Creare configurazioni iniziali per esperimenti sulla CAM-8

Silvio Capobianco, Patrizia Mentrasti

L'evoluzione di un automa cellulare è determinata, oltre che dalla funzione di transizione, dalla configurazione iniziale. Per definizioni introduttive sull'automa cellulare si veda [3]. Per sfruttare appieno le potenzialità della CAM-8 occorre dunque saper inizializzare opportunamente lo spazio-CAM.

Tale compito può essere svolto in molti modi diversi. Ad esempio, si possono scegliere configurazioni iniziali casuali, senza un'apparente regolarità, e questo sarà visto nella prima sezione. È anche possiblie scrivere direttamente sullo spazio-CAM mediante parole CAM-Forth: tale operazione sarà descritta nella sezione 2. Esiste poi un'utility detta `campat`, che permette di creare *pattern file* su disco, da caricare nella RAM quando necessario: il suo impiego è descritto nella sezione 3. Infine, è possibile creare brevi programmi in linguaggio C per creare configurazioni particolari, che `campat` non è in grado di realizzare autonomamente: le linee guida per questa operazione sono fornite nella sezione 4.

1 Inizializzazione casuale

L'inizializzazione dello spazio-CAM in modo casuale è permessa dalla parola CAM-Forth:

```
random>cam
```

Tale comando inizializza ciascun piano della sottocella 0 secondo distribuzioni uniformi indipendenti. Se si vuole inizializzare casualmente un solo piano anziché l'intera sottocella, si può usare la parola CAM-Forth:

```
random>field
```

Tale parola deve essere posta nella pila subito dopo il numero rappresentante il piano da inizializzare: per esempio, se un piano ha nome `center` , la sua inizializzazione richiederà la sequenza di istruzioni:

```
center field random>field
```

Oltre alla distribuzione uniforme, è possibile riprodurre distribuzioni bernoulliane mediante le parole-Forth:

```
/random>cam
```

e:

```
/random>field
```

Tali comandi funzionano nel modo seguente. Sia p il parametro della distribuzione: se `num` e `den` sono due valori *interi*, con `num` non negativo e `den` positivo, tali che `num/den` è all'incirca uguale a p, si possono inizializzare tutti i piani dello spazio-CAM secondo una distribuzione bernoulliana di parametro vicino a p mediante il comando:

```
num den /random>cam
```

Se si vuole inizializzare solo il piano `center`, si userà:

```
center field num den /random>field
```

Il generatore casuale della CAM-8 è piuttosto veloce, ma ha alcuni difetti. Anzitutto, come quasi sempre avviene per le funzioni random, il seme del generatore viene inizializzato allo stesso modo ad ogni avvio dell'interprete CAM-Forth, cosicché lanciando uno di questi comandi dopo un riavvio si ottiene sempre la stessa configurazione. Inoltre, una chiamata a `/random>cam` inizializza ciascun piano secondo lo stesso schema, cioè i campi di ogni cella vengono posti tutti a 0 oppure tutti a 1.

2 Lettura e scrittura dirette

L'interprete CAM-Forth mette a disposizione dell'utente opportune parole per la lettura e scrittura diretta dello spazio-CAM. Prima di adoperare tale possibilità, si suggerisce di inizializzare a zero tutte le celle con la parola CAM-Forth:

```
clear-all-subcells
```

che agisce sull'intero spazio-CAM.

Per entrare nella modalità di input-output di linea si usa la parola CAM-Forth:

```
begin-line-io
```

Per uscirne, la parola CAM-Forth da usare è:

```
end-line-io
```

All'interno della modalità di input-output di linea è possibile adoperare i normali comandi Forth. Questo consente, per esempio, di utilizzare un ciclo `do loop` per tracciare una linea nello spazio-CAM.

La parola CAM-Forth da usare per leggere il contenuto di un sito è `read-point`, la cui sintassi è:

```
x y read-point -- val
```

Con tale notazione, il simbolo `--` separa il contenuto in cima alla pila prima e dopo l'esecuzione della parola-Forth. Ciò vuol dire che `read-point` legge il contenuto del sito alle coordinate (`x`,`y`) nello spazio-CAM *come se fosse un numero* e lo pone in cima alla pila. Su tale valore potranno essere compiute le normali operazioni aritmetiche.

