l'espressione (7) vale anche in R^n dove $m|n$: i vettori di periodo m in R^n sono quelli della forma

$$\underline{v} = (a_1, \ldots, a_m, \ldots, a_1, \ldots, a_m)$$

dove $\underline{u} = (a_1, \ldots, a_m)$ ha periodo m. Il numero di vettori della forma di $\underline{v}$ in R^n è chiaramente uguale al numero di vettori della forma di $\underline{u}$ in R^m, e quindi l'espressione (7) vale anche nel caso in cui i vettori appartengano ad R^n dove $m|n$.

In conclusione, in R^n, per i divisori propri e impropri m di n, si ha

$$\begin{aligned} \Omega_R(m) &= |R|^m - \sum_{\substack{d|m \\ d\neq m}} \Omega_R(d) \\ \omega_R(m) &= \frac{1}{m}(|R|^m - \sum_{\substack{d|m \\ d\neq m}} d\ \omega_R(d)) \end{aligned}$$

Ove ciò non desse adito ad ambiguità, indicheremo in seguito $\omega_R(m)$ omettendo l'indice R, con $\omega(m)$.

4.5 Esempio

Per mezzo delle rappresentazioni orbitali e dello schema orbitale possiamo calcolare un prodotto di trasformazioni abbastanza facilmente. Per esempio, sia σ_1 l'estensione con intorno convenzionale della rappresentazione locale

$$(0,0,1,0,1,2,0,0,0,1,1,2,1,1,0,2,2,1,2,0,0,2,1,2,0,2,0)$$

e σ_2 di

$$(2,1,0,2,0,1,0,1,2,1,0,2,0,2,1,1,1,1,1,0,0,0,2,0,0,0,0)$$

nel monoide $\mathcal{A}^3(\mathbb{Z}_3)$. Vogliamo trovare τ tale che $\tau = \sigma_2 * \sigma_1$. Calcoliamo le orbite di $\mathbb{Z}_3^3$ sotto C_3. Per quanto detto sopra, lo schema orbitale è

$$S(\mathbb{Z}_3, 3) = \{(1,3),(3,8)\}$$

Le orbite di lunghezza 1 sono

$$\begin{aligned} O_0 &= \{(0,0,0)\} \\ O_1 &= \{(1,1,1)\} \\ O_2 &= \{(2,2,2)\} \end{aligned}$$

e per quelle di lunghezza 3 basta trovare 8 vettori che non siano "congrui modulo la permutazione (1 2 3):

$$\begin{aligned} O_3 &= \{\alpha^i(0,0,1) \mid i < 3\} \\ O_4 &= \{\alpha^i(0,0,2) \mid i < 3\} \\ O_5 &= \{\alpha^i(0,1,1) \mid i < 3\} \\ O_6 &= \{\alpha^i(0,1,2) \mid i < 3\} \\ O_7 &= \{\alpha^i(0,2,1) \mid i < 3\} \\ O_8 &= \{\alpha^i(0,2,2) \mid i < 3\} \\ O_9 &= \{\alpha^i(1,1,2) \mid i < 3\} \\ O_{10} &= \{\alpha^i(1,2,2) \mid i < 3\} \end{aligned}$$

Troviamo ora le rappresentazioni orbitali di σ_1 e σ_2. In generale, per $i = 1, 2$, si avrà

$$\mathcal{R}(\sigma_i)(a_2, a_1, a_0) = (\sigma_i^{(3^2a_2+3a_1+a_0)}, \sigma_i^{(3^2a_1+3a_0+a_2)}, \sigma_i^{(3^2a_0+3a_2+a_1)})$$

dove indichiamo con $\sigma_i^{(j)}$ la j-esima componente della rappresentazione locale che σ_i estende, e quindi

$$\begin{aligned} \mathcal{R}(\sigma_1) = \{&((0,0,0),(0,0,0)), ((1,1,1),(1,1,1)), ((2,2,2),(0,0,0)), \\ &((0,0,1),(2,2,1)), ((0,0,2),(0,0,0)), ((0,1,1),(1,0,2)), \\ &((0,1,2),(2,2,0)), ((0,2,1),(0,2,2)), ((0,2,2),(2,1,0)), \\ &((1,1,2),(1,1,1)), ((1,2,2),(1,0,0))\} \end{aligned}$$

e

$$\begin{aligned}\mathcal{R}(\sigma_2) &= \{((0,0,0),(0,0,0)),((1,1,1),(2,2,2)),((2,2,2),(2,2,2)),\\ &\quad ((0,0,1),(0,0,1)),((0,0,2),(0,0,2)),((0,1,1),(2,1,1)),\\ &\quad ((0,1,2),(0,2,1)),((0,2,1),(0,1,1)),((0,2,2),(1,0,0)),\\ &\quad ((1,1,2),(2,0,0)),((1,2,2),(1,1,2))\}\end{aligned}$$

Se ne deduce che

$$\begin{aligned}\mathcal{R}(\tau) &= \{((0,0,0),(0,0,0)),((1,1,1),(2,2,2)),((2,2,2),(0,0,0)),\\ &\quad ((0,0,1),(1,2,1)),((0,0,2),(0,0,0)),((0,1,1),(1,0,1)),\\ &\quad ((0,1,2),(0,0,1)),((0,2,1),(1,0,0)),((0,2,2),(1,1,0)),\\ &\quad ((1,1,2),(2,2,2)),((1,2,2),(1,0,0))\}\end{aligned}$$

e quindi la rappresentazione locale di τ è data da

$$(0,0,1,0,2,0,0,1,0,1,2,0,2,2,0,0,1,1,1,1,0,0,1,2,0,1,0)$$

□

4.4 Prodotto simbolico di trasformazioni in $\mathcal{A}^n(R)$

Rappresentazioni orbitali e schema orbitale sono tecniche molto valide per trattare le trasformazioni di $\mathcal{A}^n(R)$. Sia S un insieme di rappresentanti delle orbite di R^n sotto C_n e siano $d_1, \ldots, d_l$ i divisori propri e impropri di n. Possiamo indicizzare i rappresentanti delle orbite nel modo seguente

$$S = \{\underline{v}_{1,1}, \ldots, \underline{v}_{1,\omega(d_1)}, \ldots, \underline{v}_{l,1}, \ldots, \underline{v}_{l,\omega(d_l)}\}$$

in modo che, tenuto fisso i, i vettori $\underline{v}_{i,j}$ abbiano tutti orbite della stessa lunghezza d_i. Se applichiamo il concetto di rappresentazione orbitale ai vettori di S possiamo descrivere una $\tau \in \mathcal{A}^n(R)$ per mezzo del suo effetto sui

$\underline{v}_{i,j}$, cioè per ogni $i \leq l$ e per ogni $j \leq \omega(d_i)$ esistono interi $e_{i,j}, i', j'$ con $e < d_{i'}, i' \leq l, j' \leq \omega(d_{i'})$ tali che

$$\tau(\underline{v}_{i,j}) = \alpha^{e_{i,j}} \underline{v}_{i',j'} \tag{8}$$

dove per la proposizione (4.2) sappiamo che la lunghezza dell'orbita di $\tau(\underline{v})$ divide la lunghezza dell'orbita di $\underline{v}$, e quindi i' deve essere tale che $d_{i'}|d_i$.

È anche vero l'inverso: cioè che se τ è definita sui vettori di S come nell'equazione (8) allora τ può venire estesa a tutti i vettori di R^n in tal modo che $\tau \in \mathcal{A}^n(R)$. Infatti per ogni $\underline{v} \in R^n$ esistono un $\underline{v}_{i,j}$ e un intero m tali che $\underline{v} = \alpha^m \underline{v}_{i,j}$. Estendiamo τ nel seguente modo:

$$\tau(\underline{v}) = \alpha^m \tau(\underline{v}_{i,j}) = \alpha^{m+e_{i,j}} \underline{v}_{i',j'}$$

È evidente che τ così definita commuta con α, e quindi per il teorema (2.3) $\tau \in \mathcal{A}^n(R)$. Bisogna solo controllare che τ sia ben definita: sia $\underline{u} = \underline{v}_{i,j} = \alpha^{d_i} \underline{v}_{i,j}$ (giacché $\underline{v}_{i,j}$ ha orbita di lunghezza d_i); allora

$$\tau(\underline{u}) = \tau(\alpha^{d_i} \underline{v}_{i,j}) = \alpha^{d_i} \tau(\underline{v}_{i,j}) = \alpha^{d_i} \alpha^{e_{i,j}} \underline{v}_{i',j'}$$

Poiché $d_{i'}|d_i$ questo è uguale a

$$\alpha^{e_{i,j}} \underline{v}_{i',j'} = \tau(\underline{v}_{i,j})$$

e quindi τ è ben definita.

Si noti inoltre che se τ è data da

$$\tau(\underline{v}_{i,j}) = \alpha^{e_{i,j}} \underline{v}_{s(i),t_i(j)}$$

e σ da

$$\sigma(\underline{v}_{i,j}) = \alpha^{f_{i,j}} \underline{v}_{u(i),w_i(j)}$$

dove s, u sono funzioni da $\{1, \ldots, l\}$ in se stesso e t_i, w_i da $\{1, \ldots, \omega_{d_i}\}$ in se stesso, allora

$$(\sigma * \tau)(\underline{v}_{i,j}) = \alpha^{f_{s(i),t_i(j)} + e_{i,j}} \underline{v}_{u(s(i)),w_{s(i)}(t_i(j))}$$

Possiamo dunque esprimere ogni $\tau \in \mathcal{A}^n(R)$ per mezzo di un vettore

$$\underline{e} = (e_{1,1}, \ldots, e_{1,\omega(d_1)}, \ldots, e_{l,1}, \ldots, e_{l,\omega(d_l)})$$

di una funzione s da $\{1, \ldots, l\}$ in se stesso tale che per ogni $i \leq l$ si abbia $d_{s(i)}|d_i$, e di funzioni t_i da $\{1, \ldots, \omega(d_i)\}$ in se stesso per ogni $i \leq l$: in conclusione si ha che esiste un isomorfismo fra il monoide $(\mathcal{A}^n(R), *)$ e la struttura algebrica A i cui elementi sono del tipo

$$((e_{1,1}, \ldots, e_{1,\omega(d_1)}, \ldots, e_{l,1}, \ldots, e_{l,\omega(d_l)}), s, t_i)$$

con un prodotto definito nel modo seguente:

$$\begin{aligned}((f_{1,1}, \ldots, f_{1,\omega(d_1)}, \ldots, f_{l,1}, \ldots, f_{l,\omega(d_l)}), u, w_i) \\ ((e_{1,1}, \ldots, e_{1,\omega(d_1)}, \ldots, e_{l,1}, \ldots, e_{l,\omega(d_l)}), s, t_i) = \\ ((f_{s(1),t_1(1)} + e_{1,1}, \ldots, f_{s(1),t_1(\omega(d_1)} + e_{1,\omega(d_1)}, \ldots, f_{s(l),t_l(1)} + \\ + e_{l,1}, \ldots, f_{s(l),t_l(\omega(d_l))} + e_{l,\omega(d_l)}), u \circ s, w_{s(i)} \circ t_i)\end{aligned}$$

La struttura algebrica A appena definita verrà usata estensivamente nel capitolo 5 per analizzare la struttura del gruppo massimale contenuto in $\mathcal{A}^n(R)$.

5 Trasformazioni Cellulari Invertibili

Una delle principali difficoltà, non solo teoriche, nella trattazione del monoide $\mathcal{A}^n(R)$ è la mancanza in generale degli elementi invertibili. Per mezzo degli algoritmi proposti nel capitolo 3, tuttavia, non è difficile trovare coppie di trasformazioni cellulari che moltiplicate insieme danno l'unità. Per esempio in $\mathcal{A}_1^3(\mathbb{Z}_2)$ si possono prendere $(15, 85)$, $(142, 212)$ (e molte altre). Ci proponiamo, nel corso del capitolo, di approfondire questa idea e di ricavare delle regole per analizzare la configurazione algebrica del più grande sottoinsieme di $\mathcal{A}^n(R)$ che possiede una struttura di gruppo. Daremo inoltre dei criteri per decidere se una data trasformazione è invertibile o meno.

5.1 Il gruppo $\mathcal{G}^n(R)$

Chiamiamo $\mathcal{G}^n(R)$ il più grande sottogruppo del monoide $\mathcal{A}^n(R)$. È noto che il sottogruppo massimale di un monoide è formato da tutti gli elementi invertibili del monoide. Così

$$\mathcal{G}^n(R) = \{\sigma \in \mathcal{A}^n(R) \mid \sigma \text{è invertibile } \}$$

Possiamo ora determinare gli elementi di $\mathcal{G}^n(R)$ data la tavola di moltiplicazione[12].

5.1 Esempio

Si consideri il monoide $\mathcal{A}^2(\mathbb{Z}_2)$. Cercheremo sulle tavole di moltiplicazione tutti gli elementi per cui esiste un altro elemento tale che il loro prodotto dia l'identità. Calcoliamo prima di tutto l'identità e di $\mathcal{A}^2(\mathbb{Z}_2)$. La rappresentazione orbitale di e è

$$\mathcal{R}(e) = \{((0,0),(0,0)), ((0,1),(0,1)), ((1,1),(1,1))\}$$

[12]Cfr. capitolo 3, in particolare la sezione 3.5.1.

a cui corrispondono una rappresentazione locale su $(1,0)$-intorni $(1,0,1,0)$ ed una rappresentazione decimale 10. Dalle tavole

$*$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	15	14	9	8	9	8	9	8	7	6	1	0	1	0	1	0
2	0	0	2	2	4	4	0	0	0	0	2	2	4	4	0	0
3	15	14	11	10	13	12	9	8	7	6	3	2	5	4	1	0
4	0	0	4	4	2	2	0	0	0	0	4	4	2	2	0	0
5	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
6	0	0	6	6	6	6	0	0	0	0	6	6	6	6	0	0
7	15	14	15	14	15	14	9	8	7	6	7	6	7	6	1	0
8	0	1	0	1	0	1	6	7	8	9	8	9	8	9	14	15
9	15	15	9	9	9	9	15	15	15	15	9	9	9	9	15	15
10	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	15	15	11	11	13	13	15	15	15	15	11	11	13	13	15	15
12	0	1	4	5	2	3	6	7	8	9	12	13	10	11	14	15
13	15	15	13	13	11	11	15	15	15	15	13	13	11	11	15	15
14	0	1	6	7	6	7	6	7	8	9	14	15	14	15	14	15
15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15

si evince che $\mathcal{G}^2(\mathbb{Z}_2) = \{10, 12, 3, 5\}$ e che ognuno degli elementi del gruppo ha ordine 2, perciò $\mathcal{G}^2(\mathbb{Z}_2) = C_2 \times C_2$. □

5.2 Caratterizzazione degli elementi di $\mathcal{G}^n(R)$

Allo scopo di fornire dei criteri per determinare se un elemento τ di $\mathcal{A}^n(R)$ appartiene o meno al gruppo $\mathcal{G}^n(R)$, consideriamo la sua rappresentazione orbitale $\mathcal{R}(\tau)$. Definiamo la relazione di equivalenza $\sim$ sui vettori di R^n data dall'appartenenza alla stessa orbita sotto α

$$\underline{u} \sim \underline{v} \Leftrightarrow \underline{u} \in C_n \underline{v}$$

e diciamo che per sottoinsiemi T e U di R^n, T è **isomorfo orbitale** a U se e solo se per ogni $\underline{t}$ in T l'orbita rappresentata da $\underline{t}$ ha almeno un rappresentante anche in U e viceversa, ovvero

$$T \cong_O U \Leftrightarrow \forall\ \underline{t} \in T\ \exists\ \underline{u} \in U\ (\underline{t} \sim \underline{u}) \wedge \forall\ \underline{u} \in U \exists\ \underline{t} \in T\ (\underline{u} \sim \underline{t})$$

5.2 Teorema

Sia $\tau \in \mathcal{A}^n(R)$ e sia e l'elemento neutro di $\mathcal{A}^n(R)$. Allora τ è invertibile se e solo se Im $\mathcal{R}(\tau) \cong_O$ Im $\mathcal{R}(e)$.

Dimostrazione. Sia $S = \{\underline{v}_1, \ldots, \underline{v}_k\}$ l'insieme dei rappresentanti delle orbite di R^n sotto C_n. Per definizione

$$\mathcal{R}(e) = \{(\underline{s}, \underline{s}) \mid \underline{s} \in S\}$$

e quindi Im $\mathcal{R}(e) = S$.

($\Rightarrow$): Sia τ invertibile e supponiamo per assurdo che $\exists\ \underline{v} \in S$ tale che per ogni $\underline{u} \in$ Im $\mathcal{R}(\tau)$ si abbia $\underline{v} \not\sim \underline{u}$. Esistono allora $\underline{s}, \underline{t} \in S$ non equivalenti sotto $\sim$ tali che

$$\mathcal{R}(\tau)(\underline{s}) \sim \mathcal{R}(\tau)(\underline{t}) \tag{9}$$

Poiché abbiamo supposto τ invertibile e $\mathcal{R}(\tau)$ non è altro che la restrizione di τ a S, esiste una funzione ϕ inversa a $\mathcal{R}(\tau)$. Pertanto dalla condizione (9) si evince che

$$\phi\mathcal{R}(\tau)(\underline{s}) \sim \phi\mathcal{R}(\tau)(\underline{t}) \Rightarrow \underline{s} \sim \underline{t}$$

che è in contraddizione con la supposizione che $\underline{s}$ e $\underline{t}$ non siano equivalenti.

($\Leftarrow$): Si ha

$$\mathcal{R}(\tau) = \{(\underline{s}, \tau(\underline{s})) \mid \underline{s} \in S\}$$

Dato che Im $\mathcal{R}(\tau) \cong_O S$, gli elementi di Im $\mathcal{R}(\tau)$ sono rappresentanti delle orbite di R^n sotto C_n, pertanto l'insieme

$$\{(\tau(\underline{s}), \underline{s}) \mid \underline{s} \in S\}$$

è la rappresentazione orbitale di un elemento $\sigma \in \mathcal{A}^n(R)$. È immediato verificare che per ogni $\underline{s} \in S$

$$\mathcal{R}(\tau)\mathcal{R}(\sigma)(\underline{s}) = \mathcal{R}(\sigma)\mathcal{R}(\tau)(\underline{s}) = \underline{s}$$

e quindi σ è l'inverso di τ. □

Il teorema è direttamente applicabile. Vediamo un paio di esempi.

5.3 Esempio

Consideriamo la regola "90 in $\mathcal{A}^3(\mathbb{Z}_2)$ su $(1,1)$-intorni. La sua rappresentazione locale è $(0,1,0,1,1,0,1,0)$ da cui si ottiene una rappresentazione orbitale

$$\{((0,0,0),(0,0,0)),((0,0,1),(1,1,0)),((0,1,1),(0,1,1)),((1,1,1),(0,0,0))\}$$

In questo caso Im $\mathcal{R}(90) \cong_O \{(0,0,0),(0,1,1)\}$ e quindi la regola 90 non è invertibile. □

5.4 Esempio

Consideriamo ora la regola 5 su $(1,0)$-intorni in $\mathcal{A}^2(\mathbb{Z}_2)$, che ha rappresentazione locale $(0,1,0,1)$ e rappresentazione orbitale

$$\{((0,0),(1,1)),((0,1),(1,0)),((1,1),(0,0))\}$$

Si ottiene

$$\text{Im } \mathcal{R}(5) = \{(1,1),(1,0),(0,0)\} \cong_O \{(0,0),(0,1),(1,1)\} = \text{Im } \mathcal{R}(e)$$

dove e è l'elemento neutro di $\mathcal{A}_1^2(\mathbb{Z}_2)$, e quindi 5 è invertibile. La rappresentazione orbitale dell'inverso σ di 5 è data da

$$\{((0,0),(1,1)),((0,1),(1,0)),((1,1),(0,0))\}$$

e quindi $\sigma = 5$. Il risultato è direttamente verificabile dalle tavole di moltiplicazione a pagina 68. □

5.3 La struttura di $\mathcal{G}^n(R)$

Abbiamo visto nel capitolo 4 che lo schema orbitale di R^n sotto C_n è

$$\{(d,\omega(d)) \mid d|n\}$$

dove $\omega(d)$ è definita per ricorsione nella sezione (4.3) come

$$\begin{aligned}\omega(1) &= |R| \\ \omega(d) &= \frac{1}{d}(|R|^d - \sum_{\substack{b|d \\ b\neq d}} b\ \omega(b))\end{aligned}$$

dove $b_1, \ldots, b_k$ sono i fattori primi distinti di d con la convenzione che per $d = 1, k = 0$. Fissiamo un insieme di rappresentanti delle orbite di R^n sotto C_n che chiamiamo S, e siano $d_1, \ldots, d_l$ i divisori di n (ivi compresi 1 e n). Poiché ci sono $\omega(d_i)$ orbite di lunghezza d_i si può indicizzare S nel seguente modo:

$$S = \{\underline{v}_{1,1}, \ldots, \underline{v}_{1,\omega(d_1)}, \ldots, \underline{v}_{l,1}, \ldots, \underline{v}_{l,\omega(d_l)}\}$$

dove per ogni $i \leq l$ e per $j \neq j' \leq \omega(d_i)$ si ha $|C_n\underline{v}_{i,j}| = |C_n\underline{v}_{i,j'}| = d_i$.

Prima di presentare il teorema fondamentale di questo capitolo, dimostriamo un lemma che stabilisce una condizione necessaria all'invertibilità di una trasformazione in $\mathcal{A}^n(R)$.

5.5 Lemma

Sia $\tau \in \mathcal{A}^n(R)$. Se τ è invertibile allora per ogni $\underline{v} \in R^n$ si ha

$$|C_n\tau(\underline{v})| = |C_n\underline{v}|$$

Dimostrazione. Sia $\theta : C_n\underline{v} \to C_n\tau(\underline{v})$ dato da $\theta(\alpha^i\underline{v}) = \tau(\alpha^i\underline{v})$ per ogni i. Abbiamo provato nella dimostrazione della proposizione (4.2) che θ è un omomorfismo di gruppi. Poiché τ è invertibile otteniamo che θ è un isomorfismo, da cui la tesi. □

Cerchiamo ora di determinare la struttura di $\mathcal{G}^n(R)$. Analizzeremo gli elementi di $\mathcal{A}^n(R)$ secondo il loro effetto sui rappresentanti delle orbite di R^n sotto C_n, considerando in effetti una rappresentazione orbitale "simbolica. Si proverà che tutti gli elementi di $\mathcal{G}^n(R)$ dipendono da un insieme

di permutazioni di gruppi ciclici e da un insieme di permutazioni di gruppi simmetrici. La difficoltà maggiore è data dal fatto che le permutazioni cicliche e quelle dei gruppi simmetrici non sono indipendenti: per questo motivo non è possibile esprimere $\mathcal{G}^n(R)$ come prodotto cartesiano di gruppi noti. Daremo tuttavia una descrizione del gruppo in termini di generatori e relazioni.

Definiamo innanzitutto, per ogni i minore o uguale ad l (il numero dei divisori di n) il gruppo G_i^n. Per brevità sia $z_i = \omega(d_i)$.

$$G_i^n = \{((e_{i,1}, \ldots, e_{i,z_i}), \pi_i) \mid \forall j \ e_{i,j} \in \mathbb{Z}_{d_i}, \pi_i \in S_{z_i}\} \tag{10}$$

Il prodotto su G_i^n è definito nel modo seguente:

$$\begin{aligned} & ((f_{i,1}, \ldots, f_{i,z_i}), \zeta_i)((e_{i,1}, \ldots, e_{i,z_i}), \pi_i) \\ = \ & ((\pi_i(f_{i,1}, \ldots, f_{i,z_i})) + (e_{i,1}, \ldots, e_{i,z_i}), \zeta_i\pi_i) = \\ = \ & ((f_{i,\pi_i(1)} + e_{i,1}, \ldots, f_{i,\pi_i(z_i)} + e_{i,z_i}), \zeta_i\pi_i) \end{aligned}$$

dove le somme fra le componenti dei vettori sono ovviamente intese modulo d_i. Che il prodotto sia chiuso è evidente: ciò discende dalla chiusura di $(\mathbb{Z}_{d_i}, +)$ e di S_{z_i}. Il prodotto è associativo

$$\begin{aligned} & [((e_{i,1}, \ldots, e_{i,z_i}), \pi_i)((f_{i,1}, \ldots, f_{i,z_i}), \zeta_i)]((g_{i,1}, \ldots, g_{i,z_i}), \xi_i) = \\ = \ & ((e_{i,\zeta_i(1)} + f_{i,1}, \ldots, e_{i,\zeta_i(z_i)} + f_{i,z_i}), \pi_i\zeta_i)((g_{i,1}, \ldots, g_{i,z_i}), \xi_i) = \\ = \ & ((e_{i,\zeta_i\xi_i(1)} + f_{i,\xi_i(1)} + g_{i,1}, \ldots, e_{i,\zeta_i\xi_i(z_i)} + f_{i,\xi_i(z_i)} + g_{i,z_i}), \pi_i\zeta_i\xi_i) = \\ = \ & ((e_{i,1}, \ldots, e_{i,z_i}), \pi_i)((f_{i,\xi_i(1)} + g_{i,1}, \ldots, f_{i,\xi_i(z_i)} + g_{i,z_i}), \zeta_i\xi_i) = \\ = \ & ((e_{i,1}, \ldots, e_{i,z_i}), \pi_i)[((f_{i,1}, \ldots, f_{i,z_i}), \zeta_i)((g_{i,1}, \ldots, g_{i,z_i}), \xi_i)] \end{aligned}$$

e ammette identità $((0, \ldots, 0), \eta)$, dove η è l'identità di S_{z_i}, e inversi:

$$\begin{aligned} ((e_{i,1}, \ldots, e_{i,z_i}), \pi_i)((-e_{i,\pi_i^{-1}(1)}, \ldots, e_{i,\pi_i^{-1}(z_i)}), \pi_i^{-1}) &= ((0, \ldots, 0), \eta) \\ ((-e_{i,\pi_i^{-1}(1)}, \ldots, e_{i,\pi_i^{-1}(z_i)}), \pi_i^{-1})((e_{i,1}, \ldots, e_{i,z_i}), \pi_i) &= ((0, \ldots, 0), \eta) \end{aligned}$$

Dunque G_i^n è un gruppo.

Si noti che il prodotto nel gruppo G_i^n è costituito da due "fasi: il prodotto nella prima componente $(e_{i,1}, \ldots, e_{i,z_i})$ e quello nella seconda componente π_i. Non è difficile notare che il prodotto nella prima componente dipende dal prodotto nella seconda: variando π_i, infatti, cambia anche il modo in cui si moltiplica la prima componente. Un prodotto del genere si può esprimere in teoria dei gruppi con la nozione di prodotto semi-diretto. Siano G e H due gruppi. Il **prodotto semi-diretto** $G \otimes_S H$ è dato da un gruppo i cui elementi sono gli stessi del gruppo $G \times H$ (prodotto cartesiano) ma il cui prodotto in una delle componenti varia a seconda del prodotto nell'altra. Nel nostro caso potremmo indicare il gruppo G_i^n con $C_{d_i}^{z_i} \otimes_S S_{z_i}$.

Troviamo i generatori del gruppo G_i^n. È noto che i generatori del gruppo simmetrico S_{z_i} sono i 2-cicli $\{(s,t) \mid s < z_i, t \leq z_i, s < t\}$. Indicizziamo questi 2-cicli per mezzo dei simboli $\pi_i^{(f)}$ dove $f \leq \frac{1}{2} z_i(z_i - 1)$. Per un generico elemento $((e_{i,1}, \ldots, e_{i,z_i}), \pi_i)$ di G_i^n tale che $\pi_i = \prod_q \pi_i^{(f_q)}$ si ha che

$$((e_{i,1}, \ldots, e_{i,z_i}), \pi_i) =$$

$$= \left[\prod_q ((0, \ldots, 0), \pi_i^{(f_q)})\right] [((1, 0, \ldots, 0), \eta)^{e_{i,1}} \ldots ((0, \ldots, 0, 1), \eta)^{e_{i,z_i}}] \quad (11)$$

e quindi se $\gamma_i = (12 \ldots z_i)$ e

$$\begin{aligned} g_{i,j} &= (\gamma_i^j(1, 0, \ldots, 0), \eta) \ \ \forall\ j \leq z_i \\ h_{i,f} &= ((0, \ldots, 0), \pi_i^{(f)}) \ \ \forall\ f \leq \frac{1}{2} z_i(z_i - 1) \end{aligned}$$

abbiamo che G_i^n è generato da tutti i $g_{i,j}, h_{i,f}$. Questo insieme di generatori, tuttavia, non è minimale. Si noti che per ogni $j \leq z_i$, se f è tale che $\pi_i^{(f)}$ muove j,

$$\begin{aligned} & h_{i,f}^{-1} g_{i,j} h_{i,f} = \\ = \ & ((0, \ldots, 0), (\pi_i^{(f)})^{-1})(\gamma_i^j(1, 0, \ldots, 0), \eta)((0, \ldots, 0), \pi_i^{(f)}) = \\ = \ & ((0, \ldots, 0), (\pi_i^{(f)})^{-1})(\pi_i^{(f)} \gamma_i^j(1, 0, \ldots, 0), \pi_i^{(f)}) = \\ = \ & (\pi_i^{(f)} \gamma_i^j(1, 0, \ldots, 0), \eta) = g_{i, \pi_i^{(f)}(j)} \end{aligned}$$

e quindi in particolare coniugando $g_{i,1}$ con gli $h_{i,f}$ che muovono 1 si ottengono tutti gli altri $g_{i,j}$. Chiamiamo per comodità

$$g_i = g_{i,1} = ((1, 0, \ldots, 0), \eta)$$

e proviamo che in generale non esiste insieme di generatori di G_i^n più piccolo di $\{h_{i,f} \mid 1 \leq f \leq \frac{1}{2}z_i(z_i - 1)\} \cup \{g_i\}$. Se rimuoviamo uno degli $h_{i,f}$ allora le permutazioni $\pi_i^{(f)}$ non generano più S_{z_i}, che è in contraddizione con la definizione di G_i^n; e se rimuoviamo g_i si ottiene

$$G_i^n = \langle((0, \ldots, 0), \pi_i^{(f)})\rangle \cong S_{z_i}$$

che implica $d_i = 1$, che avviene solo se $n = 1$ (il caso banale). Si ha perciò

$$G_i^n = \langle g_i, h_{i,1}, \ldots, h_{i,\frac{1}{2}z_i(z_i-1)}\rangle$$

Cerchiamo ora le relazioni fra i generatori. Osserviamo innanzitutto che essendo i $\pi_i^{(f)}$ dei 2-cicli, si ottiene $(\pi_i^{(f)})^{-1} = \pi_i^{(f)}$ e quindi per ogni f minore o uguale di $\frac{1}{2}z_i(z_i - 1)$

$$h_{i,f}^2 = 1$$

Abbiamo osservato in precedenza (vedi eq. (11)) che

$$((1, 0, \ldots, 0), \eta)^m = ((m, 0, \ldots, 0), \eta)$$

da cui

$$g_i^{d_i} = 1$$

Per ogni f abbiamo

$$\begin{aligned}
(g_i h_{i,f})^2 &= (((1, 0, \ldots, 0), \eta)((0, \ldots, 0), \pi_i^{(f)}))^2 = \\
= (\pi_i^{(f)}(1, 0, \ldots, 0), \pi_i^{(f)})^2 &= ((1, 0, \ldots, 0) + \pi_i^{(f)}(1, 0, \ldots, 0), \eta) = \\
= ((1, 0, \ldots, 0), \pi_i^{(f)})^2 &= (((0, \ldots, 0), \pi_i^{(f)})((1, 0, \ldots, 0), \eta))^2 = (h_{i,f} g_i)^2
\end{aligned}$$

e

$$\begin{aligned}
(g_i h_{i,f}) &= (\pi_i^{(f)}(1,0,\ldots,0), \pi_i^{(f)}) \\
(g_i h_{i,f})^2 &= (\pi_i^{(f)}(1,0,\ldots,0) + (1,0,\ldots,0), \eta) \\
(g_i h_{i,f})^3 &= (\pi_i^{(f)}(2,0,\ldots,0) + (1,0,\ldots,0), \pi_i^{(f)}) \\
(g_i h_{i,f})^4 &= (\pi_i^{(f)}(2,0,\ldots,0) + (2,0,\ldots,0), \eta) \\
&\vdots \\
(g_i h_{i,f})^{2d_i} &= (\pi_i^{(f)}(d_i,0,\ldots,0) + (d_i,0,\ldots,0), \eta) = \\
&= (\pi_i^{(f)}(0,\ldots,0) + (0,\ldots,0), \eta) = ((0,\ldots,0), \eta) = 1
\end{aligned}$$

Ricapitolando, se $k_i = \frac{1}{2} z_i(z_i - 1)$ si ha che $G_i^n =$

$$\langle g_i, h_{i,1}, \ldots, h_{i,k_i} \mid \forall f \leq k_i (h_{i,f}^2 = (g_i h_{i,f})^{2d_i} = g_i^{d_i} = 1, (g_i h_{i,f})^2 = (h_{i,f} g_i)^2) \rangle \quad (12)$$

dove $\langle h_{i,1}, \ldots, h_{i,k_i} \rangle \cong S_{z_i}$.

Dimostriamo ora il teorema fondamentale di questo capitolo, in cui si prova che $\mathcal{G}^n(R)$ è un prodotto cartesiano di questi gruppi G_i^n.

5.6 Teorema

Se l è il numero di divisori di n (compresi 1 e n stesso),

$$\mathcal{G}^n(R) \cong \prod_{i=1}^{l} G_i^n$$

Dimostrazione. Siano $d_1, \ldots, d_l$ tutti i divisori di n (propri e impropri) e fissiamo un insieme di rappresentanti delle orbite di R^n sotto C_n:

$$S = \{\underline{v}_{1,1}, \ldots, \underline{v}_{1,\omega(d_1)}, \ldots, \underline{v}_{l,1}, \ldots, \underline{v}_{l,\omega(d_l)}\}$$

dove fissato $i \leq l$ i vettori $\underline{v}_{i,j}$, per $1 \leq j \leq \omega(d_i)$, hanno orbite di lunghezza d_i. Si considerino ora i vettori della forma

$$\underline{e} = (e_{1,1}, \ldots, e_{1,\omega(d_1)}, \ldots, e_{l,1}, \ldots, e_{l,\omega(d_l)})$$

dove $e_{i,j} \in \mathbb{Z}_{d_i}$ e

$$\underline{\pi} = (\pi_1, \ldots, \pi_l)$$

dove $\pi_i \in S_{\omega(d_i)}$, e definiamo per ogni $\underline{e}, \underline{\pi}$ la funzione $\rho_{\underline{e},\underline{\pi}}$ tale che per ogni $\underline{v}_{i,j} \in S$

$$\rho_{\underline{e},\underline{\pi}}(\underline{v}_{i,j}) = \alpha^{e_{i,j}} \underline{v}_{i,\pi_i(j)}$$

Si consideri ora l'insieme

$$B = \{\rho_{\underline{e},\underline{\pi}} \mid \underline{e} \in \prod_{i=1}^{l} \mathbb{Z}_{d_i}^{\omega(d_i)},\ \underline{\pi} \in \prod_{i=1}^{l} S_{\omega(d_i)}\}$$

La funzione $\rho_{\underline{e},\underline{\pi}}$ è della forma data dall'equazione (8) a pagina 65, perciò B è incluso in $\mathcal{A}^n(R)$. Sia ora $\tau \in \mathcal{G}^n(R)$ e supponiamo per assurdo che $\tau \notin B$. Ciò può avvenire solo in due casi (che possono anche verificarsi contemporaneamente):

- un vettore $\underline{v}_{i,j}$ viene mandato in uno appartenente ad un'orbita di lunghezza diversa. Ovvero, esistono $i, j, e_{i,j}$ tali che $\tau(\underline{v}_{i,j}) = \alpha^{e_{i,j}} \underline{v}_{i',\pi_i(j)}$ con $i' \neq i$. Allora

  $$|C_n \tau(\underline{v}_{i,j})| = d_{i'} \neq d_i = |C_n \underline{v}_{i,j}|$$

  e quindi per il lemma (5.5) $\tau \notin \mathcal{G}^n(R)$ contro l'ipotesi.

- due vettori $\underline{v}_{i,j}, \underline{v}_{i,j'}$ con $j \neq j'$ (cioè appartenenti ad orbite diverse ma della stessa lunghezza) vengono mandati in vettori appartenenti alla stessa orbita. Ovvero, esistono interi $e_{i,j}, f_{i,j'} \in \mathbb{Z}_{d_i}$ e $j'' \leq \omega(d_i)$ tali che

  $$\begin{aligned} \tau(\underline{v}_{i,j}) &= \alpha^{e_{i,j}} \underline{v}_{i,j''} \\ \tau(\underline{v}_{i,j'}) &= \alpha^{f_{i,j'}} \underline{v}_{i,j''} \end{aligned}$$

  che implica Im $\mathcal{R}(\tau) \not\cong_O S$ e quindi per il teorema (5.2) $\tau \notin \mathcal{G}^n(R)$ contro l'ipotesi.

Pertanto $\tau \in B$, e quindi $\mathcal{G}^n(R) \subseteq B$. Dimostriamo ora l'inclusione inversa. Sia $\rho_{\underline{e},\underline{\pi}}$ in B e dimostriamo che ρ appartiene a $\mathcal{G}^n(R)$. Per il teorema (5.2) è sufficiente provare che Im $\mathcal{R}(\rho) \cong_O S$. Poiché S è l'insieme dei rappresentanti delle orbite, per ogni $\underline{v} \in$ Im $\mathcal{R}(\rho)$ esiste $\underline{s} \in S$ tale che $\underline{v} \sim \underline{s}$. Se viceversa prendiamo $\underline{v}_{i,j} \in S$ si ha che

$$\underline{v}_{i,j} = \alpha^{-e_{i,j}} \alpha^{e_{i,j}} \underline{v}_{i,\pi_i \pi_i^{-1}(j)} =$$

$$\alpha^{-e_{i,j}} \rho_{\underline{e},\underline{\pi}}(\underline{v}_{i,\pi_i^{-1}(j)}) \sim \rho_{\underline{e},\underline{\pi}}(\underline{v}_{i,\pi_i^{-1}(j)}) \in \text{Im } \mathcal{R}(\rho)$$

Quindi $\rho \in \mathcal{G}^n(R)$. Ne segue che $B = \mathcal{G}^n(R)$. Sia ora

$$C = \prod_{i=1}^{l} G_i^n$$

e dimostriamo che $B \cong C$. Si definisca $\theta : C \to B$ data da

$$\theta(((e_{1,1}, \ldots, e_{1,\omega(d_1)}), \pi_1), \ldots, ((e_{l,1}, \ldots, e_{l,\omega(d_l)}), \pi_l)) = \rho_{\underline{e},\underline{\pi}}$$

Chiaramente θ è ben definita. Dimostriamo che θ è un isomorfismo di gruppi. Siano $\underline{c}, \underline{c}'$ due elementi di C tali che

$$\begin{aligned} \theta(\underline{c}) &= \rho_{\underline{e},\underline{\pi}} \\ \theta(\underline{c}') &= \rho_{\underline{e}',\underline{\pi}'} \end{aligned}$$

allora per ogni $\underline{v}_{i,j} \in S$

$$\theta(\underline{c}) * \theta(\underline{c}')(\underline{v}_{i,j}) = \rho(\underline{e},\underline{\pi}) * \rho(\underline{e}',\underline{\pi}')(\underline{v}_{i,j}) = \alpha^{e_{i,\pi_i'(j)}} \alpha^{e'_{i,j}} \underline{v}_{i,\pi_i\pi_i'(j)} =$$

$$= \theta(((e_{i,\pi_i'(1)} + e'_{i,1}, \ldots, e_{i,\pi_i'(\omega(d_i))} + e'_{i,\omega(d_i)}), \pi_i\pi_i')_{i=1,\ldots,l}) = \theta(\underline{c}\underline{c}')$$

e

$$\theta(\underline{c}^{-1}) = \theta(((-e_{i,\pi_i^{-1}(1)}, \ldots, -e_{i,\pi_i^{-1}(\omega(d_i))}), \pi_i^{-1})_{i=1,\ldots,l}) =$$

$$\rho_{\underline{e},\underline{\pi}}^{-1} = \theta^{-1}(\underline{c})$$

Iniettività e suriettività sono ovvie, essendo $\underline{c}$ completamente determinato da $\underline{e}, \underline{\pi}$ e viceversa. Dunque $B \cong C$ e

$$\mathcal{G}^n(R) \cong \prod_{i=1}^{l} G_i^n$$

come volevasi dimostrare. □

Con la notazione del prodotto semidiretto introdotta a pagina 5.3 si può anche esprimere $\mathcal{G}^n(R)$ direttamente in funzione di gruppi ciclici e simmetrici come

$$\mathcal{G}^n(R) \cong \prod_{d|n} (C_d^{\omega(d)} \otimes_S S_{\omega(d)})$$

Facciamo qualche esempio dell'applicazione del teorema appena dimostrato.

5.7 Esempio

Nel caso di automi di lunghezza 2 definiti su $\mathbb{Z}_2$ si ha $\mathcal{G}^2(\mathbb{Z}_2) = G_1^2 \times G_2^2$ dove, stante la definizione (10), $G_1^2 = \langle((0,0),1),((0,0),(1\ 2))\rangle \cong C_2$ e $G_2^2 = \langle((1),1)\rangle \cong C_2$, e quindi $\mathcal{G}^2(\mathbb{Z}_2) = C_2 \times C_2$, che è uguale al risultato ottenuto per computazione diretta dalle tavole a pagina 68. □

5.8 Esempio

Se consideriamo automi di lunghezza 3 si ottiene $\mathcal{G}^3(\mathbb{Z}_2) = G_1^3 \times G_2^3$ dove

$$G_1^3 = \langle((0,0),1),((0,0),(1\ 2))\rangle \cong C_2$$

e

$$G_2^3 \quad = \quad \langle g, h \mid g^3 = h^2 = (gh)^6 = 1, (gh)^2 = (hg)^2 \rangle$$

dove $g = ((1,0), e)$ e $h = ((0,0), \pi)$ con $\pi = (1\ 2)$. Le tavole di moltiplicazione di G_2^3 sono riportate nelle tabelle seguenti.

$*$	$((0,0),e)$	$((0,0),\pi)$	$((0,1),e)$	$((0,1),\pi)$	$((0,2),e)$	$((0,2),\pi)$	$((1,0),e)$	$((1,0),\pi)$	$((1,1),e)$
$((0,0),e)$	$((0,0),e)$	$((0,0),\pi)$	$((0,1),e)$	$((0,1),\pi)$	$((0,2),e)$	$((0,2),\pi)$	$((1,0),e)$	$((1,0),\pi)$	$((1,1),e)$
$((0,0),\pi)$	$((0,0),\pi)$	$((0,0),e)$	$((0,1),\pi)$	$((0,1),e)$	$((0,2),\pi)$	$((0,2),e)$	$((1,0),\pi)$	$((1,0),e)$	$((1,1),\pi)$
$((0,1),e)$	$((0,1),e)$	$((1,0),\pi)$	$((0,2),e)$	$((1,1),\pi)$	$((0,0),e)$	$((1,2),\pi)$	$((1,1),e)$	$((2,0),\pi)$	$((1,2),e)$
$((0,1),\pi)$	$((0,1),\pi)$	$((1,0),e)$	$((0,2),\pi)$	$((1,1),e)$	$((0,0),\pi)$	$((1,2),e)$	$((1,1),\pi)$	$((2,0),e)$	$((1,2),\pi)$
$((0,2),e)$	$((0,2),e)$	$((2,0),\pi)$	$((0,0),e)$	$((2,1),\pi)$	$((0,1),e)$	$((2,2),\pi)$	$((1,2),e)$	$((0,0),\pi)$	$((1,0),e)$
$((0,2),\pi)$	$((0,2),\pi)$	$((2,0),e)$	$((0,0),\pi)$	$((2,1),e)$	$((0,1),\pi)$	$((2,2),e)$	$((1,2),\pi)$	$((0,0),e)$	$((1,0),\pi)$
$((1,0),e)$	$((1,0),e)$	$((0,1),\pi)$	$((1,1),e)$	$((0,2),\pi)$	$((1,2),e)$	$((0,0),\pi)$	$((2,0),e)$	$((1,1),\pi)$	$((2,1),e)$
$((1,0),\pi)$	$((1,0),\pi)$	$((0,1),e)$	$((1,1),\pi)$	$((0,2),e)$	$((1,2),\pi)$	$((0,0),e)$	$((2,0),\pi)$	$((1,1),e)$	$((2,1),\pi)$
$((1,1),e)$	$((1,1),e)$	$((1,1),\pi)$	$((1,2),e)$	$((1,2),\pi)$	$((1,0),e)$	$((1,0),\pi)$	$((2,1),e)$	$((2,1),\pi)$	$((2,2),e)$
$((1,1),\pi)$	$((1,1),\pi)$	$((1,1),e)$	$((1,2),\pi)$	$((1,2),e)$	$((1,0),\pi)$	$((1,0),e)$	$((2,1),\pi)$	$((2,1),e)$	$((2,2),\pi)$
$((1,2),e)$	$((1,2),e)$	$((2,1),\pi)$	$((1,0),e)$	$((2,2),\pi)$	$((1,1),e)$	$((2,0),\pi)$	$((2,2),e)$	$((0,1),\pi)$	$((2,0),e)$
$((1,2),\pi)$	$((1,2),\pi)$	$((2,1),e)$	$((1,0),\pi)$	$((2,2),e)$	$((1,1),\pi)$	$((2,0),e)$	$((2,2),\pi)$	$((0,1),e)$	$((2,0),\pi)$
$((2,0),e)$	$((2,0),e)$	$((0,2),\pi)$	$((2,1),e)$	$((0,0),\pi)$	$((2,2),e)$	$((0,1),\pi)$	$((0,0),e)$	$((1,2),\pi)$	$((0,1),e)$
$((2,0),\pi)$	$((2,0),\pi)$	$((0,2),e)$	$((2,1),\pi)$	$((0,0),e)$	$((2,2),\pi)$	$((0,1),e)$	$((0,0),\pi)$	$((1,2),e)$	$((0,1),\pi)$
$((2,1),e)$	$((2,1),e)$	$((1,2),\pi)$	$((2,2),e)$	$((1,0),\pi)$	$((2,0),e)$	$((1,1),\pi)$	$((0,1),e)$	$((2,2),\pi)$	$((0,2),e)$
$((2,1),\pi)$	$((2,1),\pi)$	$((1,2),e)$	$((2,2),\pi)$	$((1,0),e)$	$((2,0),\pi)$	$((1,1),e)$	$((0,1),\pi)$	$((2,2),e)$	$((0,2),\pi)$
$((2,2),e)$	$((2,2),e)$	$((2,2),\pi)$	$((2,0),e)$	$((2,0),\pi)$	$((2,1),e)$	$((2,1),\pi)$	$((0,2),e)$	$((0,2),\pi)$	$((0,0),e)$
$((2,2),\pi)$	$((2,2),\pi)$	$((2,2),e)$	$((2,0),\pi)$	$((2,0),e)$	$((2,1),\pi)$	$((2,1),e)$	$((0,2),\pi)$	$((0,2),e)$	$((0,0),\pi)$

$*$	$((1,1),\pi)$	$((1,2),e)$	$((1,2),\pi)$	$((2,0),e)$	$((2,0),\pi)$	$((2,1),e)$	$((2,1),\pi)$	$((2,2),e)$	$((2,2),\pi)$
$((0,0),e)$	$((1,1),\pi)$	$((1,2),e)$	$((1,2),\pi)$	$((2,0),e)$	$((2,0),\pi)$	$((2,1),e)$	$((2,1),\pi)$	$((2,2),e)$	$((2,2),\pi)$
$((0,0),\pi)$	$((1,1),e)$	$((1,2),\pi)$	$((1,2),e)$	$((2,0),\pi)$	$((2,0),e)$	$((2,1),\pi)$	$((2,1),e)$	$((2,2),\pi)$	$((2,2),e)$
$((0,1),e)$	$((2,1),\pi)$	$((1,0),e)$	$((2,2),\pi)$	$((2,1),e)$	$((0,0),\pi)$	$((2,2),e)$	$((0,1),\pi)$	$((2,0),e)$	$((0,2),\pi)$
$((0,1),\pi)$	$((2,1),e)$	$((1,0),\pi)$	$((2,2),e)$	$((2,1),\pi)$	$((0,0),e)$	$((2,2),\pi)$	$((0,1),e)$	$((2,0),\pi)$	$((0,2),e)$
$((0,2),e)$	$((0,1),\pi)$	$((1,1),e)$	$((0,2),\pi)$	$((2,2),e)$	$((1,0),\pi)$	$((2,0),e)$	$((1,1),\pi)$	$((2,1),e)$	$((1,2),\pi)$
$((0,2),\pi)$	$((0,1),e)$	$((1,1),\pi)$	$((0,2),e)$	$((2,2),\pi)$	$((1,0),e)$	$((2,0),\pi)$	$((1,1),e)$	$((2,1),\pi)$	$((1,2),e)$
$((1,0),e)$	$((1,2),\pi)$	$((2,2),e)$	$((1,0),\pi)$	$((0,0),e)$	$((2,1),\pi)$	$((0,1),e)$	$((2,2),\pi)$	$((0,2),e)$	$((2,0),\pi)$
$((1,0),\pi)$	$((1,2),e)$	$((2,2),\pi)$	$((1,0),e)$	$((0,0),\pi)$	$((2,1),e)$	$((0,1),\pi)$	$((2,2),e)$	$((0,2),\pi)$	$((2,0),e)$
$((1,1),e)$	$((2,2),\pi)$	$((2,0),e)$	$((2,0),\pi)$	$((0,1),e)$	$((0,1),\pi)$	$((0,2),e)$	$((0,2),\pi)$	$((0,0),e)$	$((0,0),\pi)$
$((1,1),\pi)$	$((2,2),e)$	$((2,0),\pi)$	$((2,0),e)$	$((0,1),\pi)$	$((0,1),e)$	$((0,2),\pi)$	$((0,2),e)$	$((0,0),\pi)$	$((0,0),e)$
$((1,2),e)$	$((0,2),\pi)$	$((2,1),e)$	$((0,0),\pi)$	$((0,2),e)$	$((1,1),\pi)$	$((0,0),e)$	$((1,2),\pi)$	$((0,1),e)$	$((1,0),\pi)$
$((1,2),\pi)$	$((0,2),e)$	$((2,1),\pi)$	$((0,0),e)$	$((0,2),\pi)$	$((1,1),e)$	$((0,0),\pi)$	$((1,2),e)$	$((0,1),\pi)$	$((1,0),e)$
$((2,0),e)$	$((1,0),\pi)$	$((0,2),e)$	$((1,1),\pi)$	$((1,0),e)$	$((2,2),\pi)$	$((1,1),e)$	$((2,0),\pi)$	$((1,2),e)$	$((2,1),\pi)$
$((2,0),\pi)$	$((1,0),e)$	$((0,2),\pi)$	$((1,1),e)$	$((1,0),\pi)$	$((2,2),e)$	$((1,1),\pi)$	$((2,0),e)$	$((1,2),\pi)$	$((2,1),e)$
$((2,1),e)$	$((2,0),\pi)$	$((0,0),e)$	$((2,1),\pi)$	$((1,1),e)$	$((0,2),\pi)$	$((1,2),e)$	$((0,0),\pi)$	$((1,0),e)$	$((0,1),\pi)$
$((2,1),\pi)$	$((2,0),e)$	$((0,0),\pi)$	$((2,1),e)$	$((1,1),\pi)$	$((0,2),e)$	$((1,2),\pi)$	$((0,0),e)$	$((1,0),\pi)$	$((0,1),e)$
$((2,2),e)$	$((0,0),\pi)$	$((0,1),e)$	$((0,1),\pi)$	$((1,2),e)$	$((1,2),\pi)$	$((1,0),e)$	$((1,0),\pi)$	$((1,1),e)$	$((1,1),\pi)$
$((2,2),\pi)$	$((0,0),e)$	$((0,1),\pi)$	$((0,1),e)$	$((1,2),\pi)$	$((1,2),e)$	$((1,0),\pi)$	$((1,0),e)$	$((1,1),\pi)$	$((1,1),e)$

dove $\{e,\pi\} \cong S_2 \cong C_2$. □

5.4 Sottogruppi di G_i^n

Abbiamo visto nell'equazione (12) che i generatori di G_i^n sono dati da g_i e $h_{i,f}$ dove $1 \leq f \leq \frac{1}{2}\omega(d_i)(\omega(d_i)-1)$, e abbiamo mostrato che, se η è l'identità di $S_{\omega(d_i)}$, per ogni permutazione del tipo α^w di $(1,0,\ldots,0)$ esiste un f tale che

$$h_{i,f}^{-1} g_i h_{i,f} = (\alpha^w(1,0,\ldots,0),\eta)$$

Ora, il gruppo ciclico generato da ognuno di questi elementi è C_{d_i} e ce ne sono $\omega(d_i)$ distinti, quindi

$$H' = \underbrace{C_{d_i} \times \ldots \times C_{d_i}}_{\omega(d_i)}$$

è isomorfo ad un sottogruppo H di G_i^n. Inoltre, se $h \in H$ e $g \in G_i^n$ tali che

$$\begin{aligned} h &= ((e_1, \ldots, e_z), \eta) \\ g &= ((f_1, \ldots, f_z), \pi) \end{aligned}$$

si ottiene, per interi $p_1, \ldots, p_z$,

$$g^{-1}hg = ((p_1, \ldots, p_z), \eta) \in H$$

e quindi $H \lhd G_i^n$; pertanto esiste un endomorfismo θ di G_i^n tale che $\mathrm{Ker}\theta = H$, e non è difficile vedere che $\mathrm{Im}\theta \cong G_i^n/H \cong S_{\omega(d_i)}$.

Queste informazioni sui sottogruppi di G_i^n possono essere usate come ausilio alla costruzione della tavola dei caratteri di G_i^n: si ricorda infatti che per due gruppi G, H tali che esista un omomorfismo $\phi : G \to H$ e tali che ρ sia una rappresentazione di H, $\rho \circ \theta$ è una rappresentazione di G.

5.5 $\mathcal{G}^n(R)$ come gruppo di permutazioni

È noto che ogni gruppo finito è isomorfo ad un sottogruppo di qualche gruppo simmetrico. Svilupperemo un'ulteriore rappresentazione delle trasformazioni cellulari di $\mathcal{A}^n(R)$, che chiameremo **rappresentazione globale**, per contrapporla alla rappresentazione locale, che metterà in luce questo aspetto del problema. Introduciamo innanzitutto due funzioni, una inversa dell'altra,

$$\begin{aligned} &\mathrm{vect} : R[X] \to R^n \\ &\mathrm{vect}(v_1X^{n-1} + \cdots + v_{n-1}X + v_n) = (v_1, \ldots, v_n) \\ &\mathrm{dec} : R^n \to R[X] \\ &\mathrm{dec}((v_1, \ldots, v_n)) = v_1X^{n-1} + \cdots + v_{n-1}X + v_n \end{aligned}$$

Queste funzioni sono particolarmente utili quando R è un anello numerico, per esempio $R = \mathbb{Z}_m$, nel qual caso assegneremo a X il valore $|R|$. In pratica vect trasforma un numero decimale nel corrispondente numero in base $|R|$, e dec compie l'operazione inversa. Abbiamo visto nel capitolo 3 che la rappresentazione locale di una trasformazione $\sigma \in \mathcal{A}^n(R)$ assegna a σ un vettore

$$(v_{|R|^n-1}, \ldots, v_0) \quad \text{dove } \forall i \ \text{vect}(v_i) \in R$$

il cui significato è che se l'intorno della i-esima posizione dell'automa cellulare al tempo t è $\underline{a} = (a_1, \ldots, a_n)$ allora l'i-esima posizione al tempo $t+1$ assume il valore $v_{\text{dec}(\underline{a})}$. La rappresentazione globale, invece, assegna a σ un vettore della forma

$$(v_{|R|^n-1}, \ldots, v_0) \quad \text{dove } \forall i \ \text{vect}(v_i) \in R^n \tag{13}$$

tale che se l'automa cellulare al tempo t è dato dal vettore $\underline{a} = (a_1, \ldots, a_n)$ allora l'automa cellulare al tempo $t+1$ è dato da $\text{vect}(v_{\text{dec}(\underline{a})})$; ovvero

$$\sigma(\underline{a}) = \text{vect}(v_{\text{dec}(\underline{a})})$$

Per esempio, la trasformazione identità di $\mathcal{A}^3(\mathbb{Z}_2)$ è $(7,6,5,4,3,2,1,0)$. La trasformazione che manda tutto in zero è, ovviamente, $(0,0,0,0,0,0,0,0)$.

Si noti che non tutti i vettori della forma (13) sono rappresentazioni globali di trasformazioni cellulari in $\mathcal{A}^n(R)$: ad esempio, prendendo il caso $\mathcal{A}^3(\mathbb{Z}_2)$, supponiamo per assurdo che il vettore $(7,6,5,4,3,1,1,0)$ sia una rappresentazione globale di σ. Allora si avrebbe

$$\sigma((0,0,1)) = \sigma(\text{vect}(1)) = \text{vect}(1) = (0,0,1)$$

e

$$\sigma((0,1,0)) = \sigma(\text{vect}(2)) = \text{vect}(1) = (0,0,1)$$

da cui, essendo $\alpha = (123) \in S_3$,

$$\sigma(\alpha(0,0,1)) = \sigma(0,1,0) = (0,0,1) \neq (0,1,0) = \alpha(0,0,1) = \alpha\sigma(0,0,1)$$

e quindi si viene meno al requisito[13] che $\sigma\alpha = \alpha\sigma$, perciò σ non è una trasformazione cellulare. Perché un vettore della forma (13) sia in effetti una rappresentazione globale, è necessario verificare che la trasformazione rappresentata commuti con α.

A questo punto, imporre le condizioni per cui σ sia invertibile è immediato: $\sigma \in \mathcal{G}^n(R)$ se e solo se la rappresentazione globale di σ, che indicheremo con $\rho(\sigma)$, è una permutazione; ovvero, se e solo se $\rho(\sigma) \in S_{|R|^n}$. Di qui otteniamo immediatamente che

$$\mathcal{G}^n(R) \cong \{\sigma \in S_{|R|^n} \mid \sigma\alpha = \alpha\sigma\}$$

dove

$$\sigma(\alpha(v_1, \ldots, v_n)) = \sigma(\mathrm{dec}(v_2, \ldots, v_n, v_1))$$

e

$$\alpha\sigma((v_1, \ldots, v_n)) = \alpha\mathrm{vect}(\sigma(\mathrm{dec}((v_1, \ldots, v_n))))$$

In conclusione, $\mathcal{G}^n(R)$ è isomorfo a un sottogruppo di permutazioni di $S_{|R|^n}$.

La seguente osservazione può facilitare le operazioni numeriche se R è un anello numerico: se $\underline{u}$ appartiene a R^n, si ottiene

$$\alpha(\underline{u}) = (|R| \ \mathrm{dec}(\underline{u})) \mod (|R|^n - 1)$$

[13] Cfr. Teorema (2.3)

5.6 Altre applicazioni della rappresentazione globale

5.6.1 Prodotto di rappresentazioni globali

Usando le rappresentazioni globali diventa particolarmente facile calcolare il prodotto di due trasformazioni di $\mathcal{A}^n(R)$: se

$$\rho(\sigma) = (u_{|R|^n-1}, \ldots, u_0)$$

e

$$\rho(\tau) = (v_{|R|^n-1}, \ldots, v_0)$$

allora

$$\rho(\sigma * \tau) = (v_{u_{|R|^n-1}}, \ldots, v_{u_0})$$

e in particolare, se σ e τ sono invertibili, è possibile servirsi delle permutazioni in forma di cicli disgiunti.

5.9 Esempio

In $\mathcal{G}^3(\mathbb{Z}_2)$, moltiplichiamo la trasformazione $\sigma_{1,1}$ con rappresentazione decimale 15 e la trasformazione $\tau_{1,1}$ con rappresentazione decimale 85 (le rappresentazioni decimali sono calcolate su $(1,1)$-intorni):

$$\rho(\tau_{1,1}) = (0,2,4,6,1,3,5,7)$$

è evidente che $\rho(\tau_{1,1})$ è la permutazione

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 7 & 5 & 3 & 1 & 6 & 4 & 2 & 0 \end{pmatrix}$$

che in forma di cicli disgiunti si scrive anche $(0,7)(1,5,4,6,2,3)$. Per 15 il discorso è analogo, e si ottiene

$$\rho(\sigma_{1,1}) = (0,7)(1,3,2,6,4,5)$$

Eseguendo il prodotto, abbiamo

$$(0,7)(1,3,2,6,4,5)(0,7)(1,5,4,6,2,3) = 1 \in S_8$$

da cui concludiamo $85 * 15 = 204$, che è la rappresentazione decimale (sempre con $(1,1)$-intorni) dell'identità del gruppo degli invertibili. □

5.6.2 Conversione alla rappresentazione locale

Passare dalla rappresentazione globale su (r_0, r_1)-intorni alla rappresentazione locale di una trasformazione cellulare è quasi immediato: se

$$\rho(\sigma) = (v_{|R|^n-1}, \ldots, v_0)$$

allora basta prendere la (r_0+1)-esima componente del vettore $\text{vect}(v_i)$ per sapere come si evolverà una posizione il cui intorno è dato da $\text{vect}(i)$: e dunque si ha che la rappresentazione locale di σ è data da

$$(\lambda(\text{vect}(v_{|R|^n-1}), r_0+1), \ldots, \lambda(\text{vect}(v_0), r_0+1))$$

dove $\lambda(\underline{u}, j)$ è la j-esima componente del vettore $\underline{u}$.