La parola CAM-Forth da usare per scrivere il contenuto di un sito è `write-point`, la cui sintassi è:

```
val x y write-point --
```

`write-point` scrive sul sito alle coordinate (`x`,`y`), che viene trattato come se fosse una variabile intera a 16 bit, il valore `val`.

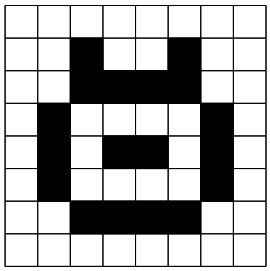
Per modificare il contenuto di un sito, si devono combinare le operazioni di lettura e scrittura.

Come esempio, costruiamo una configurazione per LIFE nel modo seguente:

```
: Cheshire
        clear-all-subcells
        begin-line-io
                1 254 253 write-point
```

```
                1 257 253 write-point
                1 254 254 write-point
                1 255 254 write-point
                1 256 254 write-point
                1 257 254 write-point
                1 253 255 write-point
                1 253 256 write-point
                1 253 257 write-point
                1 258 255 write-point
                1 258 256 write-point
                1 258 257 write-point
                1 254 258 write-point
                1 255 258 write-point
                1 256 258 write-point
                1 257 258 write-point
                1 255 256 write-point
                1 256 256 write-point
        end-line-io
        show
;
```

Questa macro, quando chiamata, ripulisce lo spazio-CAM e disegna al centro di esso la seguente configurazione:



Questa è la testa dello Stregatto[1], e si evolve prima in un largo sorriso, e infine nell'orma di una zampa.

[1] In inglese: *Cheshire cat.*

Come ultima osservazione, la modalità di input-output di linea è piuttosto laboriosa anche dal punto di vista della macchina. Per esempio, raccogliere campioni in questo modo rallenta l'esecuzione dell'esperimento molto più di quanto faccia l'operazione `*count-lut*`.

3 L'utility `campat`

Scritta da Daniel R. Risacher al Massachusetts Institute of Technology, l'utility `campat` rappresenta la scelta migliore per creare rapidamente file di configurazione non eccessivamente complessi. Questa sezione è essenzialmente una riscrittura della *man page* creata da Risacher.

`campat` legge una serie di comandi dallo standard input e ne ricava una configurazione, che scrive sullo standard output. Siccome però `campat` non è pensato per leggere dalla tastiera, e il suo output non è certo pensato per essere visualizzato su un terminale a caratteri, l'utilizzo classico prevede la redirezione dello standard input e dello standard output. L'utility si appoggia al preprocessore del linguaggio C con i flag `-B`, `-P` e `-Dcampat`; vengono accettati tutti i flag che cominciano con `-D`, e lo stesso vale per i commenti in stile C o C++.

```
/*
 * Quindi, si possono inserire queste righe nella configurazione
 * ed esse verranno ignorate dal programma
 */
// Anche questa riga sara' ignorata
```

Inoltre, le direttive `#include`, `#define` e `#ifdef` vengono trattate normalmente, e una macro `campat` viene definita. Il suffisso standard per i file di configurazione di `campat` è `.8p` .

Volendo perciò creare una configurazione di prova, l'utente editerà un file di testo, chiamato ad esempio `prova.8p` , in cui scriverà le istruzioni per la creazione del pattern, dopodiché digiterà al prompt il comando:

```
% campat < prova.8p > prova.pat
```

e premerà Invio. Si può anche sfruttare il meccanismo di *piping* per ottenere un pattern già compresso:

```
% campat < prova.8p | gzip > prova.pat.gz
```

3.1 Comandi

I comandi di `campat` consistono di una singola linea di testo con una parola chiave seguita da una serie di parametri separati da spazi bianchi. Siccome tutti i comandi devono essere su una riga sola, è prevista la possibilità di concatenare le righe mediante il carattere \, quindi:

comando parametro1 \
parametro2

è lo stesso che:

comando parametro1 parametro2

La versione di `campat` attualmente installata riconosce i quattordici comandi illustrati nel seguito.