Abbiamo visto nel capitolo 3 che dato un vettore $\underline{u}$ in R^m vale la seguente relazione:

$$\lambda(\underline{u}, j) = \left\lfloor \frac{\text{dec}(\underline{u})}{|R|^{n-j}} \right\rfloor \mod |R| \tag{14}$$

dove con $\lfloor a \rfloor$ indichiamo la parte intera di a. La dimostrazione di questo fatto non è difficile:

$$\left\lfloor \frac{\text{dec}(\underline{u})}{|R|^{n-j}} \right\rfloor \mod |R| = \left\lfloor \frac{u_1|R|^{n-1} + \cdots + u_{n-1}|R| + u_n}{|R|^{n-j}} \right\rfloor \mod |R| =$$

$$= \left\lfloor u_1|R|^{j-1} + \cdots + u_{j-1}|R| + u_j + u_{j+1}|R|^{-1} + \cdots + u_n|R|^{j-n} \right\rfloor \mod |R| =$$

$$= ((u_1|R|^{j-1} + \cdots + u_{j-1}|R| + u_j) + \left\lfloor u_{j+1}|R|^{-1} + \cdots + u_n|R|^{j-n} \right\rfloor) \mod |R| =$$

$$= (u_1|R|^{j-1} + \cdots + u_{j-1}|R| + u_j) \mod |R| = u_j = \lambda(\underline{u}, j)$$

Dunque possiamo scrivere la rappresentazione locale di σ in forma più semplice come

$$\left(\left\lfloor \frac{v_{|R|^n-1}}{|R|^{n-r_0-1}} \right\rfloor \mod |R|, \ldots, \left\lfloor \frac{v_0}{|R|^{n-r_0-1}} \right\rfloor \mod |R| \right)$$

Si noti che questa rappresentazione locale implica un raggio destro

$$r_1 = n - r_0 - 1$$

ma non è detto che sia il raggio destro minimo possibile. Per esempio, la trasformazione di $\mathcal{A}^4(\mathbb{Z}_2)$ data dalla rappresentazione locale

$$(1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0)$$

è la stessa trasformazione data dalla rappresentazione locale

$$(1, 0, 1, 0, 1, 0, 1, 0)$$

presa in $\mathcal{B}^4_{1,1}(\mathbb{Z}_2)$, ovvero una regola con raggio sinistro 1, raggio destro 1 ma applicata a vettori di lunghezza 4. Si vedano anche, a questo proposito, i paragrafi 2.3, 3.1, 3.3.

5.6.3 Invertibilità e rappresentazioni globali

Abbiamo visto nella sezione 5.5 che si può considerare il gruppo delle trasformazioni cellulari invertibili come un gruppo di permutazioni:

$$\mathcal{G}^n(R) \cong \{\sigma \in S_{|R|^n} \mid \sigma\alpha = \alpha\sigma\}$$

Si può sfruttare questo fatto per implementare un algoritmo estremamente veloce allo scopo di determinare se una data trasformazione è invertibile e di trovarne l'inversa. L'idea di base è:

- Richiedere in input i seguenti dati: dimensione di $|R|$, lunghezza n degli automi cellulari considerati, rappresentazione locale della trasformazione di cui si deve investigare l'invertibilità e raggi sinistro r_0 e destro r_1 dell'intorno.

- Trasformare la rappresentazione locale nella rappresentazione globale. La strategia per ottenere ciò consiste nell'applicare la trasformazione a tutti i vettori vect(i) per $0 \leq i < |R|^n$ (cioè tutti i vettori di R^n).

- Verificare se la rappresentazione globale è una permutazione o no; se non lo è, la trasformazione non è invertibile; se lo è, l'inverso della trasformazione è facilmente determinabile dalla rappresentazione globale.

- Come output, produrre rappresentazione globale e locale dell'inversa; la rappresentazione locale deve possibilmente essere ridotta in modo che il raggio destro sia il minimo possibile.

L'algoritmo lavora sulle rappresentazioni globali e non su quelle orbitali, e perciò ignora volutamente la commutatività delle trasformazioni cellulari con α (si veda il teorema (2.3)). La ridondanza di informazioni che ne segue è il punto forte dell'algoritmo perché lo rende molto veloce, ovviamente a spese della quantità di memoria richiesta. L'algoritmo è stato implementato nel programma `CAInvPrm.cpp` incluso nell'appendice A, che ha permesso, su un processore Intel Pentium 100 MHz con 16 Mb RAM, di ottenere in tempi piuttosto brevi (nell'ordine del minuto) informazioni sull'invertibilità di trasformazioni in $\mathcal{B}^{18}_{1,1}(\mathbb{Z}_2)$. Il limite 18 è stato imposto dalla quantità di memoria a disposizione e non dal tempo richiesto; è inoltre possibile ottimizzare ulteriormente l'uso della memoria in `CAInvPrm`. Sono stati fatti dei test analoghi anche su algoritmi basati sulla rappresentazione orbitale, e il risultato è stato che a fronte di una richiesta di memoria quasi nulla i tempi di calcolo sono diventati immediatamente enormi, con una crescita apparentemente più che esponenziale.

6 La Rappresentazione Locale

Abbiamo già detto nel capitolo 1 che all'atto pratico uno dei problemi più interessanti nello studio degli automi cellulari è stabilire se una data trasformazione cellulare sia invertibile o meno. Nel capitolo 5 abbiamo analizzato la struttura del gruppo delle trasformazioni cellulari invertibili, ma le limitazioni dei metodi usati sono enormi, almeno sul piano del calcolo numerico: essendo il tempo di calcolo e la quantità di memoria richiesta proporzionali ad un esponenziale della lunghezza dell'automa, sarebbe impensabile implementare gli algoritmi descritti, anche avendo a disposizione dei supercomputers, ad automi più lunghi di una trentina di celle. Le griglie più piccole di simulazione per il comportamento dei fluidi basate sugli automi cellulari hanno dimensioni dell'ordine del milione di elementi: questo significa che la teoria sviluppata finora ha significato dal punto di vista accademico ed estetico, ma non è utilizzabile. La ragione di queste limitazioni è che finora abbiamo usato rappresentazioni delle trasformazioni cellulari basate sul concetto — più o meno velato — di funzione $R^n \to R^n$: la trasformazione cellulare σ manda il vettore $\{v_1, \ldots, v_n\}$ ad un altro vettore $\{v'_1, \ldots, v'_n\}$, dove i vettori considerati hanno tutti la lunghezza dell'automa. In realtà la forma più diffusa e più utile di rappresentazione è quella locale: si descrive come si evolve una cella in funzione di un suo intorno, che solitamente è piccolo rispetto alla lunghezza dell'automa. Il vantaggio di questa tecnica è che la descrizione di una trasformazione dipende solo dalla forma dell'intorno e non dalla lunghezza dell'automa, e quindi una regola descritta tramite una rappresentazione locale è applicabile ad automi di lunghezza arbitraria; lo svantaggio è che in generale il metodo più breve per predire il comportamento di una trasformazione cellulare qualsiasi è quello di simularla[14], ovvero applicarla a

[14]In realtà esistono delle particolari trasformazioni cellulari non-lineari il cui comportamento può essere predetto in tempi più brevi della simulazione diretta: vengono dette

tutte le configurazioni possibili e ricordare i risultati, e abbiamo visto che questo è proibitivo in termini di tempo di calcolo e memoria, se non per i casi più semplici.

In questo capitolo cercheremo di analizzare alcune sottoclassi di trasformazioni invertibili di $\mathcal{A}^n(R)$ che rimangono invertibili anche se applicate ad automi di lunghezza maggiore di n. Faremo uso della rappresentazione locale e dei sottoinsiemi $\mathcal{B}^n_{r_0,r_1}(R)$ di $\mathcal{A}^n(R)$. Si ricorda che per definizione (si veda il capitolo 2 e i paragrafi 2.3 e 3.3) la rappresentazione locale di una trasformazione di $\mathcal{B}^n_{r_0,r_1}(R)$ è un vettore di $R^{|R|^{r_0+r_1+1}}$ la cui j-esima componente è l'evoluzione di una cella della quale vect(j) rappresenta l'intorno.

6.1 Esempio

L'identità di $\mathcal{A}^3(\mathbb{Z}_2)$ ha rappresentazione locale calcolata su $(1,1)$-intorni

$$(1,1,0,0,1,1,0,0)$$

che vuol dire che

$(1,1,1)$	$(1,1,0)$	$(1,0,1)$	$(1,0,0)$	$(0,1,1)$	$(0,1,0)$	$(0,0,1)$	$(0,0,0)$
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
1	1	0	0	1	1	0	0

Si può notare una ridondanza di informazioni: per ogni v_1, v_2 si ha che la trasformazione manda $(v_1, v_2, 0)$ e $(v_1, v_2, 1)$ allo stesso elemento: pertanto risulta evidente l'equivalenza con la rappresentazione locale $(1,0,1,0)$ calcolata su $(1,0)$-intorni

$(1,1)$	$(1,0)$	$(0,1)$	$(0,0)$
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
1	0	1	0

Se ne può concludere che l'identità di $\mathcal{A}^3(\mathbb{Z}_2)$ appartiene al sottoinsieme $\mathcal{B}^3_{1,0}(\mathbb{Z}_2)$. □

quasilineari e sono state descritte da un *team* di ricerca dell'Università di Santa Fe. Cfr. [Moo97, MD96].

6.2 Esempio

Per la stessa ragione, la trasformazione di $\mathcal{A}^4(\mathbb{Z}_2)$ data dalla rappresentazione locale su $(1,2)$-intorni

$$(1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0)$$

ha anche, come rappresentazioni locali rispettivamente su $(1,1)$- e $(1,0)$-intorni,

$$(1,1,1,1,0,0,0,0) \text{ e } (1,1,0,0)$$

Poiché i raggi destro e sinistro degli intorni di una trasformazione non possono essere negativi, non è ammessa una rappresentazione locale $(1,0)$, il cui intorno di definizione dovrebbe avere un raggio destro uguale a -1. Si noti che il processo di "riduzione del raggio destro è indipendente dalla lunghezza degli automi considerati. La rappresentazione locale più corta possibile di una trasformazione cellulare determina il raggio destro minimo dell'intorno: se l è la lunghezza del vettore della rappresentazione locale più corta, il raggio destro minimo $\overline{r_1}$ è dato da

$$\overline{r_1} = \log_{|R|}(l) - r_0 - 1$$

Un limite superiore al raggio destro è dato dal fatto che $n \geq r_0 + r_1 + 1$. □

6.1 Indipendenza dalla lunghezza dell'automa

Per comodità di notazione nei teoremi che seguono, introduciamo le funzioni

$$\mathrm{loc}_{r_0,r_1} : \mathcal{A}^n(R) \to R^{|R|^{r_0+r_1+1}}$$

tale che se $\sigma \in \mathcal{A}^n(R)$, $\mathrm{loc}_{r_0,r_1}(\sigma)$ è il vettore che ne indica la rappresentazione locale calcolata su (r_0, r_1)-intorni dove $r_1 \in \{r \mid \sigma \in \mathcal{B}^n_{r_0,r}(R)\}$ e

$$\varepsilon^k_{r_0,r_1} : R^{|R|^{r_0+r_1+1}} \to \mathcal{A}^k(R)$$

dove $\varepsilon^k_{r_0,r_1}(\underline{v})$ è la trasformazione $\tau \in \mathcal{A}^k(R)$ tale che $\mathrm{loc}_{r_0,r_1}(\tau) = \underline{v}$, cioè $\varepsilon^k_{r_0,r_1}(\underline{v})$ è l'estensione su (r_0, r_1)-intorni della regola locale data da $\underline{v}$.

Sia α_n la permutazione $(12\ldots n)$ del gruppo ciclico C_n vista come funzione $R^n \to R^n$ (lo *shift*). Abbiamo visto nella sezione 1.1 che le funzioni date da

$$\alpha^k_n(v_1, \ldots, v_n) = (v_{\alpha^k_n(1)}, \ldots, v_{\alpha^k_n(n)}) = (v_{k+1}, \ldots, v_k)$$

sono trasformazioni cellulari di $\mathcal{A}^n(R)$ per ogni $k < n$, e che formano un sottogruppo di $\mathcal{G}^n(R)$ isomorfo a C_n.

6.3 Proposizione

Per ogni $j < n$, la rappresentazione locale di α^j_n calcolata su $(r_0, n - r_1 - 1)$-intorni è data da

$$\left(\left\lfloor \frac{|R|^n - 1}{|R|^{n-j-r_0-1}} \right\rfloor \mod |R|, \ldots, \left\lfloor \frac{i}{|R|^{n-j-r_0-1}} \right\rfloor \mod |R|, \ldots, 0 \right) \qquad (15)$$

Dimostrazione. Sia $\underline{v} = (v_1, \ldots, v_n) \in R^n$. Allora

$$\alpha^j_n \underline{v} = (v_{j+1}, \ldots, v_{j+r_0+1}, \ldots, v_j)$$

da cui

$$\lambda(\alpha^j_n(v_1, \ldots, v_n), r_0 + 1) = v_{j+r_0+1} \qquad (16)$$

dove come al solito $\lambda((v_1, \ldots v_n), j) = v_j$, e perciò

$$\mathrm{loc}_{r_0,n-r_0-1}(\alpha^j_n) =$$

$$= (\lambda(|R|^n - 1, j + r_0 + 1), \ldots, \lambda(i, j + r_0 + 1), \ldots, \lambda(\mathrm{vect}(0), j + r_0 + 1))$$

Per l'espressione (14) a pagina 84,

$$\lambda(\underline{v}, j + r_0 + 1) = \left\lfloor \frac{\mathrm{dec}(\underline{v})}{|R|^{n-j-r_0-1}} \right\rfloor \mod |R|$$

da cui la tesi. □

Calcoliamo il raggio destro minimo della rappresentazione locale data da (15). Per (16), se fissiamo $v_1, \ldots, v_j$, per qualsiasi $v_{j+1}, \ldots, v_n$ si ha

$$\lambda(\alpha_n^j(v_1, \ldots, v_n), r_0 + 1) = v_{j+r_0+1}$$

e quindi il raggio destro minimo non può essere minore di j.

Definiamo ora, per ogni permutazione $\pi \in S_{|R|}$, la funzione

$$\pi_n(v_1, \ldots, v_n) = (\pi(v_1), \ldots, \pi(v_n))$$

Si noti che π_n è chiaramente una biiezione $R^n \to R^n$ con inversa π_n^{-1}.

6.4 Lemma

$$\pi_n \alpha_n = \alpha_n \pi_n$$

Dimostrazione. Per ogni $\underline{v} = (v_1, \ldots, v_n) \in R^n$ si ha

$$\begin{aligned} \pi_n\alpha_n(\underline{v}) &= \pi_n(v_2, \ldots, v_n, v_1) = \\ &= (\pi(v_2), \ldots, \pi(v_n), \pi(v_1)) = \\ &= \alpha_n(\pi(v_1), \ldots, \pi(v_n)) \end{aligned}$$

che è uguale a $\alpha_n\pi_n(\underline{v})$. □

Pertanto per il teorema (2.3) $\pi_n \in \mathcal{A}^n(R)$. Da questo segue che π_n commuta con tutte le potenze di α_n. Sia

$$\mathcal{H}^n(R) = \{\alpha_n^j \pi_n \mid 0 \leq j < n, \pi \in S_{|R|}\}$$

Si dimostra facilmente che $\mathcal{H}^n(R) \cong C_n \times S_{|R|}$: sia θ la mappa data da $\theta(\alpha_n^j \pi_n) = (\alpha_n^j, \pi)$. Per la commutatività di π_n con α_n è immediato provare che θ è una biiezione, quindi $\mathcal{H}^n(R)$ è un sottogruppo di $\mathcal{G}^n(R)$. Il motivo

dell'importanza di questo gruppo è che i suoi elementi mantengono la stessa rappresentazione locale quando vengono applicati ad automi di lunghezza diversa. Più precisamente se da un elemento di $\alpha_n^j\pi_n$ di $\mathcal{H}^n(R)$ calcoliamo la sua rappresentazione locale ϕ su $(r_0, n-r_0-1)$-intorni e poi estendiamo ϕ ad una trasformazione $\sigma \in \mathcal{G}^k(R)$ con $k \geq n$ otteniamo che $\sigma = \alpha_k^j\pi_k$.

6.5 Teorema

Sia j tale che $0 \leq j < n$ e sia $\overline{r_1}$ il raggio destro minimo di $\alpha_n^j\pi_n$. Allora per ogni intero $k \geq n$ si ha che

$$\varepsilon^k_{r_0,\overline{r_1}}(\mathrm{loc}_{r_0,n-r_0-1}(\alpha_n^j\pi_n)) = \alpha_k^j\pi_k$$

Dimostrazione. Procediamo per induzione: dimostriamo il teorema quando $k = n+1$, ovvero quando

$$\varepsilon^{n+1}_{r_0,\overline{r_1}}(\mathrm{loc}(\alpha_n^j\pi_n)) = \alpha_{n+1}^j\pi_{n+1}$$

Da ciò seguirà la tesi per ogni k. Le rappresentazioni locali con raggio destro massimo di $\alpha_n^j\pi_n$ e di $\alpha_{n+1}^j\pi_{n+1}$ sono rispettivamente

$$\mathrm{loc}_{r_0,n-r_0-1}(\alpha_n^j\pi_n) = (a_{|R|^n-1}, \ldots, a_0)$$

dove $a_i = \pi\left(\left\lfloor \frac{i}{|R|^{n-j-r_0-1}} \right\rfloor \mod |R|\right)$ e

$$\mathrm{loc}_{r_0,n-r_0}(\alpha_{n+1}^j\pi_{n+1}) = (b_{|R|^{n+1}-1}, \ldots, b_0)$$

dove $b_i = \pi\left(\left\lfloor \frac{i}{|R|^{n-j-r_0}} \right\rfloor \mod |R|\right)$. Ora, per ogni $l < |R|$ e per ogni $i < |R|^n$, si ha

$$b_{|R|i+l} = \pi\left(\left\lfloor \frac{|R|i+l}{|R|^{n-j-r_0}} \right\rfloor \mod |R|\right) =$$

$$= \pi\left(\left(\left\lfloor \frac{|R|i}{|R|^{n-j-r_0}} \right\rfloor + \left\lfloor \frac{k}{|R|^{n-j-r_0}} \right\rfloor\right) \mod |R|\right) =$$

$$= \pi\left(\left\lfloor \frac{i}{|R|^{n-j-r_0-1}} \right\rfloor \mod |R|\right) = \pi(a_i)$$

Dunque la rappresentazione locale con raggio destro $n - r_0$ di $\alpha_{n+1}^j \pi_{n+1}$ è data da

$$(\underbrace{a_{|R|^n-1}, \ldots, a_{|R|^n-1}}_{|R|}, \ldots, \underbrace{a_0, \ldots, a_0}_{|R|})$$

e quindi si può scrivere con raggio destro $n - r_0 - 1$ cosicché

$$\mathrm{loc}_{r_0,n-r_0-1}(\alpha_{n+1}^j \pi_n) = \mathrm{loc}_{r_0,n-r_0-1}(\alpha_n^j \pi_{n+1})$$

Ora, la riduzione della rappresentazione locale al raggio destro minimo non dipende dalla lunghezza degli automi considerati, perciò si ha

$$\mathrm{loc}_{r_0,\overline{r_1}}(\alpha_{n+1}^j \pi_{n+1}) = \mathrm{loc}_{r_0,\overline{r_1}}(\alpha_n^j \pi)$$

da cui la tesi. □

Le trasformazioni del gruppo $\mathcal{H}^n(R) \cong C_n \times S_{|R|}$ vengono chiamate **trasformazioni invertibili banali** [15], in quanto l'immagine mediante la rappresentazione locale di un intorno qualsiasi dipende da una sola componente dell'intorno; ovvero, essendo ϕ la rapprentazione locale di una trasformazione invertibile banale, si ha che

$$\phi(x_1, \ldots, x_n) = \pi(x_j)$$

dove $\pi \in S_{|R|}$ e $j \leq n$ sono indipendenti da $x_1, \ldots, x_n$.

È sufficiente un esempio per provare che in taluni casi non ci sono altri elementi di $\mathcal{G}^n(R)$, a parte quelli di $\mathcal{H}^n(R)$, con queste proprietà.

6.6 Esempio

Applicando le rappresentazioni locali delle trasformazioni in $\mathcal{G}^3(\mathbb{Z}_2)$, calcolate su $(1,1)$-intorni, ad automi di lunghezza 4, si ottengono i seguenti risultati.

[15] Cfr. [AP72]

Rappr. decimale	Rappr. locale	Elemento di $C_n \times S_{\lvert R\rvert}$	Invertibile
15	(0,0,0,0,1,1,1,1)	$(123)^2(01)$	Sì
27	(0,0,0,1,1,0,1,1)		No
29	(0,0,0,1,1,1,0,1)		No
39	(0,0,1,0,0,1,1,1)		No
43	(0,0,1,0,1,0,1,1)		No
45	(0,0,1,0,1,1,0,1)		No
51	(0,0,1,1,0,0,1,1)	(01)	Sì
53	(0,0,1,1,0,1,0,1)		No
57	(0,0,1,1,1,0,0,1)		No
71	(0,1,0,0,0,1,1,1)		No
75	(0,1,0,0,1,0,1,1)		No
77	(0,1,0,0,1,1,0,1)		No
83	(0,1,0,1,0,0,1,1)		No
85	(0,1,0,1,0,1,0,1)	(123)(01)	Sì
89	(0,1,0,1,1,0,0,1)		No
99	(0,1,1,0,0,0,1,1)		No
101	(0,1,1,0,0,1,0,1)		No
113	(0,1,1,1,0,0,0,1)		No
142	(1,0,0,0,1,1,1,0)		No
154	(1,0,0,1,1,0,1,0)		No
156	(1,0,0,1,1,1,0,0)		No
166	(1,0,1,0,0,1,1,0)		No
170	(1,0,1,0,1,0,1,0)	(123)	Sì
172	(1,0,1,0,1,1,0,0)		No
178	(1,0,1,1,0,0,1,0)		No
180	(1,0,1,1,0,1,0,0)		No
184	(1,0,1,1,1,0,0,0)		No
198	(1,1,0,0,0,1,1,0)		No
202	(1,1,0,0,1,0,1,0)		No
204	(1,1,0,0,1,1,0,0)	e	Sì
210	(1,1,0,1,0,0,1,0)		No
212	(1,1,0,1,0,1,0,0)		No
216	(1,1,0,1,1,0,0,0)		No
226	(1,1,1,0,0,0,1,0)		No
228	(1,1,1,0,0,1,0,0)		No
240	(1,1,1,1,0,0,0,0)	$(123)^2$	Sì

□

L'esempio seguente prova invece il fatto che in qualche caso ci sono altri elementi di $\mathcal{G}^n(R)$, a parte quelli di $\mathcal{H}^n(R)$, che possono venire applicati ad automi di particolare lunghezza (diversa da n, beninteso) conservando l'invertibilità.

6.7 Esempio

Sono state applicate le rappresentazioni locali di elementi in $\mathcal{G}^3(\mathbb{Z}_2)$, calcolate su $(1,1)$-intorni, ad automi di lunghezza 5.

Rappr. decimale	Rappr. locale	Invertibile	Inversa (R. dec.)
15	(0,0,0,0,1,1,1,1)	Sì	252645135
27	(0,0,0,1,1,0,1,1)	No	
29	(0,0,0,1,1,1,0,1)	No	
39	(0,0,1,0,0,1,1,1)	No	
43	(0,0,1,0,1,0,1,1)	No	
45	(0,0,1,0,1,1,0,1)	Sì	1267157835
51	(0,0,1,1,0,0,1,1)	Sì	16711935
53	(0,0,1,1,0,1,0,1)	No	
57	(0,0,1,1,1,0,0,1)	No	
71	(0,1,0,0,0,1,1,1)	No	
75	(0,1,0,0,1,0,1,1)	Sì	757931565
77	(0,1,0,0,1,1,0,1)	No	
83	(0,1,0,1,0,0,1,1)	No	
85	(0,1,0,1,0,1,0,1)	Sì	65535
89	(0,1,0,1,1,0,0,1)	Sì	1095024315
99	(0,1,1,0,0,0,1,1)	No	
101	(0,1,1,0,0,1,0,1)	Sì	579001725
113	(0,1,1,1,0,0,0,1)	No	
142	(1,0,0,0,1,1,1,0)	No	
154	(1,0,0,1,1,0,1,0)	Sì	3199942980
156	(1,0,0,1,1,1,0,0)	No	
166	(1,0,1,0,0,1,1,0)	Sì	3715965570
170	(1,0,1,0,1,0,1,0)	Sì	4294901760
172	(1,0,1,0,1,1,0,0)	No	
178	(1,0,1,1,0,0,1,0)	No	
180	(1,0,1,1,0,1,0,0)	Sì	3027809460
184	(1,0,1,1,1,0,0,0)	No	
198	(1,1,0,0,0,1,1,0)	No	
202	(1,1,0,0,1,0,1,0)	No	
204	(1,1,0,0,1,1,0,0)	Sì	4278255360
210	(1,1,0,1,0,0,1,0)	Sì	3537035730
212	(1,1,0,1,0,1,0,0)	No	
216	(1,1,0,1,1,0,0,0)	No	
226	(1,1,1,0,0,0,1,0)	No	
228	(1,1,1,0,0,1,0,0)	No	
240	(1,1,1,1,0,0,0,0)	Sì	4042322160

Si può notare che oltre agli elementi di $\mathcal{H}^3(\mathbb{Z}_2)$, ovvero

$$\{204, 170, 240, 51, 85, 15\}$$

anche altri elementi, in questo caso, hanno le proprietà desiderate:

R. dec.	Rappresentazione locale dell'Inversa
45	(0,1,0,0,1,0,1,1,1,0,0,0,0,1,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1)
75	(0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,0,1,1,1,1,0,0,0,1,0,1,1,0,1)
89	(0,1,0,0,0,0,0,1,0,1,0,0,0,1,0,0,1,0,1,1,1,1,1,0,1,0,1,1,1,0,1,1)
101	(0,0,1,0,0,0,1,0,1,0,0,0,0,0,1,0,1,1,0,1,1,1,0,1,0,1,1,1,1,1,0,1)
154	(1,0,1,1,1,1,1,0,1,0,1,1,1,0,1,1,0,1,0,0,0,0,0,1,0,1,0,0,0,1,0,0)
166	(1,1,0,1,1,1,0,1,0,1,1,1,1,1,0,1,0,0,1,0,0,0,1,0,1,0,0,0,0,0,1,0)
180	(1,0,1,1,0,1,0,0,0,1,1,1,1,0,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0)
210	(1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,1,0,0,0,0,1,1,1,0,1,0,0,1,0)

Tuttavia è evidente che le rappresentazioni locali delle inverse hanno tutte raggio destro minimo $n - r_0 - 1 = 3$, e quindi le trasformazioni inverse non appartengono all'insieme

$$\{15, 45, 51, 75, 85, 89, 101, 154, 166, 170, 180, 204, 210, 240\}$$

Si noti che le trasformazioni $\{45, 75, 89, 101, 154, 166, 180, 210\}$ hanno la proprietà di restare invertibili in $\mathcal{B}^5_{1,1}(\mathbb{Z}_2)$ ma non la proprietà di avere inverse a loro volta in $\mathcal{B}^5_{1,1}(\mathbb{Z}_2)$. □

6.2 Questione aperta

È stato osservato mediante computazione diretta che per automi di lunghezza pari maggiore di 3 e minore o uguale a 18 le uniche trasformazioni di raggio destro e sinistro 1 che rimangono invertibili sono quelle del gruppo $\mathcal{H}^3(\mathbb{Z}_2)$. Con la stessa tecnica si è anche osservato che le trasformazioni invertibili $\{45, 75, 89, 101, 154, 166, 180, 240\}$ discusse in precedenza sono le uniche trasformazioni non banali che rimangono invertibili per tutti gli automi di lunghezza dispari maggiore o uguale a 3 e minore di 18. Il limite superiore 18 é semplicemente un limite computazionale, e non teorico: non è infatti stato trovato nessun controesempio alla congettura che tali trasformazioni siano le sole a raggio destro e sinistro 1 definite su $\mathbb{Z}_2$ che rimangono invertibili su tutti gli automi di lunghezza dispari e solo su quelli.

A sostegno di questa congettura, con altre tecniche di calcolo si è potuto verificare che per automi di lunghezze prese nell'insieme

$$T = \{3, 5, 7, 9, 11, 13, 15\}$$

la trasformazione 180 è una "radice di una trasformazione invertibile banale, nel senso che per ogni $n \in T$, $180^{(2^{\lfloor \frac{n+3}{4} \rfloor})}$ è una trasformazione banale con permutazione associata π uguale all'identità, o in altre parole un ciclo di lunghezza n. Questo implica evidentemente che

$$180^{(2^{\lfloor \frac{n+3}{4} \rfloor})n} = e$$

e che quindi 180 è invertibile. Si suppone che un discorso analogo valga anche per tutte le trasformazioni in $\{45, 75, 89, 101, 154, 166, 180, 210\}$, ma non è ancora stata trovata alcuna prova.

Esiste altresì una dimostrazione recentissima dovuta a Jarkko Kari (cfr. [Kar97]) che prova che la trasformazione 180 è reversibile solo su automi cellulari periodici di lunghezza dispari; la dimostrazione dipende largamente dalla struttura della regola 180, e quindi è difficilmente generalizzabile. Ha tuttavia l'indiscusso valore di fare la prima luce su una questione che a detta dello stesso Kari è "nuova, e rappresenta un interessante e inesplorato argomento di ricerca[16].