3.2 Comandi per l'inizializzazione dello spazio

A questo gruppo appartengono i comandi `size` , che definisce le dimensioni dello spazio da inizializzare, e `bitpattern` , con la variante `Bitpattern` , che definisce il modo in cui una regione dello spazio da inizializzare.

3.2.1 `size`

Inizializza a zero una zona della memoria della workstation, che costituirà il supporto per la configurazione. La sua sintassi generica è:

`size` *dimX dimY*

che inizializza uno spazio-CAM di `dimX` per `dimY` siti.

Esiste anche una forma senza argomenti:

`size`

che crea uno spazio bidimensionale di 512x512 siti, che è lo standard per la maggior parte degli esperimenti alla CAM-8.

Non è possibile creare configurazioni in più di due dimensioni.

3.2.2 `bitpattern`

Specifica il pattern di bit che sarà scritto nello spazio-CAM. Ogni volta che il programma scrive un sito, i bit in quel sito vengono modificati concordemente con il pattern specificato da questo comando.

La sua sintassi è:

`bitpattern` *s0 s1 s2 ... s15*

Il simbolo `sN` stabilisce il valore del bit `N`. Sono riconosciuti i seguenti simboli:

`+` Pone il bit a 1
`-` Pone il bit a 0
`x` Lascia il bit invariato
`%n` Pone il bit a 1 con probabilità dell'n%
`=n` Esattamente n siti sul piano verranno posti a 1, e tutti gli altri a 0
`n` Come `%n`
Se sono presenti meno di 16 simboli, quelli mancanti sono considerati uguali a `x`.

3.2.3 `Bitpattern`

Questo comando (notare la B maiuscola) stampa il pattern corrente sullo standard error, che normalmente è lo schermo. Non ha argomenti, e serve solo per il debugging.

3.3 Comandi per la definizione delle regioni

La maggior parte dei comandi di `campat` è collegata alla definizione di regioni dello spazio, che verranno inizializzate secondo l'ultimo `bitpattern` definito.

3.3.1 `point`

Scrive un singolo punto nello spazio-CAM, secondo il pattern corrente. La sua sintassi è:
`point` *x y*

3.3.2 `Point`

Il comando `Point` (si noti la P maiuscola) stampa il contenuto attuale di un punto dello spazio-CAM sullo standard error. La sua sintassi è simile a quella di `point` . Questo comando è utile solo a fini di debugging.

3.3.3 `rectangle`

Disegna un rettangolo nello spazio-CAM. La sua sintassi è:
`rectangle` *x1 y1 x2 y2*
I vertici superiore sinistro e inferiore destro del rettangolo tracciato sono i punti *(x1,y1)* e *(x2,y2)* rispettivamente.

3.3.4 `oval`

Disegna un'ellisse nello spazio-CAM. La sua sintassi è simile a quella di `rectangle`:

`oval` *x1 y1 x2 y2*

L'ellisse tracciata risulterà inscritta nel rettangolo i cui vertici superiore sinistro e inferiore destro sono i punti *(x1,y1)* ed *(x2,y2)* rispettivamente.

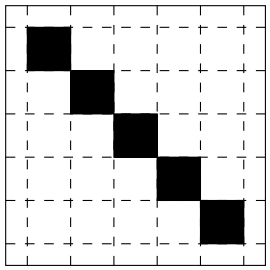
3.3.5 `line`

Traccia una linea nello spazio-CAM. La sua sintassi è:

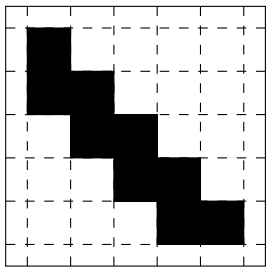
`line` *x1 y1 x2 y2*

dove i punti *(x1,y1)* ed *(x2,y2)* sono gli estremi della linea.