6.8 Proposizione (Kari)

La trasformazione cellulare con $(1, 1)$*-intorni definita su* $\mathbb{Z}_2$ *che estende la rappresentazione decimale 180 è reversibile su automi di lunghezza dispari con condizioni al contorno periodiche.*

Dimostrazione. Poiché la lunghezza degli automi considerati è finita ogni trasformazione cellulare è una funzione $R^n \to R^n$ per qualche n. Pertanto, essendo dominio e codominio finiti, se una tale funzione è iniettiva è anche

[16] Cfr. [Kar97].

suriettiva; di conseguenza è sufficiente provare che 180 è iniettiva. Si scriva innanzitutto la regola 180 nella forma seguente: sia ϕ la regola locale $R^3 \to R$ tale che per ogni $x, y \in R$,

$$\phi(x, 0, y) = x \tag{17}$$

$$\phi(x, 1, y) = x + y + 1 \mod 2 \tag{18}$$

e sia σ la corrispondente trasformazione cellulare $R^n \to R^n$. Dato che la rappresentazione locale di 180 è $(1, 0, 1, 1, 0, 1, 0, 0)$ è immediato verificare che ϕ è la regola 180. Supponiamo per assurdo che ϕ non sia iniettiva su automi di qualche lunghezza dispari n. Siano dunque

$$\begin{aligned} \underline{u} &= (u_1, \ldots, u_n) \\ \underline{v} &= (v_1, \ldots, v_n) \\ \underline{u}' &= (u'_1, \ldots, u'_n) = \sigma(\underline{u}) \\ \underline{v}' &= (v'_1, \ldots, v'_n) = \sigma(\underline{v}) \end{aligned}$$

tali che $\underline{u} \neq \underline{v}$ e

$$\underline{u}' = \underline{v}'$$

Supponiamo per assurdo che esista un i tale che $u_i = v_i$ e $u_{i+1} = v_{i+1}$. Se $u_i = 0$ allora per la definizione (17) si ha che $u_{i-1} = u'_i$. Se $u_i = 1$ per la (18) allora $u_{i-1} = u'_i + u_{i+1} + 1$; analogamente per $\underline{v}$ si ha $v_{i-1} = v'_i$ se $v_i = 0$ e $v_{i-1} = v'_i + v_{i+1} + 1$ se $v_i = 1$. Ma $u'_i = v'_i$ e $u_{i+1} = v_{i+1}$ e quindi in tutti i casi si ottiene $u_{i-1} = v_{i-1}$. Per induzione si ha $u_{i-2} = v_{i-2}$, eccetera, e quindi $u_i = v_i$ per ogni i, che implica $\underline{u} = \underline{v}$, una contraddizione. Questo significa che per ogni i se $u_i = v_i$ allora $u_{i+1} \neq v_{i+1}$. A questo punto si applica il fatto che n è dispari: non è possibile che le componenti di $\underline{u}$ e $\underline{v}$ siano uguali una posizione sì e una no, perché per la periodicità delle condizioni al contorno se

$$u_1 = v_1, u_2 \neq v_2, u_3 = v_3, \ldots$$

si ha poi, con n dispari, $u_n = v_n$ e quindi $u_1 \neq v_1$. La cosa è analoga se

$$u_1 \neq v_1, u_2 = v_2, u_3 \neq v_3, \ldots$$

Rimangono due casi possibili:

1. Esiste un i tale che $u_i = 0, u_{i+1} = 1$ e $v_i = 1, v_{i+1} = 0$. Allora per la (17) bisogna avere $v'_{i+1} = v_i = 1$ e quindi anche $u'_{i+1} = 1$; di conseguenza per la (18) $u_{i+2} = 0$. Allora per la (17) $u'_{i+2} = u_{i+1} = 1$ e quindi per la (18) $v_{i+2} = 1$. Il processo si può ripetere con alle posizioni $i + 1$ e $i+2$, e questo rende evidente il fatto che le componenti di $\underline{u}$ e $\underline{v}$ devono alternarsi fra 0 e 1, che non è possibile perché n è dispari.

2. Esiste un i tale che $u_i = u_{i+1} = 0$ e $v_i = v_{i+1} = 1$. Ne segue che per la (17) $u'_{i+1} = 0$, e quindi per la (18) $v_{i+2} = 0$. Allora per la (17) $v'_{i+2} = 1$ da cui per la (18) $u_{i+2} = 1$. Abbiamo quindi il caso precedente con i rimpiazzato da $i + 1$.

Bisogna dunque concludere che ϕ è iniettiva. □

La dimostrazione testè proposta prova anche che la regola 75 è iniettiva su lunghezze dispari. Questo risultato si ottiene applicando la seguente proposizione alla regola 180.

6.9 Proposizione

Sia $\pi \in S_{|R|}$, σ una trasformazione cellulare in $\mathcal{A}^n(R)$ e $\underline{\phi} = (\phi_1, \ldots, \phi_{|R|^n})$ la sua rappresentazione locale. Se σ è invertibile, la trasformazione cellulare τ data dalla regola locale

$$\pi(\underline{\phi}) = (\pi(\phi_1), \ldots, \pi(\phi_{|R|^n}))$$

è invertibile.

Dimostrazione. Per il teorema (5.2) se σ è invertibile allora Im $\mathcal{R}(\sigma) \cong_O$ Im $\mathcal{R}(e)$, dove e è l'identità di $\mathcal{A}^n(R)$. Per definizione di τ se $\underline{v} \in R^n$ e $\sigma(\underline{v}) = \underline{w}$ allora $\tau(\underline{v}) = \pi(\underline{w})$. Poiché $\pi \in S_{|R|}$ si ha che $\{\pi(\underline{v}) \mid \underline{v} \in R^n\} = R^n$, e quindi Im $\mathcal{R}(\tau) \cong_O$ Im $\mathcal{R}(\sigma)$, da cui Im $\mathcal{R}(\tau) \cong_O$ Im $\mathcal{R}(e)$ e quindi ancora per il teorema (5.2) τ è invertibile. □

Sia π l'unico elemento non identità di S_2. Poiché

$$\pi(180) = \pi(1,0,1,1,0,1,0,0) = (0,1,0,0,1,0,1,1) = 75$$

anche 75 ha le stesse proprietà di invertibilità di 180.

7 Automi di Lunghezza Infinita

Si può generalizzare il concetto di automa cellulare anche quando la lunghezza dello stesso è infinita. In pratica abbiamo una stringa infinita aperta da entrambe le parti che si trasforma nel tempo con una regola locale come nel caso finito. Ovviamente le condizioni al contorno non costituiscono un problema in quanto la stringa è, appunto, infinita. Chiameremo anche una stringa infinita con il termine *configurazione*.

Potrebbe sembrare che la ricerca sugli automi cellulari a lunghezza infinita sia interessante solo a livello teorico: dopotutto gli automi cellulari sono usati estensivamente nel calcolo computazionale, dove ogni numero, per forza di cose, dev'essere finito. Esistono invece molte applicazioni degli automi cellulari a lunghezza infinita, alcune delle quali in relazione agli automi invertibili. In breve, se si riesce a provare che una trasformazione cellulare su (r_0, r_1)-intorni è invertibile su automi di lunghezza infinita, per il teorema (7.2) quella trasformazione è invertibile su automi di qualsiasi lunghezza finita maggiore di $r_0 + r_1$. Poiché nelle simulazioni al computer di fenomeni fisici le proprietà della trasformazione cellulare devono essere indipendenti dalle dimensioni dell'automa è necessario usare trasformazioni che siano invertibili su ogni lunghezza. Si può vedere quindi che lo studio degli automi cellulari di lunghezza infinita è fondamentale per l'uso degli automi in simulazioni di realtà fisiche.

Formalmente una **configurazione** è una funzione

$$c : \mathbb{Z} \to R$$

dove R è un anello. Useremo talvolta la notazione c_i per indicare $c(i)$ e la notazione $\underline{c}$ per indicare la successione bi-infinita

$$(\ldots, c_{-i}, \ldots, c_{-1}, c_0, c_1, \ldots, c_j, \ldots)$$

L'insieme di tutte le configurazioni verrà indicato con $R^{\mathbb{Z}}$. Per ogni intero n, ad ogni vettore $\underline{v}$ in R^n (che chiameremo anche **configurazione finita**) corrisponde una configurazione periodica infinita

$$\underline{c} = (\ldots, v_1, \ldots, v_n, v_1, \ldots, v_n, \ldots)$$

Più precisamente c è tale che, se $\lambda(\underline{v}, i)$ è la i-esima componente di $\underline{v}$,

$$\left.\begin{array}{rrrrl} & \forall\ i \text{ tale che } 1 \leq i \leq n & c_i & = & \lambda(\underline{v}, i) \\ \forall\ k \neq 0 & \forall\ i \text{ tale che } 1 \leq i \leq n & c_{kn+i} & = & c_i \end{array}\right\} \tag{19}$$

Evidentemente ad ogni configurazione periodica infinita c con periodo n corrisponde un vettore $\underline{v}$ di lunghezza n dato da

$$\forall\ i \leq n\ \lambda(\underline{v}, i) = c_i$$

7.1 Lemma

Sia $\underline{v} \in R^n$, c la configurazione periodica ottenuta da $\underline{v}$ come in eq. (19); sia ϕ una regola locale e siano σ_n e σ_∞ le trasformazioni cellulari che estendono ϕ su (r_0, r_1)-intorni rispettivamente ad automi di lunghezza n e di lunghezza infinita. Allora $\sigma_\infty(c)$ è la configurazione periodica infinita ottenuta da $\sigma_n(\underline{v})$ come in eq. (19).

Dimostrazione. Poiché c è infinita periodica con periodo n, abbiamo che per ogni $i \leq n$ gli intorni di raggi sinistro e destro risp. r_0 e r_1 centrati alla posizione $kn+i$ sono uguali per ogni intero k, dunque anche le loro immagini sotto ϕ sono uguali; si ottiene quindi

$$\forall\ k \neq 0\ \forall\ i \text{ tale che } 1 \leq i \leq n\ (\sigma_\infty(c))_{kn+i} = (\sigma_\infty(c))_i$$

Ora, se $i > r_0$ e $i < n - r_1$ è evidente che

$$(\sigma_\infty(c))_i = \lambda(\sigma_n(\underline{v}), i)$$

Supponiamo che $i \leq r_0$. L'intorno centrato alla posizione i nell'automa finito $\underline{v}$ è dato da

$$(v_{n+i-r_0}, \ldots, v_i, \ldots, v_{i+r_1})$$

dove $v_j = \lambda(\underline{v}, j)$. L'intorno centrato alla posizione i nell'automa infinito c è dato da

$$(c_{i-r_0}, \ldots, c_i, \ldots, c_{i+r_1})$$

Poiché per costruzione per ogni $j < 1$ si ha che $c_j = v_{n-j}$, i due intorni sono uguali e quindi anche le loro immagini sotto ϕ sono uguali. Un discorso analogo vale per $i \geq r_1$, provando quindi che per ogni i tale che $1 \leq i \leq n$ si ha $(\sigma_\infty(c))_i = \lambda(\sigma_n(\underline{v}), i)$. □

Dimostriamo ora il teorema che permette di trovare applicazioni pratiche agli automi cellulari infiniti. Se una trasformazione è invertibile su automi di lunghezza infinita, lo è anche su quelli di lunghezza finita.

7.2 Teorema

Sia ϕ una regola locale la cui corrispondente estensione su (r_0, r_1)-intorni ad automi di lunghezza infinita $\sigma_\infty : R^{\mathbb{Z}} \to R^{\mathbb{Z}}$ è invertibile. Allora per ogni $n > r_0 + r_1$ le trasformazioni cellulari su automi di lunghezza n σ_n derivate da ϕ sono invertibili.

Dimostrazione. Supponiamo per assurdo che σ_∞ sia invertibile e che esista un intero n tale che σ_n non è invertibile. Poichè σ_n è una funzione da un insieme finito in se stesso, σ_n non è invertibile equivale a dire che σ_n non è iniettiva, dunque esistono due vettori diversi $\underline{u}$ e $\underline{v}$ in R^n tali che

$$\sigma_n(\underline{u}) = \sigma_n(\underline{v}) = \underline{x}$$

Ora, siano c, d, f le configurazione periodiche ottenute rispettivamente da $\underline{u}, \underline{v}, \underline{x}$ come descritto in eq. (19). Si ottiene per costruzione che $c \neq d$; per il lemma testè provato si ha che

$$\sigma_\infty(c) = f = \sigma_\infty(d)$$

contraddicendo perciò al fatto che σ_∞ sia iniettiva e quindi invertibile. □

Per i motivi sopra descritti, la ricerca sugli automi cellulari è spesso sinonimo di ricerca sugli automi cellulari infiniti; esiste pertanto una letteratura sufficiente ad esplicare con chiarezza molti dei loro tratti fondamentali. In questa sede esporremo idee tratte dagli articoli [AP72, Ric72, Hed69].

7.1 Giardini dell'Eden

Uno dei primi tentativi di investigare l'invertibilità degli automi cellulari fu quello effettuato nei primi anni sessanta da Edward F. Moore e John Myhill, basato sull'idea di **configurazione "Giardino dell'Eden** (o, più brevemente, configurazione GOE[17]). Una configurazione c è GOE rispetto ad una trasformazione cellulare σ se non esiste alcuna configurazione d tale che $\sigma(d) = c$. È evidente che l'esistenza di configurazioni GOE rispetto a σ implica immediatamente che σ non sia suriettiva, e quindi nemmeno invertibile. La limitazione di questo criterio è che non è possibile costruire un algoritmo di verifica di invertibilità: se si riesce a trovare una configurazione GOE rispetto a σ si ha che σ non è invertibile, ma non si può essere sicuri di esaurire la ricerca della configurazione GOE in un numero finito di passi (requisito necessario per definizione di algoritmo). Il lavoro di Moore e Myhill, comunque, diede i suoi frutti: essi dimostrarono un teorema che servì a D. Richardson per provare nel 1971 che una trasformazione cellulare è invertibile se e solo se è iniettiva.

Per la costruzione di un algoritmo di verifica dell'invertibilità di automi cellulari unidimensionali a lunghezza infinita si dovette attendere fino al 1972, quando S. Amoroso e Y. N. Patt, del comando di elettronica di Fort

[17] L'acronimo deriva da "Garden of Eden.

Monmouth dell'esercito statunitense pubblicarono un articolo ([AP72]) ove proposero un algoritmo di verifica che prendeva spunto dal teorema di Richardson, nel senso che l'algoritmo verifica l'iniettività della trasformazione cellulare.

Per comprendere i teoremi di Moore e Myhill e di Richardson è necessario introdurre qualche nozione preliminare. L'**insieme delle configurazioni finite**, che negli articoli viene indicato con C_F, è l'insieme $R^{<\omega}$, ovvero l'insieme di tutte le funzioni da un ordinale finito in R. Più semplicemente si ha che

$$C_F = \bigcup_{n \in \mathbb{N}} R^n$$

Data una regola locale ϕ con raggio sinistro r_0 e raggio destro r_1 possiamo definire una funzione τ del sottoinsieme di D di C_F

$$D = \bigcup_{n > r_0 + r_1} R^n$$

in modo che $\tau : D \to C_F$ sia data da

$$\tau((v_1, \ldots, v_{r_0}, v_{r_0+1}, \ldots, v_{r_0+n}, \ldots, v_{r_0+n+r_1})) = (v_{r_0+1}, \ldots, v_{r_0+n})$$

Chiameremo τ la **restrizione di ϕ a C_F**.

Una **configurazione parziale** è una funzione da un sottoinsieme finito di $\mathbb{Z}$ ad R; si dice che una configurazione parziale b è in accordo con una configurazione (parziale, finita o infinita) c se

$$c|_{\mathrm{dom}(b)} = b$$

7.3 Teorema (Moore e Myhill)

Sia ϕ una trasformazione cellulare e τ la sua restrizione a C_F. Allora τ è iniettiva se e solo se per ogni configurazione parziale b esiste una configurazione finita $\underline{v}$ nell'immagine di τ tale che b è in accordo con $\underline{v}$.

Sia $(c^{(i)}) = c^{(1)}, c^{(2)}, \ldots$ una successione infinita di configurazioni. Diciamo che c è un **punto di accumulazione** di $(c^{(i)})$ se ogni configurazione parziale che è in accordo con c è anche in accordo con $c^{(i)}$ per infiniti valori di i.

7.4 Lemma (Richardson)

Ogni successione infinita di configurazioni ha un punto di accumulazione.

Dimostrazione. Sia $(c^{(i)})$ una successione infinita di configurazioni, e sia (m_j) un buon ordine su $\mathbb{Z}$. Esiste una successione $(b^{(i)})$ di configurazioni parziali tali che il dominio di $b^{(i)}$ è

$$M_i = \{m_j \mid j \leq i\}$$

e tali che $b^(i)$ è in accordo con $c^{(k)}$ per infiniti valori di k. Se per assurdo non esistesse dovrebbe esserci un valore di i per cui esiste una sottosuccessione infinita $(c^{(k_j)})$ tale che tutte le configurazioni parziali $(c^{(k_j)})|_{M_i}$ sono distinte, il che non è possibile perché M_i è un insieme finito. Sia c la configurazione definita da

$$c_{m_i} = b^{(i)}_{m_i}$$

per ogni $i \in \mathbb{Z}$. È facile verificare che c è un punto di accumulazione per $(c^{(i)})$. □

7.5 Teorema (Richardson)

Sia ϕ una regola locale; sia σ la corrispondente trasformazione cellulare su automi di lunghezza infinita e τ la restrizione a C_F. Se τ è iniettiva allora ϕ, applicata ad automi di lunghezza infinita, è suriettiva.

Dimostrazione. Sia y una configurazione infinita e sia $(b^{(i)})$ una successione di configurazioni parziali tali che y è l'unica configurazione che è in accordo con ogni elemento di $(b^{(i)})$. Per il teorema di Moore e Myhill esistono successioni di configurazioni $(x^{(i)})$ e $(y^{(i)})$ tali che per ogni i si ha che

$$\sigma(x^{(i)}) = y^{(i)}$$

e $y^{(i)}$ è in accordo con $b^{(i)}$. Per il lemma di Richardson $(x^{(i)})$ ha un punto di accumulazione x, che per la costruzione usata nel lemma è tale che

$$\sigma(x) = y$$

Poiché y è arbitrario, σ è suriettiva. □

7.2 Algoritmo di verifica dell'invertibilità su automi cellulari ad una dimensione

Nel loro articolo [AP72], S. Amoroso e Y. N. Patt presentarono un algoritmo che verifica se l'estensione di una data regola locale è iniettiva o no. Se la trasformazione verificata risulta iniettiva, per il teorema di Richardson (7.5) essa è anche suriettiva, e quindi invertibile.

Diamo innanzitutto qualche definizione. Sia ϕ una regola locale con raggio sinistro r_0 e raggio destro r_1 data da

$$\underline{\phi} = (\phi_1, \phi_2, \ldots, \phi_{|R|^n}) \in R^{|R|^n}$$

Si dice che la regola locale ϕ è **bilanciata** se per ogni $r \in R$ esistono indici $i_1, \ldots, i_{|R|^{n-1}}$ (dipendenti da r) tali che

$$\phi_{i_1} = \cdots = \phi_{i_{|R|^{n-1}}}$$

ovvero se $\underline{\phi}$ contiene in egual quantità tutti gli elementi di R.

7.6 Lemma

Sia σ la trasformazione cellulare che estende ϕ con (r_0, r_1)-intorni applicata a vettori di lunghezza $n = r_0 + r_1 + 1$. Se σ è invertibile, ϕ è bilanciata.

Dimostrazione. Sia $(\underline{v}_i)$ il buon ordine lessicografico su R^n. Poiché σ è invertibile, si ha che

$$\sigma(R^n) = R^n$$

e quindi gli elementi della rappresentazione locale sono

$$D = \{\lambda(\underline{v}_i, r_0 + 1) \mid i \leq |R|^n\}$$

Essendo R un anello finito, esiste su di esso un buon ordine, e quindi si può scrivere

$$R = \{a_1, \ldots, a_{|R|}\}$$

Ora, la lista $(\underline{v}_i)$, scritta per esteso, appare come

$$\begin{gathered}(a_1, \ldots, a_1, a_1)\\(a_1, \ldots, a_1, a_2)\\\vdots\\(a_1, \ldots, a_1, a_{|R|})\\(a_1, \ldots, a_2, a_1)\\\vdots\\(a_{|R|}, \ldots, a_{|R|}, a_{|R|})\end{gathered}$$

dove si può vedere che le colonne hanno tutte gli stessi elementi, e in particolare la prima colonna ha gli stessi elementi (in ordine diverso) della $(r_0 + 1)$-esima; quindi

$$D = \{\lambda(\underline{v}_i, 1) \mid i \leq |R|^n\}$$

ed è ora immediato constatare che gli elementi di D, presi in ordine di i crescente, sono

$$\underbrace{a_0, \ldots, a_0}_{|R|^{n-1}}, \ldots, \underbrace{a_{|R|}, \ldots, a_{|R|}}_{|R|^{n-1}}$$

e che quindi la rappresentazione locale di ϕ è bilanciata. □

Alla base dell'algoritmo sta la seguente proposizione.

7.7 Proposizione (Amoroso e Patt)

Sia ϕ una regola locale con raggio sinistro r_0 e raggio destro r_1. Sia $n = r_0 + r_1 + 1$, $k = |R|$, e sia σ la trasformazione cellulare su automi di lunghezza infinita che estende ϕ. Allora le seguenti condizioni sono equivalenti:

(i) σ *è iniettiva*

(ii) Per ogni coppia di vettori $\underline{u} \neq \underline{v}$ *in* R^n *tali che* $\phi(\underline{u}) = \phi(\underline{v})$ *e per ogni coppia di configurazioni* c *e* d *che contengono rispettivamente* $\underline{u}$ *e* $\underline{v}$ *alle posizioni* $i - n + 1, \ldots, i$ *(per qualche intero* i*) tali che per ogni* $j \geq i$

$$(\sigma(c))_j = (\sigma(d))_j$$

si ha che per ogni intero $p \geq i + k\binom{k^{n-1}}{2}$

$$c_p = d_p$$

Dimostrazione. (i) $\Rightarrow$ (ii): Poiché σ è iniettiva, per il lemma precedente la rappresentazione locale ϕ di σ è bilanciata; dunque ci sono $M = k\binom{k^{n-1}}{2}$ possibili insiemi contenenti due vettori diversi $\underline{u}, \underline{v} \in R^n$ tali che $\phi(\underline{u}) = \phi(\underline{v})$. Supponiamo per assurdo che le n-tuple fra le posizioni $i - n + 1$ e $i + M$ siano sempre diverse in c e in d. Ora, devono esistere posizioni $j_1 < j_2$ comprese fra $i - n + 1$ e $i + M$ dove le n posizioni a partire da $i + j_1$ contengono due n-tuple distinte $\underline{u}'$ e $\underline{v}'$ rispettivamente in c e in d e le n posizioni a partire da $i + j_2$ contengono le stesse n-tuple $\underline{u}'$ e $\underline{v}'$. In un caso, le n posizioni a partire da $i + j_2$ contengono $\underline{u}'$ in c e $\underline{v}'$ in d: ne segue che esistono due configurazioni infinite periodiche c' e d' della forma

$$\begin{aligned} c' &= (\ldots, \underline{u}', \ldots, \underline{u}', \ldots) \\ d' &= (\ldots, \underline{v}', \ldots, \underline{v}', \ldots) \end{aligned}$$

tali che $\sigma(c') = \sigma(d')$, contraddicendo all'iniettività di σ. Nel secondo caso le n posizioni a partire da $i + j_2$ contengono $\underline{u}'$ in d e $\underline{v}'$ in c, e allora c' e d'

diventano rispettivamente

$$\begin{aligned} c' &= (\ldots, \underline{u}', \ldots, \underline{v}', \ldots) \\ d' &= (\ldots, \underline{v}', \ldots, \underline{u}', \ldots) \end{aligned}$$

dove $\sigma(c') = \sigma(d')$, contraddicendo all'iniettività di σ. Dunque deve esistere un intero q nell'intervallo $-n+1 \leq q \leq M$ tale che alle posizioni $i+q, \ldots, i+q+n-1$ di c e d si trovi la stessa n-tupla. Supponiamo ora che esista un intero $p \geq q$ tale che $c_p \neq d_p$. Si può allora ripetere l'argomento appena esposto con i sostituito da p: si costruirebbero allora due configurazioni che differiscono solo in qualche posizione fra p e $p+M$ con la stessa immagine sotto σ. Questo prova la prima parte del teorema.

(ii) $\Rightarrow$ (i): Se per assurdo σ non fosse iniettiva esisterebbero configurazioni $c \neq d$ con la stessa immagine sotto σ. Da queste è facile costruire due configurazioni c', d' tali che per qualche i le posizioni $i-n+1, \ldots, i$ contengono n-tuple distinte $\underline{u}$, $\underline{v}$ con $\phi(\underline{u}) = \phi(\underline{v})$ e $c_q \neq d_q$ per qualche $q > i+M$, contraddicendo l'ipotesi. Si possono costruire c' e d' nel seguente modo: sia q una posizione tale che $c_q \neq d_q$, sia $\underline{u}$ la n-tupla centrata in posizione q in c e sia $\underline{v}$ la n-tupla corrispondente in d. Sia i la posizione data da $i = q - M - r_1 - 1$ e sia c' data da c con la n-tupla centrata in i rimpiazzata da $\underline{u}$ e similmente sia d' data da d con la n-tupla centrata in i rimpiazzata da $\underline{v}$. Le configurazioni c' e d' così costruite hanno le proprietà richieste. □

Proponiamo ora l'algoritmo come descritto in [AP72], analizzando un esempio concorrentemente alla presentazione dello stesso. Per facilitare i calcoli useremo come esempio l'identità di $\mathcal{B}^{\infty}_{1,1}(\mathbb{Z}_2)$, ovvero la trasformazione con numero di Wolfram[18] 204 su $(1,1)$-intorni applicata ad automi di lunghezza infinita.

[18] Definito a pagina 21, cfr. anche nota 6 a piè di pagina.

Sia ϕ la rappresentazione locale della trasformazione cellulare di cui si vuole investigare l'iniettività. Sia n il raggio d'azione di ϕ e sia R l'anello di definizione di ϕ. Anzitutto si deve verificare che ϕ sia bilanciata. Nel caso di 204, si ha

111	110	101	100	011	010	001	000
7	6	5	4	3	2	1	0
↓	↓	↓	↓	↓	↓	↓	↓
1	1	0	0	1	1	0	0

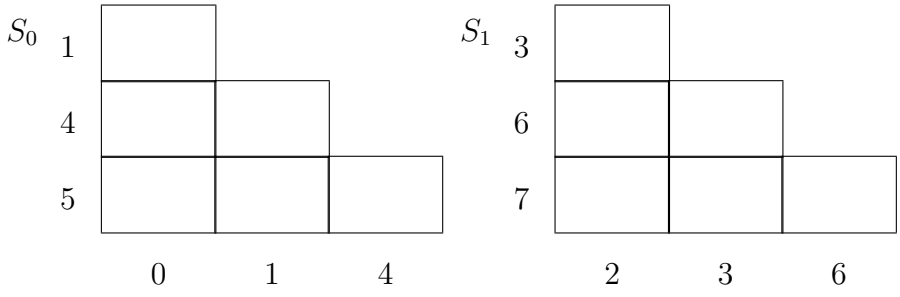
e quindi 204 è bilanciata. Procediamo ora con il test vero e proprio.

Passo 1. Si considerino gli insiemi ordinati $S_0, \ldots, S_{R-1}$ tali che

$$S_i = (k < |R|^n \mid k \geq 0 \wedge \phi(k) = i)$$

ordinati secondo l'ordine naturale degli interi, dove con $\phi(k)$ si intende ϕ applicato al vettore che è lo sviluppo in base R dell'intero k. Nel caso del nostro esempio si ha $S_0 = (0, 1, 4, 5)$ e $S_1 = (2, 3, 6, 7)$. Si noti che in generale la cardinalità di S_i è $t = R^{n-1}$.

Passo 2. Per ogni insieme S_i si costruisca la sua **tabella dei seguenti** in questo modo: la tabella ha $t-1$ righe, e la riga j-esima contiene j caselle. Si indicizzino le righe con gli ultimi $t-1$ elementi di S_i a partire dal secondo, e le colonne con i primi $t-1$ elementi di S_i. Nel nostro esempio si ha



Si noti che ad ogni casella corrisponde un unico sottoinsieme di due elementi di S_i e viceversa. Sia $\{a, b\}$ un sottoinsieme di due elementi di S_i e

siano $(a_1, \ldots, a_n)$ e $(b_1, \ldots, b_n)$ gli sviluppi in base R rispettivamente di a e di b. Un altro sottoinsieme (c, d) di due elementi di un qualche S_j è chiamato un **seguente** di (a, b) se le ultime $n-1$ componenti di a sono le prime componenti di c e analogamente per b e d, ovvero se

$$a_2 = c_1, \ldots, a_l = c_{l-1}, \ldots, a_n = c_{n-1}$$

e

$$b_2 = d_1, \ldots, b_l = d_{l-1}, \ldots, b_n = d_{n-1}$$

Si noti che poiché c e d appartengono ad un S_j per qualche j si ha $\phi(c) = \phi(d)$, e si noti anche che c e d devono essere distinti.

Passo 3. Nella casella indicizzata da $\{i, j\}$ si scrivano tutti i seguenti di $\{i, j\}$. Se per qualche casella, diciamo indicizzata da $\{a, b\}$, le ultime $n-1$ componenti di a e b sono le stesse, ovvero se

$$a_2 = b_2 \ldots, a_n = b_n$$

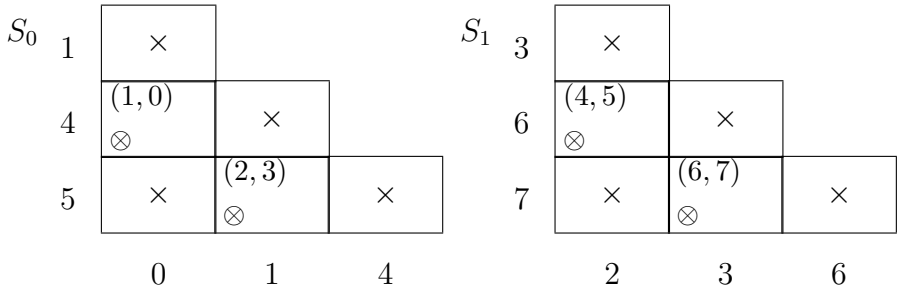
si ponga un simbolo $\otimes$ nella casella. Nelle caselle rimaste vuote si ponga un simbolo $\times$. Se esiste una casella $\{a, b\}$ che annovera $\{a, b\}$ fra i suoi seguenti l'algoritmo finisce e ϕ non è iniettiva: si avrebbe infatti che le configurazioni distinte periodiche

$$(\ldots, a_1, \quad \ldots \quad, a_n, \ldots)$$
$$(\ldots, b_1, \quad \ldots \quad, b_n, \ldots)$$

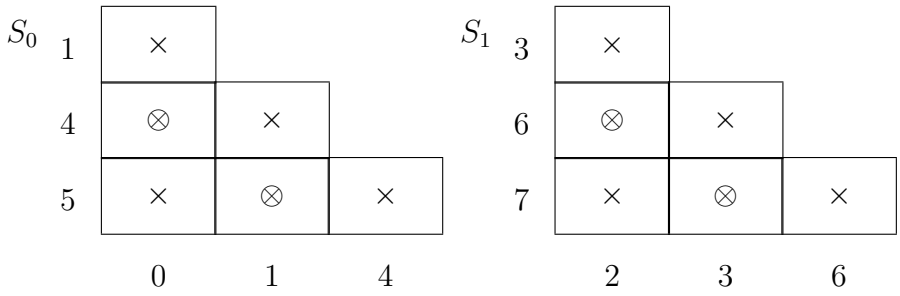
sono mandate da ϕ nella stessa configurazione

$$(\ldots, \phi(a) = \phi(b), \ldots, \phi(a) = \phi(b), \ldots)$$

A questo punto le tabelle dei seguenti di 204 sono



Passo 4. Si riducano le tabelle nel seguente modo: se $\{c, d\}$ si trova nella casella $\{a, b\}$ e la casella corrispondente a $\{c, d\}$ ha il simbolo $\times$, si cancelli $\{c, d\}$ dalla casella $\{a, b\}$. Se una casella è vuota (e non ha nemmeno il simbolo $\otimes$), si ponga in essa il simbolo $\times$. Si proceda in questo modo fino a che è possibile. Si noti che una casella non può avere entrambi i simboli $\otimes$ e $\times$. Nel caso di 204, dopo la riduzione le tabelle appaiono come segue:



Passo 5. Si assegnano interi positivi a ogni casella in cui non compare il simbolo $\times$ nel modo seguente:

1. il simbolo $\otimes$ ha peso 0.
2. se la casella $\{a, b\}$ contiene seguenti e simboli con pesi $w_1, \ldots, w_h$ allora $\{a, b\}$ ha peso $1 + \max(w_1, \ldots, w_h)$

Se esistono caselle a cui non è stato assegnato alcun intero ϕ non è iniettiva e l'algoritmo ha termine: infatti se non è stato assegnato alcun intero ad una

casella $\{a, b\}$ vuol dire che esiste una "catena di seguenti di $\{a, b\}$ che termina in un nodo della catena, e quindi entra in un ciclo periodico di seguenti. Si potrebbero allora costruire due configurazioni periodiche distinte l'immagine delle quali sotto ϕ è unica. Nel caso di 204 si ha che le caselle corrispondenti a $\{4, 0\}, \{5, 1\}, \{6, 2\}, \{7, 3\}$ hanno tutte peso 1.