L'algoritmo di tracciamento considera i siti come se avessero 4 vicini anziché 8: ciò significa che una linea diagonale non avrà questo andamento:



ma questo:



Bisogna fare attenzione a questo effetto, che in certi esperimenti potrebbe avere conseguenze spiacevoli.

3.3.6 `edge`

Traccia una linea spezzata *aperta* nello spazio-CAM. La sua sintassi è:
`edge` *x1 y1 x2 y2 ... xn yn*
dove *(x1,y1)*, *(x2,y2)*, ..., *(xn,yn)* sono i punti per i quali, *nell'ordine*, passa la linea. Il tratto di unione tra *(xn,yn)* ed *(x1,y1)* *non* viene tracciato.

3.3.7 `boundary`

Traccia una linea spezzata *chiusa* nello spazio-CAM. La sua sintassi è:
`boundary` *x1 y1 x2 y2 ... xn yn*
dove *(x1,y1)*, *(x2,y2)*, ..., *(xn,yn)* sono i punti per i quali, *nell'ordine*, passa la linea. Il tratto di unione tra *(xn,yn)* ed *(x1,y1)* *viene* tracciato.

3.3.8 `polygon`

Disegna un poligono *pieno* nello spazio-CAM. La sua sintassi è:
`polygon` *x1 y1 x2 y2 ... xn yn*
dove *(x1,y1)*, *(x2,y2)*, ..., *(xn,yn)* sono i vertici del poligono.

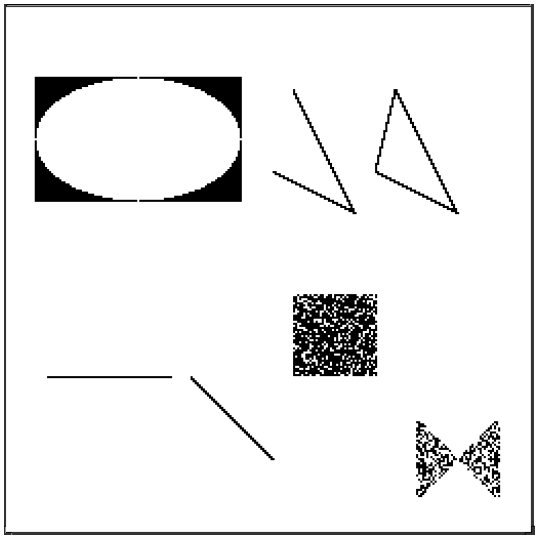
Questo comando è abbastanza "problematico". Innanzitutto, i lati del poligono sono connessi nell'ordine in cui sono specificati, e un sito è considerato interno al poligono se un raggio che parte da quel punto incontra un numero dispari di lati, ragion per cui possono esserci siti interni al poligono che non vengono disegnati. Inoltre, se il pattern corrente per un certo piano ha la forma `=n`, il numero di bit nel piano posti a 1 all'interno del poligono sarà `n` diviso per la dimensione, in siti, *dell'inviluppo convesso* dei vertici del poligono.

I comandi visti finora sono sufficienti per creare una gran quantità di configurazioni iniziali. Vediamo ora un esempio del loro utilizzo:

```
size 256 256

bitpattern +
rectangle 14 34 114 94
bitpattern -
oval 14 34 114 94
```

```
bitpattern +
line 20 180 80 180
line 90 180 130 220
edge 140 40 170 100 130 80
boundary 190 40 220 100 180 80

bitpattern =1250
polygon 140 140 180 140 180 180 140 180

bitpattern %50
polygon 200 200 200 240 240 200 240 240
```

Inserendo queste righe in un file `.8p` e visualizzando la configurazione così ottenuta, è possibile farsi un'idea delle capacità di `campat` .



3.4 Comandi per la modifica dei pattern esistenti

L'ultima categoria di comandi permette di creare nuove configurazioni a partire da configurazioni preesistenti, e usare file in formato `.pbm` (portable bitmap) come maschere.