Passo 6. Si considerino tutte le caselle a cui sono stati assegnati degli interi nel passo 5. Se fra esse non esiste alcuna casella $\{a, b\}$ tale che le prime $n - 1$ componenti di a e b sono uguali, ovvero tale che

$$a_1 = b_1, \ldots, a_{n-1} = b_{n-1}$$

allora ϕ è iniettiva; altrimenti non lo è. Nel nostro esempio abbiamo

$$\begin{aligned} 4 = 100 \quad &, \quad 0 = 000 \\ 5 = 101 \quad &, \quad 1 = 001 \\ 6 = 110 \quad &, \quad 2 = 010 \\ 7 = 111 \quad &, \quad 3 = 011 \end{aligned}$$

e quindi 204 è iniettiva.

Per la prima parte della proposizione di Amoroso e Patt si ha che se ϕ è iniettiva non è possibile estendere alcuna coppia di n-tuple distinte verso destra, mantenendo le immagini uguali, senza che prima di un certo limite finito le due configurazioni si fondano nella stessa. Questa fusione è assicurata se nel passo 5 ad ogni casella viene assegnato un intero (che è la lunghezza dell'estensione prima della fusione). Se dopo la fusione le configurazioni mantengono gli stessi simboli nelle stesse posizioni, per la seconda parte della proposizione ϕ è iniettiva; questa verifica avviene nel passo 6, dove si controlla che dopo la fusione le due configurazioni non possano divergere.

Questo algoritmo è stato implementato nel programma `CAInvInf.cpp` contenuto nell'appendice B. Per la codifica sono state usate tecniche di e-

splorazione di diagrammi ad albero; in pratica un "cursore si sposta fra le tabelle seguendo le catene di seguenti: se ogni catena ha una fine in una casella con il simbolo $\otimes$ e se la verifica del passo 6 risulta positiva ϕ è iniettiva, se esistono catene periodiche ϕ non è iniettiva e il programma termina.

7.3 Il gruppo $\mathcal{G}^\infty(R)$

Nell'analisi degli automi di lunghezza finita abbiamo indicato con $\mathcal{G}^n(R)$ il gruppo di tutte le trasformazioni cellulari invertibili su automi di lunghezza n (cfr. cap. 5). Analogamente usiamo la notazione $\mathcal{G}^\infty(R)$ per indicare il gruppo di tutte le trasformazioni cellulari invertibili su automi di lunghezza infinita. Lo studio di questo gruppo è stato intrapreso da vari ricercatori, e se ne può trovare una sintesi nell'articolo [Hed69]. Riportiamo in questa sede due proprietà caratteristiche.

1. Ogni gruppo finito è isomorfo a qualche sottogruppo di $\mathcal{G}^\infty(R)$.

2. Si consideri la trasformazione che muove ogni configurazione di una cella verso sinistra (l'analogo del ciclo $(12\ldots)$ nel caso infinito). Come nel caso finito, questa trasformazione viene comunemente chiamata *shift*[19]. Sia $\mathcal{C}^\infty(R)$ il sottogruppo di $\mathcal{G}^\infty(R)$ generato dallo *shift*. Poiché vale anche nel caso infinito un teorema analogo al teorema 2.3 si ha che tutte le trasformazioni di $\mathcal{G}^\infty(R)$ devono necessariamente commutare con lo *shift* e le sue potenze, da cui si ottiene che $\mathcal{C}^\infty(R)$ è un sottogruppo normale di $\mathcal{G}^\infty(R)$. Hedlund ha provato che il gruppo $\mathcal{G}^\infty(R)/\mathcal{C}^\infty(R)$ contiene due elementi distinti di ordine due (idempotenti) tali che il loro prodotto ha ordine infinito. Come corollario si ha che anche $\mathcal{G}^\infty(R)$ ha questa proprietà.

[19] Cfr. [Kar96, Hed69].

8 Automi Cellulari d-Dimensionali

Finora abbiamo discusso di automi cellulari unidimensionali: abbiamo definito un automa cellulare come una stringa di celle che si evolve nel tempo. Applicazioni comuni per gli automi unidimensionali sono nel campo della crittografia, dove solitamente si deve trasformare una frase in chiaro in una stringa in codice. Per le applicazioni della fisica sono molto più utili gli automi cellulari a due, tre o più dimensioni: per esempio, nel caso della simulazione dei gas è noto che le equazioni di Navier-Stokes hanno soluzioni che descrivono turbolenze solo in due o più dimensioni[20].

8.1 L'intorno

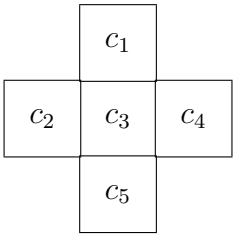
Come per gli automi unidimensionali parliamo di "stringa, per quelli bidimensionali parleremo di "griglia e per quelli d-dimensionali parleremo di "reticolo d-dimensionale. Il tipo di evoluzione dell'automa è sempre basato sul concetto di regola locale che da un intorno di una cella rende il valore della cella al tempo successivo. Mentre in una dimensione l'intorno dipende essenzialmente da due parametri, il raggio sinistro e destro, che determinano anche la lunghezza dell'intorno, nel caso di d dimensioni i parametri da cui dipende l'intorno sono plurimi e non facilmente catalogabili. Inoltre l'intorno non dipende solo dalle sue dimensioni ma anche dalla sua forma: vengono spesso usati automi cellulari bidimensionali con intorno esagonale (ad "alveare). Per varie applicazioni fisiche si usano reticoli d-dimensionali euclidei (in cui le rette delimitanti le celle siano perpendicolari e parallele) con intorno ipercubico V di lato l a d dimensioni. Formalmente V è una funzione $R^{l^d} \to R$, dove R è l'anello di definizione dell'automa. La cella di coordinate euclidee $(i_1, \ldots, i_d)$ verrà indicata con $V(i_1, \ldots, i_d)$. Per deter-

[20] Cfr. [WP85].

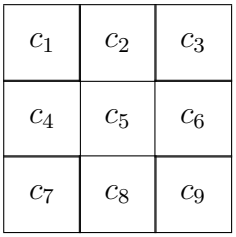
minare completamente il comportamento della legge di evoluzione ϕ è anche necessario specificare la posizione della cella $\phi(V)$. Questa verrà indicata con $(x_1^{(V)}, \ldots, x_d^{(V)})$.

Per ciò che concerne gli automi bidimensionali a reticolo quadrato si usano principalmente due tipi di intorni: l'intorno di von Neumann (a cinque celle) e l'intorno di Moore (a nove celle). È evidente dalla figura sotto che gli automi con intorno di von Neumann costituiscono un sottoinsieme di quelli con intorno di Moore. Quest'ultimo è a sua volta un esempio di intorno ipercubico a 2 dimensioni di lato 3.

von Neumann



Moore



Gli automi bidimensionali definiti su $R = \mathbb{Z}_2$ (le celle possono quindi assumere i valori 0 e 1) con intorno di von Neumann sono in tutto $2^{2^5} \simeq 4 \times 10^9$, e quelli con intorno di Moore sono $2^{2^9} \simeq 10^{154}$. In altre parole, appena abbandoniamo il caso unidimensionale la complessità computazionale di un approccio combinatoriale esaustivo del problema diventa inaffrontabile. Per semplificare le cose è stata introdotta la nozione di "dipendenza dalla somma: sia ϕ una regola locale definita su un intorno di Moore; tipicamente si avrà $\phi(c_1^t, \ldots, c_9^t) = c_5^{t+1}$, dove t è il tempo. La semplificazione consiste nel considerare solo quelle regole che possono essere scritte come

$$\psi\left(c_5^t, \sum_{i=1, i\neq 5}^{9} c_i^t\right) = c_5^{t+1}$$

ovvero che dipendono dalla cella in evoluzione e dalla somma degli stati delle altre celle dell'intorno. In questo modo bisogna prendere in considerazione

soltanto $2^{16} = 65535$ regole di evoluzione. Queste regole vengono chiamate **totalistiche esteriori**[21], e possono essere descritte da un numero $\tilde{C}$ dato da

$$\tilde{C} = \sum_{n,c} \psi(c,n)|R|^{|R|n+c}$$

8.2 Game of Life

Un esempio famosissimo di regola totalistica esteriore è dato dal *Game of Life* introdotto da Conway nel 1970, in cui una cella può essere viva (stato = 1) o morta (stato = 0). L'evoluzione avviene secondo le seguenti regole:

- Una cella morta è una cella vuota.
- Una cella vuota diventa viva se ha esattamente tre celle vive nell'intorno, altrimenti rimane vuota.
- Una cella viva rimane viva se nell'intorno sono presenti 2 o 3 celle vive.
- Una cella viva muore in tutti gli altri casi.

In pratica abbiamo $\psi(1,2) = 1$, $\psi(c,3) = 1$ per $c = 0,1$ e $\psi(c,n) = 0$ per tutti gli altri casi. La regola del *Game of Life* corrisponde pertanto a $\tilde{C} = 2^7 + 2^6 + 2^5 = 224$.

La proprietà più interessante del *Game of Life* è data dal fatto che è un computatore universale (è equivalente alla macchina di Turing). Come tale può simulare un qualsiasi computer: ovvero esistono delle configurazioni che "mimano il comportamento delle porte logiche AND, OR, NOT e di una memoria. Queste configurazioni sono basate sul concetto di aliante: un aliante è una particolare configurazione parziale che ha la proprietà di potersi spostare sulla griglia in una direzione precisa senza modificare sostanzialmente la sua

[21] Tradotto da *outer totalistic*, cfr. [WP85].

forma, con il passare del tempo. Esistono configurazioni parziali chiamate "cannoni di alianti che producono periodicamente degli alianti, ed è stato osservato che se due alianti si scontrano in modo particolare si annichilano. Posizionando in vari modi i cannoni di alianti si possono costruire le porte logiche.

8.3 Mappe transdimensionali

Avendo derivato una sostanziale quantità di risultati concernenti gli automi cellulari unidimensionali, sarebbe opportuno trovare dei modi di applicarli anche ad automi a più dimensioni. Cercheremo di costruire delle mappe particolari che permettano di simulare un automa d-dimensionale per mezzo di uno (o più) automi ad una sola dimensione. Perché la simulazione abbia senso, le mappe transdimensionali devono essere ovviamente biiettive. Solitamente le simulazioni con valore più generale consistono nell'immagazzinare le informazioni contenute nell'intorno dell'automa a d dimensioni per mezzo un anello R di definizione dell'automa unidimensionale che sia di cardinalità maggiore rispetto all'anello S su cui è definito l'automa d-dimensionale. Per fare un esempio, è possibile simulare un intorno di Moore definito su $S = \mathbb{Z}_2$ (ci sono $2^9 = 512$ intorni possibili) con un intorno a raggio destro e sinistro 1 definito su un anello $R = \mathbb{Z}_8$ ($8^3 = 512$ intorni possibili).

8.3.1 Mappa dell'intorno ipercubico

Supponendo di avere un automa cellulare d-dimensionale su un reticolo euclideo, definito su un anello S, con intorno ipercubico V di lato l a d dimensioni e con legge di evoluzione $\phi : S^{l^d} \to S$, esso può essere simulato da un insieme di automi cellulari unidimensionali definiti su un anello R di cardinalità $|S|^{l^{d-1}}$, con intorno U di lunghezza l che si evolvono in parallelo con la stessa

regola locale $\psi : R^l \to R$. Definiamo U nel modo seguente:

$$U(i) = \sum_{j_2,\ldots,j_d} V(i, j_2, \ldots, j_d)|S|^{\mathrm{dec}(j_2,\ldots,j_d)}$$

Si ricorda che $\mathrm{dec}(a_1, \ldots, a_n) = \sum_{h=1}^{n} a_h|A|^{h-1}$ dove A è l'anello di definizione degli a_h; ovvero, la funzione dec ritorna il numero in base 10 di cui $(a_1, \ldots, a_n)$ è la rappresentazione in base $|A|$. Nel caso menzionato sopra l'anello di definizione di $j_2, \ldots, j_d$ è $\mathbb{Z}_l$. Supponiamo ora che sotto l'effetto di ϕ l'intorno V si sia trasformato nell'intorno V'. Definiamo ψ come:

$$\psi(U) = \sum_{j_2,\ldots,j_d} V'(1, j_2, \ldots, j_d)|S|^{\mathrm{dec}(j_2,\ldots,j_d)}$$

8.1 Esempio

È dato l'automa cellulare bidimensionale con intorno di Moore definito su S al tempo t

c_1	c_2	c_3
c_4	c_5	c_6
c_7	c_8	c_9

con le condizioni al contorno periodiche, ovvero ripiegato in forma toroidale. Questo è equivalente ai tre automi unidimensionali con intorno di lunghezza 3 (sempre con condizioni al contorno periodiche) definiti su un anello R di cardinalità $|S|^3$

$c_7 + c_1\|S\| + c_4\|S\|^2$	$c_8 + c_2\|S\| + c_5\|S\|^2$	$c_9 + c_3\|S\| + c_6\|S\|^2$
$c_1 + c_4\|S\| + c_7\|S\|^2$	$c_2 + c_5\|S\| + c_8\|S\|^2$	$c_3 + c_6\|S\| + c_9\|S\|^2$
$c_4 + c_7\|S\| + c_1\|S\|^2$	$c_5 + c_8\|S\| + c_2\|S\|^2$	$c_6 + c_9\|S\| + c_3\|S\|^2$

Se al tempo $t+1$ la legge di evoluzione ϕ ha trasformato V in

c'_1	c'_2	c'_3
c'_4	c'_5	c'_6
c'_7	c'_8	c'_9

allora al variare di U si ha

$$\psi(c_7 + c_1|S| + c_4|S|^2, c_8 + c_2|S| + c_5|S|^2, c_9 + c_3|S| + c_6|S|^2) = c'_8 + c'_2|S| + c'_5|S|^2$$
$$\psi(c_1 + c_4|S| + c_7|S|^2, c_2 + c_5|S| + c_8|S|^2, c_3 + c_6|S| + c_9|S|^2) = c'_2 + c'_5|S| + c'_8|S|^2$$
$$\psi(c_4 + c_7|S| + c_1|S|^2, c_5 + c_8|S| + c_2|S|^2, c_6 + c_9|S| + c_3|S|^2) = c'_5 + c'_8|S| + c'_2|S|^2$$

e similmente per gli *shift* di U. È evidente che questa trasformazione è biiettiva: con i tre automi unidimensionali si può ricostruire l'automa bidimensionale convertendo i numeri in base $|R|$ in rappresentazioni vettoriali di numeri in base $|S|$. □

8.3.2 Mappa diagonale

Solitamente nella simulazione di automi si parte da un automa d-dimensionale e si cerca di simularne il comportamento con uno o più automi unidimensionali. L'idea della mappa diagonale nasce in modo opposto: prendiamo un automa ad una dimensione e vediamo quali automi pluridimensionali possono essere simulati con l'automa dato. Ne ricaveremo una regola applicabile ad una sottoclasse di automi a più dimensioni. La nozione della mappa diagonale si basa sul fatto che se n è un intero con fattorizzazione in primi $p_1^{e_1} \cdots p_k^{e_k}$ sussiste l'isomorfismo di gruppi

$$C_n \cong C_{p_1}^{e_1} \times \ldots \times C_{p_k}^{e_k} \tag{20}$$

dove C_n è il gruppo ciclico generato da $(12\ldots n)$. Questo isomorfismo suggerisce un modo naturale per mappare una stringa di lunghezza n su un reticolo k-dimensionale con lati di lunghezze $p_1^{e_1}, \ldots, p_k^{e_k}$. Per comodità di notazione esprimiamo con X^i (con i che varia fra 0 e $n-1$) la $(i+1)$-esima cella di un automa unidimensionale di lunghezza n con (r_0, r_1)-intorni tali che $r_0 + r_1 + 1 \leq n$. Chiamiamo μ l'isomorfismo (20): μ è la mappa diagonale. Allora si ha

$$\mu(X) = (X_1, \ldots, X_k)$$

dove X_j è il generatore di $C_{p_j}^{e_j}$. Usando μ si può mappare in modo coerente la i-esima cella dell'automa unidimensionale sulla cella dell'automa pluridimensionale corrispondente alla posizione

$$\mu(X^i) = (\mu(X))^i = (X_1^i, \ldots, X_k^i)$$

Questo implica che l'intorno della cella $\mu(X^i)$ sia dato dalle celle

$$\mu(X^{i-r_0}), \ldots, \mu(X^{i+r_1})$$

8.2 Esempio

Sia dato un automa unidimensionale di lunghezza 6 e applichiamo la mappa diagonale per ottenere un automa bidimensionale su una griglia 2×3. Si ottiene un isomorfismo

$$C_6 = \langle X \rangle \cong \langle X_1 \rangle \times \langle X_2 \rangle = C_2 \times C_3$$

da cui

$$\begin{aligned}
\mu(X^0) = \mu(1) &= (X_1^0, X_2^0) = (1,1) \\
\mu(X) &= (X_1, X_2) \\
\mu(X^2) &= (1, X_2^2) \\
\mu(X^3) &= (X_1, 1) \\
\mu(X^4) &= (1, X_2) \\
\mu(X^5) &= (X_1, X_2^2)
\end{aligned}$$

L'andamento dell'automa bidimensionale è meglio esplicato da un diagramma:

1	2	3	4	5	6

1	5	3
4	2	6

Si può osservare dalla figura che la forma dell'intorno dell'automa bidimensionale è diagonale (di qui il nome della mappa). □

8.4 Reversibilità degli automi bidimensionali

Abbiamo visto nella sezione 7.2 che esiste un algoritmo per determinare la reversibilità di una regola locale su automi unidimensionali di lunghezza infinita. I creatori dell'algoritmo avvertono tuttavia che "il loro risultato è limitato a stringhe ad una dimensione. Sebbene le tecniche impiegate si possano in principio estendere a più dimensioni, sembra che in pratica sia molto difficile implementarle. È estremamente probabile che una generalizzazione pluridimensionale richieda un approccio completamente diverso[22]. Jarkko Kari ha poi dimostrato (cfr. [Kar91]) che la reversibilità di automi di dimensione 2 con intorno di Moore non è decidibile, ovvero che non esiste un algoritmo[23] in grado di decidere se una data regola locale sia o non sia iniettiva. La dimostrazione di Kari si basa sul "problema della quadrettatura del piano[24]. È dato un numero finito di quadrati di lato unitario con i bordi colorati. Disponiamo di infinite copie di questi quadrati ma non possiamo

[22]Cfr. [AP72], pag. 449.

[23]Per definizione un algoritmo è una procedura di un numero finito di passi.

[24]Tradotto da *tiling problem*.

ruotarli. Una quadrettatura è valida se per ogni coppia di quadrati contigui il bordo comune ha lo stesso colore. Si deve decidere se con i quadrati dati è possibile quadrettare tutto il piano. R. Berger ha provato nel 1966 ([Ber66]) che il problema non è decidibile, e Kari ha dimostrato che il problema della decidibilità della reversibilità di un automa bidimensionale con intorno di Moore è equivalente al problema della quadrettatura del piano. La dimostrazione si basa sul fatto che con l'insieme finito di quadrati unitari è possibile costruire un particolare automa cellulare bidimensionale che è reversibile se e solo se il piano può essere quadrettato.

9 Conclusione

Fra le varie tematiche affrontate in questa tesi, tre principalmente sono quelle che abbisognano di ulteriore ricerca e sviluppo.

1. Le considerazioni riguardanti la struttura del gruppo $\mathcal{G}^n(R)$ (vedi capitolo 5 a pagina 67). In particolare si sono trovate (ahimé troppo tardi per essere incluse in questa sede) delle convenienti rappresentazioni fedeli dei gruppi G_i di cui $\mathcal{G}^n(R)$ è il prodotto diretto; inoltre sembra che ci possano essere diversi legami con la teoria dei codici correttori d'errore.

2. La "questione aperta di cui si parla a pagina 96. Purtroppo la dimostrazione di Jarkko Kari è costruita *ad hoc* per la regola "180. Sarebbe opportuno dimostrare l'ipotesi di invertibilità su lunghezze dispari in modo indipendente dalla struttura "interna della trasformazione, così da applicarla anche alle altre trasformazioni menzionate sopra.

3. L'ultimo capitolo, e in particolare i concetti di mappe transdimensionali, vanno approfonditi molto maggiormente.

È inoltre evidente dal lavoro svolto che trattare la non linearità con metodi combinatorici è sempre difficile e non sempre fruttuoso. Speriamo di avere comunque mostrato attraverso le parti di calcolo numerico che un tentativo in questo senso non deve necessariamente restare legato al limbo della teoria inapplicabile.