3.4.1 `pat_file`

Carica nella configurazione un pattern preesistente. La sintassi di questo comando è:

`pat_file` *nomefile*

Questo comando è concepito per ottenere nuove configurazioni da quelle preesistenti, ragion per cui la configurazione nel file DEVE avere la stessa dimensione dello spazio-CAM definito da `size`, o succederanno cose brutte e non documentate. In teoria, se `nomefile` è `-`, il pattern verrà letto da standard input. L'autore di `campat` non sa se questa opzione funziona, e nemmeno se può essere utile.

3.4.2 `pbm_file`

Questo comando prende in input il nome di un file `.pbm` e le coordinate di un punto, secondo la sintassi:

`pbm_file` *nomefile x y*

Qualunque punto che è nero nel bitmap sarà disegnato nello spazio-CAM.

Attenzione: non c'è nessuna garanzia che un `bitpattern` che usa il formato `=n` funzionerà correttamente con il comando `pbm_file`. Se un piano è posto a `=n` nel pattern, la distribuzione dei siti sarà n diviso la dimensione (in siti) *del bitmap*.

3.4.3 `mask_pbm`

Adopera un file `.pbm` come maschera per tutte le routine di disegno, oppure rimuove tutte le maschere in uso. La sua sintassi è:

`mask_pbm` *nomefile*

per caricare una maschera e:

`mask_pbm`

per rimuovere le maschere.

Ogni volta che un sito viene disegnato, il vero sito subirà l'effetto solo se la locazione di quel sito modulo la dimensione del bitmap di mascheratura è nera nel bitmap; in altre parole, lo spazio verrà ricoperto con tasselli uguali al bitmap, e un sito nello spazio verrà modificato secondo il `bitpattern` corrente solo se corrisponde a una zona nera della tassellatura.

Per esempio, un modo facile di disegnare una scacchiera usando `campat` consiste nel definire un bitmap nel modo seguente:

```
P1
# check.pbm
4 4
1 0 0 0
0 0 0 0
0 0 1 0
0 0 0 0
```

Dopo aver salvato il file check.pbm, si includano i comandi:

```
mask_pbm check.pbm  /* carica un bitmap a scacchiera   */
rect 10 10 20 30    /* disegna un rettangolo a scacchi */
mask_pbm            /* rimuove la mascheratura         */
```

3.5 Bug

La versione di `campat` installata è intesa solo come prototipo di prima generazione; per questo motivo, la sua implementazione non è perfetta. L'autore è disponibile per segnalazioni di bug all'indirizzo E-Mail `magnus@mit.edu`.

A volte, `campat` segnala un errore anche se, apparentemente, il file di istruzioni è corretto. Questo non è un problema del programma, ma dell'editor di testo usato per scrivere il file `.8p`. In effetti, un comando `campat` deve terminare con un carattere newline, che diversi editor (tra i quali Emacs) *non* inseriscono automaticamente alla fine del file. Il modo più veloce per risolvere il problema consiste nel dare dalla shell il comando:

```
% echo "" >> nomefile
```

o, in alternativa, nel riaprire il file con il proprio editor di testo, portarsi alla fine, premere Invio e salvare.

Il comando `pat_file` è affetto da un bug che ne rende impossibile l'utilizzo. Se si tenta di usarlo, il programma causa un errore di segmentazione con conseguente terminazione e dumping di un file `core`.

Quando si usano i comandi `polygon`, `mask_pbm` o `pbm_file`, i pattern che usano la sintassi `=n` non funzionano correttamente. Questo accade perché gli `n` siti che vengono scelti sono selezionati dall'output di un linear feedback shift register emulato, e la dimensione dello spazio da disegnare non viene calcolata al momento della generazione dei siti. Il bug potrebbe venire corretto, ma tutti questi comandi verrebbero eseguiti nel doppio del tempo.

4 Creazione di programmi in linguaggio C

Qualora l'utility `campat` non si rivelasse sufficiente, è sempre possibile ricorrere ad un programma in linguaggio C scritto per l'occasione. A volte basterà scrivere un filtro, altre volte occorrerà costruire qualcosa di più complesso.

Per gli usi che ci proponiamo, sarà in genere sufficiente conoscere i rudimenti del linguaggio e le funzioni più importanti della libreria standard. Per chi non conoscesse il linguaggio C, una introduzione veloce ed economica è costituita dal testo di Di Battista e Vargiu [1]; il consiglio è però di adoperare il volume di Kernighan e Ritchie [2], che costituisce il miglior riferimento per questo linguaggio.

4.1 Scrittura di un filtro per `campat`

Pur essendo dotato di varie funzionalità, `campat` è privo di alcune caratteristiche utili. Non è in grado, ad esempio, di copiare uno strato su un altro. Per risolvere problemi di questo tipo, la cosa più semplice da fare è scrivere un *filtro*, ossia un programma che "ritocca" l'output di `campat` prima di scrivere la configurazione su file. Conviene rimanere nella filosofia d'uso di `campat`: il filtro leggerà da standard input e scriverà su standard output.

Come esempio, supponiamo di voler creare una configurazione in cui i primi quattro piani sono inizializzati con distribuzione uniforme, e i secondi quattro bit contengono una loro copia. Dovremo costruire un file `rand03.8p` in cui inizializziamo i primi quattro bit, e scrivere un filtro `dup03` che duplica il contenuto dei primi quattro bit e lo copia nei bit da 4 a 7.

La scrittura di `rand03.8p` è immediata:

```
size 512 512

bitpattern %50 %50 %50 %50
rectangle 0 0 511 511
```

Passiamo al file sorgente `dup03.c`. Per prima cosa, includiamo l'header standard per l'input-output:

```
#include <stdio.h>
```

Poi, osservato che un sito è costituito da due byte, esattamente come uno `short` sulla nostra macchina, dichiariamo:

```
typedef unsigned short Site;
```

L'uso dello specificatore `unsigned` elimina i problemi di segno durante gli shift.

Definiamo ora due funzioni per la lettura e la scrittura dei siti. Andranno bene due macro:

```
#define siteread(p)     fread((p),2,1,stdin)
#define sitewrite(p)    fwrite((p),2,1,stdout)
```

Esaminando i prototipi delle funzioni `fread` ed `fwrite` si comprende il significato di queste macro: `siteread(p)` legge due byte da standard input, trattandoli come un unico oggetto, che copia nella locazione puntata da `p` ; invece, `sitewrite(p)` legge il contenuto della zona di memoria puntata da `p` e lo scrive sullo standard output, trattandolo come se fosse un unico oggetto di dimensione pari a due byte.

Possiamo adesso scrivere la `main`:

```
int main(void)
{
    Site site;

    while (siteread(&site) == 1)
    {
        site |= (site & 0x000f) << 4;
        sitewrite(&site);
    }

    return 0;
}
```

La macro `siteread` restituisce il valore 1 se e solamente se è stato possibile leggere un sito dallo standard input: questo permette di adoperare il filtro indipendentemente dalla dimensione della configurazione. Ogni volta che l'operazione di lettura ha avuto successo, il contenuto dei primi quattro bit della variabile `site` viene isolato mediante un AND aritmetico, spostato di quattro posizioni a destra, e copiato sui secondi quattro bit di `site` mediante un OR aritmetico. Facciamo un esempio nel caso in cui `site` ha il valore `0000000000000110`:

```
site                                0000000000000110
site & 0x000f                       0000000000000110
(site & 0x000f) << 4                0000000001100000
site | ((site & 0x000f) << 4)       0000000001100110
```

Scritto il codice, lo compiliamo con l'istruzione:

```
% gcc dup03.c -o dup03
```

e creiamo la configurazione, *già compressa*, con la sequenza:

```
% campat < rand03.8p | dup03 | gzip > rand03.pat.gz
```

4.2 Creazione diretta di una configurazione

Ci sono casi in cui la strategia dei filtri funziona male. Supponiamo, ad esempio, di avere un file di testo che rappresenta una distribuzione di cariche lungo una linea retta, e di dover ricostruire questa configurazione lungo una linea diagonale: `campat` non ci è di grande utilità, perché, come abbiamo visto, non traccia "bene" le linee diagonali. L'operazione sarebbe possibile, ad esempio, creando una configurazione vuota e filtrandola opportunamente; ma vista la semplicità della cosa, tanto vale scrivere *ex novo* un programma `diagstg` che la costruisce. Per semplicità supporremo che il file contenga caratteri 0 ed 1, eventualmente intervallati da spazi e newline.