A Appendice: Il Programma CAInvPrm

```
/*
    Name:       CAInvPrm
    Author:     Leo Liberti
    Purpose:    To find the inverse of a local CA rule (if it exists)
                with the permutation method (see dissertation)
    Source:     MS Visual C++ 4.0
    History:
                21/06/97    1.0     works.
                26/06/97    1.1     Added APIL support and an option
                                    for testing all transformations
                                    with a FOR clause.
                26/06/97    1.2     rewrote cmd line parser and added
                                    option for simpler output
                28/06/97    1.3     added -nodec flag and fixed some
                                    parser bugs. Also changed the
                                    way -all flag cycles through the
                                    transf (now much quicker, w/o APIL)
                30/06/97    1.4     added -infile: option and changed
                                    the way -simple outputs, so can take
                                    a redirection with -simple as an infile
                17/08/97    1.5     added -complex flag
    Notes:                                                  (unsigned)
         |R| ,r_0, r_1, n, *vectors, *localrules            ... int
         |R|^n, *globalrules, |*localrules|, |*globalrules| ... long int
         vectors, or numbers, are in form (v_1, \ldots, v_n);
         global rules are in form (v_0, \ldots, v_{|R|^n - 1}) (different from
         dissertation standards); local rules as in thesis

         EXIT CODES: 0 - all ok.
                     1 - not enough memory.
                     2 - r0 + r1 + 1 > n.
                     3 - -v flag set with -all flag
                     4 - other command line mistake
                     5 - can't open infile for input
                     6 - infile is 0 bytes long
                     7 - infile doesn't contain transformations
                     8 - could not read transformation from infile
                     9 - infile line too long
                    10 - etc

         N.B. This program relies on APIL.H and APIL.CPP (see later in
              this appendix).
*/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include "apil.h"

#define APIL
#ifdef MESG
    #undef MESG
#endif
#define MESG "CAInvPrm 1.5 (Using perms. & APIL 1) - Leo Liberti 1997\n"
#ifndef TRUE
    #define TRUE 1
#endif
#ifndef FALSE
    #define FALSE 0
#endif
#ifndef PLUS
    #define PLUS 1
#endif
#ifndef MINUS
    #define MINUS -1
#endif
#ifndef BASE
    #define BASE 10
#endif
#ifdef ARGS
    #undef ARGS
#endif
#define ARGS 5
#ifndef FNLEN
    #define FNLEN 80
#endif
#ifndef BUFLEN
    #define BUFLEN 160
#endif

/* -------- begin function list -------- */

unsigned int mod(long a, unsigned int n)
{
        unsigned int a3;
        long a1, a2;

        a1 = a % (long) n;
        if(a1 < 0)
                a2 = n + a1;
        else
                a2 = a1;
        a3 = (unsigned int) a2;

        return a3;
}

/* Following functions already defined in apil.cpp; only read them in
   if APIL is not defined */

#ifndef APIL
unsigned long power(unsigned long a, unsigned long n)
{
    unsigned long u, v, t;

    u = n;
    v = 1;
    t = a;
    while(u != 0)
    {
        if(u % 2 == 1)
        {
            v = v * t;
        }
        t = t * t;
        u = u / 2;
```

```
    }
    return v;
}

int sgn(long a)
{
    if(a >= 0)
        return (int) PLUS;
    else
        return (int) MINUS;
}
char int2char(unsigned short int p)
{
    return p + '0';
}
void check_char(char *t, unsigned long c)
{
    if(strlen(t) > c)
        *(t + c) = '\0';
    if(!t)
    {
        // error 2 : no memory
        fprintf(stderr, "CAInvPrm: check_char: ERROR: Not enough memory.\n");
        exit(1);
    }
}
char *itoa(unsigned long p)
{
    unsigned long digits, i;
    char *ret;
    if(p == 0)
        digits = 1;
    else
        digits = (unsigned long) floor(log10((double) p)) + 1;
    // ret = new char[digits];
    ret = (char *) malloc(digits);
    check_char(ret, digits);
    for(i = 0; i < digits; i++)
        *(ret + digits - i - 1) = int2char((p / \
                    power(BASE, (unsigned long) i)) % BASE);
    return ret;
}
#endif
/* end of functions defined in apil.cpp */

/* this is for vectors */
unsigned int *dec2vect(unsigned int R, unsigned long veclen, \
                        unsigned long s)
{
        unsigned long int i;
        unsigned int *v;
        v = (unsigned int *) malloc(veclen * sizeof(int));
        if(!v)
        {
            fprintf(stderr, "CAInvPrm: dec2vect: ERROR: Not enough memory.\n");
            exit(1);
        }
        for(i = 1; i <= veclen; i++)
                *(v + i - 1) = (s / power(R, veclen - i)) % R;
        return v;
}
/* this is for rules */
unsigned int *dec2local(unsigned int R, unsigned long veclen, \
                        unsigned long s)
{
        unsigned long int i;
        unsigned int *v;
        v = (unsigned int *) malloc(veclen * sizeof(int));
        if(!v)
        {
            fprintf(stderr, "CAInvPrm: dec2local: ERROR: Not enough memory.\n");
            exit(1);
        }
        for(i = 0; i < veclen; i++)
                *(v + veclen - i - 1) = (s / power(R, i)) % R;
        return v;
}
// ... with APIL support
unsigned int *dec2local(unsigned int R, unsigned long veclen, api s)
{
        api api_tmp;
        unsigned long int i, tmp;
        unsigned int *v, t;
        v = (unsigned int *) malloc(veclen * sizeof(int));
        if(!v)
        {
            fprintf(stderr, "CAInvPrm: dec2local (APIL): ERROR: Not enough memory.\n");
            exit(1);
        }
        tmp = 1;
        for(i = 0; i < veclen; i++)
        {
            api_tmp = (s / tmp) % R;
            t = atoi(api_tmp.api2char());
            *(v + veclen - i - 1) = t;
            tmp = tmp * R;
        }
```

```
        return v;
}
// Transform a local in a decimal rep
unsigned long local2dec(unsigned int R, unsigned long veclen, \
                        unsigned int *localrule)
{
    unsigned long i, dec, tmp;
    dec = 0;
    tmp = 1;
    for(i = 0; i < veclen; i++)
    {
        dec = dec + *(localrule + veclen - i - 1) * tmp;
        tmp = tmp * R;
    }
    return dec;
}
// ...with APIL support
api local2dec_api(unsigned int R, unsigned long veclen, \
             unsigned int *localrule)
{
    unsigned long i;
    api dec, tmp;
    dec = 0;
    tmp = 1;
    for(i = 0; i < veclen; i++)
    {
        dec = dec + tmp * (*(localrule + veclen - i - 1));
        tmp = tmp * R;
    }
    return dec;
}

unsigned long apply_rule(unsigned int R, unsigned int n, \
                         unsigned int r0, unsigned int r1, \
                         unsigned int *localrule, \
                         unsigned long vector)
{
        unsigned int i, j, l, d, m;
        unsigned int *v;
        unsigned long t, td, c, c1; //p;
        long tmp;
        i = 0;
        t = power(R, n);
        d = r0 + r1 + 1;
        td = power(R, d);
        v = dec2vect(R, n, vector);
        c = 0;
        for(i = 0; i < n; i++)
        {
                l = n - i - 1;
                c1 = 0;
                for(j = 0; j < d; j++)
                {
                        tmp = (long) l - (long) r0 + (long) j;
                        m = mod(tmp, n);
                        /* Per la definizione di mod()
                           vedi piu' sotto */
                        c1 = c1 + (*(v + m)) * power(R, d - 1 - j);
                }
                // p = (rule / power(R, c1)) % R;
                c = c + (*(localrule + td - c1 - 1)) * power(R, i);
        }
        return c;
}
unsigned int *BT2AT(unsigned int R, unsigned int n, \
                    unsigned int r0, unsigned int r1, \
                    unsigned int *psi)
{
        unsigned int d, i, j, l, t1, t2, t3;
        unsigned int *phi;
        d = r0 + r1 + 1;
        t1 = power(R, n);
        t2 = power(R, d);
        t3 = power(R, n - d);
        phi = (unsigned int *) malloc(t1 * sizeof(int));
        for(i = 0; i < t2; i++)
        {
                l = i * t3;
                for(j = 0; j < t3; j++)
                        *(phi + l + j) = *(psi + i);
        }
        return phi;
}
unsigned int *AT2BT(unsigned int R, unsigned int n, \
                    unsigned int r0, unsigned int r1, \
                    unsigned int *phi)
{
        unsigned int d, i, t1, t2, t3;
        unsigned int *psi;
        d = r0 + r1 + 1;
        t1 = power(R, n);
        t2 = power(R, d);
        t3 = power(R, n - d);
        psi = (unsigned int *) malloc(t2 * sizeof(int));
        for(i = 0; i < t2; i++)
        {
            *(psi + i) = *(phi + i * t3);
```

```
        }
        return psi;
}
unsigned long *local2global(unsigned int R, unsigned int n, \
                            unsigned int r0, unsigned int r1, \
                            unsigned int *localrule)
{
    unsigned long i, t, *globalrule;
    t = power(R, n);
    globalrule = (unsigned long *) malloc(t * sizeof(long int));
    if(!globalrule)
    {
          fprintf(stderr, "CAInvPrm: local2global: ERROR: Not enough memory.\n");
          exit(1);
    }
    for(i = 0; i < t; i++)
        *(globalrule + i) = apply_rule(R, n, r0, r1, localrule, i);
    return globalrule;
}
/* this function also attempts to find a shortened version of
   the local rule corresponding to *globalrule and sets r1 to
   the correct value (so if r1 = n - r0 - 1 it means there is
   no shortened version) */
unsigned int *global2local(unsigned int R, unsigned int n, \
                           unsigned int r0, unsigned int *r1, \
                           unsigned long *globalrule)
{
    unsigned short flag;
    unsigned int *localrule, *shortlocal, l, k;
    unsigned long t, i, j, d, y, z1, z2;
    double x;
    t = power(R, n);
    localrule = (unsigned int *) malloc(t * sizeof(unsigned int));
    /* find the long version of the local rule */
    for(i = 0; i < t; i++)
        *(localrule + t - i - 1) = (*(globalrule + i) / power(R, n - r0 - 1)) % R;
    /* see if it can be shortened */
    flag = TRUE;
    /* find a possible "n - d" */
    l = *localrule;
    i = 0;
    while(*(localrule + i + 1) == l)
        i++;
    if(i > 1)
    {
        x = log((double) i) / log((double) R);
        y = (unsigned long) floor(x + 0.5);
        /* test if n-d=y is in \mathbb{Z} */
        z1 = power(R, y);
        if(z1 == i)
        {
            /* test if n-d=y is all right for the other positions */
            d = n - y;
            /* check that d >= 2 */
            if(d < 3 && n > 2)
            {
                d = 3;
                y = n - d;
                z1 = power(R, y);
            }
            else if(d == 1 && n == 2)
            {
                d = 2;
                y = 1;
                z1 = R;
            }
            z2 = power(R, d);
            *r1 = d - r0 - 1;
            for(j = 0; j < z2; j++)
            {
                k = *(localrule + j * z1);
                for(i = 1; i < z1; i++)
                {
                    if(*(localrule + j * z1 + i) != k)
                    {
                        flag = FALSE;
                        break;
                    }
                }
                if(flag == FALSE)
                    break;
            }
            if(flag == TRUE)
                shortlocal = AT2BT(R, n, r0, *r1, localrule);
            else
            {
                *r1 = n - r0 - 1;
                shortlocal = localrule;
            }
        }
        else
        {
            *r1 = n - r0 - 1;
            shortlocal = localrule;
        }
    }
    else
    {
        *r1 = n - r0 - 1;
        shortlocal = localrule;
    }
    return shortlocal;
```

```
}
void local_out(unsigned long loc_len, unsigned int *localrule, FILE *fp)
{
    unsigned long i;
    fputs("(", fp);
    for(i = 0; i < loc_len - 1; i++)
        fprintf(fp, "%u,", *(localrule + i));
    fprintf(fp, "%u)", *(localrule + loc_len - 1));
}
/* verify that prm is a permutation, and if it is, find the inverse.
   if it isn't, return an array with zero entries */
unsigned long *prm_inverse(unsigned long prm_len, unsigned long *prm)
{
    unsigned short int *checklist;
    unsigned long int i, *inv;
    int flag;
    checklist = (unsigned short *) malloc(prm_len * sizeof(short int));
    if(!checklist)
    {
        fprintf(stderr, "CAInvPrm: prm_inverse: ERROR: Not enough memory.\n");
        exit(1);
    }
    inv = (unsigned long *) malloc(prm_len * sizeof(long int));
    if(!inv)
    {
        fprintf(stderr, "CAInvPrm: prm_inverse: ERROR: Not enough memory.\n");
        exit(1);
    }
    flag = TRUE;
    for(i = 0; i < prm_len; i++)
    {
        (*(checklist + i)) = 0;
        (*(inv + i)) = 0;
    }
    for(i = 0; i < prm_len; i++)
    {
        if(++(*(checklist + *(prm + i))) > 1)
        {
            flag = FALSE;
            break;
        }
    }
    free(checklist);
    /*
    // the simplest response is the quickest!
    if(flag == TRUE)
        *inv = 1;
    */
    if(flag == TRUE)
        for(i = 0; i < prm_len; i++)
            *(inv + *(prm + i)) = i;
    return inv;
}
/* careful: if outputting in cycle format (cycleflag == TRUE)
   and prm is not a permutation, there is an infinite loop */
void prm_out(unsigned int R, unsigned long prm_len, unsigned long *prm, \
             FILE *fp, short int cycleflag, unsigned int complexflag)
{
    unsigned long i, j, jsav, k, l;
    unsigned short *checklist;
    unsigned int length;
    char str[80], *baseNum;
    if(complexflag == TRUE)
    {
        length = (unsigned int) floor(log((double) prm_len) / log((double) R) + 0.5);
        baseNum = (char *) malloc(length * sizeof(char));
        check_char(baseNum, length);
    }
    if(cycleflag == TRUE)
    {
        checklist = (unsigned short int *) malloc(prm_len * sizeof(short int));
        for(i = 0; i < prm_len; i++)
            *(checklist + i) = 0;
        i = 0;
        while(*(prm + i) == i)
        {
            (*(checklist + i))++;
            i++;
            if(i == prm_len)
                break;
        }
        if(i == prm_len)
            fputs("(1)", fp);
        else
        {
            /* start finding cycles */
            j = i;
            for(;;)
            {
                /* follow through a cycle */
                for(k = 0;;k++)
                {
                    if(k > 0)
                    {
                        if(complexflag == FALSE)
                            fputs(itoa(j), fp);
                        else
```

```
                    {
                        for(l = 1; l <= length; l++)
                            *(baseNum + l - 1) = int2char((j / power(R, length - l)) % R);
                        fputs(baseNum, fp);
                    }
                }
                (*(checklist + j))++;
                jsav = j;
                j = *(prm + jsav);
                if(j == i)
                {
                    if(k > 0)
                    {
                        if(complexflag == FALSE)
                            fputs(")", fp);
                        else
                            fputs(")\n",fp);
                    }
                    break;
                }
                else
                {
                    if(k == 0)
                    {
                        strcpy(str, "(");
                        if(complexflag == FALSE)
                            fputs(strcat(str, itoa(jsav)), fp);
                        else
                        {
                            for(l = 1; l <= length; l++)
                                *(baseNum + l - 1) = int2char((jsav / power(R, length - l)) % R);
                            fputs("\n", fp);
                            fputs(strcat(str, baseNum), fp);
                        }
                    }
                    fputs(",", fp);
                }
            }
            /* search for an unused number */
            j = 0;
            while(*(checklist + j) > 0)
            {
                j++;
                if(j > prm_len - 1)
                    break;
            }
            /* check if there are any */
            if(j == prm_len)
                break;
            /* there are some; start again with new cycle */
            i = j;
        }
    }
    free(checklist);
}
else
{
    fputs("(", fp);
    for(i = 0; i < prm_len - 1; i++)
    {
        fputs(itoa(*(prm + i)), fp);
        fputs(",", fp);
    }
    fputs(itoa(*(prm + prm_len - 1)), fp);
    fputs(")", fp);
}
}

void help()
{
    fprintf(stdout, MESG);
    printf("    This program finds the inverse (if possible) of cellular\n");
    printf("    automata transformations expressed in decimal or local\n");
    printf("    representation (see relevant paper for details). Output is\n");
    printf("    appended both to stdout and to the file CAInvPrm.log. Usage:\n");
    printf("\n    [drive:\\dir\\]CAInvPrm [options] R n r0 r1 [sigma]\n\n");
    printf("    where R is the size of the ring of interest, n is the length of\n");
    printf("    cellular automata, r0 is the left radius, r1 is the right radius\n");
    printf("    sigma is the decimal representation of the transformation to invert.\n");
    printf("    Options: -local program asks for local representation; argument sigma\n");
    printf("                    is then NOT required. This option cannot be used with -all.\n");
    printf("             -all program tests all transformations of {\\cal B}_{r_0,r_1}^n(R)\n");
    printf("                  this option cannot be used with either -local or -infile.\n");
    printf("             -simple output is as follows:\"transformation being tested\" if it\n");
    printf("                     is invertible, nothing if it isn't.\n");
    printf("             -complex decimal representations is suppressed from cyclic\n");
    printf("                      representations of permutations.\n");
    printf("             -nodec program doesn't output any decimal representation;\n");
    printf("                    useful with -all, to avoid losing time with conversions.\n");
    printf("             -nostdout doesn't output to stdout, only to log file. This option\n");
    printf("                       saves time by not calling screen routines.\n");
    printf("             -outfile:<file> sets a name for the log file. Default is\n");
    printf("                             CAInvPrm.log.\n");
    printf("             -infile:<file> sets a name for an input file. Each line of the\n");
    printf("                     file must contain a decimal or local rep. of a transf.\n");
    printf("                     This option cannot be used with either -all or -local.\n\n");
}

int main(int argc, char *argv[])
{
    api api_sig, api_inv; // api_counter, api_limit, api_R;
```

```
int arg_i;
unsigned int R, n, r0, r1, *r1tmp, tmp, *sigm, *ta, nodecflag, invflag;
unsigned int vflag, allflag, simpleflag, stopflag, numofargs, num;
unsigned int nostdoutflag, alternatefileflag, infileflag, int_buffer;
unsigned int complexflag;
unsigned long int i, td, tdtmp, tn, *sigma, *tau, sig, longlimit;
unsigned long int counter;
FILE *f, *g;
char *just4output, alternatefile[FNLEN], infile[FNLEN], buffer[BUFLEN], c;

longlimit = ULONG_MAX;

////////////////////////////////////
// here beginneth the new parser //
////////////////////////////////////

// init
vflag = FALSE;
simpleflag = FALSE;
allflag = FALSE;
nodecflag = FALSE;
nostdoutflag = FALSE;
alternatefileflag = FALSE;
infileflag = FALSE;
complexflag = FALSE;
strcpy(alternatefile, "CAInvPrm.log");
numofargs = ARGS;
num = numofargs;
api_sig = 0;
sig = NULL;
g = NULL;

// check that there is a valid command line!
if(argc <= ARGS)
{
    help();
    exit(0);
}

// parse
for(arg_i = 1; arg_i < argc; arg_i++)
{
    if(*(argv[arg_i]) == '-')
    {
        // options
        if(strcmp(argv[arg_i], "-local") == 0)
        {
            vflag = TRUE;
            numofargs = ARGS - 1;
        }
        else if(strcmp(argv[arg_i], "-all") == 0)
        {
            allflag = TRUE;
            numofargs = ARGS - 1;
        }
        else if(strcmp(argv[arg_i], "-simple") == 0)
            simpleflag = TRUE;
        else if(strcmp(argv[arg_i], "-complex") == 0)
            complexflag = TRUE;
        else if(strcmp(argv[arg_i], "-nodec") == 0)
            nodecflag = TRUE;
        else if(strcmp(argv[arg_i], "-nostdout") == 0)
            nostdoutflag = TRUE;
        else if(strstr(argv[arg_i], "-outfile:") != NULL)
        {
            alternatefileflag = TRUE;
            strcpy(alternatefile, argv[arg_i] + 9);
        }
        else if(strstr(argv[arg_i], "-infile:") != NULL)
        {
            infileflag = TRUE;
            strcpy(infile, argv[arg_i] + 8);
            numofargs = ARGS - 1;
        }
        else
            fprintf(stderr, "CAInvPrm: main: WARNING: Unrecognized cmd line flag %s.\n", argv[arg_i]);
    }
    else
    {
        // arguments
        switch(num)
        {
        case 1:
            if(numofargs == ARGS)
            {
                api_sig = argv[arg_i];
                if(api_sig < longlimit)
                {
                    sig = atol(api_sig.api2char());
                    api_sig = 0;
                }
                num--;
            }
            else
            {
                fprintf(stderr, "CAInvPrm: main: WARNING: Expecting %u cmd line args, found %u.", \
                                numofargs, numofargs + num);
                fprintf(stderr, "CAInvPrm: main: WARNING: Ignoring extra cmd line argument %s", \
                                argv[numofargs + num]);
            }
            break;
        case 2:
            r1 = atoi(argv[arg_i]);
            num--;
            break;
        case 3:
            r0 = atoi(argv[arg_i]);
            num--;
            break;
```

```
        case 4:
            n = atoi(argv[arg_i]);
            num--;
            break;
        case 5:
            R = atoi(argv[arg_i]);
            num--;
            break;
        default:
            fprintf(stderr, "CAInvPrm: main: WARNING: Expecting %u cmd line args, found more.", numofargs);
            fprintf(stderr, "CAInvPrm: main: WARNING: Ignoring extra cmd line argument %s", argv[arg_i]);
        }
    }
}
// not enough arguments on cmd line
if(numofargs + num - ARGS> 0)
{
    help();
    fprintf(stderr, "CAInvPrm: main: ERROR: Expecting %u cmd line args, found %u\n", numofargs, numofargs - num);
    exit(4);
}
// check -local and -all flags
if(vflag == TRUE && allflag == TRUE)
{
    help();
    fprintf(stderr, "CAInvPrm: main: ERROR: -local and -all flag cannot be set together.\n");
    exit(3);
}
// check -local and -infile flags
if(vflag == TRUE && infileflag == TRUE)
{
    help();
    fprintf(stderr, "CAInvPrm: main: ERROR: -local and -infile flag cannot be set together.\n");
    exit(3);
}
// check -infile and -all flags
if(infileflag == TRUE && allflag == TRUE)
{
    help();
    fprintf(stderr, "CAInvPrm: main: ERROR: -infile and -all flag cannot be set together.\n");
    exit(3);
}
////////////////////////////////
// here endeth the new parser //
////////////////////////////////
// check input data
if(r0 + r1 + 1 > n)
{
    help();
    fprintf(stderr, "CAInvPrm: main: ERROR: r0 + r1 + 1 must be less than or equal to n.\n");
    exit(2);
}
// set up everything for work
tn = power(R, n);
td = power(R, r0 + r1 + 1);
sigm = (unsigned int *) malloc(td * sizeof(int));
// check vflag
if(vflag == TRUE)
{
    fprintf(stderr, "CAInvPrm: main: WARNING: -v option set, scanning for input.\n");
    // inputting local rep
    for(i = 1; i <= td; i++)
    {
        fprintf(stderr, "CAInvPrm: Input rule sigma, pos %lu out of %lu:", i, td);
        fscanf(stdin, "%u", &tmp);
        *(sigm + i - 1) = tmp;
    }
    // calculating decimal rep
    if(nodecflag == FALSE)
    {
        api_sig = local2dec_api(R, td, sigm);
        if(api_sig <= longlimit)
        {
            sig = atol(api_sig.api2char());
            api_sig = 0;
        }
    }
    else
    {
        api_sig = 0;
        sig = NULL;
    }
}
else if(numofargs == ARGS)
{
    if(sig == NULL)
        sigm = dec2local(R, td, api_sig);
    else
        sigm = dec2local(R, td, sig);
}
// check allflag
if(allflag == TRUE)
{
    // initialize sigm
    sigm = dec2local(R, td, 0);
    sig = NULL;
    api_sig = 0;
}
```

```
// check infileflag
if(infileflag == TRUE)
{
    // initialize sigm
    if((g = fopen(infile, "r")) == NULL)
    {
        fprintf(stderr, "CAInvPrm: main: ERROR: Can't open %s for input.\n", infile);
        exit(5);
    }
    else
    {
        if(fscanf(g, "%c", &c) == EOF)      // deal with rems and etcs
        {
            fprintf(stderr, "CAInvPrm: main: ERROR: %s is 0 bytes long.\n", infile);
            exit(6);
        }
        while(c != '(' && (c < '0' || c > '9'))
        {
            while(c != '\n')
            {
                if(fscanf(g, "%c", &c) == EOF)
                {
                    fprintf(stderr, "CAInvPrm: main: ERROR: %s does not contain transformations.\n", infile);
                    exit(7);
                }
            }
            if(fscanf(g, "%c", &c) == EOF)
            {
                fprintf(stderr, "CAInvPrm: main: ERROR: %s does not contain transformations.\n", infile);
                exit(7);
            }
        }
        if(c == '(')      // local reps
        {
            for(i = 0; i < td - 1; i++)
            {
                if(fscanf(g, "%u,", &int_buffer) == EOF)
                {
                    fprintf(stderr, "CAInvPrm: main: ERROR: Could not read transformation from %s.\n", infile);
                    exit(8);
                }
                else
                    *(sigm + i) = int_buffer;
            }
            if(fscanf(g, "%u)", &int_buffer) == EOF)
            {
                fprintf(stderr, "CAInvPrm: main: ERROR: Could not read transformation from %s.\n", infile);
                exit(8);
            }
            else
                *(sigm + td - 1) = int_buffer;
            if(nodecflag == FALSE)
            {
                api_sig = local2dec_api(R, td, sigm);
                sig = NULL;
                if(api_sig <= longlimit)
                {
                    sig = atol(api_sig.api2char());
                    api_sig = 0;
                }
            }
        }
        else                // decimal reps
        {
            i = 0;
            while(c != '\n' && c != EOF)
            {
                buffer[i] = c;
                i++;
                if(i >= BUFLEN)
                {
                    fprintf(stderr, "CAInvPrm: main: ERROR: %s - line too long.\n", infile);
                    exit(9);
                }
                fscanf(g, "%c", &c);
            }
            buffer[i] = '\0';
            api_sig = buffer;
            sig = NULL;
            if(api_sig <= longlimit)
            {
                sig = atol(buffer);
                api_sig = 0;
            }
            if(sig == NULL)
                sigm = dec2local(R, td, api_sig);
            else
                sigm = dec2local(R, td, sig);
        }
    }
}
// initialize loop
// try opening the log file
f = fopen(alternatefile, "a+");
fprintf(stdout, MESG);
fprintf(stdout, "\nCalculating inverses of transformations in B_(%u,%u)^%u(Z_%u):\n\n", r0, r1, n, R);
if(f != NULL)
{
    fprintf(f, MESG);
    fprintf(f, "\nCalculating inverses of transformations in B_(%u,%u)^%u(Z_%u):\n\n", r0, r1, n, R);
}
r1tmp = (unsigned int *) malloc(sizeof(unsigned int));
invflag = FALSE;
stopflag = FALSE;
```

```
while(stopflag == FALSE)
{
    // do all the work!
    sigma = local2global(R, n, r0, r1, sigm);
    tau = prm_inverse(tn, sigma);
    // check simpleflag
    if(simpleflag == TRUE)
    {
        // simplest things are the quickest!
        if(nodecflag == FALSE)
        {
            if(sig == NULL)
                just4output = api_sig.api2char();
            else
                just4output = itoa(sig);
            if(*tau != *(tau + 1))
            {
                if(nostdoutflag == FALSE)
                    fprintf(stdout, "%s\n", just4output);
                if(f != NULL)
                    fprintf(f, "%s\n", just4output);
                invflag = TRUE;
            }
        }
        else
        {
            if(*tau != *(tau + 1))
            {
                if(nostdoutflag == FALSE)
                {
                    local_out(td, sigm, stdout);
                    fprintf(stdout, "\n");
                }
                if(f != NULL)
                {
                    local_out(td, sigm, f);
                    fprintf(f, "\n");
                }
                invflag = TRUE;
            }
        }
    }
    else
    {
        // complex stdout output
        if(*tau != *(tau + 1))
        {
            *r1tmp = r1;
            ta = global2local(R, n, r0, r1tmp, tau);
            tdtmp = power(R, r0 + *r1tmp + 1);
            invflag = TRUE;
        }
        if(nostdoutflag == FALSE)
        {
            // output to stdout
            fputs("*************************** \n", stdout);
            // fputs(MESG, stdout);
            // fprintf(stdout, "\nCalculating inverse of transformation in B_(%u,%u)^%u(Z_%u):\n\n", r0, r1, n, R);
            if(nodecflag == FALSE)
            {
                if(sig != NULL)
                    fprintf(stdout, " Decimal representation: %lu\n", sig);
                else
                    fprintf(stdout, " Decimal representation: %s\n", api_sig.api2char());
            }
            fputs(" Local representation: ", stdout);
            local_out(td, sigm, stdout);
            fputs("\n Global representation: ", stdout);
            prm_out(R, tn, sigma, stdout, FALSE, complexflag);
            if(*tau != *(tau + 1))
            {
                fputs("\n Global representation (cycles): ", stdout);
                prm_out(R, tn, sigma, stdout, TRUE, complexflag);
                fputs("\n\nFound Inverse: \n\n", stdout);
                api_inv = local2dec_api(R, tdtmp, ta);
                if(nodecflag == FALSE)
                    fprintf(stdout, " Decimal representation: %s", api_inv.api2char());
                fprintf(stdout, "\n Local representation with r_0=%u, r_1=%u: ", r0, *r1tmp);
                local_out(tdtmp, ta, stdout);
                fputs("\n Global representation: ", stdout);
                prm_out(R, tn, tau, stdout, FALSE, complexflag);
                fputs("\n Global representation (cycles): ", stdout);
                prm_out(R, tn, tau, stdout, TRUE, complexflag);
            }
            else
                fputs("\n\nTransformation is not invertible", stdout);
            fputs("\n*************************** \n", stdout);
        }
        if(f != NULL)
        {
            fputs("*************************** \n", f);
            // fprintf(f, MESG);
            // fprintf(f, "\nCalculating inverse of transformation in B_(%u,%u)^%u(Z_%u):\n\n", r0, r1, n, R);
            if(nodecflag == FALSE)
            {
                if(sig != NULL)
                    fprintf(f, " Decimal representation: %lu\n", sig);
                else
                    fprintf(f, " Decimal representation: %s\n", api_sig.api2char());
            }
            fputs(" Local representation: ", f);
            local_out(td, sigm, f);
```

```
        fputs("\n Global representation: ", f);
        prm_out(R, tn, sigma, f, FALSE, complexflag);
        if(*tau != *(tau + 1))
        {
            fputs("\n Global representation (cycles): ", f);
            prm_out(R, tn, sigma, f, TRUE, complexflag);
            fputs("\n\nFound Inverse: \n\n", f);
            // api_inv = local2dec_api(R, tdtmp, ta);
            if(nodecflag == FALSE)
                fprintf(f, " Decimal representation: %s", api_inv.api2char());
            fprintf(f, "\n Local representation with r_0=%u, r_1=%u: ", r0, *r1tmp);
            local_out(tdtmp, ta, f);
            fputs("\n Global representation: ", f);
            prm_out(R, tn, tau, f, FALSE, complexflag);
            fputs("\n Global representation (cycles): ", f);
            prm_out(R, tn, tau, f, TRUE, complexflag);
        }
        else
            fputs("\n\nTransformation is not invertible", f);
        fputs("\n**************************** \n", f);
    }
}
// now if -all is set, increase sigm, otherwise check -infile
if(allflag == TRUE)
{
    counter = 1;
    while(*(sigm + td - counter) == R - 1)
    {
        if(counter == td)
        {
            stopflag = TRUE;
            break;
        }
        *(sigm + td - counter) = 0;
        counter++;
    }
    if(stopflag == FALSE)
    {
        (*(sigm + td - counter))++;
        if(nodecflag == FALSE)
            api_sig = local2dec_api(R, td, sigm);
    }
}
else if(infileflag == TRUE)        // check infileflag, else exit
{
    if(fscanf(g, "%c", &c) == EOF)      // normal end of file, exit
    {
        stopflag = TRUE;
    }
    if(stopflag == FALSE)
    {
        while(c != '(' && (c < '0' || c > '9'))
        {
            while(c != '\n')
            {
                if(fscanf(g, "%c", &c) == EOF)
                {
                    stopflag = TRUE;
                    break;           // another normal termination
                }
            }
            if(fscanf(g, "%c", &c) == EOF)
            {
                stopflag = TRUE;
                break;          // yet another normal termination
            }
        }
        if(stopflag == FALSE)
        {
            if(c == '(')      // local reps
            {
                for(i = 0; i < td - 1; i++)
                {
                    if(fscanf(g, "%u,", &int_buffer) == EOF)
                    {
                        fprintf(stderr, "CAInvPrm: main: ERROR: Could not read transformation from %s.\n", \
                                        infile);
                        exit(8);
                    }
                    else
                        *(sigm + i) = int_buffer;
                }
                if(fscanf(g, "%u)", &int_buffer) == EOF)
                {
                    fprintf(stderr, "CAInvPrm: main: ERROR: Could not read transformation from %s.\n", infile);
                    exit(8);
                }
                else
                    *(sigm + td - 1) = int_buffer;
                if(nodecflag == FALSE)
                {
                    api_sig = local2dec_api(R, td, sigm);
                    sig = NULL;
                    if(api_sig <= longlimit)
                    {
                        sig = atol(api_sig.api2char());
                        api_sig = 0;
                    }
                }
            }
            else                // decimal reps
            {
                i = 0;
                while(c != '\n' && c != EOF)
                {
```

```
                        buffer[i] = c;
                        i++;
                        fscanf(g, "%c", &c);
                    }
                    buffer[i] = '\0';
                    api_sig = buffer;
                    sig = NULL;
                    if(api_sig <= longlimit)
                    {
                        sig = atol(buffer);
                        api_sig = 0;
                    }
                    if(sig == NULL)
                        sigm = dec2local(R, td, api_sig);
                    else
                        sigm = dec2local(R, td, sig);
                }
            }
        }
        else
            stopflag = TRUE;
        free(sigma);          // free space before starting anew
        free(tau);
        if(invflag == TRUE && simpleflag == FALSE)
            free(ta);
    }
    free(r1tmp);
    free(sigm);
    if(f != NULL)
        fclose(f);
    if(g != NULL)
        fclose(g);
    return 0;
}
```

```
/*
    Name:       APIL.H
    Author:     Leo Liberti
    Purpose:    To implement functions for an Arbitrary Precision
                Integer Library (header file and class definition).
    Source:     MS Visual C++ 4.0 (header file)
*/

#ifdef WIN32
#include <malloc.h>
    typedef unsigned long number_length;
#endif
#ifndef WIN32
#include <alloc.h>
    typedef unsigned long number_length;
#endif
#include<string.h>
#include<stdlib.h>
#include<math.h>

typedef short int boolean;

#ifndef BASE
    #define BASE (unsigned short) 10
#endif
#ifndef TRUE
    #define TRUE (unsigned short) 1
#endif
#ifndef FALSE
    #define FALSE (unsigned short) 0
#endif
#ifndef PLUS
    #define PLUS (boolean) 1
#endif
#ifndef MINUS
    #define MINUS (boolean) -1
#endif

class api
{

public:
    /* DATA */

    char *number;
    short int sign;

    /* CONSTRUCTORS AND DESTRUCTORS */

    api(void);
    api(char *n, short int s);
    api(const api &a);
    //~api();

    /* UTILITY FUNCTIONS */

    void exchange(api *a, api *b);
    char *api2char(void);
    unsigned short int get_coefficient_power(unsigned long);

    /* MATHEMATICAL FUNCTIONS */

    api abs();
    api cut(number_length);
    friend void divide(api, api, api *, api *);
    friend void one_digit_divide(api, short int, api *, api *);
    friend api power(api, api);
    friend api power(api, unsigned long);
    friend api power(api, long);
    friend api power(api, unsigned int);
    friend api power(api, int);
    friend api power(api, unsigned short);
```

```
friend api power(api, short);
friend api power(api, char *);
/* MATHEMATICAL OPERATORS */
/* ******** Assignment Operators */
api operator =(api);
api operator =(short int);
api operator =(int);
api operator =(long int);
api operator =(unsigned short int);
api operator =(unsigned int);
api operator =(unsigned long int);
api operator =(char *n);
/* ******** Order Operators */
boolean operator <(api);
boolean operator <(unsigned long);
boolean operator <(long);
boolean operator <(unsigned int);
boolean operator <(int);
boolean operator <(unsigned short);
boolean operator <(short);
boolean operator <(char *);
boolean operator ==(api);
boolean operator ==(unsigned long);
boolean operator ==(long);
boolean operator ==(unsigned int);
boolean operator ==(int);
boolean operator ==(unsigned short);
boolean operator ==(short);
boolean operator ==(char *a);
boolean operator !=(api);
boolean operator !=(unsigned long);
boolean operator !=(long);
boolean operator !=(unsigned int);
boolean operator !=(int);
boolean operator !=(unsigned short int);
boolean operator !=(short int);
boolean operator !=(char *a);
boolean operator >(api);
boolean operator >(unsigned long);
boolean operator >(long);
boolean operator >(unsigned int);
boolean operator >(int);
boolean operator >(unsigned short);
boolean operator >(short);
boolean operator >(char *a);
boolean operator <=(api);
boolean operator <=(unsigned long);
boolean operator <=(long);
boolean operator <=(unsigned int);
boolean operator <=(int);
boolean operator <=(unsigned short);
boolean operator <=(short);
boolean operator <=(char *a);
boolean operator >=(api);
boolean operator >=(unsigned long);
boolean operator >=(long);
boolean operator >=(unsigned int);
boolean operator >=(int);
boolean operator >=(unsigned short);
boolean operator >=(short);
boolean operator >=(char *a);
/* ******** Arithmetical Operators */
api operator +();
api operator -();
api operator +(api);
api operator +(unsigned long);
api operator +(long);
api operator +(unsigned int);
api operator +(int);
api operator +(unsigned short);
api operator +(short);
api operator +(char *);
api operator -(api);
api operator -(unsigned long);
api operator -(long);
api operator -(unsigned int);
api operator -(int);
api operator -(unsigned short);
api operator -(short);
api operator -(char *);
api operator *(api);
api operator *(unsigned long);
api operator *(long);
api operator *(unsigned int);
api operator *(int);
api operator *(unsigned short);
api operator *(short);
api operator *(char *);
api operator /(api);
api operator /(unsigned long);
api operator /(long);
api operator /(unsigned int);
api operator /(int);
```

```
    api operator /(unsigned short);
    api operator /(short);
    api operator /(char *);
    api operator %(api);
    api operator %(unsigned long);
    api operator %(long);
    api operator %(unsigned int);
    api operator %(int);
    api operator %(unsigned short);
    api operator %(short);
    api operator %(char *);
};

/* api friends */
void divide(api, api, api *, api *);
void one_digit_divide(api, short int, api *, api *);
api power(api, api);
api power(api, unsigned long);
api power(api, long);
api power(api, unsigned int);
api power(api, int);
api power(api, unsigned short);
api power(api, short);
api power(api, char *);

/* other functions */
unsigned short int char2int(char *, number_length);
char int2char(unsigned short int);
char *itoa(long);
char *itoa(unsigned long);
unsigned short int mod(short int, short int);
int sgn(long);
void check_char(char *, number_length);
char *garbage_collect(char *, number_length *);
char *garbage_collect(char *);
number_length power(number_length, number_length);
```

```
/*
    Name:       APIL
    Author:     Leo Liberti
    Purpose:    To implement functions for an Arbitrary Precision
                Integer Library.
    Source:     MS Visual C++ 4.0
    History:    17/05/97  0.0     work started. Number format is a string
                                  of character digits.
                24/05/97  0.6     after lots of hassle, implemented rela-
                                  tional operators, +, -, *.
                04/06/97  1.0     first working version: added /, %, power.
*/

#include <stdio.h>
#include <string.h>
#include "apil.h"

/* UTILITIES FUNCTIONS */

/* returns the integer stored in the pos'th character of the string.
   convention is that the first char. of string is indexed with 0 */
unsigned short int char2int(char *t, number_length pos)
{
    int ret;
    char r;

    r = *(t + pos);

    if(r < '0' || r > '9')
        ret = -1;
    else
        ret = r - '0';

    return ret;
}

/* returns a char out of a positive short int */
char int2char(unsigned short int p)
{
    return p + '0';
}

/* converts multidigit integer to string */
char *itoa(unsigned long p)
{
    unsigned long digits, i;
    char *ret;

    if(p == 0)
        digits = 1;
    else
        digits = (unsigned long) floor(log10((double) p)) + 1;

    // ret = new char[digits];
    ret = (char *) malloc(digits);
    check_char(ret, digits);

    for(i = 0; i < digits; i++)
        *(ret + digits - i - 1) = int2char((p / \
                    power(BASE, (number_length) i)) % BASE);

    return ret;
}

char *itoa(long p)
{
    unsigned long digits, i, ll, q;
```

```
    char *ret;
    q = labs(p);
    if(p == 0)
        digits = 1;
    else
        digits = (unsigned long) floor(log10((double) q)) + 1;
    if(p < 0)
        ++digits;
    //ret = new char[digits];
    ret = (char *) malloc(digits);
    check_char(ret, digits);
    if(p < 0)
    {
        *ret = '-';
        ll = 1;
    }
    else
        ll = 0;
    for(i = ll; i < digits; i++)
        *(ret + i) = int2char((q / \
                    power(BASE, (number_length) i)) % BASE);
    return ret;
}
/* returns a^n */
number_length power(number_length a, number_length n)
{
    number_length u, v, t;
    u = n;
    v = 1;
    t = a;
    while(u != 0)
    {
        if(u % 2 == 1)
        {
            v = v * t;
        }
        t = t * t;
        u = u / 2;
    }
    return v;
}
/* if a>0, returns a mod b; otherwise n + (a mod b) */
unsigned short int mod(short int a, short int b)
{
    unsigned short int ret;
    if(a == 0)
        ret = a;
    else if(a > 0)
        ret = a % b;
    else if(abs(a) == b)
        ret = 0;
    else
        ret = b + (a % b);
    return ret;
}
/* returns the sign of an int */
int sgn(long a)
{
    if(a >= 0)
        return (int) PLUS;
    else
        return (int) MINUS;
}
/* frees unused space from strings */
void check_char(char *t, number_length c)
{
    if(strlen(t) > c)
        *(t + c) = '\0';
    if(!t)
    {
        /* error 2 : no memory */
        printf("APIL: Error: Could not allocate memory.\n");
        exit(2);
    }
}
char *garbage_collect(char *t, number_length *c)
{
    char *r, *s;
    s = t;
    while(*s == '0')
        s++;
    *c = strlen(s);
    /* check if t is zero */
    if(*c == 0)
    {
        s--;
        *c = 1;
    }
    //r = new char[*c];
    r = (char *) malloc(*c);
    check_char(r, *c);
    strcpy(r, s);
    return r;
}
char *garbage_collect(char *t)
{
    char *r, *s;
    s = t;
    while(*s == '0')
```

```
        ++s;
    /* check if t is zero */
    if(strlen(s) == 0)
        s--;
    //r = new char[strlen(s)];
    r = (char *) malloc(strlen(s));
    check_char(r, strlen(s));
    strcpy(r, s);
    return r;
}
/* CLASS API FUNCTIONS */
    api::api()
    {
        this->number = (char *) malloc(0);
        check_char(this->number, 0);
        *(this->number) = '\0';
    }
    api::api(char *n, short int s)
    {
        number_length ln;
        ln = strlen(n);
        this->number = (char *) malloc(ln);
        check_char(this->number, ln);
        strcpy(this->number, n);
        this->sign = s;
    }
    api::api(const api &a)
    {
        number_length ln;
        ln = strlen(a.number);
        this->number = (char *) malloc(ln);
        check_char(this->number, ln);
        strcpy(this->number, a.number);
        this->sign = a.sign;
    }
    /* work has to be done here; destructor does not work properly, so
       rem it out. The problem of course is that there is a memory leak
       without it. Compiling with DEBUG options fixes this but makes the
       executable larger */
    //api::~api()
    //{
    //    //if(this->number != NULL)
    //        //delete this->number;
    //        free(this->number);
    //}
    /* UTILITY FUNCTIONS */
    void api::exchange(api *a, api *b)
    {
        api *address;
        address = a;
        a = b;
        b = address;
    }
    char *api::api2char()
    {
        char *s;
        if(this->sign == MINUS)
        {
            s = (char *) malloc(strlen(this->number) + 1);
            check_char(s, strlen(this->number) + 1);
            *s = '-';
            strcpy(s + 1, this->number);
        }
        else
        {
            s = (char *) malloc(strlen(this->number));
            check_char(s, strlen(this->number));
            strcpy(s, this->number);
        }
        return s;
    }
    unsigned short int api::get_coefficient_power(unsigned long p)
    {
        unsigned long sl;
        unsigned short ret;
        sl = strlen(this->number) - 1;
        ret = *(this->number + sl - p) - '0';
        return ret;
    }
    /* MATHEMATICAL FUNCTIONS */
    api api::abs()
    {
        api ret(this->number, PLUS);
        return ret;
    }
    api api::cut(number_length digits)
    {
        api ret;
        number_length ln, i;
        ret.sign = this->sign;
        ln = strlen(this->number);
        ret.number = (char *) realloc(ret.number, digits);
        if(ln < digits)
        {
```

```
        check_char(ret.number, digits);
        for(i = 0; i < digits - ln; i++)
            *(ret.number + i) = '0';
        strcpy(ret.number + digits - ln, this->number);
    }
    else if(ln > digits)
        strcpy(ret.number, this->number + ln - digits);
    else
        strcpy(ret.number, this->number);
    return ret;
}
/* friend */
void one_digit_divide(api a, short int b, api *q, api *r)
{
    api zero, one, bapi;
    number_length c, la;
    unsigned short int start, tmp, n, t;
    char *ini; /* rem out when using new */
    /* make zero and identity */
    //char *ini = new char[1];
    ini = (char *) malloc(1);
    check_char(ini, 1);
    *ini = '0';
    zero = ini;
    *ini = '1';
    one = ini;
    //delete ini;
    /* preliminary checks */
    bapi = b;
    la = strlen(a.number);
    tmp = a.get_coefficient_power(la - 1);
    if(la == 1 && tmp == abs(b))
    {
        *q = one;
        q->sign = a.sign * sgn(b);
        *r = zero;
        return;
    }
    else if(la == 1 && tmp < abs(b))
    {
        *q = zero;
        *r = a;
        return;
    }
    else if(la == 1 && tmp < abs(b))
    {
        *q = tmp / b;
        *r = tmp % b;
        return;
    }
    else if(la > 1 && tmp < abs(b))
    {
        q->number = (char *) realloc(q->number, la - 1);
        check_char(q->number, la - 1);
        for(c = 0; c < la - 1; c++)
            *(q->number + c) = '0';
        start = 2;
        n = tmp * BASE + a.get_coefficient_power(la - 2);
    }
    else
    {
        q->number = (char *) realloc(q->number, la);
        check_char(q->number, la);
        for(c = 0; c < la; c++)
            *(q->number + c) = '0';
        start = 1;
        n = tmp;
    }
    /* perform calculation */
    for(c = start; c <= la; c++)
    {
        *(q->number + c - start) = int2char(n / b);
        t = n % b;
        if(c != la)
            n = t * BASE + a.get_coefficient_power(la - c - 1);
    }
    q->sign = a.sign * sgn(b);
    r->number = (char *) realloc(r->number, 2);
    check_char(r->number, 2);
    *r = a - bapi * (*q);
    return;
}

/* the following function implements the division algorithm as
   described in Knuth, The Art of Computer Programming, part II:
   Seminumerical Algorithms. */
/* friend */
void divide(api a, api b, api *q, api *r)
{
    api tmp, tmp2, tmp3, tmp4, d1, dummy, zero, one;
    unsigned short d, v1, v2, qhat, qtmp;
    number_length m, n, p, i, j, c, start, q_digits;
    boolean digit_flag, carry_flag;
    short int bsign;
    char *ini;   /* rem out when using new */
    /* make zero and identity */
    //char *ini = new char[1];
    ini = (char *) malloc(1);
    check_char(ini, 1);
    *ini = '0';
    zero = ini;
```

```
*ini = '1';
one = ini;
//delete ini;
/* preliminary checks */
if(a.abs() == b.abs())
{
    *q = one;
    q->sign = a.sign * b.sign;
    *r = zero;
    return;
}
else if(a.abs() < b.abs())
{
    *q = zero;
    *r = a;
    return;
}
/* start */
n = strlen(b.number);
m = strlen(a.number) - n;
//char *ini = new char[m + 1];
ini = (char *) realloc(ini, m + 1);
check_char(ini, m + 1);
for(i = 0; i < m + 1; i++)
    *(ini + i) = '0';
q->sign = PLUS;
q->number = (char *) realloc(q->number, m + 1);
check_char(q->number, m + 1);
strcpy(q->number, ini);
//delete ini;
//char *ini = new char[n];
ini = (char *) realloc(ini, n);
check_char(ini, n);
for(i = 0; i < n; i++)
    *(ini + i) = '0';
r->sign = PLUS;
r->number = (char *) realloc(r->number, n);
check_char(r->number, n);
strcpy(r->number, ini);
// delete ini;
/* start calculation [Knuth, Art of C.P., II, p. 257] */
v1 = b.get_coefficient_power(n - 1);
v2 = b.get_coefficient_power(n - 2);
if(v1 == 0)
{
    /* error 1 : divide by 0 */
    printf("APIL: ERROR: Divide by zero.\n");
    exit(1);
}
if(n == 1)
{
    one_digit_divide(a, v1, q, r);
    q->sign = a.sign * b.sign;
    r->sign = PLUS;
    q->number = garbage_collect(q->number);
    r->number = garbage_collect(r->number);
}
else
{
    q_digits = 0;
    /* step1: normalize */
    d = BASE / (v1 + 1);
    d1 = d;
    tmp = a * d1;
    p = strlen(tmp.number);
    if(p == m + n)
        digit_flag = TRUE;
    else
        digit_flag = FALSE;
    /* save b.sign before changing it */
    bsign = b.sign;
    b = b.abs() * d1;
    v1 = b.get_coefficient_power(n - 1);
    v2 = b.get_coefficient_power(n - 2);
    /* if(strlen(b.number) > n)
        b.number++; */
    /* step2: initialize j */
    if(digit_flag == TRUE)
    {
        /* do the first loop (j = 0) keeping in mind that
           tmp.get_coefficient_power(p - j - 1) doesn't exist
           and has to be set to 0 */
        /* step3.0: calculate qhat */
        qtmp = tmp.get_coefficient_power(p - 1);
        qhat = qtmp / v1;
        while(v2 * qhat > (qtmp - qhat * v1) * BASE + \
                          tmp.get_coefficient_power(p - 2))
            qhat--;
        /* step4.0: multiply and subtract */
        //char *ini = new char[n];
        ini = (char *) realloc(ini, n);
        check_char(ini, n);
        strncpy(ini, tmp.number, (size_t) n);
        tmp2 = ini;
        tmp3 = qhat;
        //delete ini;
        tmp2 = tmp2 - tmp3 * b;
        if(tmp2.sign == MINUS)
        {
```

```
                carry_flag = TRUE;
                //char *ini = new char[n + 1];
                ini = (char *) realloc(ini, n + 1);
                check_char(ini, n + 1);
                *ini = '1';
                for(c = 1; c <= n; c++)
                    *(ini + c) = '0';
                tmp4 = ini;
                //delete ini;
                tmp2 = tmp4 + tmp2;
            }
            else
                carry_flag = FALSE;
            tmp2 = tmp2.cut(n);
            strncpy(tmp.number, tmp2.number, n);
            /* step5.0: ``test remainder" [what? where?] */
            *(q->number) = int2char(qhat);
            q_digits++;
            /* step6.0: add back */
            if(carry_flag == TRUE)
            {
                --(*(q->number));
                tmp2 = tmp2 + b;
                strncpy(tmp.number, tmp2.number + 1, n);
            }
            start = 1;
        }
        else
            start = 0;
        if(m >= start)
        {
            for(j = 0; j <= m - start; j++)
            {
                /* step3: calculate qhat */
                qtmp = tmp.get_coefficient_power(p - j - 1) * BASE + \
                       tmp.get_coefficient_power(p - j - 2);
                if(tmp.get_coefficient_power(p - 1 - j) == v1)
                    qhat = BASE - 1;
                else
                    qhat = qtmp / v1;
                while(v2 * qhat > (qtmp - qhat * v1) * BASE + \
                                  tmp.get_coefficient_power(p - j - 3))
                    qhat--;
                /* step4: multiply and subtract */
                //char *ini = new char[n + 1];
                ini = (char *) realloc(ini, n + 1);
                check_char(ini, n + 1);
                strncpy(ini, tmp.number + j, n + 1);
                tmp2 = ini;
                tmp3 = qhat;
                tmp2 = tmp2 - tmp3 * b;
                if(tmp2.sign == MINUS)
                {
                    carry_flag = TRUE;
                    *ini = '1';
                    for(c = 1; c <= n; c++)
                        *(ini + c) = '0';
                    tmp4 = ini;
                    tmp2 = tmp4 + tmp2;
                }
                else
                    carry_flag = FALSE;
                tmp2 = tmp2.cut(n + 1);
                strncpy(tmp.number + j, tmp2.number, n + 1);
                //delete ini;
                /* step5: test remainder */
                *(q->number + j + start) = int2char(qhat);
                q_digits++;
                /* step6: add back */
                if(carry_flag == TRUE)
                {
                    --(*(q->number + j));
                    tmp2 = tmp2 + b;
                    strncpy(tmp.number + j + 1, tmp2.number + 1, n);
                }
                /* step7: loop on j */
            }
        }
        /* step8: unnormalize */
        if(m >= start)
            tmp2 = tmp.number + m - start;
        else
            tmp2 = tmp;
        //char *ini = new char[1];
        ini = (char *) realloc(ini, 1);
        check_char(ini, 1);
        *ini = '0';
        dummy = ini;
        // delete ini;
        one_digit_divide(tmp2, d, r, &dummy);
        /* use bsign (b.sign saved) to calculate q->sign */
        q->sign = a.sign * bsign;
        r->sign = PLUS;
        *(q->number + q_digits) = '\0';
        if(*(q->number) == '0')
            (q->number)++;
        r->number = garbage_collect(r->number);
    }
    return;
}
```

```
/* function power */
/* friend */
api power(api a, api n)
{
    api u, v, t;

    u = n;
    v = 1;
    t = a;
    while(u != 0)
    {
        if(u.get_coefficient_power(0) % 2 == 1)
        {
            v = v * t;
        }
        t = t * t;
        u = u / 2;
    }
    return v;
}

/* friend */
api power(api a, unsigned long b)
{
    api t;
    t = b;
    return power(a, t);
}

/* friend */
api power(api a, long b)
{
    api t;
    t = b;
    return power(a, t);
}

/* friend */
api power(api a, unsigned int b)
{
    return power(a, (unsigned long) b);
}

/* friend */
api power(api a, int b)
{
    return power(a, (long) b);
}

/* friend */
api power(api a, unsigned short b)
{
    return power(a, (unsigned long) b);
}

/* friend */
api power(api a, short b)
{
    return power(a, (long) b);
}

/* friend */
api power(api a, char *b)
{
    api t;
    t = b;
    return power(a, t);
}
/* end function power */

/* MATHEMATICAL OPERATORS */

/* ******** Assignment Operators */

api api::operator =(api a)
{
    this->number = (char *) realloc(this->number, strlen(a.number));
    check_char(this->number, strlen(a.number));
    strcpy(this->number, a.number);
    this->sign = a.sign;

    return *this;
}

api api::operator =(char *n)
{
    number_length c;

    c = 0;

    if(*n == '-')
    {
        this->sign = MINUS;
        c++;
    }
    else
        this->sign = PLUS;

    while(*(n + c) == '0')
        c++;

    /* check if *(n+c) = '0' */
    if(strlen(n + c) == 0)
        c--;

    this->number = (char *) realloc(this->number,\
                                    strlen(n + c));
    check_char(this->number, strlen(n + c));

    strcpy(this->number, n + c);

    return *this;
}

api api::operator =(long a)
{
    char *s;

    s = itoa(a);
    this->number = s;
```

```
    this->sign = sgn(a);
    return *this;
}

api api::operator =(int a)
{
    *this = (long) a;
    return *this;
}

api api::operator =(short a)
{
    *this = (long) a;
    return *this;
}

api api::operator =(unsigned long a)
{
    char *s;
    s = itoa(a);
    *this = s;
    this->sign = PLUS;
    return *this;
}

api api::operator =(unsigned int a)
{
    *this = (unsigned long) a;
    return *this;
}

api api::operator =(unsigned short a)
{
    *this = (unsigned long) a;
    return *this;
}

/* ******** Order Operators */

/* operator < */
boolean api::operator <(api a)
{
    boolean ret;
    number_length lt, la, i;

    lt = strlen(this->number);
    la = strlen(a.number);

    if(this->sign > a.sign)
        ret = FALSE;
    else if(this->sign < a.sign)
        ret = TRUE;
    else if(lt > la)
        ret = FALSE;
    else if(lt < la)
        ret = TRUE;
    else
    {
        ret = FALSE;
        for(i = 0; i < lt; i++)
        {
            if(this->number[i] < a.number[i])
            {
                ret = TRUE;
                break;
            }
            else if(this->number[i] > a.number[i])
            {
                ret = FALSE;
                break;
            }
        }
    }

    return ret;
}

boolean api::operator <(unsigned long a)
{
    api t;
    t = a;
    if(*this < t)
        return TRUE;
    else
        return FALSE;
}

boolean api::operator <(long a)
{
    api t;
    t = a;
    if(*this < t)
        return TRUE;
    else
        return FALSE;
}

boolean api::operator <(unsigned int a)
{
    return (*this < (unsigned long) a);
}

boolean api::operator <(int a)
{
    return (*this < (long) a);
}

boolean api::operator <(unsigned short int a)
{
    return (*this < (unsigned long) a);
}

boolean api::operator <(short int a)
{
    return (*this < (long) a);
}
```

```
boolean api::operator <(char *a)
{
    api t;
    t = a;
    if(*this < t)
        return TRUE;
    else
        return FALSE;
}
/* end operator < */
/* operator == */
boolean api::operator ==(api a)
{
    boolean ret;
    number_length lt, la, i;
    lt = strlen(this->number);
    la = strlen(a.number);
    if(this->sign != a.sign)
        ret = FALSE;
    else if(lt != la)
        ret = FALSE;
    else
    {
        ret = TRUE;
        for(i = 0; i < lt; i++)
        {
            if(this->number[i] != a.number[i])
            {
                ret = FALSE;
                break;
            }
        }
    }
    return ret;
}
boolean api::operator ==(unsigned long a)
{
    api t;
    t = a;
    if(*this == t)
        return TRUE;
    else
        return FALSE;
}
boolean api::operator ==(long a)
{
    api t;
    t = a;
    if(*this == t)
        return TRUE;
    else
        return FALSE;
}
boolean api::operator ==(unsigned int a)
{
    return (*this == (unsigned long) a);
}
boolean api::operator ==(int a)
{
    return (*this == (long) a);
}
boolean api::operator ==(unsigned short int a)
{
    return (*this == (unsigned long) a);
}
boolean api::operator ==(short int a)
{
    return (*this == (long) a);
}
boolean api::operator ==(char *a)
{
    api t;
    t = a;
    if(*this == t)
        return TRUE;
    else
        return FALSE;
}
/* end operator == */
/* operator != */
boolean api::operator !=(api a)
{
    return !(*this == a);
}
boolean api::operator !=(unsigned long a)
{
    return !(*this == a);
}
boolean api::operator !=(long a)
{
    return !(*this == a);
}
boolean api::operator !=(unsigned int a)
{
    return !(*this == a);
}
boolean api::operator !=(int a)
{
    return !(*this == a);
}
boolean api::operator !=(unsigned short int a)
```

```
{
    return !(*this == a);
}
boolean api::operator !=(short int a)
{
    return !(*this == a);
}
boolean api::operator !=(char *a)
{
    return !(*this == a);
}
/* end operator != */
/* operator > */
boolean api::operator >(api a)
{
    boolean ret;
    if(a < *this)
        ret = TRUE;
    else
        ret = FALSE;
    return ret;
}
boolean api::operator >(unsigned long a)
{
    api t;
    t = a;
    if(*this > t)
        return TRUE;
    else
        return FALSE;
}
boolean api::operator >(long a)
{
    api t;
    t = a;
    if(*this > t)
        return TRUE;
    else
        return FALSE;
}
boolean api::operator >(unsigned int a)
{
    return (*this > (unsigned long) a);
}
boolean api::operator >(int a)
{
    return (*this > (long) a);
}
boolean api::operator >(unsigned short int a)
{
    return (*this > (unsigned long) a);
}
boolean api::operator >(short int a)
{
    return (*this > (long) a);
}
boolean api::operator >(char *a)
{
    api t;
    t = a;
    if(*this > t)
        return TRUE;
    else
        return FALSE;
}
/* end operator > */

/* operator <= */
boolean api::operator <=(api a)
{
    boolean ret;
    if(*this < a || *this == a)
        ret = TRUE;
    else
        ret = FALSE;
    return ret;
}
boolean api::operator <=(unsigned long a)
{
    api t;
    t = a;
    if(*this <= t)
        return TRUE;
    else
        return FALSE;
}
boolean api::operator <=(long a)
{
    api t;
    t = a;
    if(*this <= t)
        return TRUE;
    else
        return FALSE;
}
boolean api::operator <=(unsigned int a)
{
    return (*this <= (unsigned long) a);
}
boolean api::operator <=(int a)
{
```

```
    return (*this <= (long) a);
}
boolean api::operator <=(unsigned short int a)
{
    return (*this <= (unsigned long) a);
}
boolean api::operator <=(short int a)
{
    return (*this <= (long) a);
}
boolean api::operator <=(char *a)
{
    api t;
    t = a;
    if(*this <= t)
        return TRUE;
    else
        return FALSE;
}
/* end operator <= */
/* operator >= */
boolean api::operator >=(api a)
{
    boolean ret;
    if(*this > a || *this == a)
        ret = TRUE;
    else
        ret = FALSE;
    return ret;
}
boolean api::operator >=(unsigned long a)
{
    api t;
    t = a;
    if(*this >= t)
        return TRUE;
    else
        return FALSE;
}
boolean api::operator >=(long a)
{
    api t;
    t = a;
    if(*this >= t)
        return TRUE;
    else
        return FALSE;
}
boolean api::operator >=(unsigned int a)
{
    return (*this >= (unsigned long) a);
}
boolean api::operator >=(int a)
{
    return (*this >= (long) a);
}
boolean api::operator >=(unsigned short int a)
{
    return (*this >= (unsigned long) a);
}
boolean api::operator >=(short int a)
{
    return (*this >= (long) a);
}
boolean api::operator >=(char *a)
{
    api t;
    t = a;
    if(*this >= t)
        return TRUE;
    else
        return FALSE;
}
/* end operator >= */
/* ******** Arithmetical Operators */
api api::operator +()
{
    api ret(this->number, this->sign);
    return ret;
}
api api::operator -()
{
    api ret(this->number, MINUS * (this->sign));
    return ret;
}
/* operator + */
api api::operator +(api a)
{
    char *t; /* rem out when using new */
    number_length lt, la, lmx, lmn, i, *c;
    short int carry, tmp, amax;
    api ret;
    c = new number_length;
    /* in order to avoid the operations of the likes of
           1 -
          12 =
          ----
         -29  (algorithm goes wrong)
        calculates -(12 - 1) instead                    */
```

```
    if(this->sign == PLUS && a.sign == MINUS && (a.abs() > *this))
        return -(-a - *this);
    /* in order to avoid the operations of the likes of
          -12 +
            1 =
          ----
          -29  (algorithm goes wrong)
        calculates -(12 - 1) instead                    */
    if(this->sign == MINUS && a.sign == PLUS && (this->abs() > a))
        return -(-*this - a);
    /* in order to avoid the operations of the likes of
          -12 +
           -1 =
          ----
          -27  (algorithm goes wrong)
        calculates -(12 - (-1)) instead                   */
    if(this->sign + a.sign == 2 * MINUS)
        return -(-*this - a);
    /* now start calculation */
    lt = strlen(this->number);
    la = strlen(a.number);
    if(lt <= la)
    {
        lmn = lt;
        lmx = la;
        amax = TRUE;
    }
    else if(lt > la)
    {
        lmx = lt;
        lmn = la;
        amax = FALSE;
    }
    //char *t = new char[lmx + 1];
    t = (char *) malloc(lmx + 1);
    check_char(t, lmx + 1);
    carry = 0;
    for (i = 1; i <= lmn; i++)
    {
        tmp = this->sign * char2int(this->number, lt - i) + \
              a.sign * char2int(a.number, la - i) + carry;
        *(t + lmx - i + 1) = int2char(mod(tmp, BASE));
        if(this->sign + a.sign == 0)
        {
            if(tmp < 0)
                carry = MINUS;
            else
                carry = 0;
        }
        else
        {
            if(tmp >= BASE)
                carry = PLUS;
            else
                carry = 0;
        }
    }
    for (i = lmn + 1; i <= lmx; i++)
    {
        if(amax == FALSE)
            tmp = char2int(this->number, lt - i);
        else
            tmp = char2int(a.number, la - i);
        tmp = tmp + carry;
        *(t + lmx - i + 1) = int2char(mod(tmp, BASE));
        if(this->sign + a.sign == 0)
        {
            if(tmp < 0)
                carry = MINUS;
            else
                carry = 0;
        }
        else
        {
            if(tmp > BASE)
                carry = PLUS;
            else
                carry = 0;
        }
    }
    if(carry != 0)
        *t = int2char(carry);
    else
        *t = '0';
    t = garbage_collect(t, c);
    check_char(t, *c);
    delete c;
    /* the cases computed here always have PLUS sign */
    ret = t;
    //delete t;
    return ret;
}
api api::operator +(unsigned long a)
{
    api t;
    t = a;
    return *this + t;
}
api api::operator +(long a)
{
    api t;
    t = a;
    return *this + t;
}
```

```
api api::operator +(unsigned int a)
{
    return *this + (unsigned long) a;
}
api api::operator +(int a)
{
    return *this + (long) a;
}
api api::operator +(unsigned short a)
{
    return *this + (unsigned long) a;
}
api api::operator +(short a)
{
    return *this + (long) a;
}
api api::operator +(char *a)
{
    api t;
    t = a;
    return *this + t;
}
/* end operator + */
/* operator - */
api api::operator -(api a)
{
    return *this + (-a);
}
api api::operator -(unsigned long a)
{
    api t;
    t = a;
    return *this - t;
}
api api::operator -(long a)
{
    api t;
    t = a;
    return *this - t;
}
api api::operator -(unsigned int a)
{
    return *this - (unsigned long) a;
}
api api::operator -(int a)
{
    return *this - (long) a;
}
api api::operator -(unsigned short a)
{
    return *this - (unsigned long) a;
}
api api::operator -(short a)
{
    return *this - (long) a;
}
api api::operator -(char *a)
{
    api t;
    t = a;
    return *this - t;
}
/* end operator - */
/* operator * */
api api::operator *(api a)
{
    char *ini; /* rem out when using new */
    number_length i, j, lret, lt, la, len_check, *c;
    unsigned short tmp, k;
    c = new number_length;
    lt = strlen(this->number);
    la = strlen(a.number);
    /* check that length of string doesn't exceed capacity */
    len_check = power(2, sizeof(number_length) * 8 - 1);
    if(la >= len_check && lt >= len_check)
    {
        api ret("APIL: Error: Multiplication Overflow", PLUS);
        return ret;
    }
    /* initialize ret */
    lret = lt + la;
    //char *ini = new char[lret];
    ini = (char *) malloc(lret);
    check_char(ini, lret);
    for(i = 0; i < lret; i++)
        *(ini + i) = '0';
    api ret(ini, PLUS);
    /* start calculation (code from NUME0.C by cerruti@dm.unito.it) */
    for(j = 1; j <= la; ++j)
    {
    k = 0;
    for(i = 1; i <= lt; ++i)
    {
        tmp = char2int(this->number, lt - i) * \
                  char2int(a.number, la - j) + \
                  char2int(ini, lret - i - j + 1) + k;
```

```
        *(ini + lret - i - j + 1) = int2char(tmp % BASE);
        k = tmp / BASE;
    }
    *(ini + lret - j - lt) = int2char(k);
    }
    ini = garbage_collect(ini, c);
    check_char(ini, *c);
    strcpy(ret.number, ini);
    ret.sign = this->sign * a.sign;
    //delete ini;
    return ret;
}
api api::operator *(unsigned long a)
{
    api t;
    t = a;
    return *this * t;
}
api api::operator *(long a)
{
    api t;
    t = a;
    return *this * t;
}
api api::operator *(unsigned int a)
{
    return *this * (unsigned long) a;
}
api api::operator *(int a)
{
    return *this * (long) a;
}
api api::operator *(unsigned short a)
{
    return *this * (unsigned long) a;
}
api api::operator *(short a)
{
    return *this * (long) a;
}
api api::operator *(char *a)
{
    api t;
    t = a;
    return *this * t;
}
/* end operator * */
/* operator / */
api api::operator /(api a)
{
    api quotient, rest;
    divide(*this, a, &quotient, &rest);
    return quotient;
}
api api::operator /(unsigned long a)
{
    api t;
    t = a;
    return *this / t;
}
api api::operator /(long a)
{
    api t;
    t = a;
    return *this / t;
}
api api::operator /(unsigned int a)
{
    return *this / (unsigned long) a;
}
api api::operator /(int a)
{
    return *this / (long) a;
}
api api::operator /(unsigned short a)
{
    return *this / (unsigned long) a;
}
api api::operator /(short a)
{
    return *this / (long) a;
}
api api::operator /(char *a)
{
    api t;
    t = a;
    return *this / t;
}
/* end operator / */
/* operator % */
api api::operator %(api a)
{
    api quotient, rest;
    divide(*this, a, &quotient, &rest);
    return rest;
```

```
    }
    api api::operator %(unsigned long a)
    {
        api t;
        t = a;
        return *this % t;
    }
    api api::operator %(long a)
    {
        api t;
        t = a;
        return *this % t;
    }
    api api::operator %(unsigned int a)
    {
        return *this % (unsigned long) a;
    }
    api api::operator %(int a)
    {
        return *this % (long) a;
    }
    api api::operator %(unsigned short a)
    {
        return *this % (unsigned long) a;
    }
    api api::operator %(short a)
    {
        return *this % (long) a;
    }
    api api::operator %(char *a)
    {
        api t;
        t = a;
        return *this % t;
    }
    /* end operator % */
/* CLASS API END */
```

B Appendice: Il Programma CAInvInf

```
/*
    Name:       CAInvInf.H
    Author:     Leo Liberti
    Purpose:    To test for injectivity of global reps
                of CA transformations (see thesis) (header file
                and class definition).
                Algorithm by S. Amoroso and Y.N. Patt, 1972
    Source:     MS Visual C++ 4.0
*/

// global variables (defaults):
int R = 2;
int n = 3;

// structures
struct triple
{
    unsigned int r;
    unsigned long row;
    unsigned long col;
    int status;
};

// classes:
class couple
{
public:
    // data
    unsigned long a;
    unsigned long b;
    class couple *prec;
    class couple **sequent;
    int *sequent_status;
    int status;
    int merge_after;
    int been_there;

    // functions
    void assign(unsigned long a, unsigned long b);

    // constructors and destructors: don't need any
};
```

```
/*
    Name:       CAInvInf
    Author:     Leo Liberti
    Purpose:    To test for injectivity of global reps
                of CA transformations (see thesis).
                Algorithm by S. Amoroso and Y.N. Patt, 1972
    Source:     MS Visual C++ 4.0
    History:

                02/09/97    0.1 first try finished
                02/09/97    1.0 it works

    Notes:      decimal numbers: unsigned long
                local rule length: unsigned long
                base: unsigned int
                local rules are in form (v_1, \ldots, v_n)

                functions are coded in order of dependency

                exit codes & error msgs
                1            not enough memory
                2            localrule is not injective in 1
                3            localrule is not injective in 2.step 3
                4            localrule is not injective in 2.step 4
                6            localrule is not injective in 2.step 6

                N.B. Actually this program is nearly in C; the only
                C++ extension used is class couple, but it could
                easily be redefined as a structure.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "CAInvInf.h"

/* Values couple.status can take */
#define CROSSED_OUT 0
#define NORMAL -1

/* Values couple.merge_after can take */
#define CROSS 0

/* Values couple.sequent_status can take */
#define CROSSED 0

/* Values couple.been_there can take */
#define NO -1

/* Values triple.status can take */
#define FOUND 0
#define NOT_FOUND -1

/* other defines */
#define TRUE 1
#define FALSE 0
#define ARGS 3
#define MSG "CAInvInf v. 1.0 - Leo Liberti 1997\n"
#define OUTFILE "CAInvInf.log"

// various utility functions *************

unsigned long power(unsigned long a, unsigned long n)
{
    unsigned long u, v, t;
```

```
    u = n;
    v = 1;
    t = a;
    while(u != 0)
    {
        if(u % 2 == 1)
        {
            v = v * t;
        }
        t = t * t;
        u = u / 2;
    }
    return v;
}
// like dec2vect but also inverts direction
unsigned int *dec2local(unsigned int R, unsigned long n, unsigned long d)
{
    unsigned int *v;
    unsigned long t, i;

    v = (unsigned int *) malloc(n * sizeof(unsigned int));
    if(v == NULL)
    {
        fprintf(stderr, "CAInvInf: dec2vect: ERROR: Not enough memory.\n");
        exit(1);
    }

    t = 1;
    for(i = 0; i < n; i++)
    {
        *(v + i) = (d / t) % R;
        t = t * R;
    }

    return v;
}

void local_out(unsigned long loc_len, unsigned int *localrule, FILE *fp)
{
    // inverts back the localrule as well
    unsigned long i;

    fputs("(", fp);
    for(i = 0; i < loc_len - 1; i++)
        fprintf(fp, "%u,", *(localrule + loc_len - i - 1));
    fprintf(fp, "%u)", *localrule);
}

// end of various utility functions *************

// class couple related functions *************

void couple::assign(unsigned long a1, unsigned long b1)
{
    unsigned int i, t;

    a = a1;
    b = b1;

    t = power(::R, 2);
    sequent = (class couple **) malloc(t * sizeof(class couple *));
    if(sequent == NULL)
    {
        fprintf(stderr, "CAInvInf: couple constructor: ERROR: Not \
                         enough memory.\n");
        exit(1);
    }
    for(i = 0; i < t; i++)
        sequent[i] = NULL;
    sequent_status = (int *) malloc(t * sizeof(int));
    if(sequent_status == NULL)
    {
        fprintf(stderr, "CAInvInf: couple constructor: ERROR: Not \
                         enough memory.\n");
        exit(1);
    }
    for(i = 0; i < t; i++)
        sequent_status[i] = CROSSED;

    // default to CROSSED_OUT
    status = CROSSED_OUT;
    merge_after = NORMAL;
    prec = NULL;
    been_there = NO;
}

// end of class related functions *************

// crucial program functions ************

/* satellite functions of injectivity_test */

/* find_couple finds the address (and stores it in a triple)
   in the array ***table of the couple (a, b)
   Notes:   struct triple address.status    meaning
                                       0    found
                                       1    not found */
struct triple find_couple(unsigned long a, unsigned long b, \
                          unsigned int *localrule, class couple ***table)
{
    struct triple address;
    unsigned int R, n, r;
    unsigned long row, col, t, t1;

    // first check that a > b; otherwise invert couple
    if(a < b)
    {
        t = a;
        a = b;
        b = t;
    }

    address.status = NOT_FOUND;
    R = ::R;
    n = ::n;
```

```
    t = power(R, n - 1) - 1;
    t1 = (t + 1) * R;
    r = localrule[a];
    if(localrule[b] == r)
    {
        address.r = r;
        for(row = 0; row < t; row++)
        {
            if(table[r][row][0].a == a)
            {
                address.row = row;
                break;
            }
        }
        for(col = 0; col <= address.row; col++)
        {
            if(table[r][address.row][col].b == b)
            {
                address.col = col;
                address.status = FOUND;
                break;
            }
        }
    }
    return address;
}

/* Note: all reference to "relevant paper" refer to
   [Amoroso & Patt, Decision Procedures for Surjectivity and
    Injectivity of Parallel Maps for Tessellation Structures,
    J. of Comp. Syst. Sci., 6, 448-464, 1972]
   Return codes: exitcode   meaning
                 0          localrule is injective
                 2          TEST 1 failed (localrule not balanced)
                 3          TEST 2 Step 3 failed
                 4          TEST 2 Step 4 failed
                 6          TEST 2 Step 6 failed
*/
int injectivity_test(unsigned int *localrule)
{
    struct triple addr;
    class couple ***table, *current, *copy;
    unsigned int R, n, i, j, k, r, sl;
    unsigned long t, t1, row, col, c1, c2;
    unsigned long *balanced, a, b, counter;
    int exitcode, flag1;
    long flag;
    FILE *f;
    f = fopen(OUTFILE, "a+");
    if(f == NULL)
    {
        fprintf(stderr, "CAInvInf: WARNING: Can't open log file, \
                         report will not be written.\n");
    }
    exitcode = 0;
    R = ::R;
    n = ::n;
    t = power(R, n - 1) - 1;   // t is the no. of rows in table[r]
    t1 = (t + 1) * R;          // t1 is the length of localrule
    r = 0;
    sl = power(R, 2);          // length of *sequent
    /* TEST 1: localrule must be "balanced" (see relevant paper) */
    balanced = (unsigned long *) malloc(R * sizeof(unsigned long));
    if(balanced == NULL)
    {
        fprintf(stderr, "CAInvInf: array \"balanced\" allocation: ERROR: \
                         Not enough memory.\n");
        exit(1);
    }
    for(i = 0; i < R; i++)
        balanced[i] = 0;
    for(c1 = 0; c1 < t1; c1++)
        balanced[*(localrule + c1)]++;
    for(i = 0; i < R; i++)
    {
        if(balanced[i] != t + 1)
        {
            exitcode = 2;
            if(f != NULL)
            {
                fprintf(f, "----------\n");
                fprintf(f, MSG);
                fprintf(f, "Local rule:\n");
                local_out(t1, localrule, f);
                fprintf(f, "\n is not injective: its table is not balanced.\n");
                fprintf(f, "\n Found %lu instances of symbol %u while expecting %lu.\n", balanced[i], i, t + 1);
                fprintf(f, "----------\n");
                fclose(f);
            }
            free(balanced);
            return exitcode;
        }
    }
    // dont't free(balanced) because it will be used later in TEST 2
    // for this purpose, void it.
    for(i = 0; i < R; i++)
        balanced[i] = t;
    /* TEST 2: as described in relevant paper */
    /* Steps 1 and 2: partition "table" of localrule and construct
       sequent tables (array table) */
    /* Initialization of array table
```

```
   note: table is a complex array:
         table[r][] contains the table of couples mapping to r
         table[r][i][j] is the (i,j)-th box of table r. The order
         (i,j) proceeds as in 0,0;1,0;1,1;2,0;2,1;2,2; ...
         there are (R^(n-1) - 1)(R^(n-1))/2 such boxes.
*/
// allocation
table = (class couple ***) malloc(R * sizeof(class couple **));
if(table == NULL)
{
    fprintf(stderr, "CAInvInf: array \"table\" allocation: ERROR: Not \
                     enough memory.\n");
    exit(1);
}
for(i = 0; i < R; i++)
{
    table[i] = (class couple **) malloc(t * sizeof(class couple *));
    if(table[i] == NULL)
    {
        fprintf(stderr, "CAInvInf: array \"table\" allocation: ERROR: Not \
                         enough memory.\n");
        exit(1);
    }
}
for(i = 0; i < R; i++)
{
    for(row = 0; row < t; row++)
    {
        table[i][row] = (class couple *) malloc((row + 1) * sizeof(class couple));
        if(table[i][row] == NULL)
        {
            fprintf(stderr, "CAInvInf: array \"table\" allocation: ERROR: Not \
                             enough memory.\n");
            exit(1);
        }
    }
}
// initialization
/* here the algorithm must search through the vector localrule and
   "fish" couples which map to the same number.
   Important: increment row and col properly, so that entries
   result in a table indexed exactly like one described
   in relevant paper, figure 5. There should be
   no overshooting because localrule is balanced
   and there are no commuted couples. */
for(c1 = t1 - 1; c1 > 0; c1--)
{
    r = localrule[c1];
    /* *balanced records (t - how many times)
       r has been "visited" already */
    balanced[r]--;
    // initialize row and col
    row = balanced[r];
    col = 0;
    for(c2 = 0; c2 < c1; c2++)
    {
        if(localrule[c2] == r)
        {
            table[r][row][col].assign(c1, c2);
            col++;
        }
    }
}
free(balanced);
/* Step 3:find sequent sets and put their addresses in
   table[r][row][col]->sequent[counter] with syntax
   table[r][row][col]->sequent[counter] = &(table[sequent...])
   To do this, we implement a cycle that goes through all
   the *table array and finds (and puts in *sequent)
   the address of all sequent sets. */
// counter for table[.]
for(r = 0; r < R; r++)
{
    // counter for table[x][*]
    for(row = 0; row < t; row++)
    {
        // counter for table[x][y][*]
        for(col = 0; col <= row; col++)
        {
            counter = 0;
            // counter for u_2, \ldots, u_n, *
            for(j = 0; j < R; j++)
            {
                a = ((table[r][row][col].a) * R + j) % t1;
                // counter for v_2, \ldots, v_n, *
                for(k = 0; k < R; k++)
                {
                    b = ((table[r][row][col].b) * R + k) % t1;
                    /* now a and b contain the "pre-sequent set" (i.e.
                       it is a sequent set provided localrule[a]==
                       ==localrule[b] and a!=b. */
                    if(a != b)
                    {
                        /* We try to find the address of (a,b) in
                        ***table with find_couple. If it is not found,
                        it means localrule[a]!=localrule[b] */
                        addr = find_couple(a, b, localrule, table);
                        if(addr.status == FOUND)
                        {
                            /* Verify (a, b) != (table...->a, table...->b).
```

```
                                   If it isn't, exit(2). This means that
                                   localrule is not injective */
                                if(addr.r == r && addr.row == row && \
                                   addr.col == col)
                                {
                                    exitcode = 3;
                                    if(f != NULL)
                                    {
                                        fprintf(f, "----------\n");
                                        fprintf(f, MSG);
                                        fprintf(f, "Local rule:\n");
                                        local_out(t1, localrule, f);
                                        fprintf(f, "\n is not injective: a couple has itself as a sequent set.\n");
                                        fprintf(f, "\n Faulting couple: (%lu, %lu).\n", a, b);
                                        fprintf(f, "----------\n");
                                        fclose(f);
                                    }
                                    // deallocate memory
                                    for(r = 0; r < R; r++)
                                        for(row = 0; row < t; row++)
                                            free(table[r][row]);
                                    for(r = 0; r < R; r++)
                                        free(table[r]);
                                    free(table);
                                    return exitcode;
                                }
                                // Put its address in *sequent entry
                                table[r][row][col].sequent[counter] = \
                                    &(table[addr.r][addr.row][addr.col]);
                                // Un-cross-out box and sequent_status
                                table[r][row][col].status = NORMAL;
                                table[r][row][col].sequent_status[counter]\
                                    = NORMAL;
                                // "counter" is the counter for sequent[*]
                                counter++;
                            }
                        }
                        // otherwise we put a cross in this box.
                        else
                            table[r][row][col].merge_after = CROSS;
                    }
                }
            }
        }
    }
}
/* Step 4 : approach is exploring the tree.
   1) Find first non CROSSED_OUT box with a valid sequent entry
      and record it as the starting point.
   2) There are 4 possibilities for the current box:
    a) box is CROSSED_OUT
       In this case go back to prec, NULL the corresponding
       pointer in *sequent and the entry in *sequent_status,
       and go back to (2). If there is no prec goto (3)
    b) box is the same as the starting point or in any case
       it goes in a loop (i.e. it is the same as one in a chain
       of sequents: trap this exception through been_there):
       localrule is not injective and exit.
    c) box has at least one valid sequent. go to it and back to
       (2)
    d) box does not have valid sequents: there are 2
       possibilities
       i)  box has no CROSS: mark it CROSSED_OUT and goto (2)
       ii) box has CROSS: if there is a prec, go back to it and
           mark out corresponding entry in *sequent_status
           then goto (2); else go to (3)
   3) increase r, row, col and consider table[r][row][col]. If
      r = R, row = t, col = row then exit loop, otherwise
      Goto (2).
*/
// 1) find first non crossed out box with valid sequent
current = NULL;
for(r = 0; r < R; r++)
{
    for(row = 0; row < t; row++)
    {
        for(col = 0; col <= row; col++)
        {
            if(table[r][row][col].status != CROSSED_OUT)
            {
                current = &table[r][row][col];
                // check sequents
                flag = FALSE;
                for(i = 0; i < sl; i++)
                {
                    if(current->sequent_status[i] == NORMAL)
                    {
                        flag = TRUE;
                        break;
                    }
                }
                if(flag == TRUE)
                    break;
                else
                {
                    current = NULL;
                    flag = FALSE;
                }
            }
        }
        if(current != NULL)
            break;
    }
    if(current != NULL)
```

```
        break;
}
if(current == NULL)
{
    // impossible: table has no non CROSSED_OUT entry with NORMAL
    // sequents
    exit(3);
}
else
{
    // initialize
    copy = current;
    current = copy->sequent[i];
    current->prec = copy;
    current->been_there++;
}
// 2)
flag = TRUE;
while(flag == TRUE)
{
    // if (a)
    if(current->status == CROSSED_OUT)
    {
        if(current->prec == NULL)
        {
            current->been_there = NO;
            // no prec: increase r, row, col and back to 2)
            if(col < row)
            {
                col++;
            }
            else
            {
                if(row < t - 1)
                {
                    row++;
                    col = 0;
                }
                else
                {
                    if(r < R - 1)
                    {
                        r++;
                        row = 0;
                        col = 0;
                    }
                    else
                    {
                        // exit point;
                        flag = FALSE;
                    }
                }
            }
            current = &table[r][row][col];
            current->prec = NULL;
            current->been_there = NO;
        }
        else
        {
            current->been_there = NO;
            copy = current;
            current = copy->prec;
            current->been_there = NO;
            // find index of sequent to mark out
            for(i = 0; i < sl; i++)
            {
                if(current->sequent[i] == copy)
                {
                    current->sequent_status[i] = CROSSED;
                    current->sequent[i] = NULL;
                    break;
                }
            }
        }
    }
    else if((current == &table[r][row][col]) && \
            ((current->prec != NULL) || current->been_there > NO))
    {
        // case (b same as starting point): not injective
        if(f != NULL)
        {
            fprintf(f, "----------\n");
            fprintf(f, MSG);
            fprintf(f, "Local rule:\n");
            local_out(t1, localrule, f);
            fprintf(f, "\n is not injective: there is a periodic path starting and\n");
            fprintf(f, "ending with couple (%lu, %lu).\n", current->a, current->b);
            fprintf(f, "----------\n");
            fclose(f);
        }
        // deallocate memory
        for(r = 0; r < R; r++)
            for(row = 0; row < t; row++)
                free(table[r][row]);
        for(r = 0; r < R; r++)
            free(table[r]);
        free(table);
        // exit
        exitcode = 4;
        return exitcode;
    }
    else
    {
        // check whether box has valid sequents or not
        flag1 = FALSE;
        for(i = 0; i < sl; i++)
        {
            if(current->sequent_status[i] == NORMAL)
            {
```

```
                flag1 = TRUE;
                break;
            }
        }
        if(flag1 == TRUE)
        {
            // it has some; case (c)
            current->been_there++;
            copy = current;
            current = copy->sequent[i];
            current->prec = copy;
        }
        else
        {
            // it has none; case (d)
            if(current->merge_after != CROSS)
            {
                // case (i)
                current->status = CROSSED_OUT;
                current->been_there = NO;
            }
            else
            {
                // case (ii)
                if(current->prec == NULL)
                {
                    current->been_there = NO;
                    // no prec: increase r, row, col and back to 2)
                    if(col < row)
                    {
                        col++;
                    }
                    else
                    {
                        if(row < t - 1)
                        {
                            row++;
                            col = 0;
                        }
                        else
                        {
                            if(r < R - 1)
                            {
                                r++;
                                row = 0;
                                col = 0;
                            }
                            else
                            {
                                // exit point;
                                flag = FALSE;
                            }
                        }
                    }
                    current = &table[r][row][col];
                    current->prec = NULL;
                    current->been_there = NO;
                }
                else
                {
                    current->been_there = NO;
                    copy = current;
                    current = copy->prec;
                    current->been_there = NO;
                    // find index of sequent to mark out
                    for(i = 0; i < sl; i++)
                    {
                        if(current->sequent[i] == copy)
                        {
                            current->sequent_status[i] = CROSSED;
                            break;
                        }
                    }
                }
            }
        }
    }
}

/* Step 5: Assigning valid merge_afters. Now we know that each
   path must end in a box containing just a CROSS, and hence
   each box will be assigned a merge_after number. The possibility
   of a box not being assigned one comes into play only because
   there may be periodic paths, but we excluded this in Step 4,
   so this step may be skipped altogether.
*/
/* Step 6: verifying that the merging cannot diverge again. We
   must check all non CROSSED_OUT boxes */
for(r = 0; r < R; r++)
{
    for(row = 0; row < t; row++)
    {
        for(col = 0; col <= row; col++)
        {
            if(table[r][row][col].status != CROSSED_OUT)
            {
                a = table[r][row][col].a / R;
                b = table[r][row][col].b / R;
                if(a == b)
                {
                    // localrule not injective
                    if(f != NULL)
                    {
                        fprintf(f, "----------\n");
                        fprintf(f, MSG);
                        fprintf(f, "Local rule:\n");
                        local_out(t1, localrule, f);
                        fprintf(f, "\n is not injective: it diverges after having merged.\n");
                        fprintf(f, "Faulting couple: (%lu, %lu).\n", table[r][row][col].a, table[r][row][col].b);
```

```
                        fprintf(f, "----------\n");
                        fclose(f);
                    }
                    // deallocate memory
                    for(r = 0; r < R; r++)
                        for(row = 0; row < t; row++)
                            free(table[r][row]);
                    for(r = 0; r < R; r++)
                        free(table[r]);
                    free(table);
                    // exit
                    exitcode = 6;
                    return exitcode;
                }
            }
        }
    }
    /* if we are here, it means that localrule is injective */
    // deallocate memory
    for(r = 0; r < R; r++)
        for(row = 0; row < t; row++)
            free(table[r][row]);
    for(r = 0; r < R; r++)
        free(table[r]);
    free(table);
    // exit
    exitcode = 0;
    if(f != NULL)
    {
        fprintf(f, "----------\n");
        fprintf(f, MSG);
        fprintf(f, "Local rule:\n");
        local_out(t1, localrule, f);
        fprintf(f, "\n is injective.\n");
        fprintf(f, "----------\n");
        fclose(f);
    }
    return exitcode;
}
// end of crucial program functions ************
// main loop functions ***********
void help(void)
{
    fprintf(stdout, MSG);
    fprintf(stdout, "\n   This program checks whether a local CA rule is injective.\n");
    fprintf(stdout, "   Details of algorithm in [Amoroso, Patt, \"Decision Procedures ...\",\n");
    fprintf(stdout, "   J. Comp. Syst. Sci., 6, 1972]. Syntax:\n\n");
    fprintf(stdout, "   CAInvInf [-local] R n [sigma]\n\n");
    fprintf(stdout, "   where R is the ring of interest, n is the length of the scope\n");
    fprintf(stdout, "   of the rule, sigma is the decimal representation of the local\n");
    fprintf(stdout, "   rule. With the option -local input of sigma is interactive and\n");
    fprintf(stdout, "   based on base R, and it must not be given on the command line.\n");
}
int main(int argc, char *argv[])
{
    int exitcode, localflag, arg_i, num, nocmd;
    unsigned long sigma, td, i;
    unsigned int *sigm, tmp;
    ////////////////////////////////////
    // here beginneth the new parser //
    ////////////////////////////////////
    // init
    localflag = FALSE;
    num = ARGS;
    nocmd = 0;
    // check that there is a valid command line!
    if(argc <= ARGS)
    {
        help();
        exit(0);
    }
    // parse
    for(arg_i = 1; arg_i < argc; arg_i++)
    {
        if(*(argv[arg_i]) == '-')
        {
            // options
            if(strcmp(argv[arg_i], "-local") == 0)
            {
                /* if an option decreases the number of requested
                   args, increase nocmd by the right amount */
                nocmd++;
                localflag = TRUE;
            }
            else
                fprintf(stderr, "CAInvInf: main: WARNING: Unrecognized cmd line flag %s.\n", argv[arg_i]);
        }
        else
        {
            // arguments
            switch(num)
            {
            case 1:
                sigma = atoi(argv[arg_i]);
                num--;
                break;
            case 2:
                ::n = atoi(argv[arg_i]);
                num--;
                break;
```

```
            case 3:
                ::R = atoi(argv[arg_i]);
                num--;
                break;
            default:
                fprintf(stderr, "CAInvPrm: main: WARNING: Expecting %u cmd line args, found more.", ARGS);
            }
        }
    }
    // not enough arguments on cmd line
    if(num - nocmd > 0)
    {
        help();
        fprintf(stderr, "CAInvInf: main: ERROR: Expecting more cmd line args than found.\n");
        exit(4);
    }
    ////////////////////////////////
    // here endeth the new parser //
    ////////////////////////////////
    td = power(::R, ::n);
    // check localflag
    if(localflag == TRUE)
    {
        fprintf(stderr, "CAInvInf: main: WARNING: -local option set, scanning for input.\n");
        sigm = (unsigned int *) malloc(td * sizeof(unsigned int));
        if(sigm == NULL)
        {
            fprintf(stderr, "CAInvInf: main: ERROR: Not enough memory.\n");
            exit(1);
        }
        // inputting local rep
        for(i = 1; i <= td; i++)
        {
            fprintf(stderr, "CAInvPrm: Input rule sigma, pos %lu out of %lu:", i, td);
            fscanf(stdin, "%u", &tmp);
            *(sigm + td - i) = tmp;
        }
    }
    else
    {
            sigm = dec2local(R, td, sigma);
    }
    exitcode = injectivity_test(sigm);
    switch(exitcode)
    {
        case 0:
            fprintf(stdout, "Input transformation is injective.\n");
            break;
        case 2:
            fprintf(stdout, "Not injective: table is not balanced.\n");
            break;
        case 3:
            fprintf(stdout, "Not injective: a couple has itself as a sequent set.\n");
            break;
        case 4:
            fprintf(stdout, "Not injective: there is a periodic path in the tree.\n");
            break;
        case 6:
            fprintf(stdout, "Not injective: can diverge after merging.\n");
            break;
    }
    fprintf(stdout, "A report has been written in .\\");
    fprintf(stdout, OUTFILE);
    fprintf(stdout, "\n");
    free(sigm);
    return exitcode;
}
// end of main loop ***********
```