Dopo aver incluso l'header `stdio.h` , si deve scrivere una funzione che legga da un file di testo i soli caratteri 0 e 1: questo compito è svolto dalla funzione `getbit` illustrata nel seguito:

```
int getbit(FILE *stream)
{
    int c;

    do {
        c = getc(stream);
    } while (c != '0' && c != '1' && !feof(stream));

    if (feof(stream)) return EOF;
    else return c - '0';
}
```

La funzione `getbit` legge un carattere alla volta dal file indicato da `stream` finché non ne trova uno che rappresenta una cifra binaria, o arriva alla fine del file: in quest'ultimo caso restituisce `EOF` (una macro definita in `stdio.h`), altrimenti restituisce il valore numerico corrispondente al carattere trovato.

Per scrivere i siti, Tommaso Toffoli propone la funzione `putsite`, la cui implementazione è:

```
void putsite(int site)
{
    putchar((site & 0xff00) >> 8);
    putchar(site & 0x00ff);
}
```

In pratica, `putsite` suddivide l'operazione di scrittura del sito in due operazioni di scrittura di caratteri: questo è senz'altro il modo più semplice ed istruttivo per la comprensione della struttura delle configurazioni. Problemi di segno non ce ne sono, perché un `int` occupa sempre più byte di un `char`.

A questo punto, scrivere la `main` è molto facile:

```
int main(int argc, char *argv[])
{
    int bit, site;
    int x, y;
    FILE *stream;

    if (argc != 2)
    {
        fprintf(stderr,
                "Utilizzo: %s nomefile > configurazione\n",
                argv[0]);
        return 0;
    }

    if ((stream=fopen(argv[1],"r")) == NULL)
    {
        fprintf(stderr,
                "Impossibile aprire il file %s\n",
                argv[1]);
```

```
        return 2;
    }

    for (y = 0; y < 512; y++)
        for (x = 0; x < 512; x++)
        {
            site = 0;
            if (x == y)
                if (bit=getbit(stream)) != EOF)
                    site |= bit;
            putsite(site);
        }

    fclose(stream);

    return 0;
}
```

Si osservi che il ciclo sulle ordinate è quello esterno, perché lo spazio-CAM viene sempre inizializzato *per righe*.

Compilato il programma con il comando:

```
% gcc diagstg.c -o diagstg
```

e supposto che il file con la distribuzione sia `distr.txt`, potremo creare la configurazione, al solito già compressa, con la sequenza:

```
% diagstg distrib.txt | gzip > diagstg.pat.gz
```

Riferimenti bibliografici

[1] G. Di Battista, F. Vargiu, *Dal linguaggio Pascal al linguaggio C*, Masson, Milano 1994

[2] B. W. Kernighan, D. M. Ritchie, *Linguaggio C - Seconda edizione*, Jackson Libri, 1989

[3] P. Mentrasti, O. Tempestini, *Istruzioni fondamentali per l'utilizzo della CAM-8*, Dipartimento di Matematica, Università degli Studi di Roma "La Sapienza", 99/34, 1999

[4] P. Mentrasti, O. Tempestini, *L'automa cellulare CAM-8: caratteristiche fisiche e dinamiche*, Dipartimento di Matematica, Università degli Studi di Roma "La Sapienza", 37/99, 1999

[5] D. R. Risacher, `campat` *man page*

[6] T. Toffoli, N. Margolus, *STEP Software Reference*, The MIT Press, Cambridge, MA 1995

[7] T. Toffoli, N. Margolus, *STEP Technical Reference*, The MIT Press, Cambridge, MA 1993