Bibliografia

[AAV97] AAVV., *Properties of CA*, 1997, `http://alife.santafe.edu/pub/topics/cas/postscript/properties.ps.gz`.

[AP72] Serafino Amoroso e Yale N. Patt, *Decision procedures for surjectivity and injectivity of parallel maps for tessellation structures*, Journal of Computer and System Sciences **6** (1972), 448–464.

[Ber66] R. Berger, *The undecidability of the domino problem*, Memoires of the American Mathematical Society **66** (1966).

[Ghi93] Giangiacomo Ghiglia, *Automi cellulari*, 1993.

[Hed69] G. A. Hedlund, *Endomorphisms and automorphisms of the shift dynamical system*, Mathematical Systems Theory **3** (1969), no. 4, 320–375.

[Kar91] Jarkko J. Kari, *Reversibility and surjectivity problems of cellular automata*, Journal of Computer and System Sciences **48** (1991), 149–182.

[Kar96] Jarkko J. Kari, *Representation of reversible cellular automata with block permutations*, Mathematical Systems Theory **29** (1996), 47–61.

[Kar97] Jarkko J. Kari, *Comunicazione privata*, 1997.

[Knu83] Donald E. Knuth, *The art of computer programming, part ii: Seminumerical algorithms*, Addison-Wesley, Reading, Massachussetts, 1983.

[KR88] Brian W. Kerninghan e Dennis M. Ritchie, *The c programming language*, second ed., Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[MD96] Cristopher Moore e Arthur A. Drisko, *Algebraic properties of the block transformation on cellular automata*, Santa Fe Institute Working Papers (1996), `http://www.santafe.edu/sfi/publications`.

[Mel93] Luisa Mellano, *Complessità computazionale e proprietà algebriche degli automi cellulari*, 1993.

[Moo97] Cristopher Moore, *Quasi-linear cellular automata*, Santa Fe Institute Working Papers (1997), `http://www.santafe.edu/sfi/publications`.

[Ric72] D. Richardson, *Tessellations with local transformations*, Journal of Computer and System Sciences **6** (1972), 373–388.

[Sat94] Tadakazu Sato, *Group structured linear cellular automata over* $\mathbb{Z}_m$, Journal of Computer and System Sciences **49** (1994), 18–23.

[Sch96] Herbert Schildt, *Guida al linguaggio c++*, Mc Graw-Hill Libri Italia, Milano, 1996.

[WMO84] Stephen Wolfram, Olivier Martin, e Andrew M. Odlyzko, *Algebraic properties of cellular automata*, Communications in Mathematical Physics **93** (1984), 219–258.

[Wol83] Stephen Wolfram, *Statistical mechanics of cellular automata*, Reviews of Modern Physics **55** (1983), 601–644.

[WP85] Stephen Wolfram e Norman H. Packard, *Two-dimensional cellular automata*, Journal of Statistical Physics **38** (1985), 901–946.

Indice analitico

algebra, 16
algoritmo, 36, 87, 106, 109, 112
Amoroso, Serafino, 106, 109, 111
anello, 10, 20, 84, 103
automa cellulare
    definizione di, 7, 10, 26
automi cellulari
    infiniti, 103

base
    conversione in, 37
    sviluppo in, 114

cicli disgiunti, 85
condizioni al contorno, 10, 16, 99
configurazione, 103
    "Giardino dell'Eden", *vedi* GOE
    finita, 104
      insieme delle, 107
    GOE, 106
    parziale, 107

dipolinomio, 15

equivalenza di vettori, 70
estensione, 11, 22, 92

garden of Eden, *vedi* configurazione GOE
generatore, 56, 74
gruppo, 69, 82, 117
    ciclico, 14, 59, 74, 92
    simmetrico, 13, 74

intorno, 10, 40, 90, 105
    convenzionale, 10, 22, 37, 64
isomorfismo, 77
    orbitale, 70

Kari, Jarkko J., 99

legge di evoluzione, 10, 20
localizzazione, 12, 34

matrice, 14, 29, 32
    circolante, 14, 16
monoide, 21, 25, 27, 30, 40
Moore, Edward F., 106, 107
Myhill, John, 106, 107

numero di Wolfram, 22, 112

orbita, 33, 41, 45, 59
    lunghezza, 59
    periodo, 59

Patt, Yale N., 106, 109, 111
permutazione, 13, 18, 33, 47, 74, 82
polinomio, 15
prodotto
    cartesiano, 74

semi-diretto, 75
punto di accumulazione, 108

raggio
d'azione, 10, 29, 37
destro, 10, 37, 87
negativo, 91
sinistro, 10, 37
rappresentazione
decimale, 38
globale, 82
locale, 37, 41, 70, 86, 90
orbitale, 61, 64, 73
rappresentazione locale, 64
regola "180", 99
regola "75", 101
regola "90", 22, 62, 72
regola locale, 11, 22, 92, 109
bilanciata, 109
restrizione, 107
Richardson, D., 106, 108

schema orbitale, 62, 64
seguente, 114
shift, 12, 21, 26, 29, 37, 92, 117
sottogruppo, 81, 82, 92, 93
normale, 82, 117

tabella dei seguenti, 113
tavola
dei caratteri, 82
dei cicli, 56
di moltiplicazione, 56, 69
trasformazione cellulare
lineare, 15
trasformazione cellulare, 12
iniettiva, 107
invertibile, 12, 69, 87, 103
banale, 95
lineare, 16
non lineare, 17
prodotto di, 24, 33, 45, 64

vettore, 18, 20